INSIDE MACINTOSH

Managing Color With ColorSync

Covers ColorSync 2.5.1

November 20, 1998

Technical Publications

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Revision History

Revision history for *Managing Color With ColorSync*.

November 1998: First release. This document combines ColorSync documentation from *Advanced Color Imaging on the Mac OS* with new material covering the ColorSync Manager through version 2.5.1. It includes updated conceptual material, code samples, and a complete API reference.

About This Document

This document describes ColorSync, the color management system from Apple Computer, Inc. that provides essential services for fast, consistent, and accurate color management. It also describes the ColorSync Manager, the application programming interface (API) to these services.

This Preface covers:

For additional information about this document, see "What's New" (page 539).

What's in This Document

This document introduces ColorSync and the concepts of color management, shows how to use ColorSync in applications and device drivers, and provides an overview of developing color management modules (CMMs). It describes features available through ColorSync version 2.5. Most existing code written to use version 2.0 or 2.1 of the ColorSync Manager should continue to work with version 2.5 without modification.

Note
There are no changes to the ColorSync Manager API between version 2.5 and version 2.5.1, so this document is up-to-date for ColorSync 2.5.1. ◆

This document includes the following sections, as well as a glossary and index.

- "Introduction to Color and Color Management Systems" (page 25) provides a general introduction to color-management, defines terms such as profile, color space, and CMM, and serves as a primer for those unfamiliar with color management systems.

- “Introduction to ColorSync” (page 45) provides an overview of ColorSync and the ColorSync Manager, including both user interface and API elements. It describes ColorSync’s support for scripting, monitor calibration, the use of multiple processors, and other features.
- “Developing ColorSync-Supportive Applications” (page 81) describes how your application can use the ColorSync Manager to provide many color management services. It includes detailed code samples.
- “Developing ColorSync-Supportive Device Drivers” (page 195) describes how you can use the ColorSync Manager to create ColorSync-supportive drivers for peripherals such as input, output, and display devices.
- “ColorSync Reference for Applications and Drivers” (page 217) describes the functions, constants, and data types defined by the ColorSync Manager for use by your application or device driver.
- “Developing Color Management Modules” (page 429) describes how to create a color management module (CMM) component that ColorSync can use to match and check colors.
- “ColorSync Reference for Color Management Modules” (page 467) describes the request code constants passed to your color management module from the Component Manager to request services your CMM provides. It also describes the functions your CMM may define to respond to ColorSync Manager request codes
- “Version and Compatibility Information” (page 525) describes the `Gestalt` information, shared library version numbers, CMM version numbers, and ColorSync header files you use with different versions of the ColorSync Manager. It also describes CPU and system requirements. In addition, it describes backward compatibility between versions of the ColorSync Manager and the profile formats they use.
- “What’s New” (page 539) lists the new features available with ColorSync 2.5 and provides links to new and revised material. It includes a summary of new and changed code listings, functions, data types, and constants. It also includes a list of features new to ColorSync version 2.1, as well as information on where to obtain documentation for other color-related technologies.

Conventions

This document uses the following conventions to help you locate information.

Quick Reference Banners

The following banners appear below section headings in reference material. They indicate features that are new in ColorSync version 2.5 or that operate differently than in previous versions.

NEW IN COLORSYNC 2.5
Functions, types, and constants introduced in ColorSync version 2.5.

CHANGED IN COLORSYNC 2.5
Functions, types, and constants that have been modified in ColorSync version 2.5, or are used differently than in previous versions.

NOT RECOMMENDED IN COLORSYNC 2.5
Functions that have been superseded by new functions and are not recommended for use with ColorSync version 2.5.

Version Notes

Functions and data types that are changed or not recommended in ColorSync version 2.5 generally contain a **VERSION NOTES** section that summarizes the changes or points to related information.

Special Fonts

All code listings, reserved words, and the names of actual data structures, constants, fields, parameters, and routines are shown in a monospaced font such as Letter Gothic (`this is Letter Gothic`).

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

Types of Notes

There are several types of notes used in this document.

Note
A note like this contains information that is interesting but possibly parenthetical to the main text. ◆

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

▲ **WARNING**
Warnings like this indicate potential problems that you should be aware of as you design your application or device driver. Failure to heed these warnings could result in system crashes or loss of data. ▲

Important Note on Code Listings

All code listings in this document are shown in C, except for listings that describe resources, which are shown in Rez-input format. Many listings are from the CSDemo application, which is available with the ColorSync 2.5 SDK. See Figures, Tables, and Listings (page 13) for the locations of all code listings in this document.

IMPORTANT
Although the listings in this document have been compiled and, to some degree, tested, Apple Computer does not promote the direct incorporation of these code samples into your application. For example, to make the code listings in this document more readable, only limited error handling is shown. You need to develop your own techniques for detecting and handling errors. ▲

CHAPTER 1

Introduction to Color and Color Management Systems

Contents

This section provides a very brief description of ColorSync, the cross-platform color management system from Apple Computer, Inc. It then provides a general introduction to the basics of color and color management systems.

Read this section to learn about color perception, additive and subtractive color systems, how different peripheral devices represent color, and how color management systems maintain consistent color among devices. If you are already familiar with these concepts, you can skip ahead to "Introduction to ColorSync" (page 45), which provides a detailed overview of ColorSync.

For more information on color theory and color spaces, see:

- Fred W. Billmeyer, Jr., and Max Saltzman. *Principles of Color Technology*, second edition. Wiley, 1981.
- James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice,* second edition. Addison-Wesley, 1990.
- Roy Hall. *Illumination and Color in Computer Generated Imagery.* New York: Springer-Verlag, 1988.
- R.W.G. Hunt. *Measuring Colour,* second edition. Prentice-Hall, 1991.
- Günther Wyszecki and W.S. Stiles. *Color Science: Concepts and Methods, Quantitative Data and Formulae*, second edition. A Wiley-Interscience Publication, 1982.

ColorSync

ColorSync is the platform-independent color management system from Apple Computer, Inc. ColorSync provides essential services for fast, consistent, and accurate desktop color calibration, proofing, and reproduction for the graphic arts, publishing, and printing industries. The **ColorSync Manager** is the application programming interface (API) to these services. ColorSync and the ColorSync Manager are described in detail in "Introduction to ColorSync" (page 45). Color management systems are defined in "Color Management Systems" (page 41).

Color: A Brief Overview

Color is a sensation and, therefore, a subjective experience. The sensation of color is one component of the visual sensation, caused by the sensitivity of the human eye to light. Light can be perceived either directly from light sources (such as the sun, a fire, incandescent or fluorescent bulbs, television screens, and computer displays) or indirectly, when light from these sources is transmitted through or reflected by objects. Color sensation is also affected by how the brain processes information and is specific to each individual. Thus color perception is avery complex phenomenon.

The foundation of the color reproduction process is **trichromatic color vision**, which describes the capacity of the human eye to respond equally to two or more sets of stimuli having different visible spectra. This means that two or more visible spectra may exist that will be perceived as the same color, a phenomenon known as **metamerism**. Because of this property, spectral color reproduction, a very expensive and impractical process, can be replaced by trichromatic color reproduction, a process that is much cheaper and easier to control.

Trichromatic color reproduction induces the illusion of a color using various amounts of only three primary colors: either red, green, and blue mixed additively or cyan, magenta, and yellow mixed subtractively. Additive and subtractive colors are described in “Additive and Subtractive Color” (page 28). Trichromatic color reproduction is the fundamental mechanism used in the majority of color reproduction devices, from television, computer display and movie screens, to magazines, newspapers, large posters, and small pages printed on your desktop printer.

Computers enable us to control color digitally and many peripherals have been developed for acquiring, displaying, and reproducing color. As a result, there is a need for a mechanism to maintain color control in an environment that can include different computer operating systems and hardware, as well as a wide variety of devices and media connected to the computer.

In the Mac OS, the ColorSync Manager is the part of the operating system that provides color management. For a detailed description, see “ColorSync Manager Overview” (page 47).

Color Perception

The eye contains two types of receptors, cones and rods. The rods measure illumination and are not sensitive to color. The cones contain a chemical known as Rhodopsin, which is variously sensitive to reds and blues and has a default sensitivity to yellow. The color the eyes see in an object depends on how much red, green, and blue light is reflected to a small region in the back of the eye called the fovea, which contains a great majority of the cones present in the eye. Black is perceived when no light is reflected to the eye.

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

Color images frequently contain hundreds of distinctly different colors. To reproduce such images on a color peripheral device is impractical. However, a very broad range of colors can be visually matched by a mixture of three "primary" lights. This allows colors to be reproduced on a display by a mixture of red, green, and blue lights (the primary colors of the additive color space shown in Figure 1-4) or on a printer by a mixture of cyan, magenta, and yellow inks or pigments (the primary colors of the subtractive color space shown in Figure 1-4). Black is printed to increase contrast and make up for the deficiency of the inks (making black the key, or K, in CMYK).

Hue, Saturation, and Value (or Brightness)

Color is described as having three dimensions. These dimensions are hue, saturation, and value. **Hue** is the name of the color, which places the color in its correct position in the spectrum. For example, if a color is described as blue, it is distinguished from yellow, red, green, or other colors. **Saturation** refers to the degree of intensity in a color, or a color's strength. A neutral gray is considered to have zero saturation. A saturated red would have a color similar to apple red. Pink is an example of an unsaturated red. **Value** (or brightness) describes differences in the intensity of light reflected from or transmitted by a color image. The hue of an object may be blue, but the terms *dark* and *light* distinguish the value, or brightness, of one object from another. The 3-dimensional color spaces based on hue, saturation and value are described in "HSV and HLS Color Spaces" (page 31).

Additive and Subtractive Color

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

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

Monitors use the additive color space, output printing devices use the subtractive color space.

Color Spaces

A **color space** describes an environment in which colors are represented, ordered, compared, or computed. A color space defines a one-, two-, three-, or four-dimensional environment whose **components** (or color components) represent intensity values. A color component is also referred to as a **color channel**. For example, RGB space is a three-dimensional color space whose stimuli are the red, green, and blue intensities that make up a given color; and red, green, and blue are color channels. Visually, these spaces are often represented by various solid shapes, such as cubes, cones, or polyhedra.

For additional information on color components, see "Color-Component Values, Color Values, and Colors" (page 39).

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

- gray spaces, used for grayscale display and printing; see "Gray Spaces" (page 29)

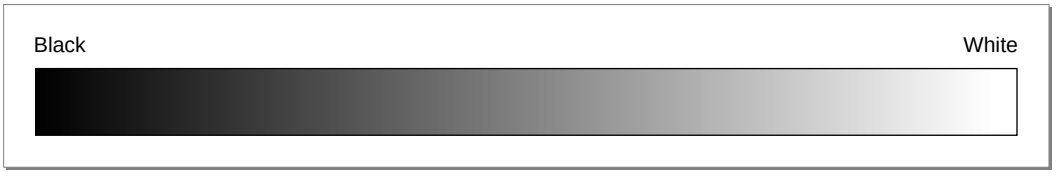
- RGB-based color spaces, used mainly for displays and scanners; see "RGB-Based Color Spaces" (page 30)
- CMYK-based color spaces, used mainly for color printing; see "CMY-Based Color Spaces" (page 33)
- device-independent color spaces, such as L*a*b, used mainly for color comparisons, color differences, and color conversion; see "Device-Independent Color Spaces" (page 34)
- named color spaces, used mainly for printing and graphic design; see "Named Color Spaces" (page 39)
- heterogeneous HiFi color spaces, also referred to as multichannel color spaces, primarily used in new printing processes involving the use of red-orange, green and blue, and also for spot coloring, such as gold and silver metallics; see "Color-Component Values, Color Values, and Colors" (page 39)

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

Gray Spaces

Gray spaces typically have a single component, ranging from black to white, as shown in Figure 1-1. Gray spaces are used for black-and-white and grayscale display and printing. A properly plotted gray space should have a fifty percent value as its midpoint.

Figure 1-1 Gray space



RGB-Based Color Spaces

The **RGB space** is a three-dimensional color space whose components are the red, green, and blue intensities that make up a given color. For example, scanners read the amounts of red, green, and blue light that are reflected from or transmitted through an image and then convert those amounts into digital values. Information displayed on a color monitor begins with digital values that are converted to analog signals for display on the monitor. The analog signals are transmitted to the phosphors on the face of the monitor, causing them to glow at various intensities of red, green, and blue (the combination of which makes up the required hue, saturation, and brightness of the desired colors).

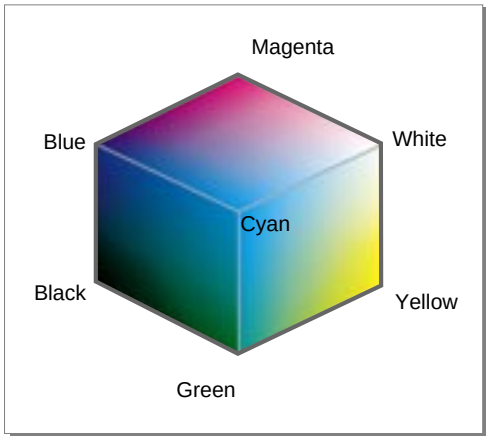
RGB-based color spaces are the most commonly used color spaces in computer graphics, primarily because they are directly supported by most color displays and scanners. RGB color spaces are device dependent and additive. The groups of color spaces within the RGB base family include

- RGB spaces
- HSV and HLS spaces

RGB Spaces

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

Figure 1-2 RGB color space (Red corner is hidden from view)



sRGB Color Space

The **sRGB** color space is based on the ITU-R BT.709 standard. It specifies a gamma of 2.2 and a white point of 6500 degrees K. You can read more about sRGB space at the International Color Consortium site at <http://www.color.org/>. This space gives a complimentary solution to the current strategies of color management systems, by offering an alternate, device-independent color definition that is easier to handle for device manufacturers and the consumer market. sRGB color space can be used if no other RGB profile is specified or available. Starting with version 2.5, ColorSync provides full support for sRGB, including an sRGB profile.

Note that as an open architecture, ColorSync is not tied to the use of the sRGB color space and can support any RGB space that the user might prefer. For example, high end users with good quality reproduction devices may find that the sRGB space, which limits colors to the sRGB gamut, is too restrictive for their required color quality.

HSV and HLS Color Spaces

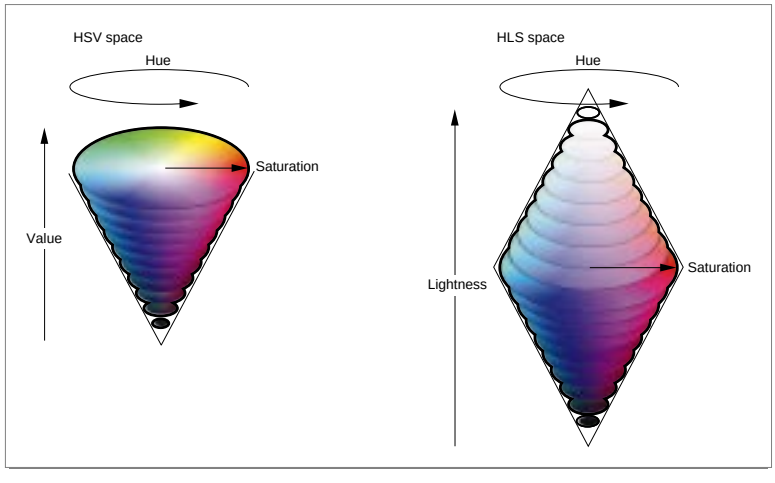
HSV space and **HLS space** are transformations of RGB space that can describe colors in terms more natural to an artist. The name *HSV* stands for *hue*, *saturation*, and *value*. (**HSB space**, or *hue*, *saturation*, and *brightness*, is synonymous with HSV space.) HLS stands for *hue*, *lightness*, and *saturation*. The

two spaces can be thought of as being single and double cones, as shown in Figure 1-3.

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

- The hue component in both color spaces is an angular measurement, analogous to position around a color wheel. A hue value of 0 indicates the color red; the color green is at a value corresponding to 120°, and the color blue is at a value corresponding to 240°. Horizontal planes through the cones in Figure 1-3 are hexagons; the primaries and secondaries (red, yellow, green, cyan, blue, and magenta) occur at the vertices of the hexagons.
- The saturation component in both color spaces describes color intensity. A saturation value of 0 (in the middle of a hexagon) means that the color is "colorless" (gray); a saturation value at the maximum (at the outer edge of a hexagon) means that the color is at maximum "colorfulness" for that hue angle and brightness.

Figure 1-3 HSV (or HSB) color space and HLS color space



- The value component in HSV describes the brightness. In both color spaces, a value of 0 represents the absence of light, or black. In HSV space, a maximum value means that the color is at its brightest. In HLS space, a maximum value

for lightness means that the color is white, regardless of the current values of the hue and saturation components.

CMY-Based Color Spaces

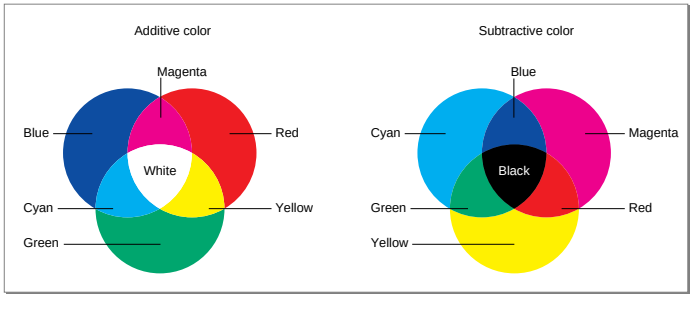
CMY-based color spaces are most commonly used in color printing systems. They are device dependent and subtractive in nature. The groups of color spaces within the CMY family include

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

The name *CMYK* refers to cyan, magenta, yellow, and key (represented by black). Cyan, magenta, and yellow are the three primary colors in this color space, and red, green, and blue are the three secondaries. Theoretically black is not needed. However, when full-saturation cyan, magenta, and yellow inks are mixed equally on paper, the result is usually a dark brown, rather than black. Therefore, black ink is overprinted in darker areas to expand the dynamic range and give a better appearance. Printing with black ink makes it possible to use less cyan, magenta, and yellow ink. This may prevent saturation, especially on materials such as plain paper which cannot accept too much ink. Using black can also reduce the cost per page because cyan, magenta, and yellow inks are generally more expensive than black ink. It can also provide a sharper image, because a single dot of black ink is used in place of three dots of other inks.

Figure 1-4 shows how additive and subtractive colors mix to form other colors.

Figure 1-4 Additive and subtractive colors



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

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

(where red, green, and blue intensities are expressed as fractional values varying from 0 to 1). In reality, the process of deriving the cyan, magenta, yellow, and black values from a color expressed in RGB space is complex, involving device-specific, ink-specific, and even paper-specific calculations of the amount of black to add in dark areas (black generation) and the amount of other ink to remove (undercolor removal) where black is to be printed. Therefore, when ColorSync converts between CMYK and RGB color spaces, it uses an elaborate system of multi-dimensional lookup tables, which ColorSync knows how to interpret. This information is stored in profiles, which are defined in the section “Color Conversion and Color Matching” (page 40).

Device-Independent Color Spaces

Some color spaces can express color in a device-independent way. Whereas RGB colors vary with display and scanner characteristics, and CMYK colors vary with printer, ink, and paper characteristics, *device-independent colors* are not dependent on any particular device and are meant to be true representations of colors as perceived by the human eye. These color representations, called **device-independent color spaces,** result from work carried out by the Commission Internationale d’Eclairage (CIE) and for that reason are also called **CIE-based color spaces**.

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

Conversion from an RGB color space to a CMYK color space involves a number of variables. The type of printer or printing press, the paper stock, and the inks used all influence the balance between cyan, magenta, yellow, and black. In addition, different devices have different **gamuts,** or ranges of colors that they can produce. Because the colors produced by RGB and CMYK specifications are specific to a device, they’re called device-dependent color spaces. Device color

spaces enable the specification of color values that are directly related to their representation on a particular device.

Device-independent color spaces can be used as **interchange color spaces** to convert color data from the native color space of one device to the native color space of another device.

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

XYZ Space

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

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

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

Yxy Space

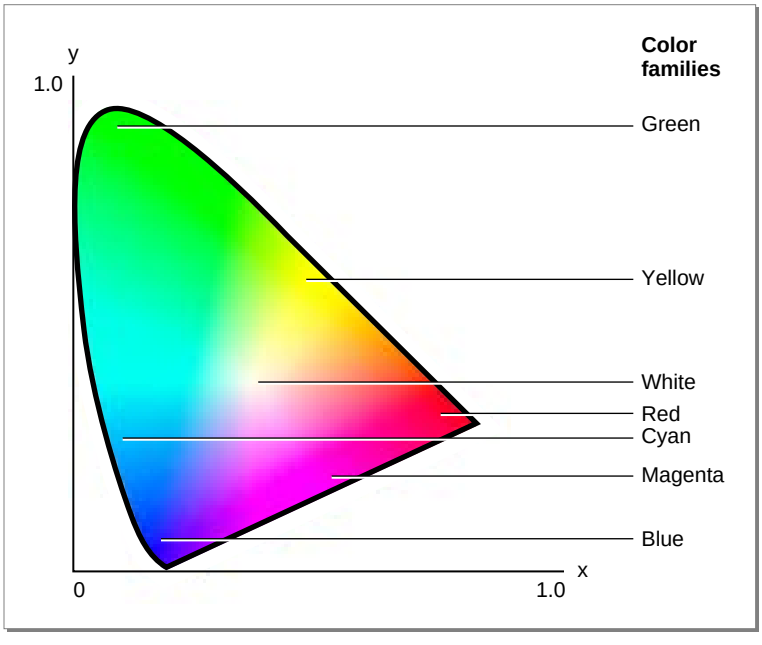
Yxy space expresses the XYZ values in terms of x and y chromaticity coordinates, somewhat analogous to the hue and saturation coordinates of HSV

space. The coordinates are shown in the following formulas, used to convert XYZ into Yxy:

```
Y = Y
x = X / (X+Y+Z)
y = Y / (X+Y+Z)
```

Note that the Z tristimulus value is incorporated into the new coordinates and does not appear by itself. Since Y still correlates to the lightness of a color, the other aspects of the color are found in the chromaticity coordinates x and y. This allows color variation in Yxy space to be plotted on a two-dimensional diagram. Figure 1-5 shows the layout of colors in the x and y plane of Yxy space.

Figure 1-5 Yxy chromaticities in the CIE color space

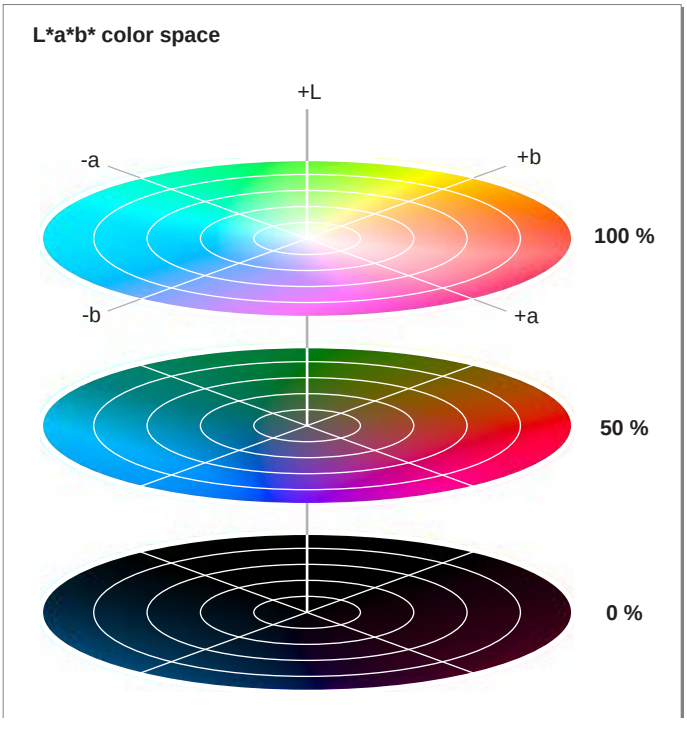


L*u*v* Space and L*a*b* Space

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

L*u*v* space and **L*a*b* space** are nonlinear transformations of the XYZ tristimulus space. These spaces are designed to have a more uniform correspondence between geometric distances and perceptual distances between colors that are seen under the same reference illuminant. A rendering of L*a*b space is shown in Figure 1-6.

Figure 1-6 L*a*b* color space



Both L*u*v* space and L*a*b* space represent colors relative to a **reference white point,** which is a specific definition of what is considered white light, represented in terms of XYZ space, and usually based on the whitest light that can be generated by a given device.

IMPORTANT

Because L*u*v* space and L*a*b* space represent colors relative to a specific definition of white light, they are not completely device independent; two numerically equal colors are truly identical only if they were measured relative to the same white point. ▲

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

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

Indexed Color Spaces

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

Indexed space is the color space you use when drawing with indirectly specified colors. An indexed color value (a color specification in indexed color space) consists of an index value that refers to a color in a color list. Color values are defined in “Color-Component Values, Color Values, and Colors” (page 39).

Named Color Spaces

In a **named color space**, each color has a name; colors are generally ordered so that each has an equal perceived distance from its neighbors in the color space. A named color space provides a relatively small number of discrete colors.

Color systems using named color spaces have existed for many years. Graphic artists and designers using named color systems can "see" the real color by looking at a color chip or swatch. Printing shops can reproduce a specified color accurately.

Named color systems are useful for spot colors, but they have several drawbacks:

- They are not useful for images, which require a continuous range of colors.
- They are highly device dependent and proprietary.
- Colors are tied to medium-specific formulations.
- Applications that use these systems require a device-specific database for each supported printer, making it difficult to add additional devices.

Color-Component Values, Color Values, and Colors

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

Each dimension, or component, in a color space has a **color-component value.** An unsigned 16-bit color-component value can vary from 0 to 65,535 (0xFFFF), although the numerical interpretation of that range is different for different color spaces. In most cases, color-component intensities are interpreted numerically as varying between 0 and 1.0. An exception occurs for the a* and b* channels of the Lab color space, where values ranging from 0 to 65,535 are interpreted numerically as varying from -128.0 to approximately 128.0.

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

Color Conversion and Color Matching

Color conversion is the process of converting colors from one color space to another. **Color matching**, which entails color conversion, is the process of selecting colors from the destination gamut that most closely approximate the colors from the source image. Color matching always involves color conversion, whereas color conversion may not entail color matching. **Rendering intent** refers to the approach taken when a CMM maps or translates the colors of an image to the color gamut of a destination device—that is, a rendering intent specifies a gamut-matching strategy.

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

When an image is output to a particular device, the device displays only those colors that are within its gamut. Likewise, when an image is created by scanning, all colors from the original image are reduced to the colors within the scanner's gamut. Devices with different gamuts cannot reproduce each other's colors exactly, but careful shifting of the colors used on one device can improve the visual match when the image is displayed on another.

Figure 1-7 Color gamuts for two devices expressed in Yxy space

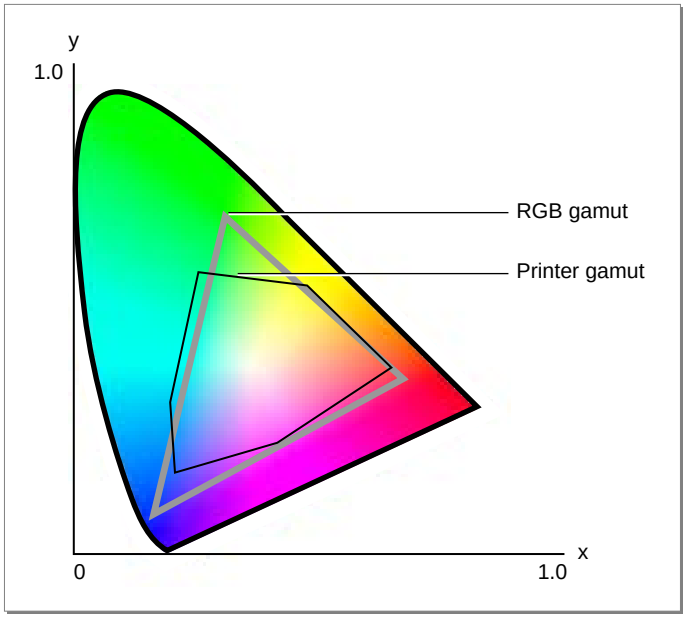


Figure 1-7 shows examples of two devices' color gamuts, projected onto Yxy space. Both devices produce less than the total possible range of colors, and the printer gamut is restricted to a significantly smaller range than the RGB gamut. The problem illustrated by Figure 1-7 is to display the same image on both devices with a minimum of visual mismatch. The solution to the problem is to match the colors of the image using profiles for both devices and one or more color management modules. A **profile** is a structure that provides a means of defining the color characteristics of a given device in a particular state. For more information, see "Profiles" (page 49).

Color Management Systems

Members of the computer and publishing industries have developed **color management systems (CMSs)** to convert colors from the color space of one

device to the color space of another device (for example, from a scanner to a monitor). The components of a color management system include

- collections of color characteristics (these collections are given various names, such as color tags, precision transforms, or profiles)
- a color management module (CMM) that performs the color matching among, and transformation between, collections of color characteristics; for more information, see “Color Management Modules” (page 58)
- a programming interface for invoking color matching

The goal of these systems is to provide consistent color across peripheral devices and across operating-system platforms. Most CMSs are proprietary, but ColorSync, the platform-independent color management system from Apple Computer, supports the industry-standard color profile specification currently defined by the **International Color Consortium** (ICC). The ICC publishes the *International Color Consortium Profile Format Specification*. To obtain a copy of the specification, or to get other information about the ICC, visit the ICC Web site at <http://www.color.org/>.

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

CHAPTER 2

Introduction to ColorSync

Contents

This section describes ColorSync and the ColorSync Manager. ColorSync is the platform-independent color management system from Apple Computer, Inc. ColorSync provides essential services for fast, consistent, and accurate color calibration, proofing, and reproduction. The ColorSync Manager is the application programming interface (API) to these services on the Mac OS.

Read this section if your software product performs color drawing, printing, or calculation or if your peripheral device supports color. You should also read this section if you are creating a **color management module (CMM)**—a component that implements color-matching, color conversion, and gamut-checking services.

If you are unfamiliar with terms and concepts such as CMM, profile, color space, and color management, or would like to review these topics, read "Introduction to Color and Color Management Systems" (page 25) before reading this section.

"What's New" (page 539) lists the new features available with ColorSync 2.5 and provides links to new and revised material in this document. It also describes where to obtain documentation for other color-related technologies.

About ColorSync

ColorSync is the first system-level implementation of an industry-standard color management system. Even in a system in which all input, output, and display devices are new and perfectly calibrated, there will be differences in device gamuts (the ranges of color that can be reproduced by the devices). ColorSync can correct for such differences in device gamuts, as well as for differences caused by aging of filter sets and lamps on scanners, phosphor decay and ambient light on monitors, and differences in pigments and substrates on output devices such as printers and presses. As a result, it is possible to maintain accurate color across many possible input, display, and output devices.

Developers writing device drivers use the ColorSync Manager to support color matching between devices. Application developers use the ColorSync Manager to communicate with drivers and to present users with color-matching information, such as a device's color capabilities.

Why You Should Use ColorSync

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

The ColorSync Manager addresses these problems by providing applications and color peripheral device drivers with device-independent color-matching and color conversion services based on the ColorSync color management system. With this ColorSync support, the user can quickly and accurately convert color images for optimal results on a specified device.

The ColorSync Advantage

There are many reasons you should consider adding ColorSync support to your products:

- Working with color is more difficult than many people realize, but ColorSync provides a highly effective system to help you perform accurate, industry-standard color management.
- Devices age and are subject to continuous inconsistence due to phosphor aging, pigment changes, substrate differences (white point, absorption, reflectivity), filter aging, and change in the life of the luminant.
- Artists, designers, and prepress experts need to achieve repeatable, reliable, and consistent color—onscreen, in print, and for electronic delivery to multimedia and the Internet. ColorSync is the tool of choice for meeting these requirements.
- There are many color measurement instruments, an array of profiling software, and a wide selection of production tools—page make-up, image

editing, illustration, image database, Photo CD applications, and more—that already support ColorSync.

- ColorSync is widely available to Macintosh users, who make up the large majority of graphic arts and publishing professionals.
- Starting with version 2.5, the ColorSync Manager provides an extensible AppleScript framework that allows users to script many common tasks, such as matching an image or embedding a profile in an image. These AppleScript capabilities, described in "Scripting Support" (page 71), make it possible to automate many workflow processes.

Color Management in Action

Firms that sell clothing through mail order catalogs and Web sites report the most common reason a customer returns an item is color: when the product arrived, it didn't match the color in the catalog or on the monitor. Working with managed color, however, these firms can ensure the original image is scanned accurately, the catalog or monitor displays the color correctly, the output device prints the color correctly, and the customers see the color as accurately as they are able.

ColorSync Manager Overview

This section provides an overview of the ColorSync Manager and how an application or device driver can use it for color conversion, color matching, color gamut checking, profile management, monitor calibration, and creating color management modules (CMMs).

The ColorSync Manager provides a set of routines contained in a system extension. ColorSync also includes a collection of display device profiles for all Apple color monitors, some default profiles for standard color spaces, and a robust default CMM. In addition, the ColorSync control panel, shown in Figure 2-1 (page 54), allows a user to specify a preferred CMM and various default profiles. CMMs and profiles are discussed throughout this material.

To provide its color-matching services, the ColorSync Manager works with color profiles and with one or more color management modules. Your application or driver can supply its own CMM, or you can use the **default CMM**, a robust CMM that is installed as part of the ColorSync extension and

supports all the required and optional functions defined by the ColorSync Manager. The ColorSync Manager relies on the Component Manager to support plug-and-play capability for third-party CMMs. The Component Manager is described in *Inside Macintosh: More Macintosh Toolbox*. For more information on CMMs, see "Color Management Modules" (page 58).

A profile is a table that contains the color characteristics of a given device in a particular state. ColorSync profiles conform to the format currently defined by the International Color Consortium (ICC). Device driver developers and peripheral manufacturers can provide their own profiles or they can obtain profiles from a number of vendors. For a list of profile vendors, see the ColorSync Web site at <http://colorsync.apple.com/>. Profiles are described in detail in "Profiles" (page 49).

ColorSync Versions

This document covers the ColorSync Manager through version 2.5. Most existing code written to use version 2.0 or 2.1 of the ColorSync Manager should continue to work with version 2.5 without modification.

This document uses a specific version number, such as "version 2.5 of the ColorSync Manager," only where necessary to identify features associated with a particular version. It may use "2.x" to refer inclusively to ColorSync versions 2.0, 2.1, and 2.5, or to refer to the profile format used with these versions—the meaning should be clear from the context. For more information on profile version numbers, see "ColorSync and ICC Profile Format Version Numbers" (page 50).

For a description of the `Gestalt` information, shared library version numbers, CMM version numbers, and ColorSync header files you use with different versions of the ColorSync Manager, see "ColorSync Version Information" (page 525).

Minimum Requirements For Running ColorSync 2.5

ColorSync version 2.5 requires Mac OS version 7.6.1 or newer, running on a Power Macintosh or on a 68K Macintosh computer with a 68020 or greater processor. For information on system and CPU requirements for previous ColorSync versions, and for related information such as Gestalt, shared library, and CMM versions, see "ColorSync Version Information" (page 525).

Programming Interface

The ColorSync Manager application programming interface (API) allows your application or device driver to handle such tasks as color matching, color conversion, profile management, and profile searching. You can access and read individual tagged elements within a profile, embed profiles in documents, modify profiles, and transform images through various CMMs to perform color matching and profile data transfer from one format to another.

The ColorSync API is summarized in "Summary of the ColorSync Manager" (page 167). You can find detailed information about individual functions, constants, and data types in "ColorSync Reference for Applications and Drivers" (page 217). For code samples that show how to use the ColorSync API, see "Developing ColorSync-Supportive Applications" (page 81). For information on the ColorSync Manager header files, see "ColorSync Header Files" (page 528).

The ColorSync Manager is implemented as a shared library on PowerPC-based computers. For a listing of the shared library version number for each released version of the ColorSync Manager, along with corresponding `Gestalt` selector codes, see "Gestalt, Shared Library, and CMM Version Information" (page 526).

Profiles

To perform color matching or color conversion across different color spaces requires the use of a *profile* for each device involved. Profiles provide the information necessary to understand how a particular device reproduces color. A profile may contain such information as lightest and darkest possible tones (referred to as white point and black point), the difference between specific "targets" and what is actually captured, and maximum densities for red, green, blue, cyan, magenta, and yellow. Together these measurements represent the data which describe a particular color gamut.

The International Color Consortium Profile Format

ColorSync supports the profile format defined by the International Color Consortium (ICC). The ICC format provides a single cross-platform standard for translating color data across devices. The ICC defines several types of profiles, including input, output, and display profiles. Each of these types specifies a different required set of information, but all follow the same format.

The founding members of the ICC include Adobe Systems Inc.; Agfa-Gevaert N.V.; Apple Computer, Inc.; Eastman Kodak Company; FOGRA (Honorary); Microsoft Corporation; Silicon Graphics, Inc.; and Sun Microsystems, Inc. These companies have committed to full support of this specification in their operating systems, platforms, and applications.

To obtain a copy of the *International Color Consortium Profile Format Specification*, or to get other information about the ICC, visit the ICC Web site at <http://www.color.org/>.

ColorSync and ICC Profile Format Version Numbers

The first version of ColorSync used a 1.0 profile format that preceded the ICC profile format definition. Starting with version 2.0 of the ColorSync Manager, ColorSync uses a 2.x profile format that supports all current ICC profile format versions. As of this writing, that includes ICC versions 2.0 and 2.1. The ICC defines the profile format version as part of the profile header. For more information on the differences between these profile format versions, see “ColorSync 1.0 Profiles and Version 2.x Profiles” (page 531).

Source and Destination Profiles

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

Displaying the image requires using another profile, which is associated with the output device, such as a display. The profile for that device is called the **destination profile.** If the image is destined for a display, ColorSync can use the display’s profile (the destination profile) along with the image’s source profile to match the image’s colors to the display’s gamut. If the image is printed, ColorSync can use the printer’s profile to match the image’s colors to the printer, including generating black and removing excessive color densities (known as **undercolor removal**, or **UCR**) where appropriate.

Profile Classes

The ColorSync Manager supports seven classes, or types, of profiles. These classes are defined below. Three of the profile classes define device profiles for different types of devices: input, output, and display devices. The other four profile classes include definitions for an abstract profile, a color space profile, a named color space profile, and a device link profile. The constants used to specify these classes are described in “Profile Class” (page 396).

A **device profile** characterizes a particular device: that is, it describes the characteristics of a color space for a physical device in a particular state. A display, for example, might have a single profile, or it might have several, based on differences in gamma value and white point. A printer might have a different profile for each paper type or ink type it uses because each paper type and ink type constitutes a different printer state. When an application calls a ColorSync Manager color-matching function to match colors between devices, such as a display and a printer, it specifies the profile for each device.

Device profiles are divided into three broad classifications:

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

Each device profile class has its own signature. The ColorSync constants for these signatures are described in “Profile Class” (page 396). For related information, see “Devices and Their Profiles” (page 196).

A **profile connection space (PCS)** is a device-independent color space used as an intermediate when converting from one device-dependent color space to another. Profile connection spaces are typically based on spaces derived from the CIE color space, which is described in “Device-Independent Color Spaces” (page 34). ColorSync supports two of these spaces, XYZ and L*a*b.

A **color space profile** contains the data necessary to convert color values between a PCS and a non-device color space (such as L*a*b to or from L*u*v, or XYZ to or from Yxy), for color matching. The ColorSync Manager uses color space profiles when mapping colors between different color spaces. Color space profiles also provide a convenient means for CMMs to convert between different non-device profiles.

L*a*b, L*u*v, XYZ, and Yxy color spaces are described in “Color Spaces” (page 28).

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

A **device link profile** represents a one-way link or connection between devices. It can be created from a set of multiple profiles, such as various device profiles associated with the creation and editing of an image. It does not represent any device model, nor can it be embedded into images.

For more information on device link profiles, see `CWNewLinkProfile` (page 267) and `CMConcatProfileSet` (page 384).

IMPORTANT

The `CMConcatProfileSet` structure used to create a device link profile includes a field that identifies the one CMM to use for the entire color-matching session across all profiles. However, you should read "How the ColorSync Manager Selects a CMM" (page 84), for a full description of the algorithm ColorSync uses to choose a CMM. ▲

A **named color space profile** contains data for a list of named colors. The profile specifies a device color value and the corresponding CIE value for each color in the list. Profiles are typically stored as individual files in the ColorSync Profiles folder. For example, device-specific profiles provided by hardware vendors should be stored in the ColorSync Profiles folder. The location and use of the ColorSync Profiles folder has changed beginning in version 2.5. For a description of these changes, see "Profile Search Locations" (page 55).

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

Note

The ICC profile format implemented in a ColorSync version 2.x profile is significantly different from the ColorSync 1.0 profile implementation. For more information, see "ColorSync and ICC Profile Format Version Numbers" (page 50). ◆

Embedding a profile in an image guarantees that the image can be rendered correctly on a different system. However, profiles can be large—the largest can be several hundred KB or even larger. A **profile identifier** is an abbreviated data structure that uniquely identifies, and possibly modifies, a profile in memory or on disk, but takes up much less space than a large profile. For example, an application might embed a profile identifier to change just the rendering intent in an image without having to embed an entire new profile. Rendering intents are described in "Rendering Intents" (page 60). For more information on embedding profile information, see "Embedding Profiles and Profile Identifiers" (page 112).

IMPORTANT

A document containing an embedded profile identifier (as opposed to an embedded profile) is not necessarily portable to different systems or platforms. ▲

Profile Properties

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

Profiles contain additional information, such as a specification for how to apply matching. For more information, see "Color Management Modules" (page 58). Profiles may also have a series of optional and private tagged elements. These private tagged elements may contain custom information used by particular color management modules.

Profile Location

In most cases, a ColorSync version 2.x profile is stored in a disk file. However, to support special requirements, a profile can also be located in memory or in an arbitrary location that is accessed by a procedure you specify. See "Profile Location" (page 360) for a description of ColorSync Manager structures for working with profiles that are stored in each of these locations. See "Opening a Profile and Obtaining a Reference to It" (page 95) for information on working with profile locations in your application.

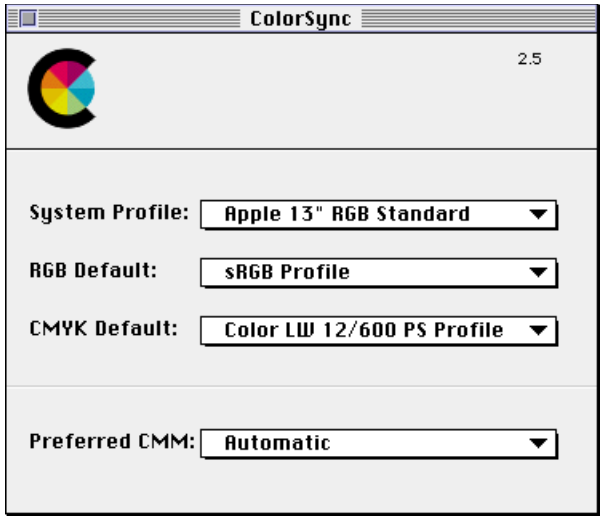
Setting Default Profiles

Prior to version 2.5, the **default system profile** (or simply the system profile) served as the default display profile; it also served as the default profile for color operations for which no profile was specified. The system profile had to be an RGB profile. A user could specify the system profile through the ColorSync control panel (formerly called the "ColorSync™ System Profile" control panel). If a user did not specify a system profile, then by default ColorSync used the Apple 13-inch color display profile.

Because the system profile was used for two dissimilar functions (default display profile and default profile for some RGB operations), there were several limitations:

- A user could not specify default profiles for color spaces other than RGB.
- A user could not specify separate profiles for more than one monitor.
- When matching an image without an embedded monitor profile, no matching occurred because the source and destination profiles were the same (system) profile.

Figure 2-1 The ColorSync control panel



Starting with ColorSync version 2.5, a user can set default profiles for RGB and CMYK color spaces, as well as for the system profile, using the ColorSync control panel shown in Figure 2-1. In addition, your application can call routines to get and set default profiles for the RGB, CMYK, Lab, and XYZ color spaces. As in previous versions of ColorSync, you can also call routines to get and set the current system profile.

Also starting with ColorSync version 2.5, a user can specify a separate profile for each monitor using the Monitors & Sound control panel. In addition, your application can call routines to get and set the profile for each display.

IMPORTANT

When a user sets a profile for a monitor in the Monitors & Sound control panel, ColorSync makes that profile the current system profile. When your application sets a profile for a monitor, it may also wish to make that profile the system profile. ▲

Because ColorSync version 2.5 provides capabilities for getting and setting default profiles for color spaces and for assigning a profile to each monitor, your application and anyone using it can more precisely specify source and destination profiles. For example, your application can set the destination profile for an operation to be the profile for a specific monitor or the source profile to be the default CMYK profile.

IMPORTANT

Under certain conditions, functions such as `NCMUseProfileComment` (page 290) still use the system profile, so you should set the system profile to an appropriate value, such as a profile for your main display. ▲

For information on getting and setting profiles in code, see “Getting and Setting Default Profiles by Color Space” (page 297). “Monitor Calibration and Profiles” (page 67), describes how a user can specify a separate profile for each available monitor.

Profile Search Locations

ColorSync uses the ColorSync Profiles folder as a common location for profile files. When you install ColorSync, for example, it puts a number of default monitor profiles in the ColorSync Profiles folder. Users should also store custom

profiles there, and ColorSync functions that search for profiles begin their search in the profiles folder.

Starting with ColorSync version 2.5, the ColorSync Profiles folder is located in the System folder; in earlier versions the folder was named "ColorSync™ Profiles" and was located in the Preferences folder. The new location protects profile files from deletion when the Preferences folder is deleted. More importantly, placement in the System folder will allow the profiles folder to become a "magic" folder, providing the following benefits:

- Starting with version 8.5 of the Mac OS, profiles dragged onto the System folder are automatically routed to the profiles folder.
- Starting with version 8.5 of the Mac OS, ColorSync can use the Toolbox `FindFolder` routine to find the profiles folder, using the Folder Manager constant `kColorSyncProfilesFolderType`.

IMPORTANT

Your application should continue to call ColorSync's `CMGetColorSyncFolderSpec` (page 302) function to obtain the location of the profiles folder—it should not use a hard-coded path to a specific folder. ▲

For backward compatibility, ColorSync automatically inserts into the new profiles folder an alias to the old location (inside the Preferences folder), if that folder exists and contains any profiles.

Where ColorSync Searches for Profiles

Prior to ColorSync 2.5, profile search routines such as `CMNewProfileSearch` (page 308) looked for profiles only in the profiles folder (within the Preferences folder). Starting with version 2.5, the search routines look in the following locations:

- in the ColorSync Profiles folder (within the System folder)
- in first-level subfolders of the ColorSync Profiles folder
- in locations specified by aliases in the ColorSync Profiles folder (whether the aliases are to single profiles or to folders containing profiles)

With this searching support, you can group profiles in subfolders within the profiles folder (one level of subfolders is currently allowed). For example, you might store all scanner profiles in one folder and a variety of monitor profiles for your primary monitor in another. You can also store aliases to other profiles

and profile folders within the ColorSync Profiles folder. ColorSync search routines will find all profiles in the specified locations.

“The Profile Cache and Optimized Searching” (page 57) describes how your application can perform optimized profile searching with ColorSync 2.5.

Where ColorSync Does Not Look for Profiles

Because profile searching in ColorSync 2.5 can only go two levels deep, ColorSync search routines will not find a profile in the following cases:

- The profile is located in a folder that is within a folder in the profiles folder (requires more than two levels of searching).
- The profile is located in a folder that is within a folder specified by an alias in the profiles folder (again, requires more than two levels of searching).
- The profile is in a folder whose name starts with a parenthesis.
- The profile is specified by an alias to an unmounted volume (only mounted volumes are searched).

Temporarily Hiding a Profile Folder

To temporarily hide a folder from ColorSync’s search path, put parentheses around the name of the folder or the alias to the folder.

The Profile Cache and Optimized Searching

Starting with version 2.5, ColorSync creates a cache file (containing private data) in the Preferences folder to keep track of all currently-installed profiles. The cache stores key information about each profile, using a smart algorithm that avoids rebuilding the cache unless the profile folder has changed.

ColorSync takes advantage of the profile cache to speed up profile searching. This optimized searching can help your application speed up some operations, such as displaying a pop-up menu of available profiles.

ColorSync’s intelligent cache scheme provides the following advantages in profile management:

- The cache contains information including the name, header, script code, and location for each installed profile, so that once the cache has been built,

ColorSync can supply the information your application needs for many tasks without having to reopen any profiles.

- When you call a search routine, ColorSync can quickly determine if there has been any change to the currently-installed profiles. If not, ColorSync can supply information from the cache immediately, giving the user a pleasing performance experience.

ColorSync 2.5 provides a flexible new routine, `CMIterateColorSyncFolder` (page 304), that takes full advantage of the profile cache to provide truly optimized searching and quick access to profile information. For an example of how to use this routine in your application, see "Performing Optimized Profile Searching" (page 130).

IMPORTANT

Your application should use the `CMIterateColorSyncFolder` (page 304) function, or one of the other ColorSync search functions described in "Profile Searching" (page 303), to search for a profile, even if you are only looking for one file. Do not search for a profile by obtaining the location of the profiles folder and searching for the file directly. ▲

Note that calls to the ColorSync search routines available before version 2.5 cannot take full advantage of the profile cache. For example, with the `CMNewProfileSearch` (page 308) routine, the caller passes in a search criteria and gets back a list of profiles that match that criteria. Before version 2.5, ColorSync had to open each profile to build the list, and the caller was likely to open each profile again after getting the list back. With version 2.5, ColorSync can at least use the profile cache to narrow down the list (unless the search criteria asks for all profiles!), but it cannot fully optimize the search process.

Color Management Modules

A color management module (CMM) is a component that implements color-matching, color conversion, and gamut-checking services. A CMM uses profiles to convert and match a color in a given color space on a given device to or from another color space or device.

Each profile header includes a field that specifies a CMM to use for performing color matching involving that profile. If two profiles in a color-matching session specify different CMMs, or if a specified CMM is unavailable or unable to perform a requested function, the ColorSync Manager follows an algorithm,

described in "How the ColorSync Manager Selects a CMM" (page 84), to determine which CMM to use.

ColorSync ships with a robust CMM that is installed as part of the ColorSync extension. This CMM supports all the required and optional functions defined by the ColorSync Manager, so it can always be used as a default CMM when another CMM is unavailable or unable to perform an operation.

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

When colors from one device's gamut are displayed on a device with a different gamut, as shown in Figure 1-7 (page 41), ColorSync can minimize the perceived differences in the displayed colors between the two devices by mapping the out-of-gamut colors into the range of colors that can by produced by the destination device.

A CMM uses lookup tables and algorithms for color matching, previewing color reproduction capabilities of one device on another, and checking for out-of-gamut colors (colors that cannot be reproduced). Although ColorSync provides a default CMM, device manufacturers and peripheral developers can create their own CMMs, tailored to the specific requirements of their device. For information on creating CMMs, see "Developing Color Management Modules" (page 429) and "ColorSync Reference for Color Management Modules" (page 467). ColorSync also provides the Kodak CMM as a "custom install" feature, for users who wish to work with that CMM.

Setting a Preferred CMM

Starting with version 2.5, the ColorSync control panel, shown in Figure 2-1 (page 54), lets you choose a **preferred CMM** from any CMMs that are present (that is, registered with the Component Manager—see "Creating a Component Resource for a CMM" (page 432) for a description of how to create a CMM the ColorSync Manager can use).

If you choose a preferred CMM with the ColorSync control panel, and if that CMM is available, ColorSync will attempt to use that CMM for all color conversion and matching operations. If you specify "Automatic" instead, or if the specified CMM is no longer present or cannot provide the required

matching service, or for versions prior to version 2.5, ColorSync follows an algorithm described in "How the ColorSync Manager Selects a CMM" (page 84) to determine which available CMM to use for matching.

Note that if you want color conversion and matching operations to use the same CMM-selection algorithm they did in versions prior to ColorSync 2.5, specify "Automatic" in the ColorSync control panel. ◆

Starting with ColorSync 2.5, your application can determine the preferred CMM by calling the function `CMGetPreferredCMM` (page 292).

Rendering Intents

Rendering intent refers to the approach taken when a CMM maps or translates the colors of an image to the color gamut of a destination device—that is, a rendering intent specifies a gamut-matching strategy. The ICC specification defines a profile tag for each of four rendering intents: **perceptual matching**, **relative colorimetric matching**, **saturation matching**, and **absolute colorimetric matching**. These rendering intents are described in Table 2-1.

Table 2-1 ICC rendering intents and typical image content

ICC term	Description	Typical content
perceptual matching	All the colors of one gamut are scaled to fit within another gamut. Colors maintain their relative positions. Usually produces better results than colorimetric matching for realistic images such as scanned photographs. The eye can compensate for gamuts differences, such as in Figure 1-7 (page 41), and when printed on a CMYK device, the image may look similar to the original on an RGB device. A side effect is that most of the colors of the original space may be altered to fit in the new space.	photographic

Table 2-1 ICC rendering intents and typical image content (continued)

ICC term	Description	Typical content
relative colorimetric matching	Colors that fall within the overlapping gamuts of both devices are left unchanged. For example, to match an image from the RGB gamut onto the CMYK printer gamut in Figure 1-7, only the colors in the RGB gamut that fall outside the printer gamut are altered. Allows some colors in both images to be exactly the same, which is useful when colors must match quantitatively. A disadvantage is that many colors may map to a single color, resulting in tone compression. All colors outside the printer gamut, for example, would be converted to colors at the edge of its gamut, reducing the number of colors in the image and possibly altering its appearance. Colors outside the gamut are usually converted to colors with the same lightness, but different saturation, at the edge of the gamut. The final may be lighter or darker overall than the original image, but the blank areas will coincide.	spot colors
saturation matching	The relative saturation of colors is maintained as well as can be achieved from gamut to gamut. Colors outside the gamut of the “to” space are usually converted to colors with the same saturation of the “from” space, but with different lightness, at the edge of the gamut. Can be useful for some graphic images, such as bar graphs and pie charts, when the actual color displayed is less important than its vividness.	business graphics
absolute colorimetric matching	Preserves native device white point of source image instead of mapping to D50 relative. Most often using in simulation or proofing operations where a device is trying to simulate the behavior of another device and media. For example, simulating newsprint on a monitor with absolute colorimetric intent would allow white space to be displayed onscreen as yellowish background because of the differences in white points between the two devices.	no typical content (most often used in proofing)

Color professionals and technically-sophisticated users are likely to be familiar with the ICC terms for rendering intent and the gamut-matching strategies they represent. If your application is aimed at novice users, however, you may prefer to add a simplified terminology based on the typical image content associated with a rendering intent (for example, perceptual matching is commonly used for photographic images). Table 2-1 lists the ICC-specified rendering intents and the corresponding image content term, where applicable. Note that there is no simplified terminology for describing absolute colorimetric matching. However, novice users are not likely to need this rendering intent.

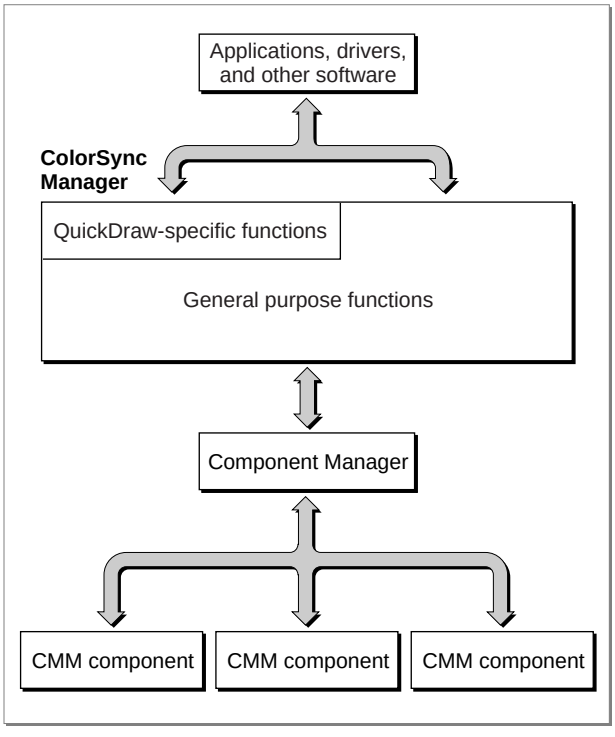
For information about the actual rendering intent tags, you can obtain the latest ICC profile specification by visiting either the ICC Web site at <http://www.color.org/> or the ColorSync Web site at <http://colorsync.apple.com/>.

When Color Matching Occurs

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

When an application or device driver uses the ColorSync Manager functions for color matching, it specifies the source and destination profiles. If it does not specify the source profile or the destination profile for a matching operation, ColorSync substitutes a default profile. See "Setting Default Profiles" (page 54) for more information.

Figure 2-2 The ColorSync Manager and the Component Manager



Color matching between the source and destination color spaces happens inside the color management module (CMM) component. Figure 2-2 shows the relationship between your application or device driver, the ColorSync Manager, the Component Manager, and one or more available CMM components.

Your application can call any ColorSync Manager function, whether QuickDraw-specific or general purpose. One of three things then happens:

- The ColorSync Manager routine performs the operation directly.
- The ColorSync Manager communicates with a CMM through the Component Manager.
- The ColorSync Manager calls other ColorSync routines that communicate with a CMM through the Component Manager.

General purpose and QuickDraw-specific functions are described in the following sections.

General Purpose Color-Matching Functions

A **general purpose color-matching function** is one that uses a color world to characterize how to perform color-matching. General purpose functions depend on the information contained in the profiles that you supply when you set up the color world. You can define a color world for color transformations between a source profile and a destination profile, or define a color world for color transformations between a series of concatenated profiles.

"Creating a Color World to Use With the General Purpose Functions" (page 105) provides a code sample for working with general purpose functions. "Matching Colors Using General Purpose Functions" (page 261) lists the general purpose functions and provides a description of each function.

In contrast to the general purpose color-matching functions, the "QuickDraw-Specific Color-Matching Functions" (page 64) are tailored for color-matching with QuickDraw. Note, however, that you can also use the general purpose functions when working with QuickDraw—for example, if you need the greater level of control the general purpose functions provide.

QuickDraw-Specific Color-Matching Functions

A **QuickDraw-specific color-matching function** is one that uses QuickDraw to provide images showing consistent colors across displays. The ColorSync Manager provides two QuickDraw-specific functions that your application can call to draw a color picture to the current display:

- `NCMBeginMatching` (page 285) uses the source and destination profiles you specify to match the colors of the source image to the colors of the device for which it is destined.
- `NCMDrawMatchedPicture` (page 288) matches a QuickDraw picture's colors to a destination device's color gamut as the picture is drawn, using the specified destination profile. Uses the system profile as the initial source profile but switches to any embedded profiles as they are encountered.

"Matching to Displays Using QuickDraw-Specific Operations" (page 101) provides a code sample for working with QuickDraw-specific functions. "Matching Colors Using QuickDraw-Specific Functions" (page 284) lists the QuickDraw-specific functions and provides a description of each function. Note

that the QuickDraw-specific functions call upon the general purpose functions to perform their operations, as shown in Figure 2-2 (page 63).

Converting Between Color Spaces

Color conversion, which does not require the use of color profiles, is a much simpler process than color matching. The ColorSync Manager provides functions your application can call to convert a list of colors within the same base family—that is, between a 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.

Note
The color conversion functions do not support conversion of HiFi colors. ◆

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 is also used in the ColorSync Manager 2.x profile header `CM2Header` (page 354).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 one of its derived spaces, you don't need to specify source and destination profiles. Instead, you just call the appropriate ColorSync Manager function to convert between the desired color spaces. In cases where you're converting between XYZ and Lab or Luv spaces, a "white reference" is required to perform the conversion. This reference is expressed in the XYZ color space, and provides a way to specify the theoretical illuminant (for example, D65) under which the colors are viewed.

Note
Prior to version 2.1, the ColorSync Manager used a component to implement color conversion. An application had to open a connection to the component with the Component Manager, then pass the component instance as a parameter to the color conversion functions. For example, the `CMXYZToLab` function performs the same conversion as `CMConvertXYZToLab`, but takes a first parameter of

```
ComponentInstance ci
```

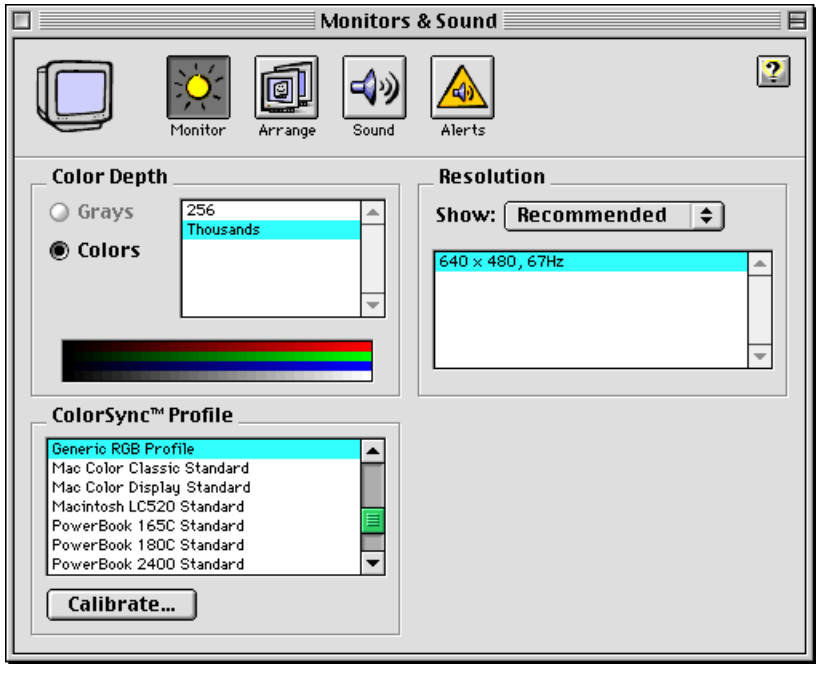
For backward compatibility, component-based color conversion functions such as `CMXYZToLab` are still supported. However, their use is discouraged, and they are not guaranteed to work in future versions (nor are they documented here). ◆

Monitor Calibration and Profiles

Since ColorSync was first introduced, a common question from end users has been "Where is the ColorSync profile for my monitor?" The answer is that because some monitor manufacturers do not supply ColorSync profiles for their products, purchasers of third-party monitors may not have access to a profile that is specific to their monitor. As a result, they are unable to use ColorSync effectively.

Even when a user has a factory-supplied profile, switching to a different monitor setup can reduce the profile's accuracy. For example, if the user changes the monitor's gamma value and white point, the original profile is no longer useful. The user needs to run a calibration application (if one is available) to generate a new ColorSync profile for the new monitor settings.

Figure 2-3 Monitors & Sound Control Panel for ColorSync 2.5



Starting with version 2.5, ColorSync uses the Monitors & Sound control panel to provide a monitor calibration framework to help users obtain the monitor profiles they need. Figure 2-3 (page 68) shows the new Monitors & Sound control panel. Note that the list of gamma values (Mac Standard Gamma, uncorrected gamma) has been removed because that function is now part of the calibration process.

Note
If you develop a utility that adjusts gamma or modifies other calibration values, you should modify your software so that it uses the monitor calibration framework to gain system-level support. u

Setting a Profile for Each Monitor

Because Monitors & Sound displays a panel for each available monitor, a user can also select, for each monitor, a separate profile from the list of available profiles. When a user sets a profile for a monitor in the Monitors & Sound control panel, ColorSync makes that profile the current system profile, as described in “Setting Default Profiles” (page 54). When your application sets a profile for a monitor, it may also wish to make that profile the system profile. If so, it must set the system profile explicitly by calling `CMSetSystemProfile` (page 295). For information on how to set monitor profiles in your code, see “Getting and Setting Monitor Profiles by AVID” (page 299).

Calibration

Calibration and characterization are related terms, but with important differences. **Calibration** is the process of setting a device’s parameters according to its factory standards. This is also known as linearizing or linearization. **Characterization** is the process of learning the color character of a monitor so that a profile can be created to describe it. Unlike input and output devices, whose calibration and characterization steps are not the same, a monitor is calibrated and characterized in one step.

The Calibrate button on the Monitors & Sound control panel provides the launching point for monitor calibration. A user can calibrate each monitor and create one or more color profiles for each, based on variations in gamma, white point, and so on. For related information on profiles, see “Profiles” (page 49) and “Devices and Their Profiles” (page 196).

AppleVision and Apple ColorSync monitors are self-calibrating, so you will not see a Calibrate button for these monitors, unless there is a third-party calibrator installed in your Extensions folder.

Calibrating a monitor can be a challenging task for a naive user, but Apple Computer supplies a default calibrator that leads the user through a series of calibration steps. Using the default calibrator, even a novice should have a reasonable chance for success.

Note
There are limits to the effectiveness of monitor calibration by users. For example, some monitors, due to age or condition, cannot be calibrated, and a small percentage of the user population is color-blind. ◆

The calibration framework uses a plug-in architecture that is fully accessible to third-party calibration plug-ins. When a user clicks on the Calibrate button, the Monitors & Sound control panel provides a list of all available calibrator plug-ins. To appear in the list, a plug-in must meet the following criteria:

- It must be stored in the Extensions folder.
- It must be a shared library (file type `'shlb'`).
- Its shared library must export the symbols `CanCalibrate` and `Calibrate`.
- It should have a unique creator type (registered with Apple).
- The name of the library's code fragment (specified in the `'cfrg'` resource) must be unique (among all currently loaded shared libraries) and begin with `'Cali'`. For example, you might want to name the library by appending your creator type to `'Cali'`.

You can find source code for sample monitor calibration plug-ins in the ColorSync 2.5.1 SDK. If you plan to create a monitor calibration plug-in, you should read the next section "Video Card Gamma" (page 70).

Video Card Gamma

Starting with version 2.5, ColorSync supports an optional profile tag for video card gamma, which you specify with the `cmVideoCardGammaTag` constant. The tag specifies gamma information, stored either as a formula or in table format, to be loaded into the video card when the profile containing the tag is put into use. When you call the function `CMSetProfileByAVID` (page 300) and specify a profile that contains a video card gamma tag, ColorSync will extract the tag from the profile and set the video card based on the tag.

IMPORTANT

The function `CMSetSystemProfile` (page 295) does not retrieve video card gamma data to set the video card. Only the `CMSetProfileByAVID` function currently sets video card gamma data. ▲

If you provide monitor calibration software, you should include the video card gamma tag in the profiles you create. For information on the constants and data types you use to work with video card gamma, see "Video Card Gamma Constants" (page 421) and "Video Card Gamma" (page 386). You can get more information about AVID values from the Display Manager SDK.

Scripting Support

Starting with version 2.5, the ColorSync Manager provides AppleScript support that allows users to script many common color-matching tasks. To provide this support, ColorSync now runs as a faceless background application (one with no user interface), rather than as a standard extension. By running as a background application, ColorSync can avoid namespace collisions and time-outs during long operations, and it can have its own AppleScript dictionary.

Note
You can examine ColorSync's full AppleScript dictionary by dragging the file "ColorSync Extension" from your Extensions folder onto the Script Editor application (usually located in the AppleScript folder within the Apple Extras folder). ◆

Scriptable Properties

ColorSync provides scriptable support for getting and setting the following properties:

- system profile (the default system profile)
- default profiles for RGB, CMYK, Lab, and XYZ color spaces
- quit delay (the time in seconds for auto-quit, where 0 = never)
- profile location (a file specification)

For the following, you can only get, not set, the property:

- profile folder (the ColorSync profile folder)

Location is the only property currently supported for profiles, but future support is planned for additional profile properties.

Scriptable Operations

ColorSync supports the following scriptable operations:

- Matching an image.
- Matching an image with a device link profile.
- Proofing an image or a series of images.

- Embedding a profile in an image (very helpful for converting archives of legacy film).

Scriptable image operations currently work only on TIFF files, but support for other formats is planned.

Extending the Scripting Framework

The scripting framework uses a plug-in architecture that is fully accessible to third-party scripting plug-ins. When a user invokes a script to perform a ColorSync operation on an image, ColorSync (operating as a faceless background application) automatically builds a list of all available scripting plug-ins. It then attempts to call each of the plug-ins in the list until one of them successfully executes the desired operation. To appear in the list, a plug-in must meet the following criteria:

- It must be stored in the Extensions folder.
- It must be a shared library (file type `'shlb'`).
- It should have a unique creator type (registered with Apple).
- The name of the library's code fragment (specified in the `'cfrg'` resource) must be unique (among all currently loaded shared libraries) and begin with `'CSSP'`. For example, you might want to name the library by appending your creator type to `'CSSP'`.

Sample Scripts

The ColorSync SDK includes several sample scripts that demonstrate how to perform common operations. You can use the scripts as is, or borrow from them for your own custom scripts. For more information, see the detailed Read Me files that accompany the sample scripts.

Multiprocessor Support

Starting with version 2.5, ColorSync's default CMM can take advantage of multiple processors. Multiprocessor support is transparent to your code—the CMM invokes it automatically if the required conditions are met.

When ColorSync Uses Multiple Processors

The default CMM takes advantage of multiprocessor support only if the following conditions are satisfied:

1. The MPLibrary was successfully loaded at boot time.
2. The CMM successfully links against the MPLibrary at runtime.
3. The number of processors available is greater than one.
4. The number of rows in the image is greater than the number of processors.
5. The source and destination buffers have the same number of bytes per row or have different locations in memory.

Unless all of these conditions are met, matching will proceed without acceleration. Multiprocessor support is currently supplied only for the following component request codes:

- `kCMMMatchBitMap`
- `kCMMMatchPixMap`

As a result, the default CMM invokes multiprocessor support only in response to the general purpose `CWMatchPixMap` (page 272) and `CWMatchBitmap` (page 276) functions, or when those calls are invoked as a result of a call to the QuickDraw-specific matching routines, such as `NCMBeginMatching` (page 285).

Efficiency of ColorSync's Multiprocessor Support

Depending on the image and other factors, ColorSync's matching algorithms take advantage of multiple processors with up to 95% efficiency (your mileage may vary). If you have two processors, for example, ColorSync can complete a matching operation in as little 53% of the time required by one processor. Additional processors should provide proportional improvement.

QuickDraw GX and the ColorSync Manager

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

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

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

QuickDraw GX version 1.1.2 or earlier uses ColorSync 1.0. However, because the ColorSync Manager provides robust backward compatibility, including continued support of the ColorSync 1.0 API, you can use the ColorSync Manager with QuickDraw GX. For more information about the ColorSync Manager's backward compatibility, see "Version and Compatibility Information" (page 525).

IMPORTANT

For information on changes to the printing and graphics architecture in the Mac OS that affect QuickDraw GX, see the technote <http://gemma.apple.com/technotes/gxchange.html>.

How the ColorSync Manager Uses Memory

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

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

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

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

What Users Can Do With ColorSync-Supportive Applications

ColorSync allows your application or driver to maintain consistent color across devices and across platforms. You can also let users perform quick and inexpensive color proofing and see in advance which colors cannot be printed on their printers. This section provides an overview of these and other color management features you can provide. See “Developing ColorSync-Supportive Applications” (page 81) and “Developing ColorSync-Supportive Device Drivers” (page 195) for information on how to implement specific features.

Display Matching

When your application displays an image that contains one or more embedded profiles, it can use ColorSync to make sure the user experiences consistent color from one display to another. If a color cannot be reproduced exactly on a particular destination device, the ColorSync Manager can map the color to a similar color that is in the color gamut of the device.

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

Gamut Checking

Because not all colors can be rendered on all devices, you may want your application to warn users when a color they choose is out of gamut for the currently selected destination device. For example, you can use gamut checking to see if a given color is reproducible on a particular printer. If the color is not directly reproducible—that is, if it is out of gamut—you can alert the user to that fact. The ColorSync Manager provides the `CWCheckPixMap` and `CWCheckColors` functions for checking a color against a device's profile to see if it is in or out of gamut for the device. Your application can then display the results of this check to the user.

Soft Proofing

Using the destination device's profile, your application can enable users to preview on a monitor what a color image will look like on a particular device. Further, it enables remote proofing between client and prepress service. This simulation of a device's output can save the user considerable time and cost.

Device Link Profiles

Most users use the same device configuration for scanning, viewing, and printing over a period of time. Your application can allow users to create a device link profile. A *device link profile* is a means of storing in a single profile a series of linked profiles that correspond to a specific configuration in the order in which the devices in that configuration are normally used. A device link profile represents a one-way link or connection between devices. It does not represent any device model, nor can it be embedded into images.

Calibration

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

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

A user may want to improve the reproduction quality of a device without returning the device to a standard state. Your application can create a profile based on the current state of the device, then use the profile to characterize that device. This approach to calibration maintains the existing dynamic density range while improving the device's overall quality.

Note
You can also provide a monitor calibration plug-in, as described in "Monitor Calibration and Profiles" (page 67). ◆

CHAPTER 3

Developing ColorSync-Supportive Applications

Contents

This section describes how your application can use the ColorSync Manager to provide many color management services. For a complete list, see “Developing Your ColorSync-Supportive Application” (page 91).

Before you read this section, you should read “Introduction to Color and Color Management Systems” (page 25) and “Introduction to ColorSync” (page 45). These sections provide an overview of color theory and color management systems (CMSs), define key terms, and describe the ColorSync Manager.

If you are developing a device driver that supports ColorSync, you should read this section in addition to “Developing ColorSync-Supportive Device Drivers” (page 195).

If your application works with images created by other applications, you should at least read “Providing Minimal ColorSync Support” (page 93), which explains how to preserve profiles embedded in images.

While reading this section, refer to “ColorSync Reference for Applications and Drivers” (page 217) for more information about the functions, constants, and data types used here.

“ColorSync Version Information” (page 525) describes the `Gestalt` information, shared library version numbers, CMM version numbers, and ColorSync header files you use with different versions of the ColorSync Manager. It also includes CPU and Mac OS system requirements.

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.

“What’s New” (page 539) explains where to get information on the Color Picker Manager, which provides your application with a standard dialog box for soliciting a color choice from users.

You should read “Important Note on Code Listings” (page 22) before working with the code in this chapter.

About ColorSync Application Development

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

color-matching system. Because ColorSync supports the profile format defined by the International Color Consortium (ICC), a color image a user creates can be color matched, rendered, and modified by another user running another application on another platform that supports the format. Conversely, your application can modify and color match images created by other applications that support ColorSync or a CMS that includes support for the ICC profile format. For information on profile format version numbers, see “ColorSync and ICC Profile Format Version Numbers” (page 50).

“ColorSync Version Information” (page 525) describes the `Gestalt` information, shared library version numbers, CMM version numbers, and ColorSync header files you use with different versions of the ColorSync Manager. It also describes CPU and Mac OS system requirements.

About the ColorSync Manager Programming Interface

The ColorSync Manager programming interface allows your application to handle tasks such as color matching, color conversion, profile management, profile searching and accessing, reading individual tagged elements within a profile, embedding profiles in documents, and modifying profiles.

The ColorSync API is summarized in “Summary of the ColorSync Manager” (page 167). You can find detailed information about individual functions, data types, and constants in “ColorSync Reference for Applications and Drivers” (page 217). The ColorSync Manager includes a number of interface files you may need for your development efforts. These files are described in “ColorSync Header Files” (page 528).

What Should a ColorSync-Supportive Application Do?

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

Your application can provide an interface that offers pop-up menus or other user interface items allowing a user to choose which profile to associate with an image and how an image is rendered. It can show the user the colors of an

image that are in or out of gamut for a particular device on which the image is to be produced and how ColorSync adjusts for colors that are out of gamut. This allows the user to preview differences that occur in the color-matching transition between gamuts and make corrections if necessary.

Most of the terms and operations mentioned in this section are defined in "Introduction to Color and Color Management Systems" (page 25) and "Introduction to ColorSync" (page 45); see also "Glossary" (page 553).

At a Minimum

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

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

It is important for your application to tag an image with the profile for the device used to create the image and to preserve existing tagging because a picture that is not tagged assumes use of a default profile as described in "Setting Default Profiles" (page 54). If the picture is moved to a different system that uses a different default profile, the picture will display differently. "Providing Minimal ColorSync Support" (page 93) explains how to preserve embedded profiles, and "Embedding Profiles and Profile Identifiers" (page 112) explains how to tag an image. Some of these features are described in greater detail in the rest of this material.

Storing and Handling Profiles

Profiles for use with the ColorSync Manager are stored in the ColorSync Profiles folder. The precise location of this folder can vary for different versions of ColorSync, as described in "Setting Default Profiles" (page 54). When you install ColorSync, the ColorSync Profiles folder contains a selection of display profiles for all Apple color monitors, as well as default profiles for standard color spaces and profiles for several Apple printers.

Starting with ColorSync 2.5, a user can select a default profile for certain color spaces from the ColorSync control panel, as described in "Setting Default Profiles" (page 54). Also starting with version 2.5, the Monitors & Sound control panel allows the user to select a separate profile for each monitor, as described in "Monitor Calibration and Profiles" (page 67).

Your application specifies the profiles for color matching when the application calls a ColorSync Manager function. For most functions, the ColorSync Manager uses one of the default profiles if your application doesn't specify a profile. Some functions require that you explicitly specify a profile by reference.

Device drivers for ColorSync-supportive input and output devices, such as scanners and printers, may install the profiles they use in the ColorSync Profiles folder, making them available to your application for color matching or gamut checking. If your application creates device link profiles, as described in "Creating and Using Device Link Profiles" (page 143), you should place those profiles in the ColorSync Profiles folder.

Your application can provide the interface to allow a user to choose a profile for a specific device. Using the ColorSync Manager functions described in "Profile Searching" (page 303), your application can search the ColorSync Profiles folder and display information about available profiles.

See "Developing Your ColorSync-Supportive Application" (page 91) for a list of programming examples that demonstrate many of these features. As described in "Providing Minimal ColorSync Support" (page 93), your application should, at a minimum, leave profile information intact in the documents and pictures that it imports or copies into its own documents.

How the ColorSync Manager Selects a CMM

When a ColorSync function performs a color matching or color checking operation, it must determine which CMM to use. You typically pass source and destination profiles to a function, either directly or as part of a color world—an abstract private data structure you create by calling either the `NCWNewColorWorld` (page 262), the `CWConcatColorWorld` (page 265), or the `CWNewLinkProfile` (page 267) function. When you call one of the latter two functions to create a color world, you use the `CMConcatProfileSet` (page 384) data structure to specify a series of one or more profiles for the color world.

A profile header contains a `CMMType` field that specifies a CMM for that profile. For example, "Signature of ColorSync's Default Color Management Module" (page 397) describes a signature for the `CMMType` field that specifies ColorSync's

default CMM. When you set up a `CMConcatProfileSet` data structure to specify a series of profiles, you set the structure's `keyIndex` field to specify the zero-based index of the profile within the array of profiles whose CMM (as indicated by its `CMMType` field) ColorSync should use. A CMM specified by this mechanism is called a **key CMM**.

As we have seen, an operation may use more than one profile and there are multiple factors that can affect the choice of a CMM. To deal with these factors, ColorSync uses the following algorithm to select a CMM:

1. Starting with version 2.5, a user can select a *preferred CMM* in the ColorSync control panel. If a user has chosen a preferred CMM, and if that CMM is available, ColorSync uses that CMM for all color checking and color matching operations the CMM can handle.

   If the preferred CMM is not available or cannot handle an operation, ColorSync uses the default CMM, as described in step 4.

2. Prior to ColorSync 2.5, or if the user has not selected a preferred CMM with the ColorSync control panel, or has selected "Automatic," and if the ColorSync function takes a color world reference and the user has initialized the color world with `CWConcatColorWorld` (page 265) or `CWNewLinkProfile` (page 267), ColorSync uses the key CMM.

   If the key CMM is not available or cannot handle an operation, ColorSync uses the default CMM, as described in step 4.

3. Prior to ColorSync 2.5, or if the user has not selected a preferred CMM with the ColorSync control panel, or has selected "Automatic," and if the ColorSync function takes a color world reference and the user has initialized the color world with `NCWNewColorWorld` (page 262) (and therefore *without* a `CMConcatProfileSet` structure), ColorSync uses an **arbitrated CMM** or CMMs—a CMM or CMMs selected from the source and destination profiles as described in "Selecting a CMM by the Arbitration Algorithm" (page 86).

   If an arbitrated CMM is not available or cannot handle an operation, ColorSync uses the default CMM, as described in step 4.

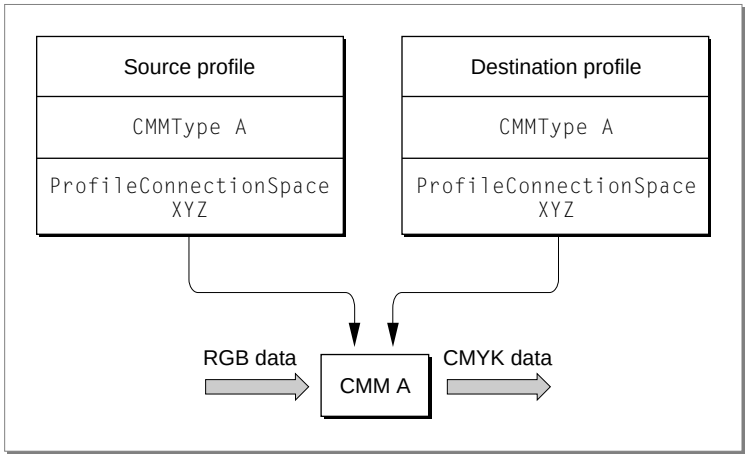
4. If a CMM is not specified by one of the previous three steps, or if a specified CMM is not available or cannot handle an operation, ColorSync uses the *default CMM*—the robust CMM that is installed as part of the ColorSync extension. The default CMM supports all the required and optional functions defined by the ColorSync Manager, and is therefore a suitable CMM of last resort. The signature for the default CMM is specified by the constant `kDefaultCMMSignature`.

Selecting a CMM by the Arbitration Algorithm

This section describes the arbitration algorithm, introduced in "How the ColorSync Manager Selects a CMM" (page 84), ColorSync uses to select one or more arbitrated CMMs for a color matching or color checking operation:

- If the source and destination profiles specify the same CMM and that CMM component is available and able to perform the matching, then the specified CMM maps the colors directly from the color space of the source profile to the color space of the destination profile. This is the simplest scenario, and Figure 3-1 illustrates it.

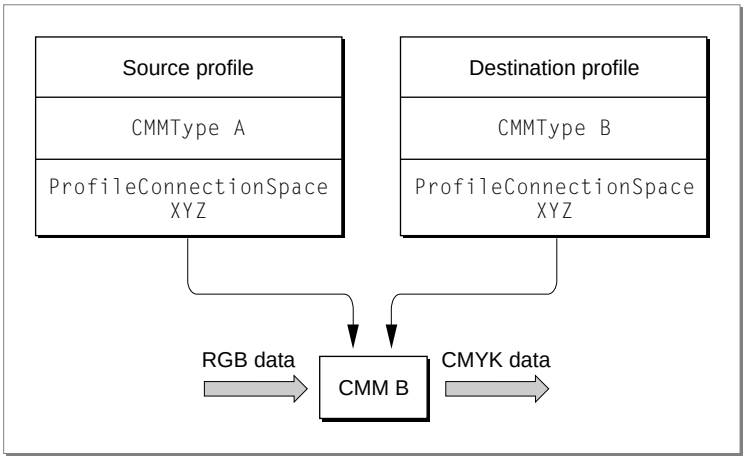
Figure 3-1 Color matching when the source and destination profiles specify the same CMM



- If the source and destination profiles specify different CMMs, then the ColorSync Manager follows these steps to choose the CMM:

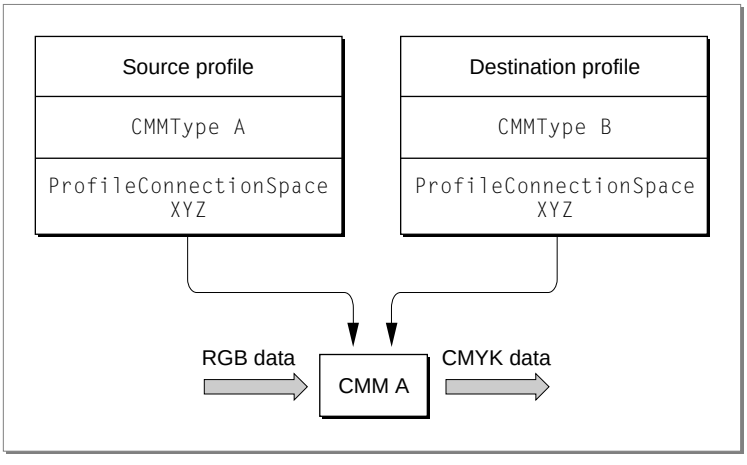
  1. If the CMM specified by the destination profile is available, is able to perform the color matching using the two profiles, and is not the default CMM, then the ColorSync Manager uses this CMM. Figure 3-2 shows this scenario.

Figure 3-2 Color matching using the destination profile's CMM



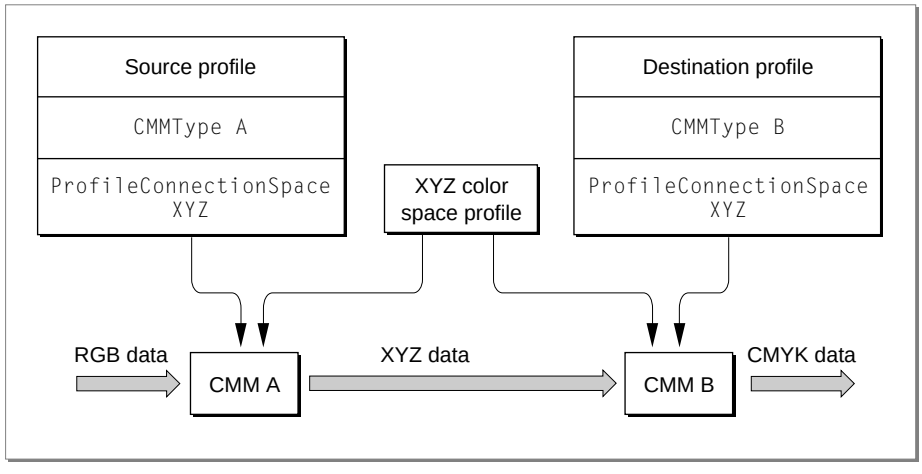
2. If the destination profile's specified CMM is unavailable or unable to perform the color-matching request using the two profiles, then the ColorSync Manager looks for the CMM specified by the source profile. If the CMM specified by the source profile is available, is able to perform the color matching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

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



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

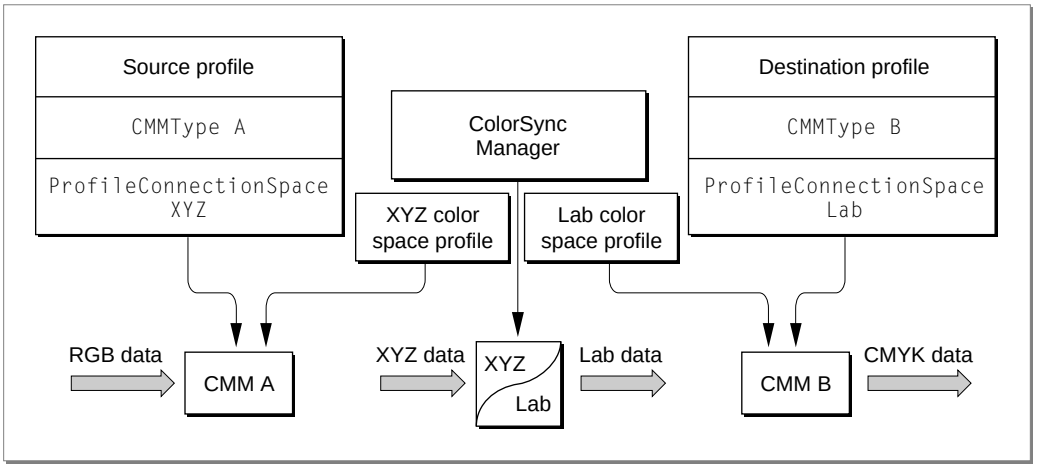
Figure 3-4 Color matching through an XYZ interchange space using both CMMs



4. If both the source-specified CMM and the destination-specified CMM are available, but neither is able to perform the match alone and both profiles specify different interchange color spaces, the ColorSync Manager uses the source profile's CMM to convert the colors of the source image from the source profile's color space to its interchange color space using the appropriate color space profile as the destination profile. The example shown in Figure 3-5 uses the XYZ color space profile as the destination profile. Then the ColorSync Manager inserts a part into the process, itself converting colors from the source profile's interchange color space to the

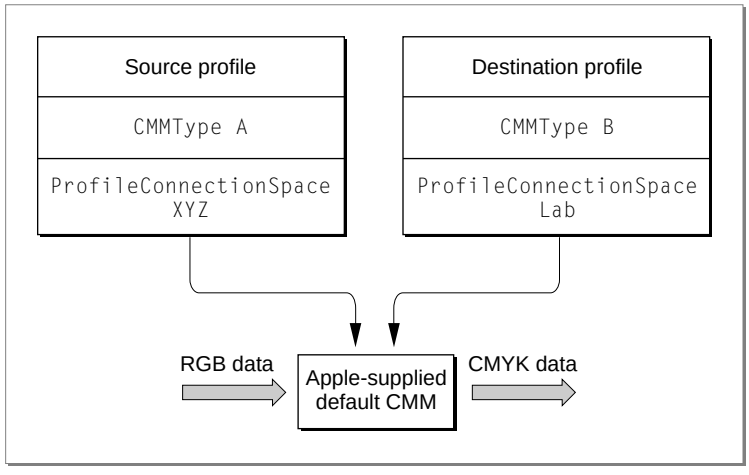
destination profile's interchange color space. Next, the ColorSync Manager uses the CMM specified by the destination profile to convert the colors now specified in the destination profile's interchange color space to colors expressed in the destination profile's color space using the appropriate color space profile as the source profile. The example shown in Figure 3-5 uses the Lab color space profile as the source profile.

Figure 3-5 Matching using both CMMs and two interchange color spaces



- If neither the source nor the destination profile's specified CMM is available or able to perform the color conversion and matching, then the ColorSync Manager uses the default CMM, which will always attempt to perform the match. Figure 3-6 shows this scenario.

Figure 3-6 Color matching using the default CMM



Developing Your ColorSync-Supportive Application

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

This section provides code samples for:

- "Determining If the ColorSync Manager Is Available" (page 92); r**evised for ColorSync 2.5**
- "Providing Minimal ColorSync Support" (page 93)
- "Opening a Profile and Obtaining a Reference to It" (page 95); r**evised for ColorSync 2.5**

- “Identifying the Current System Profile” (page 99)
- “Poor Man’s Exception Handling” (page 98); **new for ColorSync 2.5**
- “Getting the Profile for the Main Display” (page 100); r**evised for ColorSync 2.5**
- “Matching to Displays Using QuickDraw-Specific Operations” (page 101)
- “Creating a Color World to Use With the General Purpose Functions” (page 105)
- “Matching Colors Using the General Purpose Functions” (page 107)
- “Embedding Profiles and Profile Identifiers” (page 112)
- “Extracting Profiles Embedded in Pictures” (page 118)
- “Performing Optimized Profile Searching” (page 130); **new for ColorSync 2.5**
- “Searching for Specific Profiles Prior to ColorSync 2.5” (page 136)
- “Searching for a Profile That Matches a Profile Identifier” (page 139)
- “Checking Colors Against a Destination Device’s Gamut” (page 142)
- “Creating and Using Device Link Profiles” (page 143)
- “Providing Soft Proofs” (page 147)
- “Calibrating a Device” (page 149)
- “Accessing a Resource-Based Profile With a Procedure” (page 149)

Determining If the ColorSync Manager Is Available

To determine whether version 2.5 of the ColorSync Manager is available on a 68K-based or a PowerPC-based Macintosh system, you use the `Gestalt` function with the `gestaltColorMatchingVersion` selector. The function shown in Listing 3-1 returns a Boolean value of `true` if version 2.5 or later of the ColorSync Manager is installed and `false` if not.

Listing 3-1 Determining if ColorSync 2.5 is available

```
Boolean ColorSync25Available (void)
{
    Boolean haveColorSync25 = false;
```

```
    long    version;

    if (Gestalt(gestaltColorMatchingVersion, &version) == noErr)
    {
        if (version >= gestaltColorSync25)
        {
            haveColorSync25 = true;
        }
    }
    return haveColorSync25;
}
```

If your application does not depend on features added for version 2.5 of the ColorSync Manager, use the ColorSync `Gestalt` selector for the ColorSync version you require. For example, you might substitute `gestaltColorSync20` for `gestaltColorSync25` in the previous function (and rename the function appropriately). To identify other versions of ColorSync, use any of the ColorSync `Gestalt` selector constants described in "Gestalt Selector Codes for the ColorSync Manager" (page 217). For related version information, see "ColorSync Version Information" (page 525).

Providing Minimal ColorSync Support

ColorSync supports the profile format defined by the International Color Consortium (ICC). The ICC format provides a single cross-platform standard for translating color data across devices. The ICC's common profile format allows one user to electronically transfer a document containing a color image to another user with the assurance that the original image will be rendered faithfully according to the source profile for the image.

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

To support ColorSync, your application should, at a minimum, leave profile information intact in the documents and pictures it imports or copies. That is, your application should not strip out profile information from documents or pictures created with other applications. Even if your application does not use

the profile information, users may be able to take advantage of it when using the documents or pictures with other applications.

For example, profiles and profile identifiers may be embedded in pictures that a user pastes into documents created by your application. A *profile identifier* is an abbreviated data structure that identifies, and possibly modifies, a profile in memory or on disk. For more information on profile identifiers, see “Searching for a Profile That Matches a Profile Identifier” (page 139). Profiles and profile identifiers can be embedded in formats such as PICT or TIFF files. For files of type `'PICT'`, the ColorSync Manager defines the following picture comments for embedding profiles and profile identifiers, and for performing color matching:

```
/* PicComment IDs */
enum {
    cmBeginProfile          = 220,  /* begin ColorSync 1.0 profile */
    cmEndProfile            = 221,  /* end a ColorSync 2.x or 1.0
                                       profile */
    cmEnableMatching        = 222,  /* begin color matching for either
                                       ColorSync 2.x or 1.0 */
    cmDisableMatching       = 223,  /* end color matching for either
                                       ColorSync 2.x or 1.0 */
    cmComment               = 224   /* embedded ColorSync 2.x profile
                                       information */
};
```

The picture comment `kind` value of `cmComment` is defined for embedded ColorSync Manager version 2.x profiles and profile identifiers. This picture comment is followed by a 4-byte selector that describes the type of data in the picture comment.

```
/* PicComment selectors for cmComment */
enum {
    cmBeginProfileSel       = 0,    /* begining of a ColorSync 2.x
                                       profile; profile data to
                                       follow */
    cmContinueProfileSel    = 1,    /* continuation of a ColorSync
                                       2.x profile; profile data to
                                       follow */
    cmEndProfileSel         = 2     /* end of ColorSync 2.x profile
                                       data; no profile data follows */
    cmProfileIdentifierSel  = 3     /* profile identifier information
                                       follows; the matching profile
```

```
                                        may be stored in the image or
                                        on disk */
};
```

Your application should leave these comments and the embedded profile information they define intact. Similarly, if your application imports or converts file types defined by other applications, your application should maintain the profile information embedded in those files, too.

Your application can also embed picture comments and profiles in documents and pictures it creates or modifies. For information describing how to do this, see “Embedding Profiles and Profile Identifiers” (page 112). *Inside Macintosh: Imaging With QuickDraw* describes picture comments in detail.

Obtaining Profile References

Most of the ColorSync Manager functions require that your application identify the profile or profiles to use in carrying out the work of the function. For example, when your application calls functions to perform color matching or color gamut checking, you must identify the profiles to use for the session. For functions that use QuickDraw, you specify a source profile and a destination profile. For general purpose functions, you specify a color world containing source and destination profiles or a set of concatenated profiles. You can also create a device link profile, which is described in “Creating and Using Device Link Profiles” (page 143), but to do so your application must first obtain references to all the profiles that will comprise the device link profile.

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

Opening a Profile and Obtaining a Reference to It

To open a profile and obtain a reference to it, you call the function `CMOpenProfile` (page 222). You can also obtain a profile reference from the `CMCopyProfile` (page 229), `CWNewLinkProfile` (page 267), and `CMNewProfile`

(page 227) functions. To identify a profile that is file based, memory based, or accessed through a procedure, you must give its location.

The ColorSync Manager defines the `CMProfileLocation` (page 362) data type to specify a profile's location:

```
struct CMProfileLocation {
    short        locType;   /* specifies the location type */
    CMProfLoc    u;         /* structure for specified type */
};
```

The `CMProfileLocation` structure contains the `u` field of type `CMProfLoc` (page 361). The `CMProfLoc` type is a union that can provide access to any of the structures `CMFileLocation` (page 363), `CMHandleLocation` (page 363), `CMPtrLocation` (page 364), or `CMProcedureLocation` (page 364).

The data you specify in the `u` field indicates the actual location of the profile. In most cases, a ColorSync profile is stored in a disk file and you use the union for a file specification. However, a profile can also be located in memory, or in an arbitrary location (such as a resource) that is accessed through a procedure provided by your application. For more information on profile access, see "Accessing a Resource-Based Profile With a Procedure" (page 149). In addition, you can specify that a profile is temporary, meaning that it will not persist in memory after your application uses it for a color session.

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

```
enum {
    cmNoProfileBase         = 0,    /* the profile is temporary */
    cmFileBasedProfile      = 1,    /* file-based profile */
    cmHandleBasedProfile    = 2,    /* handle-based profile */
    cmPtrBasedProfile       = 3     /* pointer-based profile */
    cmProcedureBasedProfile = 4     /* procedure-based profile */
};
```

For example, for a file-based profile, the `u` field would hold a file specification and the `locType` field would hold the constant `cmFileBasedProfile`. Your application passes a `CMProfileLocation` structure when it calls the `CMOpenProfile` (page 222) function and the function returns a reference to the specified profile.

Note
If you already have a profile reference for a profile, you can call the `NCMGetProfileLocation` (page 233) function (available starting in ColorSync 2.5) or the `CMGetProfileLocation` (page 234) function (for previous versions of ColorSync) to obtain the location for the profile. ◆

Listing 3-2 shows an application-defined function, `MyOpenProfileFSSpec`, that assigns the file specification for a profile file to the `profLoc` union and identifies the location type as file-based. It then calls the `CMOpenProfile` function, passing to it the profile's file specification and receiving in return a reference to the profile.

Listing 3-2 Opening a reference to a file-based profile

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

    profLoc.u.fileLoc.spec = spec;
    profLoc.locType = cmFileBasedProfile;

    theErr = CMOpenProfile(prof, &profLoc);

    return theErr;
}
```

Reference Counts for Profile References

The ColorSync Manager keeps an internal reference count for each profile reference returned from a call to the `CMOpenProfile` (page 222), `CMCopyProfile` (page 229), `CMNewProfile` (page 227), or `CWNewLinkProfile` (page 267) functions. Calling the `CMCloneProfileRef` (page 231) function increments the count; calling the `CMCloseProfile` (page 223) function decrements it. When the count reaches 0, the ColorSync Manager releases all private memory, files, or resources allocated in association with that profile. The profile remains open as long as the reference count is greater than 0, indicating that at least one task retains a

reference to the profile. You can determine the current reference count for a profile reference by calling the `CMGetProfileRefCount` function.

When your application passes a copy of a profile reference to an independent task, whether synchronous or asynchronous, the task should call `CMCloneProfileRef` to increment the reference count. Both the called task and the caller should call `CMCloseProfile` when finished with the profile reference. This ensures that the tasks can finish independently of each other.

IMPORTANT

You call `CMCloneProfileRef` after copying a *profile reference* but not after duplicating an *entire profile* (as with the `CMCopyProfile` function). ▲

When your application passes a copy of a profile reference internally, it may not need to call `CMCloneProfileRef`, as long as the application calls `CMCloseProfile` once and only once for the profile.

IMPORTANT

In your application, make sure that `CMCloseProfile` is called once for each time a profile reference is created or cloned. Otherwise, the private memory and resources associated with the profile reference may not be properly freed, or a task may attempt to use a profile reference that is no longer valid. ▲

Poor Man's Exception Handling

Listing 3-3 shows a macro definition that is used in several subsequent code listings. In this macro, if `assertion` evaluates to `false`, execution continues at the location `exception`. Otherwise, execution continues at the next statement following the macro.

Listing 3-3 Poor man's exception handling macro

```
// Equivalent to  if ((assertion) == false) goto exception;
#define require(assertion, exception)   \
    do {                                \
```

```
        if (assertion) ;                \
        else { goto exception; }        \
    } while (false)
```

You can find examples of how to use this macro in Listing 3-4 (page 100), Listing 3-5 (page 101) and others. While this style of "poor man's exception handling" may not appeal to all developers, it does offer these advantages:

- It improves readability because it avoids pushing code off the page with multiple nested "if then else" clauses and by limiting the number of `return` statements in the code.
- You can enhance the macro to provide debug messages that supply useful runtime information about the error or where it occurred.

Identifying the Current System Profile

For the functions `NCMBeginMatching` (page 285), `NCMUseProfileComment` (page 290), and `NCWNewColorWorld` (page 262), your application can specify `NULL` to signify the system profile. For all other functions—for example, the `CMGetProfileElement` function, the `CMValidateProfile` function, and the `CMCopyProfile` function—for which you want to specify the system profile, you must give an explicit reference to the profile. You can use the `CMGetSystemProfile` (page 294) function to obtain a reference to the system profile.

IMPORTANT

Starting with ColorSync version 2.5, the system profile is used primarily for backward compatibility, as described in "Setting Default Profiles" (page 54). As a result, you should not use the system profile as a source or destination profile if you can determine a specific profile to use instead. For example, you may want to call `CMGetDefaultProfileBySpace` (page 297) to get the default profile for a specific color space or `CMGetProfileByAVID` (page 300) to get a profile for a specific display. ▲

Each profile, including the profile configured as the system profile, has a name associated with it. If your application needs to display the name of the system profile to the user, it can call `CMGetSystemProfile`, as shown in Listing 3-4, to get the system profile, then call the `CMGetScriptProfileDescription` (page 256) function to get the profile name and script code.

Listing 3-4 Identifying the current system profile

```
CMError MyPrintSystemProfileName (void)
{
    CMError         theErr;
    CMProfileRef    sysProf;
    Str255          profName;
    ScriptCode      profScript;

    theErr = CMGetSystemProfile(&sysProf);
    require(theErr == noErr, cleanup);

    theErr = CMGetScriptProfileDescription(sysProf, profName,
                                           &profScript);
    require(theErr == noErr, cleanup);

    // … call Script Mgr to get correct font for script …

    DrawString(profname);

// Do any necessary cleanup. In this case, just return.
cleanup:
    return theErr;
}
```

Getting the Profile for the Main Display

Starting with ColorSync version 2.5, a user can select a separate profile for each display, as described in “Setting a Profile for Each Monitor” (page 69). In your code, you can determine the profile for any display for which you know the AVID by calling the function `CMGetProfileByAVID` (page 300), which is also new in version 2.5. You can get more information about AVID values from the Display Manager SDK.

Listing 3-5 shows how to get the profile for the main display (the one that contains the menu bar).

Listing 3-5 Getting the profile for the main display

```
CMError GetProfileForMainDisplay (CMProfileRef *prof)
{
    CMError     theErr;
    AVIDType    theAVID;
    GDHandle    theDevice;

    // Get the main GDevice.
    theDevice = GetMainDevice();

    // Get the AVID for that device.
    theErr = DMGetDisplayIDByGDevice(theDevice, &theAVID, true);
    require(theErr == noErr, cleanup);

    // Get the profile for that AVID.
    theErr = GetProfileByAVID(theAVID, prof);
    require(theErr == noErr, cleanup);

// Do any necessary cleanup. In this case, just return.
cleanup:
    return theErr;
}
```

This code first gets a graphic device handle for the main display, then calls the Display Manager routine `DMGetDisplayIDByGDevice` to get an AVID for the device. It then passes the AVID to the ColorSync Manager routine `CMGetProfileByAVID` (page 300) to get a profile reference to the profile for the display.

Matching to Displays Using QuickDraw-Specific Operations

To provide images and pictures showing consistent colors across displays, your application can use ColorSync to match the colors in a user's pictures and documents with the colors available on the user's current display. If a color cannot be reproduced on the system's current display, ColorSync maps the color to the color gamut of the display according to the specifications defined by the profiles. "When Color Matching Occurs" (page 62) describes both QuickDraw-specific and general purpose ColorSync functions for color matching.

The ColorSync Manager provides two QuickDraw-specific functions that your application can call to draw a color picture to the current display. The function `NCMDrawMatchedPicture` (page 288) matches the picture's colors to the display's gamut defined by the specified display profile. It uses the system profile as the initial source profile but switches to any embedded profiles as they are encountered. The function `NCMBeginMatching` (page 285) uses the source and destination profiles you specify to match the colors of the source image to the colors of the device for which it is destined.

The current display device's profile is typically configured as the system profile. A user can do this with the ColorSync control panel. However, starting with ColorSync 2.5, a user can use the Monitors & Sound control panel to set a separate profile for each display, as described in "Setting a Profile for Each Monitor" (page 69). When a user sets a profile for a display, ColorSync makes that profile the current default system profile.

Because the ColorSync Manager assumes the system profile is that of the current display, you can pass a value of `NULL` to the QuickDraw-specific functions instead of supplying an explicit profile reference. Passing `NULL` for a profile reference directs the ColorSync Manager to use the system profile. Note however, that starting with ColorSync 2.5, if you know the primary display for the image, and you know the AVID for that display, you can call `CMGetProfileByAVID` (page 300) to get the profile for the specific display. For example, Listing 3-5 shows how to get the profile for the main display (the one with the menu bar).

The following sections describe how to use ColorSync's QuickDraw-specific matching functions, which automatically perform color matching in a manner acceptable to most applications. However, if your application needs a finer level of control over color matching than is supplied by the QuickDraw-specific functions, you can use the general purpose functions described in"Matching Colors Using the General Purpose Functions" (page 107) to match the colors of a bitmap, a pixel map, or a list of colors.

Matching Colors in a Picture Containing an Embedded Information

If a user copies a picture that includes a profile or profile identifier into one of your application's documents, your application can use the ColorSync Manager's QuickDraw-specific function `NCMDrawMatchedPicture` (page 288) to match the colors in that picture to the display on which you draw it.

As the picture is drawn, the `NCMDrawMatchedPicture` function automatically matches all colors to the color gamut of the display device, using the

destination profile passed in the `dst` parameter. To use this function, you need to supply only the profile for the destination display device. The function acknowledges color-matching picture comments embedded in the picture and uses embedded profiles and profile identifiers. The source profile for the device on which the image was created should be embedded in the QuickDraw picture whose handle you pass to the function; the `NCMDrawMatchedPicture` function uses the embedded source profile, if it exists. If the source profile is not embedded, the function uses the current system profile as the source profile.

A picture may have more than one profile embedded, and may embed profile identifiers that refer to, and possibly modify, embedded profiles or profiles on disk. If the profiles and profile identifiers are embedded correctly, the `NCMDrawMatchedPicture` function will use them successively, as they are encountered.

By specifying `NULL` as the destination profile when you use this function, you are assured that the system profile—typically set to the profile for the main screen—is used as the destination profile. Alternatively, your application can call the `CMGetSystemProfile` (page 294) function to obtain a reference to the profile and specify the system profile explicitly. Or, starting in ColorSync version 2.5, if you know the AVID for the display on which drawing takes place, you can call `CMGetProfileByAVID` (page 300) to get the profile for the display.

Listing 3-6 shows sample code that uses the QuickDraw-specific function `NCMDrawMatchedPicture` to perform color matching to a display. The code gets a profile for the destination display using an AVID if it is available; otherwise, it passes `NULL` to the `NCMDrawMatchedPicture` function to specify the system profile.

Listing 3-6 Matching a picture to a display

```
// Matching a picture to a display
CMError MyDrawPictureToADisplay (PicHandle thePict, AVIDType theAVID, Rect *destRect)
{
    CMError         theErr;
    CMProfileRef    destProf;

    // Init for error handling.
    theErr = noErr;
    destProf = NULL;

    // If caller supplied an AVID and CS 2.5 is running...
```

```
    if (theAVID && ColorSync25Available() ) // See Listing 3-1 (page 92).
    {
        theErr = GetProfileByAVID(theAVID, &destProf);
        require(theErr == noErr, cleanup);
    }
    else
    {
        // Use the System profile as the destination.
        destProf = NULL;
    }
    // Draw the picture, with color matching.
    NCMDrawMatchedPicture(thePict, destProf, destRect);
    theErr = QDError();
    require(theErr == noErr, cleanup);

// Do any necessary cleanup. If necessary, close the profile.
cleanup:

    if (destProf)
        CMCloseProfile(destProf);

    return theErr;
}
```

More on Embedded Information

For embedded profiles (and profile identifiers) to operate 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 encountered 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.

Note
Profile identifiers are also stored with picture comments. For more information on profile identifiers, see "Embedding Profiles and Profile Identifiers" (page 112) and "Searching for a Profile That Matches a Profile Identifier" (page 139).

If your application allows a user to modify and save an image that you color matched using the function `NCMUseProfileComment` (page 290), your application should either embed the destination profile in the picture file or convert and match the colors of the modified image to the colors of the source profile. By doing this your application ensures the integrity of the image during future operations and display. The method you choose is specific to your application.

Matching Colors as a User Draws a Picture

To use Color QuickDraw functions to draw a document with colors matched to a display, your application can simply use the `NCMBeginMatching` (page 285) function before calling Color QuickDraw functions, then conclude its drawing with the `CMEndMatching` function. For example, you might want to do this to customize settings in the profile that affect the matching operation. For more information on Color QuickDraw drawing functions, see *Inside Macintosh: Imaging With QuickDraw*.

To use the `NCMBeginMatching` function, you must specify both the source and destination profiles. The `NCMBeginMatching` (page 285) function returns a reference to the color-matching session in its `myRef` parameter. You then pass the reference to the `CMEndMatching` (page 287) function to terminate color matching. Code for performing this operation is not shown here.

Creating a Color World to Use With the General Purpose Functions

A color world is a reference to a private ColorSync structure that represents a unique color-matching session. Although profiles can be large, a color world is a compact representation of the mapping needed to match between profiles. Conceptually, you can think of a color world as a sort of "matrix multiplication" of two or more profiles that distills all the information contained in the profiles into a fast, multidimensional lookup table.

For the ColorSync Manager general purpose functions, a color world characterizes 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. “When Color Matching Occurs” (page 62) describes both general purpose and QuickDraw-specific ColorSync functions for color matching. 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 between a series of concatenated profiles.

For the general purpose ColorSync Manager functions, a color world is the equivalent of the ColorSync Manager QuickDraw-based functions’ source and destination profiles. From your application’s perspective, the difference in specifying profiles for the general purpose 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. This general purpose interface provides better performance during color-matching.

Your application calls the `NCWNewColorWorld` (page 262) function to set up a simple color world for color transformations involving two profiles—a source profile and a destination profile—and the function returns a reference to the color world it creates. Setting up a color world for color processing involving a series of concatenated profiles or a single device link profile, which contains a series of profiles, is slightly more complex. Here are the steps you take:

1. **Obtain references to the profiles to use for the concatenated color world.**

   For information describing how to obtain references to the profiles for the color world, see “Obtaining Profile References” (page 95).

2. **Set up an array containing references to the profiles comprising the set.**

   Before your application calls the function `CWConcatColorWorld` (page 265) to create the color world, you must establish the profile set. The ColorSync Manager defines the following data structure of type `CMConcatProfileSet` that you use to specify the profile set:

   ```
   struct CMConcatProfileSet {
       unsigned short      keyIndex;
       unsigned short      count;
       CMProfileRef        profileSet[1];
   };
   ```

   Your application also uses the `CMConcatProfileSet` data structure to define a profile set for a device link profile. See “Creating and Using Device Link Profiles” (page 143) for more information.

Your application creates an array that contains references to the profiles for the color world, specifying these references in processing order. You specify the one-based number of profile references in the array by setting the value of the `CMConcatProfileSet.count` field. You assign the profile array to the `CMConcatProfileSet.profileSet` field.

The ColorSync Manager defines rules governing the types of profiles you can specify in a profile array. These rules differ depending on whether you are creating a profile set to create a device link profile or to create a concatenated color world. For a list of the rules defining the types of profiles you can use for these purposes, see `CWNewLinkProfile` (page 267) and `CWConcatColorWorld` (page 265).

3. **Identify the CMM for color processing.**

   Each of the profiles whose references you give identifies a CMM for color processing involving that profile. To perform color transformation using a series of profiles, the ColorSync Manager uses only one CMM. You use the `CMConcatProfileSet.keyIndex` field to identify the index into the array corresponding to the profile whose specified CMM is to be used. The array is zero based, so you must specify the `CMConcatProfileSet.keyIndex` value as a number in the range of 0 to `count` – 1, where `count` is the number of elements in the array.

IMPORTANT

See "How the ColorSync Manager Selects a CMM" (page 84) for a complete description of the ColorSync algorithm for selecting a CMM. ▲

4. **Call the `CWConcatColorWorld` function to set up the color world.**

   You pass the `CWConcatColorWorld` function a parameter of type `CMConcatProfileSet` to specify the profile array, and the function returns a color world reference. To perform color matching or gamut checking using the profiles comprising a color world, you call the general purpose function passing it the reference to the color world.

Using a device link profile for the general purpose functions entails additional steps, described in "Creating and Using Device Link Profiles" (page 143).

Matching Colors Using the General Purpose Functions

"When Color Matching Occurs" (page 62) describes both general purpose and QuickDraw-specific ColorSync functions for color matching. Using the general

purpose functions `CWMatchPixMap` (page 272) or `CWMatchBitmap` (page 276), your application can match the colors of a pixel image or a bitmap image to the display's color gamut without relying on QuickDraw.

Color matching occurs relatively quickly, but for a session involving a large pixel image or bitmap image, the color-matching process may take some time. To keep the user informed, you can provide a progress-reporting function. For example, your function can display an indicator, such as a progress bar, to depict how much of the matching has been done and how much remains. Your function can also allow the user to interrupt the color-matching process.

When your application calls either the `CWMatchPixMap` function or the `CWMatchBitmap` function, you can pass the function a pointer to your callback progress-reporting function and a reference constant containing data, such as the progress bar dialog box's window reference. When the CMM used to match the colors calls your progress-reporting function, it passes the reference constant to it. If you provide a progress-reporting function, here is how you should declare the function, assuming you name it `MyCMBitmapCallBackProc`:

```
pascal Boolean MyCMBitmapCallBackProc (long progress, void *refCon);
```

For a complete description of the progress-reporting function declaration, see `MyCMBitmapCallBackProc` (page 345).

To use the `CWMatchPixMap` (page 272) and `CWMatchBitmap` (page 276) functions, your application must first set up a color world that specifies the profiles involved in the color-matching session as described in "Creating a Color World to Use With the General Purpose Functions" (page 105). The color world establishes how matching will take place between the profiles. Listing 3-7 shows how to match the colors of a bitmap using the general purpose functions that take a color world.

The ColorSync Manager uses the `PixMap` data type defined by Color QuickDraw. The ColorSync Manager defines and uses the `cmBitmap` data type, based on the classic QuickDraw `Bitmap` data type.

Matching the Colors of a Pixel Map to the Display's Color Gamut

Your application can call the function `CWMatchPixMap` (page 272) to match the colors of a pixel image to the display's color gamut. To use `CWMatchPixMap`, you first create a color world, as described in "Creating a Color World to Use With the General Purpose Functions" (page 105). The color world is based on the

source profile for the device used to create the pixel image and the destination profile for the display on which the image is shown.

To match the colors of a pixel image to the display's color gamut, the source profile for the color world must specify a data color space of RGB as its `dataColorSpace` element value to correspond to the pixel map data type, which is implicitly RGB. If the source profile you specify for the color world is the original source profile used to create the pixel image, most likely these values match. However, if you want to verify that the source profile's `dataColorSpace` element specifies RGB, you can use the `CMGetProfileHeader` (page 245) function to obtain the profile header. The profile header contains the `dataColorSpace` element field. For a pixel image, the display profile's `dataColorSpace` element must also be set to RGB; this is the color space commonly used for displays.

If the source profile is embedded in the document containing the pixel map, your application can extract the profile and open a reference to it before you create the color world. For information on how to extract an embedded profile, see "Extracting Profiles Embedded in Pictures" (page 118). If the source profile is installed in the ColorSync Profiles folder, your application can display a list of profiles to the user to allow the user to select the appropriate one.

Matching the Colors of a Bitmap Image to the Display's Color Gamut

Matching the colors of a bitmap image to the current system's display is similar to the process of matching a pixel map's colors, except that the data type of a bitmap image is explicitly stated in the `space` field of the bitmap. You can specify a bitmap image using any of the following data types: `cmGraySpace`, `cmGrayASpace`, `cmRGB16Space`, `cmRGB24Space`, `cmRGB32Space`, `cmARGB32Space`, `cmRGB48Space`, `cmCMYK32Space`, `cmCMYK64Space`, `cmHSV32Space`, `cmHLS32Space`, `cmYXY32Space`, `cmXYZ32Space`, `cmLUV32Space`, `cmLAB24Space`, `cmLab32Space`, `cmLAB48Space`, `cmNamedIndexed32Space`, `cmMCFive8Space`, `cmMCSix8Space`, `cmMCSeven8Space`, or `cmMCEight8Space`. The data type of the source bitmap image must correspond to the data color space specified by the color world's source profile.

When you call the `CWMatchBitmap` (page 276) function, you can pass it a pointer to a bitmap to hold the resulting image. In this case, you must allocate the pixel buffer pointed to by the `image` field of the `CMBitmap` structure. Because the `CWMatchBitmap` function allows you to specify a separate bitmap to hold the resulting color-matched image, you must ensure that the data type you specify in the `space` field of the resulting bitmap matches the destination's color data space. On input, the color space of the source profile must match the color space

of the bitmap. If you specify `NULL` for the destination bitmap, on successful output, ColorSync will change the `space` field of the source bitmap to reflect the bitmap space to which the source image was mapped.

Rather than create a bitmap for the color-matched image, you can match the bitmap in place. To do so, you specify `NULL` instead of passing a pointer to a resulting bitmap.

The code in Listing 3-7 shows how to set up a bitmap for the resulting color-matched image before calling the `CWMatchBitmap` function to perform the color matching. The `MyMatchImage` function calls the `MyGetImageProfile` function (not shown) to obtain an embedded profile from the image. If none is found, it calls the `MyGetImageSpace` function (also not shown) to determine the color space for the profile, then calls the ColorSync routine `CMGetDefaultProfileBySpace` (page 297) to obtain the default profile for that space.

The `MyMatchImage` function then calls `GetProfileForMainDisplay`, shown in Listing 3-5, to get the destination profile. It uses the source and destination profiles to set up a color world by calling `NCWNewColorWorld` (page 262), then uses the resulting color world when it calls `CWMatchBitmap` (page 276) to match the colors to the display.

Listing 3-7 Matching the colors of a bitmap using a color world

```
void MyMatchImage (FSSpec theImage)
{
    CMError         theErr;
    CMProfileRef    sourceProf;
    CMProfileRef    destProf;
    CMWorldRef      cw;
    CMBitmap        bitmap;
    OSType          theSpace;

    /* Init for error handling. If any error during process,
        jump to cleanup area and quit trying. */
    theErr = noErr;
    sourceProf = nil;
    destProf = nil;
    cw = nil;
    bitmap.image = nil;
```

```
    // Determine source profile.
    // 1st - try to find an embedded profile
    theErr = MyGetImageProfile(theImage, &sourceProf);
    if (theErr == noErr)
    {
        // 2nd - use default profile for the image space
        theErr = MyGetImageSpace(theSpace, &sourceProf);
        require(theErr == noErr, cleanup);

        theErr = CMGetDefaultProfileBySpace(theSpace, &sourceProf);
        require(theErr == noErr, cleanup);
    }
    require(theErr == noErr, cleanup);

    // Determine dest profile.
    theErr = GetProfileForMainDisplay(&destProf);
    require(theErr == noErr, cleanup);

    // Set up a color world.
    theErr = NCWNewColorWorld(&cw, sourceProf, destProf);
    require(theErr == noErr, cleanup);

    // close profiles after setting up color world.
    if (sourceProf)
        CMCloseProfile(sourceProf);
    if (destProf)
        CMCloseProfile(destProf);
    sourceProf = destProf = nil;

    // Read the image into the CMBitmap structure
    theErr = MyGetImageBitmap(theImage, &bitmap);
    require(theErr == noErr, cleanup);

    // Match bitmap in place.
    theErr = CWMatchBitmap(cw, &bitmap, nil, nil, nil);
    require(theErr == noErr, cleanup);

    // Render results here ... (code not shown)

/* Do any necessary cleanup:close profiles and dispose of
    color world and bitmap. */
```

```
cleanup:

    if (sourceProf)
        CMCloseProfile(sourceProf);
    if (destProf)
        CMCloseProfile(destProf);
    if (cw)
        CWDisposeColorWorld(cw);
    if (bitmap.image)
        DisposePtr(bitmap.image);

    return theErr;
}
```

Embedding Profiles and Profile Identifiers

When the user creates and saves a document or picture containing a color image created or modified with your application, your application can provide for future color matching by saving—along with that document or picture—the profile for the device on which the image was created or modified. In addition to a profile—or instead of a profile—your application can save a profile identifier. A profile identifier is an abbreviated data structure that identifies, and possibly modifies, a profile in memory or on disk.

When embedding source profiles or profile identifiers in the documents created by your application, you can store them in any manner that you choose. For example, you may choose to have your application store, in the resource fork of the document file, one profile for an entire image, or a separate profile for every object in an image, or a separate profile identifier that points to a profile on disk for every device on which the user modified the image.

When embedding source profiles or profile identifiers in PICT file pictures, your application should use the `cmComment` picture comment, which has a `kind` value of 224 and is defined for embedded version 2.x profiles. This comment is

followed by a 4-byte selector that describes the type of data in the comment. The following selectors are currently defined:

Selector	Value	Description
`cmBeginProfileSel`	0	Beginning of a version 2.x profile. Profile data to follow.
`cmContinueProfileSel`	1	Continuation of version 2.x profile data. Profile data to follow.
`cmEndProfileSel`	2	End of version 2.x profile data. No profile data follows.
`cmProfileIdentifierSel`	3	Profile identifier follows. A profile identifier identifies a profile that may reside in memory or on disk.

Because the `dataSize` parameter of the `PicComment` procedure is a signed 16-bit value, the maximum amount of profile data that can be embedded in a single picture comment is 32,763 bytes (32,767 – 4 bytes for the selector).

You can embed a larger profile by using multiple picture comments of selector type `cmContinueProfileSel`, as shown in Figure 3-7. You must embed the profile data in consecutive order, and you must conclude the profile data by embedding a picture comment of selector type `cmEndProfileSel`. The ColorSync Manager provides the `NCMUseProfileComment` function to automate the process of embedding profile information.

Embedded Profile Format

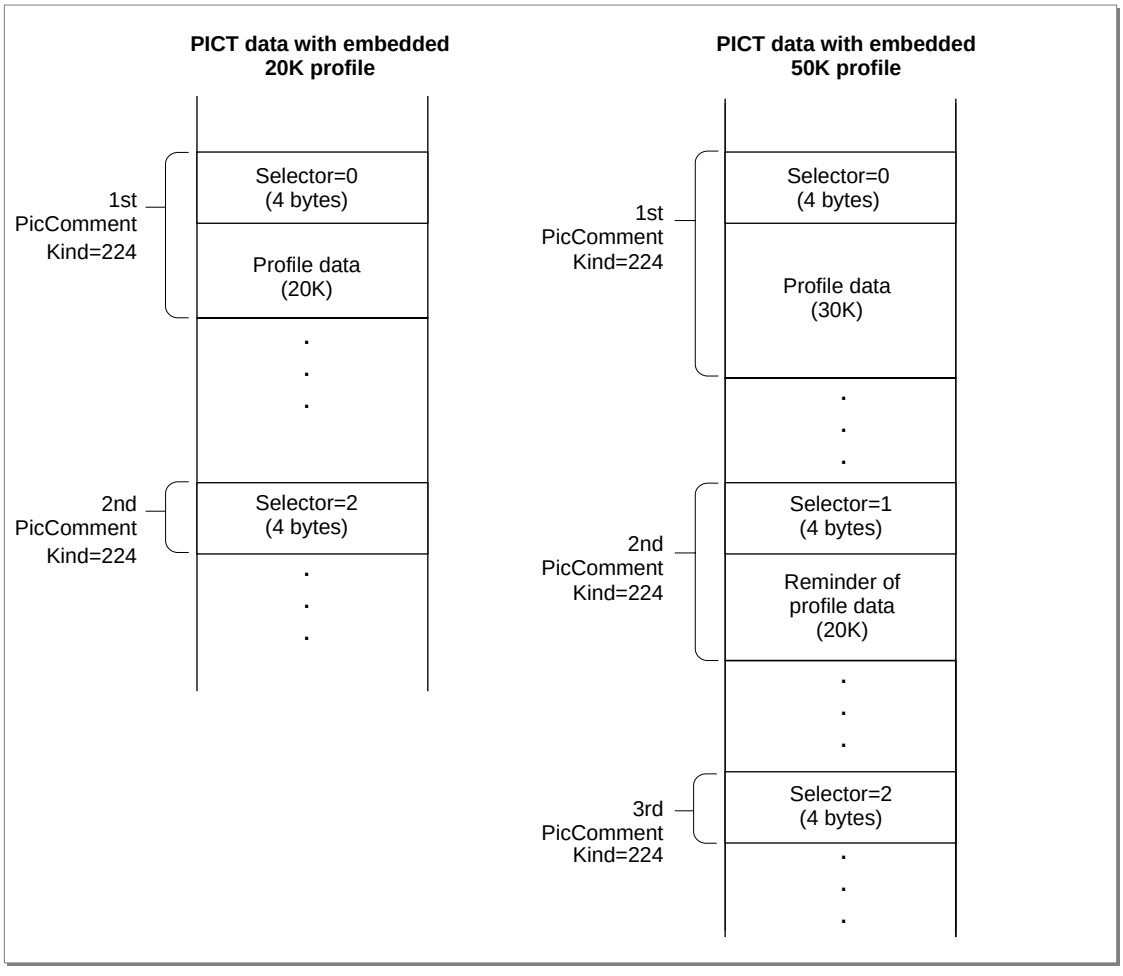
Figure 3-7 shows how profile data is embedded in a PICT file picture as a series of picture comments. The illustration shows two embedded profiles. The first profile contains less than 20K of data, so its data can be stored in one picture comment with selector type `cmBeginProfileSel`. Note, however, that a second comment of selector type `cmEndProfileSel`, containing no data, concludes the embedded profile.

The second embedded profile shown in Figure 3-7 has more than 32K of data, so its data must be stored in two consecutive picture comments. The first comment has selector type `cmBeginProfileSel`, while the second has type `cmContinueProfileSel`. If the profile were larger and required additional picture comments, each additional comment would have selector type `cmContinueProfileSel`. As with all embedded profiles, the final picture comment has selector type `cmEndProfileSel`.

Embedding Different Profile Versions

For version 1.0 of the ColorSync Manager, you use picture comment types `cmBeginProfile` and `cmEndProfile` to begin and end a picture comment. The `cmBeginProfile` comment is not supported for ColorSync version 2.x profiles; however, the you can use the `cmEndProfile` comment to end the current profile for both ColorSync 1.0 and 2.x. Following a `cmEndProfile` comment, the ColorSync Manager reverts to the system profile. You use the `cmEnableMatching` and `cmDisableMatching` picture comments to begin and end color matching in both ColorSync 1.0 and 2.x. See *Inside Macintosh: Imaging With QuickDraw* for more information about picture comments.

Figure 3-7 Embedding profile data in a PICT file picture



The NCMUseProfileComment Function

The ColorSync Manager provides the function `NCMUseProfileComment` (page 290) to automate the process of embedding a profile or profile identifier. This function generates the picture comments required to embed the specified profile or identifier into the open picture. It calls the QuickDraw `PicComment` function

with a picture comment `kind` value of `cmComment` and a 4-byte selector that describes the type of data in the picture comment: 0 to begin the profile, 1 to continue, and 2 to end the profile; or 3 for a profile identifier. For a 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.

For embedded profiles or profile identifiers to work correctly, the currently effective profile must be terminated by a picture comment of `kind cmEndProfile` after drawing operations using that profile are performed. If you do not specify a picture comment to end the profile, the profile will remain in effect until the next embedded profile is introduced with a picture comment of `kind cmBeginProfile`. It is good practice to always pair use of the `cmBeginProfile` and `cmEndProfile` picture comments. When the ColorSync Manager encounters a `cmEndProfile` picture comment, it restores use of the system profile for matching until it encounters another `cmBeginProfile` picture comment.

IMPORTANT

The `NCMUseProfileComment` function does not automatically terminate the embedded profile or profile identifier with a `cmEndProfile` picture comment. You must add a picture comment of `kind cmEndProfile` when the drawing operations to which the profile applies are complete. Otherwise, the profile remains in effect until the next embedded profile with a picture comment of `kind cmBeginProfile` is encountered. ▲

In addition to embedded profiles, an image may contain embedded profile identifiers, which are stored with the selector `cmProfileIdentifierSel`. For more information on profile identifiers, see “Searching for a Profile That Matches a Profile Identifier” (page 139), and `CMProfileIdentifier` (page 359).

Listing 3-8 shows how to embed a profile in a picture file. The `MyPreprendProfileToPicHandle` function creates a new picture, embeds the profile for the device used to create the picture, then draws the picture. The caller passes a reference for the profile as the `prof` parameter. Note that after `MyPreprendProfileToPicHandle` calls the `NCMUseProfileComment` function to embed the profile, it calls its own `MyEndProfileComment` function to embed a comment of `kind cmEndProfile`, ensuring that the profile is properly terminated.

Listing 3-8 Embedding a profile by prepending it before its associated picture

```
CMError MyPrependProfileToPicHandle (
                PicHandle pict,
                PicHandle *pictNew,
                CMProfileRef prof,
                Boolean embedAsIdentifier)
{
    OSErr           theErr;
    CGrafPtr        savePort;
    GDHandle        saveGDev;
    GWorldPtr       tempWorld;
    Rect            pictRect;
    unsigned long   flags;

    // Init for error handling.
    theErr = noErr;
    tempWorld = nil;

    // Check parameters
    if (prof == nil) theErr = paramErr;
    require(theErr == noErr, cleanup);

    // Determine whether to embed as identifier or whole profile.
    if (embedAsIdentifier)
        flags = cmEmbedProfileIdentifier;
    else
        flags = cmEmbedWholeProfile;

    // Create a temporary graphics world.
    theErr = NewSmallGWorld(&tempWorld);
    require(theErr == noErr, cleanup);

    // Save current world and switch to temporary.
    GetGWorld(&savePort, &saveGDev);
    SetGWorld(tempWorld, nil);
    pictRect = (**pict).picFrame;
    ClipRect(&pictRect);                // Important: set clipRgn.

    // Create a new picture.
    *pictNew = OpenPicture(&pictRect);  // Start recording.
```

```
    theErr = NCMUseProfileComment(prof,flags);
    DrawPicture(pict, &pictRect);
    MyEndProfileComment();              // Routine shown below.
    ClosePicture();

    if (theErr)
        KillPicture(*pictNew);

    SetGWorld(savePort, saveGDev);

// Do any necessary cleanup:dispose of graphics world.
cleanup:

    if (tempWorld)
        DisposeGWorld(tempWorld);

    return theErr;
}
```

Here is the application-defined `MyEndProfileComment` function called by `MyPrependProfileToPicHandle` to add the `cmEndProfile` picture comment to terminate the profile:

```
void MyEndProfileComment (void)
{
    PicComment(cmEndProfile, 0, 0);
}
```

Extracting Profiles Embedded in Pictures

To color match or gamut check a picture embedded in a document, your application should first check for embedded profiles in the document. If a profile is found, your application can then open a reference to the profile and use it as the source profile. This process requires you to locate and identify the profile for the image within the document and extract the profile data from the document file.

Note
If you use the QuickDraw-specific `NCMDrawMatchedPicture` (page 288) function, you do not need to extract the source profile from the PICT file. ◆

To extract an embedded profile, your application can use the function `CMUnflattenProfile` (page 239). This function takes a pointer to a low-level data-transfer function that your application supplies to transfer the profile data from the document containing it. This function assumes that your low-level data-transfer function is informed about the context of the profile. After all of the profile data has been transferred, the `CMUnflattenProfile` function returns the file specification for the profile.

Prior to ColorSync 2.5, when your application calls the `CMUnflattenProfile` function, the ColorSync Manager uses the Component Manager to pass the pointer to your low-level data-transfer function along with the reference constant your application can use as it desires. The CMM is determined by the selection process described in "How the ColorSync Manager Selects a CMM" (page 84). The CMM calls your low-level data-transfer function, directing it to open the file containing the profile, read segments of the profile data, and return the data to the CMM's calling function.

The CMM communicates with your low-level data transfer-function using a command parameter to identify the operation to perform. To facilitate the transfer of profile data from the file to the CMM, the CMM passes to your function a pointer to a data buffer for data, the size in bytes of the profile data your function should return, and the reference constant passed from the calling application.

On return, your function passes to the CMM segments of the profile data and the number of bytes of profile data you actually return.

Starting with ColorSync 2.5, the ColorSync Manager calls your transfer function directly, without going through the preferred, or any, CMM. On return from `CMUnflattenProfile`, the value of `preferredCMMnotfound` is guaranteed to be `false`.

Listing 3-9 and Listing 3-10 show portions of a sample application called CSDemo, available as part of the ColorSync SDK. You can find the complete sample application on the Developer CD series, or at the web site <http://developer.apple.com/sdk>.

In these listings, all variables beginning with a lowercase letter "g" are global variables previously defined. The application uses global variables to pass data

between functions that do not include reference constant parameters. Listing 3-9 counts the profiles in a PICT file, while Listing 3-10 extracts a profile, identified by an index number, from a PICT file.

Counting the Profiles in the PICT File

Given a `picHandle` value to a picture containing an embedded profile, the sample code shown in Listing 3-9 counts the number of profiles in the picture.

The `MyCountProfilesInPicHandle` function calls the Toolbox function `SetStdCProcs` to get the current QuickDraw drawing bottleneck procedures, then sets the bottlenecks to its own routines. It initializes its global counter, `gCount`, which holds a single count summing both ColorSync 1.0 profiles and version 2.x profiles, to zero. The `MyCountProfilesInPicHandle` function calls its own drawing function, `MyDrawPicHandleUsingBottleneck`, not shown here, to draw the picture. The drawing function sets up a port that uses the private bottleneck routines.

As the picture is drawn, the `MyCountProfilesCommentProc` bottleneck procedure counts the number of profiles encountered. `MyCountProfilesCommentProc` checks for both version 1.0 profiles and version 2.x profiles and increments the global count when it finds either type. You can easily modify this code to keep separate counts if necessary.

`MyCountProfilesInPicHandle` doesn't use any other QuickDraw bottlenecks, so it uses nonoperational routines (routines that do nothing but return) for all other bottlenecks. The prototype for a function to handle the `TextProc` bottleneck, for example, can be defined as follows:

```
static pascal void MyNoOpTextProc ( short byteCount,
                                    Ptr textAddr,
                                    Point numer,
                                    Point denom);
```

For a general discussion of customizing QuickDraw's bottleneck routines, see "Customizing QuickDraw's Text Handling" in *Inside Macintosh: Text*.

Listing 3-9 Counting the number of profiles in a picture

```
CMError MyCountProfilesInPicHandle (PicHandle pict, unsigned long *count)
{
    OSErr       theErr = noErr;
    CQDProcs    procs;

    /* Set up bottleneck for picComments so we can count the profiles. */
    SetStdCProcs(&procs);
    procs.textProc    = NewQDTextProc (MyNoOpTextProc);
    procs.lineProc    = NewQDLineProc (MyNoOpLineProc);
    procs.rectProc    = NewQDRectProc (MyNoOpRectProc);
    procs.rRectProc   = NewQDRRectProc (MyNoOpRRectProc);
    procs.ovalProc    = NewQDOvalProc (MyNoOpOvalProc);
    procs.arcProc     = NewQDArcProc (MyNoOpArcProc);
    procs.polyProc    = NewQDPolyProc (MyNoOpPolyProc);
    procs.rgnProc     = NewQDRgnProc (MyNoOpRgnProc);
    procs.bitsProc    = NewQDBitsProc (MyNoOpBitsProc);
    procs.commentProc = NewQDCommentProc(MyCountProfilesCommentProc);
    procs.txMeasProc  = NewQDTxMeasProc (MyNoOpTxMeasProc);

/* Initialize the global counter to be incremented by the commentProc. */
    gCount = 0;

/* Draw the picture and count the profiles while drawing. */
    theErr = MyDrawPicHandleUsingBottlenecks (pict, procs, nil);

/* Obtain the result from the count global variable. */
    *count = gCount;

/* Clean up and return. */
    DisposeRoutineDescriptor(procs.textProc);
    DisposeRoutineDescriptor(proc.lineProc);
    DisposeRoutineDescriptor(procs.rectProc);
    DisposeRoutineDescriptor(procs.rRectProc);
    DisposeRoutineDescriptor(procs.ovalProc);
    DisposeRoutineDescriptor(procs.arcProc);
    DisposeRoutineDescriptor(procs.polyProc);
    DisposeRoutineDescriptor(procs.rgnProc);
    DisposeRoutineDescriptor(procs.bitsProc);
    DisposeRoutineDescriptor(procs.commentProc);
```

```
    DisposeRoutineDescriptor(procs.txMeasProc);
}

pascal void MyCountProfilesCommentProc (short kind,
                                        short dataSize,
                                        Handle dataHandle)
{
    long    selector;

    switch (kind)
    {
        case cmBeginProfile
            gCount ++;  // We found a ColorSync 1.0 profile; increment the count.
            break;

        case cmComment;
            // Break if dataSize is too small to be a selector.
            if (dataSize <= 4) break;

            // Since dataSize is >= 4, we can get a selector from the first long.
            selector = *((long *)(*dataHandle));
            if (selector == cmBeginProfileSel)
                gCount ++;  // We found a ColorSync 2.xprofile; increment the count.
            break;
    }
}
```

Extracting a Profile

Flattening refers to transferring a profile stored in an independent disk file to an external profile format that can be embedded in a graphics document. Unflattening refers to transferring from the embedded format to an independent disk file.

This part of the sample application identifies the profile to unflatten, unflattens the profile, creates a temporary profile, and disposes of the original. To perform these tasks, the code must again draw the picture using the bottleneck routines.

Part A: Calling the Unflatten Function

Listing 3-10 shows the `MyGetIndexedProfileFromPicHandle` entry point function that drives the process of unflattening the profile. The function creates a

universal procedure pointer (UPP), `MyflattenUPP`, that points to the low-level data-transfer procedure.

A PICT handle may contain more than one profile. To identify the profile to unflatten, the `MyGetIndexedProfileFromPicHandle` function contains an `index` parameter that specifies the profile's index. The function stores the index in the global variable `gIndex` so that the value is accessible by the application's other functions that check for the correct profile and extract it. Then, the function calls the `CMUnflattenProfile` function, passing it the `MyflattenUPP` pointer. This invokes the `MyUnflattenProc` function shown in Listing 3-11.

The function `MyGetIndexedProfileFromPicHandle`, shown in Listing 3-10, first calls `CMUnflattenProfile` (page 239) to create an independent file-based profile, then calls the function `CMOpenProfile` (page 222) to open a temporary profile reference to the file-based profile. It then calls `CMCopyProfile` (page 229) to create a copy of the profile reference. Finally, the function disposes of the original profile. The purpose for creating a temporary profile, copying it into the specified location, then deleting the temporary profile, is to adhere to the copyright protection for embedded profiles specified by the `flags` field in the profile header.

Listing 3-10 Calling the CMUnflattenProfile function to extract an embedded profile

```
CMError MyGetIndexedProfileFromPicHandle (PicHandle pict,
                                          unsigned long index,
                                          CMProfileRef *prof,
                                          CMProfileLocation *profLoc)
{
    CMError             theErr;
    unsigned long       refCon;
    CMFlattenUPP        myFlattenUPP;
    Boolean             preferredCMMNotFound;
    Boolean             tempCreated;
    FSSpec              tempSpec;
    CMProfileRef        tempProf;
    CMProfileLocation   tempProfLoc;

    // Init for error handling.
    theErr = noErr;
    tempCreated = false;
    tempProf = nil;
```

```
    // Create a universal procedure pointer for the
    // unflatten procedure shown in Listing 3-11.
    myFlattenUPP = NewCMFlattenProc(MyUnflattenProc);

    // Pass the pict as the refcon.
    refCon = (unsigned long) pict;

    // Set the global index variable to the index of the profile we're looking for.
    gIndex = index;

    // The next call invokes the MyUnflattenProc shown in Listing 3-11.
    //On return, tempSpec identifies the newly created profile on disk.
    theErr = CMUnflattenProfile(&tempSpec, myFlattenUPP,(void*)&refCon,
                            &preferredCMMNotFound);
    DisposeRoutineDescriptor(myFlattenUPP);// Dispose of the procedure pointer.
    require(theErr == noErr, cleanup);
    tempCreated = true;

    // Open the newly created profile, create a temporary profile reference for it,
    // copy the temporary reference, then close it and delete the profile file.
    tempProfLoc.locType = cmFileBasedProfile;
    tempProfLoc.u.fileLoc.spec = tempSpec;

    theErr = CMOpenProfile(&tempProf, &tempProfLoc);
    require(theErr == noErr, cleanup);

    theErr = CMCopyProfile(prof, profLoc, tempProf);
    require(theErr == noErr, cleanup);

// Do any necessary cleanup:close profile and delete file spec.
cleanup:

    if (tempProf)
        theErr = CMCloseProfile(tempProf);

    if (tempCreated)
        theErr = FSpDelete(&tempSpec);

    return theErr;
}
```

Part B: Unflattening the Profile

Prior to ColorSync 2.5, your transfer function is called by the CMM that handles the unflatten operation. Starting with ColorSync 2.5, however, the ColorSync Manager calls your transfer function directly, without going through the preferred, or any, CMM.

When the code in `MyGetIndexedProfileFromPicHandle` (Listing 3-10) calls the `CMUnflattenProc` function, passing it a pointer to the `MyUnflattenProc` function, the `MyUnflattenProc` function (Listing 3-11) is called by ColorSync or by the CMM (depending on the version of ColorSync) to perform the low-level profile data transfer from the document file.

When the `MyUnflattenProc` function is called with an open command, the function initializes global variables, creates a graphics world, and installs bottleneck procedures in the graphics world. The only bottleneck procedure actually used is `MyUnflattenProfilesCommentProc`, which checks the picture comments as the picture is drawn offscreen to identify the desired profile. For a general discussion of customizing QuickDraw's bottleneck routines, see "Customizing QuickDraw's Text Handling" in *Inside Macintosh: Text*.

When the `MyUnflattenProc` function is called with a read command, the function reads the appropriate segment of data from a chunk and returns it. To accomplish this, it calls the `MyDrawPicHandleUsingBottlenecks` function with the appropriate bottleneck procedure installed. In turn, this invokes the `MyUnflattenProfilesCommentProc` shown in Listing 3-12.

When the `MyUnflattenProc` function is called with a close command, the function releases any memory it allocated and disposes of the graphics world and bottlenecks.

Listing 3-11 The unflatten procedure

```
pascal OSErr MyUnflattenProc (long command,
                              long *sizePtr,
                              void *dataPtr,
                              void *refConPtr)
{
    OSErr                   theErr = noErr;
    static CQDProcs         procs;
    static GWorldPtr        offscreen;
    PicHandle               pict;
```

```
switch (command)
{
    case cmOpenReadSpool:
        theErr = NewSmallGWorld(&offscreen);
        if (theErr)
            return theErr;

        /* Replace the QuickDraw bottleneck routines, mostly with routines
            that do nothing, but also with our unflatten comments routine,
            so that we can intercept the comments we are interested in and
            ignore everything else. */
        SetStdCProcs(&procs);
        procs.textProc    = NewQDTextProc (MyNoOpTextProc);
        procs.lineProc    = NewQDLineProc (MyNoOpLineProc);
        procs.rectProc    = NewQDRectProc (MyNoOpRectProc);
        procs.rRectProc   = NewQDRRectPro (MyNoOpRRectProc);
        procs.ovalProc    = NewQDOvalProc (MyNoOpOvalProc);
        procs.arcProc     = NewQDArcProc (MyNoOpArcProc);
        procs.polyProc    = NewQDPolyProc (MyNoOpPolyProc);
        procs.rgnProc     = NewQDRgnProc (MyNoOpRgnProc);
        procs.bitsProc    = NewQDBitsProc(MyNoOpBitsProc);
        procs.commentProc = NewQDCommentProc (MyUnflattenProfilesCommentProc);
        procs.txMeasProc  = NewQDTxMeasProc (MyNoOpTxMeasProc);

        gChunkBaseHndl = nil;
        gChunkIndex = 0;
        gChunkOffset = 0;
        gChunkSize = 0;
        break;

case cmReadSpool:
    if (gChunkOffset > gChunkSize)      /* If we overread the last chunk, */
    {
        return ioErr;                   /* use system I/O error value. */
    }
    if (gChunkOffset == gChunkSize)     /* If we used up the last chunk, */
    {
        if (gChunkBaseHndl !=nil)
        {
            HUnlock(gChunkBaseHndl);    /* dispose of the previous chunk. */
            DisposeHandle(gChunkBaseHndl);
```

```
            gChunkBaseHndl = nil;
        }
        gChunkIndex++;              /* Read in a new chunk. */
        gChunkOffset = 0;
        gCount = 0;
        gChunkCount = 0;
        pict = *((PicHandle *)refConPtr);
        theErr = MyDrawPicHandleUsingBottlenecks (pict, procs, offscreen);
        /* This invokes MyUnflattenProfilesCommentProc shown in Listing 3-12. */
        if (gChunkBaseHndl==nil)    /* Check to see if we're overread. */
            return ioErr;           /* If so, return system I/O error value. */
        HLock(gChunkBaseHndl);
    }
    if (gChunkOffset < gChunkSize)
    {
        *sizePtr = MIN(gChunkSize-gChunkOffset, *sizePtr);
        BlockMove((Ptr)(&((*gChunkBaseHndl)[gChunkOffset])),
            (Ptr)dataPtr, *sizePtr);
        gChunkOffset += (*sizePtr);
    }
    break;

    case cmCloseSpool:
        if (gChunkBaseHndl != nil)
        {
            HUnlock(gChunkBaseHndl);        /* Dispose of the previous chunk. */
            DisposeHandle(gChunkBaseHndl);
            gChunkBaseHndl = nil;
        }
        /* Dispose of our offscreen world and the routine descriptors
            for our bottlenect routines. */
        DisposeGWorld(offscreen);
        DisposeRoutineDescriptor(procs.MyNoOpTextPrc);
        DisposeRoutineDescriptor(procs.MyNoOpLinePrc);
        DisposeRoutineDescriptor(procs.MyNoOpRectProc);
        DisposeRoutineDescriptor(procs.MyNoOpRRectPrc);
        DisposeRoutineDescriptor(procs.MyNoOpOvalProc);
        DisposeRoutineDescriptor(procs.MyNoOpArcProc);
        DisposeRoutineDescriptor(procs.MyNoOpPolyPrc);
        DisposeRoutineDescriptor(procs.MyNoOpRgnProc);
        DisposeRoutineDescriptor(procs.MyNoOpBitsProc);
```

```
            DisposeRoutineDescriptor(procs.MyUnflattenProfilesCommentPrc);
            DisposeRoutineDescriptor(procs.MyNoOpTxMeasPrc);
            break;

        default:
            break;
    }
    return theErr;
}
```

Part C: Calling the Comment Procedure

When the `MyUnflattenProc` function's `MyDrawPicHandleUsingBottlenecks` function calls the `MyUnflattenProfilesCommentProc` function, the function shown in Listing 3-12 finds the profile identified by the index, finds the correct segment of data within the profile, and stores the data in the `gChunkBaseHndl` global variable.

Listing 3-12 The comment procedure

```
pascal void MyUnflattenProfilesCommentProc (short kind,
                                            short dataSize,
                                            Handle dataHandle)
{
    long    selector;
    OSErr   theErr;

    if (gChunkBaseHndl != nil) return;
                /* The handle is in use; this shouldn't happen. */
    if (gCount > gIndex) return;
                /* We have already found the profile. */

    switch (kind)
    {
    case cmBeginProfile:
        gCount ++;          /* We found a version 1 profile. */
        gChunkCount = 1;    /* v1 profiles should only have 1 chunk. */
        if (gCount != gIndex) break;
                            /* This is not the profile we're looking for. */
        if (gChunkCount != gChunkIndex) break;
```

```
                        /* This is not the chunk we're looking for. */
    gChunkBaseHndl = dataHandle;
    theErr = HandToHand(&gChunkBaseHndl);
    gChunkSize = dataSize;
    gChunkOffset = 0;
    break;

case cmComment:
    if (dataSize <= 4) break;
                        /* The dataSize too small for selector, so break. */
    selector = *((long *)(*dataHandle));
                        /* Get the selector from the first long in data. */
    switch (selector)
    {
        case cmBeginProfileSel:
            gCount ++;              /* We found a version 2 profile. */
            gChunkCount = 1;
            if (gCount != gIndex) break;
                                /* This is not the profile we're looking for. */
            if (gChunkCount!=gChunkIndex) break;
                                /* This is not the chunk we're looking for. */
            gChunkBaseHndl = dataHandle;
            theErr = HandToHand(&gChunkBaseHndl);
            gChunkSize = dataSize;
            gChunkOffset = 4;
            break;

        case cmContinueProfileSel:
            gChunkCount ++;
            if (gCount != gIndex) break;
                                /* This is not the profile we're looking for. */
            if (gChunkCount!=gChunkIndex) break;
                                /* This is not the chunk we're looking for. */
            gChunkBaseHndl = dataHandle;
            theErr = HandToHand(&gChunkBaseHndl);
            gChunkSize = dataSize;
            gChunkOffset = 4;
            break;

        case cmEndProfileSel:
                                /* Check to see if we're overreading. */
```

```
                gChunkCount = 0;
                break;
        }
        break;
    }
}
```

Performing Optimized Profile Searching

Starting with version 2.5, ColorSync provides a profile cache and a new routine, `CMIterateColorSyncFolder` (page 304), for optimized profile searching. The sample code shown in Listing 3-13 through Listing 3-15 takes advantage of optimized searching if ColorSync version 2.5 is available; if not, it performs a search that is compatible with earlier versions of ColorSync. The compatible search may take some advantage of the profile cache, but cannot provide fully optimized results.

As background for the code samples in Listing 3-13 to Listing 3-15, you should be familiar with the topics described in the following sections:

- "Profile Location" (page 53)
- "Profile Search Locations" (page 55)
- "The Profile Cache and Optimized Searching" (page 57)

IMPORTANT
You cannot use the ColorSync Manager search functions to search for ColorSync 1.0 profiles. ▲

The `CMIterateColorSyncFolder` function uses ColorSync's profile cache to supply your application with information about the profiles currently available in the ColorSync Profiles folder. The function calls your callback routine once for each available profile, supplying your routine with the profile header, script code, name, and location, stored in a structure of type `CMProfileIterateData` (page 366).

Even though there may be many profiles available, `CMIterateColorSyncFolder` can take advantage of ColorSync's profile cache to return profile information quickly, and (if the cache is valid) without having to open any profiles. As a result, your routine may be able to perform its function, such as building a list of profiles to display in a pop-up menu, quickly and without having to open each file-based profile.

An Iteration Function for Profile Searching With ColorSync 2.5

The `CMIterateColorSyncFolderCompat` function, shown in Listing 3-15 (page 135), performs an optimized search using the `CMIterateColorSyncFolder` (page 304) function if ColorSync version 2.5 is available. Otherwise, it calls the`CMNewProfileSearch` (page 308) function, which is available in earlier versions of ColorSync.

When you call the `CMIterateColorSyncFolderCompat` function, you pass a universal procedure pointer to a filter procedure in the `proc` parameter. `CMIterateColorSyncFolderCompat` uses that filter procedure when it performs an optimized search with `CMIterateColorSyncFolder`. Listing 3-13 provides a sample filter procedure called `MyIterateProc`.

The `MyIterateProc` function is called once for each available profile and merely stores the names of all non-display profiles (such as printer and scanner profiles) at an arbitrary position in a list. You would do something similar, for example, to display a list of profiles in a dialog.

Note that the `CMIterateColorSyncFolderCompat` function works in a similar way for ColorSync 2.5 and for earlier versions, although the search is much more efficient with version 2.5. `CMIterateColorSyncFolderCompat` either calls the `CMIterateColorSyncFolder` function, which calls the `MyIterateProc` function once for each available profile, or it calls the `CMNewProfileSearch` (page 308) function, which calls the `ProfileSearchFilter` function (Listing 3-14) once for each available profile. The `ProfileSearchFilter` function in turn calls `MyIterateProc`, so similar processing occurs.

Listing 3-13 An iteration function for profile searching with ColorSync 2.5

```
Pascal OSErr MyIterateProc(CMProfileIterateData* data, void* refcon)
{
    Cell theCell;
    // Assume we can cast refCon to a ListHandle.
    ListHandle list = (ListHandle)refcon;

    /* Assume we're interested only in non-display profiles, such as printer
        and scanner profiles. */
    if (data->header.profileClass != cmDisplayClass)
    {
        /* This code adds the profile name at an arbitrary position
            in a list. You could do something similar to display a
```

```
            list of all available profiles. */
        cell.v = LAddRow(1,999,list);
        cell.h = 0;
        // The name data in the iterate data structure is in Pascal format,
            so we use the length byte to determine how many bytes to copy. */
        LSetCell((Ptr)data->name+1, name[0], cell, list);
        cell.h = 1;
        // Store the profile's location information with the cell.
        LSetCell((Ptr)data->location, sizeof(cmProfileLocation), cell, list);
    }
    // A more complicated function might need to return an error here.
    return noErr;
};
```

A Filter Function for Profile Searching Prior to ColorSync 2.5

To search for profiles prior to version 2.5 of the ColorSync Manager, you use the `CMNewProfileSearch` (page 308) function. You supply `CMNewProfileSearch` with a search record of type `CMSearchRecord` (page 368) that identifies the search criteria. If you also provide a pointer to a filter function, `CMNewProfileSearch` uses the function to eliminate profiles from the search based on additional criteria not defined by the search record. The `ProfileSearchFilter` function shown in Listing 3-14 provides an example of a filter routine for searching with the `CMNewProfileSearch` function.

Listing 3-14 defines the `IterateCompatPtr` data type, a pointer to a structure that stores search information. When you call the `CMIterateColorSyncFolderCompat` function shown in Listing 3-15, you pass a reference to the `MyIterateProc` function (Listing 3-13) in the `proc` parameter. If ColorSync 2.5 is not available, the `CMIterateColorSyncFolderCompat` function calls the `CMNewProfileSearch` function. It passes the `ProfileSearchFilter` function (Listing 3-13) as the search filter and it passes an `IterateCompatPtr` pointer as the `refCon` parameter. It sets the `proc` field of the `IterateCompatPtr` pointer to the `MyIterateProc` function that you passed in the `proc` parameter.

The `CMNewProfileSearch` function calls the `ProfileSearchFilter` function (Listing 3-13) once for each profile. The `ProfileSearchFilter` function simply casts the passed `refCon` pointer to an `IterateCompatPtr`, then calls the function specified by the pointer's `proc` field. As a result, the `MyIterateProc` function is called once for each profile, just as it is when `CMIterateColorSyncFolderCompat` calls `CMIterateColorSyncFolder` under ColorSync 2.5.

Note that `ProfileSearchFilter` always returns `true`, indicating the profile should be filtered out of the search result returned by `CMNewProfileSearch`, because we've already gotten all the information we need from it. Note also that `ProfileSearchFilter` uses the `require` macro, which is defined in "Poor Man's Exception Handling" (page 98).

Listing 3-14 A filter function for profile searching prior to ColorSync 2.5

```
// Declare a structure to use for searching with ColorSync versions prior to 2.5.
typedef struct IterateCompat
{
    CMProfileIterateUPP     proc;
    OSErr                   osErr;
    void*                   refCon;
} IterateCompatRec, *IterateCompatPtr;

static pascal Boolean ProfileSearchFilter (CMProfileRef prof, void *refCon)
{
    OSErr                   theErr = noErr;
    IterateCompatPtr        refConCompatPtr;
    CMProfileIterateData    iterData;

    // Cast refcon to our type.
    refConCompatPtr = (IterateCompatPtr)refCon;

    // If we had an error from an earlier profile, give up
    //  by branching to cleanup location.
    theErr = refConCompatPtr->osErr;
    require(theErr == noErr, cleanup); // require is defined in Listing 3-3 (page 98).

    // Try to get the profile's location.
    theErr = CMGetProfileLocation(prof, &iterData.location);
    require(theErr == noErr, cleanup);

    // Try to get the profile's header.
    theErr = CMGetProfileHeader(prof, (CMAppleProfileHeader*)&iterData.header);
    require(theErr == noErr, cleanup);

    // Try to get the profile's name.
    theErr = CMGetScriptProfileDescription(prof, iterData.name, &iterData.code);
```

```
    require(theErr == noErr, cleanup);

    iterData.dataVersion = cmProfileIterateDataVersion1;

    // Call the iterate callback routine.
    theErr = CallCMProfileIterateProc(refConCompatPtr->proc,
                                      &iterData, refConCompatPtr->refCon);
    require(theErr == noErr, cleanup);

cleanup:

    if (theErr)
        refConCompatPtr->osErr = theErr;

    return true;   // exclude the profile;
}
```

A Compatible Function for Optimized Profile Searching

Listing 3-15 provides sample code that performs an optimized profile search if ColorSync 2.5 is available, but provides a search that is compatible with previous versions if ColorSync 2.5 is not available.

When ColorSync 2.5 is available, `CMIterateColorSyncFolderCompat` simply calls the function `CMIterateColorSyncFolder` (page 304), passing on the information it received through its parameters. As a result, `CMIterateColorSyncFolder` calls the `MyIterateProc` function, shown in Listing 3-13 (page 131), once for each available profile. Your version of `MyIterateProc` can examine the passed information for each profile and perform any required operation on the profiles it is interested in.

When ColorSync 2.5 is not available, `CMIterateColorSyncFolderCompat` sets up a search with the function `CMNewProfileSearch` (page 308). As part of this setup, it initializes a structure of type `IterateCompatRec`, defined in Listing 3-14 (page 133), which it passes to `CMNewProfileSearch` for the `refCon` parameter. The `CMNewProfileSearch` function in turn passes a pointer to the `IterateCompatRec` structure as the `refCon` parameter to `ProfileSearchFilter`, which it calls once for each available profile.

`ProfileSearchFilter` calls the `MyIterateProc` function, which gets a chance to handle each profile, just as it does in the case where ColorSync 2.5 is available. The main drawback is that without the availability of the profile cache and the

CMIterateColorSyncFolder function, searching through the profiles is likely to be a much more time-consuming task.

Note that CMIterateColorSyncFolderCompat uses the require macro, which is defined in "Poor Man's Exception Handling" (page 98).

Listing 3-15 Optimized profile searching compatible with previous versions of ColorSync

```
CMError CMIterateColorSyncFolderCompat (CMProfileIterateUPP proc,
                                        unsigned long *seed,
                                        unsigned long *count,
                                        void *refCon)
{
    CMError theErr = noErr ;

    /* Presume the caller passed a pointer to MyIterateProc to this
            function in the proc parameter. */
    if ( ColorSync25Available() ) // This routine is shown in Listing 3-1 (page 92).
        return CMIterateColorSyncFolder(proc, seed, count, refCon);
    else
    {
        CMProfileSearchRef  searchResult;
        CMSearchRecord      searchSpec;
        unsigned long       count;
        IterateCompatRec    refConCompat;

        /* Set up a search record to pass to CMNewProfileSearch. Include
            procedure pointer to search filter from Listing 3-14 (page 133). */
        searchSpec.filter = NewCMProfileFilterProc(ProfileSearchFilter);
        searchSpec.searchMask = cmMatchAnyProfile;

        /* Set up our private data structure for compatible (pre-ColorSync 2.5)
            profile searching.
            Pass the pointer to the MyIterateProc function, which was
                presumably passed to this function in the proc parameter,
                on to our filter routine, ProfileSearchFilter,
                in the refCon parameter, using an IterateCompatRec structure. */
        refConCompat.proc = proc;
        refConCompat.osErr = noErr;
```

```
        refConCompat.refCon = refCon;

        //  Start traditional search.
        theErr = CMNewProfileSearch(&searchSpec,
                            (void*)&refConCompat, &count, &searchResult);
        if (theErr == noErr)
        {
            // We don't use the result, but still must dispose of it.
            CMDisposeProfileSearch(searchResult);
            theErr = refConCompat.osErr;
        }
        DisposeRoutineDescriptor(searchSpec.filter);
    }

    return theErr;
}
```

Searching for Specific Profiles Prior to ColorSync 2.5

Starting with version 2.5, you can do fast, optimized profile searching that takes advantage of the profile cache added in ColorSync 2.5. For an overview, see "The Profile Cache and Optimized Searching" (page 57). The sample code in Listing 3-15 (page 135) takes advantage of optimized searching if ColorSync version 2.5 is available; if not, it performs a search that is compatible with earlier versions of ColorSync. The compatible search may take some advantage of the profile cache, but cannot provide fully optimized results.

Listing 3-16, shown in this section, provides an additional example of the searching mechanism available prior to ColorSync version 2.5.

IMPORTANT

You cannot use the ColorSync Manager search functions to search for ColorSync 1.0 profiles. ▲

Your application can use the ColorSync Manager search functions to obtain a list of profiles in the ColorSync Profiles folder that meet specifications you supply in a search record. For example, you can use these functions to find all profiles for printers that meet certain criteria defined in the profile. Your application can walk through the resulting list of profiles and obtain the name and script code of each profile corresponding to a specific index in the list. Your application can then display a selection menu showing the names of the

profiles. Listing 3-16 shows sample code that takes an approach similar to the one this example describes.

Note
You can also search the ColorSync Profiles folder for profiles that match a profile identifier. For more information, see “Searching for a Profile That Matches a Profile Identifier” (page 139), and `CMProfileIdentifierFolderSearch` (page 315). ◆

The `MyProfileSearch` function, shown in Listing 3-16, defines values for the search specification record fields, including the search mask, and assigns those values to the record’s fields after initializing the search result. Then `MyProfileSearch` calls the `CMNewProfileSearch` function to search the ColorSync Profiles folder for profiles that meet the search specification requirements. The `CMNewProfileSearch` (page 308) function returns a one-based count of the profiles matching the search specification and a reference to the search result list of the matching profiles.

Next the `MyProfileSearch` function calls the `CMSearchGetIndProfile` (page 312) function to obtain a reference to a specific profile corresponding to a specific index into the search result list. Passing the profile reference returned by the `CMSearchGetIndProfile` function as the `foundProf` parameter, `MyProfileSearch` calls the `CMGetScriptProfileDescription` (page 256) function to obtain the profile name and script code.

Finally, the `MyProfileSearch` function cleans up, calling the `CMCloseProfile` function to close the profile and the `CMDisposeProfileSearch` function to dispose of the search result list.

Listing 3-16 Searching for specific profiles in the ColorSync Profiles folder

```
// NOTE: The preferred mechanism for searching in ColorSync 2.5 is shown
//          in Listing 3-15 (page 135).

/* field definitions for search */
#define kCMMType        'appl'              /* ColorSync default CMM */
#define kProfileClass   cmDisplayClass      /* monitor */
#define kAttr0          0x00000000
#define kAttr1          0x00000002          /* Macintosh standard gamma */
```

```
/* Define mask to search for profiles that match on CMM type, profile class,
        and attributes. */
#define kSearchMask (cmMatchProfileCMMType + cmMatchProfileClass + cmMatchAttributes)

void MyProfileSearch (void)
{
    CMError             cmErr;
    CMProfileRef        foundProf;
    Str255              profName;
    ScriptCode          profScript;
    CMSearchRecord      searchSpec;
    CMProfileSearchRef  searchResult;
    unsigned long       searchCount;
    unsigned long       i;

    /* Init for error handling. */
    searchResult = NULL;

    /* Specify search. */
    searchSpec.CMMType = kCMMType;
    searchSpec.profileClass = kProfileClass;
    searchSpec.deviceAttributes[0 ]= kAttr0;
    searchSpec.deviceAttributes[1] = kAttr1;

    searchSpec.searchMask = kSearchMask;

    searchSpec.filter= NULL;                /* Filter proc is not used. */

    cmErr = CMNewProfileSearch(&searchSpec, NULL, &searchCount, &searchResult);

    if (cmErr == noErr)
    {
        for (i = 1; i <= searchCount; i++)
        {
            if (CMSearchGetIndProfile(searchResult, i, &foundProf) != noErr)
            {
                break;
            }

            cmErr = CMGetScriptProfileDescription(foundProf, profName, &profScript);
```

```
            if (cmErr == noErr)
            {
                /* Assume profile name ScriptCode is smRoman. */
                (void) printf("%s\n", p2cstr(profName));
            }

            (void) CMCloseProfile(foundProf);
        }
    }

    if (searchResult != NULL)
    {
        CMDisposeProfileSearch(searchResult);
    }
}
```

Searching for a Profile That Matches a Profile Identifier

Embedding a profile in an image guarantees that the image can be rendered correctly on a different system. However, profiles can be large—the largest can be more than several hundred kilobytes. The ColorSync Manager defines a profile identifier structure, `CMProfileIdentifier`, that can identify a profile but that takes up much less space than a large profile.

The profile identifier structure contains a profile header, an optional calibration date, a profile description string length, and a variable-length profile description string. Your application might use an embedded profile identifier, for example, to change just the rendering intent or flag values in an image without having to embed an entire copy of a profile. For more information on the profile identifier structure, including a description of how a match is determined between a profile reference and a profile identifier, see `CMProfileIdentifier` (page 359).

IMPORTANT

A document containing an embedded profile identifier can not necessarily be ported to different systems or platforms. ▲

The ColorSync Manager provides the `NCMUseProfileComment` (page 290) routine to embed profiles and profile identifiers in an open picture file. For information on embedding, see “Embedding Profiles and Profile Identifiers” (page 112).

Your application can embed profile identifiers in place of entire profiles, or in addition to them. A profile identifier can refer to an embedded profile or to a profile on disk.

The ColorSync Manager provides the `CMProfileIdentifierListSearch` (page 316) routine for finding a profile identifier in a list of profile identifiers and the `CMProfileIdentifierFolderSearch` (page 315) routine for finding a profile identifier in the ColorSync Profiles folder.

When your application or device driver processes an image, it typically keeps a list of profile references for each profile it encounters in the image. Each time it encounters an embedded profile identifier, your application first calls the `CMProfileIdentifierListSearch` function to see if there is already a matching profile reference in its list. That function returns a list of profile references that match the profile identifier. Although the returned list would normally contain at most one reference, it is possible to have two or more matches. If the `CMProfileIdentifierListSearch` routine does not find a matching profile reference, your application calls the `CMProfileIdentifierFolderSearch` routine to see if a matching profile can be found in the ColorSync Profiles folder.

Listing 3-17 demonstrates how your application can use the ColorSync Manager's search routines to obtain a profile reference for an embedded profile identifier. It uses the following structure to store a list of profile identifiers, along with a count of the number of items in the list.

```
typedef struct {
    long count;
    CMProfileRef profs[1];
} ProfileCacheList, **ProfileCacheHandle;
```

Listing 3-17 Searching for a profile that matches a profile identifier

```
CMError MyFindAndOpenProfileByIdentifier(ProfileCacheHandle profCache,
                                         CMProfileIdentifierPtr unique,
                                         Boolean *pFoundInCache,
                                         CMProfileRef *pProf)
{
    CMError         theErr = noErr;
    CMProfileRef    prof = nil;
    long            cacheCount = (**profCache).count;
    unsigned long   foundCount = 0;
```

```
*pFoundInCache = false;

/* If there are any profile references in the cache (the list of profile
    references for profiles or profile identifiers we have already
    encountered) look there for a match with the passed profile identifier. */
if (cacheCount)
{
    CMProfileRef *cacheList;

    cacheList = (**profCache).profs;
    foundCount = 1; // return no more than one match
    theErr = CMProfileIdentifierListSearch(unique, cacheList, cacheCount,
                                        &foundCount, &prof);
    if (foundCount && !theErr)
        *pFoundInCache = true;
    else
        prof = nil;
}

/* If we didn't find a match for the passed profile identifier in the list of
    previously encountered profiles, look for a match on disk, in the
    ColorSync Profiles folder */
if (!prof)
{
    CMProfileSearchRef search = nil;
    foundCount = 0;

    theErr = CMProfileIdentifierFolderSearch(unique, &foundCount, &search);
    /* If we found one or more matches, obtain a profile reference for the
        first matching profile; if no error, dispose of the search result. */
    if (!theErr)
    {
        if (foundCount)
            theErr = CMSearchGetIndProfile(search, 1, &prof);
        CMDisposeProfileSearch(search);
    }
}

/* If we still didn't find a match for the passed profile identifier,
    use the system profile. */
```

```
    if (!prof)
    {
        theErr = CMGetSystemProfile(&prof);
    }

    if (theErr)
        prof = nil;
    *pProf = prof;
    return theErr;
}
```

Although typically there is at most one profile reference in your application's list or one profile in the ColorSync Profiles folder that matches the searched-for profile identifier, it is possible that two or more profiles may qualify. It is not an error condition if either the `CMProfileIdentifierListSearch` or the `CMProfileIdentifierFolderSearch` routine finds no matching profile.

Checking Colors Against a Destination Device's Gamut

Different imaging devices (scanners, displays, printers) work in different color spaces, and each can have a different gamut or range of colors that they can produce. The process of matching colors between devices entails adjusting the colors of an image from the color gamut of one device to the color gamut of another device so that the resulting image looks as similar as possible to the original image. Not all colors can be rendered on all devices. The rendering intent used in the color transformation process dictates how the colors are matched, strongly influencing the outcome. Your application can give a user some control over the outcome by allowing the user to select the rendering intent. However, some users might want to know in advance which colors are out of gamut for the destination device so that they can choose other appropriate colors within the gamut.

Using the ColorSync Manager general purpose color-checking functions, your application can check the colors of a pixel map (using the `CWCheckPixMap` (page 274) function), the colors of a bitmap (using the `CWCheckBitMap` (page 279) function), or a list of colors (using the `CWCheckColors` function) against the color gamut of the destination device and provide a warning when a color is out of gamut for that device.

There are a number of ways in which your application can provide gamut-checking services. For example, you can use gamut checking to see if a

given color is reproducible on a particular printer. If the color is not directly reproducible—that is, if it is out of gamut—you could alert the user to that fact.

You can allow a user to specify a list of colors that fall within the gamut of a source device to see if they fit within the gamut of a destination device before the user color matches an image. Your application could display the results in a window, indicating which colors are in the gamut and which are out. This feature, too, gives the user the opportunity to test colors and select different ones for portions of an image whose colors fall out of gamut. To handle this feature, your application can call the `CWCheckColors` (page 283) function.

In addition to providing features that allow a user to anticipate which colors are out of gamut for a particular device, your application can also show results. Your application can provide a print preview dialog box, showing which colors in a printed image, for example, are out of gamut for the image as it appears on the screen.

For an image that your application prepares, for example, your application can present a print preview dialog box that signifies those colors within the image that the printer cannot accurately reproduce. Your application can also allow users to choose whether and how to match colors in the image with those available on the printer.

You can provide a gamut-checking feature that marks the areas of a displayed image, showing the colors that do not fall within the destination device's gamut. For example, your application can color check an image against a destination device and create a black-and-white version of the image drawn to the display using black to indicate the portions of the source image that are out of gamut. The CSDemo sample application takes this approach. For information on how to obtain the CSDemo application, see "Extracting Profiles Embedded in Pictures" (page 118).

Creating and Using Device Link Profiles

To accommodate users who use a specific configuration requiring a combination of device profiles and possibly non-device profiles repeatedly over time, your application can create device link profiles. A device link profile offers a means of saving and storing a series of profiles corresponding to a specific configuration in a concatenated format. This feature provides an economy of effort for both your application and its user.

There are many uses for device link profiles. For example, a user might want to store multiple profiles, such as various device profiles and color space profiles associated with the creation and editing of an image.

Most users use the same device configuration to scan, view, and print graphics over a period of time, often soft proofing images before they print them. To enhance your application's soft-proofing feature, you can allow users to store the contents of the profiles involved in the soft-proofing process in a device link profile. Your application can use the appropriate device link profile each time a user enacts the soft-proofing feature, instead of opening a profile reference to each of the profiles to create a color world to pass to the color-matching functions. For additional information about soft proofing, see "Providing Soft Proofs" (page 147).

A device link profile is especially useful when a scanner application does not embed the source profile in the document containing the image it creates. By storing the scanner's profile, your application eliminates the need to query the user for the appropriate source profile each time the user wants to soft proof using the configuration involving that scanner.

A user may want to see how a scanned image will look when printed using a specific printer. The user may want to look at many images captured on the same scanner at different times before printing the image. Because the same devices are involved in the process, if your application has offered the user the opportunity to create device link profiles, your application could display a list of device link profiles that the user had previously created for various configurations and allow the user to select the appropriate one for the current soft proofing.

Here are the steps your application should take in creating a device link profile:

1. **Open the profiles corresponding to the devices and transformations involved in the configuration and obtain references to them.**

   To create a device link profile, your application must first obtain references to the profiles involved in the configuration. If the profile for an input device, such as a scanner, is embedded in the document containing the image, you must first extract the profile. For a description of how to obtain a profile reference, see "Obtaining Profile References" (page 95). For information describing how to extract a profile from a document, see "Extracting Profiles Embedded in Pictures" (page 118).

2. **Create an array containing references to the profiles, specifying the profile references in processing order.**

You supply the profile references as an array of type `CMProfileRef` within a data structure of type `CMConcatProfileSet`. The order of the profiles must correspond to the order in which you want the colors of the image to be processed. For example, for soft proofing an image, you should specify the scanner profile reference first, followed by the printer profile reference, and then the display profile reference because the goal is to match the colors of the scanned image to the color gamut of the printer for which the image is destined and then display the results to the user.

In the `count` field, specify a one-based number identifying how many profiles the array holds. A device link profile represents a one-way link between devices.

Here is the `CMConcatProfileSet` data type:

```
struct CMConcatProfileSet {
    unsigned short      keyIndex;           /* zero-based */
    unsigned short      count;              /* one-based */
    CMProfileRef        profileSet[1];
};
```

You must adhere to the rules that govern the type of profiles you can specify in the array. For example, the first and last profiles must be device profiles. For a list of these rules, see `CMConcatProfileSet` (page 384).

3. **Specify the index corresponding to the profile whose specified CMM is used to perform the processing.**

   The header of each profile specifies a CMM for that profile. Only one CMM is used for all transformations across the profiles of a device link profile. You identify the profile whose CMM is used by supplying the zero-based index of that profile in the `keyIndex` field of the `CMConcatProfileSet` (page 384) data type.

IMPORTANT

See "How the ColorSync Manager Selects a CMM" (page 84) for a complete description of the ColorSync algorithm for selecting a CMM. ▲

4. **Using the `CMProfileLocation` data type, provide a file specification for the new device link profile.**

   If the function `CWNewLinkProfile` (page 267) is successful, the ColorSync Manager creates a device link profile in the location that you specify, opens a reference to the profile, and returns the profile reference to your application.

To tell the ColorSync Manager where to create the new profile, your application must provide a file specification. The ColorSync Manager defines a data structure of type `CMProfileLocation` containing a `CMProfLoc` union that you use to give a file specification.See Listing 3-2 (page 97), which assigns values to a `CMProfileLocation` data structure.

5. **Call the `CWNewLinkProfile` function to create the device link profile.**

   After you set up `CMConcatProfileSet` and `CMProfileLocation` (page 362), your application can call the function `CWNewLinkProfile` (page 267), passing these values to it. If the function completes successfully, it returns a reference to the newly created device link profile.

   Note that you should not embed a device link profile into a document along with an image that uses it, as embedded profiles specify source device characteristics only.

6. **Using the `CWConcatColorWorld` function, create a color world based on the device link profile.**

   You can use a device link profile with the general purpose ColorSync Manager functions only. To use a device link profile for a color-matching or color gamut-checking function, you must first create a color world using the `CWConcatColorWorld` function, passing to it a data structure of type `CMConcatProfileSet` (page 384). The `CMConcatProfileSet` data structure is the same data type that you used to specify the array of profiles when you created the new device link profile. To create the color world, however, you specify the device link profile as the only member of the `CMConcatProfileSet` array. If the `CWConcatColorWorld` function is successful, it returns a reference to a color world that your application can pass to other general purpose functions for color-matching and color gamut-checking sessions. A device link profile remains intact and available for use again after your application calls the `CWDisposeColorWorld` (page 271) function to dispose of the concatenated color world.

Considerations

Here are some points to consider about how the ColorSync Manager uses information contained in the profiles comprising a device link profile:

- When you use a device link profile, the quality flag setting—indicating normal mode, draft mode, or best mode—specified by the first profile prevails for the entire session; the quality flags of following profiles in the

sequence are ignored. The quality flag setting is stored in the `flags` field of the profile header.

- The ColorSync Manager uses 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.

When your application is finished with the device link profile, it must close the profile with the `CMCloseProfile` function.

Providing Soft Proofs

Your application can use ColorSync to provide soft-proofing. Soft-proofing enables a user to preview the printed results of a color image on the system's display or local printer without actually outputting the image to the printer that will produce the final image. The destination printer's profile provides the ColorSync Manager with the information required to determine how the colors of the image will appear when printed. You can soft proof an image by showing on the system's display the outcome a printer would produce because most displays support a wider color gamut than do printers. Therefore, a display will probably be able to show all the colors a printer could support.

Providing a feature that simulates the printed outcome for the user to preview can save users considerable time and cost by allowing them to intervene and adjust colors before sending the image to a printing shop. For example, without the ability to soft proof and correct the colors of an image using a color management system such as ColorSync, a graphics designer producing a poster to be printed by a printing press would require the services of a prepress shop to achieve the correct results before sending the image to the printing press. The graphics designer might print the image to a local desktop printer with a color gamut more limited than that of a printing press and then submit the output to the prepress to correct the colors, repeating this process until the results were satisfactory. Your application can eliminate the need for the intermediate steps by allowing the user to color match the image to the color gamut of the final printing press, display the image, and adjust the colors accordingly.

You can use the general purpose color-matching functions `CWMatchPixMap` (page 272) and `CWMatchBitmap` (page 276) to perform the color matching, or you can match a list of colors using the `CWMatchColors` (page 281) function. To use these functions, your application must first define a color world that encompasses the profiles for the devices involved in the soft-proofing process.

For example, suppose a user intends to create a color image by drawing to the display, then color matching the image to the color gamut of the printing press and printing the image to a local desktop printer before delivering it to the printing press. The user intends to repeat this process until he or she is satisfied with the color rendering. To allow the user to do this, your application must build a color world using the profile for the display device, the profile for the printing press, and the profile for the local desktop printer; you must specify the profiles in processing order. Because the process involves three profiles, your application must use the function `CWConcatColorWorld` (page 265) to set up the color world. "Creating a Color World to Use With the General Purpose Functions" (page 105) describes how to set up a color world.

You can preserve the series of profiles from a soft-proofing process for future use by creating a device link profile representing the configuration and passing the device link profile to the `CWConcatColorWorld` function to set up a color world. For information on how to create and use a device link profile to build a color world, see "Creating and Using Device Link Profiles" (page 143).

Your application can also use the QuickDraw-specific `NCMBeginMatching` (page 285) and `CMEndMatching` (page 287) functions for soft proofing of a color image drawn to the display that a user wants to color match to the gamut of a printing press and print to a desktop printer.

The `NCMBeginMatching` function matches the colors using the two profiles that you specify, and the `CMEndMatching` function terminates the color-matching session. Because the `NCMBeginMatching` function takes two profiles only—a source profile and a destination profile—you must call sets of these functions to enact soft proofing.

QuickDraw matches to the most recently added profiles first. Therefore, to use the `NCMBeginMatching` and `CMEndMatching` pair to perform soft proofing from a displayed image to a printing press output image to a desktop printer image, you would first call the `NCMBeginMatching` function with the printing press to desktop printer profile references and then call `NCMBeginMatching` with the display to printing press profile references. QuickDraw will color match all drawing from display to printing press and then to the desktop printer.

To use the `NCMBeginMatching` function, you specify the source and destination profiles. Passing `NULL` as the source profile assures that the ColorSync Manager uses the system profile as the source profile. Similarly, passing `NULL` as the destination profile uses the system profile as the destination profile.

Calibrating a Device

A calibration application either creates a profile or tunes a profile to represent the current state of the device.

A profile contains two types of device information: the actual calibration information describing how to perform the color match and the device settings at the time the match was made, for example, paper type, ink flow, or film exposure time. A device may have several profiles, each for a different setting, such as paper type or ink.

Your calibration program should first turn off matching on the device and generate its image. You should then perform the calibration and generate a profile. For related information, see "Monitor Calibration and Profiles" (page 67)

Accessing a Resource-Based Profile With a Procedure

The ColorSync Manager provides for multiple concurrent accesses to a single profile through the use of a private data structure called a *profile reference.* When you call the `CMOpenProfile` (page 222) function to open a profile or the `CMNewProfile` (page 227), `CWNewLinkProfile` (page 267), or `CMCopyProfile` (page 229) functions to create or copy a profile, you pass a profile location and the function returns a profile reference. To specify the profile location, you use a structure of type `CMProfileLocation`, as described in "Opening a Profile and Obtaining a Reference to It" (page 95).

A ColorSync profile that you open or create is typically stored in one of the following locations:

- In a disk file. The `u` field (a union) of the `CMProfileLocation` data structure contains a file specification for a profile that is disk-file based. This is the most common way to store a ColorSync profile.
- In relocatable memory. The `u` field of the profile location data structure contains a handle specification for a profile that is stored in a handle.
- In nonrelocatable memory. The `u` field of the profile location data structure contains a pointer specification for a profile that is pointer based.
- In an arbitrary location accessed by a procedure you provide. The `u` field of the profile location data structure contains a universal procedure pointer to your access procedure, as well a pointer that may point to data associated with your procedure.

The sample code in Listing 3-18 to Listing 3-29 demonstrates how to use a profile access procedure to provide access to a resource-based profile.

Note
While the following sample code includes some error handling, more complete error handling is left as an exercise for the reader. ◆

Defining a Data Structure for a Resource-Based Profile

The sample code listings that follow use the application-defined `MyResourceLocRec` data structure. It stores information to describe a resource-based profile, including

- the resource file specification
- the resource type
- the resource ID
- the resource file reference
- the resource handle
- the profile access procedure pointer
- the resource name

```
struct MyResourceLocRec {
    FSSpec              resFileSpec;
    ResType             resType;
    short               resID;
    short               resFileRef;
    Handle              resHandle;
    CMProfileAccessUPP  proc;
    Str255              resName;
};

typedef struct MyResourceLocRec MyResourceLocRec, *MyResourceLocPtr;
```

The ColorSync Manager defines the `CMProfileAccessUPP` type as follows:

```
typedef UniversalProcPtr CMProfileAccessUPP;
```

Setting Up a Location Structure for Procedure Access to a Resource-Based Profile

The `MyCreateProcedureProfileAccess` routine shown in Listing 3-18 sets up a `CMProfileLocation` (page 362) structure for procedure access to a resource-based profile. The `MyDisposeProcedureProfileAccess` routine, shown in Listing 3-19, disposes of memory allocated by `MyCreateProcedureProfileAccess`. Your application uses these routines (or similar ones that you write) in the following way:

1. Before calling a ColorSync Manager routine such as `CMCopyProfile` (page 229), you call the `MyCreateProcedureProfileAccess` routine to set up a `CMProfileLocation` structure that you can pass to the ColorSync Manager routine. The location structure specifies your profile-access procedure and may provide other information as well. A sample profile-access procedure is shown in Listing 3-20.
2. During the course of its operations, the ColorSync Manager may call your profile-access procedure many times.
3. After the ColorSync Manager routine has completed its operation, and if your application does not need to use the `CMProfileLocation` structure for another operation, you call the `MyDisposeProcedureProfileAccess` routine to dispose of memory allocated by `MyCreateProcedureProfileAccess`.

For the sample `MyCreateProcedureProfileAccess` routine shown in Listing 3-18, you pass a pointer to a `CMProfileLocation` structure to fill in, a pointer to a file specification for the resource file containing the profile resource, the type of the resource, the ID for the resource, and optionally the name of the resource (stored as a Pascal string, where the first byte is a length byte for the string).

Note
Listing 3-18 assumes the profile access routine, `MyCMProfileAccessProc`, is within the scope of the `MyCreateProcedureProfileAccess` routine. Optionally, you could add a parameter to pass in a procedure pointer for the profile access routine. ◆

Listing 3-18 Setting up a location structure for procedure access to a resource-based profile

```
OSErr MyCreateProcedureProfileAccess (
                                    CMProfileLocation *profileLocation,
                                    FSSpec *resourceSpec,
                                    Str255 resourceName,
                                    OSType resourceType,
                                    short resourceID)
{
    OSErr               theErr = noErr;
    MyResourceLocPtr    resourceInfo;

    /* Allocate memory for our private resource info structure. */
    resourceInfo = (MyResourceLocPtr) NewPtrClear(sizeof(MyResourceLocRec));
    if (!resourceInfo)
        theErr = MemError();

    if (!theErr)
    {
        /* Set up our private resource info structure. */
        resourceInfo->resFileSpec = *resourceSpec;
        resourceInfo->resType = resourceType;
        resourceInfo->resID = resourceID;
        resourceInfo->resFileRef = 0;
        resourceInfo->resHandle = 0;
        resourceInfo->proc = NewCMProfileAccessProc(MyCMProfileAccessProc);
        /* If a resource name was passed in, copy it to the structure;
            since it's a Pascal string, first byte is length;
            note that BlockMoveData is faster than BlockMove for a
            move that involves data only. */
        if (resourceName)
            BlockMoveData(resourceName, resourceInfo->resName,
                          resourceName[0]+1);

        /* set up the profile location structure */
        profileLocation->locType = cmProcedureBasedProfile;
        profileLocation->u.procLoc.refCon = (void*) resourceInfo;
        profileLocation->u.procLoc.proc = resourceInfo->proc;
```

```
    }
    return theErr;
}
```

If the `MyCreateProcedureProfileAccess` routine is able to set up the profile location pointer for procedure access to a resource-based profile, it returns a value of `noErr`.

Disposing of a Resource-Based Profile Access Structure

Your application calls the `MyDisposeProcedureProfileAccess` routine (Listing 3-19) to dispose of any memory allocated by the `MyCreateProcedureProfileAccess` routine (Listing 3-18).

Listing 3-19 Disposing of a resource-based profile access structure

```
void MyDisposeProcedureProfileAccess (CMProfileLocation *profileLocation)
{
    DisposeRoutineDescriptor(profileLocation->u.procLoc.proc);

    /* Dispose of our private resource info structure. */
    DisposePtr((Ptr)profileLocation->u.procLoc.refCon);
}
```

This routine first disposes of the universal procedure pointer to your profile access procedure, then disposes of the pointer used to store resource data in a `MyResourceLocRec` structure.

Responding to a Procedure-Based Profile Command

For information on the procedure declaration for a profile access procedure, see `MyCMProfileAccessProc` (page 348). The ColorSync Manager calls your procedure when the profile is created, initialized, opened, read, updated, or closed, passing a command constant that specifies the current command. Your profile access procedure must be able to respond to each of the following command constants, which are described in “Profile Access Procedure Operation Codes” (page 395):

```
enum {
    cmOpenReadAccess    = 1,
    cmOpenWriteAccess   = 2,
    cmReadAccess        = 3,
    cmWriteAccess       = 4,
    cmCloseAccess       = 5,
    cmCreateNewAccess   = 6,
    cmAbortWriteAccess  = 7,
    cmBeginAccess       = 8,
    cmEndAccess         = 9
};
```

The profile access procedure shown in Listing 3-20, `MyCMProfileAccessProc`, consists of a single switch statement, which calls the appropriate routine based on the value of the `command` parameter. Each of the nine routines called by `MyCMProfileAccessProc` is described and listed in the sections that follow Listing 3-20, and each refers back to Listing 3-20.

Listing 3-20 Responding to a procedure-based profile command

```
pascal OSErr MyCMProfileAccessProc (long command,
                                    long offset,
                                    long *sizePtr,
                                    void *dataPtr,
                                    void *refConPtr)
{
    OSErr   theErr = noErr;
    switch (command)
    {
        case cmBeginAccess:
            theErr = DoBeginAccess(refConPtr);
            break;

        case cmCreateNewAccess:
            theErr = DoCreateNewAccess(refConPtr);
            break;

        case cmOpenReadAccess:
            theErr = DoOpenReadAccess(refConPtr);
            break;
```

```
        case cmOpenWriteAccess:
            theErr = DoOpenWriteAccess(sizePtr, refConPtr);
            break;

        case cmReadAccess:
            theErr = DoReadAccess(offset, sizePtr, dataPtr, refConPtr);
            break;

        case cmWriteAccess:
            theErr = DoWriteAccess(offset, sizePtr, dataPtr, refConPtr);
            break;

        case cmCloseAccess:
            theErr = DoCloseAccess(refConPtr);
            break;

        case cmAbortWriteAccess:
            theErr = DoAbortWriteAccess(refConPtr);
            break;

        case cmEndAccess:
            theErr = DoEndAccess(refConPtr);
            break;

        default:
            theErr = paramErr;
            break;
    }

    return theErr;
}
```

Note that the `MyCMProfileAccessProc` routine passes its parameter data as necessary to the routines it calls. The parameters have the following values:

`command` A command value indicating the operation to perform. The possible values for command constants are shown elsewhere in this section.

`offset` For read and write operations, the offset from the beginning of the profile at which to read or write data.

`size`	For the `cmReadAccess` and `cmWriteAccess` command constants, a pointer to a value indicating the number of bytes to read or write; for the `cmOpenWriteAccess` command, the total size of the profile. On output after reading or writing, the actual number of bytes read or written.
`data`	A pointer to a buffer containing data to read or write. On output, for a read operation, contains the data that was read.
`refConPtr`	A reference constant pointer that can store private data for the `MyCMProfileAccessProc` procedure. For example, Listing 3-18 (page 152) shows how to set up a location structure for procedure access to a resource-based profile. That routine sets the location structure's `refCon` field to a pointer to a `MyResourceLocRec` structure, which is described in "Defining a Data Structure for a Resource-Based Profile" (page 150). That same structure pointer is passed to the `MyCMProfileAccessProc` routine in the `refConPtr` parameter, and provides access to all the stored information about the resource location.

Handling the Begin Access Command

When your application calls the `CMOpenProfile` (page 222) routine, specifying as a location a procedure-based profile, the ColorSync Manager invokes your specified profile access procedure with the `cmBeginAccess` command. This gives your procedure an opportunity to perform any required initialization or validation tasks, such as determining whether the data pointed to by the `refcon` parameter is valid. If your procedure returns an error (any value except `noErr`), the ColorSync Manager will not call your profile access procedure again.

For the `cmBeginAccess` command, the sample profile access procedure shown in Listing 3-20 calls the `DoBeginAccess` routine, shown in Listing 3-21. `DoBeginAccess` interprets the `refcon` parameter as a `MyResourceLocPtr` type. If the parameter does not have a resource type of `kProcResourceType`, `DoBeginAccess` returns an invalid profile error, which effectively cancels the procedure-based profile access.

Listing 3-21 Handling the begin access command

```
static OSErr DoBeginAccess (void *refcon)
{
    OSErr               theErr;
    MyResourceLocPtr    resourceInfo = refcon;

    resourceInfo->resFileRef = 0;

    if (resourceInfo->resType != kProcResourceType)
        theErr = cmInvalidProfileLocation;
    else
        theErr = noErr;

    return theErr;
}
```

Handling the Create New Access Command

When your application calls the `CMCopyProfile` (page 229) or `CMUpdateProfile` (page 226) routine, specifying as a location a procedure-based profile, the ColorSync Manager invokes the specified profile access procedure with the `cmBeginAccess` command, as described in "Handling the Begin Access Command" (page 156).

If your profile access procedure returns without error, ColorSync calls the procedure again with the `cmCreateNewAccess` command. Your procedure should create a new data stream for the actual physical location of the profile. The size of the profile is not known at this point.

For the `cmCreateNewAccess` command, the sample profile access procedure shown in Listing 3-20 calls the `DoCreateNewAccess` routine. `DoCreateNewAccess` interprets the `refcon` parameter as a `MyResourceLocPtr` type, and calls the Toolbox routine `FSpCreateResFile` to create an empty resource fork based on the file specification provided by the `MyResourceLocPtr` type. If the resource fork does not already exist and cannot be created, `DoCreateNewAccess` returns an error.

Note that for this example, the file type for a resource-based profile was chosen arbitrarily to be `'rprf'`.

Listing 3-22 Handling the create new access command

```
OSErr DoCreateNewAccess (void *refcon)
{
    OSErr               theErr;
    MyResourceLocPtr    resourceInfo = refcon;

    FSpCreateResFile(&(resourceInfo->resFileSpec), '????', 'rprf', 0);
    theErr = ResError();
    if (theErr == dupFNErr)
        theErr = noErr;

    return theErr;
}
```

Handling the Open Read Access Command

When your application calls a ColorSync Manager routine to read information from a procedure-based profile, the ColorSync Manager first calls your profile access procedure with the `cmOpenReadAccess` command. Then it calls your profile access routine once for each read session. The sample profile access procedure shown in Listing 3-20 calls the `DoOpenReadAccess` routine.

The `DoOpenReadAccess` routine shown in Listing 3-23 uses information from the `refcon` parameter, interpreted as type `MyResourceLocPtr`, to open the resource fork for the resource-based profile with read permission. If it can open the resource file, `DoOpenReadAccess` then attempts to load the profile resource.

The `DoOpenReadAccess` routine shows good citizenship by saving the current resource file before performing its operations and restoring the resource file afterward.

Listing 3-23 Handling the open read access command

```
static OSErr DoOpenReadAccess (void *refcon)
{
    OSErr               theErr;
    MyResourceLocPtr    resourceInfo = refcon;
    short               currentResFile;
```

```
    /* Save current resource file. */
    currentResFile = CurResFile();

    /* Open the file's resource fork. */
    resourceInfo->resFileRef = FSpOpenResFile(&(resourceInfo->resFileSpec), fsRdPerm);
    theErr = ResError();

    /* Get the resource handle, but don't force it to be loaded into memory. */
    if (!theErr)
    {
        SetResLoad(false);
        resourceInfo->resHandle = GetResource(resourceInfo->resType,
                                              resourceInfo->resID);
        theErr = ResError();
        SetResLoad(true);
    }

    /* Restore previous resource file. */
    UseResFile(currentResFile);

    return theErr;
}
```

Handling the Open Write Access Command

When your application calls the `CMUpdateProfile` (page 226) routine to update a procedure-based profile or the `CMCopyProfile` (page 229) routine to copy a profile, the ColorSync Manager calls your profile access procedure with the `cmOpenWriteAccess` command. The sample profile access procedure shown in Listing 3-20 calls the `DoOpenWriteAccess` routine.

The `DoOpenWriteAccess` routine shown in Listing 3-24 uses information from the `refcon` parameter, interpreted as type `MyResourceLocPtr`, to open the resource fork for the resource-based profile with read / write permission. If it can open the resource file, `DoOpenWriteAccess` then attempts to open the specified profile resource. If it can't open the resource, `DoOpenWriteAccess` creates a new resource. It then sets the size of the resource based on the passed `setProfileSize` pointer value and updates the resource file.

The `DoOpenWriteAccess` routine shows good citizenship by saving the current resource file before performing its operations and restoring the resource file afterward.

Note
If the `cmOpenWriteAccess` command succeeds, the ColorSync Manager guarantees an eventual call to the profile access procedure with the `cmCloseAccess` command, possibly after multiple `cmWriteAccess` commands, and possibly after a `cmAbortWriteAccess` command. ◆

Listing 3-24 Handling the open write access command

```
static OSErr DoOpenWriteAccess (long *setProfileSize, void *refcon)
{
    OSErr               theErr;
    MyResourceLocPtr    resourceInfo = refcon;
    Size                resourceSize;
    short               currentResFile;

    /* Save current resource file. */
    currentResFile = CurResFile();

    /* Open the file's resource fork. */
    resourceInfo->resFileRef = FSpOpenResFile(&(resourceInfo->resFileSpec),
                                              fsRdWrPerm);
    theErr = ResError();

    /* Get the resource handle, but don't force it to be loaded into memory. */
    if (!theErr)
    {
        SetResLoad(false);
        resourceInfo->resHandle = GetResource(resourceInfo->resType,
                                              resourceInfo->resID);
        theErr = ResError();
        SetResLoad(true);
    }

    /* Call GetResourceSizeOnDisk to see if resource is already there. */
    if (!theErr)
    {
        /* Get size of the resource. */
        resourceSize = GetResourceSizeOnDisk(resourceInfo->resHandle);
        theErr = ResError();
```

```
}

/* If the above call to GetResourceSizeOnDisk returns resNotFound,
    then we need to create a new resource */
if (theErr == resNotFound)
{
    /* Allocate a temporary handle just so that we can call AddResource. */
    resourceInfo->resHandle = NewHandle(sizeof(long));
    theErr = MemError();

    /* Add resource to the file and release the temp handle. */
    if (!theErr)
    {
        AddResource(resourceInfo->resHandle, resourceInfo->resType,
                    resourceInfo->resID, resourceInfo->resName);
        theErr = ResError();
        ReleaseResource(resourceInfo->resHandle);
    }

/* Get the resource handle, but don't force it to be loaded into memory. */
    if (!theErr)
    {
        SetResLoad(false);
        resourceInfo->resHandle = GetResource(resourceInfo->resType,
                                              resourceInfo->resID);
        theErr = ResError();
        SetResLoad(true);
    }
}

/* Change the resource size to fit the profile. */
if (!theErr)
{
    SetResourceSize(resourceInfo->resHandle, *setProfileSize);
    theErr = ResError();
}

/* Force an update of the resource file. */
if (!theErr)
{
    UpdateResFile(resourceInfo->resFileRef);
```

```
        theErr = ResError();
    }

    /* Restore previous resource file. */
    UseResFile(currentResFile);

    return theErr;
}
```

Handling the Read Access Command

When your application calls a ColorSync Manager routine to read information from a procedure-based profile, the ColorSync Manager first calls your profile access procedure with the `cmOpenReadAccess` command, as described in "Handling the Open Read Access Command" (page 158). Your profile access routine can be called with the `cmReadAccess` command at any time after the `cmOpenReadAccess` command is called. When the sample profile access procedure shown in Listing 3-20 receives the `cmReadAccess` command, it calls the `DoReadAccess` routine.

The `DoReadAccess` routine shown in Listing 3-25 uses the `refcon` parameter, interpreted as type `MyResourceLocPtr`, to get a resource handle for the resource-based profile. From other parameters, it gets values for the offset at which to start reading, the number of bytes to read, and a pointer to a buffer in which to store the data that it reads. It then calls the Toolbox routine `ReadPartialResource` to do the actual reading.

If an error occurs while reading, `DoReadAccess` returns the error.

Listing 3-25 Handling the read access command

```
static OSErr DoReadAccess ( long offset,
                            long *sizePtr,
                            void *dataPtr,
                            void *refcon)
{
    OSErr               theErr;
    MyResourceLocPtr    resourceInfo = refcon;

    ReadPartialResource(resourceInfo->resHandle,
                            offset, dataPtr, *sizePtr);
```

```
    theErr = ResError();

    return theErr;
}
```

Handling the Write Access Command

When your application calls the `CMUpdateProfile` (page 226) routine to update a procedure-based profile, the ColorSync Manager first calls your profile access procedure with the `cmOpenWriteAccess` command. The `DoOpenWriteAccess` routine shown in Listing 3-24 performs certain operations to prepare to write a resource-based profile.

Your profile access routine can be called with the `cmWriteAccess` command at any time after the `cmOpenWriteAccess` command is called. When the sample profile access procedure shown in Listing 3-20 receives the `cmWriteAccess` command, it calls the `DoWriteAccess` routine.

The `DoWriteAccess` routine shown in Listing 3-26 uses the `refcon` parameter, interpreted as type `MyResourceLocPtr`, to get a resource handle for the resource-based profile. From other parameters, it gets values for the offset at which to start writing, the number of bytes to write, and a pointer to a buffer from which to get the data that it writes. It then calls the Toolbox routine `WritePartialResource` to do the actual writing.

If an error occurs while writing, `DoWriteAccess` returns the error.

Note
After ColorSync calls the profile access procedure with the `cmWriteAccess` command, ColorSync is guaranteed to eventually call the profile access procedure with the `cmCloseAccess` command—possibly after additional calls with the `cmWriteAccess` command, and possibly after a call with the `cmAbortWriteAccess` command. ◆

Listing 3-26 Handling the write access command

```
static OSErr DoWriteAccess (long offset,
                            long *sizePtr,
                            void *dataPtr,
                            void *refcon)
```

```
{
    OSErr               theErr;
    MyResourceLocPtr    resourceInfo = refcon;

    WritePartialResource(resourceInfo->resHandle,
                         offset, dataPtr, *sizePtr);
    theErr = ResError();

    return theErr;
}
```

Handling the Close Access Command

The ColorSync Manager calls your profile access procedure with the `cmCloseAccess` command to indicate that reading or writing is finished for the moment. A `cmCloseAccess` command can be followed by a `cmOpenReadAccess` command to begin reading again, a `cmOpenWriteAccess` command to begin writing again, or a `cmEndAccess` command to terminate the procedure-based profile access.

The sample profile access procedure shown in Listing 3-20 calls the `DoCloseAccess` routine.

The `DoCloseAccess` routine shown in Listing 3-27 uses information from the `refcon` parameter, interpreted as type `MyResourceLocPtr`, to close and update the resource file for the resource-based profile. If `DoCloseAccess` is unsuccessful, it returns an error value.

Listing 3-27 Handling the close access command

```
static OSErr DoCloseAccess (void *refcon)
{
    OSErr               theErr;
    MyResourceLocPtr    resourceInfo = refcon;

    /* Close and update resource file. */
    if (resourceInfo->resFileRef)
    {
        CloseResFile(resourceInfo->resFileRef);
        theErr = ResError();
        resourceInfo->resFileRef = 0;
```

```
    }
    else theErr = paramErr;

    return theErr;
}
```

Handling the Abort Write Access Command

If an error occurs between a `cmOpenWriteAccess` command and a `cmCloseAccess` command, the ColorSync Manager calls your profile access procedure with the `cmAbortWriteAccess` command. This allows your access procedure to perform any cleanup necessary for the partially written profile.

For the `cmAbortWriteAccess` command, the sample profile access procedure shown in Listing 3-20 calls the `DoAbortWriteAccess` routine.

The `DoAbortWriteAccess` routine shown in Listing 3-28 uses information from the `refcon` parameter, interpreted as type `MyResourceLocPtr`, to call the Toolbox routine `RemoveResource` to delete the partially written resource. If `DoAbortWriteAccess` is unsuccessful, it returns an error value.

Note
The ColorSync Manager will call your profile access procedure with the `cmCloseAccess` command after a `cmAbortWriteAccess` command. ◆

Listing 3-28 Handling the abort write access command

```
static OSErr DoAbortWriteAccess (void *refcon)
{
    OSErr               theErr;
    MyResourceLocPtr    resourceInfo = refcon;

    /* Delete the resource that we started. */
    if (resourceInfo->resHandle)
    {
        RemoveResource(resourceInfo->resHandle);
        theErr = ResError();
    }
    else theErr = paramErr;
```

```
    return theErr;
}
```

Handling the End Access Command

When access to a procedure-based profile is complete, the ColorSync Manager calls your profile access procedure with the `cmEndAccess` command. This allows your procedure to do any final cleanup, such as freeing memory allocated by the procedure.

For the `cmEndAccess` command, the sample profile access procedure shown in Listing 3-20 calls the `DoEndAccess` routine. Because there is no additional memory to free or other cleanup to take care of, the `DoEndAccess` routine shown in Listing 3-29 does nothing.

Note
The `MyCreateProcedureProfileAccess` routine, shown in Listing 3-18, *does* allocate memory, which is freed by a call to the `MyDisposeProcedureProfileAccess` routine, shown in Listing 3-19. Your application calls the `MyCreateProcedureProfileAccess` routine before calling a ColorSync Manager routine such as `CMCopyProfile` with a procedure-based profile. After the copy is complete, your application calls the `MyDisposeProcedureProfileAccess` routine to perform any necessary deallocation. ◆

Listing 3-29 Handling the end access command

```
pascal OSErr DoEndAccess (void *refcon)
{
    OSErr   theErr = noErr;

    return theErr;
}
```

Summary of the ColorSync Manager

This section provides a quick-reference summary of the functions, data types, and constants that make up the ColorSync Manager programming interface.

Functions

Accessing Profiles

```
pascal CMError CMOpenProfile      (CMProfileRef *prof,
                                   const CMProfileLocation *theProfile);

pascal CMError CMCloseProfile     (CMProfileRef prof);

pascal CMError CMProfileModified  (CMProfileRef prof,
                                   Boolean *modified);

pascal CMError CMUpdateProfile    (CMProfileRef prof);

pascal CMError CMNewProfile       (CMProfileRef *prof,
                                   const CMProfileLocation *theProfile);

pascal CMError CMCopyProfile      (CMProfileRef *targetProf,
                                   const CMProfileLocation *targetLocation,
                                   CMProfileRef prof);

pascal CMError CMCloneProfileRef  (CMProfileRef prof);

pascal CMError CMGetProfileRefCount (
                                   CMProfileRef prof,
                                   long *count);

/* NCMGetProfileLocation is new in ColorSync 2.5 */

pascal CMError NCMGetProfileLocation (
                                   CMProfileRef prof,
                                   CMProfileLocation * profLoc,
                                   unsigned long * locationSize);
```

```
/* CMGetProfileLocation is not recommended in ColorSync 2.5 */

pascal CMError CMGetProfileLocation (
                                    CMProfileRef prof,
                                    CMProfileLocation *theProfile);

pascal CMError CMValidateProfile  (CMProfileRef prof,
                                    Boolean *valid,
                                    Boolean *preferredCMMnotfound);

/* Use of CMFlattenProfile is changed in ColorSync 2.5 */

pascal CMError CMFlattenProfile (CMProfileRef prof,
                                    unsigned long flags,
                                    CMFlattenUPP proc,
                                    void *refCon,
                                    Boolean *preferredCMMnotfound);

/* Use of CMUnflattenProfile is changed in ColorSync 2.5 */

pascal CMError CMUnflattenProfile(FSSpec *resultFileSpec,
                                    CMFlattenUPP proc,
                                    void *refCon,
                                    Boolean *preferredCMMnotfound);
```

Accessing Profile Elements

```
pascal CMError CMProfileElementExists (
                                    CMProfileRef prof,
                                    OSType tag,
                                    Boolean *found);

pascal CMError CMCountProfileElements (
                                    CMProfileRef prof,
                                    unsigned long *elementCount);

pascal CMError CMGetProfileElement (
                                    CMProfileRef prof,
                                    OSType tag,
                                    unsigned long *elementSize,
                                    void *elementData);

pascal CMError CMGetProfileHeader (CMProfileRef prof,
                                    CMAppleProfileHeader *header);
```

```
pascal CMError CMGetPartialProfileElement (
                                    CMProfileRef prof,
                                    OSType tag,
                                    unsigned long offset,
                                    unsigned long *byteCount,
                                    void *elementData);

pascal CMError CMSetProfileElementSize (
                                    CMProfileRef prof,
                                    OSType tag,
                                    unsigned long elementSize);

pascal CMError CMGetIndProfileElementInfo (
                                    CMProfileRef prof,
                                    unsigned long index,
                                    OSType *tag,
                                    unsigned long *elementSize,
                                    Boolean *refs);

pascal CMError CMGetIndProfileElement (
                                    CMProfileRef prof,
                                    unsigned long index,
                                    unsigned long *elementSize,
                                    void *elementData);

pascal CMError CMSetPartialProfileElement (
                                    CMProfileRef prof,
                                    OSType tag,
                                    unsigned long offset,
                                    unsigned long byteCount,
                                    void *elementData);

pascal CMError CMSetProfileElement (
                                    CMProfileRef prof,
                                    OSType tag,
                                    unsigned long elementSize,
                                    void *elementData);

pascal CMError CMSetProfileHeader(CMProfileRef prof,
                                    const CMAppleProfileHeader *header);

pascal CMError CMSetProfileElementReference (
                                    CMProfileRef prof,
                                    OSType elementTag,
                                    OSType referenceTag);
```

```
pascal CMError CMRemoveProfileElement (
                                    CMProfileRef prof, OSType tag);

pascal CMError CMGetScriptProfileDescription (
                                    CMProfileRef prof,
                                    Str255 name,
                                    ScriptCode *code);
```

Accessing Named Color Profile Values

```
pascal CMError CMGetNamedColorInfo (
                                    CMProfileRef prof,
                                    unsigned long *deviceChannels,
                                    OSType *deviceColorSpace,
                                    OSType *PCSColorSpace,
                                    unsigned long *count,
                                    StringPtr prefix,
                                    StringPtr suffix);

pascal CMError CMGetNamedColorValue (
                                    CMProfileRef prof,
                                    StringPtr name,
                                    CMColor *deviceColor,
                                    CMColor *PCSColor)

pascal CMError CMGetIndNamedColorValue (
                                    CMProfileRef prof,
                                    unsigned long index,
                                    CMColor *deviceColor,
                                    CMColor *PCSColor);

pascal CMError CMGetNamedColorIndex (
                                    CMProfileRef prof,
                                    StringPtr name,
                                    unsigned long *index);

pascal CMError CMGetNamedColorName (
                                    CMProfileRef prof,
                                    unsigned long index,
                                    StringPtr name)
```

Matching Colors Using General Purpose Functions

```
/* Use of NCWNewColorWorld is changed in ColorSync 2.5 */

pascal CMError NCWNewColorWorld(CMWorldRef *cw,
                                CMProfileRef src,
                                CMProfileRef dst);

/* Use of CWConcatColorWorld is changed in ColorSync 2.5 */

pascal CMError CWConcatColorWorld (CMWorldRef *cw,
                                CMConcatProfileSet *profileSet);

pascal CMError CWNewLinkProfile(CMProfileRef *prof,
                                const CMProfileLocation *targetLocation,
                                CMConcatProfileSet *profileSet);

/* Use of CMGetCWInfo is changed in ColorSync 2.5 */

pascal CMError CMGetCWInfo (CMWorldRef cw,
                                CMCWInfoRecord *info);

pascal void CWDisposeColorWorld (CMWorldRef cw);

pascal CMError CWMatchPixMap (CMWorldRef cw,
                                PixMap *myPixMap,
                                CMBitmapCallBackUPP progressProc,
                                void *refCon);

pascal CMError CWCheckPixMap (CMWorldRef cw,
                                PixMap *myPixMap,
                                CMBitmapCallBackUPP progressProc,
                                void *refCon,
                                BitMap *resultBitMap);

pascal CMError CWMatchBitmap (CMWorldRef cw,
                                CMBitMap *bitMap,
                                CMBitmapCallBackUPP progressProc,
                                void *refCon,
                                CMBitMap *matchedBitMap);

pascal CMError CWCheckBitmap (CMWorldRef cw,
                                const CMBitMap *bitMap,
                                CMBitmapCallBackUPP progressProc,
                                void *refCon,
                                CMBitMap *resultBitMap);
```

```
pascal CMError CWMatchColors (CMWorldRef cw,
                                  CMColor *myColors,
                                  unsigned long count);

pascal CMError CWCheckColors (CMWorldRef cw,
                                  CMColor *myColors,
                                  unsigned long count,
                                  long *result);
```

Matching Colors Using QuickDraw-Specific Functions

```
/* Use of NCMBeginMatching is changed in ColorSync 2.5 */

pascal CMError NCMBeginMatching   (CMProfileRef src,
                                  CMProfileRef dst,
                                  CMMatchRef *myRef);

pascal void CMEndMatching         (CMMatchRef myRef);

pascal void CMEnableMatchingComment (
                                  Boolean enableIt);

/* Use of NCMDrawMatchedPicture is changed in ColorSync 2.5 */

pascal void NCMDrawMatchedPicture (PicHandle myPicture,
                                  CMProfileRef dst,
                                  Rect *myRect);
```

Embedding Profile Information in Pictures

```
pascal CMError NCMUseProfileComment (
                                  CMProfileRef prof,
                                  unsigned long flags);
```

Getting the Preferred CMM

```
/* CMGetPreferredCMM is new in ColorSync 2.5 */

pascal CMError CMGetPreferredCMM (
                                  OSType *cmmType,
                                  Boolean *preferredCMMnotfound)
```

Getting and Setting the System Profile File

```
/* Use of CMGetSystemProfile is changed in ColorSync 2.5 */

pascal CMError CMGetSystemProfile (CMProfileRef *prof);

/* Use of CMSetSystemProfile is changed in ColorSync 2.5 */

pascal CMError CMSetSystemProfile (const FSSpec *profileFileSpec);
```

Getting and Setting Default Profiles by Color Space

```
/* CMGetDefaultProfileBySpace is new in ColorSync 2.5 */

pascal CMError CMGetDefaultProfileBySpace(
                                OSType dataColorSpace,
                                CMProfileRef * prof);

/* CMSetDefaultProfileBySpace is new in ColorSync 2.5 */

pascal CMError CMSetDefaultProfileBySpace (
                                OSType dataColorSpace,
                                CMProfileRef prof);
```

Getting and Setting Monitor Profiles by AVID

```
/* CMGetProfileByAVID is new in ColorSync 2.5 */

pascal CMError CMGetProfileByAVID (
                                AVIDType theAVID,
                                CMProfileRef *prof);

/* CMSetProfileByAVID is new in ColorSync 2.5 */

pascal CMError CMSetProfileByAVID (
                                AVIDType theAVID,
                                CMProfileRef prof);
```

Locating the ColorSync Profiles Folder

```
pascal CMError CMGetColorSyncFolderSpec (
                                short vRefNum,
                                Boolean createFolder,
                                short *foundVRefNum,
                                long *foundDirID);
```

Searching for Profiles With ColorSync 2.5

```
/* CMIterateColorSyncFolder is new in ColorSync 2.5 */

pascal CMError CMIterateColorSyncFolder (
                                    CMProfileIterateUPP proc,
                                    unsigned long * seed,
                                    unsigned long * count,
                                    void * refCon);
```

Searching for Profiles Prior to ColorSync 2.5

```
/* The functions in this group are not recommended in ColorSync 2.5 */

pascal CMError CMNewProfileSearch (CMSearchRecord *searchSpec,
                                    void *refCon,
                                    unsigned long *count,
                                    CMProfileSearchRef *searchResult);

pascal CMError CMUpdateProfileSearch (
                                    CMProfileSearchRef search,
                                    void *refCon,
                                    unsigned long *count);

pascal void CMDisposeProfileSearch (
                                    CMProfileSearchRef search);

pascal CMError CMSearchGetIndProfile (
                                    CMProfileSearchRef search,
                                    unsigned long index,
                                    CMProfileRef *prof);

pascal CMError CMSearchGetIndProfileFileSpec (
                                    CMProfileSearchRef search,
                                    unsigned long index,
                                    FSSpec *profileFile);
```

Searching For a Profile by Profile Identifier

```
pascal CMError CMNewProfileSearch (CMSearchRecord *searchSpec,
                                    void *refCon,
                                    unsigned long *count,
                                    CMProfileSearchRef *searchResult);
```

```
pascal CMError CMProfileIdentifierListSearch (
                                    CMProfileIdentifierPtr ident,
                                    CMProfileRef *profileList,
                                    unsigned long listSize,
                                    unsigned long *matchedCount,
                                    CMProfileRef *matchedList);
```

Converting Between Color Spaces

```
pascal ComponentResult CMXYZToLab (ComponentInstance ci,
                                    const CMColor *src,
                                    const CMXYZColor *white,
                                    CMColor *dst,
                                    unsigned long count);

pascal ComponentResult CMLabToXYZ (ComponentInstance ci,
                                    const CMColor *src,
                                    const CMXYZColor *white,
                                    CMColor *dst,
                                    unsigned long count);

pascal ComponentResult CMXYZToLuv (ComponentInstance ci,
                                    const CMColor *src,
                                    const CMXYZColor *white,
                                    CMColor *dst,
                                    unsigned long count);

pascal ComponentResult CMLuvToXYZ (ComponentInstance ci,
                                    const CMColor *src,
                                    const CMXYZColor *white,
                                    CMColor *dst,
                                    unsigned long count);

pascal ComponentResult CMXYZToYxy (ComponentInstance ci,
                                    const CMColor *src,
                                    CMColor *dst,
                                    unsigned long count);

pascal ComponentResult CMYxyToXYZ (ComponentInstance ci,
                                    const CMColor *src,
                                    CMColor *dst,
                                    unsigned long count);
```

```
pascal ComponentResult CMXYZToFixedXYZ (
                                ComponentInstance ci,
                                const CMXYZColor *src,
                                CMFixedXYZColor *dst,
                                unsigned long count);

pascal ComponentResult CMFixedXYZToXYZ (
                                ComponentInstance ci,
                                const CMFixedXYZColor *src,
                                CMXYZColor *dst,
                                unsigned long count);

pascal ComponentResult CMRGBToHLS (ComponentInstance ci,
                                const CMColor *src,
                                CMColor *dst,
                                unsigned long count);

pascal ComponentResult CMHLSToRGB (ComponentInstance ci,
                                const CMColor *src,
                                CMColor *dst,
                                unsigned long count);

pascal ComponentResult CMRGBToHSV (ComponentInstance ci,
                                const CMColor *src,
                                CMColor *dst,
                                unsigned long count);

pascal ComponentResult CMHSVToRGB (ComponentInstance ci,
                                const CMColor *src,
                                CMColor *dst,
                                unsigned long count);

pascal ComponentResult CMRGBToGray (
                                ComponentInstance ci,
                                const CMColor *src,
                                CMColor *dst,
                                unsigned long count);
```

Color-Matching With PostScript™ Devices

```
pascal CMError CMGetPS2ColorSpace (CMProfileRef srcProf,
                                unsigned long flags,
                                CMFlattenUPP proc,
                                void *refCon,
                                Boolean *preferredCMMnotfound);
```

```
pascal CMError CMGetPS2ColorRenderingIntent (
                                    CMProfileRef srcProf,
                                    unsigned long flags,
                                    CMFlattenUPP proc,
                                    void *refCon,
                                    Boolean *preferredCMMnotfound);

pascal CMError CMGetPS2ColorRendering (
                                    CMProfileRef srcProf,
                                    CMProfileRef dstProf,
                                    unsigned long flags,
                                    CMFlattenUPP proc,
                                    void *refCon,
                                    Boolean *preferredCMMnotfound);

extern pascal CMError CMGetPS2ColorRenderingVMSize (
                                    CMProfileRef srcProf,
                                    CMProfileRef dstProf,
                                    unsigned long *vmSize,
                                    Boolean *preferredCMMnotfound);
```

Converting 2.x Profiles to 1.0 Format

```
pascal CMError CMConvertProfile2to1 (
                                    CMProfileRef profv2,
                                    CMProfileHandle *profv1);
```

Application-Supplied Functions for the ColorSync Manager

```
/* MyProfileIterateProc is new in ColorSync 2.5 */

pascal OSErr MyProfileIterateProc (
                                    CMProfileIterateData *iterateData,
                                    void *refCon);

pascal OSErr MyColorSyncDataTransfer (
                                    long command,
                                    long *size,
                                    void *data,
                                    void *refCon);

pascal Boolean MyCMBitmapCallBackProc (
                                    long progress,
                                    void *refCon);
```

```
pascal Boolean MyCMProfileFilterProc (
                                    CMProfileRef prof,
                                    void *refCon);
```

Data Structures

```
/* date and time structure */
struct CMDateTime {
    unsigned short  year;
    unsigned short  month;
    unsigned short  dayOfTheMonth;
    unsigned short  hours;
    unsigned short  minutes;
    unsigned short  seconds;
};

/* ColorSync 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;
};

/* ColorSync Manager profile 2.x header structure */
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];
};

/* Apple profile header */
union CMAppleProfileHeader {
    CMHeader        cm1;
    CM2Header       cm2;
};

/* profile reference abstract data type */
typedef struct OpaqueCMProfileRef *CMProfileRef;

/* profile identifier structure */
struct CMProfileIdentifier {
    CM2Header           profileHeader;
    CMDateTime          calibrationDate;
    unsigned long       ASCIIProfileDescriptionLen;
    char                ASCIIProfileDescription[1]; /* variable length */
};

/* profile location union */
union CMProfLoc {
    CMFileLocation          fileLoc;
    CMHandleLocation        handleLoc;
    CMPtrLocation           ptrLoc;
    CMProcedureLocation     procLoc;
};

/* profile location structure */
struct CMProfileLocation{
    short       locType;
    CMProfLoc   u;
};
```

```
/* file specification for file-based profiles */
struct CMFileLocation {
    FSSpec   spec;
};

/* handle specification for memory-based profiles */
struct CMHandleLocation {
    Handle   h;
};

/* pointer specification for memory-based profiles */
struct CMPtrLocation {
    Ptr  p;
};

/* procedure specification for procedure access profiles */
struct CMProcedureLocation {
    CMProfileAccessUPP      proc;
    void                    *refCon;
};

/* CMProfileIterateProcPtr and CMProfileIterateData are new in ColorSync 2.5 */

/* Cached profile searching */
pascal OSErr (*CMProfileIterateProcPtr )
                    (CMProfileIterateData *iterateData,
                     void *refCon);

struct CMProfileIterateData {
    unsigned long   dataVersion;    /* cmProfileIterateDataVersion1 */
    CM2Header       header;
    ScriptCode      code;
    Str255          name;
    CMProfileLocation location;
};
typedef struct CMProfileIterateData CMProfileIterateData;

/* Non-cached profile searching */
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;
};

/* profile search result reference abstract data type */
struct OpaqueCMProfileSearchRef *CMProfileSearchRef;

/* XYZ color-component values */
typedef unsigned short CMXYZComponent;

/* XYZ color value */
struct CMXYZColor {
    CMXYZComponent  X;
    CMXYZComponent  Y;
    CMXYZComponent  Z;
};

/* fixed XYZ color value */
struct CMFixedXYZColor {
    Fixed   X;
    Fixed   Y;
    Fixed   Z;
};

/* L*a*b* color value */
struct CMLabColor {
    unsigned short  L;
    unsigned short  a;
    unsigned short  b;
};

/* L*u*v* color value */
struct CMLuvColor {
    unsigned short  L;
    unsigned short  u;
    unsigned short  v;
};
```

```
/* Yxy color value */
struct CMYxyColor {
    unsigned short  capY;   /* 0..65535 maps to 0..1 */
    unsigned short  x;      /* 0..65535 maps to 0..1 */
    unsigned short  y;      /* 0..65535 maps to 0..1 */
};

/* RGB color value */
struct CMRGBColor {
    unsigned short  red;
    unsigned short  green;
    unsigned short  blue;
};

/* HLS color value */
struct CMHLSColor {
    unsigned short  hue;
    unsigned short  lightness;
    unsigned short  saturation;
};

/* HSV color value */
typedef struct CMHSVColor {
    unsigned short  hue;
    unsigned short  saturation;
    unsigned short  value;
};

/* CMYK color value */
struct CMCMYKColor {
    unsigned short  cyan;
    unsigned short  magenta;
    unsigned short  yellow;
    unsigned short  black;
};

/* CMY color value */
struct CMCMYColor {
    unsigned short  cyan;
    unsigned short  magenta;
    unsigned short  yellow;
};
```

```
/* HiFi color values */
struct CMMultichannel5Color {
    unsigned char   components[5];
};

struct CMMultichannel6Color {
    unsigned char   components[6];
};

struct CMMultichannel7Color {
    unsigned char   components[7];
};

struct CMMultichannel8Color {
    unsigned char   components[8];
};

/* gray color value */
struct CMGrayColor {
    unsigned short  gray;
};

/* named color value */
struct CMNamedColor {
    unsigned long namedColorIndex;  /* 0..a lot */
};

/* color union */
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;
    CMNamedColor            namedColor;
};

/* ColorSync Manager bitmap */
struct CMBitmap {
    char                *image;
    long                width;
    long                height;
    long                rowBytes;
    long                pixelSize;
    CMBitmapColorSpace  space;
    long                user1;
    long                user2;
};

/* QuickDraw-specific color-matching session reference abstract data type */
struct OpaqueCMMatchRef *CMMatchRef;

/* color world information record */
struct CMCWInfoRecord {
    unsigned long       cmmCount;
    CMMInfoRecord       cmmInfo[2];
};

/* color world reference abstract data type */
struct OpaqueCMWorldRef *CMWorldRef;

/* concatenated profile set structure */
struct CMConcatProfileSet {
    unsigned short      keyIndex;
    unsigned short      count;
    CMProfileRef        profileSet[1];
};

/* color management module (CMM) information record structure */
struct CMMInfoRecord {
    OSType      CMMType;
    long        CMMVersion;
};

/* The video card gamma data types are new in ColorSync 2.5 */
/* video card gamma type */
```

```
struct CMVideoCardGammaType
{
    OSType              typeDescriptor;
    unsigned long       reserved;
    CMVideoCardGamma    gamma;
};
typedef struct CMVideoCardGammaType CMVideoCardGammaType;

/* video card gamma table */
struct CMVideoCardGammaTable
{
    unsigned short      channels;
    unsigned short      entryCount;
    unsigned short      entrySize;
    char                data[1];
};
typedef struct CMVideoCardGammaTable CMVideoCardGammaTable;

/* video card gamma formula */
struct CMVideoCardGammaFormula {
    Fixed           redGamma;
    Fixed           redMin;
    Fixed           redMax;
    Fixed           greenGamma;
    Fixed           greenMin;
    Fixed           greenMax;
    Fixed           blueGamma;
    Fixed           blueMin;
    Fixed           blueMax;
};

/* video card gamma */
struct CMVideoCardGamma
{
    unsigned long                   tagType;
    union
    {
        CMVideoCardGammaTable       table;
        CMVideoCardGammaFormula     formula;
    }                               u;
};
typedef struct CMVideoCardGamma CMVideoCardGamma;
```

```
/* color matching while printing */
struct TEnableColorMatchingBlk {
    short       iOpCode;
    short       iError;
    long        lReserved;
    THPrint     hPrint;
    Boolean     fEnableIt;
    SInt8       filler;
};

/* PostScript color rendering dictionary (CRD) virtual memory size tag structure */
struct CMIntentCRDVMSize {
    long                rendering   Intent;
    unsigned long       VMSize;
};

struct CMPS2CRDVMSizeType {
    OSType  typeDescriptor;
    unsigned long   reserved;
    unsigned long   count;
    CMIntentCRDVMSize   intentCRD[1];
};
```

Constants

```
/* constants for profile location type */
enum {
    cmNoProfileBase         = 0,
    cmFileBasedProfile      = 1,
    cmHandleBasedProfile    = 2,
    cmPtrBasedProfile       = 3,
    cmProcedureBasedProfile = 4
};

/* commands for profile access procedure */
enum {
    cmOpenReadAccess    = 1,
    cmOpenWriteAccess   = 2,
    cmReadAccess        = 3,
    cmWriteAccess       = 4,
    cmCloseAccess       = 5,
```

```
    cmCreateNewAccess   = 6,
    cmAbortWriteAccess  = 7,
    cmBeginAccess       = 8,
    cmEndAccess         = 9
};

/* profile classes */

enum {
    cmInputClass        = 'scnr',
    cmDisplayClass      = 'mntr',
    cmOutputClass       = 'prtr',
    cmLinkClass         = 'link',
    cmAbstractClass     = 'abst',
    cmColorSpaceClass   = 'spac',
    cmNamedColorClass   = 'nmcl'
};

/* signature of ColorSync's default color management module (CMM) */
enum {
    kDefaultCMMSignature    = 'appl'
};

/* commands for calling the application-supplied MyColorSyncDataTransfer */
enum {
    openReadSpool = 1,
    openWriteSpool,
    readSpool,
    writeSpool,
    closeSpool
};
/* PostScript data formats */

enum {
    cmPS7bit    = 1,    /* data is 7-bit safe */
    cmPS8bit    = 2     /* data is 8-bit safe */
};

/* picture comment IDs for profiles and color matching */
enum {
    cmBeginProfile      = 220,
    cmEndProfile        = 221,
    cmEnableMatching    = 222,
```

```
    cmDisableMatching   = 223,
    cmComment           = 224
};

/* picture comment selectors for the cmComment ID */
enum {
    cmBeginProfileSel       = 0,
    cmContinueProfileSel    = 1,
    cmEndProfileSel         = 2,
    cmProfileIdentifierSel  = 3
};

/* color space signatures */
enum {
    cmXYZData   = 'XYZ ',
    cmLabData   = 'Lab ',
    cmLuvData   = 'Luv ',
    cmYxyData   = 'Yxy ',
    cmRGBData   = 'RGB ',
    cmGrayData  = 'GRAY',
    cmHSVData   = 'HSV ',
    cmHLSData   = 'HLS ',
    cmCMYKData  = 'CMYK',
    cmCMYData   = 'CMY ',
    cmMCH5Data  = 'MCH5',
    cmMCH6Data  = 'MCH6',
    cmMCH7Data  = 'MCH7',
    cmMCH8Data  = 'MCH8',
    cmNamedData     = 'NAME'
};

/* cm48_16ColorPacking and cm64_16ColorPacking were added in ColorSync version 2.5 */
/* color packing for color spaces */
enum {
    cmNoColorPacking        = 0x0000,
    cmAlphaSpace            = 0x0080,
    cmWord5ColorPacking     = 0x0500,
    cmLong8ColorPacking     = 0x0800,
    cmLong10ColorPacking    = 0x0a00,
    cmAlphaFirstPacking     = 0x1000,
    cmOneBitDirectPacking   = 0x0b00,
    cmAlphaLastPacking      = 0x0000,
```

```
    cm24_8ColorPacking      = 0x2100,
    cm32_8ColorPacking      = cmLong8ColorPacking,
    cm40_8ColorPacking      = 0x2200,
    cm48_8ColorPacking      = 0x2300,
    cm56_8ColorPacking      = 0x2400,
    cm64_8ColorPacking      = 0x2500,
    cm32_16ColorPacking     = 0x2600,
    cm32_32ColorPacking     = 0x2700,
    cm48_16ColorPacking     = 0x2900,
    cm64_16ColorPacking     = 0x2A00
};

/* cmRGBASpace and cmGrayASpace were moved to this enum from color space constants with
packing formats in ColorSync version 2.5 */
/* abstract color spaces */
enum {
    cmNoSpace           = 0,
    cmRGBSpace          = 1,
    cmCMYKSpace         = 2,
    cmHSVSpace          = 3,
    cmHLSSpace          = 4,
    cmYXYSpace          = 5,
    cmXYZSpace          = 6,
    cmLUVSpace          = 7,
    cmLABSpace          = 8,
    cmReservedSpace1    = 9,
    cmGraySpace         = 10,
    cmReservedSpace2    = 11,
    cmGamutResultSpace  = 12,
    cmNamedIndexedSpace = 16,
    cmMCFiveSpace       = 17,
    cmMCSixSpace        = 18,
    cmMCSevenSpace      = 19,
    cmMCEightSpace      = 20,
    cmRGBASpace         = cmRGBSpace + cmAlphaSpace,
    cmGrayASpace        = cmGraySpace + cmAlphaSpace
};

/* cmGray16Space, cmGrayA32Space, cmRGB48Space, cmCMYK64Space, and cmLAB48Space were
added in ColorSync version 2.5 */
/* color space constants with packing formats */
```

```
enum {
    cmGray16Space           = cmGraySpace,
    cmGrayA32Space          = cmGrayASpace,
    cmRGB16Space            = cmWord5ColorPacking + cmRGBSpace,
    cmRGB24Space            = cm24_8ColorPacking + cmRGBSpace,
    cmRGB32Space            = cm32_8ColorPacking + cmRGBSpace,
    cmRGB48Space            = cm48_16ColorPacking + cmRGBSpace,
    cmARGB32Space           = cm32_8ColorPacking + cmAlphaFirstPacking + cmRGBASpace,
    cmRGBA32Space           = cm32_8ColorPacking + cmAlphaFirstPacking + cmRGBASpace,
    cmCMYK32Space           = cm32_8ColorPacking + cmCMYKSpace,
    cmCMYK64Space           = cm64_16ColorPacking + cmCMYKSpace,
    cmHSV32Space            = cmLong10ColorPacking + cmHSVSpace,
    cmHLS32Space            = cmLong10ColorPacking + cmHLSSpace,
    cmYXY32Space            = cmLong10ColorPacking + cmYXYSpace,
    cmXYZ32Space            = cmLong10ColorPacking + cmXYZSpace,
    cmLUV32Space            = cmLong10ColorPacking + cmLUVSpace,
    cmLAB24Space            = cm24_8ColorPacking + cmLABSpace,
    cmLAB32Space            = cmLong10ColorPacking + cmLABSpace,
    cmLAB48Space            = cm48_16ColorPacking + cmLABSpace,
    cmGamutResult1Space     = cmOneBitDirectPacking + cmGamutResultSpace
    cmNamedIndexed32Space   = cm32_32ColorPacking + cmNamedIndexedSpace,
    cmMCFive8Space          = cm40_8ColorPacking + cmMCFiveSpace,
    cmMCSix8Space           = cm48_8ColorPacking + cmMCSixSpace,
    cmMCSeven8Space         = cm56_8ColorPacking + cmMCSevenSpace,
    cmMCEight8Space         = cm64_8ColorPacking + cmMCEightSpace
};

/* flag mask values for version 2.x profiles */
enum {
    cmICCReservedFlagsMask  = 0x0000FFFF,
    cmEmbeddedMask          = 0x00000001,
    cmEmbeddedUseMask       = 0x00000002,
    cmCMSReservedFlagsMask  = 0xFFFF0000,
    cmQualityMask           = 0x00030000,
    cmInterpolationMask     = 0x00040000,
    cmGamutCheckingMask     = 0x00080000
};

/* quality flag values for version 2.x profiles */
enum {
    cmNormalMode    = 0,
    cmDraftMode     = 1,
```

```
    cmBestMode      = 2
};
/* Several unused mask constants were removed for ColorSync version 2.5 */
/* device attribute values for version 2.x profiles */
enum {
    /* if bit 0 is 0 then reflective media, if 1 then transparent media */
    cmReflectiveTransparentMask = 0x00000001,
    /* if bit 1 is 0 then glossy media, if 1 then matte media*/
    cmGlossyMatteMask = 0x00000002
};

/* rendering intent values for version 2.x profiles */
enum {
    cmPerceptual            = 0,
    cmRelativeColorimetric  = 1,
    cmSaturation            = 2,
    cmAbsoluteColorimetric  = 3
};

/* defines for the CMSearchRecord.searchMask field */
enum {
    cmMatchAnyProfile               = 0x00000000,
    cmMatchProfileCMMType           = 0x00000001,
    cmMatchProfileClass             = 0x00000002,
    cmMatchDataColorSpace           = 0x00000004,
    cmMatchProfileConnectionSpace   = 0x00000008,
    cmMatchManufacturer             = 0x00000010,
    cmMatchModel                    = 0x00000020,
    cmMatchAttributes               = 0x00000040,
    cmMatchProfileFlags             = 0x00000080
};

/* The video card constants are new in ColorSync 2.5 */
/* Video card gamma tag. */
enum
{
    …,
    cmVideoCardGammaTag = FOUR_CHAR_CODE('vcgt')
};
```

```
/* Video card gamma tag type. */

enum
{
    cmSigVideoCardGammaType = FOUR_CHAR_CODE('vcgt')
};

/* Video card gamma storage type. */

enum
{
    cmVideoCardGammaTableType = 0,
    cmVideoCardGammaFormulaType = 1,
};

/* PrGeneral operation codes */
enum {
    enableColorMatchingOp   = 12,
    registerProfileOp       = 13
};

/* ColorSync Manager element tags and their signatures for version 1.0 profiles */

enum {
    cmCS1ChromTag   = 'chrm',
    cmCS1TRCTag     = 'trc ',
    cmCS1NameTag    = 'name',
    cmCS1CustTag    = 'cust'
};
```

CHAPTER 4

Developing ColorSync-Supportive Device Drivers

Contents

This section describes how you can use the ColorSync Manager to provide ColorSync-supportive device drivers for peripherals. It first describes how input, display, and output devices work with color profiles. It then discusses how devices interact with color management modules (CMMs). Next it describes what you must do to provide minimum ColorSync support, as well as how you can provide more extensive support. Finally, it uses a QuickDraw-based printer device driver to demonstrate some of the color-matching features a device driver can provide.

Read this section if your device driver for an input, display, or output device will support the ColorSync Manager and allow users to produce color-matched images.

Before you read this section, you should read "Introduction to ColorSync" (page 45) and "Introduction to Color and Color Management Systems" (page 25). These sections provide an overview of ColorSync, explain color theory and color management systems, and define key terms.

Although the features described here are commonly provided by printer device drivers, the code samples in "Developing ColorSync-Supportive Applications" (page 81) may also be of use in developing your device driver.

"ColorSync Reference for Applications and Drivers" (page 217) describes constants, data structures, functions, and result codes for ColorSync-supportive applications and device drivers.

"ColorSync Version Information" (page 525) describes the `Gestalt` information, shared library version numbers, CMM version numbers, and ColorSync header files you use with different versions of the ColorSync Manager. It also includes CPU and Mac OS system requirements.

About ColorSync-Supportive Device Driver Development

A device driver that supports ColorSync should provide at least one profile for the device. In addition, it can provide its own CMM, designed to perform the best possible color matching for the device. If you are creating your own CMM, you should read "Developing Color Management Modules" (page 429) and "ColorSync Reference for Color Management Modules" (page 467).

Devices and Their Profiles

To assess the way each device interprets color, color scientists and profile developers perform device characterizations. This process, which entails measuring the gamut of a device, yields a color profile for that device. For an overview of profiles, see “Profiles” (page 49).

Device profiles are of paramount importance to any color management system because they characterize the unique color behavior of each device and provide the data needed for color matching and color conversion. Device profiles are used by CMMs that perform the low-level calculations required to match colors from a source device to a destination device.

The ICC defines a device profile class for each of three types of devices:

- An input device, such as a scanner or a digital camera.
- A display device, such as a monitor or a liquid crystal display.
- An output device, such as a printer.

These classes are described in detail in “Profile Classes” (page 51).

Each device profile class has its own signature. The ColorSync constants for these signatures are described in “Profile Class” (page 396). You can create a device driver for any of the device classes. When you create a profile for your device, you specify the signature in the profile header’s `profileClass` field. For more information on profile headers, see “Profile Header” (page 351)

Whether you create a profile for your device or obtain one from a profile vendor, your device driver must provide at least one profile for its device. However, you can provide more than one profile for the same device to characterize different states. Although a printer that your device driver supports may have a number of profiles for different conditions, such as the use of foils or different grades of paper, all of its profiles will use the same profile signature, `cmOutputClass`.

The Profile Format and Its Cross-Platform Use

Device profiles follow the ICC profile format, an industry standard described in “Profiles” (page 49). You can provide a single profile or a set of profiles that can be used across different operating systems for the device your driver supports. The common profile format specified by the ICC allows end users to transparently move profiles and images with embedded profiles between different operating systems.

Note
The ICC publishes the *International Color Consortium Profile Format Specification*.To obtain a copy of the specification, visit the ICC Web site at <http://www.color.org/>. ◆

The profile structure is defined as a header followed by a tag table which, in turn, is followed by a series of tagged elements that your device driver can access randomly and individually. Using ColorSync Manager functions, you can read the profile header and modify its contents and you can get and set individual tags and their element data.

ColorSync Profile Format Version Numbers

This document uses "2.x" to refer to ColorSync profiles for ColorSync Manager version 2.0 and greater, as described in "ColorSync and ICC Profile Format Version Numbers" (page 50). Version 2.x profiles require more information and are larger than ColorSync 1.0 profiles, which were originally memory based. Because version 2.x profiles are larger, they are disk-based. The ICC profile format specification defines how profiles can be stored as disk files and how profiles can be embedded in common graphics file formats such as PICT and TIFF. The ColorSync Manager provides the `CMProfileLocation` (page 362) data structure to identify the location of a profile. It also provides functions you can use to embed a profile in or extract if from a PICT file, as described in "Embedding Profile Information in Pictures" (page 290).

Storing and Handling Device Profiles

Device profiles reside in the ColorSync Profiles folder, in pictures, or with device drivers. Files that contain profiles store profile data in the data fork and have a file type of `'prof'`.

By convention, profiles not embedded in image documents are stored in the ColorSync Profiles folder, as described in "Profile Search Locations" (page 55). You can store your profile files wherever you want, but if you want other drivers or applications to have access to them, you should store them in the ColorSync Profiles folder. Applications that perform soft proofing or gamut checking can use ColorSync Manager routines to search the folder for specific types of profiles to provide a pop-up menu or list to the user. If your profiles are not available, these applications will not be able to include these profiles in menus or lists, and will not be able to color match to your device (unless they provide a profile for your device themselves).

Some applications may place special-purpose profiles for your device in the ColorSync Profiles folder. For this reason, when your device driver itself displays a pop-up menu or list to the user, you should search not only your private profile location storage, if you use one, but also the ColorSync Profiles folder, to make sure that you offer a complete list of available profiles for your device.

The ColorSync Profiles folder can contain both ColorSync 1.0 profiles and version 2.x profiles. However, your device driver will be able to search for only version 2.x profiles. This is because ColorSync Manager 2.x search functions do not acknowledge ColorSync 1.0 profiles. Support for 1.0 profiles may be even more limited in the future. For more information on this and other limitations of the 1.0 profile format, see “ColorSync 1.0 Profile Support” (page 530).

How a Device Driver Uses Profiles

For most ColorSync Manager functions that your device driver calls, you will need to supply references to profiles for both the source device on which the image was created and the destination device for which it is to be color matched and where it will be rendered.

The driver for an input device such as a scanner typically embeds the scanner profile used to create the image in the document containing the image. The driver for a device that displays an existing image on the system’s display or a printer device that prints a color-matched image typically extracts the embedded profile that accompanies the image from the document containing the image, and uses that profile as the source profile when matching.

Images created using input devices are commonly color matched using the profile for the input device as the source profile and the system profile for the display as the destination profile. “Setting Default Profiles” (page 54) describes how to set the default system profile and other default profiles. Images that are created, depicted, or modified on a display device and that are destined for an output device such as a printer are color matched using the profile for the display as the source profile and the printer’s profile as the destination profile.

To use a profile, you must first obtain a reference to the profile. For a description of how to do this, see “Obtaining Profile References” (page 95).

Devices and Color Management Modules

Your device driver can use the color conversion functions, described in "Converting Between Color Spaces" (page 318), to convert colors between color spaces belonging to the same base family without relying on a CMM. However, color matching, gamut checking, providing color rendering dictionaries to PostScript printers, and other tasks you perform using ColorSync Manager functions all require use of a CMM. It is the CMM that actually carries out the work of the ColorSync Manager functions, such as performing the low-level calculations required to match colors from a source device to a destination device.

If your ColorSync-supportive device driver can use the Apple-supplied default CMM, you need only provide one or more profiles for your device. However, you may want to provide a custom CMM that is optimized for your device and its profiles. For example, a profile can provide private tags containing information a custom CMM might use to achieve better results for the device.

To provide your own CMM, you can create one or obtain one from a vendor. For information describing how to create a CMM, see "Developing Color Management Modules" (page 429) and "ColorSync Reference for Color Management Modules" (page 467)

For additional information on CMMs, see "Setting a Preferred CMM" (page 59) and "How the ColorSync Manager Selects a CMM" (page 84).

Providing ColorSync-Supportive Device Drivers

Your ColorSync-supportive device driver can provide users with various color-matching features based on the type of device you support. This section describes:

- "Providing Minimum ColorSync Support" (page 199)
- "Providing More Extensive ColorSync Support" (page 200)

Providing Minimum ColorSync Support

The minimum level of ColorSync support you should provide differs depending on the type of device your driver supports.

For a scanner, you should embed the scanner profile used to create the image in the document containing the image; this is also referred to as *tagging* an image. If you do not tag the image with the profile, you should at least make the profile

for the image available so that it can be used for color matching. If you do not provide the scanner profile, an application or driver that attempts to color match the scanned image will use the system profile as the source profile and may produce results inconsistent with the colors of the original image.

For a display device driver or a printer device driver, you must preserve images tagged with a profile by not stripping out picture comments used to embed profiles or by leaving profiles in documents that use other methods to include them. For example, if your driver displays or prints PICT files but does not perform color matching, your driver should not strip out the ColorSync-related picture comments that are used to embed profiles in PICT files, begin and end use of a specific profile, and enable and disable color matching. Even though your driver may not make use of the comments, another display or printer driver or an application may use them.

If you don't perform color matching but you want to allow other applications to produce images that are color matched for your device, you should provide a device profile to be used as the destination profile. If you provide a profile for your display or printer and place it in the ColorSync Profiles folder, applications that perform color matching can use it to create a color-matched image expressed in the colors of your device's gamut. A user can then print a color-matched image using the printer your driver supports.

Providing More Extensive ColorSync Support

Instead of relying on an application to color match an image for your printer, your printer driver can color match the image itself before sending it to the printer. To perform color matching, your printer driver must obtain a reference to the source profile. Documents containing images to be printed often contain an embedded profile along with the image. To use the source profile, your printer driver must be able to extract it. If an image is not accompanied by a source profile, the default system profile is used, as described in "Setting Default Profiles" (page 54). In this case, your driver should provide an interface that allows the user to select the rendering intent to be used. Rendering intents are described in "Rendering Intents" (page 60).

You can provide an interface that offers additional features. Your interface can

- allow a user to turn ColorSync color matching on or off before printing
- offer pop-up menus, allowing the user to choose
  - the rendering method to be used in color matching the image (perceptual, colorimetric, or saturation)

- the color-image quality (normal, draft, or best)

Some of these features are discussed below and in “Developing ColorSync-Supportive Applications” (page 81).

Developing Your ColorSync-Supportive Device Driver

This section describes how your device driver can implement certain color matching and related features with ColorSync. It includes the following:

Many of the tasks that your device driver performs to support ColorSync can also be performed by other kinds of color-matching applications. Some of these tasks are mentioned here, but not explained in detail. For a list of code samples shown elsewhere, see “Developing Your ColorSync-Supportive Application” (page 91).

Determining If the ColorSync Manager Is Available

To determine if the ColorSync Manager (version 2.x) is available, call the `Gestalt` function with the `gestaltColorMatchingVersion` selector. For sample code that demonstrates how to perform this operation, see “Determining If the ColorSync Manager Is Available” (page 92). ColorSync constants for use with the `Gestalt` function are described in “Constants for ColorSync Manager Gestalt Selectors and Responses” (page 217). For related information on ColorSync versions, see Table 8-1 (page 526), which lists version numbers for releases of the ColorSync Manager, along with corresponding shared library version numbers, `Gestalt` selector codes, and hardware and system requirements.

Interacting With the User

Using lists and dialog boxes, you can provide choices that influence the color-matching process. For example, you can offer any of the following:

- A list of profiles to select from. You can allow the user to choose the appropriate profile for your printer in its current state. To provide a list of profiles for the user to select from, you must first search for the relevant profiles:
  - Starting with version 2.5 of the ColorSync Manager, you should use the algorithm shown in "Performing Optimized Profile Searching" (page 130) to search for profiles. That algorithm uses `CMIterateColorSyncFolder` (page 304).
  - For versions of the ColorSync Manager prior to 2.5, you can use the functions described in "Searching for Profiles Prior to ColorSync 2.5" (page 306).
- A dialog box for specifying how the image will be color matched. If the source profile is embedded with the image, the source profile specifies the rendering intent to be used. However, if the source profile is not provided and the system profile is used as the source profile, you should allow the user to select the rendering intent to be used. After the user chooses a rendering intent, you can use the selection to set the source profile's header. "Setting a User-Selected Rendering Intent" (page 202) explains this process.
- A dialog box for choosing which color-matching quality of image to produce. A user may want to produce a draft of the image quickly for review before producing the best possible quality of the image. After the user chooses a color-matching quality, you can use the selection to set the source profile's header. "Setting a User-Selected Color-Matching Quality Flag" (page 205) explains how to do this.
- A dialog box for turning ColorSync color matching on or off before printing. If an application that creates or modifies an image has already performed color matching using your printer profile as the destination profile, the user might want to turn off color matching. To provide this capability, your driver should support the `PrGeneral` function with the `enableColorMatchingOp` operation code. For information on the `PrGeneral` function, see *Inside Macintosh: Imaging With QuickDraw*. The `enableColorMatchingOp` operation code constant defined by the ColorSync Manager is described in "PrGeneral Function Operation Codes" (page 423).

Setting a User-Selected Rendering Intent

The ColorSync Manager supports the four standard rendering intents defined by the ICC—perceptual, relative colorimetric, saturated, and absolute colorimetric. Every profile supports these four intents, which are commonly

used to match the colors of a source image to the color gamut of the destination device in the most optimum way for the type of image. These intents are described in detail in “Rendering Intents” (page 60).

If the source profile is embedded with the image, the source profile specifies the rendering intent to be used. However, if the source profile is not available and the system profile must be used as the source profile, you should allow the user to select the rendering intent to be used.

Note
Starting with ColorSync 2.5, your application can call `CMGetDefaultProfileBySpace` (page 297) to obtain an appropriate source profile for matching, rather than using the default system profile. However, you may still wish to allow the user to specify a rendering intent. ◆

To allow users to choose the rendering intent most appropriate for color matching a graphical image, you can provide a pop-up menu or a dialog box identifying the rendering intent options available. By providing a description of the available rendering intents, you can help a user select the rendering intent that best maintains important aspects of the image.

Color professionals and technically-sophisticated users are likely to be familiar with the ICC terms for rendering intent and the gamut-matching strategies they represent. If your application is aimed at novice users, however, you may prefer to use a simplified terminology based on the typical image content associated with a rendering intent, as described in Table 2-1 (page 60). For example, you might note the following:

- For *perceptual matching*, all the colors of a given gamut may be scaled to fit within another gamut. This intent is the best choice for realistic images, such as scanned photographs.
- For *saturation matching*, the relative saturation of colors is maintained from gamut to gamut. Rendering the image using this intent gives the strongest colors and is the best choice for bar graphs and pie charts, in which the actual color displayed is less important than its vividness.
- For *relative colorimetric matching*, the colors that fall within the gamuts of both devices are left unchanged. Some colors in both images will be exactly the same, a useful outcome when colors must match quantitatively. This intent is best suited for logos or “spot colors.”

- For *absolute colorimetric matching*, a close appearance match may be achieved over most of the tonal range, but if the minimum density of the idealized image is different from that of the output image, the areas of the image that are left blank will be different. Colors that fall within the gamuts of both devices are left unchanged.

After the user selects the intent to be used, you must modify the `renderingIntent` field of the system profile's header to reflect the choice. To put the rendering intent chosen by the user in the profile header, follow these steps:

1. **Obtain a profile reference to the system profile.**

   "Identifying the Current System Profile" (page 99) describes how to do this.

2. **Get the profile header of the system profile.**

   You call the function `CMGetProfileHeader` (page 245), passing the profile reference, to obtain the profile's header. The function returns the profile header using a union of type `CMAppleProfileHeader` (page 357). You can use this function for both ColorSync 1.0 profiles and version 2.x profiles.

   For a version 2.x profile, you use the data structure `CM2Header` (page 354). For a version 1.0 profile, you use the `CMHeader` (page 351) data structure. For more information on profile headers, see "Profile Header" (page 351).

3. **Assign the new rendering intent to the header field.**

   To assign a rendering intent to the system profile header's `renderingIntent` field, use the constants defined by the following enumeration:

```
enum {
    cmPerceptual            = 0,
    cmRelativeColorimetric  = 1,
    cmSaturation            = 2,
    cmAbsoluteColorimetric  = 3
};
```

   These constants are described in "Rendering Intent Values for Version 2.x Profiles" (page 419).

4. **Set the modified profile header of the system profile.**

   After you assign the rendering intent, you must replace the header by calling the function `CMSetProfileHeader` (page 254). You can use this function to set a header for a version 1.0 or a version 2.x ColorSync profile. You pass the header using the union `CMAppleProfileHeader` (page 357).

You can now use the system profile to create a color world for the color-matching process. For information on how to create a color world, see "Creating a Color World to Use With the General Purpose Functions" (page 105).

IMPORTANT

When you call `CMSetProfileHeader`, the profile header is modified temporarily. The rendering intent change is discarded when you call the function `CMCloseProfile` (page 223). To preserve the change, you must call the function `CMUpdateProfile` (page 226). ▲

Listing 4-1 includes code that uses the `cmSaturation` constant to set the rendering intent for a profile.

Setting a User-Selected Color-Matching Quality Flag

The ColorSync Manager provides a feature, called the *quality flags settings*, that controls the quality of the color-matching process in relation to the time required to perform the match. This feature, which is not a standard feature defined by the ICC profile format specification, works by letting you manipulate certain bits of the profile header's `flags` field. There are three quality flag settings: normal, draft, and best. For a description of the profile header's `flags` field, see "Quality Flag Values for Version 2.x Profiles" (page 417).

Normal mode is the default setting. Color matching using draft mode takes the least time and produces the least exact results. Color matching using best mode takes the longest time but produces the finest results.

Users sometimes want to produce review drafts of images quickly before expending the time to produce the best-quality final copy. Your interface can allow them this flexibility by offering a dialog box that provides the three options.

After the user selects the color-matching quality, you can use the selection to set the appropriate bits of the source profile's `flags` field. To set the color-matching quality chosen by the user, follow these steps:

1. **Obtain a profile reference to the source profile.**

   "Obtaining Profile References" (page 95) describes how to do this.

2. **Get the profile header of the source profile.**

   You call the function `CMGetProfileHeader` (page 245), passing the profile reference, to obtain the profile's header. The function returns the profile header using a union of type `CMAppleProfileHeader` (page 357).

3. **Optionally, test the current setting of the source profile header's flags.**

   The `flags` field of the source profile header is a long word coded in big-endian notation. Big-endian notation is a means of encoding data in which the first byte within 16-bit and 32-bit quantities is the most significant. The ICC profile consortium reserves the first 2 bits of the low word for its own use. The least significant 2 bits of the high word constitute the quality flag settings used to specify the quality for the color matching. The bit definitions for the `flags` field are shown in Figure 5-1 (page 414).

   To evaluate and interpret the current setting of the quality flags bits, you can take these steps, in order:

   - Right-shift by 16 bits.
   - Mask off the high 14 bits.
   - Compare the result with values defined by the following enumeration:

```
enum
{
 cmNormalMode = 0,
 cmDraftMode = 1,
 cmBestMode = 2
};
```

   These constants are described in "Flag Mask Definitions for Version 2.x Profiles" (page 414).

4. **Set the quality flags bits to the user-selected value.**

   To set the quality flag, you can use the constants defined by the enumeration provided by the ColorSync Manager and shown in step 3.

5. **Set the source profile with the modified profile header.**

   After you set the `flags` field based on the user's selection, you must replace the header by calling the function `CMSetProfileHeader` (page 254). You pass the header using the union `CMAppleProfileHeader` (page 357).

You can now use the source profile to create a color world for the color-matching process. For information on how to create a color world, see

"Creating a Color World to Use With the General Purpose Functions" (page 105).

IMPORTANT

When you call `CMSetProfileHeader`, the profile header is modified temporarily. Changes to the `flags` field are discarded when you call the function `CMCloseProfile` (page 223). To preserve the change, you must call the function `CMUpdateProfile` (page 226). ▲

Listing 4-1 shows how to set the system profile's quality flag to best mode for producing the highest-quality color-matched image. It also sets the rendering intent to saturation before setting up a color world based on the modified system profile and the printer profile.

The `MySetHeader` function shown in Listing 4-1 initializes the `CMProfileRef` (page 358) data structures it will use for the system profile and the printer profile before it calls the following two functions—the ColorSync Manager function `CMGetSystemProfile` (page 294) to obtain a reference to the system profile and its own function `MyGetPrinterProfile` to obtain a reference to the profile for its printer.

The source profile (in this case, the system profile), not the printer profile, determines the quality mode and the rendering intent to be used in color matching the image to the destination printer. Now that it has a reference to the system profile, the code can obtain the profile's header. It does this by calling the function `CMGetProfileHeader` (page 245), specifying the reference it obtained to the system profile.

Using the `kSpeedAndQualityFlagMask` constant it defined earlier, the code clears the quality mode bits of the system profile's `flags` field. Then it sets the quality mode bits to `cmBestMode` to specify best mode quality for color matching. The least significant 2 bits of the `flags` field's high word constitute the quality flag. After setting the quality flag, the code sets the system profile header's `renderingIntent` field to `cmSaturation`.

Now that the code has modified the system profile's header to indicate the user's selections, it calls the `CMSetProfileHeader` function to write the profile header to the profile. Because the driver code intends to use the values the user selected only to color match the image to be printed, it does not permanently preserve the header field changes by calling `CMUpdateProfile` (page 226) to write the changes to the profile. When the code closes its reference to the system

profile after having built the color world, the system profile's header modifications are discarded.

The code calls the `NCWNewColorWorld` function, passing the temporarily modified system profile, to create the color world. It then closes its references to both the system and printer profiles and color matches the image before sending it to the printer. When it no longer needs the color world, the code calls the `CWDisposeColorWorld` function to close the color world and release the memory it uses. Finally, the code tests to ensure that the profile references are closed.

Listing 4-1 Modifying a profile header's quality flag and setting the rendering intent

```
void MySetHeader(void);

CMError MyGetPrinterProfile(CMProfileRef *printerProf);

/* for CM2Header.profileVersion */
#define kMajorVersionMask 0XFF000000

/* two bits used to specify speed & quality */
/* must be shifted left 16 bits in flag's long word */
#define kSpeedAndQualityFlagMask 0X00000003

void MySetHeader(void)
{
    CMError                     cmErr;
    CMProfileRef                sysProf;
    CMAppleProfileHeader        sysHeader;
    CMProfileRef                printerProf;
    CMWorldRef                  cw;

    sysProf = NULL;
    printerProf= NULL;
    cw  = NULL;

    cmErr = CMGetSystemProfile(&sysProf);
    if (cmErr == noErr)
    {
        cmErr = MyGetPrinterProfile(&printerProf);
    }
```

```
if (cmErr == noErr)
{
    cmErr = CMGetProfileHeader(sysProf, &sysHeader);
}

if (cmErr == noErr)
{
    /* clear the current quality and then set it to best */
    sysHeader.cm2.flags &= ~(kSpeedAndQualityFlagMask << 16);
    sysHeader.cm2.flags |= (cmBestMode << 16);

    /* set rendering intent to saturation */
    sysHeader.cm2.renderingIntent = cmSaturation;

    cmErr = CMSetProfileHeader(sysProf, &sysHeader);
}

if (cmErr == noErr)
{
    cmErr = NCWNewColorWorld(&cw, sysProf, printerProf);
}

/* close any open profiles */
if (sysProf != NULL)
{
    (void) CMCloseProfile(sysProf);
}

if (printerProf != NULL)
{
    (void) CMCloseProfile(printerProf);
}
                .
                .
                .
    /* device-driver functions that use the color world to color match
        the image and send it to the printer belong here */
                .
                .
                .
```

```
    if (cw != NULL)
    {
        CWDisposeColorWorld(cw);
    }
}
```

Color Matching an Image to Be Printed

The ColorSync Manager provides QuickDraw-specific and general purpose color-matching functions, as described in “When Color Matching Occurs” (page 62). Printer device drivers usually perform color matching using the general purpose ColorSync Manager functions to match all QuickDraw operations as they pass through the bottleneck routines of the printing grafport.

Note
The general-purpose functions can perform all the operations performed by the QuickDraw-specific functions, but the reverse is not true. ◆

When the stream of QuickDraw data sent to your printer device driver contains a profile embedded using picture comments, your driver should extract the embedded profile using the ColorSync Manager’s `CMUnflattenProfile` function. After you extract the profile and open a reference to it, you should create a new color world based on the extracted profile and a profile for your printer. For information on how to extract an embedded profile, see “Extracting Profiles Embedded in Pictures” (page 118). “Creating a Color World to Use With the General Purpose Functions” (page 105) describes how to create a color world.

If the QuickDraw data stream does not contain embedded profiles, your driver should use the system profile as the source profile in creating the color world.

You should then match subsequent QuickDraw operations using the color world before sending them to your printer. See “Setting Default Profiles” (page 54) for information on how the user and how your code can set default profiles.

ColorSync Reference for Applications and Drivers

Contents

This section describes the functions, constants, and data types defined by the ColorSync Manager for use by your application or device driver. The ColorSync Manager allows your application or device driver to maintain consistent color across devices and across platforms. You can use the ColorSync Manager for color conversion, color matching, color gamut checking, profile management, device calibration, and creating color management modules (CMMs) and calibration plug-ins that perform these services.

This reference has been revised for ColorSync version 2.5.

Note
This document is up-to-date for ColorSync 2.5.1, which introduces no changes to the ColorSync Manager API. ◆

"ColorSync Version Information" (page 525) describes the `Gestalt` information, shared library version numbers, CMM version numbers, and ColorSync header files you use with different versions of the ColorSync Manager. It also includes CPU and Mac OS system requirements.

Gestalt Selector Codes for the ColorSync Manager

This section provides information on the `Gestalt` selector codes you use to determine which version (if any) of the ColorSync Manager is currently available. For an example of how to use these selector codes, see "Determining If the ColorSync Manager Is Available" (page 92). For additional `Gestalt` information, see "Gestalt, Shared Library, and CMM Version Information" (page 526).

Constants for ColorSync Manager Gestalt Selectors and Responses

CHANGED IN COLORSYNC 2.5

You use the following constants with the `Gestalt` function to determine which version of ColorSync is present. For a code sample, see "Determining If the ColorSync Manager Is Available" (page 92). The constant `gestaltColorSync25` was added in ColorSync 2.5. For additional information on ColorSync versions, see Table 8-1 (page 526).

```
enum {
    gestaltColorMatchingVersion     = 'cmtc', /* Selector for version info. */
    gestaltColorSync10              = 0x0100, /* ColorSync 1.0; no QD matching */
    gestaltColorSync11              = 0x0110, /* ColorSync 1.0.3 */
    gestaltColorSync104             = 0x0104, /* ColorSync 1.0.4 */
    gestaltColorSync105             = 0x0105, /* ColorSync 1.0.5 */
    gestaltColorSync20              = 0x0200, /* ColorSync 2.0 */
    gestaltColorSync21              = 0x0210, /* ColorSync 2.1 */
    gestaltColorSync25              = 0x0250, /* ColorSync 2.5 */
    gestaltColorSync251             = 0x0251  /* ColorSync 2.5.1 */
};
```

Enumerator descriptions

`gestaltColorMatchingVersion`

The selector for obtaining version information. Use when calling the `Gestalt` function to determine whether the ColorSync Manager is available.

`gestaltColorSync10`

A `Gestalt` response value of `gestaltColorSync10` indicates version 1.0 of the ColorSync Manager is present. This version supports general purpose color matching only and does not provide QuickDraw-specific matching functions.

`gestaltColorSync11`

A `Gestalt` response value of `gestaltColorSync11` indicates version 1.0.3 of the ColorSync Manager is present.

`gestaltColorSync104`

A `Gestalt` response value of `gestaltColorSync104` indicates version 1.4 of the ColorSync Manager is present.

`gestaltColorSync105`

A `Gestalt` response value of `gestaltColorSync105` indicates version 1.5 of the ColorSync Manager is present.

`gestaltColorSync20`

A `Gestalt` response value of `gestaltColorSync20` indicates version 2.0 of the ColorSync Manager is present.

`gestaltColorSync21`

A `Gestalt` response value of `gestaltColorSync21` indicates version 2.1 of the ColorSync Manager is present.

`gestaltColorSync25`

A `Gestalt` response value of `gestaltColorSync25` indicates version 2.5 of the ColorSync Manager is present.

`gestaltColorSync251`

A `Gestalt` response value of `gestaltColorSync251` indicates version 2.5.1 of the ColorSync Manager is present. Note that version 2.5.1 introduces no new API.

Older ColorSync Gestalt Selectors

NOT RECOMMENDED

The following constants were added to ColorSync version 2.0 to aid in the transition from 68K to PowerPC systems. They are not recommended for new applications and are not guaranteed to be carried forward in future versions of ColorSync. However, they are still supported as of version 2.5 for backward compatibility. If you call the `Gestalt` function passing the selector `gestaltColorMatchingAttr`, you can test the bit fields of the returned value with the `gestaltColorMatchingLibLoaded` constant to determine if the ColorSync Manager shared libraries are loaded, or with the `gestaltHighLevelMatching` constant to determine if the ColorSync QuickDraw-specific functions are present.

```
enum {
    gestaltColorMatchingAttr        = 'cmta',   /* Selector for version info. */
    gestaltHighLevelMatching        = 0,        /* bit 0 set if ColorSync present */
    gestaltColorMatchingLibLoaded   = 1         /* bit 1 set if ColorSync present on
                                                    PowerPC-based machine; cleared if
                                                    on 68K machine. */
};
```

Enumerator descriptions

`gestaltColorMatchingAttr`

The selector for obtaining version information. Use when calling the `Gestalt` function to check for particular ColorSync Manager features.

`gestaltHighLevelMatching`

This constant is provided for backward compatibility only.

Bit 0 of the `Gestalt` response value is always set if ColorSync is present.

`gestaltColorMatchingLibLoaded`

This constant is provided for backward compatibility only. Bit 1 of the `Gestalt` response value is always set on a Power Macintosh machine if ColorSync is present. It is always cleared on a 68K machine if ColorSync is present.

Functions for the ColorSync Manager

This section describes the functions defined by the ColorSync Manager for your application's use. The functions are organized into the following categories:

Accessing Profiles

This section describes the functions you use to perform operations on profiles:

- `CMOpenProfile` (page 222) opens the specified profile and returns a reference to the profile.
- `CMCloseProfile` (page 223) decrements the reference count for the specified profile reference and, if the reference count reaches 0, frees all private memory and other resources associated with the profile.
- `CMProfileModified` (page 225) indicates whether the specified profile has been modified since it was created or last updated.
- `CMUpdateProfile` (page 226) saves modifications to the specified profile.
- `CMNewProfile` (page 227) creates a new profile and associated backing copy.
- `CMCopyProfile` (page 229) duplicates the specified existing profile.
- `CMCloneProfileRef` (page 231) increments the reference count for the specified profile reference.
- `CMGetProfileRefCount` (page 232) obtains the current reference count for the specified profile.
- `NCMGetProfileLocation` (page 233) obtains the location of a profile based on the specified profile reference; **new in ColorSync 2.5**.
- `CMGetProfileLocation` (page 234) obtains the location of a profile based on the specified profile reference (not recommended in ColorSync version 2.5).
- `CMValidateProfile` (page 236) indicates whether the specified profile contains the minimum set of elements required by the current color management module for color matching or color checking.
- `CMFlattenProfile` (page 237) transfers a profile stored in an independent disk file to an external profile format that can be embedded in a graphics document; **changed in ColorSync 2.5**.
- `CMUnflattenProfile` (page 239) transfers a profile embedded in a graphics document to an independent disk file; **changed in ColorSync 2.5**.

CMOpenProfile

Opens the specified profile and returns a reference to the profile.

```
pascal CMError CMOpenProfile (
                        CMProfileRef *prof
                        const CMProfileLocation *theProfile);
```

`prof`	A pointer to a profile reference of type `CMProfileRef` (page 358). On return, the reference refers to the opened profile.
`theProfile`	A pointer to a profile location of type `CMProfileLocation` (page 362) for the profile to open. Commonly a profile is disk-file based, but it may instead be temporary, handle-based, pointer-based, or accessed through a procedure supplied by your application.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

If the `CMOpenProfile` function executes successfully, the profile reference refers to the opened profile. Your application uses this reference, for example, when it calls functions 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 unless your application calls the function `CMUpdateProfile` (page 226) to update the profile. If you call the function `CMCopyProfile` (page 229) 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 open. 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. If the profile is accessed through a procedure provided by your application, use the profile location data type to supply a universal procedure pointer to your procedure.

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 can:

- open a profile
- call the `CMGetProfileHeader` function to obtain the profile's header to modify its values
- set new values
- call the `CMSetProfileHeader` function to replace the modified header
- pass the profile reference to a function such as `NCWNewColorWorld` (page 262) 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 function `CMCloseProfile` (page 223), your changes are discarded (unless you called the `CMUpdateProfile` function).

CMCloseProfile

Decrements the reference count for the specified profile reference and, if the reference count reaches 0, frees all private memory and other resources associated with the profile.

```
pascal CMError CMCloseProfile (CMProfileRef prof);
```

`prof`	A profile reference of type `CMProfileRef` (page 358) that identifies the profile that may need to be closed.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The ColorSync Manager keeps an internal reference count for each profile reference returned from a call to the `CMOpenProfile` (page 222), `CMNewProfile` (page 227), `CMCopyProfile` (page 229), or `CWNewLinkProfile` (page 267) functions. Calling the function `CMCloneProfileRef` (page 231) increments the count; calling the `CMCloseProfile` (page 223) function decrements it. The profile remains open as long as the reference count is greater than 0, indicating there is at least one remaining reference to the profile. When the count reaches 0, the ColorSync Manager releases all private memory, files, or resources allocated in association with that profile.

When the ColorSync Manager releases all private memory and resources associated with a profile, any temporary changes your application made to the profile are not saved unless you first call the `CMUpdateProfile` function to update the profile.

When your application passes a copy of a profile reference to an independent task, whether synchronous or asynchronous, it should call the function `CMCloneProfileRef` (page 231) to increment the reference count. Both the called task and the caller should call `CMCloseProfile` when finished with the profile reference.

Note
You call `CMCloneProfileRef` after copying a *profile reference*, but not after duplicating an *entire profile* (as with the `CMCopyProfile` function). ◆

When your application passes a copy of a profile reference internally, it may not need to call `CMCloneProfileRef`, as long as the application calls `CMCloseProfile` once for the profile.

IMPORTANT
In your application, make sure that `CMCloseProfile` is called once for each time a profile reference is created or cloned. Otherwise, the private memory and resources associated with the profile reference may not be properly freed, or an application may attempt to use a profile reference that is no longer valid. ▲

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 function `CMUpdateProfile` (page 226).

CMProfileModified

Indicates whether the specified profile has been modified since it was created or last updated.

```
pascal CMError CMProfileModified (
                    CMProfileRef prof,
                    Boolean *modified);
```

`prof`	A profile reference of type `CMProfileRef` (page 358) to the profile to examine.
`modified`	A pointer to a Boolean variable. On output, the value of `modified` is set to `true` if the profile has been modified, `false` if it has not.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMProfileModified` function returns, through the `modified` parameter, the current state of the modified flag for the specified profile.

When a profile is first opened, its modified flag is set to `false`. On calls that add to, delete from, or set the profile header or tags, the modified flag is set to `true`. After calling the function `CMUpdateProfile` (page 226), the modified flag is reset to `false`.

CMUpdateProfile

Saves modifications to the specified profile.

```
pascal CMError CMUpdateProfile (CMProfileRef prof);
```

`prof`	A profile reference of type `CMProfileRef` (page 358) to the profile to update.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMUpdateProfile` function makes permanent any changes or additions your application has made to the profile identified by the profile reference, if no other references to that profile exist.

The ColorSync Manager maintains a modified flag to track whether a profile has been modified. After updating a profile, the `CMUpdateProfile` function sets the value of the modified flag for that profile to `false`.

Each time an application calls the function `CMOpenProfile` (page 222), the function creates a unique reference to the profile. An application can also duplicate a profile reference by passing a copy to another task. You cannot use the `CMUpdateProfile` function to update a profile if more than one reference to the profile exists—attempting to do so will result in an error return. You can call the function `CMGetProfileRefCount` (page 232) to determine the reference count for a profile reference.

You cannot use the `CMUpdateProfile` function to update a ColorSync 1.0 profile. For information on updating a ColorSync 1.0 profile, see "Using ColorSync 1.0 Profiles With Newer Versions of the ColorSync Manager" (page 534).

After you fill in tags and their data elements for a new profile created by calling the function `CMNewProfile` (page 227), you must call the `CMUpdateProfile` function 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 the function `CMCloseProfile` (page 223). Otherwise, the changes are discarded.

To modify a profile header, you use the function `CMGetProfileHeader` (page 245) and the function `CMSetProfileHeader` (page 254).

To set profile elements outside the header, you use the function `CMSetProfileElement` (page 253), the function `CMSetProfileElementSize` (page 250), and the function `CMSetPartialProfileElement` (page 251).

CMNewProfile

Creates a new profile and associated backing copy.

```
pascal CMError CMNewProfile (
                    CMProfileRef *prof,
                    const CMProfileLocation *theProfile);
```

`prof` A pointer to a profile reference of type `CMProfileRef` (page 358). On output, a reference to the new profile.

`theProfile` A pointer of type `CMProfileLocation` (page 362) to the profile location where the new profile should be created. A profile is commonly disk-file based—the disk file type for a profile is `'prof'`. However, to accommodate special requirements, you can create a handle- or pointer-based profile, you can create a temporary profile that isn't saved after you call the `CMCloseProfile` function, or you can create a profile that is accessed through a procedure provided by your application. To create a temporary profile, you either specify `cmNoProfileBase` as the kind of profile in the profile location structure or specify `NULL` for this parameter.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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 the function `CMUpdateProfile` (page 226) to save the element data to the profile file. The default ColorSync profile contents include a profile header of type `CM2Header` (page 354) and an element table.

To set profile elements outside the header, you use the function `CMSetProfileElement` (page 253), the function `CMSetProfileElementSize` (page 250), and the function `CMSetPartialProfileElement` (page 251). 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 0 except the `size` and `profileVersion` fields. To set the header elements, you call the function `CMGetProfileHeader` (page 245) to get a copy of the header, assign values to the header fields, then call the function `CMSetProfileHeader` (page 254) to write the new header to the profile.

For each profile class, 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.

Note
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*. For information on how to obtain the ICC format specification, see the section “The International Color Consortium Profile Format” (page 49) in this document.

CMCopyProfile

Duplicates the specified existing profile.

```
pascal CMError CMCopyProfile (
                    CMProfileRef *targetProf,
                    const CMProfileLocation *targetLocation,
                    CMProfileRef srcProf);
```

`targetProf` A pointer to a profile reference of type `CMProfileRef` (page 358). On output, points to the profile copy that was created.

`targetLocation`
A pointer to a location specification that indicates the location, such as 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 handle- or pointer-based profile, you can create a profile that is accessed through a procedure provided by your application, or you can create a temporary profile that isn't saved after you call the `CMCloseProfile` function. To create a temporary profile, you either specify `cmNoProfileBase` as the kind of profile in the profile location structure or specify `NULL` for this parameter. To specify the location, you use the data type `CMProfileLocation` (page 362).

`srcProf` A profile reference to the profile to duplicate.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMCopyProfile` function duplicates an entire 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 duplicated profile. They are not saved to the original profile unless you call `CMUpdateProfile` for that profile.

The ColorSync Manager maintains a modified flag to track whether a profile has been modified. After copying a profile, the `CMCopyProfile` function sets the value of the modified flag for that profile to `false`.

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 function `CMGetProfileElement` (page 243), specifying the `cmCopyrightTag` tag signature for the copyright element (defined in the `CMICCProfile.h` header file).

You should also check the `flags` field of the profile header structure `CM2Header` (page 354) for copyright information. You can test the `cmEmbeddedUseMask` bit of the flags field to determine whether the profile can be used independently. If the bit is set, you should use this profile as an embedded profile only and not copy the profile for your own purposes. The `cmEmbeddedUseMask` mask is described in "Flag Mask Definitions for Version 2.x Profiles" (page 414). The following code snippet shows how you might perform a test using the `cmEmbeddedUseMask` mask:

```
if (myCM2Header.flags & cmEmbeddedUseMask)
{
    // profile should only be used as an embedded profile
}
else
{
    // profile can be used independently
}
```

A calibration program, for example, might use the `CMCopyProfile` function to copy a device's original profile, then modify the copy to reflect the current state of the device. Or an application might want to copy a profile after unflattening 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 function `CMOpenProfile` (page 222). To create a new profile, use the function `CMNewProfile` (page 227). As an alternative to using the `CMCopyProfile` function to duplicate an entire profile, you can use the same profile reference more than once. To do so, you call the function `CMCloneProfileRef` (page 231) to increment the reference count for the reference each time you reuse it. Calling the `CMCloneProfileRef` function increments the count; calling the function `CMCloseProfile` (page 223) decrements it. The profile remains open as long as the reference count is greater than 0, indicating at least one routine retains a reference to the profile.

CMCloneProfileRef

Increments the reference count for the specified profile reference.

```
pascal CMError CMCloneProfileRef (CMProfileRef prof);
```

`prof`	A profile reference of type `CMProfileRef` (page 358) to the profile whose reference count is incremented.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The ColorSync Manager keeps an internal reference count for each profile reference returned from a call to the `CMOpenProfile`, `CMNewProfile`, or `CMCopyProfile` functions. Calling the `CMCloneProfileRef` function increments the count; calling the function `CMCloseProfile` (page 223) decrements it. The profile remains open as long as the reference count is greater than 0, indicating that at least one routine retains a reference to the profile. When the count reaches 0, the ColorSync Manager releases all private memory, files, or resources allocated in association with that profile.

When your application creates a copy of an entire profile with `CMCopyProfile`, the copy has its own reference count. The `CMCloseProfile` routine should be called for the copied profile, just as for the original. When the reference count reaches 0, private resources associated with the copied profile are freed.

When your application merely duplicates a profile reference, as it may do to pass a profile reference to a synchronous or an asynchronous task, it should call `CMCloneProfileRef` to increment the reference count. Both the called task and the caller should call `CMCloseProfile` when finished with the profile reference.

IMPORTANT

In your application, you must make sure that `CMCloseProfile` is called once for each time a profile reference is created or cloned. Otherwise, the memory and resources associated with the profile reference may not be properly freed, or an application may attempt to use a profile reference that is no longer valid. ▲

CMGetProfileRefCount

Obtains the current reference count for the specified profile.

```
pascal CMError CMGetProfileRefCount (
                    CMProfileRef prof,
                    long *count);
```

`prof`	A profile reference of type `CMProfileRef` (page 358) to the profile whose reference count is obtained.
`count`	A pointer to a reference count. On output, the reference count for the specified profile reference.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The ColorSync Manager keeps an internal reference count for each profile reference returned from calls such as `CMOpenProfile` (page 222) or `CMNewProfile` (page 227). Calling the function `CMCloneProfileRef` (page 231) increments the count; calling the function `CMCloseProfile` (page 223) decrements it. The profile remains open as long as the reference count is greater than 0, indicating at least one routine retains a reference to the profile. When the count reaches 0, the ColorSync Manager releases all memory, files, or resources allocated in association with that profile.

An application that manages profiles closely can call the `CMGetProfileRefCount` function to obtain the reference count for a profile reference, then perform special handling if necessary, based on the reference count.

SEE ALSO

To copy a profile with the function `CMCopyProfile` (page 229), you must obtain a reference to that profile by either opening the profile or creating it. To open a profile, use the function `CMOpenProfile` (page 222). To create a new profile, use the function `CMNewProfile` (page 227). As an alternative to using the `CMCopyProfile` function to duplicate an entire profile, you can use the same profile reference more than once. To do so, you call the function `CMCloneProfileRef` (page 231) to increment the reference count for the reference each time you reuse it. Calling the `CMCloneProfileRef` function increments the

count; calling the function `CMCloseProfile` (page 223) decrements it. The profile remains open as long as the reference count is greater than 0, indicating at least one routine retains a reference to the profile.

NCMGetProfileLocation

NEW IN COLORSYNC 2.5

Obtains either a profile location structure for a specified profile or the size of the location structure for the profile.

```
pascal CMError NCMGetProfileLocation (
                    CMProfileRef prof,
                    CMProfileLocation * profLoc,
                    unsigned long * locationSize);
```

`prof` A profile reference of type `CMProfileRef` (page 358). Before calling `NCMGetProfileLocation`, you set the reference to specify the profile for which you wish to obtain the location or location structure size.

`profLoc` A pointer to a profile location structure, as described in "Profile Location" (page 360). If you pass `NULL`, `NCMGetProfileLocation` returns the size of the profile location structure for the profile specified by `prof` in the `locationSize` parameter. If you instead pass a pointer to memory you have allocated for the structure, on return, the structure specifies the location of the profile specified by `prof`.

`locationSize` A pointer to a value of type `long`. If you pass `NULL` for the `profLoc` parameter, on return, `locationSize` contains the size in bytes of the profile location structure for the profile specified by `prof`. If you pass a pointer to a profile location structure in `profLoc`, set `locationSize` to the size of the structure before calling `NCMGetProfileLocation`, using the constant `cmCurrentProfileLocationSize`.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `NCMGetProfileLocation` function is available starting with ColorSync version 2.5. It differs from its predecessor, `CMGetProfileLocation` (page 234), in that the newer version has a parameter for the size of the location structure for the specified profile.

You should use `NCMGetProfileLocation` rather than `CMGetProfileLocation` for the following reasons:

- Code using the older version (`CMGetProfileLocation`) may not be as easily ported to other platforms.
- Specifying the size of the profile location structure ensures that it can grow, if necessary, in the future.

The best way to use `NCMGetProfileLocation` is to call it twice:

1. Pass a reference to the profile to locate in the `prof` parameter and `NULL` for the `profLoc` parameter. `NCMGetProfileLocation` returns the size of the location structure in the `locationSize` parameter.
2. Allocate enough space for a structure of the returned size, then call the function again, passing a pointer in the `profLoc` parameter; on return, the structure specifies the location of the profile.

It is possible to call `NCMGetProfileLocation` just once, using the constant `cmCurrentProfileLocationSize` for the size of the allocated profile location structure and passing the same constant for the `locationSize` parameter. The constant `cmCurrentProfileLocationSize` may change in the future, but will be consistent within the set of headers you build your application with. However, if the size of the `CMProfileLocation` structure changes in a future version of ColorSync (and the value of `cmCurrentProfileLocationSize` as well) and you do not rebuild your application, `NCMGetProfileLocation` may return an error.

CMGetProfileLocation

NOT RECOMMENDED IN COLORSYNC 2.5

Obtains the location of a profile based on the specified profile reference.

```
pascal CMError CMGetProfileLocation (
                    CMProfileRef prof,
                    CMProfileLocation *theProfile);
```

`prof`	A profile reference of type `CMProfileRef` (page 358). Before calling `CMGetProfileLocation`, you set the reference to specify the profile you wish to obtain the location for.
`theProfile`	A pointer to a profile location structure of type `CMProfileLocation` (page 362). On output, specifies the location of the profile. Commonly, a profile is disk-file based, but it may instead be temporary, handle-based, pointer-based, or accessed through a procedure supplied by your application.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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.

VERSION NOTES

Starting with ColorSync version 2.5, you should use the function `NCMGetProfileLocation` (page 233) instead of `CMGetProfileLocation`.

As of version 2.5, if you call `CMGetProfileLocation`, it will just call `NCMGetProfileLocation` in turn, passing the profile specified by `prof`, the profile location specified by `theProfile`, and a location size value of `cmOriginalProfileLocationSize`.

SEE ALSO

To open a profile and obtain a reference to it, use the function `CMOpenProfile` (page 222).

CMValidateProfile

Indicates whether the specified profile contains the minimum set of elements required by the current color management module (CMM) for color matching or color checking.

```
pascal CMError CMValidateProfile (
                    CMProfileRef prof,
                    Boolean *valid,
                    Boolean *preferredCMMnotfound);
```

`prof` A profile reference of type `CMProfileRef` (page 358) to the profile to validate.

`valid` A pointer to a valid profile flag. On output, has the value `true` if the profile contains the minimum set of elements to be valid and `false` if it doesn't.

`preferredCMMnotfound`
A pointer to a flag for whether the preferred CMM was found. On output, has the value `true` if the CMM specified by the profile was not available to perform validation or does not support this function and the default CMM was used. Has the value `false` if the profile's preferred CMM *is* able to perform validation.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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 class, which the CMM requires to perform color matching or gamut checking. For each profile class, 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 default CMM to handle the validation request.

CMFlattenProfile

CHANGED IN COLORSYNC 2.5

Transfers a profile stored in an independent disk file to an external profile format that can be embedded in a graphics document.

```
pascal CMError CMFlattenProfile (
                    CMProfileRef prof,
                    unsigned long flags,
                    CMFlattenUPP proc,
                    void *refCon,
                    Boolean *preferredCMMnotfound);
```

`prof` A profile reference of type `CMProfileRef` (page 358) to the profile to flatten.

`flags` Reserved for future use.

`proc` A pointer to a function that you provide to perform the low-level data transfer. For more information, see the function `MyColorSyncDataTransfer` (page 342).

`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.

Starting with ColorSync version 2.5, the ColorSync Manager calls your transfer function directly, without going through the preferred, or any, CMM.

`preferredCMMnotfound`
A pointer to a flag for whether the preferred CMM was found. On output, has the value `true` if the CMM specified by the profile was not available to perform flattening or does not support this function and the default CMM was used. Has the value `false` if the profile's preferred CMM *is* able to perform flattening.
Starting with ColorSync 2.5, the ColorSync Manager calls your transfer function directly, without going through the preferred, or any, CMM. On return, the value of `preferredCMMnotfound` is guaranteed to be `false`.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The version notes that follow this section describe changes to how the `CMFlattenProfile` function works starting in ColorSync version 2.5.

Prior to version 2.5, 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 CMM is used.

The ColorSync Manager passes to the CMM the pointer to your profile-flattening function. The CMM calls your function `MyColorSyncDataTransfer` (page 342) to perform the actual data transfer.

VERSION NOTES

Starting with ColorSync version 2.5, the ColorSync Manager calls your transfer function directly, without going through the preferred, or any, CMM. As a result, the value returned in the `preferredCMMnotfound` parameter is guaranteed to be `false`.

SEE ALSO

To unflatten a profile embedded in a graphics document to an independent disk file, use the function `CMUnflattenProfile` (page 239).

CMUnflattenProfile

CHANGED IN COLORSYNC 2.5

Transfers a profile embedded in a graphics document to an independent disk file.

```
pascal CMError CMUnflattenProfile (
                    FSSpec *resultFileSpec,
                    CMFlattenUPP proc,
                    void *refCon,
                    Boolean *preferredCMMnotfound);
```

`resultFileSpec`
A pointer to a file specification. On return, the file specification identifies an independent disk file containing the extracted profile.

`proc` A pointer to a function provided by your application to receive the profile data from the CMM and store it in a file.

`refCon` A reference constant for application data which the CMM passes to the `MyColorSyncDataTransfer` function each time it calls the function.
Starting with ColorSync 2.5, the ColorSync Manager calls your transfer function directly, without going through the preferred, or any, CMM.

`preferredCMMnotfound`
A pointer to a flag for whether the preferred CMM was found. On output, has the value `true` if the CMM specified by the profile was not available to perform unflattening or does not support this function and the default CMM was used. Has the value `false` if the profile's preferred CMM *is* able to perform unflattening.
Starting with ColorSync 2.5, the ColorSync Manager calls your

transfer function directly, without going through the preferred, or any, CMM. On return, the value of `preferredCMMnotfound` is guaranteed to be `false`.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The version notes that follow this section describe changes to how the `CMUnflattenProfile` function works starting in ColorSync version 2.5.

Prior to version 2.5, the ColorSync Manager dispatches the `CMUnflattenProfile` function (passed by your application in the `proc` parameter) to the CMM specified by the profile that is to be transferred to a disk file. If the preferred CMM is unavailable or it doesn't support this function, then the default 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 be able to buffer at least 8 bytes of data. For a description of an unflattening function prototype, see `MyColorSyncDataTransfer` (page 342). For a sample of an actual unflattening function, see "Part B: Unflattening the Profile" (page 125).

The CMM calls your version of the `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.

When you have finished using the profile, you must call the `CMCloseProfile` (page 223) function to close the profile and call the File Manager's `FSpDelete` function to delete the file.

VERSION NOTES

Starting with ColorSync version 2.5, the ColorSync Manager calls your transfer function directly, without going through the preferred, or any, CMM. As a result, the value returned in the `preferredCMMnotfound` parameter is guaranteed to be `false`.

Accessing Profile Elements

This section describes the functions you use to examine, set, and change individual elements of a profile.

- `CMProfileElementExists` (page 242) tests whether the specified profile contains a specific element based on the element's tag signature.
- `CMCountProfileElements` (page 243) counts the number of elements in the specified profile.
- `CMGetProfileElement` (page 243) obtains element data from the specified profile based on the specified element tag signature.
- `CMGetProfileHeader` (page 245) obtains the profile header for the specified profile.
- `CMGetPartialProfileElement` (page 246) obtains a portion of the element data from the specified profile based on the specified element tag signature.
- `CMGetIndProfileElementInfo` (page 247) obtains the element tag and data size of an element by index from the specified profile.
- `CMGetIndProfileElement` (page 249) obtains the element data corresponding to a particular index from the specified profile.
- `CMSetProfileElementSize` (page 250) reserves the element data size for a specific tag in the specified profile before setting the element data.
- `CMSetPartialProfileElement` (page 251) sets part of the element data for a specific tag in the specified profile.
- `CMSetProfileElement` (page 253) sets or replaces the element data for a specific tag in the specified profile.
- `CMSetProfileHeader` (page 254) sets the header for the specified profile.
- `CMSetProfileElementReference` (page 254) adds a tag to the specified profile to refer to data corresponding to a previously set element.
- `CMRemoveProfileElement` (page 255) removes an element corresponding to a specific tag from the specified profile.
- `CMGetScriptProfileDescription` (page 256) obtains the internal name (or description) of a profile and the script code identifying the language in which the profile name is specified from the specified profile.

CMProfileElementExists

Tests whether the specified profile contains a specific element based on the element's tag signature.

```
pascal CMError CMProfileElementExists (
                    CMProfileRef prof,
                    OSType tag,
                    Boolean *found);
```

`prof` A profile reference of type `CMProfileRef` (page 358) that specifies the profile to examine.

`tag` The tag signature (for example, '`A2B0`', or constant `cmAToB0Tag`) 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 the ICC format specification, see the section "The International Color Consortium Profile Format" (page 49) in this document. The signatures for profile tags are defined in the `CMICCProfile.h` header file.

`found` A pointer to a flag for whether the element was found. On output, the flag has the value `true` if the profile contains the element or `false` if it doesn't.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

You cannot use this function to test whether certain data in the `CM2Header` profile header exists. Instead, you must call the function `CMGetProfileHeader` (page 245) to copy the entire profile header and read its contents.

CMCountProfileElements

Counts the number of elements in the specified profile.

```
pascal CMError CMCountProfileElements (
                    CMProfileRef prof,
                    unsigned long *elementCount);
```

`prof` A profile reference of type `CMProfileRef` (page 358) to the profile to examine.

`elementCount`
A pointer to an element count. On output, a one-based count of the number of elements.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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" (page 53).

CMGetProfileElement

Obtains element data from the specified profile based on the specified element tag signature.

```
pascal CMError CMGetProfileElement (
                    CMProfileRef prof,
                    OSType tag,
                    unsigned long *elementSize,
                    void *elementData);
```

`prof` A profile reference of type `CMProfileRef` (page 358) to the profile containing the target element.

`tag`	The tag signature (for example, '`A2B0`', or constant `cmAToBOTag`) 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*. For information on how to obtain the ICC format specification, see the section "The International Color Consortium Profile Format" (page 49) in this document. The signatures for profile tags are defined in the `CMICCProfile.h` header file.
`elementSize`	A pointer to a size value. On input, you specify the size of the element data to copy. Specify `NULL` to copy the entire element data. To obtain a portion of the element data, specify the number of bytes to copy. On output, the size of the data returned.
`elementData`	A pointer to memory for element data. On input, you allocate memory. On output, 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.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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 so you can allocate a buffer to hold the returned data.

The `CMGetProfileElement` function serves two purposes: to get an element's size and to obtain an element's data. In both instances, you provide a 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 copy in the `elementSize` parameter. In this case, on output 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 function `CMGetPartialProfileElement` (page 246).

You cannot use this function to obtain a portion of the `CM2Header` profile header. Instead, you must call the function `CMGetProfileHeader` (page 245) to copy the entire profile header and read its contents.

CMGetProfileHeader

Obtains the profile header for the specified profile.

```
pascal CMError CMGetProfileHeader (
                    CMProfileRef prof,
                    CMAppleProfileHeader *header);
```

`prof` A profile reference of type `CMProfileRef` (page 358) to the profile whose header is to be copied.

`header` A pointer to a profile header. On input, depending on the profile version, you may allocate a ColorSync 2.x or 1.0 header. On output, contains the profile data. For information about the ColorSync 2.x profile header structure, see “CM2Header” (page 354). For information about the ColorSync 1.0 header, see “CMHeader” (page 351) and “How ColorSync 1.0 Profiles and Version 2.x Profiles Differ” (page 531).

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMGetProfileHeader` function returns the header for a ColorSync 2.x or ColorSync 1.0 profile. To return the header, the function uses a union of type `CMAppleProfileHeader` (page 357), with variants for version 1.0 and 2.x headers.

A 32-bit version value is located at the same offset in either header. For ColorSync 2.x profiles, this is the `profileVersion` field. For ColorSync 1.0 profiles, this is the `applProfileVersion` field. You can inspect the value at this offset to determine the profile version, and interpret the remaining header fields accordingly.

SEE ALSO

To copy a profile header to a profile after you modify the header's contents, use the function `CMSetProfileHeader` (page 254).

CMGetPartialProfileElement

Obtains a portion of the element data from the specified profile based on the specified element tag signature.

```
pascal CMError CMGetPartialProfileElement (
                    CMProfileRef prof,
                    OSType tag,
                    unsigned long offset,
                    unsigned long *byteCount,
                    void *elementData);
```

`prof` A profile reference of type `CMProfileRef` (page 358) to the profile containing the target element.

`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 the

	ICC format specification, see the section "The International Color Consortium Profile Format" (page 49) in this document. 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`	A pointer to a data byte count. On input, the number of bytes of element data to copy, beginning from the offset specified by the `offset` parameter. On output, the number of bytes actually copied.
`elementData`	A pointer to memory for element data. On input, you pass a pointer to allocated memory. On output, this buffer holds the element data.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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 the `CM2Header` profile header. Instead, you must call the function `CMGetProfileHeader` (page 245) to get the entire profile header and read its contents.

CMGetIndProfileElementInfo

Obtains the element tag and data size of an element by index from the specified profile.

```
pascal CMError CMGetIndProfileElementInfo (
                    CMProfileRef prof,
                    unsigned long index,
```

```
                    OSType *tag,
                    unsigned long *elementSize,
                    Boolean *refs);
```

`prof`	A profile reference of type `CMProfileRef` (page 358) to the profile containing the element.
`index`	A one-based element index within the range returned by the `elementCount` parameter of the `CMCountProfileElements` function.
`tag`	A pointer to an element signature. On output, the tag signature of the element corresponding to the index.
`elementSize`	A pointer to an element size. On output, the size in bytes of the element data corresponding to the tag.
`refs`	A pointer to a reference count flag. On output, set to `true` if more than one tag in the profile refers to element data associated with the tag corresponding to the index.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425). See the discussion for this function for one possible error return value.

DISCUSSION

Before calling the `CMGetIndProfileElementInfo` function, you should call the function `CMCountProfileElements` (page 243), which returns the total number of elements in the profile in the `elementCount` parameter. The number you specify for the `index` parameter when calling `CMGetIndProfileElementInfo` should be in the range of 1 to `elementCount`; otherwise the function will return a result code of `cmIndexRangeErr`. 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

Obtains the element data corresponding to a particular index from the specified profile.

```
pascal CMError CMGetIndProfileElement (
                    CMProfileRef prof,
                    unsigned long index,
                    unsigned long *elementSize,
                    void *elementData);
```

`prof` A profile reference of type `CMProfileRef` (page 358) to the profile containing the element.

`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` A pointer to an element data size. On input, specify the size of the element data to copy (except when `elementData` is set to `NULL`). Specify `NULL` to copy the entire element data. To obtain a portion of the element data, specify the number of bytes to be copy.

On output, the size of the element data actually copied.

`elementData` A pointer to memory for element data. On input, you allocate memory. On output, 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.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Before you call the `CMGetIndProfileElement` function to obtain the element data for an element at a specific index, you first determine the size in bytes of the element data. To determine the data size, you can

- call the function `CMGetIndProfileElementInfo` (page 247), passing the element's index

- call the `CMGetIndProfileElement` function itself, specifying a pointer to an unsigned long data type in the `elementSize` field and a `NULL` value in the `elementData` field

Once you have determined the size of the element data, you allocate a buffer to hold as much of the data as you need. If you want all of the element data, you specify `NULL` in the `elementSize` parameter. If you want only a portion of the element data, you specify the number of bytes you want in the `elementSize` parameter. You supply a pointer to the data buffer in the `elementData` parameter. After calling `CMGetIndProfileElement`, the `elementSize` parameter contains the size in bytes of the element data actually copied.

SEE ALSO

Before calling this function, you should call the function `CMCountProfileElements` (page 243). It returns the profile's total element count in the `elementCount` parameter.

CMSetProfileElementSize

Reserves the element data size for a specific tag in the specified profile before setting the element data.

```
pascal CMError CMSetProfileElementSize (
                    CMProfileRef prof,
                    OSType tag,
                    unsigned long elementSize);
```

`prof` A profile reference of type `CMProfileRef` (page 358) to the profile in which the element data size is reserved.

`tag` The tag signature for the element whose size is 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*. For information on how to obtain the ICC format specification, see the section "The International Color Consortium Profile Format" (page 49) in this document. The signatures for profile tags are defined in the `CMICCProfile.h` header file.

`elementSize`	The total size in bytes to reserve for the element data.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Your application can use the `CMSetProfileElementSize` function to reserve the size of element data for a specific tag before you call the function `CMSetPartialProfileElement` (page 251) 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

Sets part of the element data for a specific tag in the specified profile.

```
pascal CMError CMSetPartialProfileElement (
                    CMProfileRef prof,
                    OSType tag,
                    unsigned long offset,
                    unsigned long byteCount,
                    void *elementData);
```

`prof`	A profile reference of type `CMProfileRef` (page 358) to the profile containing the tag for which the element data is set.
`tag`	The tag signature for the element whose data is 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*. For information on how to obtain the ICC format specification, see

	the section "The International Color Consortium Profile Format" (page 49) in this document. The signatures for profile tags are defined in the `CMICCProfile.h` header file.
`offset`	The offset in the existing element data where data transfer should begin.
`byteCount`	The number of bytes of element data to transfer.
`elementData`	A pointer to the buffer containing the element data to transfer to the profile.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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 this 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 the function `CMSetProfileElementSize` (page 250) to reserve it before you call `CMSetPartialProfileElement` 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 function `CMSetProfileElement` (page 253).

CMSetProfileElement

Sets or replaces the element data for a specific tag in the specified profile.

```
pascal CMError CMSetProfileElement (
                    CMProfileRef prof, OSType tag,
                    unsigned long elementSize,
                    void *elementData);
```

`prof`	A profile reference of type `CMProfileRef` (page 358) to the profile containing the tag for which the element data is set.
`tag`	The tag signature for the element whose data is 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*. For information on how to obtain the ICC format specification, see the section "The International Color Consortium Profile Format" (page 49) in this document. The signatures for profile tags are defined in the `CMICCProfile.h` header file.
`elementSize`	The size in bytes of the element data set.
`elementData`	A pointer to the buffer containing the element data to transfer to the profile.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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 transfer.

CMSetProfileHeader

Sets the header for the specified profile.

```
pascal CMError CMSetProfileHeader (
                    CMProfileRef prof,
                    const CMAppleProfileHeader *header);
```

`prof` A profile reference of type `CMProfileRef` (page 358) to the profile whose header is set.

`header` A pointer to the new header to set for the profile.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

You can use the `CMSetProfileHeader` function to set a header for a version 1.0 or a version 2.x 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.x profile, you use a data structure of type `CM2Header` (page 354). For a version 1.0 profile, you use a data structure of type `CMHeader` (page 351). You pass the header you supply in the `CMAppleProfileHeader` union, described in "CMAppleProfileHeader" (page 357).

CMSetProfileElementReference

Adds a tag to the specified profile to refer to data corresponding to a previously set element.

```
pascal CMError CMSetProfileElementReference (
                    CMProfileRef prof,
                    OSType elementTag,
                    OSType referenceTag);
```

`prof` A profile reference of type `CMProfileRef` (page 358) to the profile to add the tag to.

`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 add to the profile to refer to the element data corresponding to `elementTag`.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

After the `CMSetProfileElementReference` function executes 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 can set a reference to an element that was originally a reference itself without circularity.

If you call the function `CMSetProfileElement` (page 253) 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

Removes an element corresponding to a specific tag from the specified profile.

```
pascal CMError CMRemoveProfileElement (
                    CMProfileRef prof,
                    OSType tag);
```

`prof`	A profile reference of type `CMProfileRef` (page 358) to the profile containing the tag remove.
`tag`	The tag signature for the element to remove.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMRemoveProfileElement` function deletes the tag as well as the element data from the profile.

CMGetScriptProfileDescription

Obtains the internal name (or description) of a profile and the script code identifying the language in which the profile name is specified from the specified profile.

```
pascal CMError CMGetScriptProfileDescription (
                    CMProfileRef prof,
                    Str255 name,
                    ScriptCode *code);
```

`prof`	A profile reference of type `CMProfileRef` (page 358) to the profile whose profile name and script code are obtained.
`name`	A pointer to a name string. On output, the profile name.
`code`	A pointer to a script code. On output, the script code.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The element data of the text description tag (which has the signature `'desc'` or constant `cmSigProfileDescriptionType`, defined in the `CMICCProfile.h` header file) specifies the profile name and script code. The `name` parameter returns the profile name as a Pascal string. Use this function so that your application does not need to obtain and parse the element data, which contains other information.

Accessing Named Color Profile Values

This section describes the functions you use to retrieve information from a named color profile.

- `CMGetNamedColorInfo` (page 257) obtains information about a named color space from its profile reference.
- `CMGetNamedColorValue` (page 258) obtains device and PCS color values for a specific color name from a named color space profile.

- `CMGetIndNamedColorValue` (page 259) obtains device and PCS color values for a specific named color index from a named color space profile.
- `CMGetNamedColorIndex` (page 260) obtains a named color index for a specific color name from a named color space profile.
- `CMGetNamedColorName` (page 260) obtains a named color name for a specific named color index from a named color space profile.

CMGetNamedColorInfo

Obtains information about a named color space from its profile reference.

```
pascal CMError CMGetNamedColorInfo (
                    CMProfileRef prof,
                    unsigned long *deviceChannels,
                    OSType *deviceColorSpace,
                    OSType *PCSColorSpace,
                    unsigned long *count,
                    StringPtr prefix,
                    StringPtr suffix);
```

`prof` A profile reference of type `CMProfileRef` (page 358) to a named color space profile to obtain named color information from.

`deviceChannels`
A pointer to a count value. On output, the number of device channels in the color space for the profile. It should agree with the "data color space" field in the profile header. For example, Pantone maps to CMYK, a four-channel color space. A value of 0 indicates no device channels were available.

`deviceColorSpace`
A pointer to a device color space. On output, a device color space, such as CMYK.

`PCSColorSpace`
A pointer to a profile connection space color space. On output, an interchange color space, such as Lab.

`count` A pointer to a count value. On output, the number of named colors in the profile.

`prefix`	A pointer to a Pascal string. On output, the string contains a prefix, such as “Pantone”, for each color name. The prefix identifies the named color system described by the profile.
`suffix`	A pointer to a Pascal string. On output, the string contains a suffix for each color name, such as “CVC”.
function result	A result code of type `CMError`. For possible values, see “Result Codes for the ColorSync Manager” (page 425).

DISCUSSION

The `CMGetNamedColorInfo` function returns information about the named color space referred to by the passed profile reference.

CMGetNamedColorValue

Obtains device and PCS color values for a specific color name from a named color space profile.

```
pascal CMError CMGetNamedColorValue (
                    CMProfileRef prof,
                    StringPtr name,
                    CMColor *deviceColor,
                    CMColor *PCSColor);
```

`prof`	A profile reference of type `CMProfileRef` (page 358) to a named color space profile to obtain color values from.
`name`	A pointer to a Pascal string. You supply a color name string for the color to get information for.
`deviceColor`	A pointer to a device color. On output, a device color value in `CMColor` union format. If the profile does not contain device values, `deviceColor` is undefined.
`PCSColor`	A pointer to a profile connection space color. On output, an interchange color value in `CMColor` union format.
function result	A result code of type `CMError`. For possible values, see “Result Codes for the ColorSync Manager” (page 425).

DISCUSSION

Based on the passed color name, the `CMGetNamedColorValue` function does a lookup into the named color tag and, if the name is found in the tag, returns device and color PCS values. Otherwise, `CMGetNamedColorValue` returns an error code.

CMGetIndNamedColorValue

Obtains device and PCS color values for a specific named color index from a named color space profile.

```
pascal CMError CMGetIndNamedColorValue (
                    CMProfileRef prof,
                    unsigned long index,
                    CMColor *deviceColor,
                    CMColor *PCSColor);
```

`prof`	A profile reference of type `CMProfileRef` (page 358) to a named color space profile to obtain color values from.
`index`	A one-based index value for a named color.
`deviceColor`	A pointer to a device color. On output, a device color value in `CMColor` union format. If the profile does not contain device values, `deviceColor` is undefined.
`PCSColor`	A pointer to a profile connection space color. On output, an interchange color value in `CMColor` union format.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Based on the passed named color index, the `CMGetIndNamedColorValue` function does a lookup into the named color tag and returns device and PCS values. If the index is greater than the number of named colors, `CMGetIndNamedColorValue` returns an error code.

CMGetNamedColorIndex

Obtains a named color index for a specific color name from a named color space profile.

```
pascal CMError CMGetNamedColorIndex (
                    CMProfileRef prof,
                    StringPtr name,
                    unsigned long *index);
```

`prof` A profile reference of type `CMProfileRef` (page 358) to a named color space profile to obtain a named color index from.

`name` A pointer to a Pascal string. You supply a color name string value for the color to obtain the color index for.

`index` A pointer to an index value. On output, an index value for a named color.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Based on the passed color name, the `CMGetNamedColorIndex` function does a lookup into the named color tag and, if the name is found in the tag, returns the index. Otherwise, `CMGetNamedColorIndex` returns an error code.

CMGetNamedColorName

Obtains a named color name for a specific named color index from a named color space profile.

```
pascal CMError CMGetNamedColorName (
                    CMProfileRef prof,
                    unsigned long index,
                    StringPtr name)
```

`prof` A profile reference of type `CMProfileRef` (page 358) to a named color space profile to obtain a named color name from.

`index`	An index value for a named color to obtain the color name for.
`name`	A pointer to a Pascal string. On output, a color name string.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Based on the passed color name index, the `CMGetNamedColorName` function does a lookup into the named color tag and returns the name. If the index is greater than the number of named colors, `CMGetNamedColorName` returns an error code.

Matching Colors Using General Purpose Functions

This section describes the functions you use to perform color matching with general purpose ColorSync Manager functions that are independent of QuickDraw. To use the general purpose functions, you first create a color-matching world, which establishes how matching will take place between the given profiles.

"General Purpose Color-Matching Functions" (page 64) provides an overview of these functions, while "Creating a Color World to Use With the General Purpose Functions" (page 105) provides a code sample for working with them.

Once you create a color world, it persists until you dispose of it, independent of the functions for which you use it. The QuickDraw-specific functions described in "Matching Colors Using QuickDraw-Specific Functions" (page 284) take source and destination profile reference parameters and are state-based, whereas the general purpose functions described here are not state-based.

You use the following functions to perform color-matching using general purpose functions.

- `NCWNewColorWorld` (page 262) creates a color world for color matching based on the specified source and destination profiles; **changed in ColorSync 2.5**.
- `CWConcatColorWorld` (page 265) sets up a color world that includes a set of profiles for various color transformations among devices in a sequence; **changed in ColorSync 2.5**.
- `CWNewLinkProfile` (page 267) creates a device link profile based on the specified set of profiles.

- `CMGetCWInfo` (page 270) obtains information about the color management modules (CMMs) used for a specific color world; **changed in ColorSync 2.5**.
- `CWDisposeColorWorld` (page 271) releases the private storage associated with a color world when your application has finished using the color world.
- `CWMatchPixMap` (page 272) matches a pixel map in place based on a specified color world.
- `CWCheckPixMap` (page 274) checks the colors of a pixel map using the profiles of a specified color world to determine whether the colors are in the gamut of the destination device.
- `CWMatchBitmap` (page 276) matches the colors of a bitmap to the gamut of a destination device using the profiles specified by a color world.
- `CWCheckBitMap` (page 279) tests the colors of the pixel data of a bitmap to determine whether the colors map to the gamut of the destination device.
- `CWMatchColors` (page 281) matches colors in a color list, using the specified color world.
- `CWCheckColors` (page 283) tests a list of colors using a specified color world to see if they fall within the gamut of a destination device.

NCWNewColorWorld

CHANGED IN COLORSYNC 2.5

Creates a color world for color matching based on the specified source and destination profiles.

```
pascal CMError NCWNewColorWorld (
                    CMWorldRef *cw,
                    CMProfileRef src,
                    CMProfileRef dst);
```

`cw` A pointer to a color world. On output, a reference to a matching session color world of type `CMWorldRef` (page 383). You pass this reference to other functions that use the color world.

`src`	A profile reference of type `CMProfileRef` (page 358) that specifies the source profile for 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. Starting with ColorSync version 2.5, you can call `CMGetDefaultProfileBySpace` (page 297) to get the default profile for a specific color space or `CMGetProfileByAVID` (page 300) to get a profile for a specific display. With any version of ColorSync, you can specify a `NULL` value to indicate the ColorSync system profile. Note, however, that starting with version 2.5, use of the system profile has changed, as described in "Setting Default Profiles" (page 54).
`dst`	A profile reference of type `CMProfileRef` (page 358) that specifies the destination profile for the color-matching world. This profile's `dataColorSpace` element corresponds to the destination data type for subsequent calls to functions using this color world. Starting with ColorSync version 2.5, you can call `CMGetDefaultProfileBySpace` (page 297) to get the default profile for a specific color space or `CMGetProfileByAVID` (page 300) to get a profile for a specific display. With any version of ColorSync, you can specify a `NULL` value to indicate the ColorSync system profile. Note, however, that starting with version 2.5, use of the system profile has changed, as described in "Setting Default Profiles" (page 54).
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

You must set up a color world before your application can perform general purpose color-matching or color-checking operations. To set up a color world for these operations, your application can call `NCWNewColorWorld` after obtaining references to the profiles to use as the source and destination profiles for the color world. The following rules govern the types of profiles allowed:

- You can specify a device profile or a color space conversion profile for the source and destination profiles.
- You can not specify a device link profile or an abstract profile for either the source profile or the destination profile.

- Only one profile can be a named color profile.
- You can specify the system profile explicitly by reference or by giving `NULL` for either the source profile or the destination profile.

You should call the function `CMCloseProfile` (page 223) for both the source and destination profiles to dispose of their references after execution of the `NCWNewColorWorld` function.

When you pass a color world to a color-matching or color-checking function, the ColorSync Manager uses the selection scheme described in "How the ColorSync Manager Selects a CMM" (page 84) to determine the CMM or CMMs to use for the session.

The quality flag setting (indicating normal mode, draft mode, or best mode) specified by the source profile prevails for the entire session. The quality flag setting is stored in the `flags` field of the profile header. See `CM2Header` (page 354) and "Flag Mask Definitions for Version 2.x Profiles" (page 414) for more information on the use of flags. The rendering intent specified by the source profile also prevails for the entire session.

For more information on color worlds, see `CMConcatProfileSet` (page 384).

VERSION NOTES

The parameter descriptions for `src` and `dst` describe changes in how this function is used starting with ColorSync version 2.5.

SEE ALSO

The function `CWConcatColorWorld` (page 265) also allocates a color world reference of type `CMWorldRef` (page 383).

CWConcatColorWorld

CHANGED IN COLORSYNC 2.5

Sets up a color world that includes a set of profiles for various color transformations among devices in a sequence.

```
pascal CMError CWConcatColorWorld (
                    CMWorldRef *cw,
                    CMConcatProfileSet *profileSet);
```

`cw` A pointer to a color world. On output, a reference to a color world of type `CMWorldRef` (page 383). You pass the returned reference to other functions that use the color world for color-matching and color-checking sessions.

`profileSet` A pointer of type `CMConcatProfileSet` (page 384) to an array of profiles describing the processing to carry out. You create the array and initialize it in processing order—source through destination.
You set the `keyIndex` field of the `CMConcatProfileSet` data structure to specify the zero-based index of the profile within the profile array whose specified CMM should 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.
Note that starting with ColorSync 2.5, the user can set a preferred CMM with the ColorSync control panel, as described in "Setting a Preferred CMM" (page 59). If that CMM is available, ColorSync will use that CMM for all color conversion and matching operations the CMM is capable of performing.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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 pointer to a data structure of type `CMConcatProfileSet` to identify the profile array.

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 `CM2Header` (page 354) and "Flag Mask Definitions for Version 2.x Profiles" (page 414) 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 for 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 can 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 link profile. If you specify a device link profile, you cannot specify any other profiles in the profiles array; a device link 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. However, when you set up a color-matching session with a named color space profile and other profiles, the named color profile must be first or the last profile in the color world—it cannot be in the middle.
- You cannot specify `NULL` to indicate the system profile. Note that starting with version 2.5, use of the system profile has changed, as described in "Setting Default Profiles" (page 54).
- If you specify a color space profile in the middle of the profile sequence, it is ignored by the default CMM.
- If you specify a named color profile, it must be the first or the last profile. Otherwise, `CWConcatColorWorld` returns the value `cmCantConcatenateError`.

A after executing the `CWConcatColorWorld` function, you should call the function `CMCloseProfile` (page 223) for each profile to dispose of its reference.

For more information on color worlds, see `CMConcatProfileSet` (page 384).

VERSION NOTES

The parameter description for `profileSet` includes changes in how this function is used starting with ColorSync version 2.5.

When you pass a color world created with the `CWConcatColorWorld` function to a color-matching or color-checking function, the ColorSync Manager uses the selection scheme described in "How the ColorSync Manager Selects a CMM" (page 84) to determine the CMM or CMMs to use for the session.

Note also that starting with version 2.5, use of the system profile has changed, as described in "Setting Default Profiles" (page 54).

SEE ALSO

Instead of passing in an array of profiles, you can specify a device link profile. For information on how to create a device link profile, see the `CWNewLinkProfile` function, which is described next.

CWNewLinkProfile

CHANGED IN COLORSYNC 2.5

Creates a device link profile based on the specified set of profiles.

```
pascal CMError CWNewLinkProfile (
                    CMProfileRef *prof,
                    const CMProfileLocation *targetLocation,
                    CMConcatProfileSet *profileSet);
```

`prof` A pointer to an uninitialized profile reference of type `CMProfileRef` (page 358). On output, points to the new device link profile reference.

`targetLocation`
On output, a pointer to a location specification for the resulting profile. A device link profile cannot be a temporary profile: that is, it cannot have a location type of `cmNoProfileBase`.

`profileSet`	On input, an array of profiles describing the processing to carry out. The array is in processing order—source through destination. For a description of the `CMConcatProfileSet` (page 384) data type, see `CMHeader` (page 351).
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

This discussion is accurate for versions of ColorSync prior to 2.5. See the version notes below for changes starting with version 2.5.

You can use this function to create a new single profile containing a set of profiles and pass the device link profile to the function `CWConcatColorWorld` (page 265) instead of specifying each profile in an array. A device link profile provides a means of storing in concatenated format a series of device profiles and non-device profiles that are used repeatedly in the same sequence.

IMPORTANT

The only way to use a device link profile is to pass it to the `CWConcatColorWorld` function as the sole profile specified by the array passed in the `profileSet` parameter. ▲

The zero-based `keyIndex` field of the `CMConcatProfileSet` data structure specifies the index of the profile within the device link profile whose preferred CMM is used for the entire color-matching or color-checking session. The profile header's `CMMType` field specifies the preferred CMM for the specified profile. 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 `CM2Header` (page 354) 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 link profile:

- The first and last profiles you specify in the profiles array for a device link profile must be device profiles.
- You cannot specify a named color profile.
- You cannot include another device link profile in the series of profiles you specify in the profiles array.
- The only way to use a device link profile is to pass it to the `CWConcatColorWorld` function as the sole profile specified by the array passed in the `profileSet` parameter.
- You cannot embed a device link profile in an image.
- You cannot specify `NULL` to indicate the system profile.

When your application is finished with the device link profile, it must close the profile with the function `CMCloseProfile` (page 223).

This function privately maintains all the profile information required by the color world for color-matching and color-checking sessions. Therefore, after executing the `CWNewLinkProfile` function, you should call the `CMCloseProfile` (page 223) function for each profile used to build a device link profile (to dispose of each profile reference).

VERSION NOTES

When you pass a color world created with the `CWNewLinkProfile` function to a color-matching or color-checking function, the ColorSync Manager uses the selection scheme described in "How the ColorSync Manager Selects a CMM" (page 84) to determine the CMM or CMMs to use for the session.

Note also that starting with version 2.5, use of the system profile has changed, as described in "Setting Default Profiles" (page 54).

CMGetCWInfo

CHANGED IN COLORSYNC 2.5

Obtains information about the color management modules (CMMs) used for a specific color world.

```
pascal CMError CMGetCWInfo (
                    CMWorldRef cw,
                    CMCWInfoRecord *info);
```

`cw` A reference to the color world of type `CMWorldRef` (page 383) about which you want information.

`info` A pointer to a color world information record of type `CMCWInfoRecord` (page 382) that your application supplies. On output, the ColorSync Manager returns information in this structure describing the number of CMMs involved in the matching session and the CMM type and version of each CMM used.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

This discussion is accurate for versions of ColorSync prior to 2.5. See the version notes below for changes starting with version 2.5.

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 CMM used, your application must first obtain a reference to the color world. To obtain a reference to a ColorSync color world, you (or some other process) must have created the color world using the function `NCWNewColorWorld` (page 262) or the function `CWConcatColorWorld` (page 265).

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 used for a device link profile or a concatenated color world. However, you cannot be certain if the specified CMM will actually be used until the ColorSync Manager determines internally if the CMM is available and able to perform the requested function. For example, when the specified CMM is not available, the default CMM is used.

The `CMGetCWInfo` function identifies the CMM or CMMs to use. 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 that will be used and an array of two members containing structures of type `CMMInfoRecord` (page 385). 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 General Purpose Functions" (page 261).

VERSION NOTES

Starting with ColorSync 2.5, a user can select a preferred CMM with the ColorSync control panel, as described in "Setting a Preferred CMM" (page 59). If the user has selected a preferred CMM, and if it is available, then it will be used for all color conversion and matching operations. For related information, see "Color Management Modules" (page 58) and "How the ColorSync Manager Selects a CMM" (page 84).

SEE ALSO

The functions `NCWNewColorWorld` (page 262) and `CWConcatColorWorld` (page 265) both allocate color world references of type `CMWorldRef` (page 383).

CWDisposeColorWorld

Releases the private storage associated with a color world when your application has finished using the color world.

```
pascal void CWDisposeColorWorld (CMWorldRef cw);
```

`cw` A color world reference of type `CMWorldRef` (page 383).

function result This routine does not return an error value.

SEE ALSO

The function `NCWNewColorWorld` (page 262) and the function `CWConcatColorWorld` (page 265) both allocate color world references of type `CMWorldRef` (page 383).

The following functions use color worlds. If you create a color world to pass to one of these functions, you must dispose of the color world when your application is finished with it.

- `CWMatchColors` (page 281)
- `CWCheckColors` (page 283)
- `CWMatchBitmap` (page 276)
- `CWCheckBitMap` (page 279)
- `CWMatchPixMap` (page 272),
- `CWCheckPixMap` (page 274)

CWMatchPixMap

Matches a pixel map in place based on a specified color world.

```
pascal CMError CWMatchPixMap (
                    CMWorldRef cw,
                    PixMap *myPixMap,
                    CMBitmapCallBackUPP progressProc,
                    void *refCon);
```

`cw` A reference to the color world of type `CMWorldRef` (page 383) in which matching is to occur.

`myPixMap` A pointer to the pixel map to match. 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 function supplied by your application to monitor progress or abort the operation as the pixel map colors are matched. The default CMM calls your function approximately every half-second, unless matching is completed in less time.

If the function returns a result of `true`, the operation is aborted. You 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, refer to the function `MyCMBitmapCallBackProc` (page 345).

`refCon` A reference constant for application data that is passed as a parameter to calls to `progressProc`.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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 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 match. Every CMM must support the `CMMatchColors` function.

For this function to execute successfully, the source and destination profiles' data color spaces (`dataColorSpace` field) must be RGB to match the data color space of the pixel map, which is implicitly RGB. For color spaces other than RGB, you should use the function `CWMatchBitmap` (page 276).

If you specify a pointer to a callback 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 Manager 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 progress bar 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 function `MyCMBitmapCallBackProc` (page 345).

SEE ALSO

The functions `NCWNewColorWorld` (page 262) and `CWConcatColorWorld` (page 265) both allocate color world references of type `CMWorldRef` (page 383).

Note
Applications do not interact directly with the function `CMMatchColors` (page 470). ◆

CWCheckPixMap

Checks the colors of a pixel map using the profiles of a specified color world to determine whether the colors are in the gamut of the destination device.

```
pascal CMError CWCheckPixMap (
                    CMWorldRef cw,
                    PixMap *myPixMap,
                    CMBitmapCallBackUPP progressProc,
                    void *refCon,
                    BitMap *resultBitMap);
```

`cw` A reference to the color world of type `CMWorldRef` (page 383) in which color checking is to occur.

`myPixMap` A pointer to the pixel map to check colors for. 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 default CMM calls your function approximately every half-second unless color checking occurs in less time; 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 function `MyCMBitmapCallBackProc` (page 345).

`refCon` A reference constant for application data passed as a parameter to calls to your `MyCMBitmapCallBackProc` function pointed to by `progressProc`.

`resultBitMap`

A pointer to a QuickDraw bitmap. On output, bits 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`.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425). `CWCheckPixMap` returns `cmCantGamutCheckError` if the color world does not contain gamut information. For more information, see "Flag Mask Definitions for Version 2.x Profiles" (page 414).

DISCUSSION

The `CWCheckPixMap` function performs a gamut test of the pixel data of the `myPixMap` pixel map to determine if its colors 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 default CMM to perform the matching. If the preferred CMM is available but 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 match. Every CMM must support the `CMCheckColors` function.

For this function to execute successfully, the source and destination profiles' data color spaces (`dataColorSpace` field) must be RGB to match the data color space of the pixel map, which is implicitly RGB.

If you specify a pointer to a callback 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 progress bar 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 function `MyCMBitmapCallBackProc` (page 345).

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 functions `NCWNewColorWorld` (page 262) and `CWConcatColorWorld` (page 265) both return color world references of type `CMWorldRef` (page 383).

CWMatchBitmap

CHANGED WITH COLORSYNC 2.5

Matches the colors of a bitmap to the gamut of a destination device using the profiles specified by a color world.

```
pascal CMError CWMatchBitmap (
                    CMWorldRef cw,
                    CMBitMap *bitMap,
                    CMBitmapCallBackUPP progressProc,
                    void *refCon,
                    CMBitMap *matchedBitMap);
```

`cw` A reference to a color world of type `CMWorldRef` (page 383) in which matching is to occur.

`bitMap` A pointer to a bitmap of type `CMBitmap` (page 380) whose colors are to be matched.

`progressProc`	A calling program–supplied universal procedure pointer to a callback function that allows your application to monitor progress or abort the operation as the bitmap colors are matched. The default CMM calls your function approximately every half-second unless color matching occurs in less time; 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 parameter. For a description of the function your application supplies, see the function `MyCMBitmapCallBackProc` (page 345).
`refCon`	A reference constant for application data passed through as a parameter to calls to the `progressProc` function.
`matchedBitMap`	A pointer to a bitmap. On output, contains the color-matched image. You must allocate the pixel buffer pointed to by the `image` field of the structure `CMBitmap` (page 380). If you specify `NULL` for `matchedBitMap`, then the source bitmap is matched in place.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CWMatchBitmap` function matches a bitmap using the profiles specified by the given color world.

The ColorSync Manager dispatches this function to the CMM determined by the process described in "How the ColorSync Manager Selects a CMM" (page 84).

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, defined in "Color Space Constants With Packing Formats" (page 409), are currently supported for the `CWMatchBitmap` function:

- `cmGray16Space`
- `cmGrayA32Space`
- `cmRGB16Space`
- `cmRGB24Space`

- cmRGB32Space
- cmRGB48Space
- cmARGB32Space
- cmRGBA32Space
- cmCMYK32Space
- cmCMYK64Space
- cmHSV32Space
- cmHLS32Space
- cmYXY32Space
- cmXYZ32Space
- cmLUV32Space
- cmLAB24Space
- cmLAB32Space
- cmLAB48Space
- cmGamutResult1Space
- cmNamedIndexed32Space
- cmMCFive8Space
- cmMCSix8Space
- cmMCSeven8Space
- cmMCEight8Space

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 execute 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, unless the destination profile is a named color space profile.

IMPORTANT

If you set `matchedBitMap` to `NULL` to specify in-place matching, you must be sure the space required by the destination bitmap is less than or equal to the size of the source bitmap. ▲

VERSION NOTES

Support for the following color space constants, defined in "Color Space Constants With Packing Formats" (page 409), was added with ColorSync version 2.5:

- `cmGray16Space`
- `cmGrayA32Space`
- `cmRGB48Space`.
- `cmCMYK64Space`
- `cmLAB48Space`

SEE ALSO

The functions `NCWNewColorWorld` (page 262) and `CWConcatColorWorld` (page 265) both allocate color world references of type `CMWorldRef` (page 383).

CWCheckBitMap

Tests the colors of the pixel data of a bitmap to determine whether the colors map to the gamut of the destination device.

```
pascal CMError CWCheckBitMap (
                    CMWorldRef cw,
                    const CMBitMap *bitMap,
                    CMBitmapCallBackUPP progressProc,
                    void *refCon,
                    CMBitMap *resultBitMap);
```

`cw` A reference to the color world of type `CMWorldRef` (page 383) to use for the color check.

`bitMap` A pointer to a bitmap of type `CMBitmap` (page 380) whose colors are to be checked.

`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 default CMM calls your function approximately every half-second unless color checking occurs in less time; 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 function `MyCMBitmapCallBackProc` (page 345).

`refCon` A reference constant for application data passed as a parameter to calls to `progressProc`.

`resultBitMap`

A pointer to a bitmap. On output, contains the results 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 structure `CMBitmap` (page 380) 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, as described in "Abstract Color Space Constants" (page 406).

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425). `CWCheckBitMap` returns `cmCantGamutCheckError` if the color world does not contain gamut information. For more information, see "Flag Mask Definitions for Version 2.x Profiles" (page 414).

DISCUSSION

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 "Selecting a CMM by the Arbitration Algorithm" (page 86) 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 default CMM is used to perform the color matching.

For the `CWCheckBitMap` function to execute 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. `CWCheckBitMap` is not supported if the color world was initialized with a named color space profile.

SEE ALSO

The functions `NCWNewColorWorld` (page 262) and `CWConcatColorWorld` (page 265) both allocate color world references of type `CMWorldRef` (page 383).

CWMatchColors

Matches colors in a color list, using the specified color world.

```
pascal CMError CWMatchColors (
                    CMWorldRef cw,
                    CMColor *myColors,
                    unsigned long count);
```

`cw` A reference to the color world of type `CMWorldRef` (page 383) that describes how matching is to occur in the color-matching session.

`myColors`	A pointer to an array containing a list of colors of type `CMColor` (page 378). On input, contains the list of colors to match. On output, 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.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CWMatchColors` function matches colors according to the profiles corresponding to the specified color world. On input, the color values in the `myColors` array are assumed to be specified in the data color space of the source profile. On output, the color values in the `myColors` array are transformed to the data color space of the destination profile.

All color management modules (CMM)s must support this function. To determine which CMM to use for the color-matching session, the ColorSync Manager follows the arbitration scheme described in "Introduction to ColorSync" (page 45).

This function supports color-matching sessions set up with one of the multichannel color data types.

SEE ALSO

The functions `NCWNewColorWorld` (page 262) and `CWConcatColorWorld` (page 265) both allocate color world references of type `CMWorldRef` (page 383).

CWCheckColors

Tests a list of colors using a specified color world to see if they fall within the gamut of a destination device.

```
pascal CMError CWCheckColors (
                    CMWorldRef cw,
                    CMColor *myColors,
                    unsigned long count,
                    long *result);
```

`cw` A reference to the color world of type `CMWorldRef` (page 383) describing how the test is to occur.

`myColors` A pointer to an array containing a list of colors of type `CMColor` (page 378) to be checked . 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 32-bit data. On output, each 32-bit value is interpreted as a bit field with each bit representing a color in the array pointed to by `myColors`. You 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.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425). `CWCheckBitMap` returns `cmCantGamutCheckError` if the color world does not contain gamut information. For more information, see "Flag Mask Definitions for Version 2.x Profiles" (page 414).

DISCUSSION

The color test provides a preview of color matching using the specified color world.

All CMMs must support the `CWCheckColors` function. To determine which CMM to use for the color-checking session, the ColorSync Manager follows the arbitration scheme described in "Introduction to ColorSync" (page 45).

The `result` bit array indicates 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.

If you have set a profile's gamut-checking mask so that no gamut information is included—see "Flag Mask Definitions for Version 2.x Profiles" (page 414)—`CWCheckColors` returns the `cmCantGamutCheckError` error.

The `CWCheckColors` function supports matching sessions set up with one of the multichannel color data types. `CWCheckColors` is not supported if the color world was initialized with a named color space profile.

SEE ALSO

The functions `NCWNewColorWorld` (page 262) and `CWConcatColorWorld` (page 265) both allocate color world references of type `CMWorldRef` (page 383).

Matching Colors Using QuickDraw-Specific Functions

This section describes the functions you use to perform color-matching when working with QuickDraw. "QuickDraw-Specific Color-Matching Functions" (page 64) provides an overview of these functions, while "Matching to Displays Using QuickDraw-Specific Operations" (page 101) provides a code sample for working with them. "Matching Colors Using General Purpose Functions" (page 261) describes color-matching functions that don't rely on QuickDraw.

- `NCMBeginMatching` (page 285) sets up a QuickDraw-specific ColorSync matching session, using the specified source and destination profiles; **changed in ColorSync 2.5**.
- `CMEndMatching` (page 287) concludes a QuickDraw-specific ColorSync matching session initiated by a previous call to the `NCMBeginMatching` function.
- `CMEnableMatchingComment` (page 288) inserts a comment into the currently open picture to turn matching on or off.
- `NCMDrawMatchedPicture` (page 288) matches a picture's colors (using the system profile as the initial source profile but switching to any embedded profiles as they are encountered) to a destination device's color gamut, as the picture is drawn, using the specified destination profile; **changed in ColorSync 2.5**.

NCMBeginMatching

CHANGED IN COLORSYNC 2.5

Sets up a QuickDraw-specific ColorSync matching session, using the specified source and destination profiles.

```
pascal CMError NCMBeginMatching (
                    CMProfileRef src,
                    CMProfileRef dst,
                    CMMatchRef *myRef);
```

`src` A profile reference of type `CMProfileRef` (page 358) that specifies the source profile for the matching session. Starting with ColorSync version 2.5, you can call `CMGetDefaultProfileBySpace` (page 297) to get the default profile for a specific color space or `CMGetProfileByAVID` (page 300) to get a profile for a specific display.
With any version of ColorSync, you can specify a `NULL` value to indicate the ColorSync system profile. Note, however, that starting with version 2.5, use of the system profile has changed, as described in "Setting Default Profiles" (page 54).

`dst` A profile reference of type `CMProfileRef` (page 358) that specifies the destination profile for the matching session. Starting with ColorSync version 2.5, you can call `CMGetDefaultProfileBySpace` (page 297) to get the default profile for a specific color space or `CMGetProfileByAVID` (page 300) to get a profile for a specific display.
With any version of ColorSync, you can specify a `NULL` value to indicate the ColorSync system profile. Note, however, that starting with version 2.5, use of the system profile has changed, as described in "Setting Default Profiles" (page 54).

`myRef` A pointer to a matching session. On output, it specifies the QuickDraw-specific matching session that was set up.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `NCMBeginMatching` function sets up a QuickDraw-specific matching session, 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 function`CMEndMatching` (page 287) 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 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 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. The ColorSync Manager general purpose functions (which include the use of concatenated profiles well suited to print-preview processing) offer an easier and faster way to do this. These functions are described in "Matching Colors Using General Purpose Functions" (page 261).

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 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 their use, you must call the function `NCMUseProfileComment` (page 290) for color matching using picture comments.

This function causes matching for the specified devices rather than for the current color graphics port.

VERSION NOTES

The parameter descriptions for `src` and `dst` describe changes in how this function is used starting with ColorSync version 2.5.

SEE ALSO

The `NCMBeginMatching` function uses QuickDraw and 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 general purpose functions described in "Matching Colors Using General Purpose Functions" (page 261).

For background information on graphics devices, see *Inside Macintosh: Imaging With QuickDraw*.

CMEndMatching

Concludes a QuickDraw-specific ColorSync matching session initiated by a previous call to the `NCMBeginMatching` 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.

function result This routine does not return an error value.

DISCUSSION

The `CMEndMatching` function releases private memory allocated for the QuickDraw-specific matching session.

After you call the `NCMBeginMatching` function and before you call `CMEndMatching` to end the matching session, embedded color-matching picture comments, such as `cmEnableMatching` and `cmDisableMatching`, are not acknowledged.

CMEnableMatchingComment

Inserts a comment into the currently open picture to turn matching on or off.

```
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`.

function result This routine does not return an error value.

If you call this function when no picture is open, it will have no effect.

NCMDrawMatchedPicture

CHANGED IN COLORSYNC 2.5

Matches a picture's colors, using the system profile as the initial source profile but switching to any embedded profiles as they are encountered, to a destination device's color gamut, as the picture is drawn, using the specified destination profile.

```
pascal void NCMDrawMatchedPicture (
                    PicHandle myPicture,
                    CMProfileRef dst,
                    Rect *myRect);
```

`myPicture` The QuickDraw picture whose colors are to be matched.

`dst`	A profile reference of type `CMProfileRef` (page 358) to the profile of the destination device. Starting with ColorSync version 2.5, if you know the destination display device, you can call `CMGetProfileByAVID` (page 300) to get the specific profile for the display, or you can call `CMGetDefaultProfileBySpace` (page 297) to get the default profile for the RGB color space,. With any version of ColorSync, you can specify a `NULL` value to indicate the ColorSync system profile. Note, however, that starting with version 2.5, use of the system profile has changed, as described in "Setting Default Profiles" (page 54).
`myRect`	A pointer to a destination rectangle for rendering the picture specified by `myPicture`.
function result	This routine does not return an error value. Instead, after calling `NCMDrawMatchedPicture` you call the `QDError` routine to determine if an error has occurred.

DISCUSSION

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.)

The ColorSync Manager defines five picture comment kinds, as described in "Picture Comment Kinds for Profiles and Color Matching" (page 399). For embedding to work correctly, each embedded profile that is used for matching must be terminated by a picture comment of kind `cmEndProfile`. If a picture comment is not specified to end the profile after drawing operations using that profile are performed, the profile will remain in effect until another embedded profile is introduced that has a picture comment kind of `cmBeginProfile`. To avoid unexpected matching effects, always pair use of the `cmBeginProfile` and `cmEndProfile` picture comments. When the ColorSync Manager encounters a `cmEndProfile` picture comment, it restores use of the system profile for matching until it encounters another `cmBeginProfile` picture comment.

The picture is drawn with matched colors to all screen graphics devices. If the current graphics device is not a screen device, matching occurs for that graphics device only.

If the current port is not a color graphics port, then calling this function is equivalent to calling `DrawPicture`, in which case no color matching occurs.

VERSION NOTES

The parameter description for `dst` describes changes in how this function is used starting with ColorSync version 2.5.

Embedding Profile Information in Pictures

Applications use the QuickDraw `PicComment` function, described in *Inside Macintosh: Imaging With QuickDraw*, to add picture comments to a picture. The ColorSync Manager provides the `NCMUseProfileComment` (page 290) function for automatically embedding profile information with the `PicComment` function.

NCMUseProfileComment

Automatically embeds a profile or a profile identifier into an open picture.

```
pascal CMError NCMUseProfileComment (
                    CMProfileRef prof,
                    unsigned long flags);
```

`prof` A profile reference of type `CMProfileRef` (page 358) to the profile to embed.

`flags` A flag value in which individual bits determine settings. "Constants for Embedding Profiles and Profile Identifiers" (page 402) describes constants for use with this parameter. For example, you pass `cmEmbedWholeProfile` to embed a whole profile or `cmEmbedProfileIdentifier` to embed a profile identifier. No other values are currently defined; all other bits are reserved for future use.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `NCMUseProfileComment` function automatically generates the picture comments required to embed the specified profile or profile identifier into the open picture.

To embed a profile, you use the constant `cmEmbedWholeProfile` to set the `flags` parameter before calling `NCMUseProfileComment`. The `NCMUseProfileComment` function 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: `cmBeginProfileSel` to begin the profile, `cmContinueProfileSel` to continue, and `cmEndProfileSel` to end the profile. These constants are described in "Picture Comment Selectors for Embedding Profile Information" (page 400).

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 `cmContinueProfileSel` to embed each segment.

To embed a profile identifier of type `CMProfileIdentifier` (page 359), you use the constant `cmEmbedProfileIdentifier` to set the `flags` parameter before calling `NCMUseProfileComment`. The function extracts the necessary information from the profile reference (`prof`) to embed a profile identifier for the profile. The profile reference can refer to a previously embedded profile, or to a profile on disk in the ColorSync Profiles folder.

IMPORTANT

You can use this function to embed most types of profiles in an image, including device link 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 embedding to work 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. Always pair use of the `cmBeginProfile` and `cmEndProfile` picture comments. When the ColorSync Manager encounters a `cmEndProfile` picture comment, it restores use

of the system profile for matching until it encounters another `cmBeginProfile` picture comment.

For more information on the `PicComment` selector values used by `NCMUseProfileComment`, see "Picture Comment Selectors for Embedding Profile Information" (page 400).

VERSION NOTES

In ColorSync 2.0, the `flags` parameter was ignored and the routine always embedded the entire profile.

In ColorSync 2.0, if the `prof` parameter refers to a version 1.0 profile, the profile is not embedded into the picture correctly. In ColorSync versions starting with 2.1, this bug has been fixed. One possible workaround for this problem in ColorSync 2.0 is to call `CMCopyProfile` to copy the 1.0 profile reference into a handle. The handle can then be embedded into the picture using `CMUseProfileComment`.

Getting the Preferred CMM

Starting with ColorSync version 2.5, the ColorSync control panel lets a user choose a preferred CMM from any CMMs that are present, as described in "Setting a Preferred CMM" (page 59).

The ColorSync Manager provides the function `CMGetPreferredCMM` (page 292) so that you can determine the preferred CMM in your code.

CMGetPreferredCMM

NEW IN COLORSYNC 2.5

Identifies the preferred CMM specified by the ColorSync control panel.

```
pascal CMError CMGetPreferredCMM (
                    OSType *cmmType,
                    Boolean *preferredCMMnotfound)
```

`cmmType` A pointer to an `OSType`. On return, the component subtype for the preferred CMM. For example, the subtype for ColorSync's default CMM is `'appl'` and the subtype for the Kodak CMM is `'KCMS'`. A return value of `NULL` indicates the preferred CMM in the ColorSync control panel is set to Automatic, as described in "Setting a Preferred CMM" (page 59).

`preferredCMMnotfound` A pointer to a Boolean flag for whether the preferred CMM was not found. On return, has the value `true` if the CMM was not found, `false` if it was found.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

The `CMGetPreferredCMM` function returns in the `cmmType` parameter a value that identifies the preferred CMM the user last specified in the ColorSync control panel. `CMGetPreferredCMM` returns `false` in the `preferredCMMnotfound` parameter if the preferred CMM is currently available and `true` if it is not. The preferred CMM may not be available, for example, because a user specifies a preferred CMM in the ColorSync control panel, then reboots with extensions off. ColorSync does not change the preferred CMM setting when the preferred CMM is not available.

Getting and Setting the System Profile File

The ColorSync Manager provides the following functions your code can call to identify a profile as the system profile or obtain a reference to that profile. These functions replace the capability provided by the ColorSync 1.0 Profile Responder. "Setting Default Profiles" (page 54) describes changes in how the ColorSync Manager uses the system profile starting in version 2.5; it also describes use of the system profile in previous versions.

- `CMGetSystemProfile` (page 294) obtains a reference to the current system profile; **changed in ColorSync 2.5**.
- `CMSetSystemProfile` (page 295) sets the current system profile; **changed in ColorSync 2.5**.

CMGetSystemProfile

CHANGED IN COLORSYNC 2.5

Obtains a reference to the current system profile.

```
pascal CMError CMGetSystemProfile (CMProfileRef *prof);
```

`prof`	A pointer to a profile reference of type `CMProfileRef` (page 358). On output, a reference to the current system profile.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

"Setting Default Profiles" (page 54) describes how the ColorSync Manager uses the system profile, both in version 2.5 and in previous versions. For example, the system profile may serve as the default profile for color operations for which no profile is specified.

The following functions allow you to pass `NULL` as a parameter value to specify the system profile as a source or destination profile:

- `CMNewProfile` (page 227)
- `NCWNewColorWorld` (page 262)
- `NCMBeginMatching` (page 285)
- `NCMDrawMatchedPicture` (page 288)

Note that instead of passing `NULL`, you can pass a profile reference to a specific profile, including the system profile.

If you want to specify the system profile for any other function that requires a profile reference, such as `CWConcatColorWorld` (page 265) and `CWNewLinkProfile` (page 267), you must use an explicit reference. You can obtain such a reference with the `CMGetSystemProfile` function.

There are other reasons you might need to obtain a reference to the current system profile. For example, your application might need to display the name of the current system profile to a user.

To identify the location of the physical file, call the function `CMGetProfileLocation` (page 234).

VERSION NOTES

Starting with version 2.5, use of the system profile has changed, as described in "Setting Default Profiles" (page 54). So rather than call `CMGetSystemProfile` to obtain a reference to the system profile, you may be able to obtain a profile that's more appropriate for the current operation by calling `CMGetDefaultProfileBySpace` (page 297) to get the default profile for a color space or by calling `CMGetProfileByAVID` (page 300) to get the profile for a specific display.

SEE ALSO

When your application has finished using the current system profile, it must close the reference to the profile by calling the function `CMCloseProfile` (page 223).

CMSetSystemProfile

CHANGED IN COLORSYNC 2.5

Sets the current system profile.

```
pascal CMError CMSetSystemProfile (
                    const FSSpec *profileFileSpec);
```

`profileFileSpec`
A pointer to a file specification structure. Before calling `CMSetSystemProfile`, set the structure to specify the desired system profile.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

By default, a standard RGB profile is configured as the system profile. By calling the `CMSetSystemProfile` function, your application can specify a new system profile. You can configure only a display device profile as the system profile.

VERSION NOTES

Starting with version 2.5, use of the system profile has changed, as described in "Setting Default Profiles" (page 54).

The function `CMSetSystemProfile` does not retrieve video card gamma data (introduced in ColorSync version 2.5) to set the video card; use the function `CMSetProfileByAVID` (page 300) instead.

SEE ALSO

The `FSSpec` data type you use to specify the profile file location is described in *Inside Macintosh: Files.*

Getting and Setting Default Profiles by Color Space

This section describes the ColorSync functions, new in version 2.5, that you use to get and set default profiles for RGB, CMYK, Lab, and XYZ color spaces. Note that a user can set the default profile for the RGB and CMYK color spaces with the ColorSync control panel, as described in "Setting Default Profiles" (page 54).

- `CMGetDefaultProfileBySpace` (page 297) gets the default profile for the specified color space; **new in ColorSync 2.5**.
- `CMSetDefaultProfileBySpace` (page 298) sets the default profile for the specified color space; **new in ColorSync 2.5**.

CMGetDefaultProfileBySpace

NEW IN COLORSYNC 2.5

Gets the default profile for the specified color space.

```
pascal CMError CMGetDefaultProfileBySpace(
                    OSType dataColorSpace,
                    CMProfileRef * prof);
```

`dataColorSpace`
A four-character identifier of type `OSType`. You pass a color space signature that identifies the color space you wish to get the default profile for. The currently-supported values are `cmRGBData`, `cmCMYKData`, `cmLabData`, and `cmXYZData`. These constants are a subset of the constants described in "Color Space Signatures" (page 402). If you supply a value that isn't supported, the `CMGetDefaultProfileBySpace` function returns an error value of `paramErr`.

`prof`
A pointer to a profile reference. On return, the reference specifies the current profile for the color space specified by `dataColorSpace`. `CMGetDefaultProfileBySpace` currently supports only file-based profiles.

function result
A result code of type `CMError`. For possible values, see the description of `dataColorSpace` for this function, as well as "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMGetDefaultProfileBySpace` function currently supports the RGB, CMYK, Lab, and XYZ color spaces. The signature constants for these color spaces (shown above with the `dataColorSpace` parameter description) are described in "Color Space Signatures" (page 402). Support for additional color spaces may be provided in the future. `CMGetDefaultProfileBySpace` returns an error value of `paramErr` if you pass a color space constant it doesn't currently support.

The `CMGetDefaultProfileBySpace` function always attempts to return a file-based profile for a supported color space. For example, if the user has not specified a default profile in the ColorSync control panel for the specified color space, or if the profile is not found (the user may have deleted the profiles in the ColorSync Profiles folder or even the folder itself), `CMGetDefaultProfileBySpace` creates a profile, stores it on disk, and returns a reference to that profile. However, you should always check for an error return—for example, a user may have booted from a CD, so that `CMGetDefaultProfileBySpace` cannot save a profile file to disk.

CMSetDefaultProfileBySpace

NEW IN COLORSYNC 2.5

Sets the default profile for the specified color space.

```
pascal CMError CMSetDefaultProfileBySpace (
                    OSType dataColorSpace,
                    CMProfileRef prof);
```

`dataColorSpace`

A four-character identifier of type `OSType`. You pass a color space signature that identifies the color space you wish to set the default profile for. The currently-supported values are `cmRGBData`, `cmCMYKData`, `cmLabData`, and `cmXYZData`. These constants are a subset of the constants described in "Color Space Signatures" (page 402). If you supply a value that isn't supported, the `CMGetDefaultProfileBySpace` function returns an error value of `paramErr`.

`prof`	A profile reference. Before calling `CMSetDefaultProfileBySpace`, set the reference to specify the default profile for the color space. The profile must be file-based; otherwise, the function returns a `CMInvalidProfileLocation` error.
function result	A result code of type `CMError`. For possible values, see the descriptions of `dataColorSpace` and `prof` for this function, as well as "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMSetDefaultProfileBySpace` function currently supports the RGB, CMYK, Lab, and XYZ color spaces. The signature constants for these color spaces (shown above with the `dataColorSpace` parameter description) are described in "Color Space Signatures" (page 402). Support for additional color spaces may be provided in the future. `CMSetDefaultProfileBySpace` returns a value of `paramErr` if you pass a color space constant it doesn't currently support.

Note that a user can also use the ColorSync control panel to specify a default profile for the RGB and CMYK color spaces, as described in "Setting Default Profiles" (page 54).

Getting and Setting Monitor Profiles by AVID

This section describes the ColorSync functions, new in version 2.5, that you use to get and set a profile for each monitor. These routines work with the `AVIDType` data type, which is defined by the Display Manager and used to specify a device such as a monitor. Note that a user can set a profile for each display with the Monitors & Sound control panel, as described in "Setting a Profile for Each Monitor" (page 69). You can get more information about AVID values from the Display Manager SDK.

- `CMGetProfileByAVID` (page 300) gets the current profile for a monitor; **new in ColorSync 2.5**.
- `CMSetProfileByAVID` (page 300) sets the profile for the specified monitor, optionally setting video card gamma; **new in ColorSync 2.5**.

CMGetProfileByAVID

NEW IN COLORSYNC 2.5

Gets the current profile for a monitor.

```
pascal CMError CMGetProfileByAVID (
                    AVIDType theAVID,
                    CMProfileRef *prof);
```

`theAVID` A Display Manager ID value. You pass the ID value for the monitor for which to get the profile. You can get more information about AVID values from the Display Manager SDK.

`prof` A pointer to a profile reference. On return, a reference to the current profile for the monitor specified by `theAVID`.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

If the Display Manager supports ColorSync, the `CMGetProfileByAVID` function calls on the Display Manager to get the profile for the specified display. This is the case if the version of the Display Manager is 2.2.5 or higher (if `gestaltDisplayMgrAttr` has the `gestaltDisplayMgrColorSyncAware` bit set).

CMSetProfileByAVID

NEW IN COLORSYNC 2.5

Sets the profile for the specified monitor, optionally setting video card gamma.

```
pascal CMError CMSetProfileByAVID (
                    AVIDType theAVID,
                    CMProfileRef prof);
```

`theAVID` A Display Manager ID value. You pass the ID value for the monitor for which to set the profile. You can get more information about AVID values from the Display Manager SDK.

`prof`	A profile reference. Before calling `CMSetProfileByAVID`, set the reference to identify the profile for the monitor specified by `theAVID`.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

If you specify a profile that contains the optional profile tag for video card gamma, `CMSetProfileByAVID` extracts the tag and sets the video card based on the tag data, as described in "Video Card Gamma" (page 70). This is the only ColorSync function that sets video card gamma. The tag constant `cmVideoCardGammaTag` is described with the constants in "Video Card Gamma Constants" (page 421).

When a user sets a display profile using the Monitors & Sound control panel, the system profile is set to the same profile, as described in "Setting a Profile for Each Monitor" (page 69). When you call `CMSetProfileByAVID` to set a profile for a monitor, you may also wish to make that profile the system profile. If so, you must call `CMSetSystemProfile` (page 295) explicitly—calling `CMSetProfileByAVID` alone has no affect on the system profile.

Note that if the Display Manager supports ColorSync, the `CMSetProfileByAVID` function calls on the Display Manager to set the profile for the specified display. This is the case if the version of the Display Manager is 2.2.5 or higher (if `gestaltDisplayMgrAttr` has the `gestaltDisplayMgrColorSyncAware` bit set).

Locating the ColorSync Profiles Folder

The ColorSync Manager provides the function `CMGetColorSyncFolderSpec` (page 302) to obtain the location of the ColorSync Profiles folder. See the function description for changes starting with ColorSync version 2.5.

CMGetColorSyncFolderSpec

CHANGED IN COLORSYNC 2.5.

Obtains the volume reference number and the directory ID for the ColorSync Profiles folder.

```
pascal CMError CMGetColorSyncFolderSpec (
                    short vRefNum,
                    Boolean createFolder,
                    short *foundVRefNum,
                    long *foundDirID);
```

`vRefNum` The reference number of the volume to examine. 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 the `Folders.h` header file, to assign a value to the flag.

`foundVRefNum` A pointer to a volume reference number. On output, the volume reference number for the volume on which the ColorSync Profiles folder resides.

`foundDirID` A pointer to a directory ID. On output, the directory ID for the volume on which the ColorSync Profiles folder resides.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

If the ColorSync Profiles folder does not already exist, you can use this function to create it.

VERSION NOTES

Starting with version 2.5, the name and location of the profile folder changed, as described in "Profile Search Locations" (page 55).

Your application should use the function `CMIterateColorSyncFolder` (page 304), available starting in ColorSync version 2.5, or one of the search functions described in "Searching for Profiles Prior to ColorSync 2.5" (page 306), to search for a profile file, even if it is only looking for one file. Do not search for a profile file by obtaining the location of the profiles folder and searching for the file directly.

SEE ALSO

For information about the Macintosh file system, see *Inside Macintosh: Files.*

Profile Searching

This section describes the ColorSync functions you use to search for profiles.

IMPORTANT

Your application should use one of the ColorSync search functions described here to search for a profile file, even if you are only looking for one file. Do not search for a profile file by obtaining the location of the profiles folder and searching for the file directly. ▲

- "Searching for Profiles With ColorSync 2.5" (page 303)
- "Searching for Profiles Prior to ColorSync 2.5" (page 306)
- "Searching for a Profile by Profile Identifier" (page 314)

Searching for Profiles With ColorSync 2.5

Starting with version 2.5, ColorSync provides a profile cache, described in "The Profile Cache and Optimized Searching" (page 57), for keeping track of profile files. A flexible new routine, `CMIterateColorSyncFolder`, takes advantage of the profile cache to provide truly optimized searching and quick access to profile information. Your application can quickly examine all the profile files in the ColorSync Profiles folder to find those that match a desired criteria.

- `CMIterateColorSyncFolder` (page 304) Iterates over the available profiles; **new in ColorSync 2.5**.

For additional information on profile searching, see “Searching for Profiles Prior to ColorSync 2.5” (page 306) and “Searching for a Profile by Profile Identifier” (page 314).

CMIterateColorSyncFolder

NEW IN COLORSYNC 2.5

Iterates over the available profiles.

```
pascal CMError CMIterateColorSyncFolder (
                    CMProfileIterateUPP proc,
                    unsigned long * seed,
                    unsigned long * count,
                    void * refCon);
```

`proc` A universal procedure pointer of type `CMProfileIterateUPP`, which is described in `CMProfileIterateProcPtr` (page 365). If you do not wish to receive callbacks, pass `NULL` for this parameter. Otherwise, pass a pointer to your callback routine.

`seed` A pointer to a value of type `long`. The first time you call `CMIterateColorSyncFolder`, you typically set the value to 0. In subsequent calls, you set the value to the seed value obtained from the previous call. ColorSync uses the value in determining whether to call your callback routine, as described in the discussion for this function.
On return, the value is the current seed for the profile cache (unless you pass `NULL`, as described in the discussion).

`count` A pointer to a value of type `long`. On return, the value is the number of available profiles. `CMIterateColorSyncFolder` provides the number of profiles even when no iteration occurs (unless you pass `NULL`, as described in the discussion below). To determine the count alone, without iteration, call `CMIterateColorSyncFolder` and pass a value of `NULL` for all parameters except `count`.

`refCon`	An untyped pointer to arbitrary data supplied by your application. `CMIterateColorSyncFolder` passes this data to your callback routine. If you pass `NULL` for the `refCon` parameter, `CMIterateColorSyncFolder` passes `NULL` to your callback routine.
function result	A result code of type `CMError`. If your callback function returns an error, `CMIterateColorSyncFolder` stops iterating and returns the error value to its caller (presumably your code). For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Starting with ColorSync version 2.5, when your application needs information about the profiles currently available in the ColorSync Profiles folder, it can call the `CMIterateColorSyncFolder` routine, which in turn calls your callback routine once for each profile.

Note
Starting with version 2.5, the name and location of the profile folder changed. In addition, the folder can now contain profiles within nested folders, as well as aliases to profiles or aliases to folders containing profiles. There are limits on the nesting of folders and aliases. For details, see "Profile Search Locations" (page 55). ◆

Even though there may be many profiles available, `CMIterateColorSyncFolder` can take advantage of ColorSync's profile cache to return profile information quickly, and (if the cache is valid) without having to open any profiles. For each profile, `CMIterateColorSyncFolder` supplies your routine with the profile header, script code, name, and location, in a structure of type `CMProfileIterateData` (page 366). As a result, your routine may be able to perform its function, such as building a list of profiles to display in a pop-up menu, without further effort (such as opening each file-based profile).

IMPORTANT
Only 2.x profiles are included in the profile search result. ▲

Before calling `CMIterateColorSyncFolder` for the first time, you typically set `seed` to 0. ColorSync compares 0 to its current seed for the profile cache. It isn't likely they will match—the odds are roughly one in two billion against it. If the values don't match, the routine iterates through all the profiles in the cache, calling

your callback routine once for each profile. `CMIterateColorSyncFolder` then returns the actual seed value in `seed` (unless you passed `NULL` for that parameter).

If you pass the returned seed value in a subsequent call, and if there has been no change in the available profiles, the passed seed will match the stored cache seed and no iteration will take place.

Note that you can pass a `NULL` pointer for the `seed` parameter without harm. The result is the same as if you passed a pointer to 0, in that the function iterates through the available profiles, calling your callback routine once for each profile. However, the function doesn't return a seed value, since you haven't passed a valid pointer.

You can force ColorSync to call your callback routine (if any profiles are available) by passing a `NULL` pointer or by passing 0 for the seed value. But suppose you have an operation, such as building a pop-up menu, that you only want to perform if the available profiles have changed. In that case, you pass the seed value from a previous call to `CMIterateColorSyncFolder`. If the profile folder has not changed, ColorSync will not call your callback routine.

Note that if there are no profiles available, ColorSync does not call your callback routine.

Note
You can safely pass `NULL` for any or all of the parameters to the `CMIterateColorSyncFolder` function. If you pass `NULL` for all of the parameters, calling the function merely forces rebuilding of the profile cache, if necessary. ◆

For sample code demonstrating how to use `CMIterateColorSyncFolder`, see "Performing Optimized Profile Searching" (page 130).

Searching for Profiles Prior to ColorSync 2.5

NOT RECOMMENDED IN COLORSYNC 2.5

This section describes the functions you use to search for profiles with versions of ColorSync prior to version 2.5. These functions are not recommended with version 2.5—see "Searching for Profiles With ColorSync 2.5" (page 303) instead.

The functions described here allow your application to search for profile files within the ColorSync Profiles folder based on certain criteria, and to obtain references to the found profiles and their file specifications.

Note
Starting with version 2.5, the name and location of the profile folder changed, as described in "Profile Search Locations" (page 55). ◆

Code that uses these functions still works in version 2.5, but does not take full advantage of ColorSync's profile cache and optimized searching, which is described in "Searching for Profiles With ColorSync 2.5" (page 303).

- `CMNewProfileSearch` (page 308) searches the ColorSync Profiles folder and returns a list of 2.x profiles that match the search specification; **not recommended in ColorSync 2.5**.
- `CMUpdateProfileSearch` (page 310) searches the ColorSync Profiles folder and updates an existing search result obtained originally from the `CMNewProfileSearch` function; **not recommended in ColorSync 2.5**.
- `CMDisposeProfileSearch` (page 311) frees the private memory allocated for a profile search after your application has completed the search; **not recommended in ColorSync 2.5**.
- `CMSearchGetIndProfile` (page 312) opens the profile corresponding to a specific index into a specific search result list and obtains a reference to that profile; **not recommended in ColorSync 2.5**.
- `CMSearchGetIndProfileFileSpec` (page 313) obtains the file specification for the profile at a specific index into a search result; **not recommended in ColorSync 2.5**.

For additional information on profile searching, see "Searching for a Profile by Profile Identifier" (page 314).

IMPORTANT
Only 2.x profiles are included in the profile search result. ▲

CMNewProfileSearch

NOT RECOMMENDED IN COLORSYNC 2.5

Searches the ColorSync Profiles folder and returns a list of 2.x profiles that match the search specification.

```
pascal CMError CMNewProfileSearch (
                    CMSearchRecord *searchSpec,
                    void *refCon,
                    unsigned long *count,
                    CMProfileSearchRef *searchResult);
```

`searchSpec` A pointer to a search specification. For a description of the information you can provide in a search record of type `CMSearchRecord` to define the search, see `CMSearchRecord` (page 368).

`refCon` An untyped pointer to arbitrary data supplied by your application. `CMNewProfileSearch` passes this data to your filter routine. For a description of the filter routine, see the function `MyCMProfileFilterProc` (page 347).

`count` A pointer to a profile count. On output, a one-based count of profiles matching the search specification.

`searchResult`
A pointer to a search result reference. On output, a reference to the profile search result list. For a description of the `CMProfileSearchRef` private data type, see `CMProfileSearchRef` (page 370).

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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.x profiles that meet the criteria and returns a list of these profiles in an internal private data structure whose reference is returned to you

in the `searchResult` parameter. For a complete description of where the function searches for profiles, see "Profile Search Locations" (page 55).

You must provide a search record of type `CMSearchRecord` identifying the search criteria. You specify which fields of the search record to use 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 filter function to use 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 function `MyCMProfileFilterProc` (page 347).

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. These functions are described in the "See Also" section.

When your application has completed its search, it should call the function `CMDisposeProfileSearch` (page 311) to free the private memory allocated for the search.

VERSION NOTES

The `CMNewProfileSearch` function does not take full advantage of the optimized profile searching available starting with ColorSync version 2.5, as described in "The Profile Cache and Optimized Searching" (page 57). Use `CMIterateColorSyncFolder` (page 304) instead.

SEE ALSO

To obtain a reference to a profile corresponding to a specific index in the list, use the function `CMSearchGetIndProfile` (page 312). To obtain the file specification for a profile corresponding to a specific index in the list, use the function `CMSearchGetIndProfileFileSpec` (page 313). To update the search result list, use the function `CMUpdateProfileSearch` (page 310). To free the private memory allocated for a profile search after your application has completed the search, use the function `CMDisposeProfileSearch` (page 311).

CMUpdateProfileSearch

NOT RECOMMENDED IN COLORSYNC 2.5

Searches the ColorSync Profiles folder and updates an existing search result obtained originally from the `CMNewProfileSearch` 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 `CMProfileSearchRef` (page 370).

`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 function `MyCMProfileFilterProc` (page 347).

`count` A pointer to a profile count. On output, if the function result is `noErr`, a one-based count of the number of profiles matching the original search specification passed to the `CMNewProfileSearch` function. Otherwise undefined.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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. You must use this function 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 modification of the contents of the ColorSync Profiles folder to occur at any time.

VERSION NOTES

Starting with version 2.5, you should use the function `CMIterateColorSyncFolder` (page 304) for profile searching.

SEE ALSO

For a description of the function you call to begin a new search, see the function `CMNewProfileSearch` (page 308). That function specifies the filter function referred to in the description of the `refCon` parameter.

CMDisposeProfileSearch

NOT RECOMMENDED IN COLORSYNC 2.5

Frees the private memory allocated for a profile search after your application has completed the search.

```
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, see `CMProfileSearchRef` (page 370).

function result This routine does not return value.

VERSION NOTES

Starting with version 2.5, you should use the function `CMIterateColorSyncFolder` (page 304) for profile searching.

SEE ALSO

To set up a search, use the function `CMNewProfileSearch` (page 308). To obtain a reference to a profile corresponding to a specific index in the list, use the

function CMSearchGetIndProfile (page 312). To obtain the file specification for a profile corresponding to a specific index in the list, use the function CMSearchGetIndProfileFileSpec (page 313). To update the search result list, use the function CMUpdateProfileSearch (page 310).

CMSearchGetIndProfile

NOT RECOMMENDED IN COLORSYNC 2.5

Opens the profile corresponding to a specific index into a specific search result list and obtains a reference to that profile.

```
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 `CMProfileSearchRef` (page 370).

`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 index must be less than or equal to the value returned as the `count` parameter of the `CMNewProfileSearch` function or the `CMUpdateProfileSearch` function; otherwise `CMSearchGetIndProfile` returns a result code of `cmIndexRangeErr`.

`prof` A pointer to a profile reference of type `CMProfileRef` (page 358). On output, the reference refers to the profile associated with the specified index.

function result A result code of type `CMError`. One possible result code is described with the `index` parameter above. For other possible values, see "Result Codes for the ColorSync Manager" (page 425).

VERSION NOTES

Starting with version 2.5, you should use the function `CMIterateColorSyncFolder` (page 304) for profile searching.

SEE ALSO

Before your application can call the `CMSearchGetIndProfile` function, it must call the function `CMNewProfileSearch` (page 308) 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 function `CMCloseProfile` (page 223).

CMSearchGetIndProfileFileSpec

NOT RECOMMENDED IN COLORSYNC 2.5

Obtains the file specification for the profile at a specific index into a search result.

```
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 `CMProfileSearchRef` (page 370).

`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. The index must be less than or equal to the value returned as the `count` parameter of the `CMNewProfileSearch` function or the `CMUpdateProfileSearch` function; otherwise `CMSearchGetIndProfile` returns a result code of `cmIndexRangeErr`

`profileFile`	A pointer to a file specification. On output, this parameter points to a file specification for the profile at the location specified by `index`. For a description of the `FSSpec` data type, see *Inside Macintosh: Files.*
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Before your application can call the `CMSearchGetIndProfileFileSpec` function, it must call the function `CMNewProfileSearch` (page 308) 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.

VERSION NOTES

Starting with version 2.5, you should use the function `CMIterateColorSyncFolder` (page 304) for profile searching.

Searching for a Profile by Profile Identifier

Starting with version 2.1, the ColorSync Manager defines the structure `CMProfileIdentifier` (page 359), which identifies a profile but takes up much less space than most profiles. A profile identifier can refer to an embedded profile or to a profile file stored in the ColorSync Profiles folder. Your application can embed profile identifiers in place of entire profiles, or in addition to them.

ColorSync provides the following functions for searching for profile identifiers:

- `CMProfileIdentifierFolderSearch` (page 315) searches the ColorSync Profiles folder and returns a list of profile references, one for each profile that matches the specified profile identifier.
- `CMProfileIdentifierListSearch` (page 316) searches a list of profile references and returns a list of all references that match a specified profile identifier.

IMPORTANT

Only version 2.x profiles are included in the profile search result. ▲

For sample code that demonstrates how to use the profile identifier search functions, see "Searching for a Profile That Matches a Profile Identifier" (page 139).

For information on searching for entire profiles, see "Profile Searching" (page 303).

CMProfileIdentifierFolderSearch

Searches the ColorSync Profiles folder and returns a list of profile references, one for each profile that matches the specified profile identifier.

```
pascal CMError CMProfileIdentifierFolderSearch (
                    CMProfileIdentifierPtr ident,
                    unsigned long *matchedCount,
                    CMProfileSearchRef *searchResult);
```

`ident` A pointer to a profile identifier structure specifying the profile to search for.

`matchedCount` A pointer to a value of type `unsigned long`. On output, the one-based count of profiles that match the specified profile identifier. The count is typically 0 or 1, but can be higher.

`searchResult` A pointer to a search result reference of type `CMProfileSearchRef` (page 370). On output, a reference to the profile search result list.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425). It is not an error condition if this function finds no matching profiles. It returns an error only if a File Manager or other low-level system error occurs.

DISCUSSION

When your application or device driver processes an image, it can keep a list of profile references for each profile it encounters in the image. Each time it

encounters an embedded profile identifier, your application can call the function `CMProfileIdentifierListSearch` (page 316) to see if there is already a matching profile reference in its list. If not, it can call the function `CMProfileIdentifierFolderSearch` (page 315) to see if the profile is located in the ColorSync Profiles folder.

Although there should typically be at most one profile in the ColorSync Profiles folder that matches the profile identifier, two or more profiles with different filenames may qualify. It is not considered an error condition if the `CMProfileIdentifierFolderSearch` function finds no matching profiles.

For sample code demonstrating how to use `CMProfileIdentifierListSearch`, see "Searching for a Profile That Matches a Profile Identifier" (page 139).

CMProfileIdentifierListSearch

Searches a list of profile references and returns a list of all references that match a specified profile identifier.

```
pascal CMError CMProfileIdentifierListSearch (
                    CMProfileIdentifierPtr ident,
                    CMProfileRef *profileList,
                    unsigned long listSize,
                    unsigned long *matchedCount,
                    CMProfileRef *matchedList);
```

`ident` A pointer to a profile identifier. The function looks for profile references in `profileList` that match the profile described by this identifier. For information on how a profile identifier match is determined, see `CMProfileIdentifier` (page 359).

`profileList` A pointer to a list of profile references to search.

`listSize` The number of profile references in `profileList`.

`matchedCount` A pointer to a count of matching profile references. If you set `matchedList` to `NULL`, on output `matchedCount` specifies the number of references in `profileList` that match `ident`. The count is typically 0 or 1, but can be higher.
If you do not set `matchedList` to `NULL`, on input you set `matchedCount` to the maximum number of matching references to

be returned in `matchedList`. On output, the value of `matchedCount` specifies the actual number of matching references returned, which is always equal to or less than the number passed in.

`matchedList` A pointer to a list of profile references. If you set `matchedList` to `NULL` on input, on output nothing is returned in the parameter, and the actual number of matching references is returned in `matchedCount`.
If you do *not* set `matchedList` to `NULL` on input, it is treated as a pointer to allocated memory. On output, the allocated memory will contain a list, in no particular order, of profile references that match `ident`. The number of references in the list is equal to or less than the value you pass in the `matchedCount` parameter. You must allocate enough memory for `matchedList` to store the requested number of profile references.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425). It is not an error condition if the `CMProfileIdentifierListSearch` function finds no matching profiles. The function returns an error only if a Memory Manager or other low-level system error occurs.

DISCUSSION

When your application or device driver processes an image, it can keep a list of profile references for each unique profile or profile identifier it encounters in the image. Each time it encounters an embedded profile identifier, your application can call the `CMProfileIdentifierListSearch` function to see if there is already a matching profile reference in the list. Although your list of profile references would normally contain at most one reference that matches the profile identifier, it is possible to have two or more matches. For information on how a profile identifier match is determined, see `CMProfileIdentifier` (page 359).

If no matching profile is found in the list, your application can call the function `CMProfileIdentifierFolderSearch` (page 315) to see if a matching profile can be found in the ColorSync Profiles folder.

To determine the amount of memory needed for the list of profile references that match a profile identifier, your application may want to call `CMProfileIdentifierListSearch` twice. The first time, on input you set `matchedList` to `NULL` and ignore `matchedCount`. On output, `matchedCount` specifies

the number of matching profiles. You then allocate enough memory to hold that many profile references (or a smaller number if you don't want all the references) and call `CMProfileIdentifierListSearch` again. This time you set `matchedList` to a pointer to the allocated memory and set `matchedCount` to the number of references you wish to obtain. To allocate memory, you use code such as the following:

```
myProfileRefListPtr = NewPtr(sizeof(CMProfileRef) * matchedCount);
```

If your application is interested in obtaining only the first profile that matches the specified profile, you need call `CMProfileIdentifierListSearch` only once. To do so, you just allocate enough memory to store one profile reference, set `matchedList` to point to that memory (or just set `matchedList` to point to a local variable), and set `matchedCount` to 1. On return, if `matchedCount` still has the value 1, then `CMProfileIdentifierListSearch` found a matching profile.

For sample code demonstrating how to use `CMProfileIdentifierFolderSearch`, see "Searching for a Profile That Matches a Profile Identifier" (page 139).

Converting Between Color Spaces

See "Converting Between Color Spaces" (page 65) for a description of the color conversion capabilities the ColorSync Manager provides. That section also describes color conversion prior to ColorSync version 2.1.

The ColorSync Manager provides the following functions to convert colors between a base color space and any of its derived color spaces or between two derivatives of the same base family.

- `CMConvertXYZToLab` (page 319) converts colors specified in the XYZ color space to the L*a*b* color space.
- `CMConvertLabToXYZ` (page 320) converts colors specified in the L*a*b* color space to the XYZ color space.
- `CMConvertXYZToLuv` (page 321) converts colors specified in the XYZ color space to the L*u*v* color space.
- `CMConvertLuvToXYZ` (page 322) converts colors specified in the L*u*v* color space to the XYZ color space.
- `CMConvertXYZToYxy` (page 323) converts colors specified in the XYZ color space to the Yxy color space.

- `CMConvertYxyToXYZ` (page 324) converts colors specified in the Yxy color space to the XYZ color space.
- `CMConvertXYZToFixedXYZ` (page 325) converts 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`.
- `CMConvertFixedXYZToXYZ` (page 326) converts 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`.
- `CMConvertRGBToHLS` (page 327) converts colors specified in the RGB color space to equivalent colors defined in the HLS color space.
- `CMConvertHLSToRGB` (page 328) converts colors specified in the HLS color space to equivalent colors defined in the RGB color space.
- `CMConvertRGBToHSV` (page 329) converts colors specified in the RGB color space to equivalent colors defined in the HSV color space when the device types are the same.
- `CMConvertHSVToRGB` (page 330) converts colors specified in the HSV color space to equivalent colors defined in the RGB color space.
- `CMConvertRGBToGray` (page 331) converts colors specified in the RGB color space to equivalent colors defined in the Gray color space.

CMConvertXYZToLab

Converts colors specified in the XYZ color space to the L*a*b* color space.

```
pascal CMError CMConvertXYZToLab (
                    const CMColor *src,
                    const CMXYZColor *white,
                    CMColor *dst,
                    unsigned long count);
```

`src` A pointer to an array containing the list of XYZ colors to convert to L*a*b* colors.

`white` A pointer to a 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 convert.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMConvertXYZToLab` function converts one or more colors defined in the XYZ color space to equivalent colors defined in the L*a*b* color space. Both color spaces are device independent.

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 `CMConvertXYZToLab` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information about the color conversion routines in previous versions of ColorSync, see "Converting Between Color Spaces" (page 318).

CMConvertLabToXYZ

Converts colors specified in the L*a*b* color space to the XYZ color space.

```
pascal CMError CMConvertLabToXYZ (
                    const CMColor *src,
                    const CMXYZColor *white,
                    CMColor *dst,
                    unsigned long count);
```

`src`	A pointer to a buffer containing the list of L*a*b* colors to convert to XYZ colors.
`white`	A pointer to a 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 convert.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMConvertLabToXYZ` function converts one or more colors defined in the L*a*b color space to equivalent colors defined in the XYZ color space. Both color spaces are device independent.

SEE ALSO

For information about the color conversion routines in previous versions of ColorSync, see "Converting Between Color Spaces" (page 318).

CMConvertXYZToLuv

Converts colors specified in the XYZ color space to the L*u*v* color space.

```
pascal CMError CMConvertXYZToLuv (
                    const CMColor *src,
                    const CMXYZColor *white,
                    CMColor *dst,
                    unsigned long count);
```

`src`	A pointer to an array containing the list of XYZ colors to convert to L*u*v* colors.
`white`	A pointer to a 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 convert.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMConvertXYZToLuv` function converts one or more colors defined in the XYZ color space to equivalent colors defined in the L*u*v* color space. Both color spaces are device independent.

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 `CMConvertXYZToLuv` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information about the color conversion routines in previous versions of ColorSync, see “Converting Between Color Spaces” (page 318).

CMConvertLuvToXYZ

Converts colors specified in the L*u*v* color space to the XYZ color space.

```
pascal CMError CMConvertLuvToXYZ (
                    const CMColor *src,
                    const CMXYZColor *white,
                    CMColor *dst,
                    unsigned long count);
```

`src`	A pointer to an array containing the list of L*u*v* colors to convert.
`white`	A pointer to a 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 convert.
function result	A result code of type `CMError`. For possible values, see “Result Codes for the ColorSync Manager” (page 425).

DISCUSSION

The `CMConvertLuvToXYZ` function converts one or more colors defined in the L*u*v color space to equivalent colors defined in the XYZ color space. Both color spaces are device independent.

SEE ALSO

For information about the color conversion routines in previous versions of ColorSync, see “Converting Between Color Spaces” (page 318).

CMConvertXYZToYxy

Converts colors specified in the XYZ color space to the Yxy color space.

```
pascal CMError CMConvertXYZToYxy (
                    const CMColor *src,
                    CMColor *dst,
                    unsigned long count);
```

`src`	A pointer to an array containing the list of XYZ colors to convert 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 convert.
function result	A result code of type `CMError`. For possible values, see “Result Codes for the ColorSync Manager” (page 425).

DISCUSSION

The `CMConvertXYZToYxy` function converts one or more colors defined in the XYZ color space to equivalent colors defined in the Yxy color space. Both color spaces are device independent.

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 `CMConvertXYZToYxy` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information about the color conversion routines in previous versions of ColorSync, see “Converting Between Color Spaces” (page 318).

CMConvertYxyToXYZ

Converts colors specified in the Yxy color space to the XYZ color space.

```
pascal CMError CMConvertYxyToXYZ (
                    const CMColor *src,
                    CMColor *dst,
                    unsigned long count);
```

`src`	A pointer to an array containing the list of Yxy colors to convert.
`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 convert.
function result	A result code of type `CMError`. For possible values, see “Result Codes for the ColorSync Manager” (page 425).

DISCUSSION

The `CMConvertYxyToXYZ` function converts one or more colors defined in the Yxy color space to equivalent colors defined in the XYZ color space. Both color spaces are device independent.

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 `CMConvertYxyToXYZ` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information about the color conversion routines in previous versions of ColorSync, see “Converting Between Color Spaces” (page 318).

CMConvertXYZToFixedXYZ

Converts 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`.

```
pascal CMError CMConvertXYZToFixedXYZ (
                    const CMXYZColor *src,
                    CMFixedXYZColor *dst,
                    unsigned long count);
```

`src`	A pointer to an array containing the list of XYZ colors to convert 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 convert.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMConvertXYZToFixedXYZ` 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 XYZ color space is device independent.

SEE ALSO

For information about the color conversion routines in previous versions of ColorSync, see "Converting Between Color Spaces" (page 318).

CMConvertFixedXYZToXYZ

Converts 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`.

```
pascal CMError CMConvertFixedXYZToXYZ (
                    const CMFixedXYZColor *src,
                    CMXYZColor *dst,
                    unsigned long count);
```

`src`	A pointer to an array containing the list of Fixed XYZ colors to convert 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 convert.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMConvertFixedXYZToXYZ` function converts one or more colors defined in the Fixed XYZ color space to equivalent colors defined in the XYZ color space. The XYZ color space is device independent.

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 `CMConvertFixedXYZToXYZ` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information about the color conversion routines in previous versions of ColorSync, see "Converting Between Color Spaces" (page 318).

CMConvertRGBToHLS

Converts colors specified in the RGB color space to equivalent colors defined in the HLS color space.

```
pascal CMError CMConvertRGBToHLS (
                    const CMColor *src,
                    CMColor *dst,
                    unsigned long count);
```

`src`	A pointer to an array containing the list of RGB colors to convert 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 convert.
function result	A result code of type `CMError`. For possible values, see “Result Codes for the ColorSync Manager” (page 425).

DISCUSSION

The `CMConvertRGBToHLS` function converts one or more colors defined in the RGB color space to equivalent colors defined in the HLS color space. Both color spaces are device dependent.

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 `CMConvertRGBToHLS` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information about the color conversion routines in previous versions of ColorSync, see “Converting Between Color Spaces” (page 318).

CMConvertHLSToRGB

Converts colors specified in the HLS color space to equivalent colors defined in the RGB color space.

```
pascal CMError CMConvertHLSToRGB (
                    const CMColor *src,
                    CMColor *dst,
                    unsigned long count);
```

`src`	A pointer to an array containing the list of HLS colors to convert 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 convert.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMConvertHLSToRGB` function converts one or more colors defined in the HLS color space to equivalent colors defined in the RGB color space. Both color spaces are device dependent.

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 `CMConvertHLSToRGB` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information about the color conversion routines in previous versions of ColorSync, see "Converting Between Color Spaces" (page 318).

CMConvertRGBToHSV

Converts colors specified in the RGB color space to equivalent colors defined in the HSV color space when the device types are the same.

```
pascal CMError CMConvertRGBToHSV (
                    const CMColor *src,
                    CMColor *dst,
                    unsigned long count);
```

`src`	A pointer to an array containing the list of RGB colors to convert 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 convert.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMConvertRGBToHSV` function converts one or more colors defined in the RGB color space to equivalent colors defined in the HSV color space. Both color spaces are device dependent.

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 `CMConvertRGBToHSV` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information about the color conversion routines in previous versions of ColorSync, see "Converting Between Color Spaces" (page 318).

CMConvertHSVToRGB

Converts colors specified in the HSV color space to equivalent colors defined in the RGB color space.

```
pascal CMError CMConvertHSVToRGB (
                    const CMColor *src,
                    CMColor *dst,
                    unsigned long count);
```

`src`	A pointer to an array containing the list of HSV colors to convert 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 convert.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMConvertHSVToRGB` function converts one or more colors defined in the HSV color space to equivalent colors defined in the RGB color space. Both color spaces are device dependent.

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 `CMConvertHSVToRGB` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information about the color conversion routines in previous versions of ColorSync, see "Converting Between Color Spaces" (page 318).

CMConvertRGBToGray

Converts colors specified in the RGB color space to equivalent colors defined in the Gray color space.

```
pascal CMError CMConvertRGBToGray (
                    const CMColor *src,
                    CMColor *dst,
                    unsigned long count);
```

`src`	A pointer to an array containing the list of colors specified in RGB space to convert 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 convert.
function result	A result code of type `CMError`. For possible values, see “Result Codes for the ColorSync Manager” (page 425).

DISCUSSION

The `CMConvertRGBToGray` function converts one or more colors defined in the RGB color space to equivalent colors defined in the Gray color space. Both color spaces are device dependent.

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 `CMConvertRGBToGray` function to overwrite the source colors with the resulting converted color specifications.

SEE ALSO

For information about the color conversion routines in previous versions of ColorSync, see “Converting Between Color Spaces” (page 318).

Color-Matching With PostScript Devices

The ColorSync Manager provides three functions that support color matching by PostScript Level 2 devices. The default CMM implements these functions if the preferred CMM corresponding to the profile does not.

- `CMGetPS2ColorSpace` (page 333) obtains 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.
- `CMGetPS2ColorRenderingIntent` (page 335) obtains 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.
- `CMGetPS2ColorRendering` (page 336) obtains the color rendering dictionary (CRD) element data usable as the parameter to the PostScript `setColorRendering` operator, which specifies the PostScript color rendering dictionary to use for the following graphics data.
- `CMGetPS2ColorRenderingVMSize` (page 338) determines the virtual memory size of the color rendering dictionary (CRD) for a printer profile before your application or driver obtains the CRD and sends it to the printer.

Starting with PostScript version 2016, to provide better support for ColorSync and ICC profiles, Postscript Level 2 supports up to four-component color spaces through the addition of CIEBasedDEF and CIEBasedDEFG color spaces.

To use these new color spaces, starting with ColorSync version 2.1, the `CMGetPS2ColorSpace` function supports profiles with four components, as well as scanner and monitor profiles that contain multidimensional table information. In previous versions of ColorSync, routines such as `CMGetPS2ColorSpace` returned an error if asked to generate PostScript code for a profile with more than three components.

The CIEBasedDEF and CIEBasedDEFG color spaces are extensions to the CIEBasedABC color space. To work with these color spaces, PostScript defines the RangeDEF, RangeHIJK, DecodeDEFG, and Table arrays. You can read more about how these arrays are used to convert CIEBasedDEF and CIEBasedDEFG color space values in the *PostScript Language Reference Manual Supplement*, version 2016.

IMPORTANT

If you use ColorSync to generate PostScript output for CIEBasedDEF and CIEBasedDEFG color spaces, be sure the printer has PostScript version 2016 or later. ▲

CMGetPS2ColorSpace

Obtains 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.

```
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.

`flags` If the value of `flags` is equal to `cmPS8bit`, the generated PostScript will utilize 8-bit encoding whenever possible to achieve higher data compaction. If the value of `flags` is *not* equal to `cmPS8bit`, the generated data will be 7-bit safe, in either ASCII or ASCII base-85 encoding.

`proc` A pointer to a callback flatten function to receive the PostScript data from the CMM. For information, see the function `MyColorSyncDataTransfer` (page 342).

`refCon` An untyped pointer to arbitrary data supplied by your application. `CMGetPS2ColorSpace` passes this data in calls to your `MyColorSyncDataTransfer` (page 342) function.

`preferredCMMnotfound`
A pointer to a flag for whether the preferred CMM was found. On output, has the value `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, has the value `false`.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMGetPS2ColorSpace` function obtains PostScript color space data from the source profile. The valid profile classes for the `CMGetPS2ColorSpace` function are display, input, and output profiles with at most four components.

To determine which profile elements to use to generate the PostScript color space data, the CMM:

- uses the PostScript `cmPS2CSATag`, if it exists
- otherwise, uses the multidimensional table tag (`cmAToB0`, `cmAToB1`, or `cmAToB2`), if it exists, for the rendering intent currently specified by the profile
- otherwise, uses the multidimensional table tag `cmAToB0`, if it exists
- otherwise, for display profiles only, uses the tristimulus tags (`cmRedColorantTag`, `cmGreenColorantTag`, `cmBlueColorantTag`) and the tonal curve tags (`cmRedTRCTag`, `cmGreenTRCTag`, and `cmBlueTRCTag`)

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 pass 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 default CMM.

CMGetPS2ColorRenderingIntent

Obtains 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.

```
pascal CMError CMGetPS2ColorRenderingIntent (
                    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.

`flags` If the value of `flags` is equal to `cmPS8bit`, the generated PostScript will utilize 8-bit encoding whenever possible to achieve higher data compaction. If the value of `flags` is *not* equal to `cmPS8bit`, the generated data will be 7-bit safe, in either ASCII or ASCII base-85 encoding.

`proc` A low-level data transfer function supplied by the calling application to receive the PostScript data from the CMM. For more information, see the function `MyColorSyncDataTransfer` (page 342).

`refCon` An untyped pointer to arbitrary data supplied by your application. `CMGetPS2ColorSpace` passes this data in calls to your `MyColorSyncDataTransfer` (page 342) function.

`preferredCMMnotfound`
A pointer to a flag for whether the preferred CMM was found. On output, has the value `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, has the value `false`.

function result A result code of type `CMError`. For possible values, see “Result Codes for the ColorSync Manager” (page 425).

DISCUSSION

The `CMGetPS2ColorRenderingIntent` function obtains PostScript rendering intent information from the header of the source profile. It returns data by calling your low-level data transfer procedure and passing the PostScript data to it. Typically, your low-level data transfer function returns this data to the calling application or device driver to pass 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 function to the default CMM.

CMGetPS2ColorRendering

Obtains the color rendering dictionary (CRD) element data usable as the parameter to the PostScript `setColorRendering` operator, which specifies the PostScript color rendering dictionary to use for the following graphics data.

```
pascal CMError CMGetPS2ColorRendering (
                    CMProfileRef srcProf,
                    CMProfileRef dstProf,
                    unsigned long flags,
                    CMFlattenUPP proc,
                    void *refCon,
                    Boolean *preferredCMMnotfound);
```

`srcProf` A profile reference to a profile that supplies the rendering intent for the CRD.

`dstProf` A profile reference to a profile from which to extract the CRD data.

`flags` If the value of `flags` is equal to `cmPS8bit`, the generated PostScript will utilize 8-bit encoding whenever possible to achieve higher data compaction. If the value of `flags` is *not* equal to `cmPS8bit`, the generated data will be 7-bit safe, in either ASCII or ASCII base-85 encoding.

`proc`	A pointer to a callback flatten function to perform the data transfer. For information, see the function `MyColorSyncDataTransfer` (page 342).
`refCon`	An untyped pointer to arbitrary data supplied by your application. `CMGetPS2ColorSpace` passes this data in calls to your `MyColorSyncDataTransfer` (page 342) function.
`preferredCMMnotfound`	A pointer to a flag for whether the preferred CMM was found. On output, has the value `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, has the value `false`.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMGetPS2ColorRendering` function obtains CRD data from the profile specified by the `dstProf` parameter. To be valid, the parameter must specify an output profile with at most four components. The CMM uses the rendering intent from the profile specified by the `srcProf` parameter to determine which of the PostScript tags (`ps2CR0Tag`, `ps2CR1Tag`, `ps2CR2Tag`, or `ps2CR3Tag`) to use in creating the CRD. If none of these tags exists in the profile, the CMM creates the CRD from one of the multidimensional table tags (`cmAToB0`, `cmAToB1`, or `cmAToB2`), again chosen according to the rendering intent of the profile specified by the `srcProf` parameter.

This 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, the ColorSync Manager dispatches this function to the default CMM.

The CMM obtains the PostScript data and passes it to your low-level data transfer procedure, specified by the `proc` parameter. 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 pass to a PostScript printer.

SEE ALSO

Before your application or device driver sends the CRD to the printer, it can call the function `CMGetPS2ColorRenderingVMSize` (page 338) to determine the virtual memory size of the CRD.

CMGetPS2ColorRenderingVMSize

Determines the virtual memory size of the color rendering dictionary (CRD) for a printer profile before your application or driver obtains the CRD and sends it to the printer.

```
pascal CMError CMGetPS2ColorRenderingVMSize (
                    CMProfileRef srcProf,
                    CMProfileRef dstProf,
                    unsigned long *vmSize,
                    Boolean *preferredCMMnotfound);
```

`srcProf` A profile reference to a profile that supplies the rendering intent for the CRD.

`dstProf` A profile reference to the destination printer profile.

`vmSize` A pointer to a memory size. On return, the virtual memory size of the CRD.

`preferredCMMnotfound`
A pointer to a flag for whether the preferred CMM was found. On output, has the value `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, has the value `false`.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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 “CMConcatProfileSet” (page 384), then the default 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 can be larger than the actual maximum VM size.

The CMM uses the profile specified by the `srcProf` parameter to determine the rendering intent to use.

Converting 2.x Profiles to 1.0 Format

The ColorSync Manager provides the `CMConvertProfile2to1` function to convert 2.x format profiles to the 1.0 profile format. These format version numbers are described in “ColorSync and ICC Profile Format Version Numbers” (page 50).

Because 1.0 and 2.x scanner and monitor profiles generally carry the same required color information, converting between them will not result in lost accuracy. With printer profiles, however, some accuracy may be lost by conversion, leading to significantly different results. Because of the possible loss of accuracy in some cases, 2.x to1.0 profile conversion is not encouraged.

Note
ColorSync fully supports 1.0 format profiles, but this support is not guaranteed to continue in future versions. Apple strongly recommends that developers using the 1.0 format move to the 2.x format. ◆

CMConvertProfile2to1

Converts the specified ColorSync profile from the 2.x format to the 1.0 format.

```
pascal CMError CMConvertProfile2to1 (
                    CMProfileRef profv2,
                    CMProfileHandle *profv1);
```

`profv2` A reference to a ColorSync 2.x format profile to convert to 1.0 format.

`profv1`	A pointer to a profile handle. On output, the handle contains a 1.0 format version of the 2.x format profile referred to by `profv2`. In some cases there may be loss of information in creating the version 1.0 profile.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

Application-Defined Functions for the ColorSync Manager

Your application supplies the following functions for use with ColorSync Manager functions. The ColorSync Manager functions that use your functions take a universal procedure pointer to your function as an input parameter.

- `MyProfileIterateProc` (page 340) is called once for each found profile by the `CMIterateColorSyncFolder` (page 304) function as it iterates over the available profiles; **new in ColorSync 2.5**.
- `MyColorSyncDataTransfer` (page 342) transfers profile data from the format for embedded profiles to disk file format or vice versa.
- `MyCMBitmapCallBackProc` (page 345) reports on the progress of a color-matching or color-checking session being performed for a bitmap or a pixel map.
- `MyCMProfileFilterProc` (page 347) examines the profile whose reference you specify and determines whether to include it in the profile search result list.
- `MyCMProfileAccessProc` (page 348) provides procedure-based access to a profile.

MyProfileIterateProc

NEW IN COLORSYNC 2.5

Application-defined function that the `CMIterateColorSyncFolder` (page 304) function calls once for each found profile file as it iterates over the available profiles. Used, for example, to obtain a list of profiles to display in a pop-up menu.

This application-supplied function must conform to the following declaration, although the function name is arbitrary:

```
pascal OSErr MyProfileIterateProc (
                    CMProfileIterateData *iterateData,
                    void *refCon);
```

`iterateData` A pointer to a structure of type `CMProfileIterateData` (page 366). When the function `CMIterateColorSyncFolder` (page 304) calls `MyProfileIterateProc`, as it does once for each found profile, the structure contains key information about the profile.

`refCon` An untyped pointer to arbitrary data your application previously passed to the function `CMIterateColorSyncFolder` (page 304).

function result A result code of type `OSErr`. If `MyProfileIterateProc` returns an error, `CMIterateColorSyncFolder` stops iterating and returns the error value to its caller (presumably your code). For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

When your application needs information about the profiles currently available in the profiles folder, it calls the function `CMIterateColorSyncFolder` (page 304), which, depending on certain conditions, calls your callback routine once for each profile. See the description of `CMIterateColorSyncFolder` for information on when it calls the `MyProfileIterateProc` routine.

Your `MyProfileIterateProc` routine examines the structure pointed to by the `iterateData` parameter to obtain information about the profile it describes. The routine determines whether to do anything with that profile, such as list its name in a pop-up menu of available profiles.

MyColorSyncDataTransfer

CHANGED IN COLORSYNC 2.5

Application-defined function that transfers profile data from the format for embedded profiles to disk file format or vice versa. Used, for example, by 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, although the function name is arbitrary:

```
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` A pointer to a size value. On input, the size in bytes of the data to transfer. On output, the size of the data actually transferred.

`data` A pointer to the buffer supplied by the ColorSync Manager to use for the data transfer.

`refCon` A reference constant that holds the application data passed in from the functions `CMFlattenProfile` (page 237), `CMUnflattenProfile` (page 239), `CMGetPS2ColorSpace` (page 333), `CMGetPS2ColorRenderingIntent` (page 335), or `CMGetPS2ColorRendering` (page 336). Each time the CMM calls your `MyColorSyncDataTransfer` function, it passes this data to the function.
Starting in ColorSync version 2.5, the ColorSync Manager calls your function directly, without going through the preferred, or any, CMM.

function result A result code of type `OSErr`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Starting in ColorSync version 2.5, the ColorSync Manager calls your data transfer function directly, without going through the preferred, or any, CMM. So any references to the CMM in the discussion that follows are applicable only to versions of ColorSync prior to version 2.5. Where the discussion does not involve CMMs, it is applicable to all versions of ColorSync.

Your `MyColorSyncDataTransfer` function is called to flatten and unflatten profiles or to transfer PostScript-related data from a profile to the PostScript format to send to an application or device driver.

The ColorSync Manager and the CMM communicate with the `MyColorSyncDataTransfer` function using the command parameter to identify the operation to perform. To read and write profile data, your function must support the following commands: `openReadSpool`, `openWriteSpool`, `readSpool`, `writeSpool`, and `closeSpool`.

You determine the behavior of your `MyColorSyncDataTransfer` function. The following sections describe 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 CMM, dispatching the `CMFlattenProfile` function to it.

The CMM communicates with the `MyColorSyncDataTransfer` function using a command parameter to identify the operation to perform. 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 is embedded in a graphics document to an independent disk file, the ColorSync Manager calls your `MyColorSyncDataTransfer` function with the following sequence of commands: `openReadSpool`, `readSpool`, `closeSpool`. Here is how you should handle these commands:

- When the ColorSync Manager calls your function with the `openReadSpool` command, you should perform any initialization required to read from the embedded profile format.
- The ColorSync Manager calls 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 ColorSync Manager in the `data` buffer. For each call, the ColorSync Manager specifies in the `size` parameter the number of bytes of data you should return. Each time your function is called it should read and return the requested data; it should also specify in the `size` parameter the actual number of bytes of data it returns.
- When the ColorSync Manager calls your function with the `closeSpool` command, you should perform any required cleanup processes.

VERSION NOTES

Starting in ColorSync version 2.5, the ColorSync Manager calls your function directly, without going through the preferred, or any, CMM.

MyCMBitmapCallBackProc

Application-defined function that 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, although the function name is arbitrary:

```
pascal Boolean MyCMBitmapCallBackProc (
                    long progress,
                    void *refCon);
```

`progress` A byte count that begins at an arbitrary value when the function is first called. On each subsequent call, the value is decremented by an amount that can vary from call to call, but that reflects how much of the matching process has completed since the previous call. If the function is called at all, it will be called a final time with a byte count of 0 when the matching is complete.

`refCon` The reference constant passed to your `MyCMBitmapCallBackProc` function each time the color management module (CMM) calls your function.

function result A result code of type `Boolean`. A return value of `false` indicates the color-matching or color-checking session should continue. A return value of `true` indicates the session should be aborted—for example, the user may be holding down the Command–period keys.

DISCUSSION

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` (page 272), `CWCheckPixMap` (page 274), `CWMatchBitmap` (page 276), and `CWCheckBitMap` (page 279). Note that your callback function may not be called at all if the operation completes in a very short period.

The CMM used for the color-matching session calls your function at regular intervals. For example, the default CMM calls your function approximately every half-second unless the color matching or checking occurs in less time; this happens when there is a small amount of data to match or check.

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 can display a progress bar or other indicator to show how much of the operation has been completed. You might, for example, use the reference constant to pass to the callback function a window reference to a dialog box. You obtain information on how much of the operation has completed from the `progress` parameter. The first time your callback is called, this parameter contains an arbitrary byte count. On each subsequent call, the value is decremented by an amount that can vary from call to call, but that reflects how much of the matching process has completed since the previous call. Using the current value and the original value, you can determine the percentage that has completed. If the callback function is called at all, it will be called a final time with a byte count of 0 when the matching is complete.

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 0 to indicate the completion of the matching or checking process. You should use the 0 byte count as a signal to perform any cleanup operations your function requires, such as filling the 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

Application-defined function that examines the profile whose reference you specify and determines whether to include it in the profile search result list.

After a profile has been included in the profile search result based on criteria specified in the search record, your `MyCMProfileFilterProc` function can further examine the profile. For example, you may wish to include or exclude the profile 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, although the function name is arbitrary:

```
pascal Boolean MyCMProfileFilterProc (
                    CMProfileRef prof,
                    void *refCon);
```

`prof` A profile reference of type `CMProfileRef` (page 358) to the profile to test.

`refCon` A reference constant that holds data passed through from the `CMNewProfileSearch` function or the `CMUpdateProfileSearch` function.

function result A result code of type `Boolean`. A return value of `false` indicates that the profile should be included. A return value of `true` indicates that the profile should be filtered out.

DISCUSSION

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 `CMNewProfileSearch`, it passes a reference to a search specification record of type `CMSearchRecord` of type `CMSearchRecord` (page 368) that contains a `filter` field. If the `filter` field contains a pointer to your `MyCMProfileFilterProc` function, then your function is called to determine whether to exclude a profile 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.

MyCMProfileAccessProc

Application-defined function that provides procedure-based access to a profile.

When your application calls the `CMOpenProfile`, `CMNewProfile`, `CMCopyProfile`, or `CMNewLinkProfile` functions, it may supply the ColorSync Manager with a profile location structure of type `CMProfileLocation` (page 362) that specifies a procedure that provides access to a profile. In the structure, you provide a universal procedure pointer to a profile access procedure supplied by you and, optionally, a pointer to data your procedure can use. The ColorSync Manager calls your procedure when the profile is created, initialized, opened, read, updated, or closed.

The profile access procedure supplied by your application must conform to the following declaration, although the procedure name is arbitrary.

```
pascal OSErr MyCMProfileAccessProc (
                    long command,
                    long offset,
                    long *size,
                    void *data,
                    void *refConPtr);
```

`command` A command value indicating the operation to perform. Operation constants are described in "Profile Access Procedure Operation Codes" (page 395).

`offset` For read and write operations, the offset from the beginning of the profile at which to read or write data.

`size` A pointer to a size value. On input, for the `cmReadAccess` and `cmWriteAccess` command constants, a pointer to a value indicating the number of bytes to read or write; for the `cmOpenWriteAccess` command, the total size of the profile. On output, after reading or writing, the actual number of bytes read or written.

`data`	A pointer to a buffer containing data to read or write. On output, for a read operation, contains the data that was read.
`refConPtr`	A reference constant pointer that can store private data for the `MyCMProfileAccessProc` procedure.
function result	A result code of type `OSErr`. If an error occurs during processing, your routine returns the appropriate error value. If no error occurs, it returns `noErr`.

DISCUSSION

When the ColorSync Manager calls your profile access procedure, it passes a constant indicating the operation to perform. The operations include creating a new profile, reading from the profile, writing the profile, and so on. Operation constants are described in "Profile Access Procedure Operation Codes" (page 395). Your procedure must be able to respond to each of these constants.

Data Types for the ColorSync Manager

This section describes the data types defined by the ColorSync Manager for your application's use. The types are organized into the following categories:

- “Video Card Gamma” (page 386); **new in ColorSync 2.5**
- “Color Matching While Printing” (page 390)
- “Color Rendering Dictionary Virtual Memory Size” (page 390)

Date and Time

The ColorSync Manager defines the `CMDateTime` type for specifying a date and time.

CMDateTime

The ColorSync Manager defines the `CMDateTime` data structure to specify a date and time in year, month, day of the month, hours, minutes, and seconds. Other ColorSync structures use the `CMDateTime` structure to specify information such as the creation date or calibration date for a color space profile.

The `CMDateTime` structure is similar to the Macintosh Toolbox structure `DateTimeRec`, and like it, is intended to hold date and time values only for a Gregorian calendar.

Note
The `CMDateTime` structure is platform independent. However, when used with Macintosh Toolbox routines such as `SecondsToDate` and `DateToSeconds`, which use seconds to designate years, the range of years that can be represented is limited. ◆

```
struct CMDateTime {
    unsigned short  year;
    unsigned short  month;
    unsigned short  dayOfTheMonth;
    unsigned short  hours;
    unsigned short  minutes;
    unsigned short  seconds;
};
```

Field descriptions

`year`	The year. Note that to indicate the year 1984, this field would store the integer 1984, not just 84.
`month`	The month of the year, where 1 represents January, and 12 represents December.
`dayOfTheMonth`	The day of the month, ranging from 1 to 31.
`hours`	The hour of the day, ranging from 0 to 23, where 0 represents midnight and 23 represents 11:00 P.M.
`minutes`	The minutes of the hour, ranging from 0 to 59.
`seconds`	The seconds of the minute, ranging from 0 to 59.

Profile Header

The ColorSync Manager defines a profile header type for version 2.x profiles, a separate header for version 1.0 profiles, and a header union that can provide access to either a version 2.x or a version 1.0 profile. For more information on profile version numbers, see "ColorSync and ICC Profile Format Version Numbers" (page 50).

- `CMHeader` (page 351) defines the version 1.0 profile header; **not recommended starting with ColorSync 2.0**.
- `CM2Header` (page 354) supports the header format specified by the ICC format specification for version 2.x profiles.
- `CMAppleProfileHeader` (page 357) provides access to both version 2.x and version 1.0 profiles.

CMHeader

NOT RECOMMENDED

ColorSync 1.0 defined a version 1.0 profile whose structure and format are different from that of the ICC version 2.x profile. The `CMHeader` data type represents the version 1.0 profile header. For more information on profile version numbers, see "ColorSync and ICC Profile Format Version Numbers" (page 50). To obtain a copy of the *International Color Consortium Profile Format*

Specification, or to get other information about the ICC, visit the ICC Web site at <http://www.color.org/>.

Your application cannot use ColorSync Manager functions to update a version 1.0 profile or to search for version 1.0 profiles. However, your application can use other ColorSync Manager functions that operate on version 1.0 profiles. For example, your application can open a version 1.0 profile using the function `CMOpenProfile` (page 222), obtain the version 1.0 profile header using the function `CMGetProfileHeader` (page 245), and access version 1.0 profile elements using the function `CMGetProfileElement` (page 243).

To make it possible to operate on both version 1.0 profiles and version 2.x profiles, the ColorSync Manager defines the union `CMAppleProfileHeader` (page 357), which supports either profile header version. The `CMHeader` data type defines the version 1.0 profile header, while the `CM2Header` (page 354) data type defines the version 2.x profile header.

```
struct CMHeader {
    unsigned long       size;                   /* byte size of profile */
    OSType              CMMType;                /* signature of preferred CMM */
    unsigned long       applProfileVersion;     /* Apple profile version */
    OSType              dataType;               /* type of color data, such as rgb */
    OSType              deviceType;             /* device type, such as monitor */
    OSType              deviceManufacturer;     /* device manufacturer */
    unsigned long       deviceModel;            /* as specified by manufacturer */
    unsigned long       deviceAttributes[2];    /* private info on ink, paper, etc. */
    unsigned long       profileNameOffset;      /* offset to name from top of data */
    unsigned long       customDataOffset;       /* offset to custom data from top */
    CMMatchFlag         flags;                  /* misc. info used by drivers */
    CMMatchOption       options;                /* matching type, such as perceptual */
    CMXYZColor          white;                  /* white point in XYZ space */
    CMXYZColor          black;                  /* black point in XYZ space */
};
```

Field descriptions

`size` The total size in bytes of the profile, including any custom data.

`CMMType` The signature of the preferred CMM for color-matching and color-checking sessions for this profile. To avoid conflicts with other CMMs, this signature must be registered with the ICC. For the signature of the default

CMM, see “Signature of ColorSync’s Default Color Management Module” (page 397).

`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 flags:

```
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.)

Use the `CMTurnOffCache` flag for 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 preferred matching for this profile; the default is `CMPerceptualMatch`; other values are `CMColorimetricMatch` or `CMSaturationMatch`. The options are set by the image creator.
`white`	The profile illuminant white reference point, expressed in the XYZ color space.
`black`	The black reference point for this profile, expressed in the XYZ color space.

For more information on ColorSync 1.0 headers, see "How ColorSync 1.0 Profiles and Version 2.x Profiles Differ" (page 531).

VERSION NOTES

Use of the `CMHeader` type is not recommended for ColorSync versions starting with 2.0. Use `CM2Header` (page 354) instead.

CM2Header

The ColorSync Manager defines the `CM2header` profile structure to support the header format specified by the ICC format specification for version 2.x profiles. For more information on profile version numbers, see "ColorSync and ICC Profile Format Version Numbers" (page 50). For a description of `CMHeader`, the ColorSync 1.0 profile header, see `CMHeader` (page 351). To obtain a copy of the *International Color Consortium Profile Format Specification*, or to get other information about the ICC, visit the ICC Web site at <http://www.color.org/>.

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 function `CMGetProfileHeader` (page 245) and replace the header using the function `CMSetProfileHeader` (page 254).

```
struct CM2Header {
    unsigned long       size;                   /* total size of profile */
    OSType              CMMType;                /* CMM signature, registered with
                                                    CS2 consortium */
    unsigned long       profileVersion;         /* version of the profile format */
```

```
    OSType              profileClass;           /* input, display, output, devicelink,
                                                    abstract, color conversion, or
                                                    named color profile class */
    OSType              dataColorSpace;         /* color space of data */
    OSType              profileConnectionSpace; /* profile connection color space */
    CMDateTime          dateTime;               /* date & time of profile creation */
    OSType              CS2profileSignature;    /* 'acsp' constant, required by ICC */
    OSType              platform;               /* primary profile platform, registered
                                                    with CS2 consortium */
    unsigned long       flags;                  /* gives hints for certain options */
    OSType              deviceManufacturer;     /* registered with ICC consortium */
    unsigned long       deviceModel;            /* registered with ICC consortium */
    unsigned long       deviceAttributes[2];    /* attributes such as paper type */
    unsigned long       renderingIntent;        /* preferred rendering intent of object
                                                    tagged with this profile */
    CMFixedXYZColor     white;                  /* profile illuminant */
    OSType              creator;                /* profile creator */
    char                reserved[44];           /* reserved for future use */
};
```

Field descriptions

`size` The total size in bytes of the profile.

`CMMType` The signature of the preferred CMM for color-matching and color-checking sessions for this profile. To avoid conflicts with other CMMs, this signature must be registered with the ICC. For the signature of the default CMM, see "Signature of ColorSync's Default Color Management Module" (page 397).

`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 seven profile classes supported by the ICC: input, display, output, named color space, device link, color

space conversion, or abstract. For the signatures representing profile classes, see “Profile Class” (page 396).

`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” (page 402).

`profileConnectionSpace` The profile connection space, or PCS. The signatures for the two profile connection spaces supported by ColorSync, `cmXYZData` and `cmLabData`, are described in “Color Space Signatures” (page 402).

`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. For information on the date and time format, see “Date and Time” (page 350).

`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*.

`flags` Flags that provide hints, such as preferred quality and speed options, to the preferred CMM. The `flags` field consists of an unsigned long data type. The 16 bits in the low word, 0-15, are reserved for use by the ICC. The 16 bits in the high word, 16-31, are available for use by color management systems. For information on how these bits are defined and how your application can set and test them, see “Flag Mask Definitions for Version 2.x Profiles” (page 414).

`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, paper, and ink types. The data type for this field is an array of two unsigned longs. The low word of `deviceAttributes[0]` is reserved by the ICC. The high word of `deviceAttributes[0]` and the entire word of

	`deviceAttributes[1]` are available for vendor use. For information on how the bits in `deviceAttributes` are defined and how your application can set and test them, see "Device Attribute Values for Version 2.x Profiles" (page 418).
`renderingIntent`	The preferred rendering intent for the object or file tagged with this profile. Four types of rendering intent are defined: perceptual, relative colorimetric, saturation, and absolute colorimetric. The `renderingIntent` field consists of an unsigned long data type. The low word is reserved by the ICC and is used to set the rendering intent. The high word is available for use. For information on how the bits in `renderingIntent` are defined and how your application can set and test them, see "Rendering Intent Values for Version 2.x Profiles" (page 419).
`white`	The profile illuminant white reference point, expressed in the XYZ color space.
`creator`	Signature identifying the profile creator.
`reserved`	This field is reserved for future use.

CMAppleProfileHeader

The ColorSync Manager defines the `CMAppleProfileHeader` structure to provide access to both version 2.x and version 1.0 profiles, as specified by the International Color Consortium. For related information, see "ColorSync and ICC Profile Format Version Numbers" (page 50). To obtain a copy of the *International Color Consortium Profile Format Specification*, or to get other information about the ICC, visit the ICC Web site at <http://www.color.org/>.

```
union CMAppleProfileHeader {
    CMHeader        cm1;    /* ColorSync version 1.0 profile header */
    CM2Header       cm2;    /* ColorSync version 2.x profile header */
};
```

Field descriptions

`cm1`	A version 1.0 profile header. For a description of the ColorSync version 1.0 profile header, see `CMHeader` (page 351).
`cm2`	A current profile header. For a description of the ColorSync profile header, see `CM2Header` (page 354).

Profile Reference

The ColorSync Manager defines the `CMProfileRef` type to provide access to a specific profile.

CMProfileRef

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` (page 222), `CMCopyProfile` (page 229), and `CMNewProfile` (page 227) functions. This allows multiple applications concurrent access to a profile. The ColorSync Manager defines an abstract private data structure of type `OpaqueCMProfileRef` for the profile reference.

```
typedef struct OpaqueCMProfileRef *CMProfileRef;
```

Profile Identifier

The ColorSync Manager defines the `CMProfileIdentifier` type to store a profile identifier.

CMProfileIdentifier

Embedding a profile in an image guarantees that the image can be rendered correctly on a different system. However, profiles can be large—as much as several hundred kilobytes. The ColorSync Manager defines a profile identifier structure, `CMProfileIdentifier`, that can identify a profile but that takes up much less space than a large profile.

The profile identifier structure contains a profile header, an optional calibration date, a profile description string length, and a variable-length profile description string. Your application might use an embedded profile identifier, for example, to change just the rendering intent or the flag values in an image without having to embed an entire copy of a profile. Rendering intent is described in “Rendering Intent Values for Version 2.x Profiles” (page 419) and flag values are described in “Flag Mask Definitions for Version 2.x Profiles” (page 414).

IMPORTANT

A document containing an embedded profile identifier cannot necessarily be ported to different systems or platforms. ▲

The ColorSync Manager provides the function routine `NCMUseProfileComment` (page 290) to embed profiles and profile identifiers in an open picture file. Your application can embed profile identifiers in place of entire profiles, or in addition to them. A profile identifier can refer to an embedded profile or to a profile on disk.

The ColorSync Manager provides two routines for finding a profile identifier:

- `CMProfileIdentifierListSearch` (page 316) for finding a profile identifier in a list of profile identifiers
- `CMProfileIdentifierFolderSearch` (page 315) for finding a profile identifier in the ColorSync Profiles folder.

The descriptions of those functions provide information on searching algorithms. See also `CMProfileSearchRef` (page 370) for additional information on profile searching.

```
struct CMProfileIdentifier {
    CM2Header           profileHeader;              /* version 2.x profile header */
    CMDateTime          calibrationDate;            /* optional; may be set to 0 */
```

```
    unsigned long       ASCIIProfileDescriptionLen; /* length of following array */
    char                ASCIIProfileDescription[1]; /* variable length */
};
typedef struct CMProfileIdentifier CMProfileIdentifier;
typedef CMProfileIdentifier *CMProfileIdentifierPtr;
```

Field descriptions

`profileHeader` A version 2.x profile header structure. For more information, see `CM2Header` (page 354). In determining a profile match, all header fields are considered, except for primary platform, flags, and rendering intent.

`calibrationDate` A structure of type `Date and Time` (page 350), which specifies year, month, day of month, hours, minutes, and seconds. This field is optional—when set to 0, it is not considered in determining a profile match. When nonzero, it is compared to the `'calt'` tag data.

`ASCIIProfileDescriptionLen`
The length of the ASCII description string that follows.

`ASCIIProfileDescription`
The ASCII profile description string, as specified by the profile description tag.

The `CMProfileIdentifierPtr` type definition defines a pointer to a profile identifier structure.

Profile Location

In most cases, a ColorSync version 2.x profile is stored in a disk file. (For information on profile version numbers see “ColorSync and ICC Profile Format Version Numbers.”) However, to support special requirements, a profile can also be located in memory or in an arbitrary location that is accessed by a procedure you specify. The ColorSync Manager provides the following data types for working with profile locations:

- `CMProfLoc` (page 361) is a union that can describe a file-, handle-, pointer-, or procedure-based profile location.
- `CMProfileLocation` (page 362) is a structure that combines a union of type `CMProfLoc` with a tag to identify the location type in the union.

- `CMFileLocation` (page 363) is a structure that specifies the location of a file-based profile.
- `CMHandleLocation` (page 363) is a structure that specifies the location of a handle-based profile.
- `CMPtrLocation` (page 364) is a structure that specifies the location of a pointer-based profile.
- `CMProcedureLocation` (page 364) is a structure that specifies the location of a procedure-based profile.

IMPORTANT

Starting with ColorSync version 2.5, you should use the `NCMGetProfileLocation` (page 233) function to obtain a profile location, rather than the `CMGetProfileLocation` (page 234) function. ▲

CMProfLoc

You use a union of type `CMProfLoc` to identify the location of a profile. You specify the union in the `u` field of the data type `CMProfileLocation` (page 362). Your application passes a pointer to a `CMProfileLocation` structure when it calls the `CMOpenProfile` (page 222) function to identify the location of a profile or the `CMNewProfile` (page 227), `CMCopyProfile` (page 229), or `CWNewLinkProfile` (page 267) functions to specify the location for a newly created profile.

You also pass a pointer to a `CMProfileLocation` structure to the `NCMGetProfileLocation` (page 233) and `CMGetProfileLocation` (page 234) functions to get the location of an existing profile. The `NCMGetProfileLocation` function is available starting with ColorSync version 2.5. It differs from its predecessor, `CMGetProfileLocation`, in that the newer version has a parameter for the size of the location structure for the specified profile.

```
union CMProfLoc {
    CMFileLocation          fileLoc;    /* specifies location on disk*/
    CMHandleLocation        handleLoc;  /* specifies location in relocatable memory */
    CMPtrLocation           ptrLoc;     /* specifies location in nonrelocatable
                                           memory */
    CMProcedureLocation     procLoc;    /* specifies access procedure */
};
```

Field descriptions

`fileLoc` A data structure containing a file system specification record specifying the location of a profile disk file.

`handleLoc` A data structure containing a handle that indicates the location of a profile in relocatable memory.

`ptrLoc` A data structure containing a pointer that points to a profile in nonrelocatable memory.

`procLoc` A data structure containing a universal procedure pointer that points to a profile access procedure supplied by you. The ColorSync Manager calls your procedure when the profile is created, initialized, opened, read, updated, or closed.

CMProfileLocation

Your application passes a profile location structure of type `CMProfileLocation` when it calls:

- the function `CMOpenProfile` (page 222), specifying the location of a profile to open
- the `CMNewProfile` (page 227), `CWNewLinkProfile` (page 267), or `CMCopyProfile` (page 229) functions, specifying the location of a profile to create or duplicate

```
struct CMProfileLocation {
    short        locType;   /* location type for profile */
    CMProfLoc    u;         /* location information for profile */
};
```

Field descriptions

`locType` The type of data structure that the `u` field's `CMProfLoc` union holds—a file specification, a handle, a pointer, or a universal procedure pointer. To specify the type, you use the constants defined in the enumeration described in "Profile Location Type" (page 393).

`u` A union of type `CMProfLoc` (page 361) identifying the profile location.

CMFileLocation

Your application uses the `CMFileLocation` structure to provide a file specification for a profile stored in a disk file. You provide a file specification structure in the `CMProfileLocation` structure's `u` field to specify the location of an existing profile or a profile to be created.

```
struct CMFileLocation {
    FSSpec   spec;  /* specifies profile file location on disk */
};
```

Field descriptions

`spec` A file system specification structure giving 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.*

CMHandleLocation

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 `CMProfileLocation` structure's `u` field to specify an existing profile or a profile to be created.

```
struct CMHandleLocation {
    Handle   h; /* handle that specifies profile's location in memory */
};
```

Field descriptions

`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.*

CMPtrLocation

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 `CMProfileLocation` structure's `u` field to point to an existing profile.

```
struct CMPtrLocation {
    Ptr  p; /* pointer that specifies profile's location in memory */
};
```

Field descriptions

`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*.

CMProcedureLocation

Your application uses the `CMProcedureLocation` structure to provide a universal procedure pointer to a profile access procedure. You provide this structure in the `CMProfileLocation` structure's `u` field. The `CMProcedureLocation` structure also contains a pointer field to specify data associated with the profile access procedure.

The ColorSync Manager calls your profile access procedure when the profile is created, initialized, opened, read, updated, or closed.

```
struct CMProcedureLocation {
    CMProfileAccessUPP      proc;   /* profile access function universal
                                       procedure pointer */
    void *                  refCon; /* pointer to access procedure's
                                       private data, if any */
};
```

Field descriptions

`proc` A universal procedure pointer to a profile access procedure. For a description of the procedure, see the function `MyCMProfileAccessProc` (page 348).

`refCon`	A pointer to the profile access procedure's private data, such as a file or resource name, a pointer to a current offset, and so on.

The ColorSync Manager defines the `CMProfileAccessUPP` type as follows:

```
typedef UniversalProcPtr CMProfileAccessUPP;
```

Cached Profile Searching

The function `CMIterateColorSyncFolder` (page 304) takes advantage of the profile cache available starting with ColorSync version 2.5 to provide optimized searching and quick access to profile information. The function iterates through the available profiles, calling a function you supply to process each profile you are interested in. For more information, see"Searching for Specific Profiles Prior to ColorSync 2.5" (page 136) and "Searching for Profiles With ColorSync 2.5" (page 303).

The ColorSync Manager defines the following types for profile searching with the `CMIterateColorSyncFolder` function:

- `CMProfileIterateProcPtr` (page 365) defines a callback routine you implement to process the profiles found during a search; **new in ColorSync 2.5**.
- `CMProfileIterateData` (page 366) stores information about a specific profile; passed to your callback routine during a search; **new in ColorSync 2.5**.

CMProfileIterateProcPtr

NEW IN COLORSYNC 2.5

The function `CMIterateColorSyncFolder` (page 304) has a parameter of type `CMProfileIterateUPP`. ColorSync defines the procedure pointer

CMProfileIterateProcPtr to use for this parameter. For a description of the application-defined function itself, see MyCMProfileFilterProc (page 347).

CMProfileIterateProcPtr is defined as follows:

```
pascal OSErr (*CMProfileIterateProcPtr )
                    (CMProfileIterateData *iterateData,
                     void *refCon);
```

iterateData — A pointer to a structure of type CMProfileIterateData (page 366). When the CMIterateColorSyncFolder (page 304) function calls your application-defined function, as it does once for each found profile, the structure contains key information about the profile.

refCon — An untyped pointer to arbitrary data your application previously passed to the function CMIterateColorSyncFolder (page 304).

callback return value — A result code of type CMError. If your callback function returns an error, CMIterateColorSyncFolder stops iterating and returns the error value to its caller (presumably your code). For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

When you call CMIterateColorSyncFolder (page 304), you pass a universal procedure pointer of type CMProfileIterateProcPtr (page 365) that points to a callback function you provide. For more information on the callback function, see MyProfileIterateProc (page 340).

CMProfileIterateData

NEW IN COLORSYNC 2.5

The ColorSync Manager defines the CMProfileIterateData structure to provide your CMProfileIterateProcPtr (page 365) callback routine with a description of a profile during an iteration through the available profiles that takes place when you call CMIterateColorSyncFolder (page 304).

```
struct CMProfileIterateData {
    unsigned long   dataVersion;    /* cmProfileIterateDataVersion1 */
    CM2Header       header;
    ScriptCode      code;
    Str255          name;
    CMProfileLocation location;
};
typedef struct CMProfileIterateData CMProfileIterateData;
```

`dataVersion` A value identifying the version of the structure. Currently set to `cmProfileIterateDataVersion1`.

`header` A ColorSync version 2.x profile header structure of type `CM2Header` (page 354), containing information such as the profile size, type, version, and so on.

`code` A script code identifying the script system used for the profile description. You can learn more about script codes in *Inside Macintosh: Text*. The `ScriptCode` data type is defined in the MacTypes.h header file.

`name` The profile name, stored as a Pascal-type string (with length byte first) of up to 255 characters.

`location` A structure specifying the profile location. With ColorSync 2.5, the location is always file-based, but that may not be true for future versions. Your code should always verify that the location structure contains a file specification before attempting to use it.

Non-Cached Profile Searching

The ColorSync Manager defines the following types your application to use in searching for profiles.

IMPORTANT

These types do not take advantage of the profile cache added in ColorSync version 2.5. They are used with the searching described in “Searching for Profiles Prior to ColorSync 2.5” (page 306). ▲

- `CMSearchRecord` (page 368) provides the ColorSync Manager with search criteria to use in determining which version 2.x profiles to include in a search result list and which to filter out.
- `CMProfileSearchRef` (page 370) stores a search result consisting of a list of profiles matching certain search criteria.

CMSearchRecord

NOT RECOMMENDED IN COLORSYNC 2.5

Your application supplies a search record of type `CMSearchRecord` as the `searchSpec` parameter to the function `CMNewProfileSearch` (page 308). The search record structure provides the ColorSync Manager with search criteria to use in determining which version 2.x profiles to include in the result list and which to filter out.

Most of the fields in the `CMSearchRecord` structure are identical to corresponding fields in the `CM2Header` structure for version 2.x profiles. When you set a bit in the `searchMask` field of the `CMSearchRecord` structure, you cause the search criteria to include the data specified by that bit. For example, if you set the `cmMatchProfileCMMType` bit, the search result will not include a profile unless the data in the profile header's `CMMType` field matches the data you specify in the `CMSearchRecord` structure's `CMMType` field.

IMPORTANT

If you specify a bit in the `searchMask` field, you must supply information in the `CMSearchRecord` field that corresponds to that bit. ▲

The ColorSync Manager preserves the search criteria 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.

IMPORTANT

You cannot use the ColorSync Manager search functions to search for ColorSync 1.0 profiles. ▲

A search record is defined by the `CMSearchRecord` type definition.

```
struct CMSearchRecord {
    OSType                      CMMType;                /* CMM signature */
    OSType                      profileClass;           /* profile signature */
    OSType                      dataColorSpace;         /* data color space */
    OSType                      profileConnectionSpace; /* profile connection
                                                           color space */
    unsigned long               deviceManufacturer;     /* device manufacturer */
    unsigned long               deviceModel;            /* device model */
    unsigned long               deviceAttributes[2];    /* specifies attributes such as
                                                           paper or ink type */
    unsigned long               profileFlags;           /* hints to CMM */
    unsigned long               searchMask;             /* bitmap specifying search
                                                           mask fields to use */
    CMProfileFilterUPP          filter;                 /* pointer to function that
                                                           determines whether to
                                                           exclude profile */
};
```

Field descriptions

`CMMType` The signature of a CMM. The signature of the default CMM is specified by the `kDefaultCMMSignature` constant.

`profileClass` The class signature identifying the type of profile to search for. For a list of profile class signatures, see “Profile Class” (page 396).

`dataColorSpace` A data color space. For a list of the color space signatures, see “Color Space Signatures” (page 402).

`profileConnectionSpace` A profile connection color space. The signatures for the two profile connection spaces supported by ColorSync, `cmXYZData` and `cmLabData`, are described in “Color Space Signatures” (page 402).

`deviceManufacturer` The signature of the manufacturer.

`deviceModel` The model of a device.

`deviceAttributes` Attributes for a particular device setup, such as media, paper, and ink types.

`profileFlags`	Flags that indicate hints for the preferred CMM, such as quality, speed, and memory options. 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 use in the profile search. Here are the defined bitmask values:
	`cmMatchAnyProfile0x00000000`
	`cmMatchProfileCMMType 0x00000001`
	`cmMatchProfileClass 0x00000002`
	`cmMatchDataColorSpace 0x00000004`
	`cmMatchProfileConnectionSpace 0x00000008`
	`cmMatchManufacturer 0x00000010`
	`cmMatchModel 0x00000020`
	`cmMatchAttributes 0x00000040`
	`cmMatchProfileFlags 0x00000080`

`filter`	A pointer to an application-supplied function that determines whether to exclude a profile from the profile search result list. For more information, see the function `MyCMProfileFilterProc` (page 347).

VERSION NOTES

This type does not take advantage of the profile cache added in ColorSync version 2.5. It is used with the searching described in "Searching for Profiles Prior to ColorSync 2.5" (page 306). See "Cached Profile Searching" (page 365) for information on data structures used with searching in version 2.5.

CMProfileSearchRef

NOT RECOMMENDED IN COLORSYNC 2.5

A search result consists of a list of profiles matching certain search criteria. When your application calls the function `CMNewProfileSearch` (page 308) 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. Your application passes the search result reference to these ColorSync functions:

- `CMUpdateProfileSearch` (page 310) updates a search result list.

- `CMDisposeProfileSearch` (page 311) disposes of a search result list.
- `CMSearchGetIndProfile` (page 312) opens a reference to a profile at a specific position in a search result list.
- `CMSearchGetIndProfileFileSpec` (page 313) obtains the file specification for a profile in a search result list.

The ColorSync Manager uses an abstract private data structure of type `OpaqueCMProfileSearchRef` in defining the search result reference.

```
struct OpaqueCMProfileSearchRef *CMProfileSearchRef;
```

VERSION NOTES

This type does not take advantage of the profile cache added in ColorSync version 2.5. It is used with the searching described in "Searching for Profiles Prior to ColorSync 2.5" (page 306). See "Cached Profile Searching" (page 365) for information on data structures used with searching in version 2.5.

Color Values

The ColorSync Manager defines the following data types for storing standard color values:

- `CMXYZComponent` (page 372)
- `CMXYZColor` (page 372)
- `CMFixedXYZColor` (page 373)
- `CMLabColor` (page 373)
- `CMLuvColor` (page 374)
- `CMYxyColor` (page 374)
- `CMRGBColor` (page 374)
- `CMHLSColor` (page 375)
- `CMHSVColor` (page 375)
- `CMCMYKColor` (page 376)
- `CMCMYColor` (page 376)

- CMGrayColor (page 376)
- CMNamedColor (page 377)
- HiFi Color Values (page 377)
- CMColor (page 378)

CMXYZComponent

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;  /* expresses value in XYZ color space; 1 bit
                                           for integer part, 15 bits for fraction */
```

CMXYZColor

Three color component values defined by the 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 use in general purpose 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;  /* X component of color in XYZ color space */
    CMXYZComponent  Y;  /* Y component of color in XYZ color space */
    CMXYZComponent  Z;  /* Z component of color in XYZ color space */
};
```

CMFixedXYZColor

ColorSync uses the `CMFixedXYZColor` data type to specify the profile illuminant in the profile header's `white` field and to specify other profile element values. 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.

Your application can 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). To convert color values, you use the functions `CMConvertFixedXYZToXYZ` (page 326) and `CMConvertXYZToFixedXYZ` (page 325).

```
struct CMFixedXYZColor {
    Fixed   X;  /* Fixed X component of color in XYZ color space */
    Fixed   Y;  /* Fixed Y component of color in XYZ color space */
    Fixed   Z;  /* Fixed Z component of color in XYZ color space */
};
```

CMLabColor

A color expressed in the L*a*b* color space is composed of `L`, `a`, and `b` component values. The `L` color component is expressed as a numeric value within the range of 0 to 65535, which maps to 0 to 100 inclusive. Note that this encoding is slightly different from the 0 to 65280 encoding of the L channel defined in the ICC specification for PCS L*a*b values. The `a` and b components range from 0 to 65535, which maps to –128 to 127.996 inclusive. The color value is defined by the `CMLabColor` type definition.

```
struct CMLabColor {
    unsigned short  L;  /* L component of color in Lab color space */
    unsigned short  a;  /* a component of color in Lab color space */
    unsigned short  b;  /* b component of color in Lab color space */
};
```

CMLuvColor

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` and `v` components, this maps to –128 to 127.996 inclusive. The color value is defined by the `CMLuvColor` type definition.

```
struct CMLuvColor {
    unsigned short  L;  /* L component of color in Luv color space */
    unsigned short  u;  /* u component of color in Luv color space */
    unsigned short  v;  /* v component of color in Luv color space */
};
```

CMYxyColor

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 */
};
```

CMRGBColor

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;
};
```

CMHLSColor

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;
};
```

CMHSVColor

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.

```
struct CMHSVColor {
    unsigned short  hue;
    unsigned short  saturation;
    unsigned short  value;
};
```

CMCMYKColor

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;
};
```

CMCMYColor

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;
};
```

CMGrayColor

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;
};
```

CMNamedColor

A color value expressed in a named color space is composed of a single component, `namedColorIndex`, represented as a numeric value within the range of an unsigned long, or 1 to $2^{32} - 1$ inclusive.

```
struct CMNamedColor {
    unsigned long namedColorIndex;  /* 1..a lot */
};
```

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];
};
```

CMColor

Your application can use a union of type `CMColor` to specify a color value defined by one of the 15 data types supported by the union. Your application uses an array of color unions to specify a list of colors to match, check, or convert. The array is passed as a parameter to the general purpose color matching, color checking, or color conversion functions. The following functions use a color union:

- The function `CWMatchColors` (page 281) matches the colors in the color list array to the data color space of the destination profile specified by the color world.
- The function `CWCheckColors` (page 283) 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 in "Converting Between Color Spaces" (page 318), take source and destination array parameters of type `CMColor` specifying lists of colors to convert from one color space to another.

IMPORTANT

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. Each of the color types included in the union is defined previously.

```
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;
    CMNamedColor            namedColor;
};
```

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` (page 374).

`hsv` A color value expressed in the HSV color space as data of type `CMHSVColor` (page 375).

`hls` A color value expressed in the HLS color space as data of type `CMHLSColor` (page 375).

`XYZ` A color value expressed in the XYZ color space as data of type `CMXYZColor` (page 372).

`Lab` A color value expressed in the L*a*b* color space as data of type `CMLabColor` (page 373).

`Luv` A color value expressed in the L*u*v* color space as data of type `CMLuvColor` (page 374).

`Yxy` A color value expressed in the Yxy color space as data of type `CMYxyColor` (page 374).

`cmyk` A color value expressed in the CMYK color space as data of type `CMCMYKColor` (page 376).

`cmy` A color value expressed in the CMY color space as data of type `CMCMYColor` (page 376).

`gray` A color value expressed in the Gray color space as data of type `CMGrayColor` (page 376).

`mc5` A color value expressed in the five-channel multichannel color space as data of type `CMMultichannel5Color`. See “HiFi Color Values” (page 377) 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” (page 377) 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" (page 377) 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" (page 377) for a description of the `CMMultichannel8Color` data type.
`namedColor`	A color value expressed as an index into a named color space. See `CMNamedColor` (page 377) for a description of the `CMNamedColor` data type.

Bitmap Information

The ColorSync Manager defines the `CMBitmap` type to describe color bitmap images.

CMBitmap

The ColorSync Manager defines a bitmap structure of type `CMBitmap` to describe color bitmap images. When your application calls the function `CWMatchBitmap` (page 276), you pass a pointer to a source bitmap of type `CMBitmap` containing the image whose colors are to be matched to the color gamut of the device 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 function `CWCheckBitMap` (page 279), it passes a pointer to a 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 in the same way QuickDraw GX does. ColorSync supports indexed bitmaps only when the `cmNamedIndexed32Space` color space constant is used in conjunction with a named color space profile. ▲

```
struct CMBitmap {
    char                *image;     /* a bit image */
    long                width;      /* pixel width of a row in the image */
    long                height;     /* number of rows in the image */
    long                rowBytes;   /* offset in bytes from 1 row to the next */
    long                pixelSize;  /* number of bits per pixel */
    CMBitmapColorSpace  space;      /* color space for colors of bitmap image */
    long                user1;      /* not used by ColorSync*/
    long                user2;      /* not used by ColorSync */
};
```

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. The pixel size should correspond to the packing size specified in the `space` field. This requirement is not enforced as of ColorSync version 2.5, but it may be enforced in future versions.
`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 Space Constants With Packing Formats” (page 409).
`user1`	Not used by ColorSync. It is recommended that you set this field to 0.
`user2`	Not used by ColorSync. It is recommended that you set this field to 0.

Color Matching Reference

The ColorSync Manager defines the `CMMatchRef` type to refer to a color-matching session.

CMMatchRef

The ColorSync Manager defines an abstract private data structure of type `OpaqueCMMatchRef` for the color-matching-session reference. When your application calls the function `NCMBeginMatching` (page 285) to begin a QuickDraw-specific color-matching session, the ColorSync Manager returns a reference pointer to the color-matching session which you must later pass to the `CMEndMatching` function to conclude the session.

```
struct OpaqueCMMatchRef *CMMatchRef;
```

Color Worlds

The ColorSync Manager defines the following types for working with color worlds:

- `CMCWInfoRecord` (page 382) stores information about a specific color world.
- `CMWorldRef` (page 383) identifies a color world for functions that perform color-matching and color-checking sessions and dispose of the color world.
- `CMConcatProfileSet` (page 384) establishes a color world with a sequential relationship among several profiles.
- `CMMInfoRecord` (page 385) stores information about one or two CMMs used in a given color world.

CMCWInfoRecord

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;   /* number of CMMs in the session; 1 or 2 */
    CMMInfoRecord       cmmInfo[2]; /* records describing CMM type and version */
};
```

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` (page 385) reporting the CMM type and version number.

If `cmmCount` is 1, the first element of the array (`cmmInfo[0]`) describes the CMM and the contents of the second element of the array (`cmmInfo[1]`) is undefined.

If `cmmCount` is 2, the first element of the array (`cmmInfo[0]`) describes the source CMM and the second element of the array (`cmmInfo[1]`) describes the destination CMM.

CMWorldRef

Your application passes a color world reference as a parameter on calls to functions to perform color-matching and color-checking sessions and to dispose of the color world. When your application calls the function `NCWNewColorWorld` (page 262) and the function `CWConcatColorWorld` (page 265) 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 `OpaqueCMWorldRef` for the color world reference.

```
struct OpaqueCMWorldRef *CMWorldRef;
```

The color world is affected by the rendering intent, lookup flag, gamut flag, and quality flag of the profiles that make up the color world. For more information, see “Rendering Intent Values for Version 2.x Profiles” (page 419), “Flag Mask Definitions for Version 2.x Profiles” (page 414), and “Quality Flag Values for Version 2.x Profiles” (page 417).

CMConcatProfileSet

You can call the function `NCWNewColorWorld` (page 262) to create a color world for operations such as color matching and color conversion. A color world is normally based on two profiles—source and destination. But it can include a series of profiles that describe the processing for a work-flow sequence, such as scanning, printing, and previewing an image. To create a color world that includes a series of profiles, you use the function `CWConcatColorWorld` (page 265).

You use an array to hold the set of profile references used in your operations. You provide this array in 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 link 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 link profile accommodates users who use a specific configuration requiring a combination of device profiles and possibly non-device profiles repeatedly over time.

To set up a color world that includes a concatenated set of profiles, your application uses the function `CWConcatColorWorld` (page 265), passing it a structure of type `CMConcatProfileSet`. The array you pass may contain a set of profile references or it may contain only the profile reference of a device link profile. To create a device link profile, your application calls the function `CWNewLinkProfile` (page 267), passing a structure of type `CMConcatProfileSet`.

```
struct CMConcatProfileSet {
    unsigned short      keyIndex;       /* 0-based index into array of profiles,
                                           specifying profile to use CMM for */
    unsigned short      count;          /* 1-based count of profiles in array;
                                           minimum is one profile */
    CMProfileRef        profileSet[1];  /* array of profile references */
};
```

Field descriptions

`keyIndex` A zero-based index into the array of profile references identifying the profile whose CMM is used for the entire session. The profile's `CMMType` field identifies the CMM.

`count` The one-based count of profiles in the profile array. A minimum of one profile is required.

`profileSet` A variable-length array of profile references. The references 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 function `CWConcatColorWorld` (page 265) or for the function `CWNewLinkProfile` (page 267). See the function descriptions for details.

CMMInfoRecord

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;    /* CMM signature */
    long        CMMVersion; /* CMM version */
};
```

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.

Video Card Gamma

Starting with version 2.5, ColorSync supports an optional profile tag for video card gamma. The tag specifies gamma information, stored either as a formula or in table format, to be loaded into the video card when the profile containing the tag is put into use.

The ColorSync Manager defines the following data types for working with the video card gamma profile tag

- `CMVideoCardGammaType` (page 386) identifies a video card gamma profile tag; **new in ColorSync 2.5**.
- `CMVideoCardGammaTable` (page 387) stores video card gamma data in table format; **new in ColorSync 2.5**.
- `CMVideoCardGammaFormula` (page 388) stores video card gamma data in formula format; **new in ColorSync 2.5**.
- `CMVideoCardGamma` (page 389) specifies the video gamma data, in either table or formula format, to store with a video gamma profile tag; **new in ColorSync 2.5**.

CMVideoCardGammaType

NEW IN COLORSYNC 2.5

The ColorSync Manager defines the `CMVideoCardGammaType` data structure to specify a video card gamma profile tag.

```
struct CMVideoCardGammaType
{
    OSType              typeDescriptor;
    unsigned long       reserved;
    CMVideoCardGamma    gamma;
};
typedef struct CMVideoCardGammaType CMVideoCardGammaType;
```

Field descriptions

`typeDescriptor` The signature type for a video card gamma tag. There is currently only one type possible, `cmSigVideoCardGammaType`.

`reserved` This field is reserved and must contain the value 0.

`gamma`	A structure that specifies the video card gamma data for the profile tag, as described in "CMVideoCardGamma" (page 389).

CMVideoCardGammaTable

NEW IN COLORSYNC 2.5

The ColorSync Manager defines the `CMVideoCardGammaTable` data structure to specify video card gamma data in table format. You specify the number of channels, the number of entries per channel, and the size of each entry. The last field in the structure is an array of size one that serves as the start of the table data. The actual size of the array is equal to the number of channels times the number of entries times the size of each entry.

```
struct CMVideoCardGammaTable
{
    unsigned short      channels;
    unsigned short      entryCount;
    unsigned short      entrySize;
    char                data[1];
};
typedef struct CMVideoCardGammaTable CMVideoCardGammaTable;
```

Field descriptions

`channels`	Number of gamma channels (1 or 3). If `channels` is set to 1 then the red, green, and blue lookup tables (LUTs) of the video card will be loaded with the same data. If `channels` is set to 3, then if the video card supports separate red, green, and blue LUTs, then the video card LUTs will be loaded with the data for the three channels from the `data` array.
`entryCount`	Number of entries per channel (1-based). The number of entries must be greater than or equal to 2.
`entrySize`	Size in bytes of each entry.
`data`	Variable-sized array of data. The size of the data is equal to `channels * entryCount * entrySize`.

CMVideoCardGammaFormula

NEW IN COLORSYNC 2.5

The ColorSync Manager defines the `CMVideoCardGammaFormula` data structure to specify video card gamma data by providing three values each for red, blue and green gamma. The values represent the actual gamma, the minimum gamma, and the maximum gamma for each color. Specifying video gamma information by formula takes less space than specifying it with a table, but the results may be less precise.

```
struct CMVideoCardGammaFormula {
    Fixed           redGamma;
    Fixed           redMin;
    Fixed           redMax;
    Fixed           greenGamma;
    Fixed           greenMin;
    Fixed           greenMax;
    Fixed           blueGamma;
    Fixed           blueMin;
    Fixed           blueMax;
};
```

Field descriptions

`redGamma` The gamma value for red. It must be greater than 0.0.

`redMin` The minimum gamma value for red. It must be greater than 0.0 and less than 1.0.

`redMax` The maximum gamma value for red. It must be greater than 0.0 and less than 1.0.

`greenGamma` The gamma value for green. It must be greater than 0.0.

`greenMin` The minimum gamma value for green. It must be greater than 0.0 and less than 1.0.

`greenMax` The maximum gamma value for green. It must be greater than 0.0 and less than 1.0.

`blueGamma` The gamma value for blue. It must be greater than 0.0.

`blueMin` The minimum gamma value for blue. It must be greater than 0.0 and less than 1.0.

`blueMax` The maximum gamma value for blue. It must be greater than 0.0 and less than 1.0.

CMVideoCardGamma

NEW IN COLORSYNC 2.5

The ColorSync Manager defines the `CMVideoCardGamma` data structure to specify the video gamma data to store with a video gamma profile tag. The structure is a union that can store data in either table or formula format.

```
struct CMVideoCardGamma
{
    unsigned long                   tagType;
    union
    {
        CMVideoCardGammaTable       table;
        CMVideoCardGammaFormula     formula;
    }                               u;
};
typedef struct CMVideoCardGamma CMVideoCardGamma;
```

Field descriptions

`tagType` A "Video Card Gamma Storage Type" (page 422) constant that specifies the format of the data currently stored in the union. To determine the type of structure present in a specific instance of the `CMVideoCardGamma` structure, you test this union tag. If you are setting up a `CMVideoCardGamma` structure to store video card gamma data, you set `tagType` to a constant value that identifies the structure type you are using. The possible constant values are described in "Video Card Gamma Storage Type" (page 422).

`table` A structure of type `CMVideoCardGammaTable`. If the `tagType` field has the value `cmVideoCardGammaTableType`, the `CMVideoCardGamma` structure's union field should be treated as a table, as described in "CMVideoCardGammaTable" (page 387).

`formula` A structure of type `CMVideoCardGammaFormula`. If the `tagType` field has the value `cmVideoCardGammaFormulaType`, the `CMVideoCardGamma` structure's union field represents a formula, as described in "CMVideoCardGammaFormula" (page 388).

Color Matching While Printing

The ColorSync Manager defines the `TEnableColorMatchingBlk` type for use with the Toolbox `PrGeneral` function to enable or disable color matching while printing.

TEnableColorMatchingBlk

You pass a structure defined by the `TEnableColorMatchingBlk` data type to the `PrGeneral` function when you use the `EnableColorMatchingOp` opcode, described in "PrGeneral Function Operation Codes" (page 423). 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.
`filler`	Filler.

Color Rendering Dictionary Virtual Memory Size

The ColorSync Manager defines the `CMIntentCRDVMSize` type for specifying the maximum virtual memory size of a color rendering dictionary.

CMIntentCRDVMSize

To specify the maximum virtual memory (VM) 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 optional Apple-defined `'psvm'` tag. The PostScript CRD virtual memory size tag structure'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 default CMM uses the tag and returns the values specified by the tag when your application or device driver calls the function `CMGetPS2ColorRenderingVMSize` (page 338).

If a PostScript printer profile does not include this tag, the CMM uses an algorithm to determine the VM size of the CRD. This may result in a size that is greater than the actual 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;    /* rendering intent value */
    unsigned long   VMSize;             /* virtual memory size of CRD */
};
```

Field descriptions

`renderingIntent` The rendering intent whose CRD virtual memory size you want to obtain. The following rendering intent values are described in "Rendering Intent Values for Version 2.x Profiles" (page 419):

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.

CMPS2CRDVMSizeType

The `CMPS2CRDVMSizeType` data type defines the Apple-defined `'psvm'` optional tag.

```
struct CMPS2CRDVMSizeType {
    OSType              typeDescriptor; /* PostScript VM signature */
    unsigned long       reserved;       /* reserved */
    unsigned long       count;          /* entries in CRD array */
    CMIntentCRDVMSize   intentCRD[1];   /* variable-sized array */
};
```

Field 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 four entries: 0, 1, 2, and 3.
`intentCRD`	A variable-sized array of four or more members defined by the `CMIntentCRDSize` data type.

Constants for the ColorSync Manager

This section describes the constants defined by the ColorSync Manager for your application's use. The constants are organized into the following categories:

- "ColorSync Flag Constants" (page 413)
- "Video Card Gamma Constants" (page 421)
- "PrGeneral Function Operation Codes" (page 423)
- "Element Tags and Signatures for Version 1.0 Profiles" (page 424)

Profile Location Type

Your application specifies the location for a profile using a profile location structure of type `CMProfileLocation` (page 362). A ColorSync profile that you open or create is typically stored in one of the following locations:

- In a disk file. The `u` field (a union) of the profile location data structure contains a file specification for a profile that is disk-file based. This is the most common way to store a ColorSync profile.
- In relocatable memory. The `u` field of the profile location data structure contains a handle specification for a profile that is stored in a handle.
- In nonrelocatable memory. The `u` field of the profile location data structure contains a pointer specification for a profile that is pointer based.
- In an arbitrary location, accessed by a procedure you provide. The `u` field of the profile location data structure contains a universal procedure pointer to your access procedure, as well as a pointer that may point to data associated with your procedure.

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 is not saved after the session. For this case, the ColorSync Manager allows you to specify the profile location as having no specific location.

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

- the `CMOpenProfile` function to obtain a reference to a profile
- the `CMNewProfile`, `CWNewLinkProfile`, or `CMCopyProfile` functions to create a new profile
- the `CMGetProfileLocation` function to get the location of an existing profile

Your application identifies the type of data the `CMProfileLocation` `u` field holds—a file specification, a handle, and so on—in the `CMProfileLocation`

structure's `locType` field. You use the constants defined by the following enumeration to identify the location type.

```
enum {
    cmNoProfileBase         = 0, /* profile is temporary */
    cmFileBasedProfile      = 1, /* profile is disk-based */
    cmHandleBasedProfile    = 2, /* profile in relocatable memory */
    cmPtrBasedProfile       = 3, /* profile in nonrelocatable memory */
    cmProcedureBasedProfile = 4  /* profile is accessed by procedure */
};
```

Enumerator 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 stored in a disk-file and the `CMProfLoc` union of type `CMProfLoc` (page 361) holds a structure of type `CMFileLocation` (page 363) identifying the profile file. You can specify this type of profile location with the `CMOpenProfile`, `CMNewProfile`, `CMCopyProfile`, and `CMNewLinkProfile` functions.

`cmHandleBasedProfile` The profile is stored in relocatable memory and the `CMProfLoc` union of type `CMProfLoc` (page 361) holds a handle to the profile in a structure of type `CMHandleLocation` (page 363). You can specify this type of profile location with the `CMOpenProfile`, `CMNewProfile`, and `CMCopyProfile` functions.

`cmPtrBasedProfile` The profile is stored in nonrelocatable memory and the `CMProfLoc` union of type `CMProfLoc` (page 361) holds a pointer to the profile in a structure of type `CMPtrLocation` (page 364). You can specify this type of profile location with the `CMOpenProfile` function only.

`cmProcedureBasedProfile` The profile is in an arbitrary location, accessed through a procedure supplied by you. The `CMProfLoc` union of type `CMProfLoc` (page 361) holds a universal procedure pointer to

your profile access procedure in a structure of type `CMProcedureLocation` (page 364). You can specify this type of profile location with the `CMOpenProfile`, `CMNewProfile`, `CMCopyProfile`, and `CMNewLinkProfile` functions. For a description of an application-supplied profile access procedure, see "MyCMProfileAccessProc" (page 348). For sample code demonstrating procedure-based profile access, see "Accessing a Resource-Based Profile With a Procedure" (page 149).

Profile Access Procedure Operation Codes

When your application calls the `CMOpenProfile`, `CMNewProfile`, `CMCopyProfile`, or `CMNewLinkProfile` functions, it can supply the ColorSync Manager with a profile location structure of type `CMProcedureLocation` (page 364) to specify a procedure that provides access to a profile. The ColorSync Manager calls your procedure when the profile is created, initialized, opened, read, updated, or closed. The profile access procedure declaration is described in "MyCMProfileAccessProc" (page 348). For sample code demonstrating procedure-based profile access, see "Accessing a Resource-Based Profile With a Procedure" (page 149).

When the ColorSync Manager calls your profile access procedure, it passes one of the following constants in the `command` parameter to specify an operation. Your procedure must be able to respond to each of these constants.

```
enum {
    cmOpenReadAccess    = 1,    /* open profile for reading */
    cmOpenWriteAccess   = 2,    /* open profile for writing */
    cmReadAccess        = 3,    /* read specified bytes from profile */
    cmWriteAccess       = 4,    /* write specifies bytes to profile */
    cmCloseAccess       = 5,    /* close profile for read or write */
    cmCreateNewAccess   = 6,    /* create new data stream for profile */
    cmAbortWriteAccess  = 7,    /* cancel current write process */
    cmBeginAccess       = 8,    /* begin procedure access */
    cmEndAccess         = 9     /* end procedure access */
};
```

Enumerator descriptions

`cmOpenReadAccess` Open the profile for reading.

`cmOpenWriteAccess`	Open the profile for writing. The total size of the profile is specified in the `size` parameter.
`cmReadAccess`	Read the number of bytes specified by the `size` parameter.
`cmWriteAccess`	Write the number of bytes specified by the `size` parameter.
`cmCloseAccess`	Close the profile for reading or writing.
`cmCreateNewAccess`	Create a new data stream for the profile.
`cmAbortWriteAccess`	Cancel the current write attempt.
`cmBeginAccess`	Begin the process of procedural access. This is always the first operation constant passed to the access procedure. If the call is successful, the `cmEndAccess` operation is guaranteed to be the last call to the procedure.
`cmEndAccess`	End the process of procedural access. This is always the last operation constant passed to the access procedure (unless the `cmBeginAccess` call failed).

Profile Class

The ColorSync Manager supports seven classes, or types, of profiles, as described in “Profile Classes” (page 51).

A profile creator specifies the profile class in the profile header’s `profileClass` field. For a description of the profile header, see “CM2Header” (page 354). The following enumeration defines the profile class signatures:

```
enum {
    cmInputClass            = 'scnr',   /* input device profile */
    cmDisplayClass          = 'mntr',   /* display device profile */
    cmOutputClass           = 'prtr',   /* output device profile */
    cmLinkClass             = 'link',   /* device link profile */
    cmAbstractClass         = 'abst',   /* abstract profile */
    cmColorSpaceClass       = 'spac'    /* color space profile */
    cmNamedColorClass       = 'nmcl'    /* named color profile */
};
```

Enumerator descriptions

`cmInputClass`	An input device profile defined for a scanner.
`cmDisplayClass`	A display device profile defined for a monitor.
`cmOutputClass`	An output device profile defined for a printer.

`cmLinkClass`	A device link profile.
`cmAbstractClass`	An abstract profile.
`cmColorSpaceClass`	A color space profile.
`cmNamedColorClass`	A named color space profile.

Signature of ColorSync's Default Color Management Module

A color management module (CMM) uses profiles to convert and match a color in a given color space on a given device to or from another color space or device. For more information on CMMs, including a description of the default CMM supplied with ColorSync, see "Color Management Modules" (page 58).

To specify the default CMM, set the `CMMType` field of the profile header to the default signature defined by the following enumeration. You use a structure of type `CM2Header` (page 354) for a ColorSync 2.x profile and a structure of type `CMHeader` (page 351) for a 1.0 profile header.

```
enum {
    kDefaultCMMSignature = 'appl'
};
```

Enumerator descriptions

`kDefaultCMMSignature`
Signature for the default CMM supplied with the ColorSync Manager.

Commands for Caller-Supplied ColorSync Data Transfer Functions

When your application calls the function `CMFlattenProfile` (page 237), the function `CMUnflattenProfile` (page 239), or the PostScript-related functions of type `Color-Matching With PostScript Devices` (page 332), the selected CMM—or, for the `CMUnflattenProfile` function, the ColorSync Manager—calls the flatten function you supply to transform profile data. The call passes one of the command constants defined by this enumeration.

Your application provides a pointer to your ColorSync data transfer function as a parameter to the functions. The ColorSync Manager or the CMM calls your data transfer function, passing the command in the `command` parameter. For more information on the flatten function, see `CMFlattenProfile` (page 237).

```
enum {
    openReadSpool   = 1,    /* start read data process */
    openWriteSpool  = 2,    /* start write data process */
    readSpool       = 3,    /* read specified number of bytes */
    writeSpool      = 4,    /* write specified number of bytes */
    closeSpool      = 5     /* complete data transfer process */
};
```

Enumerator 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.

Constants for PostScript Data Formats

The ColorSync Manager provides the following constant declarations to specify the format of PostScript data.

```
enum {
    cmPS7bit    = 1,    /* data is 7-bit safe */
    cmPS8bit    = 2     /* data is 8-bit safe */
};
```

Enumerator descriptions

`cmPS7bit` The data is 7-bit safe—therefore the data could be in 7-bit ASCII encoding or in ASCII base-85 encoding.

`cmPS8bit` The data is 8-bit safe—therefore the data could be in 7-bit or 8-bit ASCII encoding.

Picture Comments

Your application uses the QuickDraw `PicComment` function, described in *Inside Macintosh: Imaging With QuickDraw*, to specify picture comments for beginning

or ending use of an embedded profile, turning color matching on or off, or embedding a profile identifier. The following sections describe constants you use to perform these operations:

Picture Comment Kinds for Profiles and Color Matching

The ColorSync Manager defines five picture comment kinds. You use these comments to embed a profile identifier, begin or end use of an embedded profile, and enable or disable color matching within drawing code sent to an output device. The `PicComment` function's `kind` parameter specifies the kind of picture comment.

IMPORTANT

Use a picture comment of kind `cmEndProfile` to explicitly terminate use of the currently effective embedded profile and begin use of the system profile. Otherwise, the currently effective profile remains in effect, leading to unexpected results if another picture follows that is meant to use the system profile and so isn't preceded by a profile. ▲

```
enum {
    cmBeginProfile      = 220,  /* begins 1.0 profile */
    cmEndProfile        = 221,  /* ends 2.x or 1.0 profile */
    cmEnableMatching    = 222,  /* turns on color matching */
    cmDisableMatching   = 223,  /* turns off color matching */
    cmComment           = 224   /* profile or profile identifier
                                   is embedded */
};
```

Enumerator descriptions

`cmBeginProfile` Indicates the beginning of a version 1.0 profile to embed. (To start a 2.x profile, you use `cmComment`.)

`cmEndProfile` Signals end of the use of an embedded version 2.x or 1.0 profile.

`cmEnableMatching`	Turns on color matching for the ColorSync Manager. Do not nest `cmEnableMatching` and `cmDisableMatching` pairs.
`cmDisableMatching`	Turns off color matching for the ColorSync Manager. 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 that point, the most recently used profile is reinstated.
`cmComment`	Provides information about a 2.x embedded profile or embedded profile identifier reference. This picture comment is followed by a 4-byte selector identifying what follows. "Picture Comment Selectors for Embedding Profile Information" (page 400) describes the possible selectors.

Picture Comment Selectors for Embedding Profile Information

To embed a version 2.x profile or profile identifier reference in a picture destined for display on another system or on a device such as a printer, your application uses the QuickDraw `PicComment` function. The ColorSync Manager provides the function `NCMUseProfileComment` (page 290) to embed picture comments. You specify a picture comment `kind` value of `cmComment` and a 4-byte selector describing the data in the picture comment. For sample code showing how to use `NCMUseProfileComment` to embed profile information, see "Embedding Profiles and Profile Identifiers" (page 112).

Because a profile may exceed QuickDraw's 32 KB size limit for a picture comment, your application can use an ordered series of picture comments to embed a large profile. Figure 3-7 (page 115) shows how a large profile is embedded in a PICT file picture.

You can also embed a profile identifier reference in a picture. The profile identifier may refer to a previously embedded profile, so that you don't have to embed the entire profile again, or it may refer to a profile stored on disk. When you embed a profile identifier, you can change certain values for the referred-to profile, including the quality flags and rendering intent. For more information on profile identifiers, see `CMProfileIdentifier` (page 359).

The following enumeration defines the 4-byte selector values your application uses to identify the beginning and continuation of profile data and to signal the end of it.

```
enum {
    cmBeginProfileSel           = 0,    /* start 2.x profile data */
    cmContinueProfileSel        = 1,    /* continuation of 2.x data */
    cmEndProfileSel             = 2,    /* end 2.x profile data */
    cmProfileIdentifierSel      = 3     /* profile identifier data */
};
```

Enumerator descriptions

`cmBeginProfileSel` Identifies the beginning of version 2.x profile data. The amount of profile data you can specify is limited to 32K minus 4 bytes for the selector.

`cmContinueProfileSel`
Identifies the continuation of version 2.x profile data. The amount of profile data you can specify is limited to 32K minus 4 bytes for the selector. You can use this selector repeatedly until all the profile data is embedded.

`cmEndProfileSel` Signals the end of version 2.x 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 `cmContinueProfileSel`, you must terminate the process with `cmEndProfileSel`. Note that this selector has a behavior that is different from the `cmEndProfile` picture comment described in “Picture Comment Kinds for Profiles and Color Matching” (page 399).

`cmProfileIdentifierSel`
Identifies the inclusion of profile identifier data. For information on embedding a profile identifier, see the function `NCMUseProfileComment` (page 290). For information on the format of profile identifier data, see “CMProfileIdentifier” (page 359).

Constants for Embedding Profiles and Profile Identifiers

The ColorSync Manager provides the following constant declarations to use with the function `NCMUseProfileComment` (page 290) for embedding picture comments. You use these constants to set the `flags` parameter to indicate whether to embed an entire profile or just a profile identifier.

```
enum {
    cmEmbedWholeProfile         = 0x00000000,   /* embed the whole profile */
    cmEmbedProfileIdentifier    = 0x00000001    /* embed just the profile identifier */
};
```

Enumerator descriptions

`cmEmbedWholeProfile`

When the `flags` parameter has the value `cmEmbedWholeProfile`, the `NCMUseProfileComment` function embeds the entire specified profile.

`cmEmbedProfileIdentifier`

When the `flags` parameter has the value `cmEmbedProfileIdentifier`, the `NCMUseProfileComment` function embeds a profile identifier for the specified profile.

Color Space Constants

The ColorSync Manager defines signature constants to identify a color space in a profile header, as well as constants for defining the color spaces themselves. These constants are described in the following sections:

- "Color Space Signatures" (page 402)
- "Color Packing for Color Spaces" (page 404); **changed in ColorSync 2.5**
- "Abstract Color Space Constants" (page 406); **changed in ColorSync 2.5**
- "Color Space Constants With Packing Formats" (page 409); **changed in ColorSync 2.5**

Color Space Signatures

A ColorSync profile header contains a `dataColorSpace` field that carries the signature of the data color space in which the color values in an image using the

profile are expressed. This enumeration defines the signatures for the color spaces supported by ColorSync for version 2.x 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',
    cmNamedData     = 'NAME'
};
```

Enumerator descriptions

`cmXYZData`	The XYZ data color space.
`cmLabData`	The L*a*b* data color space.
`cmLuvData`	The L*u*v* data color space.
`cmYxyData`	The Yxy data color space.
`cmRGBData`	The RGB data color space.
`cmGrayData`	The Gray data color space.
`cmHSVData`	The HSV data color space.
`cmHLSData`	The HLS data color space.
`cmCMYKData`	The CMYK data color space.
`cmCMYData`	The CMY data color space.
`cmMCH5Data`	The five-channel multichannel (HiFi) data color space.
`cmMCH6Data`	The six-channel multichannel (HiFi) data color space.
`cmMCH7Data`	The seven-channel multichannel (HiFi) data color space.
`cmMCH8Data`	The eight-channel multichannel (HiFi) data color space.

Color Packing for Color Spaces

CHANGED IN COLORSYNC 2.5

The ColorSync bitmap data type `CMBitmap` (page 380) 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 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 the constants described in "Abstract Color Space Constants" (page 406) to create values that identify a bitmap's color space. Your application does not specify color packing constants directly, but rather uses the combined constants, which are described in "Color Space Constants With Packing Formats" (page 409).

```
enum {
    cmNoColorPacking        = 0x0000,
    cmAlphaSpace            = 0x0080,
    cmWord5ColorPacking     = 0x0500,
    cmLong8ColorPacking     = 0x0800,
    cmLong10ColorPacking    = 0x0a00,
    cmAlphaFirstPacking     = 0x1000,
    cmOneBitDirectPacking   = 0x0b00
    cmAlphaLastPacking      = 0x0000,
    cm24_8ColorPacking      = 0x2100,
    cm32_8ColorPacking      = cmLong8ColorPacking,
    cm40_8ColorPacking      = 0x2200,
    cm48_8ColorPacking      = 0x2300,
    cm56_8ColorPacking      = 0x2400,
    cm64_8ColorPacking      = 0x2500,
    cm32_16ColorPacking     = 0x2600,
    cm32_32ColorPacking     = 0x2700,
    cm48_16ColorPacking     = 0x2900,
    cm64_16ColorPacking     = 0x2A00
};
```

Enumerator descriptions

`cmNoColorPacking` This constant is not used for ColorSync bitmaps.

`cmAlphaSpace` An alpha channel component is added to the color value.

`cmWord5ColorPacking`

The color values for three 5-bit color channels are stored consecutively in 16-bits, with the highest order bit unused.

`cmLong8ColorPacking`

The color values for three or four 8-bit color channels are stored consecutively in a 32-bit long. For three channels, this constant is combined with either `cmAlphaFirstPacking` or `cmAlphaLastPacking` to indicate whether the unused eight bits are located at the beginning or end.

`cmLong10ColorPacking`

The color values for three 10-bit color channels are stored consecutively in a 32-bit long, with the two highest order bits unused.

`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 function `CWCheckBitMap` (page 279); the bitmap must be only 1 bit deep.

`cm24_8ColorPacking`

The color values for three 8-bit color channels are stored in consecutive bytes, for a total of 24 bits.

`cm32_8ColorPacking`

The color values for four 8-bit color channels are stored in consecutive bytes, for a total of 32 bits.

`cm40_8ColorPacking`

The color values for five 8-bit color channels are stored in consecutive bytes, for a total of 40 bits.

`cm48_8ColorPacking`

The color values for six 8-bit color channels are stored in consecutive bytes, for a total of 48 bits.

`cm56_8ColorPacking`

The color values for seven 8-bit color channels are stored in consecutive bytes, for a total of 56 bits.

`cm64_8ColorPacking`
: The color values for eight 8-bit color channels are stored in consecutive bytes, for a total of 64 bits.

`cm32_16ColorPacking`
: The color values for two 16-bit color channels are stored in a 32-bit word.

`cm32_32ColorPacking`
: The color value for a 32-bit color channel is stored in a 32-bit word.

`cm48_16ColorPacking`
: The color values for three 16-bit color channels are stored in 48 consecutive bits.

`cm64_16ColorPacking`
: The color values for four 16-bit color channels are stored in 64 consecutive bits.

VERSION NOTES

The constants `cm48_16ColorPacking` and `cm64_16ColorPacking` were added in ColorSync version 2.5.

Abstract Color Space Constants

CHANGED IN COLORSYNC 2.5

The data type `CMBitmap` (page 380) defines a bitmap for an image whose colors can be matched with the function `CWMatchBitmap` (page 276) or color-checked with the function `CWCheckBitMap` (page 279).

The `space` field of the `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. These values together make up 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.

For the `CWMatchBitmap` function to perform color matching successfully, the color space specified in the `CMBitmap` data type's `space` field must correspond to the color space specified in the profile's `dataColorSpace` field. The source bitmap and source profile values must match and the destination bitmap and

destination profile values must match. For the `CWCheckBitMap` function to perform color checking successfully, 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 execute successfully as long as the color spaces are the same without regard for the packing format specified by the bitmap.

The following enumeration defines constants for abstract color spaces which, when combined with a packing format constant as described in "Color Packing for Color Spaces" (page 404), can be used in the `space` field of the `CMBitmap` structure. The combined constants are shown in "Color Space Constants With Packing Formats" (page 409).

```
enum {
    cmNoSpace           = 0,
    cmRGBSpace          = 1,
    cmCMYKSpace         = 2,
    cmHSVSpace          = 3,
    cmHLSSpace          = 4,
    cmYXYSpace          = 5,
    cmXYZSpace          = 6,
    cmLUVSpace          = 7,
    cmLABSpace          = 8,
    cmReservedSpace1    = 9,
    cmGraySpace         = 10,
    cmReservedSpace2    = 11,
    cmGamutResultSpace  = 12,
    cmNamedIndexedSpace = 16,
    cmMCFiveSpace       = 17,
    cmMCSixSpace        = 18,
    cmMCSevenSpace      = 19,
    cmMCEightSpace      = 20,
    cmRGBASpace         = cmRGBSpace + cmAlphaSpace,
    cmGrayASpace        = cmGraySpace + cmAlphaSpace
};
```

Enumerator 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 for the resulting bitmap pointed to by the `resultBitMap` field of the function `CWCheckBitMap` (page 279). A bitmap never uses this constant alone. Instead, it uses the constant `cmGamutResult1Space`, which combines

	`cmGamutResultSpace` and `cmOneBitDirectPacking` to define a bitmap that is 1 bit deep.
`cmMCFiveSpace`	A five-channel multichannel (HiFi) data color space.
`cmMCSixSpace`	A six-channel multichannel (HiFi) data color space.
`cmMCSevenSpace`	A seven-channel multichannel (HiFi) data color space.
`cmMCEightSpace`	An eight-channel multichannel (HiFi) data color space.
`cmRGBASpace`	An RGB color space composed of red, green, and blue color value components and an alpha channel component. ColorSync does not currently support bitmaps that use this constant alone. Instead, this constant indicates 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.
`cmGrayASpace`	A luminance color space with two components, a gray component followed by an alpha channel component. Each component value is 16 bits.

VERSION NOTES

The constants `cmRGBASpace` and `cmGrayASpace` were moved to this enum from "Color Space Constants With Packing Formats" (page 409) in ColorSync version 2.5.

Color Space Constants With Packing Formats

CHANGED IN COLORSYNC 2.5

The following enumeration defines constants for color spaces which can specify color values for a bitmap image. As a rule, these constants include a packing format, defined in "Color Packing for Color Spaces" (page 404). You can use these constants to set the `space` field of the `CMBitmap` type definition identifies the color space in which the colors of the bitmap image are specified, as described in "Abstract Color Space Constants" (page 406).

```
enum {
    cmGray16Space           = cmGraySpace,
    cmGrayA32Space          = cmGrayASpace,
```

```
    cmRGB16Space            = cmWord5ColorPacking + cmRGBSpace,
    cmRGB24Space            = cm24_8ColorPacking + cmRGBSpace,
    cmRGB32Space            = cm32_8ColorPacking + cmRGBSpace,
    cmRGB48Space            = cm48_16ColorPacking + cmRGBSpace,
    cmARGB32Space           = cm32_8ColorPacking + cmAlphaFirstPacking + cmRGBASpace,
    cmRGBA32Space           = cm32_8ColorPacking + cmAlphaFirstPacking + cmRGBASpace,
    cmCMYK32Space           = cm32_8ColorPacking + cmCMYKSpace,
    cmCMYK64Space           = cm64_16ColorPacking + cmCMYKSpace,
    cmHSV32Space            = cmLong10ColorPacking + cmHSVSpace,
    cmHLS32Space            = cmLong10ColorPacking + cmHLSSpace,
    cmYXY32Space            = cmLong10ColorPacking + cmYXYSpace,
    cmXYZ32Space            = cmLong10ColorPacking + cmXYZSpace,
    cmLUV32Space            = cmLong10ColorPacking + cmLUVSpace,
    cmLAB24Space            = cm24_8ColorPacking + cmLABSpace,
    cmLAB32Space            = cmLong10ColorPacking + cmLABSpace,
    cmLAB48Space            = cm48_16ColorPacking + cmLABSpace,
    cmGamutResult1Space     = cmOneBitDirectPacking + cmGamutResultSpace
    cmNamedIndexed32Space   = cm32_32ColorPacking + cmNamedIndexedSpace,
    cmMCFive8Space          = cm40_8ColorPacking + cmMCFiveSpace,
    cmMCSix8Space           = cm48_8ColorPacking + cmMCSixSpace,
    cmMCSeven8Space         = cm56_8ColorPacking + cmMCSevenSpace,
    cmMCEight8Space         = cm64_8ColorPacking + cmMCEightSpace
};
```

Enumerator descriptions

`cmGray16Space` A luminance color space with a single 16-bit component, gray.

`cmGrayA32Space` 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.

`cmRGB24Space` 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 24 bits.

`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.
`cmRGB48Space`	An RGB color space composed of red, green, and blue components whose values are packed with 16 bits of storage per component. The storage size for a color value expressed in this color space is 48 bits.
`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.
`cmRGBA32Space`	An RGB color space composed of red, green, and blue color value components, followed by an alpha channel component. 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.
`cmCMYK64Space`	A CMYK color space composed of cyan, magenta, yellow, and black components whose values are packed with 16 bits of storage per component. The storage size for a color value expressed in this color space is 64 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.

`cmLAB24Space` An L*a*b* color space composed of L*, a*, and b* components whose values are packed with 8 bits per component. The storage size for a color value expressed in this color space is 24 bits. The 8-bit unsigned a* and b* channels are interpreted numerically as ranging from -128.0 to approximately 128.0.

`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. The 10-bit unsigned a* and b* channels are interpreted numerically as ranging from -128.0 to approximately 128.0.

`cmLAB48Space` An L*a*b* color space composed of L*, a*, and b* components whose values are packed with 16 bits per component. The storage size for a color value expressed in this color space is 48 bits. The 16-bit unsigned a* and b* channels are interpreted numerically as ranging from -128.0 to approximately 128.0.

`cmGamutResult1Space`

A gamut result color space for the resulting bitmap pointed to by the `resultBitMap` field of the function `CWCheckBitMap` (page 279), with 1-bit direct packing. A pixel in the returned bitmap with value 1 (displayed as black) indicates an out-of-gamut color, while a pixel value of 0 (white) indicates a color that is in gamut.

`cmNamedIndexed32Space`

A color space where each color is stored as a single 32-bit

	value, specifying an index into a named color space. The storage size for a color value expressed in this color space is 32 bits.
`cmMCFive8Space`	A five-channel multichannel (HiFi) data color space, whose values are packed with 8 bits per component. The storage size for a color value expressed in this color space is 40 bits.
`cmMCSix8Space`	A six-channel multichannel (HiFi) data color space, whose values are packed with 8 bits per component. The storage size for a color value expressed in this color space is 48 bits.
`cmMCSeven8Space`	A seven-channel multichannel (HiFi) data color space, whose values are packed with 8 bits per component. The storage size for a color value expressed in this color space is 56 bits.
`cmMCEight8Space`	A eight-channel multichannel (HiFi) data color space, whose values are packed with 8 bits per component. The storage size for a color value expressed in this color space is 64 bits.

VERSION NOTES

The constants `cmRGBASpace` and `cmGrayASpace` were moved to "Abstract Color Space Constants" (page 406) in ColorSync version 2.5.

The constants `cmGray16Space`, `cmGrayA32Space`, `cmRGB48Space`, `cmCMYK64Space`, and `cmLAB48Space` were added in ColorSync version 2.5.

ColorSync Flag Constants

The ColorSync Manager defines the structure `CM2Header` (page 354) to represent the profile header for the version 2.x profile format defined by the ICC. The ColorSync Manager also defines constants to set and test flag bits in the `flags`, `deviceAttributes`, and `renderingIntent` fields of the `CM2Header` profile header structure.

The next sections describe ColorSync constants for evaluating and setting bits in these fields

- "Flag Mask Definitions for Version 2.x Profiles" (page 414)
- "Quality Flag Values for Version 2.x Profiles" (page 417)

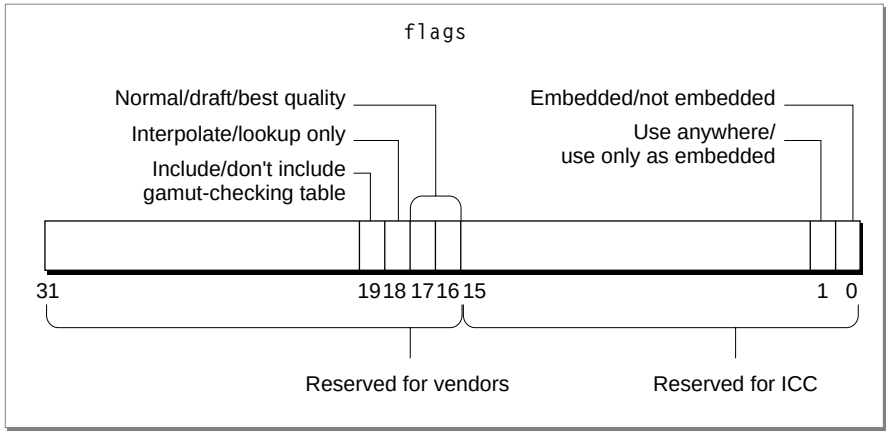
■ “Device Attribute Values for Version 2.x Profiles” (page 418)

■ “Rendering Intent Values for Version 2.x Profiles” (page 419)

Flag Mask Definitions for Version 2.x Profiles

The `flags` field of the structure `CM2Header` (page 354) is an unsigned long value whose bits specify information about a profile. The ICC reserves the use of bits 0 to 15 and has assigned values to bits 0 and 1. Bits 16 to 31 are reserved for use by color management system (CMS) vendors. ColorSync has assigned values to bits 16 through 19. Figure 5-1 shows the bit assignments of the `flags` field specified by ColorSync and by the ICC.

Figure 5-1 The `flags` field of the `CM2Header` structure



The following enumeration defines masks your application can use to set or test various bits in the `flags` field of the `CM2Header` structure:

```
enum {
/* these bits of the flags field are defined and reserved by the ICC */
    cmICCReservedFlagsMask  = 0x0000FFFF,
/* if bit 0 is 0 then not embedded profile, if 1 then embedded profile */
    cmEmbeddedMask          = 0x00000001,
/* if bit 1 is 0 then ok to use anywhere, if 1 then use as embedded profile only */
```

```
    cmEmbeddedUseMask       = 0x00000002,
/* these bits of the flags field are defined and reserved for CMS vendors */
    cmCMSReservedFlagsMask  = 0xFFFF0000,
/* if bits 16-17 == 0 then normal, if 1 then draft, if 2 then best */
    cmQualityMask           = 0x00030000,
/* if bit 18 is 0 then interpolation, if 1 then lookup only */
    cmInterpolationMask     = 0x00040000,
/* if bit 19 is 0 then create gamut-checking info, if 1 then no gamut-checking info */
    cmGamutCheckingMask     = 0x00080000
};
```

Enumerator descriptions

`cmICCReservedFlagsMask`

This mask provides access to bits 0 through 15 of the `flags` field, which are defined and reserved by the ICC. For more information, see the *International Color Consortium Profile Format Specification*, and the next two mask definitions.

To obtain a copy of the ICC specification, or to get other information about the ICC, visit the ICC Web site at <http://www.color.org/>.

`cmEmbeddedMask` This mask provides access to bit 0 of the `flags` field, which specifies whether the profile is embedded. It has the value 1 if the profile is embedded, 0 if it is not.

`cmEmbeddedUseMask` This mask provides access to bit 1 of the `flags` field, which specifies whether the profile can be used independently or can only be used as an embedded profile. It has the value 0 if the profile can be used anywhere, 1 if it must be embedded.

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 `cmCopyrightTag` profile tag (defined in the `CMICCProfile.h` header file).

`cmCMSReservedFlagsMask`

This mask provides access to bits 16 through 31 of the `flags` field, which are available for a color management system (CMS) vendor, such as ColorSync. ColorSync's default

CMM uses bits 16 through 19 to provide hints for color matching, as described in the following three mask definitions. Other CMM vendors should follow the same conventions.

`cmQualityMask` This mask provides access to bits 16 and 17 of the `flags` field, which specify the preferred quality and speed preferences for color matching. In general, the higher the quality the slower the speed. For example, best quality is slowest, but produces the highest quality result.

Bits 16 and 17 have the value 0 for normal quality, 1 for draft quality, and 2 for best quality. "Quality Flag Values for Version 2.x Profiles" (page 417) describes the constants ColorSync defines to test or set these bits.

This feature is provided by the ColorSync Manager; it is not defined by the ICC profile specification.

`cmInterpolationMask`

This mask provides access to bit 18 of the `flags` field, which specifies whether to use interpolation in color matching. The value 0 specifies interpolation. The value 1 specifies table lookup without interpolation. Specifying lookup only improves speed but can reduce accuracy. You might use lookup only for a monitor profile, for example, when high resolution is not crucial.

This feature is provided by the ColorSync Manager; it is not defined by the ICC profile specification.

`cmGamutCheckingMask`

This mask provides access to bit 19 of the `flags` field. When you use a profile to create a color world, bit 19 specifies whether the color world should include information for gamut checking. It has the value 0 if the color world should include a gamut-checking table, 1 if gamut-checking information is not required. ColorSync can create a color world without a gamut table more quickly and in less space.

Many applications do not perform gamut checking, so they should set this bit to 1. However, if you call a color checking function such as `CWCheckColors` (page 283), `CWCheckBitMap` (page 279), or `CWMatchPixMap` (page 272), after setting a profile's gamut-checking bit so that the color

world does not contain gamut information, these routines return the `cmCantGamutCheckError` error.

This feature is provided by the ColorSync Manager; it is not defined by the ICC profile specification.

Quality Flag Values for Version 2.x Profiles

The following enumeration defines the possible values for the quality bits in the `flags` field of the `CM2Header` structure. To determine the value of the quality flag, you mask the `flags` field of the profile header with the `cmQualityMask` mask, right shift 16 bits, then compare the result to the enumerated constants shown below. For more information on the quality flag, see "Flag Mask Definitions for Version 2.x Profiles" (page 414).

When you start a color-matching session, ColorSync sends all involved profiles to the color management module (CMM). The CMM extracts the information it needs from the profiles and stores an internal representation in private memory. ColorSync's default CMM samples the input space and stores the results in a lookup table, a common technique that speeds up conversion for runtime applications. The size of the table is based on the quality flag setting in the source profile header. The setting of the quality flag can affect the memory requirements, accuracy, and speed of the color-matching session. In general, the higher the quality setting, the larger the lookup table, the more accurate the matching, and the slower the matching process. Note however, that the default CMM currently produces the same results for both normal and draft mode.

```
enum {
    cmNormalMode    = 0,    /* use default method for quality */
    cmDraftMode     = 1,    /* sacrifice quality to minimize resource
                               requirements */
    cmBestMode      = 2     /* ensure highest possible quality */
};
```

Enumerator descriptions

`cmNormalMode` This is the default setting. Normal mode indicates that the CMM should use its default method to compromise between performance and resource requirements.

`cmDraftMode` Draft mode indicates that the CMM should sacrifice quality, if necessary, to minimize resource requirements.

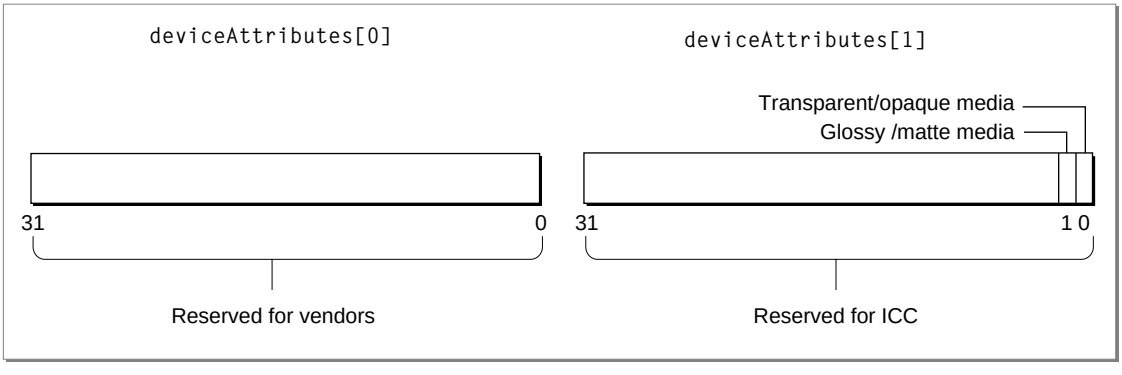
Note that the default CMM currently produces the same results for both normal and draft mode.

`cmBestMode` Best mode indicates that the CMM should maximize resource usage to ensure the highest possible quality.

Device Attribute Values for Version 2.x Profiles

The ColorSync Manager defines the structure `CM2Header` (page 354) to represent the profile header for the version 2.x profile format defined by the ICC.The `deviceAttributes` field of the `CM2Header` structure is an array of two unsigned long values whose bits specify information about a profile. The ICC reserves the use of `deviceAttributes[1]` and has assigned values to bits 0 and 1. All the bits of `deviceAttributes[0]` are reserved for use by color management system (CMS) vendors. Figure 5-2 shows the bit assignments for the `deviceAttributes` field.

Figure 5-2 The `deviceAttributes` field of the `CM2Header` structure



The following enumeration defines masks your application can use to set or test bits in `deviceAttributes[1]`.

```
enum {
    /* if bit 0 is 0 then reflective media, if 1 then transparent media */
    cmReflectiveTransparentMask = 0x00000001,
```

```
    /* if bit 1 is 0 then glossy media, if 1 then matte media*/
    cmGlossyMatteMask = 0x00000002
};
```

Enumerator descriptions

`cmReflectiveTransparentMask`

Bit 0 of `deviceAttributes[1]` specifies whether the media is transparent or reflective. If it has the value 0, the media is reflective; if it has the value 1, the media is transparent. Use the `cmReflectiveTransparentMask` mask to set the transparent / reflective bit in `deviceAttributes[1]` or to clear all bits except the transparent / reflective bit.

`cmGlossyMatteMask`

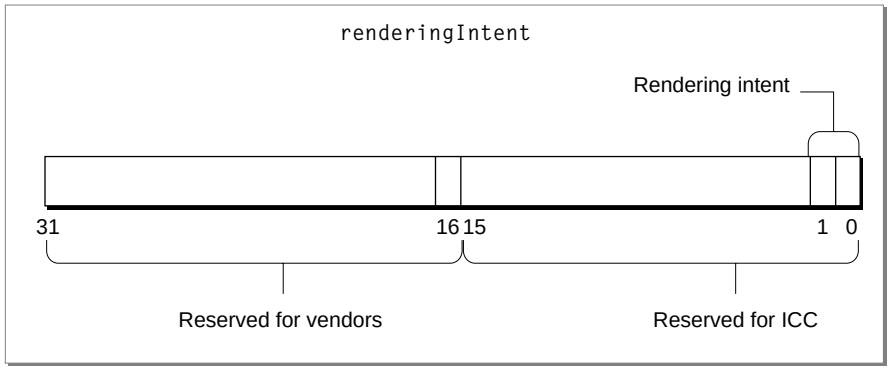
Bit 1of `deviceAttributes[1]` specifies whether the media is glossy or matte. If it has the value 0, the media is glossy; if it has the value 1, the media is matte. Use the `cmGlossyMatteMask` mask to set the glossy / matte bit in `deviceAttributes[1]` or to clear all bits except the glossy / matte bit.

Rendering Intent Values for Version 2.x Profiles

The ColorSync Manager defines the structure `CM2Header` (page 354) to represent the profile header for the version 2.x profile format defined by the ICC. The `renderingIntent` field of the `CM2Header` structure is an unsigned long value whose bits specify information about a profile. The ICC reserves the use of bits 0 to 15 and has assigned values to bits 0 and 1. Bits 16 to 31 are reserved for use by color management system (CMS) vendors. Figure 5-3 shows the bit assignments of the `renderingIntent` field specified by the ICC.

Rendering intent controls the approach a CMM uses to translate the colors of an image to the color gamut of a destination device. Your application can set a profile's rendering intent, for example, based on a user's choice of the preferred approach for rendering an image.

Figure 5-3 The `renderingIntent` field of the `CM2Header` structure



The following enumeration defines the four possible values for the rendering intent bits of the `renderingIntent` field. Because rendering intent is specified by the low two bits, and because no other bits are currently defined for this field, you can use the constants defined here to test or set the value of the entire field, without concern for possible information stored in other bits.

```
enum {
    cmPerceptual            = 0,    /* scale colors to fit in gamut */
    cmRelativeColorimetric  = 1,    /* don't change colors that fall in
                                       the gamuts of both devices */
    cmSaturation            = 2,    /* maintain relative saturation */
    cmAbsoluteColorimetric  = 3     /* base on idealized, device-
                                       independent color space */
};
```

Enumerator descriptions

`cmPerceptual` All the colors of a given gamut can 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.

Video Card Gamma Constants

NEW IN COLORSYNC 2.5

Starting with version 2.5, ColorSync supports an optional profile tag for video card gamma. The tag specifies gamma information, stored either as a formula or in table format, to be loaded into the video card when the profile containing the tag is put into use. As of version 2.5, the only ColorSync function that attempts to take advantage of video card gamma data is `CMSetProfileByAVID` (page 300).

The following sections describe the constants you use to work with the video card gamma profile tag:

- "Video Card Gamma Tag" (page 421)
- "Video Card Gamma Tag Type" (page 422)
- "Video Card Gamma Storage Type" (page 422)

Video Card Gamma Tag

NEW IN COLORSYNC 2.5

When you create a tag to store video card gamma data in a profile, you use the `cmVideoCardGammaTag` constant to specify the tag.

```
enum
{
    …,
    cmVideoCardGammaTag = FOUR_CHAR_CODE('vcgt')
};
```

Enumerator descriptions

`cmVideoCardGammaTag`
Constant for profile tag that specifies video card gamma information.

Video Card Gamma Tag Type

NEW IN COLORSYNC 2.5

You use the `cmSigVideoCardGammaType` constant to specify the signature type for a video card gamma tag. That is, you use this constant to set the `typeDescriptor` field of the `CMVideoCardGammaType` (page 386) structure. There is currently only one type possible for a video card gamma tag.

```
enum
{
    cmSigVideoCardGammaType = FOUR_CHAR_CODE('vcgt')
};
```

Enumerator descriptions

`cmSigVideoCardGammaType`
Constant that specifies video card gamma type signature in a video card gamma profile tag.

Video Card Gamma Storage Type

NEW IN COLORSYNC 2.5

A video card gamma profile tag can store gamma data either as a formula or as a table of values. You use a storage type constant to specify which data storage type the tag uses.

IMPORTANT
If the video card uses a different format than the format you specify (for example, the card uses data in table format and you supply data in formula format), ColorSync will adapt the data you supply to match the format the card expects. ▲

```
enum
{
    cmVideoCardGammaTableType = 0,
    cmVideoCardGammaFormulaType = 1,
};
```

Enumerator descriptions

`cmVideoCardGammaTableType`
The video card gamma data is stored in a table format. See “CMVideoCardGammaTable” (page 387) for a description of the table format.

`cmVideoCardGammaFormulaType`
The video card gamma tag data is stored as a formula. See “CMVideoCardGammaFormula” (page 388) for a description of the formula format.

PrGeneral Function Operation Codes

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,   /* enable or disable color matching; supported in
                                       both ColorSync 1.0 and 2.x */
    registerProfileOp       = 13    /* register profile with driver; supported in
                                       ColorSync 1.0 only */
};
```

Enumerator descriptions

`enableColorMatchingOp`
Use this operation code with the `PrGeneral` function to turn color matching on or off. This code is supported by both ColorSync 2.x and ColorSync 1.0. For more information, see `TEnableColorMatchingBlk` (page 390).

`registerProfileOp` Use this operation code with the `PrGeneral` function to register a profile with a driver. This code is supported only by ColorSync 1.0.

Element Tags and Signatures for Version 1.0 Profiles

The ICC version 2.x profile format differs from the version 1.0 profile format, and ColorSync Manager routines for updating a profile and searching for profiles do not work with version 1.0 profiles. However, your application can use version 1.0 profiles with all other ColorSync routines. For example, you can open a version 1.0 profile using the function `CMOpenProfile` (page 222), obtain the version 1.0 profile header using the function `CMGetProfileHeader` (page 245), and access version 1.0 profile elements using the function `CMGetProfileElement` (page 243).

To make this possible, the ColorSync Manager 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',   /* signature for XYZ chromaticities tag */
    cmCS1TRCTag         = 'trc ',   /* signature for profile tonal response curve
                                       data from associated device */
    cmCS1NameTag        = 'name',   /* signature for profile name string tag */
    cmCS1CustTag        = 'cust'    /* signature for private data for custom CMM */
};
```

Enumerator descriptions

`cmCS1ChromTag` 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).

`cmCS1TRCTag` The tag signature for profile tonal response curve data for the associated device.

`cmCS1NameTag` 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.

`cmCS1CustTag` Private data for a custom CMM.

Result Codes for the ColorSync Manager

noErr	0	No error (not specific to ColorSync)
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 use
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	Tag index out of range
cmCantDeleteElement	–4202	Can't delete the specified profile element
cmFatalProfileErr	–4203	Returned from File Manager while updating a profile file in response to CMUpdateProfile; profile content may be corrupted
cmInvalidProfile	–4204	Profile reference is invalid or refers to an inappropriate profile
cmInvalidProfileLocation	–4205	Operation not supported for this profile location
cmInvalidSearch	–4206	Bad search handle
cmSearchError	–4207	Internal error occurred during profile search
cmErrIncompatibleProfile	–4208	Unspecified profile error
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 end matching—no graphics devices 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
`cmNamedColorNotFound`	–4216	The specified named color was not found in the specified profile
`cmCantGamutCheckError`	-4217	Gamut checking not supported by this color world—that is, the color world does not contain a gamut table because it was built with gamut checking turned off

CHAPTER 6

Developing Color Management Modules

Contents

This section gives a brief overview of color management modules (CMMs) and the role a CMM plays in the ColorSync color management system. You should read this section if you are a third-party developer who creates CMMs that ColorSync (versions 2.0 and greater) can use instead of, or in conjunction with, the default CMM.

Before reading this section, you should read “Introduction to ColorSync” (page 45) for a more complete conceptual explanation of how a CMM fits within the ColorSync system. If you are unfamiliar with terms and concepts such as profile, color space, CMM, and color management, or would like to review these topics, you should also read “Introduction to Color and Color Management Systems” (page 25).

At a minimum, a ColorSync-compatible CMM must be able to match colors across color spaces belonging to different base families and check colors expressed in the color gamut of one device against the color gamut of another device.

In addition to the minimum set of requests a CMM must service, a CMM can also implement support for other requests a ColorSync-supportive application or device driver might make. Among the optional services a CMM might provide are verifying if a particular profile contains the base set of required elements for a profile of its type and directing the process of converting profile data embedded in a graphics file to data in an external profile file accessed through a profile reference and vice versa. A CMM can also provide services for PostScript printers by obtaining or deriving from a profile specific data required by PostScript printers for color-matching processes and returning the data in a format that can be sent to the PostScript printer.

This section provides a high-level discussion of the required and optional ColorSync Manager request codes your CMM might be called to handle, and also describes the Component Manager required request codes to which every component must respond.

For complete details on components and their structure, see the chapter “Component Manager” in *Inside Macintosh: More Macintosh Toolbox*.

About Color Management Modules

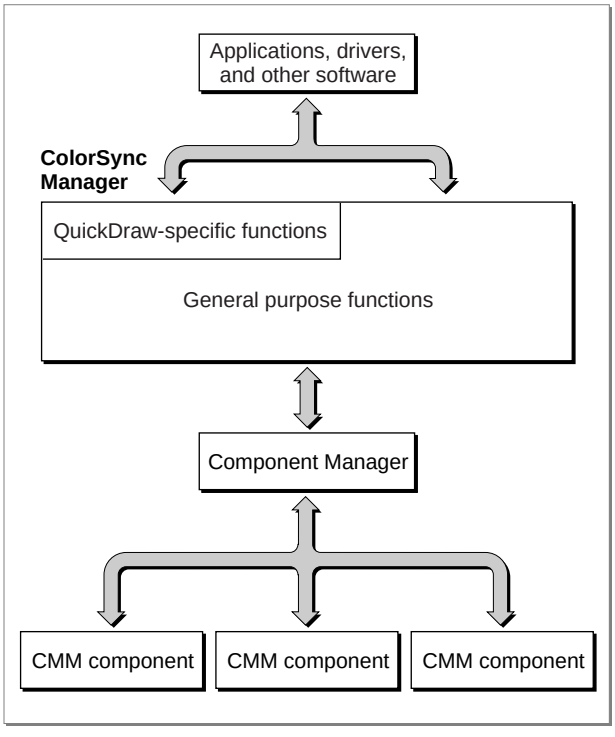
A color management module (CMM) is a component that implements color matching, color gamut checking, and other services and performs these services in response to requests from ColorSync-supportive applications or device drivers.

A CMM component interacts directly with the Component Manager, which calls the CMM on behalf of the ColorSync Manager and the requesting application or driver. When they call ColorSync Manager functions to request color-matching and color gamut-checking services, ColorSync-supportive applications and device drivers specify the profiles to use. These profiles characterize the devices involved; they include information giving the color spaces and the color gamuts of the devices and the preferred CMM to carry out the work. A CMM uses the information contained in these profiles to perform the processing required to service requests. Figure 6-1 shows the relationship between a ColorSync-supportive application or driver, the ColorSync Manager, the Component Manager, and one or more available CMM components.

A CMM should support all seven classes of profiles defined by the ICC. For information on the seven classes of profiles, see “Profile Class” (page 396) or the *International Color Consortium Profile Format Specification*, version 2.x or higher. To obtain a copy of the specification, or to get other information about the ICC, visit the ICC Web site at <http://www.color.org/>.

In some cases, a CMM will not be able to convert and match colors directly from the color space of one profile to that of another. Instead, it will need to convert colors to the device-independent color space specified by the profile. A CMM uses device-independent color spaces, or interchange color spaces, to interchange color data from the native color space of one device to the native color space of another device.The profile connection space field of a profile header specifies the interchange color space for that profile. Version 2.x of the ColorSync Manager supports two interchange color spaces: XYZ and Lab.

Figure 6-1 The ColorSync Manager and the Component Manager



When interchange color spaces are involved, the ColorSync Manager handles the process, which is largely transparent to the CMM. The ColorSync Manager passes to the CMM the correct profiles for color matching. For example, in a case in which both the source and destination profile's CMMs are required to complete the color matching using color space profiles, the ColorSync Manager calls the source profile's CMM with the source profile and an interchange color space profile. Then it calls the destination profile's CMM with an interchange color space profile and the destination profile. The ColorSync Manager assesses the requirements and breaks the process down so that the correct CMM is called with the correct set of profiles. This process is described from the perspective of an application or device driver in "How the ColorSync Manager Selects a CMM" (page 84).

A CMM uses lookup tables and algorithms for color matching, using one device to preview the color reproduction capabilities of another device, and checking for colors that cannot be reproduced.

Creating a Color Management Module

This section describes how to create a CMM component, including how to respond to required Component Manager and ColorSync Manager requests and optional ColorSync Manager requests. For more detailed information on working with components, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.

Creating a Component Resource for a CMM

A CMM is stored as a component resource. It contains a number of resources, including the standard component resource (a resource of type `'thng'`) required of any Component Manager component. In addition, a CMM must contain code to handle required request codes passed to it by the Component Manager. This includes support for Component Manager required request codes as well as ColorSync Manager required request codes. For an example of the resources your CMM should include, refer to the DemoCMM project available with the ColorSync SDK.

To allow the ColorSync Manager to use your CMM when a profile specifies it as its preferred CMM, your CMM should be located in the Extensions folder, where it will automatically be registered at startup. The file type for component files must be set to `'thng'`.

The Component Resource

The component resource contains all the information needed to register a code resource as a component. Information in the component resource tells the Component Manager where to find the code for the component. As part of the component resource, you must provide a component description record that specifies the component type, subtype, manufacturer, and flags. Here is the data structure for the component description:

```
struct ComponentDescription {
    OSType              componentType;
    OSType              componentSubType;
    OSType              componentManufacturer;
    unsigned long       componentFlags;
    unsigned long       componentFlagsMask;
};
```

The following are the key fields of the component description data structure for creating a CMM component:

- The `componentType` field contains a unique 4-byte code specifying the resource type and resource ID of the component's executable code. For your CMM, set this field to `'cmm '`.
- The `componentSubType` field indicates the type of services your CMM provides. You should set this field to your CMM name. This value must match exactly the value specified in the profile header's `CMMType` field. You must register this value with the International Color Consortium (ICC). To obtain information about the ICC, visit the ICC Web site at <http://www.color.org/>.
- The `componentManufacturer` field indicates the creator of the CMM. You may set this field to any value you wish.
- The `componentFlags` field is a 32-bit field that provides additional information about your CMM component. The high-order 8 bits are reserved for definition by the Component Manager. The low-order 24 bits are specific to each component type. You can use these flags to indicate any special capabilities or features of your component.

Note
Values you specify for all fields except the `componentType` field must include at least one uppercase character. Apple Computer reserves values containing all lowercase characters for its own use. ◆

The Extended Component Resource

Since it was first defined, the component resource has been extended to include additional information. That additional information includes the following field for specifying the version of your component:

```
long    componentVersion;   /* version of Component */
```

- The `componentVersion` field indicates the version of the CMM. For related information on specifying the CMM version, see "Required Component Manager Request Codes" (page 435).

For more information on component data types, see the following files from the Universal Interfaces distributed with development systems for the Mac OS:

- Components.h
- Components.r

How Your CMM Is Called by the Component Manager

Because a CMM is a direct client of the Component Manager, it must conform to the Component Manager's interface requirements, including supporting and responding to required Component Manager calls.

The code for your CMM should be contained in a resource. The Component Manager expects the entry point to this resource to be a function having this format:

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

Whenever the Component Manager receives a request for your CMM, it calls your component's entry point and passes any parameters, along with information about the current connection, in a data structure of type `ComponentParameters`. This entry point must be the first function in your CMM's code segment. The Component Manager also passes a handle to the private storage (if any) associated with the current instance of your component. Here is the component parameters data structure, which is described in detail in *Inside Macintosh: More Macintosh Toolbox*.

```
struct ComponentParameters {
        unsigned char       flags;
        unsigned char       paramSize;
        short               what;
        long                params[1];
};
```

The first field of the `ComponentParameters` data structure is reserved. The following three fields carry information your CMM needs to perform its processing. The `what` field contains a value that identifies the type of request. The `paramSize` field specifies the size in bytes of the parameters passed from the ColorSync-supportive calling application to your CMM. The parameters themselves are passed in the `params` field.

Required Component Manager Request Codes

At a minimum, your CMM must handle the required Component Manager and required ColorSync Manager request codes. The required Component Manager request codes are defined by these constants:

- `kComponentOpenSelect` (`-1`)
  Requests that you open an instance of the component. For more information, see "Establishing the Environment for a New Component Instance" (page 440).
- `kComponentCloseSelect` (`-2`)
  Requests that you close the component instance. For more information, see "Releasing Private Storage and Closing the Component Instance" (page 440).
- `kComponentCanDoSelect` (`-3`)
  Requests that you tell whether your CMM handles a specific request. For more information, see "Determining Whether Your CMM Supports a Request" (page 441).
- `kComponentVersionSelect` (`-4`)
  Requests that you return your CMM's version number. For more information, see "Providing Your CMM Version Number" (page 441). Note that if you provide your version number in an extended component resource, the Component Manager can obtain the version number without having to call your code that handles this request code.

Required ColorSync Manager Request Codes

Your CMM must also be able to handle the required ColorSync Manager request codes defined by these constants:

- `kCMMMatchColors` (`1`)
  Requests that you color match the specified colors from one color space to

another. For more information, see “Matching a List of Colors to the Destination Profile’s Color Space” (page 443).

- `kCMMCheckColors` (2)
  Requests that you check the specified colors against the gamut of the destination device whose profile is specified. For more information, see “Checking a List of Colors” (page 443).

- `kNCMMInit` (6)
  Requests that you initialize the current component instance of your CMM for a ColorSync Manager 2.x session. For more information, see “Initializing the Current Component Instance for a Two-Profile Session” (page 442).

Optional ColorSync Manager Request Codes

The Component Manager may also call your CMM with the following ColorSync Manager request codes that are considered optional. A CMM may support these requests, although you are not required to do so.

- `kCMMInit` (0)
  Requests that you initialize the current component instance of your CMM for a ColorSync 1.0 session. This is a required request code only if your CMM supports ColorSync 1.0 profiles.

- `kCMMMatchPixMap` (3)
  Requests that you match the colors of a pixel map image to the color gamut of a destination profile, replacing the original pixel colors with their corresponding colors. For more information, see “Matching the Colors of a Pixel Map Image” (page 448).

- `kCMMCheckPixMap` (4)
  Requests that you check the colors of a pixel map image against the gamut of a destination device for inclusion and report the results. For more information, see “Checking the Colors of a Pixel Map Image” (page 449).

- `kCMMConcatenateProfiles` (5)
  This request code is for backward compatibility with ColorSync 1.0.

- `kCMMConcatInit` (7)
  Requests that you initialize any private data your CMM will need for a color session involving the set of profiles specified by the profile array pointed to by the `profileSet` parameter. For more information, see “Initializing the Component Instance for a Session Using Concatenated Profiles” (page 450).

- `kCMMValidateProfile` (8)
  Requests that you test a specific profile to determine if the profile contains the minimum set of elements required for a profile of its type. For more information, see “Validating That a Profile Meets the Base Content Requirements” (page 445).
- `kCMMMatchBitmap` (9)
  Requests that you match the colors of a source image bitmap to the color gamut of a destination profile. For more information, see “Matching the Colors of a Bitmap” (page 446).
- `kCMMCheckBitmap` (10)
  Requests that you check the colors of a source image bitmap against the color gamut of a destination profile. For more information, see “Checking the Colors of a Bitmap” (page 447).
- `kCMMGetPS2ColorSpace` (11)
  Requests that you obtain or derive the color space data from a source profile and pass the data to a low-level data-transfer function supplied by the calling application or device driver. For more information, see “Obtaining PostScript-Related Data From a Profile” (page 452).
- `kCMMGetPS2ColorRenderingIntent` (12)
  Requests that you obtain the color-rendering intent from the header of a source profile and then pass the data to a low-level data-transfer function supplied by the calling application or device driver. For more information, see “Obtaining PostScript-Related Data From a Profile” (page 452).
- `kCMMGetPS2ColorRendering` (13)
  Requests that you obtain the rendering intent from the source profile’s header, generate the color rendering dictionary (CRD) data from the destination profile, and then pass the data to a low-level data-transfer function supplied by the calling application or device driver. For more information, see “Obtaining PostScript-Related Data From a Profile” (page 452).
- `kCMMGetPS2ColorRenderingVMSize` (17)
  Requests that you obtain or assess the maximum virtual memory (VM) size of the color rendering dictionary (CRD) specified by a destination profile. For more information, see “Obtaining the Size of the Color Rendering Dictionary for PostScript Printers” (page 454).
- `kCMMFlattenProfile` (14)
  Requests that you extract profile data from the profile to be flattened and

pass the profile data to a function supplied by the calling program. For more information, see “Flattening a Profile for Embedding in a Graphics File” (page 455).

Changed in ColorSync 2.5: Starting with ColorSync version 2.5, the ColorSync Manager calls the function provided by the calling program directly, without going through the preferred, or any, CMM. Your CMM only needs to handle this request code for versions of ColorSync prior to version 2.5.

- `kCMMUnflattenProfile` (`15`)
  Requests that you create a file in the temporary items folder in which to store profile data you receive from a function. The calling program supplies the function. You call this function to obtain the profile data. For more information, see “Unflattening a Profile” (page 456).

  Changed in ColorSync 2.5: Starting with ColorSync version 2.5, the ColorSync Manager calls the function provided by the calling program directly, without going through the preferred, or any, CMM. Your CMM only needs to handle this request code for versions of ColorSync prior to version 2.5.

- `kCMMNewLinkProfile` (`16`)
  Requests that you create a single device link profile that includes the profiles passed to you in an array. For more information, see “Creating a Device Link Profile and Opening a Reference to It” (page 451).

- `kCMMGetNamedColorInfo (70)`
  Requests that you extract and return named color data from the passed profile reference.

- `kCMMGetNamedColorValue (71)`
  Requests that you extract and return device and profile connection space (PCS) color values for the specified color name from the passed profile reference.

- `kCMMGetIndNamedColorValue (72)`
  Requests that you extract and return device and PCS color values for the specified named color index from the passed profile reference.

- `kCMMGetNamedColorIndex (73)`
  Requests that you extract and return a named color index for the specified color name from the passed profile reference.

- `kCMMGetNamedColorName (74)`
  Requests that you extract and return a named color name for the specified named color index from the passed profile reference.

Handling Request Codes

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

Your entry point routine can call a separate subroutine to handle each type of request. "ColorSync Reference for Color Management Modules" (page 467) describes the prototypes for functions your CMM must supply to handle the corresponding ColorSync Manager request codes. The entry routine itself can unpack the parameters from the `params` parameter to pass to its subroutines, or it can call the Component Manager's `CallComponentFunctionWithStorage` routine or `CallComponentFunction` routine to perform these services.

The `CallComponentFunctionWithStorage` function is useful if your CMM uses private storage. When you call this function, you pass it a handle to the storage for this component instance, the `ComponentParameters` data structure, and the address of your subroutine handler. Each time it calls your entry point function, the Component Manager passes to your function the storage handle along with the `ComponentParameters` data structure. For a description of how you associate private storage with a component instance, see "Establishing the Environment for a New Component Instance" (page 440). The Component Manager's `CallComponentFunctionWithStorage` function extracts the calling application's parameters from the `ComponentParameters` data structure and invokes your function, passing to it the extracted parameters and the private storage handle.

For sample code that illustrates how to respond to the required Component Manager and ColorSync Manager requests, see the DemoCMM project available with the ColorSync SDK. You may also wish to refer to the Apple technical note QT05, "Component Manager Version 3.0." This technical note shows how to create a fat component, which is a single component usable for both 68K-based and PowerPC-based systems.

For more information describing how your CMM component should respond to request code calls from the Component Manager, see "Creating Components" in *Inside Macintosh: More Macintosh Toolbox*.

Responding to Required Component Manager Request Codes

This section describes some of the processes your CMM can perform in response to the following Component Manager requests that it must handle:

- "Establishing the Environment for a New Component Instance" describes how to handle a `kComponentOpenSelect` request.
- "Releasing Private Storage and Closing the Component Instance" describes how to handle a `kComponentCloseSelect` request.
- "Determining Whether Your CMM Supports a Request" (page 441) describes how to handle a `kComponentCanDoSelect` request.
- "Providing Your CMM Version Number" describes how to handle a `kComponentVersionSelect` request.

Establishing the Environment for a New Component Instance

When a ColorSync-supportive application or device driver first calls a function that requires the services of your CMM, the Component Manager calls your CMM with a `kComponentOpenSelect` request to open and establish an instance of your component for the calling program. The component instance defines a unique connection between the calling program and your CMM.

In response to this request, you should allocate memory for any private data you require for the connection. You should allocate memory from the current heap zone. 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.

For more information on how to respond to this request and open connections to other components, see "Creating Components" in *Inside Macintosh: More Macintosh Toolbox*.

Releasing Private Storage and Closing the Component Instance

To call your CMM with a close request, the Component Manager sets the `what` field of the `ComponentParameters` data structure to `kComponentCloseSelect`. In response to this request code, your CMM should dispose of the storage memory associated with the connection.

Determining Whether Your CMM Supports a Request

Before the ColorSync Manager calls your CMM with a request code on behalf of a ColorSync-supportive application or driver that called the corresponding function, the Component Manager calls your CMM with a can do request to determine if your CMM implements support for the request.

To call your CMM with a can do request, the Component Manager sets the `what` field of the `ComponentParameters` data structure to the value `kComponentCanDoSelect`. In response, you should set your CMM entry point function's result to 1 if your CMM supports the request and 0 if it doesn't.

Providing Your CMM Version Number

To call your CMM requesting its version number, the Component Manager sets the `what` field of the `ComponentParameters` data structure to the value `kComponentVersionSelect`. In response, you should set your CMM entry point function's result to the CMM version number. Use the high-order 16 bits to represent the major version and the low-order 16 bits to represent the minor version. The major version should represent the component specification level; the minor version should represent your implementation's version number.

If your CMM supports the ColorSync Manager version 2.x, your CMM should return the constant for the major version defined by the following enumeration when the Component Manager calls your CMM with the `kComponentVersionSelect` request code:

```
enum {
    CMMInterfaceVersion = 1
    };
```

Note that if you provide your version number in an extended component resource, the Component Manager can obtain the version number without having to call your code that handles this request code.

Responding to Required ColorSync Manager Request Codes

This section describes some of the processes your CMM can perform in response to the following ColorSync Manager requests that it must handle:

- "Initializing the Current Component Instance for a Two-Profile Session" describes how to handle the `kNCMMInit` request.

- "Matching a List of Colors to the Destination Profile's Color Space" describes how to handle a `kCMMMatchColors` request.
- "Checking a List of Colors" describes how to handle a `kCMMCheckColors` request.

Initializing the Current Component Instance for a Two-Profile Session

The Component Manager calls your CMM with an initialization request, setting the `what` field of the `ComponentParameters` data structure to `kNCMMInit`. In most cases the Component Manager calls your CMM with an initialization request before it calls your CMM with any other ColorSync Manager requests.

In response to this request, your CMM should call its `NCMInit` initialization subroutine. For a description of the function prototype your initialization subroutine must adhere to, see `NCMInit` (page 468).

Using the private storage you allocated in response to the open request, your initialization subroutine should instantiate any private data it needs for the component instance. Before your entry point function returns a function result to the Component Manager, your subroutine should store any profile information it requires. In addition to the standard profile information, you should store the profile header's quality flags setting, the profile size, and the rendering intent. After you return control to the Component Manager, you cannot use the profile references again.

The `kNCMMInit` request gives you the opportunity to examine the profile contents before storing them. If you do not support some aspect of the profile, then you should return an unimplemented error in response to this request. For example, if your CMM does not implement multichannel color support, you should return an "unimplemented" error at this point.

The Component Manager may call your CMM with the `kNCMMInit` request code multiple times after it calls your CMM with a request to open the CMM. For example, it may call your CMM with an initialization request once with one pair of profiles and then again with another pair of profiles. For each call, you need to reinitialize the storage based on the content of the current profiles.

Your CMM should support all seven classes of profiles defined by the ICC. For the constants used to specify the seven classes of profiles, see "Profile Class" (page 396).

Matching a List of Colors to the Destination Profile's Color Space

When a ColorSync-supportive application or device driver calls the `CWMatchColors` function for your CMM to handle, the Component Manager calls your CMM with a color-matching session request, setting the `what` field of the `ComponentParameters` data structure to `kCMMMatchColors` and passing you a list of colors to match. The Component Manager may also call your CMM with this request code to handle other cases, for example, when a ColorSync-supportive program calls the `CWMatchPixMap` function.

Before it calls your CMM with this request, the Component Manager calls your CMM with one of the initialization requests—`kCMMInit`, `kNCMMInit`, or `kCMMConcatInit`—passing to your CMM in the `params` field of the `ComponentParameters` data structure the profiles for the color-matching session.

In response to the `kCMMMatchColors` request, your CMM should call its `CMMatchColors` subroutine by calling the Component Manager's `CallComponentFunctionWithStorage` function and passing it a handle to the storage for this component instance, the `ComponentParameters` data structure, and the address of your `CMMatchColors` subroutine. For a description of the function prototype to which your subroutine must adhere, see `CMMatchColors` (page 470).

The parameters passed to your CMM for this request include an array of type `CMColor` containing the list of colors to match and a one-based count of the number of colors in the list.

To handle this request, your CMM must match the source colors in the list to the color gamut of the destination profile, replacing the color value specifications in the `myColors` array with the matched colors specified in the destination profile's data color space. You should use the rendering intent and the quality flag setting of the source profile in matching the colors. For a description of the color list array data structure, see `CMColor` (page 378).

Checking a List of Colors

When a ColorSync-supportive application or device driver calls the `CWCheckColors` function for your CMM to handle, the Component Manager calls your CMM with a color gamut-checking session request, setting the `what` field of the `ComponentParameters` data structure to `kCMMCheckColors` and passing you a list of colors to check.

Before the Component Manager calls your CMM with the `kCMMCheckColors` request, it calls your CMM with one of the initialization requests—`kCMMInit`,

kNCMMInit, or kCMMConcatInit—passing to your CMM in the params field of the ComponentParameters data structure the profiles for the color gamut-checking session.

In response to the kCMMCheckColors request, your CMM should call its CMCheckColors subroutine. For example, if you use the Component Manager's CallComponentFunctionWithStorage function, you pass it a handle to the storage for this component instance, the ComponentParameters data structure, and the address of your CMCheckColors subroutine. For a description of the function prototype to which your subroutine must adhere, see CMCheckColors (page 472).

In addition to the handle to the private storage containing the profile data, the CallComponentFunctionWithStorage function passes to your CMCheckColors subroutine an array of type CMColor containing the list of colors to gamut check, a one-based count of the number of colors in the list, and an array of longs.

To handle this request, your CMM should test the given list of colors against the gamut specified by the destination profile to determine whether the colors fall within a destination device's color gamut. For each source color in the list that is out of gamut, you must set the corresponding bit in the result array to 1.

Responding to ColorSync Manager Optional Request Codes

This section describes some of the processes your CMM can perform in response to the optional ColorSync Manager requests if your CMM supports them. Before the Component Manager calls your CMM with any of these requests, it first calls your CMM with a can do request to determine if you support the specific optional request code. This section includes the following:

- "Validating That a Profile Meets the Base Content Requirements" (page 445) describes how to handle a kCMMValidateProfile request.
- "Matching the Colors of a Bitmap" (page 446) describes how to handle a kCMMMatchBitmap request.
- "Checking the Colors of a Bitmap" (page 447) describes how to handle a kCMMCheckBitmap request.
- "Matching the Colors of a Pixel Map Image" (page 448) describes how to handle the kCMMMatchPixMap request.
- "Checking the Colors of a Pixel Map Image" (page 449) describes how to handle the kCMMCheckPixMap request.

- "Initializing the Component Instance for a Session Using Concatenated Profiles" (page 450) describes how to handle a `kCMMConcatInit` request.
- "Creating a Device Link Profile and Opening a Reference to It" (page 451) describes how to handle a `kCMMNewLinkProfile` request.
- "Obtaining PostScript-Related Data From a Profile" (page 452) describes how to handle the `kCMMGetPS2ColorSpace`, `kCMMGetPS2ColorRenderingIntent`, and `kCMMGetPS2ColorRendering` requests.
- "Obtaining the Size of the Color Rendering Dictionary for PostScript Printers" (page 454) describes how to handle a `kCMMGetPS2ColorRenderingVMSize` request.
- "Flattening a Profile for Embedding in a Graphics File" (page 455) describes how to handle a `kCMMFlattenProfile` request.
- "Unflattening a Profile" (page 456) describes how to handle a `kCMMUnflattenProfile` request.
- "Supplying Named Color Space Information" (page 457) describes how to handle the `kCMMGetNamedColorInfo`, `kCMMGetNamedColorValue`, `kCMMGetIndNamedColorValue`, `kCMMGetNamedColorIndex`, and `kCMMGetNamedColorName` requests.

Validating That a Profile Meets the Base Content Requirements

When a ColorSync-supportive application or device-driver calls the `CMValidateProfile` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the `ComponentParameters` data structure set to `kCMMValidateProfile` if your CMM supports the request.

In response to this request code, your CMM should call its `CMMValidateProfile` subroutine. One way to do this, for example, is by calling the Component Manager's `CallComponentFunction` function, passing it the `ComponentParameters` data structure and the address of your `CMMValidateProfile` subroutine. To handle this request, you don't need private storage for ColorSync profile information, because the profile reference is passed to your function. However, if your CMM uses private storage for other purposes, you should call the Component Manager's `CallComponentFunctionWithStorage` function. For a description of the function prototype to which your subroutine must adhere, see `CMMValidateProfile` (page 476).

The `CallComponentFunction` function passes to your `CMMValidateProfile` subroutine a reference to the profile whose contents you must check and a flag whose value you must set to report the results.

To handle this request, your CMM should test the profile contents against the baseline profile elements requirements for a profile of this type as specified by the International Color Consortium. It should determine if the profile contains the minimum set of elements required for its type and set the response flag to `true` if the profile contains the required elements and `false` if it doesn't.

To obtain a copy of the *International Color Consortium Profile Format Specification*, version 2.x, visit the ICC Web site at <http://www.color.org/>.

The ICC also defines optional tags, which may be included in a profile. Your CMM might use these optional elements to optimize or improve its processing. Additionally, a profile might include private tags defined to provide your CMM with processing capability it uses. The profile developer can define these private tags, register the tag signatures with the ICC, and include the tags in a profile.

If your CMM is dependent on optional or private tags, your `CMMValidateProfile` function should check for the existence of these tags also.

Instead of itself checking the profile for the minimum profile elements requirements for the profile class, your `CMMValidateProfile` function may use the Component Manager functions to call the default CMM and have it perform the minimum defaults requirements validation.

To call the default CMM when responding to a `kCMMValidateProfile` request from an application, your CMM can use the standard mechanisms applications use to call a component. For information on these mechanisms, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.

Matching the Colors of a Bitmap

When a ColorSync-supportive application or device driver calls the `CWMatchBitMap` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the `ComponentParameters` data structure set to `kCMMMatchBitmap` if your CMM supports the request. If your CMM supports this request code, your CMM should be prepared to receive any of the bitmap types defined by the ColorSync Manager.

In response to this request code, your CMM should call its `CMMatchBitmap` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage

handle for this component instance, the `ComponentParameters` data structure, and the address of your `CMMatchBitmap` subroutine. For a description of the function prototype to which your subroutine must adhere, see `CMMatchBitmap` (page 477).

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `CMMatchBitmap` subroutine a pointer to the bitmap containing the source image data whose colors your function must match, a pointer to a callback function supplied by the calling program, a reference constant your subroutine must pass to the callback function when you invoke it, and a pointer to a bitmap in which your function stores the resulting color-matched image.

The callback function supplied by the calling function monitors the color-matching progress as your function matches the bitmap colors. You should call this function at regular intervals. Your `CMMatchBitmap` function should monitor the progress function for a returned value of `true`, which indicates that the user interrupted the color-matching process. In this case, you should terminate the color-matching process.

To handle this request, your `CMMatchBitmap` function must match the colors of the source image bitmap to the color gamut of the destination profile using the profiles specified by a previous `kNCMInit`, `kCMMInit`, or `kCMMConcatInit` request to your CMM for this component instance. You must store the color-matched image in the bitmap result parameter passed to your subroutine. If you are passed a `NULL` parameter, you must match the bitmap in place.

For a description of the prototype of the callback function supplied by the calling program, see `MyCMBitmapCallBackProc` (page 345).

Checking the Colors of a Bitmap

When a ColorSync-supportive application or device driver calls the `CWCheckBitMap` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the `ComponentParameters` data structure set to `kCMMCheckBitmap` if your CMM supports the request. If your CMM supports this request code, your CMM should be prepared to receive any of the bitmap types defined by the ColorSync Manager.

In response to this request code, your CMM should call its `CMCheckBitmap` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the `ComponentParameters` data structure,

and the address of your `CMCheckBitmap` subroutine. For a description of the function prototype to which your subroutine must adhere, see `MyCMBitmapCallBackProc` (page 345).

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `CMCheckBitmap` subroutine a pointer to the bitmap containing the source image data whose colors your function must check, a pointer to a callback progress-reporting function supplied by the calling program, a reference constant your subroutine must pass to the callback function when you invoke it, and a pointer to a resulting bitmap whose pixels your subroutine must set to show if the corresponding source color is in or out of gamut. A black pixel (value 1) in the returned bitmap indicates an out-of-gamut color, while a white pixel (value 0) indicates the color is in gamut.

The callback function supplied by the calling function monitors the color gamut-checking progress. You should call this function at regular intervals. Your `CMCheckBitmap` function should monitor the progress function for a returned value of `true`, which indicates that the user interrupted the color gamut-checking process. In this case, you should terminate the process.

For a description of the prototype of the callback function supplied by the calling program, see `MyCMBitmapCallBackProc` (page 345).

Using the content of the profiles that you stored at initialization time for this component instance, your `CMCheckBitmap` subroutine must check the colors of the source image bitmap against the color gamut of the destination profile. If a pixel is out of gamut, your function must set the corresponding pixel in the result image bitmap to 1. The ColorSync Manager returns the resulting bitmap to the calling application or driver to report the outcome of the check.

For complete details on the `CMCheckBitmap` subroutine parameters and how your `CMCheckBitmap` subroutine communicates with the callback function, see `MyCMBitmapCallBackProc` (page 345).

Matching the Colors of a Pixel Map Image

When a ColorSync-supportive application or device driver calls the `CWMatchPixMap` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the `ComponentParameters` data structure set to `kCMMMatchPixMap` if your CMM supports the request. If your CMM supports this request code, your `CMMatchPixMap` function should be prepared to receive any of the pixel map types defined by QuickDraw.

In response to this request code, your CMM should call its `CMMatchPixMap` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the `ComponentParameters` data structure, and the address of your `CMMatchPixMap` subroutine. For a description of the function prototype to which your subroutine must adhere, see `CMMatchPixMap` (page 486).

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `CMMatchPixMap` subroutine a pointer to the pixel map containing the source image to match, a pointer to a callback progress-reporting function supplied by the calling program, and a reference constant your subroutine must pass to the callback function when you invoke it.

To handle this request, your `CMMatchPixMap` subroutine must match the colors of the source pixel map image to the color gamut of the destination profile, replacing the original pixel colors of the source image with their corresponding colors expressed in the data color space of the destination profile. The ColorSync Manager returns the resulting color-matched pixel map to the calling application or driver.

The callback function supplied by the calling function monitors the color-matching progress. You should call this function at regular intervals. Your `CMMatchPixMap` function should monitor the progress function for a returned value of `true`, which indicates that the user interrupted the color-matching process. In this case, you should terminate the process.

For a description of the prototype of the callback function supplied by the calling program, see `MyCMBitmapCallBackProc` (page 345).

Checking the Colors of a Pixel Map Image

When a ColorSync-supportive application or device-driver calls the `CWCheckPixMap` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the `ComponentParameters` data structure set to `kCMMCheckPixMap` if your CMM supports the request.

In response to this request code, your CMM should call its `CMCheckPixMap` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the `ComponentParameters` data structure, and the address of your `CMCheckPixMap` subroutine. For a description of the

function prototype to which your subroutine must adhere, see `CMCheckPixMap` (page 488).

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `CMCheckPixMap` subroutine a pointer to the pixel map containing the source image to check, a QuickDraw bitmap in which to report the color gamut-checking results, a pointer to a callback progress-reporting function supplied by the calling program, and a reference constant your subroutine must pass to the callback function when you invoke it.

Using the content of the profiles passed to you at initialization time, your `CMCheckPixMap` subroutine must check the colors of the source pixel map image against the color gamut of the destination profile to determine if the pixel colors are within the gamut. If a pixel is out of gamut, your subroutine must set to 1 the corresponding pixel of the result bitmap. The ColorSync Manager returns the bitmap showing the color gamut-checking results to the calling application or device driver.

Initializing the Component Instance for a Session Using Concatenated Profiles

When a ColorSync-supportive application or device driver calls the `CWConcatColorWorld` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the `ComponentParameters` data structure set to `kCMMConcatInit` if your CMM supports the request.

In response to this request code, your CMM should call its `CMConcatInit` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the `ComponentParameters` data structure, and the address of your `CMConcatInit` subroutine. For a description of the function prototype to which your subroutine must adhere, see `CMConcatInit` (page 483).

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `CMConcatInit` subroutine a pointer to a data structure of type `CMConcatProfileSet` containing an array of profiles to use in a subsequent color-matching or color gamut-checking session. The profiles in the array are in processing order—source through destination. The `profileSet` field of the data structure contains the array. If the profile array contains only one profile, that

profile is a device link profile. For a description of the `CMConcatProfileSet` data structure, see `CMConcatProfileSet` (page 384).

Using the storage passed to your entry point function in the `CMSession` parameter, your `CMConcatInit` function should initialize any private data your CMM will need for a subsequent color session involving the set of profiles. Before your function returns control to the Component Manager, your subroutine should store any profile information it requires. In addition to the standard profile information, you should store the profile header's quality flags setting, the profile size, and the rendering intent. After you return control to the Component Manager, you cannot use the profile references again.

A color-matching or color gamut-checking session for a set of profiles entails various color transformations among devices in a sequence for which your CMM is responsible. Your CMM may use Component Manager functions to call other CMMs if necessary.

There are special guidelines your CMM must follow in using a set of concatenated profiles for subsequent color-matching or gamut-checking sessions. These guidelines are described in `CMConcatInit` (page 483).

Creating a Device Link Profile and Opening a Reference to It

When a ColorSync-supportive application or device driver calls the `CWNewLinkProfile` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the `ComponentParameters` data structure set to `kCMMNewLinkProfile` if your CMM supports the request.

In response to this request code, your CMM should call its `CMNewLinkProfile` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the `ComponentParameters` data structure, and the address of your `CMNewLinkProfile` subroutine. For a description of the function prototype to which your subroutine must adhere, see `CMNewLinkProfile` (page 491).

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `CMNewLinkProfile` subroutine a pointer to a data structure of type `CMConcatProfileSet` containing the array of profiles that will make up the device link profile.

To handle this request, your subroutine must create a single device link profile of type `DeviceLink` that includes the profiles passed to you in the array pointed

to by the `profileSet` parameter. Your CMM must create a file specification for the device link profile. A device link profile cannot be a temporary profile: that is, you cannot specify a location type of `cmNoProfileBase` for a device link profile. For information on how to specify the file location, see "Profile Location Type" (page 393).

The profiles in the array are in the processing order—source through destination—which you must preserve. After your CMM creates the device link profile, it must open a reference to the profile and return the profile reference along with the location specification.

Obtaining PostScript-Related Data From a Profile

There are three very similar PostScript-related request codes that your CMM may support. Each of these codes requests that your CMM obtain or derive information required by a PostScript printer from the specified profile and pass that information to a function supplied by the calling program.

When a ColorSync-supportive application or device driver calls the high-level function corresponding to the request code and your CMM is specified to handle it, the Component Manager calls your CMM with the `what` field of the `ComponentParameters` data structure set to the corresponding request code if your CMM supports it. Here are the three high-level functions and their corresponding request codes:

- When the application or device driver calls the `CMGetPS2ColorSpace` function, the Component Manager calls your CMM with a `kCMMGetPS2ColorSpace` request code. To respond to this request, your CMM must obtain the color space data from a source profile and pass the data to a low-level data-transfer function supplied by the calling application or device driver.
- When the application or device driver calls the `CMGetPS2ColorRenderingIntent` function, the Component Manager calls your CMM with a `kCMMGetPS2ColorRenderingIntent` request code. To respond to this request, your CMM must obtain the color rendering intent from the source profile and pass the data to a low-level data-transfer function supplied by the calling application or device driver.
- When the application or device driver calls the `CMGetPS2ColorRendering` function, the Component Manager calls your CMM with a `kCMMGetPS2ColorRendering` request code. To respond to this request, your CMM must obtain the rendering intent from the source profile's header. Then your CMM must obtain or derive the color rendering dictionary for that

rendering intent from the destination profile and pass the CRD data to a low-level data-transfer function supplied by the calling application or device driver.

In response to each of these request codes, your CMM should call its subroutine that handles the request. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the `ComponentParameters` data structure, and the address of your subroutine handler.

For a description of the function prototypes to which your subroutine must adhere for each of these requests, see "ColorSync Reference for Color Management Modules" (page 467).

- For `kCMMGetPS2ColorSpace`, see `CMMGetPS2ColorSpace` (page 493)
- For `kCMMGetPS2ColorRenderingIntent`, see `CMMGetPS2ColorRenderingIntent` (page 495)
- For `kCMMGetPS2ColorRendering`, see `CMMGetPS2ColorRendering` (page 497).

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your subroutine a reference to the source profile containing the data you must obtain or derive, a pointer to the function supplied by the calling program, and a reference constant that you must pass to the supplied function each time your CMM calls it. For `kCMMGetPS2ColorRendering`, your CMM is also passed a reference to the destination profile.

To handle each of these requests, your subroutine must allocate a `data` buffer in which to pass the particular PostScript-related data to the function supplied by the calling application or driver. Your subroutine must call the supplied function repeatedly until you have passed all the data to it. For a description of the prototype of the application or driver-supplied function, see `MyColorSyncDataTransfer` (page 342).

For a description of how each of your subroutines must interact with the calling program's supplied function, see the descriptions of the prototypes for the subroutines in "Application-Defined Functions for the ColorSync Manager" (page 340).

Obtaining the Size of the Color Rendering Dictionary for PostScript Printers

When a ColorSync-supportive application or device driver calls the `CMGetPS2ColorRenderingVMSize` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the `ComponentParameters` data structure set to `kCMMGetPS2ColorRenderingVMSize` if your CMM supports the request.

In response to this request code, your CMM should call its `CMMGetPS2ColorRenderingVMSize` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the `ComponentParameters` data structure, and the address of your `CMMGetPS2ColorRenderingVMSize` subroutine. For a description of the function prototype to which your subroutine must adhere, see "ColorSync Reference for Color Management Modules" (page 467).

In addition to the storage handle for global data for this component instance, the `CallComponentFunctionWithStorage` function passes to your `CMMGetPS2ColorRenderingVMSize` subroutine a reference to the source profile identifying the rendering intent and a reference to the destination profile containing the color rendering dictionary (CRD) for the specified rendering intent.

To handle this request, your CMM must obtain or assess and return the maximum VM size for the CRD of the specified rendering intent.

If the destination profile contains the Apple-defined private tag `'psvm'`, described in the next paragraph, then your CMM may read the tag and return the CRD VM size data supplied by this tag for the specified rendering intent. If the destination profile does not contain this tag, then you must assess the VM size of the CRD.

The `CMPS2CRDVMSizeType` data type defines the Apple-defined `'psvm'` optional tag that a printer profile may contain to identify the maximum VM size of a CRD for different rendering intents.

This tag's element data includes an array containing one entry for each rendering intent and its virtual memory size. For a description of the data structures that define the tag's element data, see "Color Rendering Dictionary Virtual Memory Size" (page 390).

Flattening a Profile for Embedding in a Graphics File

Flattening refers to transferring a profile stored in an independent disk file to an external profile format that can be embedded in a graphics document. Unflattening refers to transferring from the embedded format to an independent disk file.

Starting With ColorSync 2.5

Starting with ColorSync version 2.5, when a ColorSync-supportive application or device driver calls the `CMFlattenProfile` function, the ColorSync Manager calls the flatten function provided by the calling program or driver directly, without going through the preferred, or any, CMM.

Prior to ColorSync 2.5

Prior to ColorSync version 2.5, when a ColorSync-supportive application or device driver calls the `CMFlattenProfile` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the `ComponentParameters` data structure set to `kCMMFlattenProfile` if your CMM supports the request.

In response to this request code, your CMM should call its `CMMFlattenProfile` subroutine. For example, to do this, your CMM may call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the `ComponentParameters` data structure, and the address of your `CMMFlattenProfile` subroutine. For a description of the function prototype to which your subroutine must adhere, see `CMMFlattenProfile` (page 503).

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `CMMFlattenProfile` subroutine a reference to the profile to be flattened, a pointer to a function supplied by the calling program, and a reference constant your subroutine must pass to the calling program's function when you invoke it.

To handle this request, your subroutine must extract the profile data from the profile, allocate a buffer in which to pass the profile data to the supplied function, and pass the profile data to the function, keeping track of the amount of data remaining to pass.

For a description of the prototype of the function supplied by the calling program, see `MyColorSyncDataTransfer` (page 342). See also `CMMFlattenProfile`

(page 503) for details on how your `CMMFlattenProfile` subroutine communicates with the function supplied by the calling program.

Unflattening a Profile

Unflattening refers to transferring from the embedded format to an independent disk file. Flattening refers to transferring a profile stored in an independent disk file to an external profile format that can be embedded in a graphics document.

Starting With ColorSync 2.5

Starting with ColorSync version 2.5, when a ColorSync-supportive application or device driver calls the `CMUnflattenProfile` function, the ColorSync Manager calls the unflatten function provided by the calling program or driver directly, without going through the preferred, or any, CMM.

Prior to ColorSync 2.5

Prior to ColorSync version 2.5, when a ColorSync-supportive application or device driver calls the `CMUnflattenProfile` function, the Component Manager calls your CMM with the `what` field of the `ComponentParameters` data structure set to `kCMMUnflattenProfile`, if your CMM supports that request code.

In response to the `kCMMUnflattenProfile` request code, your CMM should call its `CMMUnflattenProfile` function. To do this, your CMM can call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the `ComponentParameters` data structure, and the address of your `CMMUnflattenProfile` function. For more information, see `CMMUnflattenProfile` (page 505).

In addition to the storage handle for private storage for this component instance, the `CallComponentFunctionWithStorage` function passes to your `CMMUnflattenProfile` function a pointer to a function supplied by the calling program and a reference constant. Your function passes the reference constant to the calling program's function when you invoke it. The calling program's function obtains the profile data and returns it to your subroutine. For a more information on the data transfer function, see `MyColorSyncDataTransfer` (page 342).

To handle this request, your subroutine must create a file in which to store the profile data. You should create the file in the temporary items folder. Your `CMMUnflattenProfile` subroutine must call the supplied `ColorSyncDataTransfer`

function repeatedly to obtain the profile data. Before calling the `ColorSyncDataTransfer` function, your `CMMUnflattenProfile` function must allocate a buffer to hold the returned profile data.

Your `CMMUnflattenProfile` function must identify the profile size and maintain a counter to track the amount of data transferred and the amount of data remaining. This information allows you to determine when to call the `ColorSyncDataTransfer` function for the final time.

Supplying Named Color Space Information

When a ColorSync-supportive application or device driver calls the `CMGetNamedColorInfo` function for your CMM to handle, the Component Manager calls your CMM with the `what` field of the `ComponentParameters` data structure set to `kCMMGetNamedColorInfo` if your CMM supports the request.

In response to this request code, your CMM should call its `CMMGetNamedColorInfo` subroutine. To do this, your CMM might call the Component Manager's `CallComponentFunctionWithStorage` function, passing it the storage handle for this component instance, the `ComponentParameters` data structure, and the address of your `CMMGetNamedColorInfo` subroutine.

The `CMMGetNamedColorInfo` function returns information about a named color space from its profile reference. For a description of the function prototype to which your subroutine must adhere, see `CMMGetNamedColorInfo` (page 508).

A named color profile has a value of `'nmcl'` in the Profile/Device class field of its header. If the source profile passed to your `CMMGetNamedColorInfo` subroutine is a named color profile, you can extract the necessary information to return in the parameters of the `CMMGetNamedColorInfo` routine.

Your CMM can obtain named color information as well as profile header information by reading the `namedColor2Tag` tag (signature `'ncl2'`). This tag's element data includes a count of named colors, the number of device channels, and a prefix and suffix for each named color name. The data also includes the named color names themselves, along with profile connection space (PCS) and device color information for each named color. For information on the format of the `namedColor2Tag` tag, see the *International Color Consortium Profile Format Specification*.

Your CMM responds similarly for other named color requests:

- The `CMGetNamedColorValue` (page 258) routine generates a `kCMMGetNamedColorValue` request, which you respond to in your

CMMGetNamedColorValue routine. The CMMGetNamedColorValue routine returns device and PCS color values from a named color space profile for a specific color name.

- The CMGetIndNamedColorValue (page 259) routine generates a kCMMGetIndNamedColorValue request, which you respond to in your CMMGetIndNamedColorValue routine. The CMMGetIndNamedColorValue routine returns device and PCS color values from a named color space profile for a specific named color index.
- The CMGetNamedColorIndex (page 260) routine generates a kCMMGetNamedColorIndex request, which you respond to in your CMMGetNamedColorIndex routine. The CMMGetNamedColorIndex routine returns a named color index from a named color space profile for a specific color name.
- The CMGetNamedColorName (page 260) routine generates a kCMMGetNamedColorName request, which you respond to in your CMMGetNamedColorName routine. The CMMGetNamedColorName routine returns a named color name from a named color space profile for a specific named color index.

Summary of the Color Management Modules

Functions

Required Functions

```
pascal CMError NCMInit (          ComponentInstance CMSession,
                                   CMProfileRef srcProfile,
                                   CMProfileRef dstProfile);

pascal CMError CMMatchColors(     ComponentInstance CMSession,
                                   CMColor *myColors,
                                   unsigned long count);

pascal CMError CMCheckColors(     ComponentInstance CMSession,
                                   CMColor *myColors,
                                   unsigned long count,
                                   long *result);

pascal CMError CMInit(ComponentInstance CMSession,
                                   CMProfileHandle srcProfile,
                                   CMProfileHandle dstProfile)
```

Optional Functions

```
pascal CMError CMMValidateProfile (
                                   ComponentInstance CMSession,
                                   CMProfileRef prof,
                                   Boolean *valid);

pascal CMError CMMatchBitmap(     ComponentInstance CMSession,
                                   const CMBitmap *bitmap,
                                   CMBitmapCallBackUPP progressProc,
                                   void *refCon,
                                   CMBitmap *matchedBitmap);
```

```
pascal CMError CMCheckBitmap(     ComponentInstance CMSession,
                                   const CMBitmap *bitmap,
                                   CMBitmapCallBackUPP progressProc,
                                   void *refCon,
                                   CMBitmap *resultBitmap);

pascal CMError CMConcatInit (     ComponentInstance CMSession,
                                   CMConcatProfileSet *profileSet);

pascal CMError CMMatchPixMap(     ComponentInstance CMSession,
                                   PixMap *myPixMap,
                                   CMBitmapCallBackUPP progressProc,
                                   void *refCon);

pascal CMError CMCheckPixMap(     ComponentInstance CMSession,
                                   const PixMap *myPixMap,
                                   CMBitmapCallBackUPP progressProc,
                                   BitMap *myBitMap,
                                   void *refCon);

pascal CMError CMNewLinkProfile (ComponentInstance CMSession,
                                   CMProfileRef *prof,
                                   const CMProfileLocation *targetLocation,
                                   CMConcatProfileSet *profileSet);

pascal CMError CMConcatenateProfiles (
                                   ComponentInstance CMSession,
                                   CMProfileHandle thru,
                                   CMProfileHandle dst,
                                   CMProfileHandle *newDst);

pascal CMError CMMGetPS2ColorSpace (
                                   ComponentInstance CMSession,
                                   CMProfileRef srcProf,
                                   unsigned long flags,
                                   CMFlattenUPP proc,
                                   void *refCon);

pascal CMError CMMGetPS2ColorRenderingIntent (
                                   ComponentInstance CMSession,
                                   CMProfileRef srcProf,
                                   unsigned long flags,
                                   CMFlattenUPP proc,
                                   void *refCon);
```

```
pascal CMError CMMGetPS2ColorRendering (
                                    ComponentInstance CMSession,
                                    CMProfileRef srcProf,
                                    CMProfileRef dstProf,
                                    unsigned long flags,
                                    CMFlattenUPP proc,
                                    void *refCon);

pascal CMError CMMGetPS2ColorRenderingVMSize (
                                    ComponentInstance CMSession,
                                    CMProfileRef srcProf,
                                    CMProfileRef dstProf,
                                    unsigned long vmSize);

pascal CMError CMMFlattenProfile (
                                    ComponentInstance CMSession,
                                    CMProfileRef prof,
                                    unsigned long flags,
                                    CMFlattenUPP proc,
                                    void *refCon);

pascal CMError CMMUnflattenProfile (
                                    ComponentInstance CMSession,
                                    FSSpec *resultFileSpec,
                                    CMFlattenUPP proc,
                                    void *refCon);

pascal CMError CMMGetNamedColorInfo(
                                    ComponentInstance CMSession,
                                    CMProfileRef srcProf,
                                    unsigned long *deviceChannels,
                                    OSType *deviceColorSpace,
                                    OSType *PCSColorSpace,
                                    unsigned long *count,
                                    StringPtr prefix,
                                    StringPtr suffix);

pascal CMError CMMGetNamedColorValue(
                                    ComponentInstance CMSession,
                                    CMProfileRef prof,
                                    StringPtr name,
                                    CMColor *deviceColor,
                                    CMColor *PCSColor);
```

```
pascal CMError CMMGetIndNamedColorValue(
                                    ComponentInstance CMSession,
                                    CMProfileRef prof,
                                    unsigned long index,
                                    CMColor *deviceColor,
                                    CMColor *PCSColor);

pascal CMError CMMGetNamedColorIndex(
                                    ComponentInstance CMSession,
                                    CMProfileRef prof,
                                    StringPtr name,
                                    unsigned long *index);

pascal CMError CMMGetNamedColorName(
                                    ComponentInstance CMSession,
                                    CMProfileRef prof,
                                    unsigned long index,
                                    StringPtr name);
```

Constants

```
enum {
    CMMInterfaceVersion = 1
    };

/* request codes (required) */
enum {
    kCMMInit            = 0,
    kCMMMatchColors     = 1,
    kCMMCheckColors     = 2
    kNCMMInit           = 6,
    };

/* request codes (optional) */
enum {
    kCMMMatchPixMap                 = 3,
    kCMMCheckPixMap                 = 4,
    kCMMConcatenateProfiles         = 5,    /* For backward compatibility
                                               with ColorSync 1.0 only. */
    kCMMConcatInit                  = 7,
    kCMMValidateProfile             = 8,
    kCMMMatchBitmap                 = 9,
    kCMMCheckBitmap                 = 10,
```

```
kCMMGetPS2ColorSpace            = 11,
kCMMGetPS2ColorRenderingIntent  = 12,
kCMMGetPS2ColorRendering        = 13,
kCMMFlattenProfile              = 14,
kCMMUnflattenProfile            = 15,
kCMMNewLinkProfile              = 16,
kCMMGetPS2ColorRenderingVMSize  = 17,
kCMMGetNamedColorInfo           = 70,
kCMMGetNamedColorValue          = 71,
kCMMGetIndNamedColorValue       = 72,
kCMMGetNamedColorIndex          = 73,
kCMMGetNamedColorName           = 74
};
```

ColorSync Reference for Color Management Modules

Contents

This section 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 other codes as well.

This section also describes the functions your CMM may define to respond to ColorSync Manager request codes. For information on how to develop a CMM that responds to ColorSync Manager request codes, see "Developing Color Management Modules" (page 429).

- "Required CMM-Defined Functions" (page 467) describes the functions that your CMM should define to handle ColorSync Manager required request codes.
- "Optional CMM-Defined Functions" (page 474) describes the functions that your CMM should define to handle ColorSync Manager optional request codes.
- "Constants" (page 514) describes the constants for the CMM component interface version and the ColorSync Manager request codes.

"ColorSync Version Information" (page 525) describes the `Gestalt` information, shared library version numbers, CMM version numbers, and ColorSync header files you use with different versions of the ColorSync Manager. It also includes CPU and Mac OS system requirements.

Required CMM-Defined Functions

This section describes the functions that your CMM should define to handle ColorSync Manager required request codes.

- `NCMInit` (page 468) handles the `kNCMMInit` request by performing any required private initialization.
- `CMMatchColors` (page 470) handles the `kCMMMatchColors` request by matching the specified colors to the gamut of the destination profile.
- `CMCheckColors` (page 472) handles the `kCMMCheckColors` request by checking the specified colors against the gamut of the destination profile.

NCMInit

Handles the `kNCMMInit` request by performing any required private initialization.

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, `NCMInit` to handle the request.

The `NCMInit` function is a color management module–defined subroutine.

```
pascal CMError NCMInit (
                    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 use in the color-matching or color-checking session. 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 use in the color-matching or color-checking session. 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.)

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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 `NCMInit` function.

Using the storage pointed to by the `CMSession` handle, your `NCMInit` 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 to maintain a connection for the requesting application or driver. You should allocate memory from the current heap zone. If that attempt fails, you should allocate memory from the system heap or the temporary heap. You can use the `SetComponentInstanceStorage` function to associate the allocated memory with the component instance. 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 seven classes of profiles defined by the ICC. For information on the seven classes of profiles, see “ColorSync Reference for Applications and Drivers” (page 217).

CMMatchColors

Handles the `kCMMMatchColors` request by matching the specified colors to the gamut of the destination profile.

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, `CMMatchColors`) to handle the request by matching colors in the color list.

The `CMMatchColors` function is a color management module–defined subroutine.

```
pascal CMError CMMatchColors (
                    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 a color union array of type `CMColor` (page 378) specified by the calling application or device driver. On input, this array contains the list of colors to match. 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 output, this array contains the list of matched colors specified by your function in the data color space of the destination profile.

`count` A one-based count of the number of colors in the color list of the `CMColor` array.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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 use 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 use 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 link profile specified by the calling application to use for the color-matching session. For information about the `ConcatProfileSet` data type, see "CMConcatProfileSet" (page 384).

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 `CMMatchColors` 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.

For a color-matching session with a named color space profile and other profiles, the named color profile must be first in the color world. A color world of this type cannot be used with bitmap or pixel map functions—it can only be used with a function such as `CMMatchColors`, with the `myColors` color list containing the named color indexes. For more information on the rules governing the types of profiles you can specify in a profile array, see the following:

- `CMHeader` (page 351)
- `NCWNewColorWorld` (page 262)
- `CWConcatColorWorld` (page 265)

- CWNewLinkProfile (page 267)

CMCheckColors

Handles the kCMMCheckColors request by checking the specified colors against the gamut of the destination profile.

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, CMCheckColors) to handle the request.

The CMCheckColors function is a color management module–defined subroutine.

```
pascal CMError CMCheckColors (
                    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 a color union array of type CMColor (page 378), specified by the calling application or device driver, that contains the list of colors to check 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.

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 bit field, 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.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

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 use 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 use 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 link profile specified by the calling program to use 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 `CMCheckColors` 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 these color data types, you should return an unimplemented error in response to the initialization request code. See the functions `NCMInit` (page 468) and `CMConcatInit` (page 483) for more information.

For each color in the list, your `CMCheckColors` 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 CMM-Defined Functions

This section describes the functions that your CMM should define to handle ColorSync Manager optional request codes.

- `CMMValidateProfile` (page 476) handles the `kCMMValidateProfile` request by determining if the specified profile contains the minimum set of elements required for a profile of its type.
- `CMMatchBitmap` (page 477) handles the `kCMMMatchBitmap` request by matching the colors of the source image bitmap to the color gamut of the destination profile.
- `CMCheckBitmap` (page 480) handles the `kCMMCheckBitmap` request by checking the colors of the source image bitmap against the color gamut of the destination profile.
- `CMConcatInit` (page 483) handles the `kCMMConcatInit` request by initializing any private data the CMM will need for a color session involving the specified set of profiles.
- `CMMatchPixMap` (page 486) handles the `kCMMMatchPixMap` request by matching the colors of the specified pixel map image to the destination profile's color gamut.
- `CMCheckPixMap` (page 488) handles the `kCMMCheckPixMap` request by checking the colors of the specified pixel map image against the color gamut of the destination profile.
- `CMNewLinkProfile` (page 491) handles the `kCMMNewLinkProfile` request by creating a single device link profile that includes the profiles in the specified profile set.

- `CMMGetPS2ColorSpace` (page 493) handles the `kCMMGetPS2ColorSpace` request by obtaining or deriving the color space element data from the source profile.
- `CMMGetPS2ColorRenderingIntent` (page 495) handles the `kCMMGetPS2ColorRenderingIntent` request by obtaining the rendering intent from the source profile.
- `CMMGetPS2ColorRendering` (page 497) handles the `kCMMGetPS2ColorRendering` request by obtaining the rendering intent from the header of the source profile.
- `CMMGetPS2ColorRenderingVMSize` (page 500) handles the `kCMMGetPS2ColorRenderingVMSize` request by obtaining the maximum virtual memory (VM) size of the color rendering dictionary (CRD) for the rendering intent specified by the source profile.
- `CMMFlattenProfile` (page 503) handles the `kCMMFlattenProfile` request by extracting profile data from the profile to flatten and passing it to the specified function.
- `CMMUnflattenProfile` (page 505) handles the `kCMMUnflattenProfile` request by creating a uniquely-named file in the temporary items folder to store the profile data.
- `CMMGetNamedColorInfo` (page 508) handles the `kCMMGetNamedColorInfo` request by returning information about a named color space from its profile reference.
- `CMMGetNamedColorValue` (page 510) handles the `kCMMGetNamedColorValue` request by returning device and PCS color values from a named color space profile for a specific color name.
- `CMMGetIndNamedColorValue` (page 511) handles the `kCMMGetIndNamedColorValue` request by returning device and PCS color values from a named color space profile for a specific named color index.
- `CMMGetNamedColorIndex` (page 512) handles the `kCMMGetNamedColorIndex` request by returning a named color index from a named color space profile for a specific color name.
- `CMMGetNamedColorName` (page 513) handles the `kCMMGetNamedColorName` request by returning a named color name from a named color space profile for a specific named color index.

CMMValidateProfile

Handles the `kCMMValidateProfile` request by determining if the specified profile contains the minimum set of elements required for a profile of its type.

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, `CMMValidateProfile`) to handle the request.

The `CMMValidateProfile` function is a color management module–defined subroutine.

```
pascal CMError CMMValidateProfile (
                    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 validate.

`valid` A pointer to 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.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Your `CMMValidateProfile` 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 class, 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 `CMMValidateProfile` 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 class, your `CMMValidateProfile` function may use the Component Manager functions to call ColorSync's default CMM and have it perform the minimum defaults requirements validation. The signature of the default CMM is `'appl'`.

To call the default 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*.

CMMatchBitmap

Handles the `kCMMMatchBitmap` request by matching the colors of the source image bitmap to the color gamut of the destination profile.

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, `CMMatchBitmap`) to handle the request.

The `CMMatchBitmap` function is a color management module–defined subroutine.

```
pascal CMError CMMatchBitmap(
                    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 `CMMatchBitmap` 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 `CMMatchBitmap` function must pass through as a parameter to calls it makes to the `CMBitmapCallBackProc` 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 `CMMatchBitmap` function must match the bitmap colors in place.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

If your CMM supports this request code, your `CMMatchBitmap` function should be prepared to receive any of the bitmap types defined by the ColorSync Manager. Your `CMMatchBitmap` 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 use 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 use 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 link profile specified by the calling program to use for the color-matching session. For information about the ConcatProfileSet data type, see "CMConcatProfileSet" (page 384).

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 CMMatchBitmap 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 CMMatchBitmap 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 CMMatchBitmap 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 default CMM calls the CMBitmapCallBackProc function approximately every half-second, unless color matching takes less time; this happens when there is a small amount of data to match.

Here is the prototype for the CMBitmapCallBackProc function pointed to by the progressProc parameter:

```
pascal Boolean CMBitmapCallBackProc (
                            long progress,
                            void *refCon);
```

Each time your `CMMatchBitmap` function calls the `CMBitmapCallBackProc` 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 `CMBitmapCallBackProc` function, your function must pass it a byte count in the `progress` parameter identifying the remaining number of bytes. The last time your `CMMatchBitmap` function calls the `CMBitmapCallBackProc` 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.

CMCheckBitmap

Handles the `kCMMCheckBitmap` request by checking the colors of the source image bitmap against the color gamut of the destination profile.

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, `CMCheckBitmap`) to handle the request.

The CMCheckBitmap function is a color management module–defined subroutine.

```
pascal CMError CMCheckBitmap(
                    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 CMCheckBitmap 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 CMCheckBitmap function must pass through as a parameter to calls it makes to the CMBitmapCallBackProc function.
resultBitmap	A pointer to the resulting bitmap allocated by the calling application or device driver. Your CMCheckBitmap 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.
function result	A result code of type CMError. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

If your CMM supports this request code, your CMMCheckBitmap function should be prepared to receive any of the bitmap types defined by the ColorSync Manager. Your CMCheckBitmap 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 references to the profiles to use for the color-checking session and sending your CMM a request 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 use 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 use for the session. For information about the ConcatProfileSet data type, see "CMHeader" (page 351).

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 CMCheckBitmap 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 CMCheckBitmap 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 CMCheckBitmap 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 default CMM calls the CMBitmapCallBackProc function approximately every half-second, unless the gamut checking takes less time; this happens when there is a small amount of data to check.

Here is the prototype for the CMBitmapCallBackProc function pointed to by the progressProc parameter:

```
pascal Boolean CMBitmapCallBackProc (
                            long progress,
                            void *refCon);
```

Each time your `CMCheckBitmap` function calls the `CMBitmapCallBackProc` 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 `CMBitmapCallBackProc` function, your function must pass it a byte count in the `progress` parameter identifying the remaining number of bytes to check. The last time your `CMMatchBitmap` function calls the `CMBitmapCallBackProc` 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 `CMMatchColors` 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.

CMConcatInit

Handles the `kCMMConcatInit` request by initializing any private data the CMM will need for a color session involving the specified set of profiles.

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, `CMConcatInit`) to handle the request.

The `CMConcatInit` function is a color management module–defined subroutine.

```
pascal CMError CMConcatInit (
                    ComponentInstance CMSession,
                    CMConcatProfileSet *profileSet);
```

`CMsession` 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 an array of profiles of type `CMConcatProfileSet` (page 384) to use 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.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Using the private storage pointed to by the `CMSession` handle, your `CMConcatInit` 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 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.

When your CMM uses a device link 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 default CMM ignores this profile. Your CMM should also ignore it.

For specific guidelines on handling device link profiles and additional information on handling concatenated profiles, see "ColorSync Reference for Applications and Drivers" (page 217).

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 to maintain a connection for the requesting application or driver. You should attempt to allocate memory from the current heap zone. If that attempt fails, you should allocate memory from the system heap or the temporary heap. You can use the `SetComponentInstanceStorage` function to associate the allocated memory with the component instance. 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 seven classes of profiles defined by the ICC. For information on the seven classes of profiles, see "ColorSync Reference for Applications and Drivers" (page 217).

CMMatchPixMap

Handles the `kCMMMatchPixMap` request by matching the colors of the specified pixel map image to the destination profile's color gamut.

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, `CMMatchPixMap`) to handle the request.

The `CMMatchPixMap` function is a color management module–defined subroutine.

```
pascal CMError CMMatchPixMap(
                    ComponentInstance CMSession,
                    PixMap *myPixMap,
                    CMBitmapCallBackUPP progressProc,
                    void *refCon);
```

`CMsession` 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 match. 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 `CMMatchPixMap` 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 `CMMatchPixMap` function must pass through as a parameter to calls it makes to the `CMBitmapCallBackProc` function.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

If your CMM supports this request code, your `CMMatchPixMap` function should be prepared to receive any of the pixel map types defined by QuickDraw.Your `CMMatchPixMap` 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 `CMMatchPixMap` 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 `CMMatchPixMap` 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 `CMMatchPixMap` 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 default CMM calls the progress function approximately every half-second, unless color matching takes less time; this happens when there is a small amount of data to match.

Here is the prototype for the `CMBitmapCallBackProc` function pointed to by the `progressProc` parameter:

```
pascal Boolean CMBitmapCallBackProc (
                            long progress,
                            void *refCon);
```

Each time your `CMMatchPixMap` function calls the `CMBitmapCallBackProc` 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 `CMBitmapCallBackProc` function, your function must pass it a byte count in the `progress` parameter identifying the remaining number of bytes. The last time your `CMMatchPixMap` function calls the `CMBitmapCallBackProc` 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, it 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 `CMMatchColors` 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.

CMCheckPixMap

Handles the `kCMMCheckPixMap` request by checking the colors of the specified pixel map image against the color gamut of the destination profile.

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, `CMCheckPixMap`) to handle the request.

The `CMCheckPixMap` function is a color management module–defined subroutine.

```
pascal CMError CMCheckPixMap(
                    ComponentInstance CMSession,
                    const PixMap *myPixMap,
```

```
                    CMBitmapCallBackUPP progressProc,
                    BitMap *myBitMap,
                    void *refCon);
```

`CMsession`	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 `CMCheckPixMap` function must call this function at regular intervals to allow it to report progress to the user.
`myBitMap`	A pointer to a QuickDraw bitmap whose boundaries equal those of the pixel map indicated by the `myPixMap` parameter. Your `CMCheckPixMap` 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 `CMCheckPixMap` function must pass through as a parameter to calls it makes to the `CMBitmapCallBackProc` function.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

If your CMM supports this request code, your `CMCheckPixMap` function should be prepared to receive any of the pixel map types defined by QuickDraw.Your `CMCheckPixMap` 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 `CMCheckPixMap` 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 `CMMatchPixMap` 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 `CMCheckPixMap` 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 default CMM calls the progress function approximately every half-second, unless color checking takes less time; this happens when there is a small amount of data to match.

Here is the prototype for the `CMBitmapCallBackProc` function pointed to by the `progressProc` parameter:

```
pascal Boolean CMBitmapCallBackProc (
                            long progress,
                            void *refCon);
```

Each time your `CMCheckPixMap` function calls the `CMBitmapCallBackProc` 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 `CMBitmapCallBackProc` function, your function must pass it a byte count in the `progress` parameter identifying the remaining number of bytes to check. As your `CMCheckPixMap` 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 `CMCheckPixMap`

function calls the CMBitmapCallBackProc 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, it 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 CMMatchColors 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.

CMNewLinkProfile

Handles the kCMMNewLinkProfile request by creating a single device link profile that includes the profiles in the specified profile set.

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, CMNewLinkProfile) to handle the request.

The CMNewLinkProfile function is a color management module–defined subroutine.

```
pascal CMError CMNewLinkProfile(
                    ComponentInstance CMSession,
                    CMProfileRef *prof,
                    const CMProfileLocation *targetLocation,
                    CMConcatProfileSet *profileSet);
```

CMsession — A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`prof`	A pointer to a reference to a device link profile of type `DeviceLink`. Your `CMNewLinkProfile` function creates this profile, opens it to obtain a reference to it, and returns a pointer to 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`	A pointer to a 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 "CMProfLoc" (page 361) and "CMProfileLocation" (page 362).
`profileSet`	A pointer to an array of profiles of type `CMConcatProfileSet` (page 384). Your function must include these profiles in order in any device link 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.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Your `CMNewLinkProfile` function must create a single device link 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 "Profile Class" (page 396). See also `CWNewLinkProfile` (page 267), which describes how to create a device-link profile.

After your function creates the device link 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.x, document revision 3.x, also describes device link profiles. For information on how to obtain a copy of this document, contact the Developer Support organization of Apple Computer.

CMMGetPS2ColorSpace

Handles the `kCMMGetPS2ColorSpace` request by obtaining or deriving the color space element data from the source profile.

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, `CMMGetPS2ColorSpace`) to handle the request.

The `CMMGetPS2ColorSpace` function is a color management module–defined subroutine.

```
pascal CMError CMMGetPS2ColorSpace(
                    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 from which you must obtain or derive the color space element data.

`flags` Reserved for future use.

`proc` A pointer to a `ColorSyncDataTransfer` function supplied by the calling application or device driver. Your `CMMGetPS2ColorSpace` 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 `CMMGetPS2ColorSpace` function must pass to the `ColorSyncDataTransfer` function.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Only for special cases should a custom CMM need to support this request code. If your CMM supports this function, your `CMMGetPS2ColorSpace` 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 `ColorSyncDataTransfer` 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 `ColorSyncDataTransfer` function supplied by the calling application or driver. Your `CMMGetPS2ColorSpace` function must call the `ColorSyncDataTransfer` function repeatedly until you have passed all the data to it. Here is the prototype for the `ColorSyncDataTransfer` function pointed to by the `proc` parameter:

```
pascal OSErr ColorSyncDataTransfer(
                        long command,
                        long *size,
                        void *data,
                        void *refCon);
```

Your `CMMGetPS2ColorSpace` function communicates with the `ColorSyncDataTransfer` function, using a command parameter to identify the operation to perform. Your function should call the `ColorSyncDataTransfer` function first with the `openWriteSpool` command to direct the `ColorSyncDataTransfer` function to begin the process of writing the profile color space element data you pass it in the `data` buffer. Next, you should call the `ColorSyncDataTransfer` function with the `writeSpool` command. After the `ColorSyncDataTransfer` function returns in the `size` parameter the amount of data it actually wrote, you should call the `ColorSyncDataTransfer` 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 `ColorSyncDataTransfer` function with the `closeSpool` command.

When your function calls the `ColorSyncDataTransfer` function, it passes in the `data` buffer the profile data to transfer to the `ColorSyncDataTransfer` function and the size in bytes of the buffered data in the `size` parameter. The `ColorSyncDataTransfer` function may not always write all the data you pass it in the `data` buffer. Therefore, on return the `ColorSyncDataTransfer` function

command passes back in the `size` parameter the number of bytes it actually wrote. Your `CMMGetPS2ColorSpace` function keeps track of the number of bytes of remaining color space element data.

Each time your `CMMGetPS2ColorSpace` function calls the `ColorSyncDataTransfer` 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.

CMMGetPS2ColorRenderingIntent

Handles the `kCMMGetPS2ColorRenderingIntent` request by obtaining the rendering intent from the source profile.

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, `CMMGetPS2ColorRenderingIntent`) to handle the request.

The `CMMGetPS2ColorRenderingIntent` function is a color management module–defined subroutine.

```
pascal CMError CMMGetPS2ColorRenderingIntent(
                    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 `CMMGetPS2ColorRenderingIntent` 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 `CMMGetPS2ColorRenderingIntent` function must pass to the `ColorSyncDataTransfer` function.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Only for special cases should a custom CMM need to support this request code. If your CMM supports this function, your `CMMGetPS2ColorRenderingIntent` 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 `ColorSyncDataTransfer` 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 `ColorSyncDataTransfer` function supplied by the calling application or driver. Your `CMMGetPS2ColorRenderingIntent` function must call the `ColorSyncDataTransfer` function repeatedly until you have passed all the data to it.

Here is the prototype for the `ColorSyncDataTransfer` function pointed to by the `proc` parameter:

```
pascal OSErr ColorSyncDataTransfer(
                        long command,
                        long *size,
                        void *data,
                        void *refCon);
```

Your `CMMGetPS2ColorRenderingIntent` function communicates with the `ColorSyncDataTransfer` function using a command parameter to identify the operation to perform. Your function should call the `ColorSyncDataTransfer` function first with the `openWriteSpool` command to direct the `ColorSyncDataTransfer` function to begin the process of writing the profile color-rendering intent element data you pass it in the `data` buffer. Next, you should call the `ColorSyncDataTransfer` function with the `writeSpool` command. After the `ColorSyncDataTransfer` function returns in the `size` parameter the amount of data it actually read, you should call the `ColorSyncDataTransfer` 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 `ColorSyncDataTransfer` function with the `closeSpool` command.

When your function calls the `ColorSyncDataTransfer` function, it passes in the `data` buffer the profile data to transfer to the `ColorSyncDataTransfer` function and the size in bytes of the buffered data in the `size` parameter. The `ColorSyncDataTransfer` function may not always write all the data you pass it in the `data` buffer. Therefore, on return the `ColorSyncDataTransfer` function command passes back in the `size` parameter the number of bytes it actually wrote. Your `CMMGetPS2ColorRenderingIntent` function keeps track of the number of bytes of remaining color-rendering intent element data.

Each time your `CMMGetPS2ColorRenderingIntent` function calls the `ColorSyncDataTransfer` 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.

CMMGetPS2ColorRendering

Handles the `kCMMGetPS2ColorRendering` request by obtaining the rendering intent from the header of the source profile.

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, `CMMGetPS2ColorRendering`) to handle the request.

The `CMMGetPS2ColorRendering` function is a color management module–defined subroutine.

```
pascal CMError CMMGetPS2ColorRendering(
                    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 color rendering dictionary (CRD).

`dstProf` A profile reference to the destination profile from which you obtain or derive the CRD.

`flags` Reserved for future use.

`proc` A pointer to a function supplied by the calling application or device driver. Your `CMMGetPS2ColorRendering` function calls this function repeatedly as necessary until you have passed all the CRD element data to this function.

`refCon` A reference constant, containing data specified by the calling application or device driver, that your `CMMGetPS2ColorRendering` function must pass to the `ColorSyncDataTransfer` function.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Only for special cases should a custom CMM need to support this request code. If your CMM supports this function, your `CMMGetPS2ColorRendering` function must obtain the rendering intent from the header of the source profile identified by the `srcProf` parameter. The rendering intent identifies the color rendering dictionary (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 `ColorSyncDataTransfer` 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 `ColorSyncDataTransfer` function supplied by the calling application or driver. Your `CMMGetPS2ColorRendering` function must call the `ColorSyncDataTransfer` function repeatedly until you have passed all the data to it. Here is the prototype for the `ColorSyncDataTransfer` function pointed to by the `proc` parameter:

```
pascal OSErr ColorSyncDataTransfer(
                        long command,
                        long *size,
                        void *data,
                        void *refCon);
```

Your `CMMGetPS2ColorRendering` function communicates with the `ColorSyncDataTransfer` function using a command parameter to identify the operation to perform. Your function should call the `ColorSyncDataTransfer` function first with the `openWriteSpool` command to direct the `ColorSyncDataTransfer` function to begin the process of writing the profile CRD data you pass it in the `data` buffer. Next, you should call the `ColorSyncDataTransfer` function with the `writeSpool` command. After the `ColorSyncDataTransfer` function returns in the `size` parameter the amount of data it actually wrote, you should call the `ColorSyncDataTransfer` 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 `ColorSyncDataTransfer` function with the `closeSpool` command.

When your function calls the `ColorSyncDataTransfer` function, it passes in the `data` buffer the profile data to transfer to the `ColorSyncDataTransfer` function and the size in bytes of the buffered data in the `size` parameter. The `ColorSyncDataTransfer` function may not always write all the data you pass it in the `data` buffer. Therefore, on return the `ColorSyncDataTransfer` function command passes back in the `size` parameter the number of bytes it actually wrote. Your `CMMGetPS2ColorRendering` function keeps track of the number of bytes of remaining CRD data.

Each time your `CMMGetPS2ColorRendering` function calls the `ColorSyncDataTransfer` 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.

CMMGetPS2ColorRenderingVMSize

Handles the `kCMMGetPS2ColorRenderingVMSize` request by obtaining the maximum virtual memory (VM) size of the color rendering dictionary (CRD) for the rendering intent specified by the source profile.

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, `CMMGetPS2ColorRenderingVMSize`) to handle the request.

The `CMMGetPS2ColorRenderingVMSize` function is a color management module–defined subroutine.

```
pascal CMError CMMGetPS2ColorRenderingVMSize(
                    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 use.
`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.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Only for special cases should a custom CMM need to support this request code. If your CMM supports this function, your `CMMGetPS2ColorRenderingVMSize` 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            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 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.

`intentCRD`	A variable-sized array of four or more members defined by the `CMIntentCRDSize` data type.

CMMFlattenProfile

CHANGED IN COLORSYNC 2.5

Handles the `kCMMFlattenProfile` request by extracting profile data from the profile to flatten and passing it to the specified function.

A CMM may respond to the `kCMMFlattenProfile` request code, but it is not required to do so. Most CMMs can rely on the default CMM to handle this request code adequately. 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, `CMMFlattenProfile`).

The `CMMFlattenProfile` function is a color management module–defined subroutine.

```
pascal CMError CMMFlattenProfile (
                    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 flatten.
`flags`	Reserved for future use.
`proc`	A pointer to the `ColorSyncDataTransfer` function supplied by the calling application or device driver to perform the low-level data transfer. Your `CMMFlattenProfile` 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.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Only in rare circumstances should a custom CMM need to support this request code. The process of flattening a profile is complex, and the 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 `CMMFlattenProfile` function must extract the profile data from the profile to flatten, identified by the `prof` parameter, and pass the profile data to the function specified in the `proc` parameter.

Your `CMMFlattenProfile` function calls the `ColorSyncDataTransfer` function supplied by the calling application. Here is the prototype for the `ColorSyncDataTransfer` function pointed to by the `proc` parameter:

```
pascal OSErr ColorSyncDataTransfer(
                        long command,
                        long *size,
                        void *data,
                        void *refCon);
```

Your `CMMFlattenProfile` function communicates with the `ColorSyncDataTransfer` function using a command parameter to identify the operation to perform. Your function should call the `ColorSyncDataTransfer` function first with the `openWriteSpool` command to direct the `ColorSyncDataTransfer` function to begin the process of writing the profile data you pass it in the `data` buffer. Next, you should call the `ColorSyncDataTransfer` function with the `writeSpool` command. After the `ColorSyncDataTransfer` function returns in the `size` parameter the amount of data it actually wrote, you should call the `ColorSyncDataTransfer` 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 `ColorSyncDataTransfer` function with the `closeSpool` command.

When your function calls the `ColorSyncDataTransfer` function, it passes in the `data` buffer the profile data to transfer to the `ColorSyncDataTransfer` function and the size in bytes of the buffered data in the `size` parameter. The `ColorSyncDataTransfer` function may not always write all the data you pass it in the `data` buffer. Therefore, on return the `ColorSyncDataTransfer` 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 `CMMFlattenProfile` function is responsible for obtaining the profile data from the profile, allocating a buffer in which to pass the data to the `ColorSyncDataTransfer` function, and keeping track of the amount of remaining data to transfer to the `ColorSyncDataTransfer` function.

Each time your `CMMFlattenProfile` function calls the `ColorSyncDataTransfer` function, you pass it the reference constant
passed to your function in the reference constant parameter.

VERSION NOTES

Starting with ColorSync version 2.5, the ColorSync Manager calls the function provided by the calling program directly, without going through the preferred, or any, CMM. Your CMM only needs to handle the `kCMMFlattenProfile` request code for versions of ColorSync prior to version 2.5.

CMMUnflattenProfile

CHANGED IN COLORSYNC 2.5

Handles the `kCMMUnflattenProfile` request by creating a uniquely-named file in the temporary items folder to store the profile data.

A CMM may respond to the `kCMMUnflattenProfile` request code, but it is not required to do so. Most CMMs can rely on the default CMM to handle this request code adequately. 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, `CMMUnflattenProfile`).

The `CMMUnflattenProfile` function is a color management module–defined subroutine.

```
pascal CMError CMMUnflattenProfile (
                    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 pointer to 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 `CMMFlattenProfile` 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.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Only in rare circumstances should a custom CMM need to support this request code. The process of unflattening a profile is complex, and the 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 `CMMUnflattenProfile` 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 `CMMUnflattenProfile` function calls the `ColorSyncDataTransfer` function supplied by the calling application or device driver. Here is the prototype for the `ColorSyncDataTransfer` function pointed to by the `proc` parameter:

```
pascal OSErr ColorSyncDataTransfer (
                        long command,
                        long *size,
                        void *data,
                        void *refCon);
```

Before calling the `ColorSyncDataTransfer` function, your `CMMUnflattenProfile` function must allocate a buffer to hold the profile data returned to you from the `ColorSyncDataTransfer` function in the `data` parameter.

Your `CMMUnflattenProfile` function communicates with the `ColorSyncDataTransfer` function using a command parameter to identify the operation to perform. Your function should call the `ColorSyncDataTransfer` function first with the `openReadSpool` command to direct the `ColorSyncDataTransfer` function to begin the process of transferring data. Following this, you should call the `ColorSyncDataTransfer` function with the `readSpool` command as often as necessary until the `ColorSyncDataTransfer` 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 `ColorSyncDataTransfer` function with the `closeSpool` command.

Each time you call the `ColorSyncDataTransfer` function, you should pass it a pointer to the `data` buffer you created, the size in bytes of the profile data to return to you in the buffer, and the reference constant passed to you from the calling application.

On return, the `ColorSyncDataTransfer` 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 `ColorSyncDataTransfer` function will not always transfer the number of bytes of profile data you requested. Therefore, the `ColorSyncDataTransfer` 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 `CMMUnflattenProfile` function must

identify the profile size and maintain a counter tracking the amount of data transferred to you and the amount of remaining data to determine when to call the `ColorSyncDataTransfer` 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 `ColorSyncDataTransfer` function information the `ColorSyncDataTransfer` function requires to transfer the data.

VERSION NOTES

Starting with ColorSync version 2.5, the ColorSync Manager calls the function provided by the calling program directly, without going through the preferred, or any, CMM. Your CMM only needs to handle the `kCMMUnflattenProfile` request code for versions of ColorSync prior to version 2.5.

CMMGetNamedColorInfo

Handles the `kCMMGetNamedColorInfo` request by returning information about a named color space from its profile reference.

A CMM may respond to the `kCMMGetNamedColorInfo` request code, but it is not required to do so. Most CMMs can rely on the default CMM to handle this request code adequately. The ColorSync Manager sends this request code to your CMM on behalf of an application or device driver that called the `CMMGetNamedColorInfo` function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM that handles the `kCMMGetNamedColorInfo` request code typically responds by calling a CMM-defined function (for example, `CMMGetNamedColorInfo`).

The `CMMGetNamedColorInfo` function is a color management module–defined subroutine.

```
pascal CMError CMMGetNamedColorInfo(
                    ComponentInstance CMSession,
                    CMProfileRef srcProf,
                    unsigned long *deviceChannels,
                    OSType *deviceColorSpace,
                    OSType *PCSColorSpace,
```

```
                    unsigned long *count,
                    StringPtr prefix,
                    StringPtr suffix)
```

`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 a named color space profile to supply information about.

`deviceChannels`
A pointer to a count of device channels. On output, the number of device channels in the color space for the profile. It should agree with the "data color space" field in the profile header. For example, Pantone maps to CMYK, a 4-channel color space. A value of 0 indicates no device channels were available.

`deviceColorSpace`
A pointer to a device color space. On output, specifies a device color space, such as CMYK, for the `srcProf` profile.

`PCSColorSpace`
A pointer to a profile connection space color space. On output, specifies an interchange color space, such as Lab, for the `srcProf` profile.

`count` A pointer to a named color count. On output, the number of named colors in the `srcProf` profile.

`prefix` A pointer to a Pascal string. On output, the string contains a prefix, such as "Pantone", for each color name. The prefix identifies the named color system described by the `srcProf` profile.

`suffix` A pointer to a Pascal string. On output, the string contains a suffix, such as "CVC", for each color name.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

The `CMMGetNamedColorInfo` function returns information about the named color space referred to by the passed profile reference.

CMMGetNamedColorValue

Handles the `kCMMGetNamedColorValue` request by returning device and PCS color values from a named color space profile for a specific color name.

A CMM may respond to the `kCMMGetNamedColorValue` request code, but it is not required to do so. Most CMMs can rely on the default CMM to handle this request code adequately. The ColorSync Manager sends this request code to your CMM on behalf of an application or device driver that called the `CMMGetNamedColorValue` function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM that handles the `kCMMGetNamedColorValue` request code typically responds by calling a CMM-defined function (for example, `CMMGetNamedColorValue`).

The `CMMGetNamedColorValue` function is a color management module–defined subroutine.

```
pascal CMError CMMGetNamedColorValue(
                    ComponentInstance CMSession,
                    CMProfileRef prof,
                    StringPtr name,
                    CMColor *deviceColor,
                    CMColor *PCSColor)
```

`CMSession` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`prof` A profile reference of type `CMProfileRef` (page 358) that identifies the named color space profile to extract named color information from.

`name` A pointer to a Pascal color name string that identifies the named color to return color values for.

`deviceColor` A pointer to a device color. On output, a device color value in `CMColor` union format. If the profile does not contain device values, `deviceColor` is undefined.

`PCSColor` A pointer to a profile connection space color. On output, an interchange color value in `CMColor` union format.

function result A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Based on the passed color name, the `CMMGetNamedColorValue` function does a lookup into the named color tag in the profile whose reference is passed in the `prof` parameter and, if the name is found in the tag, returns device and PCS color values. Otherwise, `CMMGetNamedColorValue` returns an error code.

CMMGetIndNamedColorValue

Handles the `kCMMGetIndNamedColorValue` request by returning device and PCS color values from a named color space profile for a specific named color index.

A CMM may respond to the `kCMMGetIndNamedColorValue` request code, but it is not required to do so. Most CMMs can rely on the default CMM to handle this request code adequately. The ColorSync Manager sends this request code to your CMM on behalf of an application or device driver that called the `CMGetIndNamedColorValue` function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM that handles the `kCMMGetIndNamedColorValue` request code typically responds by calling a CMM-defined function (for example, `CMMGetIndNamedColorValue`).

The `CMMGetIndNamedColorValue` function is a color management module–defined subroutine.

```
pascal CMError CMMGetIndNamedColorValue(
                    ComponentInstance CMSession,
                    CMProfileRef prof,
                    unsigned long index,
                    CMColor *deviceColor,
                    CMColor *PCSColor)
```

`CMSession` A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.

`prof`	A profile reference of type `CMProfileRef` (page 358) to a named color space profile.
`index`	An index value for the named color to get color values for.
`deviceColor`	A pointer to a device color. On output, a device color value in `CMColor` union format. If the profile does not contain device values, `deviceColor` is undefined.
`PCSColor`	A pointer to a profile connection space color. On output, an interchange color value in `CMColor` union format.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Based on the passed named color index, the `CMMGetIndNamedColorValue` function does a lookup into the named color tag of the profile whose reference is passed in the `prof` parameter and returns device and PCS color values. If the index is greater than the number of named colors, `CMMGetIndNamedColorValue` returns an error code.

CMMGetNamedColorIndex

Handles the `kCMMGetNamedColorIndex` request by returning a named color index from a named color space profile for a specific color name.

A CMM may respond to the `kCMMGetNamedColorIndex` request code, but it is not required to do so. Most CMMs can rely on the default CMM to handle this request code adequately. The ColorSync Manager sends this request code to your CMM on behalf of an application or device driver that called the `CMMGetNamedColorIndex` function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM that handles the `kCMMGetNamedColorIndex` request code typically responds by calling a CMM-defined function (for example, `CMMGetNamedColorIndex`).

The `CMMGetNamedColorIndex` function is a color management module–defined subroutine.

```
pascal CMError CMMGetNamedColorIndex(
                    ComponentInstance CMSession,
                    CMProfileRef prof,
                    StringPtr name,
                    unsigned long *index)
```

`CMSession`	A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.
`prof`	A profile reference of type `CMProfileRef` (page 358) to a named color space profile.
`name`	A pointer to a Pascal color name string that identifies the named color to return the index value for.
`index`	A pointer to a value of type `unsigned long`. On output, it specifies the index value for the named color specified by `name`.
function result	A result code of type `CMError`. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Based on the passed color name, the `CMMGetNamedColorIndex` function does a lookup into the named color tag of the profile whose reference is passed in the `prof` parameter and, if the name is found in the tag, returns the index. Otherwise, `CMMGetNamedColorIndex` returns an error code.

CMMGetNamedColorName

Handles the `kCMMGetNamedColorName` request by returning a named color name from a named color space profile for a specific named color index.

A CMM may respond to the `kCMMGetNamedColorName` request code, but it is not required to do so. Most CMMs can rely on the default CMM to handle this request code adequately. The ColorSync Manager sends this request code to your CMM on behalf of an application or device driver that called the

CMMGetNamedColorName function. The ColorSync Manager dispatches this request to the Component Manager, which calls your CMM to service the request. A CMM that handles the kCMMGetNamedColorName request code typically responds by calling a CMM-defined function (for example, CMMGetNamedColorName).

The CMMGetNamedColorName function is a color management module–defined subroutine.

```
pascal CMError CMMGetNamedColorName(
                    ComponentInstance CMSession,
                    CMProfileRef prof,
                    unsigned long index,
                    StringPtr name)
```

CMSession	A handle to your CMM's storage for the instance of your component associated with the calling application or device driver.
prof	A profile reference to a named color space profile.
index	An index value for a named color.
name	A pointer to a Pascal color name string. On output, it identifies the named color specified by index.
function result	A result code of type CMError. For possible values, see "Result Codes for the ColorSync Manager" (page 425).

DISCUSSION

Based on the passed color name index, the CMMGetNamedColorName function does a lookup into the named color tag of the profile whose reference is passed in the prof parameter and returns the name. If the index is greater than the number of named colors, CMMGetNamedColorName returns an error code.

Constants

This section describes the constants for the CMM component interface version and the ColorSync Manager request codes.

- "Color Management Module Component Interface" (page 515) describes how to specify the ColorSync Manager version your CMM supports.
- "Required Request Codes" (page 515) describes ColorSync Manager request codes your CMM must respond to.
- "Optional Request Codes" (page 517) describes ColorSync Manager request codes your CMM may optionally respond to.

Color Management Module Component Interface

If your CMM supports the ColorSync Manager version 2.x, 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     /* Version 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 only 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 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 “Developing Color Management Modules” (page 429).

The ColorSync Manager defines the following required request codes:

```
enum {
    kCMMInit            = 0,    /* initialize (for 1.0 compatability) */
    kCMMMatchColors     = 1,    /* match colors */
    kCMMCheckColors     = 2     /* gamut-check colors */
    kNCMMInit           = 6,    /* initialize (2.x) */
};
```

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.x profiles, you must support this request code. If you support only ColorSync Manager version 2.x 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 function `CMMatchColors` (page 470).

`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 function `CMCheckColors` (page 472).

`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 function `NCMInit` (page 468).

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" (page 429).

The ColorSync Manager defines the following optional request codes:

```
enum {
    kCMMMatchPixMap                 = 3,    /* match colors of pix map image */
    kCMMCheckPixMap                 = 4,    /* gamut-check pix map colors */
    kCMMConcatenateProfiles         = 5,    /* concatenate profiles (for backward
                                               compatibility with ColorSync 1.0) */
    kCMMConcatInit                  = 7,    /* init prior to concat color worlds */
    kCMMValidateProfile             = 8,    /* validate profile elements */
    kCMMMatchBitmap                 = 9,    /* match colors of bit map image */
    kCMMCheckBitmap                 = 10,   /* gamut-check bit map colors */
    kCMMGetPS2ColorSpace            = 11,   /* get PostScript color space */
    kCMMGetPS2ColorRenderingIntent  = 12,   /* get PostScript rendering intent */
    kCMMGetPS2ColorRendering        = 13,   /* get PostScript color rendering
                                               dictionary */
    kCMMFlattenProfile              = 14,   /* extract data from profile */
    kCMMUnflattenProfile            = 15,   /* create file for unflattened data */
    kCMMNewLinkProfile              = 16,   /* create linked profile */
    kCMMGetPS2ColorRenderingVMSize  = 17,   /* get PostScript version 2 color rendering
                                               dictionary virtual memory size */
    kCMMGetNamedColorInfo           = 70,   /* get named color information */
    kCMMGetNamedColorValue          = 71,   /* get color values for a named color */
    kCMMGetIndNamedColorValue       = 72,   /* get index for a named color */
    kCMMGetNamedColorIndex          = 73,   /* get named color index from name */
    kCMMGetNamedColorName           = 74    /* get named color name from index */
};
```

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 function `CMMatchPixMap` (page 486).

`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 function `CMCheckPixMap` (page 488).

`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 function `CMConcatInit` (page 483).

`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 function `CMMValidateProfile` (page 476).

`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 function `CMMatchBitmap` (page 477).

`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 function `CMCheckBitmap` (page 480).

`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 function `CMMGetPS2ColorSpace` (page 493).

`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 function `CMMGetPS2ColorRenderingIntent` (page 495).

`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 function `CMMGetPS2ColorRendering` (page 497).

`kCMMFlattenProfile`

In response to this request code, your CMM must extract the profile data from the profile to flatten, 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 function `CMMFlattenProfile` (page 503).

Changed in ColorSync 2.5: Starting with ColorSync version 2.5, the ColorSync Manager calls the function provided by the calling program directly, without going through the preferred, or any, CMM. Your CMM only needs to handle this request code for versions of ColorSync prior to version 2.5.

`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 function `CMMUnflattenProfile` (page 505).

Changed in ColorSync 2.5: Starting with ColorSync version 2.5, the ColorSync Manager calls the function provided by the calling program directly, without going through the preferred, or any, CMM. Your CMM only needs to handle this request code for versions of ColorSync prior to version 2.5.

`kCMMNewLinkProfile`

In response to this request code, your CMM must create a single device link 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 function `CMNewLinkProfile` (page 491).

`kCMMGetPS2ColorRenderingVMSize`

In response to this request code, your CMM must obtain or assess the maximum virtual memory (VM) size of the color rendering dictionary (CRD) specified by the destination profile. You must return the size of the CRD for the rendering intent specified by the source profile. See the function `CMMGetPS2ColorRenderingVMSize` (page 500) for more information about how your CMM should respond to this request code.

`kCMMGetNamedColorInfo`

In response to this request code, your CMM extracts named color data from the profile whose reference is passed in the `srcProf` parameter. For more information, see the function `CMMGetNamedColorInfo` (page 508).

`kCMMGetNamedColorValue`

In response to this request code, your CMM extracts device and profile connection space (PCS) color values for a specific color name from the profile whose reference is passed in the `prof` parameter. For more information, see the function `CMMGetNamedColorValue` (page 510).

`kCMMGetIndNamedColorValue`

In response to this request code, your CMM extracts device and PCS color values for a specific named color index from the profile whose reference is passed in the `prof` parameter. For more information, see the function `CMMGetIndNamedColorValue` (page 511).

`kCMMGetNamedColorIndex`

In response to this request code, your CMM extracts a named color index for a specific color name from the profile whose reference is passed in the `prof` parameter. For more information, see the function `CMMGetNamedColorIndex` (page 512).

`kCMMGetNamedColorName`

In response to this request code, your CMM extracts a named color name for a specific named color index from the profile whose reference is passed in the `prof` parameter. For more information, see the function `CMMGetNamedColorName` (page 513).

CHAPTER 8

Version and Compatibility Information

Contents

This section describes the `Gestalt` information, shared library version numbers, CMM version numbers, and ColorSync header files you use with different versions of the ColorSync Manager. It also describes CPU and system requirements.

This section also describes backward compatibility support for ColorSync 1.0 functions, profiles, and CMMs provided by the ColorSync Manager in versions 2.0 and later.

In addition, this section explains how to use the ColorSync Manager for color matching between a ColorSync 1.0 profile and a version 2.x profile.

Version 2.5 of the ColorSync Manager replaces all earlier versions of the product, including ColorSync 2.1, 2.0, and 1.0.

Note
There are no changes to the ColorSync Manager API between version 2.5 and version 2.5.1, so this document is up-to-date for ColorSync 2.5.1. ◆

Although ColorSync 1.0 used a proprietary profile format, the ColorSync Manager provides backward compatibility for applications and device drivers written for ColorSync 1.0. Your application that uses the ColorSync Manager can match, convert, and color check colors using version 2.x profiles or, when necessary, using a combination of version 2.x profiles and ColorSync 1.0 profiles.

IMPORTANT
Although ColorSync version 2.5 fully supports 1.0 format profiles, this support is not guaranteed to continue in future versions. Apple strongly recommends that developers using the 1.0 format move to the 2.x format. ▲

ColorSync Version Information

This section describes the `Gestalt` information, shared library version numbers, CMM version numbers, CPUs, system versions, and ColorSync header files you use with different versions of the ColorSync Manager. Information is provided in the following sections:

- "Gestalt, Shared Library, and CMM Version Information" (page 526)
- "CPU and System Requirements" (page 527)
- "ColorSync Header Files" (page 528)

For additional version information, see the section "ColorSync Versions" (page 48) and "ColorSync and ICC Profile Format Version Numbers" (page 50).

Gestalt, Shared Library, and CMM Version Information

Table 8-1 lists the version number for each release of the ColorSync Manager, along with the `Gestalt` version number, shared library version number, and `Gestalt` selector code for that version. Note that only the ColorSync version numbers and `Gestalt` version numbers are unique for each version. For more information on `Gestalt` selectors, see "Gestalt Selector Codes for the ColorSync Manager" (page 217).

Table 8-1 ColorSync Manager version numbers, with corresponding shared library version numbers and `Gestalt` selectors

ColorSync Version	Gestalt Version	Gestalt Selector	Shared Library Version	Color Management Module (CMM) Version
1.0	$00000100	`gestaltColorSync10`	$00000000	$00000001
1.0.3	$00000110	`gestaltColorSync11`	$00000000	$00000001
1.0.4	$00000104	`gestaltColorSync104`	$00000000	$00000001
1.0.5	$00000105	`gestaltColorSync105`	$00000000	$00000001
2.0	$00000200	`gestaltColorSync20`	$02000000	$00010001
2.0.1	$00000200	`gestaltColorSync20`	$02000000	$00010001
2.1.0	$00000210	`gestaltColorSync21`	$02100000	$00010001
2.1.1	$00000211	`gestaltColorSync21`	$02100000	$00010001
2.1.2	$00000212	`gestaltColorSync21`	$02100000	$00010001
2.5	$00000250	`gestaltColorSync25`	$02500000	$00010002
2.5.1	$00000251	`gestaltColorSync251`	$02500000	$00010002

CPU and System Requirements

Table 8-2 lists the CPU and system requirements for each release of the ColorSync Manager.

Table 8-2 ColorSync Manager CPU and system requirements

ColorSync Version	CPU	System Version
1.0, 1.0.3, 1.0.4, 1.0.5	68K or PowerPC	On 68K, requires either System 7.0 or System 6.0.7 with 32-bit QuickDraw, version 1.2. For PowerPC, requires System 7.0.
2.0, 2.0.1, 2.1.0, 2.1.1, 2.1.2	68020 or greater or PowerPC	Requires System 7.0 or greater with Color QuickDraw.
2.5, 2.5.1	68020 or greater or PowerPC	Requires System 7.6.1 or greater.

ColorSync Header Files

Table 8-3 describes the ColorSync Manager header files. Note that some header files are no longer used or are not recommended.

Table 8-3 ColorSync header files

Header File	Description	First Used	Status
CMAcceleration.h	CMM acceleration component interface.	2.0	Not used starting with 2.1.
CMApplication.h	ColorSync Manager functions, constants, and data types for applications, device drivers, and CMMs.	1.0	Supported.
CMCalibrator.h	Interface for developing monitor calibrator plug-ins.	2.5	Supported, but not documented here.
CMComponent.h	Old component interface for CMMs. Replaced by CMMComponent.h.	1.0	Not used starting with 2.0.
CMConversions.h	Component interface for old-style conversion routines.	2.0	Supported, but not recommended starting with 2.1.
CMICCProfile.h	Constants and data types for working with ICC profiles.	1.0	Supported.
CMMComponent.h	Component interface for ColorSync CMMs.	1.0	Supported.
CMPRComponent.h	Component interface for ColorSync 1.0 profile responders.	1.0	Supported, but not recommended starting with 2.0.
CMScriptingPlugin.h	Interface for developing scripting plug-ins.	2.5	Supported, but not documented here.

ColorSync Manager 2.x Backward Compatibility

The following sections describe backward compatibility for ColorSync Manager versions greater than 2.0.

ColorSync 2.1 Support in Version 2.5

Existing code written to use version 2.1 of the ColorSync Manager should continue to work with ColorSync 2.5 without modification. Existing code may operate more efficiently in some cases due to optimizations provided with version 2.5, especially for multiple processors, as described in “When ColorSync Uses Multiple Processors” (page 73). However, existing code can not take full advantage of some new features; for example, see “The Profile Cache and Optimized Searching” (page 57).

For a guide to the new features in version 2.5 of the ColorSync Manager, see “New Features in ColorSync Manager Version 2.5” (page 539).

ColorSync 2.0 Support in Version 2.1

Existing code written to use version 2.0 of the ColorSync Manager should continue to work with ColorSync 2.1 without modification, although it will not necessarily take advantage of the new features described in “New Features in ColorSync Manager Version 2.1” (page 550). For example, code written for ColorSync version 2.0 cannot use the *profile identifier*, an abbreviated data structure that identifies, and possibly modifies, a profile in memory or on disk. An embedded profile identifier requires much less space than an entire profile.

ColorSync Manager 1.0 Backward Compatibility

The ColorSync Manager continues to fully support the ColorSync 1.0 interface, including the ColorSync 1.0 profile responder. Note, that this support is provided primarily for backward compatibility. If you are writing new code, you should take advantage of the many new features added between version 2.0 and version 2.5. However, existing applications and drivers that use

ColorSync 1.0 functions will continue to work properly, as will ColorSync 1.0 profiles, ColorSync 1.0 CMMs, and QuickDraw GX 1.0.

Although newer versions of the ColorSync Manager continue to support use of the profile responder from ColorSync 1.0, this feature is not supported by the ColorSync Manager interface.

ColorSync 1.0 Profile Support

The ColorSync Manager continues to support the use of ColorSync 1.0 profiles. For example, you should always use ColorSync 1.0 functions with ColorSync 1.0 profiles, if possible. For example, always use ColorSync 1.0 functions to match colors between the color gamuts of two devices if both devices have ColorSync 1.0 profiles. The four ColorSync 1.0 functions and their new counterparts are listed in Table 8-4 (page 536).

However, there are times when you may need to use a ColorSync 1.0 profile with a ColorSync 2.x function. The ColorSync Manager's backward compatibility allows you to do this. For example, a document containing an image to be color matched may include an embedded ColorSync 1.0 source profile for the image. To match the colors of the source image to a device that has a version 2.x profile, you must use 2.x functions because ColorSync 1.0 functions cannot gain access to a version 2.x profile.

IMPORTANT

Although ColorSync version 2.5 fully supports 1.0 format profiles, this support is not guaranteed to continue in future versions. Apple strongly recommends that developers using the 1.0 format move to the 2.x format. ▲

One of the main differences between ColorSync 1.0 and 2.x functions is the profile format used. The 2.x functions accommodate ColorSync 1.0 profiles so that you can use those profiles if you must. Before describing how to use a ColorSync 1.0 profile with the 2.x functions, this section explains the differences between the ColorSync 1.0 profile format and the version 2.x profile format defined by the International Color Consortium (ICC) and used by the ColorSync Manager.

ColorSync 1.0 Profiles and Version 2.x Profiles

The ColorSync 1.0 profile format was designed by Apple Computer. This profile is memory resident and follows an internal structure based on tables. Although it is an open format, it is not an industry standard.

The ICC profile format implemented in the ColorSync Manager is significantly different from the profile format implemented for ColorSync 1.0. The version 2.x profile format is specified by the ICC and provides an industry standard that allows for interoperability across platforms and devices. A version 2.x profile created for a particular device can be used on systems running different operating systems.

Because the ColorSync 1.0 and version 2.x profile formats differ, the ColorSync Manager must resolve any compatibility issues involving accessing profiles and color matching between profiles. The next section describes how these profile formats differ.

How ColorSync 1.0 Profiles and Version 2.x Profiles Differ

A ColorSync 1.0 profile is smaller than a version 2.x profile and can therefore reside in memory. It is handle-based. A version 2.x profile as implemented by the ColorSync Manager is commonly file-based, but it can also be memory-based. You use an abstract internal data structure, called a profile reference, to access a version 2.x profile.

A ColorSync 1.0 profile contains a header, a copy of the Apple `CMProfileChromaticities` record, profile response data for the associated device, and a profile name string for use in dialog boxes. Custom profiles may also have additional, private data. ColorSync 1.0 defines the following profile data structure:

```
struct CMProfile {
    CMHeader                    header;
    CMProfileChromaticities     profile;
    CMProfileResponse           response;
    CMIString                   profileName;    /* variable length */
    char                        customData[anyNumber];
                                    /* optional custom CMM data */
};
```

The `response` data fields contain nine tables. The first table is for grayscale values. The next three are red, green, and blue values, followed by three for

cyan, magenta, and yellow values. The eighth and ninth tables are for CMYK printers requiring undercolor removal and black generation data.

The ColorSync 1.0 profile header, defined by the data structure `CMHeader` (page 351), and the version 2.x profile header, defined by the structure `CM2Header` (page 354), contain many fields in common. However, some fields in the ColorSync 1.0 profile header reflect its table-based nature, while a version 2.x profile has a tagged-element structure. A version 2.x profile also supports use of lookup table transforms that allow for faster processing.

CMMs and Mixed Profiles

Although version 2.x of the ColorSync Manager supports using a mix of ColorSync 1.0 and version 2.x profiles, the success of a matching session involving a ColorSync 1.0 profile depends on the CMM component performing the process. Third-party CMMs may choose not to support ColorSync 1.0 profiles. The default CMM is able to establish a matching session involving one or more ColorSync 1.0 profiles.

For device link profiles, you must include only version 2.x profiles. You cannot mix ColorSync 1.0 and version 2.x profiles in a device link profile.

Converting a 2.x Profile to the 1.0 Format

The ColorSync Manager provides the `CMConvertProfile2to1` function to convert 2.x format profiles to the 1.0 profile format. Because 1.0 and 2.x *scanner and monitor profiles* generally carry the same required color information, no accuracy is lost in converting from one to the other. With *printer profiles*, however, some accuracy will be lost by conversion, leading to significantly different results. Because of the possible loss of accuracy in some cases, 2.x to 1.0 profile conversion is not encouraged.

Using Newer Versions of the ColorSync Manager With ColorSync 1.0 Profiles

Despite differences between the version 2.x and ColorSync 1.0 profile formats, you can use most of the ColorSync Manager 2.x functions to gain access to ColorSync 1.0 profiles and their contents and to color match to and from the two disparate profile formats, if necessary. The ColorSync Manager makes this possible.

You can open a reference to a ColorSync 1.0 profile using 2.x functions and special data structures that accommodate both profile styles. You can also match the colors of an image expressed in the color gamut of one device whose characteristics are described by a ColorSync 1.0 profile to the colors within the gamut of another device whose characteristics are described by a version 2.x profile.

IMPORTANT

If you are color matching between devices that both use ColorSync 1.0 profiles, you should use the ColorSync functions that work with 1.0 profiles for the process. ▲

The next section describes

- which version 2.x functions you cannot use for ColorSync 1.0 profiles
- how you can use the ColorSync Manager with ColorSync 1.0 profiles

ColorSync Manager 2.x Functions Not Supported for ColorSync 1.0 Profiles

You cannot use the ColorSync Manager's `CMUpdateProfile` function to update a ColorSync 1.0 profile. The ColorSync Manager does not provide functions for profile version conversions. This is the domain of profile-building tools and calibration applications.

The ColorSync Manager 2.x versions provide a set of functions to search the ColorSync Profiles folder for specific profiles that meet search criteria. These functions act on version 2.x profiles only. If the ColorSync Profiles folder contains ColorSync 1.0 profiles, these functions do not acknowledge them or return results that include them. The 2.x search functions, which are not supported for ColorSync 1.0 functions, are the `CMIterateColorSyncFolder` (page 304), `CMNewProfileSearch` (page 308), `CMUpdateProfileSearch` (page 310), `CMDisposeProfileSearch` (page 311), `CMSearchGetIndProfile` (page 312), `CMSearchGetIndProfileFileSpec` (page 313), `CMProfileIdentifierFolderSearch` (page 315), and `CMProfileIdentifierListSearch` (page 316) functions.

You cannot use the ColorSync Manager's `NCMUseProfileComment` function to generate automatically the picture comments required to embed a ColorSync 1.0 profile. This function is designed to work with version 2.x profiles only.

Using ColorSync 1.0 Profiles With Newer Versions of the ColorSync Manager

You can use versions 2.0 and higher of the ColorSync Manager to match a document image with an embedded 1.0 source profile to the color gamut of a printer defined by a version 2.x profile. Newer versions of the ColorSync Manager are able to contend with both profile formats.

The sections that follow explain how to obtain a reference to the ColorSync 1.0 profile, get the profile's header, and get its synthesized tags.

Opening a ColorSync 1.0 Profile

To use a ColorSync 1.0 profile, you must obtain a reference to the profile. Obtaining a reference to the profile is synonymous with opening the profile for your program's use. If the profile is embedded in a document, you must extract the profile before you can open it.

You can use the `CMOpenProfileFile` function to obtain a reference to a ColorSync 1.0 profile. Other ColorSync Manager functions that you use to gain access to the profile's contents or perform color matching based on the profile require the profile reference as a parameter.

Obtaining a ColorSync 1.0 Profile Header

After you obtain a reference to a profile, you can gain access to the profile's contents. To gain access to the contents of any of the fields of a profile header, you must get the entire header. The ColorSync Manager allows you to do this using the `CMGetProfileHeader` function. You pass this function the profile reference and a data structure to hold the returned header. The ColorSync Manager defines the following union of type `CMAppleProfileHeader`, containing variants for ColorSync 1.0 and version 2.x ColorSync profile headers for this purpose:

```
union CMAppleProfileHeader {
    CMHeader        cm1;
    CM2Header       cm2;
};
```

You use the `cm1` variant for a ColorSync 1.0 profile header. You can easily test for the version of a profile header to determine which variant to use because the offset of the header version is at the same place for both ColorSync 1.0 profiles and version 2.x profiles.

Obtaining ColorSync 1.0 Profile Elements

The ColorSync Manager provides four tags to allow you to obtain four ColorSync 1.0 profile elements pointed to from the profile header or contained outside the header. To obtain the profile element, you specify its associated tag signature as a parameter to the `CMGetProfileElement` function along with the profile reference. The ColorSync Manager provides the following enumeration that defines these tags:

```
enum {
        cmCS1ChromTag   = 'chrm',
        cmCS1TRCTag     = 'trc ',
        cmCS1NameTag    = 'name',
        cmCS1CustTag    = 'cust'
};
```

`cmCS1ChromTag` Profile chromaticities tag signature. Element data for this tag specifies the XYZ chromaticities for the six primary and secondary colors (red, green, blue, cyan, magenta, and yellow).

`cmCS1TRCTag` Profile response data tag signature. Element data for this tag specifies the profile response data for the associated device.

`cmCS1NameTag` Profile name string tag signature. Element data for this tag specifies the profile name string. This is an international string consisting of a Macintosh script code followed by a length byte and up to 63 additional bytes composing a text string that identifies the profile.

`cmCS1CustTag` Custom tag signature. Element data for this tag specifies the private data for a custom CMM.

Embedding ColorSync 1.0 Profiles

In ColorSync 1.0, picture comment types `cmBeginProfile` and `cmEndProfile` are used to begin and end a picture comment.

The `cmEnableMatching` and `cmDisableMatching` picture comments are used to begin and end color matching in ColorSync 1.0 and in newer versions of the ColorSync Manager.

ColorSync 1.0 Functions With Parallel 2.x Counterparts

Starting with version 2.0, the ColorSync Manager implements new versions of four of the functions supported by ColorSync 1.0. In the new version of these functions, for example, a parameter used to specify a profile takes a profile reference.

It is easy to spot a ColorSync 2.x function that is a new version of a ColorSync 1.0 function, because the function name begins with an uppercase letter *N*, signifying that it is new. The four ColorSync 1.0 functions and their new counterparts are listed in Table 8-4.

Table 8-4 ColorSync 1.0 functions and their ColorSync Manager counterparts

ColorSync 1.0 function	ColorSync 2.x function
`pascal CWNewColorWorld (CMWorldRef *cw, CMProfileHandle src, CMProfileHandle dst);`	`pascal CMError NCWNewColorWorld (CMWorldRef *cw, CMProfileRef src, CMProfileRef dst);`
`pascal CMError CMBeginMatching (CMProfileHandle src, CMProfileHandle dst, CMMatchRef *myRef);`	`pascal CMError NCMBeginMatching (CMProfileRef src, CMProfileRef dst, CMMatchRef *myRef);`
`pascal void CMDrawMatchedPicture (PicHandle myPicture, CMProfileHandle dst, Rect *myRect);`	`pascal void NCMDrawMatchedPicture (PicHandle myPicture, CMProfileRef dst, Rect *myRect);`
`pascal CMError CMUseProfileComment (CMProfileHandle profile);`	`pascal CMError NCMUseProfileComment (CMProfileRef prof, unsigned long flags);`

If you are writing a new ColorSync-supportive program, you should always use the new ColorSync Manager functions. The ColorSync 1.0 version of these functions will not be supported indefinitely in new releases of the ColorSync Manager.

CHAPTER 9

What's New

Contents

This section lists the new features available with version 2.5 of the ColorSync Manager, provides links to new and revised material in other sections, and summarizes changes to ColorSync functions, data types, and constants. It also contains a brief summary of features that were added for ColorSync 2.1.

Note
There are no changes to the ColorSync Manager API between version 2.5 and version 2.5.1, so this document is up-to-date for ColorSync 2.5.1. ◆

This section includes the following:

- "New Features in ColorSync Manager Version 2.5" (page 539) lists the features new to version 2.5 and provides links to new and revised material.
- "New and Revised Functions, Data Types, and Constants" (page 544) provides tables that include a brief description of all new and changed functions, data types, and constants, as well as links to more detailed descriptions.
- "New and Revised Code Listings" (page 549) describes new and revised code listings for ColorSync 2.5.
- "New Features in ColorSync Manager Version 2.1" (page 550) lists the features new to ColorSync version 2.1.
- "Other Color Documentation" (page 551) explains where you can find information on earlier versions of ColorSync, and on other color technologies such as the Color Picker Manager.

For related information, see "Revision History" (page 17) and "About This Document" (page 19).

New Features in ColorSync Manager Version 2.5

Version 2.5 of the ColorSync Manager provides many new or enhanced features. The following sections present a brief overview of these features, with links to detailed information in other sections.

New Profile Folder Location

Earlier versions of ColorSync placed the ColorSync Profiles folder inside the Preferences folder. Version 2.5 places the profiles folder at the first level inside the System folder. For backward compatibility, ColorSync may put an alias to the original folder location inside the new profiles folder.

You can now organize profiles by storing them in one level of subfolders within the profiles folder. You can also store aliases to other profiles and profile folders. Profile searching can find profiles in any of these locations.

For an overview of this and related topics, see:

- "Profile Search Locations" (page 55)
- "Where ColorSync Searches for Profiles" (page 56)
- "Where ColorSync Does Not Look for Profiles" (page 57)
- The function description for `NCMGetProfileLocation` (page 233)
- "Optimized Profile Searching" (page 540)

Optimized Profile Searching

ColorSync 2.5 uses a cache file to keep track of currently-installed profiles. A flexible new routine, `CMIterateColorSyncFolder`, takes advantage of the profile cache to perform fast profile searches and provide profile information quickly.

For an overview of this topic, see:

- "The Profile Cache and Optimized Searching" (page 57)

For related information, including sample code that demonstrates optimized searching, see:

- "Performing Optimized Profile Searching" (page 130)
- The function description for `CMIterateColorSyncFolder` (page 304)

Monitor Calibration Framework and Per/Monitor Profiles

ColorSync 2.5 uses the Monitors & Sound control panel to provide a monitor calibration framework and per / monitor profiles. Among the features: you can select a separate profile for each available monitor; you can calibrate monitors and, for each monitor, create one or more color profiles (based on variations in

gamma, white point, and so on); Apple provides a default calibration plug-in, but you can create your own calibration plug-in or use third-party versions; you can choose from any available calibrator to create a monitor profile.

For an overview of these features, see:

- "Monitor Calibration and Profiles" (page 67)

Starting with version 2.5, ColorSync also offers new features for working with displays: you can call ColorSync functions to get or set a monitor profile by AVID; you can use an optional profile tag, which you specify with the `cmVideoCardGammaTag` constant, to provide video card gamma data for a profile—when you call the function `CMSetProfileByAVID` (page 300), it retrieves the video card gamma data and sets the video card.

For sample code that uses the function `CMGetProfileByAVID` (page 300), see:

- "Getting the Profile for the Main Display" (page 100)

For an overview of video card gamma, see:

- "Video Card Gamma" (page 70)

For descriptions of the data types and constants you use with video card data, see:

- "Video Card Gamma" (page 386)
- "Video Card Gamma Constants" (page 421)

Scripting Support

ColorSync 2.5 provides an extensible AppleScript framework that allows users to script many common tasks. Among the features:

- Scriptable operations include setting the system profile, matching an image, and embedding a profile in an image.
- Several sample scripts demonstrate how to automate repetitive tasks.
- The scripting framework uses a plug-in architecture that is fully accessible to third-party scripting plug-ins.

For more information, see:

- "Scripting Support" (page 71)
  - "Scriptable Properties" (page 71)

- □ "Scriptable Operations" (page 71)
- □ "Extending the Scripting Framework" (page 72)
- □ "Sample Scripts" (page 72)

Multiprocessor Support

ColorSync's default Color Matching Module, or CMM, now supports multiple processors for some color matching functions. Multiprocessor support is transparent to your code—it is invoked automatically when the required conditions are met. Matching algorithms take advantage of multiple processors with up to 95% efficiency. As a result, an operation can be performed nearly twice as fast when two processors are available. Performance is scalable.

For more information on this topic, see:

- "Multiprocessor Support" (page 73)
  - □ "When ColorSync Uses Multiple Processors" (page 73)
  - □ "Efficiency of ColorSync's Multiprocessor Support" (page 73)

Sixteen-bit Channel Support

ColorSync's default Color Matching Module now supports 16-bits-per-channel color spaces. The new formats supported are:

- RBG stored in 48 bits per pixel
- CMYK stored in 64 bits per pixel
- Lab stored in 48 bits per pixel

To make use of these new spaces, you specify one of the following constants in the color space field (`space`) of the `CMBitmap` structure:

```
cmRGB48Space
cmCMYK64Space
cmLAB48Space
```

For more information on these constants, see "Color Space Constants With Packing Formats" (page 409).

Flexibility in Choosing CMMs and Default Profiles

The ColorSync control panel, which replaces the ColorSync™ System Profile control panel, now lets you choose a preferred CMM from any CMMs that are present.

Related changes include the following:

- ColorSync previously supported only one default profile—the RGB "System" profile. Users can now use the ColorSync control panel to set default profiles for RGB and CMYK color spaces as well.
- ColorSync provides functions your code can call to get and set default color space profiles for RGB, CMYK, Lab, and XYZ color spaces.

For more information, see:

- "Setting a Preferred CMM" (page 59)
- "Setting Default Profiles" (page 54)
- "Getting and Setting Default Profiles by Color Space" (page 297)

Additional Features

Version 2.5 of the ColorSync Manager ships with the following additional features:

- The Kodak Color Matching Module (available as an install option). Some cross-platform applications use the Kodak Color Management System on the Windows platform. Users working with Macintosh versions of those applications can use the Kodak CMM to ensure consistent output.
- New versions of the ColorSync Photoshop plug-ins that take advantage of ColorSync 2.5. The Filter plug-in is accessible from the Photoshop "Filters" menu, while the Export and Import filters are accessible from the "File" menu.
- Commonly-requested profiles, including SWOP (standard web offset press) and sRGB (standardized RGB monitor).
- Support for an optional video card gamma tag in profiles. For more information, see "Monitor Calibration Framework and Per / Monitor Profiles" (page 540).
- A ColorPicker Manager extension that works with ColorSync 2.x.

- A revised version of the CSDemo application provides sample code that demonstrates how to use many of the new features of ColorSync 2.5.

New and Revised Functions, Data Types, and Constants

The tables in this section provide a brief description of new and changed functions, data types, and constants in ColorSync version 2.5, as well as links to more detailed information.

- Table 9-1 shows new and revised functions.
- Table 9-2 shows new and revised data types.
- Table 9-3 shows new and revised constants.

Table 9-1 New and revised functions in ColorSync 2.5

Function	Version 2.5 Notes
`NCMGetProfileLocation` (page 233)	**New.** Obtains either a profile location structure for a specified profile or the size of the location structure for the profile. Has parameter to specify size of location structure.
`CMGetProfileLocation` (page 234)	**Not recommended.** Use `NCMGetProfileLocation` (page 233) instead.
`CMFlattenProfile` (page 237)	**Changed.** The ColorSync Manager now calls the transfer function directly, without going through the preferred, or any, CMM.
`CMUnflattenProfile` (page 239)	**Changed.** The ColorSync Manager now calls the transfer function directly, without going through the preferred, or any, CMM.
`NCWNewColorWorld` (page 262)	**Changed.** Use of the system profile has changed, as described in "Setting Default Profiles" (page 54). This could affect use of `src` and `dst` parameters.

Table 9-1 New and revised functions in ColorSync 2.5 (continued)

Function	Version 2.5 Notes
`CWConcatColorWorld` (page 265)	**Changed.** Selection of preferred CMM has changed, as described in "Setting a Preferred CMM" (page 59) and "How the ColorSync Manager Selects a CMM" (page 84).
`CWNewLinkProfile` (page 267)	**Changed.** Selection of preferred CMM has changed, as described in "Setting a Preferred CMM" (page 59) and "How the ColorSync Manager Selects a CMM" (page 84).
`CMGetCWInfo` (page 270)	**Changed.** Selection of preferred CMM has changed, as described in "Setting a Preferred CMM" (page 59) and "How the ColorSync Manager Selects a CMM" (page 84).
`CWMatchBitmap` (page 276)	**Changed.** Now supports additional color space constants: `cmGray16Space`, `cmGrayA32Space`, `cmRGB48Space`, `cmCMYK64Space`, and `cmLAB48Space`.
`NCMBeginMatching` (page 285)	**Changed.** Use of the system profile has changed, as described in "Setting Default Profiles" (page 54). This could affect use of `src` and `dst` parameters.
`NCMDrawMatchedPicture` (page 288)	**Changed.** Use of the system profile has changed, as described in "Setting Default Profiles" (page 54). This could affect use of `dst` parameter.
`CMGetPreferredCMM` (page 292)	**New.** Identifies the preferred CMM specified by the ColorSync control panel.
`CMGetSystemProfile` (page 294)	**Changed.** Use of the system profile has changed, as described in "Setting Default Profiles" (page 54).
`CMSetSystemProfile` (page 295)	**Changed.** Use of the system profile has changed, as described in "Setting Default Profiles" (page 54).
`CMGetDefaultProfileBySpace` (page 297)	**New.** Gets the default profile for the specified color space.
`CMSetDefaultProfileBySpace` (page 298)	**New.** Sets the default profile for the specified color space.

Table 9-1 New and revised functions in ColorSync 2.5 (continued)

Function	Version 2.5 Notes
`CMGetProfileByAVID` (page 300)	**New.** Gets the current profile for a monitor.
`CMSetProfileByAVID` (page 300)	**New.** Sets the current profile for a monitor.
`CMGetColorSyncFolderSpec` (page 302)	**Changed.** The name and location of the profile folder changed, as described in "Profile Search Locations" (page 55).
`CMIterateColorSyncFolder` (page 304)	**New.** Provides optimized profile searching by iterating over available profiles.
`CMNewProfileSearch` (page 308)	**Not recommended.** Use `CMIterateColorSyncFolder` (page 304) instead.
`CMUpdateProfileSearch` (page 310)	**Not recommended.** Use `CMIterateColorSyncFolder` (page 304) instead.
`CMDisposeProfileSearch` (page 311)	**Not recommended.** Use `CMIterateColorSyncFolder` (page 304) instead.
`CMSearchGetIndProfile` (page 312)	**Not recommended.** Use `CMIterateColorSyncFolder` (page 304) instead.
`CMSearchGetIndProfileFileSpec` (page 313)	**Not recommended.** Use `CMIterateColorSyncFolder` (page 304) instead.
`MyProfileIterateProc` (page 340)	**New.** Application-defined function that the `CMIterateColorSyncFolder` (page 304) function calls once for each found profile file as it iterates over the available profiles.
`MyColorSyncDataTransfer` (page 342)	**Changed.** The ColorSync Manager calls the function directly, without going through the preferred, or any, CMM

Table 9-2 shows new and revised data types.

Table 9-2 New and revised data types in ColorSync 2.5

Data type	Version 2.5 Notes
`CMProfileIterateProcPtr` (page 365)	**New.** Universal procedure pointer definition for application-defined function you pass to the function `CMIterateColorSyncFolder` (page 304).
`CMProfileIterateData` (page 366)	**New.** Provides concise description of key profile data during iteration over available profiles.
`CMSearchRecord` (page 368)	**Not recommended.** Use `CMProfileIterateData` (page 366) instead.
`CMProfileSearchRef` (page 370)	**Not recommended.** Use `CMProfileIterateData` (page 366) instead.
`CMVideoCardGammaType` (page 386)	**New.** Optional profile tag for video card gamma.
`CMVideoCardGammaTable` (page 387)	**New.** Specifies video card gamma data in table format, based on the specified number of channels, entries per channel, and entry size.
`CMVideoCardGammaFormula` (page 388)	**New.** Specifies video card gamma data as a formula, based on specified actual, minimum, and maximum values for red, blue and green gamma.
`CMVideoCardGamma` (page 389)	**New.** Specifies video gamma data to store with a video gamma profile tag, in either table or formula format.

Table 9-3 shows new and revised constants.

Table 9-3 New and revised constants in ColorSync 2.5

Constants	Version 2.5 Notes
"Color Packing for Color Spaces" (page 404)	**Changed.** The constants `cm48_16ColorPacking` and `cm64_16ColorPacking` were added.
"Abstract Color Space Constants" (page 406)	**Changed.** The constants `cmRGBASpace` and `cmGrayASpace` were moved from "Color Space Constants With Packing Formats" (page 409).
"Color Space Constants With Packing Formats" (page 409)	**Changed.** The constants `cmRGBASpace` and `cmGrayASpace` were moved to "Abstract Color Space Constants" (page 406). The constants `cmGray16Space`, `cmGrayA32Space`, `cmRGB48Space`, `cmCMYK64Space`, and `cmLAB48Space` were added.
"Device Attribute Values for Version 2.x Profiles" (page 418)	**Changed.** The illustration was revised to show the correct ICC definitions for the `deviceAttributes` field in the `CM2Header` **(page 354)** data structure. Unused enums were removed.
"Video Card Gamma Tag" (page 421)	**New.** Specifies the video card gamma tag in a profile.
"Video Card Gamma Tag Type" (page 422)	**New.** Specifies the signature type for a video card gamma profile tag.
"Video Card Gamma Storage Type" (page 422)	**New.** Specifies whether the data in a video card gamma tag is in table or formula format.

New and Revised Code Listings

This section provides a brief description of new and revised code listings.

Table 9-4 New and revised code listings for ColorSync 2.5

Listing	Version 2.5 Notes
Listing 3-1 (page 92), "Determining if ColorSync 2.5 is available"	**Revised.** Checks for version 2.5.
Listing 3-2 (page 97), "Opening a reference to a file-based profile"	**Revised.** Replaced `profLoc.u.file.spec` with `profLoc.u.fileLoc.spec`.
Listing 3-3 (page 98), "Poor man's exception handling macro"	**New.** Provides the `require` macro for simple error handling.
Listing 3-4 (page 100), "Identifying the current system profile"	**Revised.** Returns `CMError` instead of `void`. Uses `require` error-handling macro.
Listing 3-5 (page 101), "Getting the profile for the main display"	**New.** Uses the new `CMGetProfileByAVID` (page 300) function to get the profile for the main display.
Listing 3-6 (page 103), "Matching a picture to a display"	**Revised.** Formerly called both `NCMBeginMatching` (page 285) and `NCMDrawMatchedPicture` (page 288). Now calls only the latter. Uses `require` error-handling macro.
Listing 3-7 (page 110), "Matching the colors of a bitmap using a color world"	**Revised.** Formerly called both `CWMatchPixMap` (page 272) and `CWMatchBitmap` (page 276). Now calls only the latter (fixes bug 1669727). Uses `require` error-handling macro.
Listing 3-8 (page 117), "Embedding a profile by prepending it before its associated picture"	**Revised.** Uses `require` error-handling macro. Disposes of graphics world if necessary on error condition.
Listing 3-9 (page 121), "Counting the number of profiles in a picture"	**Revised.** Renamed bottleneck procedures for clarity.
Listing 3-10 (page 123), "Calling the CMUnflattenProfile function to extract an embedded profile"	**Revised.** Uses `require` error-handling macro. Performs cleanup if necessary on error condition.

Table 9-4 New and revised code listings for ColorSync 2.5 (continued)

Listing	Version 2.5 Notes
Listing 3-13 (page 131), "An iteration function for profile searching with ColorSync 2.5"	**New.** Provides an iteration function for optimized profile searching with the new `MyProfileIterateProc` (page 340) function.
Listing 3-14 (page 133), "A filter function for profile searching prior to ColorSync 2.5"	**New.** Provides a filter function to perform profile searching with the `CMNewProfileSearch` (page 308) function that mimics the optimized searching supported by the `MyProfileIterateProc` (page 340) function.
Listing 3-15 (page 135), "Optimized profile searching compatible with previous versions of ColorSync"	**New.** Provides sample code that performs an optimized profile search if ColorSync 2.5 is available, but provides a compatible (though not optimized) search if it is not.
Listing 4-1 (page 208), "Modifying a profile header's quality flag and setting the rendering intent"	**Revised.** Additional comments.

New Features in ColorSync Manager Version 2.1

This section describes new features added to version 2.1 of the ColorSync Manager. These features are documented throughout this document. If you are interested in documentation that covers only version 2.1, see "Other Color Documentation" (page 551).

- **procedure-based profiles:** You can specify your own profile-access procedure that ColorSync will call when the profile is created, initialized, opened, read, updated, or closed.
- **support for named color spaces:** The ColorSync Manager provides data structures and routines for working with named color spaces.
- **profile identifiers:** The ColorSync Manager defines the *profile identifier,* an abbreviated data structure that identifies, and possibly modifies, a profile in memory or on disk. An embedded profile identifier requires much less space than an entire profile.

- **additional PostScript support:** Postscript Level 2 now supports up to four-component color spaces. This allows the creation of device-independent color space definitions that can support calibrated CMYK spaces and provide more flexible support for calibrated scanner and monitor spaces.
- **color conversion without components:** Color conversion routines are an integral part of the ColorSync Manager and are no longer implemented as a separate component.
- **support for new bitmap formats:** The ColorSync Manager supports bitmap formats for many additional color spaces, including 24-bit RGB, 32-bit RGB with an alpha last channel, and 24-bit Lab.
- **profile reference counts:** The ColorSync Manager maintains an internal reference count for each profile reference so that it can efficiently free private memory associated with that profile reference once it is no longer in use.
- **profile changed flag:** The ColorSync Manager maintains a flag that indicates whether the content of a profile has changed.
- **speed and accuracy enhancements:** You can use a "lookup only" flag to skip interpolation and speed up runtime color conversion. You can also disable gamut checking to speed up initialization and reduce profile size.
- **revised sample application:** A revised version of the CSDemo application provides sample code that demonstrates how to use many of the new features of ColorSync 2.1.

For a guide to the new features in version 2.1 of the ColorSync Manager, see the document *What's New in Advanced Color Imaging on the Mac OS*, available with the ColorSync 2.1 SDK.

Other Color Documentation

For documentation that covers only features available with ColorSync Manager 2.1 and earlier versions, see *Advanced Color Imaging on the Mac OS Revised for ColorSync 2.1* and *Advanced Color Imaging Reference Revised for ColorSync 2.1*. These documents also describe the Color Picker Manager (Version 2.0), Color Manager, and Palette Manager.

An earlier, paper version of Advanced Color Imaging on the Mac OS, covering ColorSync through version 2.0, was published by Addison-Wesley Publishing Company. It has the catalog number ISBN 0-201-48949-X.

Technote 1100, "Color Picker 2.1" describes version 2.1 of the Color Picker Manager. Note that Color Picker Manager version 2.1 works with ColorSync Manager versions 2.0 and greater.

The electronic documents described here are available at <http://developer.apple.com/>.

Glossary

absolute colorimetric matching A **rendering intent** that is used for a device-independent color space in which the result is an idealized print viewed on a perfect paper having a large dynamic range and color gamut. In reality, paper cannot reproduce densities less than a particular minimum density.

abstract profile A profile that allows applications to perform special color effects independent of the devices on which the effects are rendered. Abstract space profiles perform affects between two PCS color spaces. See also **profile**, **color space profile**, **device profile**, and **named color space profile**.

additive color theory The process of mixing red, green, and blue lights, which are each approximately one-third of the visible spectrum. Additive color theory explains how red, green, and blue light can be added to make white light.

animated color A color that the Palette Manager uses for special animation effects. Animated colors work only on devices that have a color table; that is, they do not work on direct devices.

application-owned dialog box A dialog box, created by an application, for presenting a color picker.

arbitrated CMM A CMM selected by the ColorSync Manager from the available source and destination profiles to perform a specified operation. Compare with **default CMM**, **key CMM**, and **preferred CMM**.

brightness A term in color theory used to describe differences in the intensity of light reflected from or transmitted by a color image. Also known as *value*. 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.**

calibration The process of setting a device's parameters according to its factory standards. Compare with **characterization**.

characterization The process of learning the color character of a monitor so that a profile can be created to describe it. Compare with **calibration**.

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.**

CMS See **color management system.**

CMY space A color space in which cyan, magenta, and yellow are the three primary colors. Used for some low-end printers.

CMYK space A color space in which cyan, magenta, and yellow are the three primary colors. Unlike CMY space, the CMYK color space models the way inks or dyes are applied to paper in printing, in which black ink is overprinted in darker areas to give a better appearance.

color channel See **color component.**

color component A dimension of a color value expressed as a numeric value. For the ColorSync Manager, depending on the color space, a color value may consist of one, two, three, four, or eight components, also referred to as channels.

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

color conversion The process of converting colors from one color space to another in a mathematically reversible way.

color gamut See **gamut.**

color management module (CMM) 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. A CMM interprets the information stored in a **profile**.

color management system (CMS) Software that provides consistent color across peripheral devices and across operating-system platforms by converting colors from the color space of one device to the color space of another device.

Color Manager A set of system software functions that supply color-selection support for Color QuickDraw. Most applications never need to call the Color Manager directly.

color matching The process of adjusting or *matching* converted colors appropriately to achieve maximum similarity from the gamut of one color space to the other. Color matching always involves color conversion, whereas color conversion may not entail color matching. Matching also involves devices, and may not be reversible.

color space An environment in which colors are represented, ordered, compared, or computed. 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 convert color values between a PCS and a non-device color space (such as L*a*b to or from L*u*v, or XYZ to or from Yxy), as necessary for color matching. Color space profiles provide a convenient means for CMMs to convert between

different nondevice profiles. See also **profile**, **abstract profile**, **device profile**, and **named color space profile**.

ColorSync A platform-independent color management system from Apple Computer that provides services for fast, consistent, and accurate desktop color calibration, proofing, and reproduction.

ColorSync Manager A set of system software functions (or API) that provide device-independent color-matching and color conversion services for device drivers and applications; the implementation of ColorSync for the Mac OS.

color value A complete specification of a color in a given color space. Depending on the color space used, one, two, three, or four color-component values combine to make a color value.

courteous color A color that accepts whatever value the Color Manager determines is the closest match available in the color table. Compare **tolerant color**.

default CMM A CMM supplied with ColorSync that supports all the required and optional functions defined by the ColorSync Manager, and is therefore a suitable CMM of last resort when a specified CMM is not available or cannot perform a specified operation. Compare with **arbitrated CMM**, **key CMM**, and **preferred CMM**.

default system profile The system profile that serves as the default display profile, as well as the default profile for color conversion and matching operations for which no profile is specified. Unless the ColorSync Manager control panel is used to select a different profile, which must be an RGB profile, ColorSync uses the Apple 13-inch color display. See “Setting Default Profiles” (page 54) for changes with ColorSync 2.5.

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 link profile A profile that represents a one-way link or connection between devices. It can be created from a set of multiple profiles, such as various device profiles associated with the creation and editing of an image. It does not represent any device model, nor can it be embedded into images.

device profile A structure that contains the color characteristics of a given device in a particular state. See also **profile**, **abstract profile**, **color space profile**, and **named color space profile**.

explicit 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.

general purpose color-matching function One that uses a color world to characterize how to perform color-matching. See also **QuickDraw-specific color-matching function**.

gray space A color space that typically has a single component, ranging from black to white.

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.*

HSB space A transformation of RGB space that is analogous to HSV space. HSB stands for *hue, saturation, and brightness.* where brightness is synonymous with *value* in HSV space. Compare with **HSV** and **HLS** space.

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.* Compare with **HSB** and **HLS** space.

hue The name of the color that places the color in its correct position in the spectrum. For example, if a color is described as blue, it is distinguished from yellow, red, green, or other colors. Compare with **brightness** and **saturation.**

indexed color space The color space used when drawing with indirectly specified colors.

inhibited color A color that is prevented from appearing on particular screens. Colors can be specifically inhibited on a 2-bit, 4-bit, and 8-bit color or grayscale screen.

interchange color space Device-independent color spaces that are used for the interchange of color data from the native color space of one device to the native color space of another device. Compare **profile connection space (PCS)**.

International Color Consortium (ICC) International color organization that publishes the *International Color Consortium Profile Format Specification*. The ICC Web site is at <http://www.color.org/>.

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.

key CMM In a series of CMMs specified by a `CMConcatProfileSet` (page 384) structure, the CMM indicated by the zero-based value of the structures `keyIndex` field. Compare with **arbitrated CMM**, **default CMM**, and **preferred CMM**.

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.

metamerism The capability of the human eye to perceive two or more visible spectra as the same color. See also **trichromatic color vision**.

named color space A color space in which each color has a name; colors are generally ordered so that each has an equal perceived distance from its neighbors in the color space.

named color space profile A profile that contains data for a list of named colors. The profile specifies a device color value and the corresponding CIE value for each color in the list. See also **profile**, **abstract profile**, **color space profile**, **device profile**, and **named color space profile**.

perceptual matching A **rendering intent** in which all the colors of a given gamut may be scaled to fit within another gamut. The colors maintain their relative positions, so the relationship between colors is maintained.

pixel value A number used by system software and a graphics device to represent a color. The translation from the color that an application specifies in an `RGBColor` data structure to a pixel value is performed at the time the application draws the color. The process differs for indexed and direct devices.

preferred CMM Starting with ColorSync 2.5, a user-selected CMM, chosen with the ColorSync control panel, that is used for all color checking and matching operations that the CMM can handle. Compare with **arbitrated CMM**, **default CMM**, and **key CMM**.

profile A structure that provides a means of defining the color characteristics of a given device in a particular state. A profile 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 profile classes. Each profile class must include a different required set of information, but all of these classes follow the same format. See also **abstract profile**, **color space profile**, **device profile**, and **named color space profile**.

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).

profile connection space (PCS) A device-independent color space used as an intermediate when converting from one device-dependent color space to another. Profile connection spaces are typically based on spaces derived from CIE color spaces. Compare **interchange color space**.

profile identifier An abbreviated data structure that uniquely identifies, and possibly modifies, a profile in memory or on disk.

profile reference A unique reference to a profile, returned by ColorSync and based on a private data structure; the profile reference is the means by which your application identifies a profile and gains access to it.

QuickDraw-specific color-matching function One that uses QuickDraw to provide images showing consistent colors across displays. See also **general purpose color-matching function**.

reference white point A specific definition of what is considered white light represented in terms of XYZ space and usually based on the whitest light that can be generated by a given device.

relative colorimetric matching A **rendering intent** in which the colors that fall within the gamuts of both devices are left unchanged. Relative colorimetric matching allows some colors in both images to be exactly the same, which is useful when colors must match quantitatively. A disadvantage of relative colorimetric matching is that many colors may map to a single color resulting in tone compression.

rendering intent The approach taken when a CMM maps or translates the colors of an image to the color gamut of a destination device. Each profile supports four different rendering intents: **perceptual matching, relative colorimetric matching, saturation matching,** and **absolute colorimetric matching.**

RGB space A three-dimensional color space whose components are the red, green, and blue intensities that make up a given color. Compare **sRGB space**.

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.

sRGB space A three-dimensional color space that attempts to create a standard RGB space based on a calibrated, colorimetric RGB definition that calls for a gamma of 2.2, a white point of 6500 degrees K, and P-22 phosphors. Compare **RGB space**.

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 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.

trichromatic color vision The capacity of the human eye to responds equally to two or more sets of stimuli having different visible spectra. See also **metamerism**.

trichromatic color reproduction The process of inducing the illusion of a color using various amounts of only three primary colors: either red, green, and blue mixed additively or cyan, magenta, and yellow mixed subtractively.

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.

undercolor removal (UCR) The removal of excessive color densities when printing an image.

value See **brightness.**

XYZ color space The fundamental CIE-based color space that allows colors to be expressed as a mixture of the three **tristimulus values** X, Y, and Z.

Yxy color space A color space belonging to the XYZ base family that expresses the XYZ values in terms of x and y chromaticity coordinates, somewhat analogous to the hue and saturation coordinates of HSV space.

Index

D

E

F

G

H

I, J, K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y, Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software.

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
Steve Evangelou, Tony Francis, Michael Kline, Judy Melanson

DEVELOPMENTAL EDITORS
Jeanne Woodward, Wendy Krafft, Beverly McGuire

ILLUSTRATORS
Dave Arrigoni, Bruce Lee, Ruth Anderson, Lisa Hymel

PRODUCTION EDITORS
GLen Frank, Gerri Gray, Pat Christenson, Alan Morgenegg

PROJECT MANAGER
Tony Francis

LEAD WRITER
Steve Evangelou

LEAD EDITOR
Jeanne Woodward

LEAD ILLUSTRATOR
Bruce Lee

Special thanks to Eric Broadbent, John Calhoun, Anil Gursahani, David Hayward, Ingrid Kelly, Gabriel Marcu, Roger Siminoff, and Steve Swen.

Acknowledgment to Wei-Ling Chu, Richard Collyer, Rob Dearborn, John Gnaegy, Donna Lee, Edgar Lee, John Myer, Don Moccia, Tom Mohr, Han Nguyen, Forrest Tanaka, David Van Brink, Josh Weisberg, and John Wang.