Coloratura™
Programmer's Guide

Document Number 007-3442-001

CONTRIBUTORS

Written by Leif Wennerberg
Illustrated by Martha Levine
Production by Allen Clardy
Engineering contributions by Todd Newman
St. Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA.
Disk Thrower image courtesy of Xavier Berenguer, Animatica.

Coloratura™ Programmer's Guide
Document Number 007-3442-001

Contents

List of Figures

About This Guide

The Silicon Graphics Coloratura™ color management system (CMS) is a C library, available as a dynamically shared object, that provides an application program interface to color manipulation modules and ICC device characterization profiles. The interface allows you to use profiles to develop transformations between device-specific interpretations of color so that the appearances of color images are largely independent of input or output device characteristics. You can also use the Coloratura CMS along with appropriate device data to characterize a device.

Audience

To use the Coloratura CMS effectively, you should be a C programmer familiar with color image file formats. Knowledge of the Image Format Library, which is part of the ImageVision Library™, will simplify writing code to access image files. You will benefit from familiarity with terms from color management, such as gamut mapping or CIEXYZ, and from familiarity with the *ICC Profile Format Specification*.

What This Guide Contains

Chapter 1, "Color Management," briefly describes the color management problem and the major parts of the Coloratura CMS: profiles, tags, transforms, and color manipulation modules. It provides pseudocode to illustrate elementary components of the Coloratura CMS and describes the operations of a sample color manipulation application.

Chapter 2, "Using the Coloratura CMS Programming Environment," shows you how to link your application to the Coloratura library and begin using the color management system. It also briefly describes commands and sample programs included in the Coloratura CMS.

Chapter 3, “Profile Management,” describes how to find profiles, create and modify profiles, import profiles embedded in image files, and export profiles so that they can be embedded in image files.

Chapter 4, “Tag Management,” describes how to access the fundamental data in profiles— tags and headers, read tags to determine if a profile is appropriate to your needs, and create or modify tags to change the contents of a profile.

Chapter 5, “Transform Management,” describes how to build transforms from profiles and apply transforms to pixel data.

Chapter 6, “Color Manipulation Module Management,” describes how to determine the default CMM and examine available CMMs.

Appendix A, “Summary of Functions and Data Structures,” summarizes Coloratura data structures and functions.

Appendix B, “Listing of the Application cocoifl,” presents a sample color conversion application.

The Glossary defines a few terms from color management.

Related Materials

You may find the following web sites useful for getting information on device-independent color and general topics of color processing:

- The International Color Consortium’s home page: http://www.color.org. The *ICC Profile Format Specification* is available from this site.
- The Colour Science and Technology page has many useful links: http://www.ziggy.derby.ac.uk/web/colour.html.

The *Colortron User Manual* by Fred Bunting, published by Lightsource Computer Images, Inc., 1994, has nice introductory discussions of color science and color management, and an extensive bibliography.

ImageVision Library ™ Programming Guide, document Number 007-1387-050, available through IRIS Insight™ and at *http://techpubs.engr.sgi.com/lib/makepage.cgi?007-1387-050*, documents the Image Format Library.

IFL(3). The reference page introduces the Image Format Library, on which the ImageVision library is built. IFL is a convenient library with C and C++ bindings for manipulating graphics files in Coloratura applications.

Impressario Programming Guide, document number 007-1633-050, available through IRIS Insight and at *http://techpubs.engr.sgi.com/lib/makepage.cgi?007-1633-050*. An appendix discusses color management within the Impressario printer and scanner environment.

MIPS Compiling and Performance Tuning Guide, document Number 008-2479-001, *http://techpubs.engr.sgi.com/lib/makepage.cgi?007-2479-001*, discusses dynamic shared objects (DSOs).

Xlib Programming Manual for Version 11 of the X Window System by Adrian Nye, O'Reilly & Associates, Inc., 1992, discusses X-resource files.

Conventions Used in This Guide

These type conventions and symbols are used in this guide:

Bold	Function names, language keywords, and data types
Italics	Filenames, glossary entries, manual/book titles, new terms, and program variables
`fixed width`	Code examples
ALL CAPS	Environment variables and defined constants
“”	(Double quotation marks) References in text to document section titles
()	(Parentheses) Follow function names—surround function arguments.

Chapter 1

Color Management

This chapter briefly introduces you to color management, the main components of the Coloratura color management system, the basic structure of a Coloratura application, and outlines a color conversion application.

These are the topics discussed in this chapter:

The Color Management Problem

This section briefly surveys the main problems of color management.

How digital color devices interpret colors varies from device to device. Even if two devices appear identical, detail differences can yield different images from the same data. You could say that no two devices "see" color in exactly the same way. Often the differences can dramatically affect image appearance if you transfer a color image from one device to another without a color management tool. These effects occur largely because the "recipe" for a particular color on one device rarely gives the same color on another device, that is, the specifications of which primary colors to use and how much of each do not agree in any obvious way.

A device's *color space*, that is, its set of colors and how they are parameterized, is a useful way of describing its interpretation of color. A device's color space results from a combination of design choices and physical constraints. Color spaces can differ in two main ways:

- primary colors
- color gamut

Device *primary colors* determine the method of parameterizing colors, and can differ in two fundamental ways:

- The set of primaries used to describe colors can be obviously different. For example, a *RGB* device describes colors with combinations of red, green, and blue; but a *CMYK* device uses cyan, magenta, yellow, and black.
- Spectral contents of primaries can differ, even if two devices use nominally the same primary colors; one device's red is not necessarily the same as another device's.

A central task of color management is maintaining a consistent method for translating between differing sets of color primaries.

The *color gamut* of a device is the set of possible colors that the device can produce. Clearly, these sets can vary. A typical practical problem for image development is the set of colors available on a color monitor is larger than that available on a printer.

How the Coloratura Color Management System Helps

The Coloratura color management system provides a framework for IRIX™ applications to reconcile device color spaces and to communicate colors accurately; that is it defines a *color management system (CMS)*. The Coloratura CMS provides an API for access to device characterization profiles (ICC profiles) and for control of color transformation computations. For example, you can develop an application that will match your printer output to a scanned image, or simulate on your monitor the output of your printer.

In general terms, the elements required for a CMS are the following:

- A standard file format to characterize device color characteristics.
- A standard, device-independent color space with which to describe each device's color space.

- Procedures to manipulate the standard descriptive information.
- Tools to transform color image data.

The next few sections elaborate on how the Coloratura CMS provides these elements.

Profiles: Device-Independent Color Spaces

A device *profile* is a file in a standard format that characterizes a device's color interpretation. The characterization translates the device's color space into either of two device-independent color spaces, CIELAB or CIEXYZ, developed by the *CIE* (Commission Internationale d'Eclarage, that is, International Committee of Illumination) to describe human color perception. The profile format used by the Coloratura CMS is described by the *International Color Consortium Color Profile Format Specification*, which can be found at *http://www.color.org*.

The device-independent color space is called a *profile connection space (PCS)*; it allows you to transform data characterized by one *ICC* profile to data characterized by another, with a minimum distortion of color information. The PCS provides a common language to describe how colors appear to devices.

The data in a profile is organized into individual blocks, *tag* data. Tag data detail specific device characteristics, the information needed to transform image data. For example, profiles can supply parameters indicating a preferred color conversion algorithm. Another example of tag data: the *ICC Profile Format Specification* includes a tag to characterize *rendering intent*, which describes how to reconcile differing gamuts when image data are converted from one color space to another. This tag is useful because the ideal process of transforming from one color space to another by simply translating a given color from one color space to another rarely occurs; typically the color gamuts of input and output devices differ. In any case, often with a change in medium or lighting conditions you want to modify color content. For more information on rendering intents, see the *ICC Profile Format Specification*.

The Coloratura CMS provides tools to examine profiles on disk, read them, write them, delete them, and edit their tag data. You can also read and write profile data embedded in image files. See Chapter 3, "Profile Management," and Chapter 4, "Tag Management."

Transforms: Color Processing from Profiles

You combine profiles into a *transform* to determine the flow of color information. For example, suppose you want to drive a printer, which typically uses CMYK, with data taken from a display screen, which uses RGB. You combine the input profile for the screen and the output profile for the printer into a transform. Then you apply this transform to each buffer full of data from the screen to create a buffer full of data for the printer.

As another example, to simulate on your monitor the output of a printer using input from your scanner, you create a transform using a sequence of profiles for these devices in the following order: scanner to printer to monitor.

The Coloratura CMS provides tools to build transforms from profiles, save them, examine the gamut of transformed image data, and apply transforms to image data. See Chapter 5, "Transform Management." To apply the transforms, you need to select a color manipulation module, discussed in the next section.

Color Manipulation Modules: Algorithms from Transforms

A *color manipulation module* (*CMM)* uses the information in a transform to define a color conversion algorithm. Profiles can be more or less exact in how they characterize the relationship between a device's color space and the profile connection space, and CMMs can be distinguished by how they interpret sparse profile data.

The Coloratura CMS allows you to use several CMMs, which are implemented as dynamic shared objects (DSOs). The Coloratura framework serves as a dispatcher between your application and available CMMs, supplying a unified API to them. A default CMM ships with the Coloratura CMS. Not all CMMs necessarily support all possible profiles, however the default CMM does.

A CMM is likely to achieve the best quality with profiles generated specifically for that CMM. Profiles can identify a "preferred" CMM, which best interprets the profile, as well as any specified rendering intent. This information guides the development of the transformation algorithm. When performing a transformation, a CMM uses the rendering intent to make appropriate color adjustments. This can be particularly useful for generating accurate image simulations.

The Coloratura CMS provides tools to examine the available CMMs on your system. See Chapter 6, "Color Manipulation Module Management."

Schematics of Program Architecture and Data Flow

You Coloratura application does not interact directly with profile data on disk, nor with CMMs. The Coloratura CMS mediates the interaction by establishing the relationships between your application, ICC profiles on disk and CMMs that are illustrated in Figure 1-1. Also shown in Figure 1-1 are the internal data structures the Coloratura CMS uses to manage profiles, tags, and transforms. Blocks that are adjacent in the figure indicate direct communication between the objects.

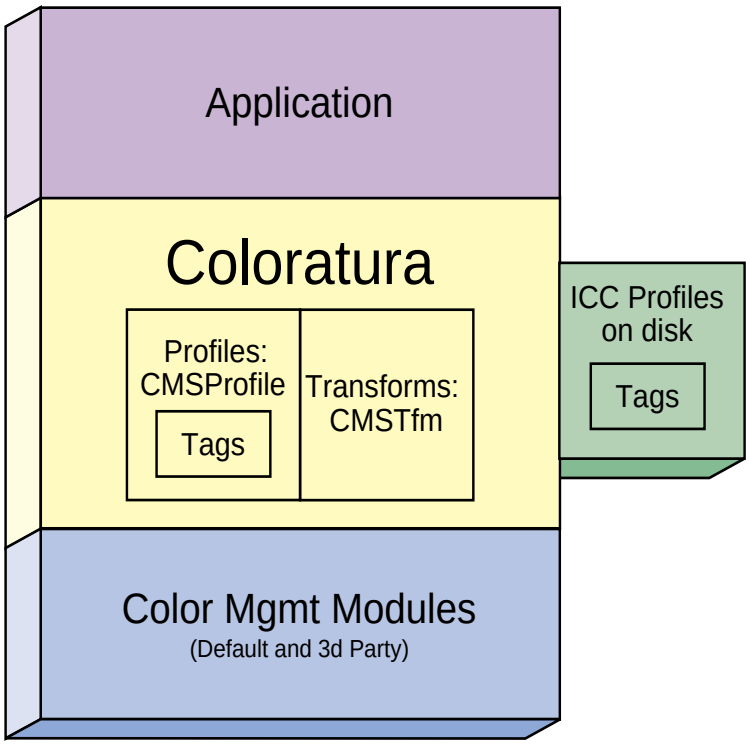

Figure 1-1 Components of a Color-Management Application

Figure 1-2 illustrates the data flow for a typical application. It schematically compares the flow to that of an application that does not use color management, but, in this example, uses a "1-minus" conversion from RGB to CMYK. Two input and output paths are shown for the Coloratura application, corresponding to two likely sources and destinations for data: devices or image files. Ironically, color managed images are unavailable for this document, so the color distortions shown in Figure 1-2 can only be rough illustrations of the effects.

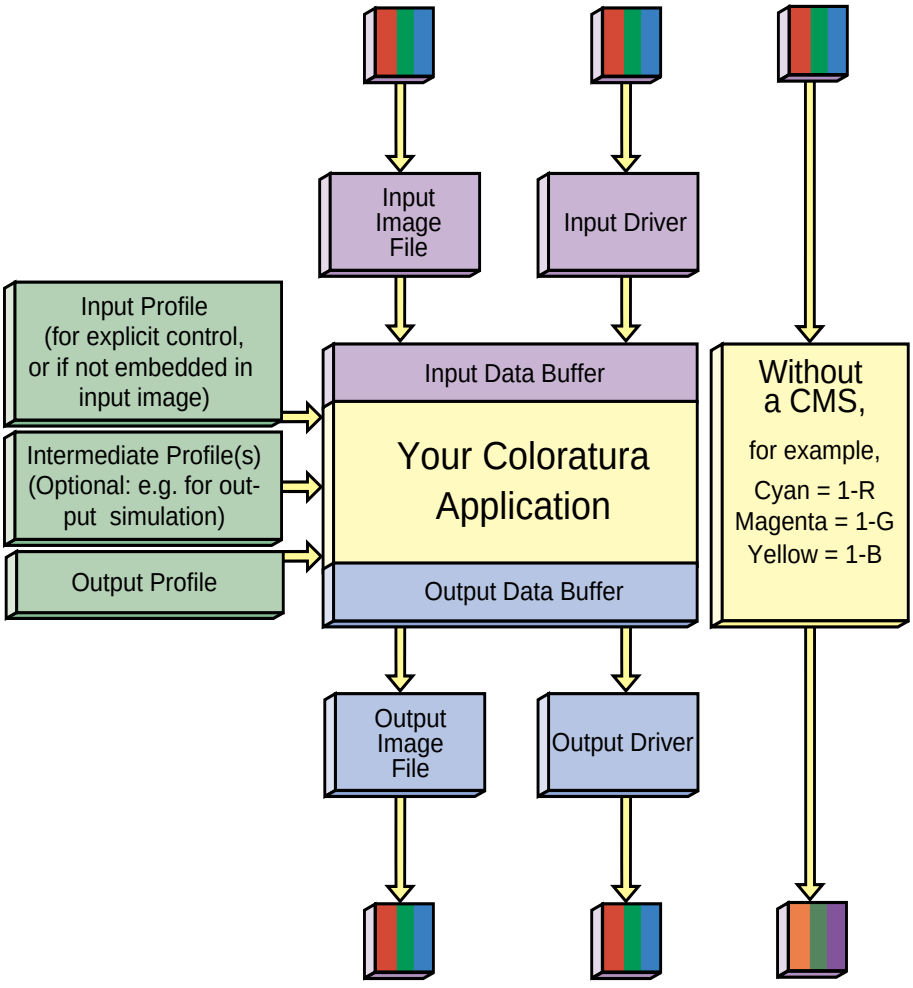

Figure 1-2 Data Flow for a Typical Color-Management Application

Pseudocode Example for a Color Conversion

The following lines of code illustrate the basic programming elements of the Coloratura CMS. The example is not working code, but provides the essential components of a program for driving a CMYK printer with data taken from a workstation's monitor. The next section outlines in more detail the procedures of a color management application, and Appendix B, "Listing of the Application cocoifl," presents the code of a working program that implements these operations.

Note: For the Coloratura CMS, function names all start with "cms", type names all start with "CMS", and library constant names all start with "CMS_".

- First, allocate data structures and open the Coloratura CMS. The header file *cms.h* defines the data types and specifies the function prototypes.

```
/* Typedefs for some of the variables used in the pseudocode */
CMSContext      ctxt;
CMSProfile      profSource, profDest, profiles[2];
CMSParameter    param;
char           *profSpecDest;
CMSTransform    tfm;

cmsOpen( &ctxt );
```

- Next, find the necessary device profiles. The following lines get the source, that is the monitor's, profile from the settings and assumes you know the destination profile.

```
cmsOpenProfile( ctxt, CMS_DEFAULT_MONITOR, &profSource );
cmsOpenProfile( ctxt, profSpecDest, &profDest );
```

- The first profile added to a transform is assumed to be the input device; the last profile, the output device. For this example, we need only two profiles:

```
profiles[0] = profSource;
profiles[1] = profDest;
cmsCreateTfm( ctxt, 2, profiles, CMS_USE_DEFAULT_CMM, &tfm )
```

- Finally, for each buffer full of pixels from the screen, apply the transform that corrects for the differing device color mappings and converts the pixels from the display screen's RGB space into the printer's CMYK space. The example skips over allocating data storage for the input and output buffers. Note that buffers are usually of different sizes.

```
while (GetPixelsFromScreen( &pbufIn.data) != NO_MORE_PIXELS) {
    cmsTfmApply( ctxt, tfm, &pbufIn, &pbufOut );
    WritePixelsToPrinter( pbufOut.data );
}
```

Example Outline of a Color Conversion Program

This section outlines the procedures in a color conversion application that is more realistic than the example in the last section. The procedures introduce the main Coloratura data structures and functions, and the discussion directs you to details in the rest of this guide.

The data flow of the program is the same as that outlined in Figure 1-2, but without any intermediate profiles. C++ code for the program, called cocoifl, is in */usr/cms/examples/* and is listed in Appendix B, "Listing of the Application cocoifl."

Below are the program steps described in this section. Some details, error handling, for example, have been left out; look at the source code to see how these details are handled.

- "Loading Header Files" on page 9
- "Declaring Variables" on page 9
- "Opening the Coloratura CMS, the Input Image File, and the Output Profile" on page 9
- "Preparing the Output Pixel Buffer and Open an Output Image File" on page 10
- "Preparing the Output Pixel Buffer and Open an Output Image File" on page 10
- "Selecting an Input Profile" on page 10
- "Creating a Transform and Initializing Buffers" on page 10
- "Embedding the Output Profile in the Output Image File" on page 11
- "Transforming Pixel Data and Cleaning Up" on page 11

Loading Header Files

The program begins by including standard C++ library header files, Image Format Library (see IFL(3)) headers, and the two files that are sufficient to use the Coloratura CMS, *ic.h* and *cms.h.* These latter two declare Coloratura functions, and ICC profile structures.

Declaring Variables

The program declares local variables and the Coloratura data structures it needs:

- **CMSContext** appears in every Coloratura function call. It is discussed in "Working State of the Coloratura CMS: CMSContext" on page 15.
- **CMSProfile**, points to an opaque data structure that holds ICC profile data. It is discussed in "Data Structure for Profiles: CMSProfile" on page 26.
- **CMSPixelBuffer** holds pixel data used in color transformations. It is discussed in "Data Structure for Pixels: CMSPixelBuffer" on page 47.
- **CMSTfm**, points to an opaque data structure that holds a color transformation. It is discussed in "Data Structure for Transforms: CMSTfm" on page 47.
- **icHeader**, declared in the file *ic.h* and discussed in the *ICC Profile Format Specification*.
- **iflColorModel**, and **iflSize** are IFL objects. See the *ImageVision Library™ Programming Guide* and the reference page IFL(3).

Opening the Coloratura CMS, the Input Image File, and the Output Profile

In any Coloratura application, the first call is to **cmsOpen()**, discussed in the section "Initializing the Coloratura CMS: cmsOpen()" on page 15. The program then manages the command-line arguments and uses an IFL function to open the input image file.

With a call to **cmsOpenProfile()**, the program then brings an output ICC profile from disk into the Coloratura data structure identified by a **CMSProfile**, discussed in "Loading Profile Data: cmsOpenProfile()" on page 26. From the open profile, cocoifl gets ICC header information by using the function **cmsGetProfileHeader()**, discussed in "Getting Open-Profile Header Information: cmsGetProfileHeader()" on page 30.

Preparing the Output Pixel Buffer and Open an Output Image File

The program places some of the output profile header information into the output **CMSPixelBuffer**, and sets the IFL color model. It uses an IFL function to creates the output image file.

Selecting an Input Profile

The program has a hierarchy of possible input profile sources:

- a profile designated to override profile information that might be embedded in the source image
- an profile embedded in the source
- a profile designated to be used if no embedded profile is found
- a default profile

If no overriding profile is specified, cocoifl determines if an ICC profile is embedded in the source image and uses an IFL function to read it from the image file. You call **cmsImportProfile()** to convert the profile data to the Coloratura CMS's internal data structure. This function is discussed in "Importing an ICC Profile from a Buffer: cmsImportProfile()" on page 34.

Without explicit profile information, cocoifl uses one of two default profiles, depending on whether the image data are RGB or CMYK data. The default profiles are specified by the constants CMS_DEFAULT_MONITOR and CMS_DEFAULT_CMYK. See the section "Loading Profile Data: cmsOpenProfile()" on page 26.

Creating a Transform and Initializing Buffers

With a source and destination profile, the program creates a transform by calling **cmsCreateTfm()**, discussed in "Creating a Transform: cmsCreateTfm()" on page 49.

The **iflRGBPalette** color map creates a special case, because of the unique way it handles color information; the pixel data are indices to a color palette. For this case, the program cocoifl sets up the input and output **CMSPixelBuffer**s, and transforms the data.

For all other color maps, pixel data are coordinates in device color spaces. The program only initialize the output **CMSPixelBuffer**, and sets up the input **CMSPixelBuffer** with header information and pixel data. The program applies the color transformation later.

If no input profile was found, or if the input data use the *iflRGBPalette* color map and, so, have already been transformed, the program passes the input directly to the output buffer.

Embedding the Output Profile in the Output Image File

The program calls **cmsExportProfile()** to create an ICC formatted data structure, and embeds the output profile in the output image with a call to an IFL function. The function **cmsExportProfile()** is discussed in "Creating an ICC Profile in a Buffer: cmsExportProfile()" on page 33.

In general, when you are through with data structures created by the Coloratura CMS, you free the allocated memory. After embedding the profile in the output image, cocoifl calls **cmsFreeProfileExport()** to clear the data created by **cmsExportProfile()**. **cmsFreeProfileExport()** is discussed in "Deleting an ICC Profile Buffer: cmsFreeProfileExport()" on page 33

Transforming Pixel Data and Cleaning Up

The program uses IFL calls to get image-file data one tile at a time, reads data from the input file to the input **CMSPixelBuffer**, and, with a call to **cmsApplyTfm()**, applies the transform that was created by **cmsCreateTfm()** (see "Applying a Transform: cmsApplyTfm()" on page 50).

Finally the program frees memory it has allocated.

Chapter 2

Using the Coloratura CMS Programming Environment

This chapter discusses the basic procedures that apply to the development of any Coloratura application. It also discusses useful tools that are included in the library: commands and sample code. These are the topics discussed:

Compiling With the Coloratura CMS

To use the Coloratura library, which is implemented as a dynamically shared object, include the two header files *cms.h* and *ic.h* in your source code and compile your program with the switch **-lcms**.

The file *cms.h* defines data structures, function prototypes and parameters specific to the Coloratura CMS. You do not need to declare Coloratura data structures before you call Coloratura functions; this is done by including the file *cms.h* in your source code.

The file *ic.h* establishes data structures for the *ICC Color Profile Format Specification*.

Managing Memory

Coloratura applications use one of two approaches to managing memory, depending on whether a given data object has a fixed size.

- If a data object does not have a fixed size:
  - Use Coloratura functions to create the data storage and set a pointer value.
  - When you no longer need the data, call a Coloratura function to free the data. For each type of data, there is a specific function. For example, **cmsGetCmmList()** returns a variable-length list of available CMMs. Your application passes to **cmsGetCmmList()** the address of a pointer to the list, and frees the list by calling **cmsFreeCmmList()**.
- If a data object has a fixed size, you need no special Coloratura function to free storage; you can use standard ANSI C memory management functions.

If your application requires custom memory management, layer the management on top of the three standard ANSI C functions that are used by the Coloratura CMS: **malloc()**, **free()**, and **realloc()**. Using two different memory allocation schemes can cause problems such as memory leaks, data corruption, or core dumps. If you use other functions, check that all the memory allocators work together.

Error Messages

Every Coloratura function returns CMS_SUCCESS, if there are no errors.

Each Coloratura function has a set of specific error codes that it returns if there are errors. Before proceeding after a function call, you should always check the value returned.

The Coloratura CMS provides a standard message catalog in */usr/lib/locale/C/LC_MESSAGES*. You can call the function **pfmt()** to interpret the error codes returned by Coloratura functions (see the reference pages pfmt(1) and pfmt(3C) for more details). Because the message catalog is accessed by line numbers, do not modify the catalog; if you need more error messages, create another catalog.

Establishing Program Access to the Coloratura CMS

The Coloratura CMS maintains a certain amount of bookkeeping information that all Coloratura functions use. The information is stored in the data structure **CMSContext**, which is initialized by **cmsOpen()**. You must call **cmsOpen()** before you call any other Coloratura functions. When you are through with the CMS, call **cmsClose()** to free storage allocated by **cmsOpen()**.

Working State of the Coloratura CMS: CMSContext

The pointer **CMSContext** refers to an opaque data structure that stores the internal working state of the Coloratura CMS and is used in all function calls. The data type declaration is:

```
typedef struct _CMSContext *CMSContext;
```

Initializing the Coloratura CMS: cmsOpen()

The function **cmsOpen()** establishes a **CMSContext** that is an argument in all subsequent Coloratura calls. Call this function before you call any other Coloratura function. You pass the address of a **CMSContext** and **cmsOpen()** initializes it appropriately.

The function **cmsOpen()** searches the system to find out which CMMs are available and stores the information in the **CMSContext**. Your application cannot use any CMM that you add after a call to **cmsOpen()**.

- This is the prototype for **cmsOpen()**:

  ```
  int32  cmsOpen(CMSContext *pctxt);
  ```

- This is the active argument of **cmsOpen()**:

  pctxt A pointer to the opaque data structure **CMSContext**.

- These are the error codes returned by c**msOpen()**:

  CMS_OUT_OF_MEMORY
  Occurs if there is insufficient memory to create a **CMSContext**.

  CMS_FAILURE
  Occurs if the Coloratura CMS is unable to open one of the CMMs on disk.

Terminating the Coloratura CMS: cmsClose()

Call **cmsClose()** when you are finished with the Coloratura CMS; no Coloratura function calls may be made afterwards. The function **cmsClose()** first requests that any open CMMs close themselves, and then it closes open resources and frees storage that the Coloratura CMS still controls. You must make your own calls to free memory that your application allocated and that cannot be freed by a Coloratura function.

- This is the prototype for **cmsClose()**:

```
int32  cmsClose(CMSContext ctxt;
```

- This is the active argument of **cmsClose()**:

  ctxt The context initialized by **cmsOpen()**.

Coloratura Commands

Along with the function library that you can use in your application, the Coloratura CMS provides several commands that perform useful, elementary operations: embedding profiles and simple color transformations of common image formats.

Below is a brief description of the commands. See the man pages for further details.

cmssgi2*

The following commands perform file format conversions that retain embedded ICC profile information:

- cmssgi2jpg(1)
- cmssgi2stiff(1)

The commands differ according to the image file format they produce; the format is indicated by the string following *cmssgi2* in the command name.

cmstag*

The following commands all embed either a complete ICC device profile or its filename in one or more image files:

- cmstag(1)
- cmstaggif(1)
- cmstagjpeg(1)
- cmstagsgi(1)
- cmstagstiff(1)

Embedding the filename is only useful if the image data and profile remain in the same file system.

The commands differ according to the image file format they act on; the format is indicated by the string following *tag* in the command name. *cmstag* embeds profile information in files of several different formats.

coco*

The following commands use ICC profiles to perform a color management operation ("color correction," hence "coco") on image files:

- cocogif(1)
- cocojpeg(1)
- cocostiff(1)

The operation is a color space conversion from a specified input device color space to a specified output device color space. The image file format each command acts on is indicated by the string following *coco.*

These commands are essentially compiled variants of the sample code cocoifl, outlined in "Example Outline of a Color Conversion Program" on page 8, and presented in Appendix B, "Listing of the Application cocoifl."

Sample Code and Test Profile

The Coloratura CMS includes several code examples to illustrate how to use the library in typical situations (see */usr/cms/examples/*). The library also includes a test profile, which provides visual feedback indicating that a color transformation program is working properly.

cocoifl

This example code illustrates a basic color conversion program, using the Image Format Library to read most image files. The basic operations performed by cocoifl are described in “Example Outline of a Color Conversion Program” on page 8. Appendix B, “Listing of the Application cocoifl,” shows the source code.

makedevlink and applydevlink

These examples illustrate a special case of cocoifl: applying a device-link profile to image data. makedevlink is easily extended.

threeprof

This example illustrates combining three profiles, an operation you would be likely to perform when simulating an output.

howtagged

This example extracts a profile from an image file and gets the profile’s tag data.

invert.pf

This profile provides a simple visual test that something happened in your color management application. This will not find flaws in a Coloratura application, as there are no exact digital values guaranteed, but the visual test gives a good indication of how (and whether) your application is performing.

The profile is like a monitor profile, but the tone curve is flipped. If you embed it in an image and use the default monitor profile, the image should look somewhat like a photographic negative. If the Coloratura CMS fails to process the image, you probably get back the original image.

Or you can make this profile your default monitor profile, by placing it in */var/cms/profiles/local* and calling it *monitor.pf*. Then if your image has a source device profile embedded in it, you also see a “sort-of-negative” image.

Chapter 3

Profile Management

This chapter describes how to manipulate ICC profiles with the Coloratura CMS. Profiles contain the data required to perform color manipulations.

This chapter does not describe the profile format, which is covered in the *ICC Profile Format Specification*. The format of the profile on disk is irrelevant in any case, because when the Coloratura CMS loads a profile from disk, it stores the data in an opaque data structure, which is then used in all subsequent interactions with the profile data. Details about interacting with profile data are covered in Chapter 4, "Tag Management."

This chapter discusses profile data structures and accessor functions in the following sections:

Identifying a Profile

The first task you face in a Coloratura application is to identify the profiles you need. There are three ways to specify profiles:

- predefined information
- default profiles
- profile iteration

You can use predefined information when you know which device profile you need. For example, if you are writing an Impressario™ model file for a printer, the profile will have a name matching the device name, and the profile can be embedded directly in the model file. Another useful place to put profile names is in an X resource file.

The constants CMS_DEFAULT_MONITOR and CMS_DEFAULT_CMYK specify default profile names. You can specify preferred values for these parameters, if you do not want to use those set in *cms.h*. Alternatively, you can use the default filenames, but use versions in directories controlled by the environment variable CMS_DEFAULT_PATH. For more information on CMS_DEFAULT_PATH, see "Loading Profile Data: cmsOpenProfile()" on page 26.

If you do not know the profile you need and do not want to use a default, you can examine the profiles available on your system and select one. The technique for doing this is *profile iteration*. Typically, you select a profile based on its header information, and then open it, that is, load the data into a Coloratura data structure for further interaction. You can also use profile iteration to make a list for a profile selection menu. For example, you could create a list of all the profiles for a device type or make a list of all profiles that use a given profile connection space.

Profile Iteration: Pseudocode Example

To perform a profile iteration, you first create a **CMSProfileIterator** by calling **cmsStartProfileIteration()**, and then repeatedly call **cmsNextProfileIteration()**. If there is another profile in the iteration sequence, **cmsNextProfileIteration()** sets a pointer to its filename. When there are no more profiles on disk, **cmsNextProfileIteration()** returns NULL.

To efficiently select a profile or display a set of profiles in a menu, you typically need only the minimal information kept in profile headers, which are only 128 bytes. Use **cmsGetHeaderProfileSpec()** to query a profile on disk for header information.

When there are no more profiles available, you dispose of the iterator by calling **cmsEndProfileIteration()**.

The following code fragment illustrates a profile iteration. In the interest of clarity, the proper error checking is not shown.

```
char               *pspec;
CMSProfileIterator  theIterator;
icHeader            header;

cmsStartProfileIteration(context, &theIterator);
while (cmsNextProfileIteration(context, theIterator, &pspec) !=
        NULL) {

    cmsGetProfileSpecHeader(context, spec, &header);
    if (/* tests on fields in header pass */) {
        /* code to add profile to the application */
    }
    free(pspec);
}
cmsEndProfileIteration(context, theIterator);
```

The next sections detail the data structures and function calls in this example, and introduce related functions.

The data structure **icHeader** is not discussed because it is not a Coloratura structure; it is the profile header data structure defined by the *ICC Color Profile Format Specification*. **icHeader** is defined in *ic.h*. Since **icHeader** is of fixed size, you are responsible for allocating and freeing storage for it, as discussed in "Managing Memory" on page 14.

Data Structure for Profile Iteration: CMSProfileIterator

The pointer **CMSProfileIterator** refers to an opaque data structure that keeps track of position in a list of profiles during an iteration. This is the data type declaration:

```
typedef struct _CMSProfileIterator  *CMSProfileIterator;
```

Starting Profile Iteration: cmsStartProfileIteration()

The function **cmsStartProfileIteration()** starts a profile iteration and creates a **CMSProfileIterator**.

- This is the prototype for **cmsStartProfileIteration()**:

```
int32  cmsStartProfileIteration(CMSContext ctxt,
                                CMSProfileIterator *iterator);
```

- These are the arguments of **cmsStartProfileIteration()**:

  ctxt The context initialized by **cmsOpen()**.

  iterator The newly generated iterator.

- This is the error code returned by **cmsStartProfileIteration()**:

  CMS_PROFILE_NOT_FOUND
  There are no profiles available on the system.

Stepping Through Profiles: cmsNextProfileIteration()

The function **cmsNextProfileIteration()** finds the next filename in a profile iteration. When the iteration is complete, the function returns NULL.

- This is the prototype for **cmsNextProfileIteration()**:

```
int32 cmsNextProfileIteration(CMSContext ctxt,
                              CMSProfileIterator profIterator,
                              char **spec);
```

- These are the arguments of **cmsNextProfileIteration()**:

  ctxt The context initialized by **cmsOpen()**.

  profIterator The profile iterator.

  spec The filename of the next profile.

- These are the error codes returned by **cmsNextProfileIteration()**:

  CMS_OUT_OF_MEMORY
  There is insufficient memory for the next iteration.

  CMS_NO_MORE_PROFILES
  There are no more profiles over which to iterate.

Examining Headers of Profiles on Disk: cmsGetProfileSpecHeader()

The function **cmsGetProfileHeader()** reads the ICC profile header from a profile on disk.

You can also read a profile header from an open profile, one that has been read into a Coloratura data structure; see "Getting Open-Profile Header Information: cmsGetProfileHeader()" on page 30.

- This is the prototype for **cmsGetProfileSpecHeader()**:

```
int32 cmsGetProfileSpecHeader(CMSContext ctxt,
                              char *spec,
                              icHeader *pHeader);
```

- These are the arguments of **cmsGetProfileSpecHeader()**:

  ctxt The context initialized by **cmsOpen()**.

  spec The filename of the profile.

  pHeader A pointer to the data structure in which the profile header is to be stored.

- The error code **cmsGetProfileSpecHeader()** returns is

  CMS_PROFILE_NOT_FOUND
  Is self-explanatory.

Stopping a Profile Iteration: cmsEndProfileIteration()

The function **cmsEndProfileIteration()** ends a profile iteration and disposes of the iterator.

- This is the prototype for **cmsEndProfileIteration()**:

```
int32 cmsEndProfileIteration(CMSContext ctxt,
            CMSProfileIterator profIterator);
```

- These are the arguments of **cmsEndProfileIteration()**:

  ctxt The context initialized by **cmsOpen()**.

  profIterator The iterator to be disposed of.

- This is the error code returned by **cmsEndProfileIteration()**:

  CMS_FAILURE
  Occurs if you have an invalid profile iterator.

Opening, Closing, and Deleting Profiles

After you identify the profiles you need, you must open them to place the data in appropriate data structures and make the information available to Coloratura functions. Interactions with profile data then occur via the pointer **CMSProfile**, which refers to an opaque Coloratura data structure that holds the profile data; you cannot make any assumptions about the data structure.

To make profile data available to the Coloratura CMS, call **cmsOpenProfile()**, which places the data in a **CMSProfile** data structure. When you are done working with the profile, call **cmsCloseProfile()** to allow the framework to recover some memory. To completely remove a profile from the file system, call **cmsDeleteProfile()**.

The following sections provide more detail about **CMSProfile** and the open, close, and delete functions.

Data Structure for Profiles: CMSProfile

The pointer **CMSProfile** refers to an opaque data structure that stores profile data. All of your program's interactions with profile data are mediated by a **CMSProfile**.

- This is the prototype for **CMSProfile**:

```
typedef struct _CMSProfile  *CMSProfile;
```

Loading Profile Data: cmsOpenProfile()

You call the function **cmsOpenProfile()** to provide Coloratura functions with access to profile data that are on disk. Only after the profile is open can your application read from it, write to it, or include it in a transform. The function **cmsOpenProfile()** needs a filename to get the data from disk, but all subsequent references to the profile data are through the **CMSProfile** pointer set by **cmsOpenProfile()**.

- This is the prototype for **cmsOpenProfile()**:

```
int32  cmsOpenProfile(CMSContext ctxt, char *spec,
                      CMSProfile *prof);
```

- These are the arguments of **cmsOpenProfile()**:

  ctxt The context initialized by **cmsOpen()**.

  spec The input filename. This need not include a full pathname. The Coloratura CMS searches for a file sequentially in the directories specified by the environment variable CMS_DEFAULT_PATH, which is a colon-separated list of pathnames similar to that used in the environment variable MANPATH.

  The value of CMS_DEFAULT_PATH defined in *cms.h* is "*/var/cms/profiles/local:/var/cms/profiles:.*", which allows you to place profiles you prefer in */var/cms/profiles/local*, and generic profiles, which might have the same names, in */var/cms/profiles* or your working directory.

  prof A pointer to the newly created profile data structure.

- These are the error codes returned by **cmsOpenProfile()**:

  CMS_OUT_OF_MEMORY
  The Coloratura CMS cannot allocate memory for the **CMSProfile** data structure.

  CMS_PROFILE_NOT_FOUND
  Is self-explanatory.

Terminating Access to Profile Data: cmsCloseProfile()

The function **cmsCloseProfile()** closes the copy of a profile in memory, that is, a **CMSProfile** data structure, and frees the memory allocated. If you want to remove the profile data from disk, use **cmsDeleteProfile()**, discussed below.

The function **cmsCloseProfile()** does not automatically save the **CMSProfile** data to disk. To save changes to a profile before you call **cmsCloseProfile()**, call **cmsSaveProfile()**, which is discussed in "Saving Profile Changes to Disk: cmsSaveProfile()" on page 31, or **cmsSaveProfileAs()**, which is discussed in "Saving to a New File on Disk: cmsSaveProfileAs()" on page 32.

- This is the prototype for **cmsCloseProfile()**:

```
int32  cmsCloseProfile(CMSContext ctxt, CMSProfile prof);
```

- These are the arguments of **cmsCloseProfile()**:

  ctxt The context initialized by **cmsOpen()**.

  prof The profile to be closed.

Deleting a Profile from Disk: cmsDeleteProfile()

The function **cmsDeleteProfile()** deletes a profile from the file system. The usual IRIX permission system applies; you might not have permission to delete a given profile.

- This is the prototype for **cmsDeleteProfile()**:

```
int32  cmsDeleteProfile (CMSContext ctxt, char *name);
```

- These are the arguments of **cmsDeleteProfile()**:

  ctxt The context initialized by **cmsOpen()**.

  name The profile name.

- If **cmsDeleteProfile()** cannot find the profile, it will return the error

  CMS_PROFILE_NOT_FOUND
  The named profile cannot be found to be deleted.

Creating New Profiles, Getting and Setting Headers, and Saving Edits

To calibrate devices such as scanners, monitors, or printers, or to fine tune an existing profile, you often need to create and modify profiles. You create new profiles by calling **cmsCreateProfile()**.

The function **cmsSetProfileHeader()** sets new header values for an open profile: a profile you just created, or a profile you have edited whose header information you want to change. To inspect the header of an open profile, you can call **cmsGetProfileHeader()**.

This returns the same information as **cmsGetProfileSpecHeader()** (see "Examining Headers of Profiles on Disk: cmsGetProfileSpecHeader()" on page 25), but it uses the **CMSProfile**, rather than the filename, to identify the profile data.

If you want to change other data in the profile, see Chapter 4, "Tag Management," which describes the calls you use to create or modify a profile's tag data.

To save a modified version of a profile to disk and retain the original, call **cmsSaveProfileAs()**. Typically, you call this function when you want to do one of the following:

- save an edited profile under a new name
- save changes to a newly created profile

To overwrite the original version of a profile on disk that you have opened and edited, use **cmsSaveProfile().**

Creating a New Profile: cmsCreateProfile()

The **cmsCreateProfile()** function creates a new **CMSProfile** data structure that contains no tag data. The new profile is not available to the file system until you save it with a call to **cmsSaveProfileAs()**, which is discussed in "Saving to a New File on Disk: cmsSaveProfileAs()" on page 32.

- This is the prototype for **cmsCreateProfile()**:

  ```
  int32  cmsCreateProfile(CMSContext ctxt, CMSProfile *prof);
  ```

- These are the arguments of **cmsCreateProfile()**:

  ctxt The context initialized by **cmsOpen()**.

  prof A pointer to the new **CMSProfile**.

- This is the error code returned by **cmsCreateProfile()**:

  CMS_OUT_OF_MEMORY
  There is not enough memory available to create a new profile data structure.

Getting Open-Profile Header Information: cmsGetProfileHeader()

The function **cmsGetProfileHeader()** returns the same information as **cmsGetProfileSpecHeader()** (see "Examining Headers of Profiles on Disk: cmsGetProfileSpecHeader()" on page 25), but differs in that it takes a **CMSProfile**, rather than a filename.

cmsGetProfileHeader() gets the header information from an open profile and returns a pointer to an **icHeader** data structure, which is declared in *ic.h (*see the *ICC Device Profile Format Specification* for a definition of what is stored in an **icHeader** structure). Since **icHeader** is of fixed size, you must allocate and free storage for it, as discussed in "Managing Memory" on page 14.

- This is the prototype for **cmsGetProfileHeader()**:

```
int32  cmsGetProfileHeader(CMSContext ctxt, CMSProfile prof,
                           icHeader *pHeader);
```

- These are the arguments of **cmsGetProfileHeader():**

  ctxt The context initialized by **cmsOpen()**.

  prof The **CMSProfile** where the header data come from.

  pHeader A pointer to the data structure in which to store the information.

- This is the error code returned by **cmsGetProfileHeader()**:

  CMS_FAILURE

Setting Profile Header Information: cmsSetProfileHeader()

The function **cmsSetProfileHeader()** stores a header in an open profile. Typically, you use **cmsSetProfileHeader()** to put a header in a new profile.

The Coloratura CMS does not provide a function to update selected fields of a header, which it reads or writes as a whole. To update a specific field, you should first read the header into an **icHeader** structure, using **cmsGetProfileHeader()**. Then update the field with your own code and write the header to the profile with **cmsSetProfileHeader()**.

The header file *ic.h* defines **icHeader**. Since **icHeader** is of fixed size, you must allocate and free storage for it. See the *ICC Profile Format Specification* for a definition of what is stored in a header.

- This is the prototype for **cmsSetProfileHeader()**:

```
int32 cmsSetProfileHeader(CMSContext ctxt, CMSProfile prof,
                          icHeader *pHeader);
```

- These are the arguments of **cmsSetProfileHeader()**:

  ctxt The context initialized by **cmsOpen()**.

  prof The profile whose header you want to set.

  pHeader A pointer to the data structure from which to read the information.

- This is the error code returned by **cmsSetProfileHeader()**:

  CMS_OUT_OF_MEMORY
  There is not enough memory to add the header data to the profile.

Saving Profile Changes to Disk: cmsSaveProfile()

The **cmsSaveProfile()** function saves an open profile to permanent storage, making it available to the file system. It does not remove the **CMSProfile** data structure from memory; the data remains available for further editing. Use **cmsCloseProfile()** to delete the **CMSProfile** data structure.

- This is the prototype for **cmsSaveProfile()**:

```
int32  cmsSaveProfile(CMSContext ctxt, CMSProfile prof);
```

- These are the arguments of **cmsSaveProfile()**:

  ctxt The context initialized by **cmsOpen()**.

  prof The profile.

- The function **cmsSaveProfile()** returns an error if you try to save a profile that does not have a filename. Use **cmsSaveProfileAs()** to save a profile and attach a name to it. These are the error codes returned by **cmsSaveProfile()**:

  CMS_FAILURE
  Something went wrong; no further diagnostic information is available.

  CMS_OUT_OF_MEMORY
  There is not enough memory for the profile.

Saving to a New File on Disk: cmsSaveProfileAs()

The function **cmsSaveProfileAs()** is similar to **cmsSaveProfile()**, in that it writes a profile to permanent storage, but it creates a new file with a filename you specify.

- This is the prototype for **cmsSaveProfileAs()**:

```
int32 cmsSaveProfileAs(CMSContext ctxt, CMSProfile prof, char*spec);
```

- These are the arguments of **cmsSaveProfileAs()**:

  ctxt The context initialized by **cmsOpen()**.

  prof The profile to be saved.

  spec The filename of the saved profile.

- These are the error codes returned by **cmsSaveProfileAs()**:

  CMS_FAILURE
  Something went wrong; no further diagnostic information is available.

  CMS_OUT_OF_MEMORY
  There is not enough memory for the new profile.

  CMS_EXACT_PROFILE_EXISTS
  A profile already exists with the name *spec*.

Importing and Exporting Embedded Profiles

The *ICC Color Profile Format Specification* prescribes a way to embed ICC profiles in image data. This allows you to move color data to different computers and operating systems without concern for whether the necessary profiles are present at the destination.

The functions **cmsExportProfile()** and **cmsImportProfile()** exchange data between a **CMSProfile** data structure and data in the ICC format. This simplifies interactions with image data that include embedded profiles. You free storage for the exported data with **cmsFreeProfileExport().**

Creating an ICC Profile in a Buffer: cmsExportProfile()

The function **cmsExportProfile()** converts the data in a **CMSProfile** data structure to the ICC file format. Storage for the exported data is allocated by the Coloratura CMS, so it must be freed by calling **cmsFreeProfileExport()**.

- This is the prototype for **cmsExportProfile()**:

```
int32  cmsExportProfile(CMSContext ctxt, CMSProfile prof,
                        uint32 *length,  void **outputData);
```

- These are the arguments of **cmsExportProfile()**:

ctxt	The context initialized by **cmsOpen()**.
prof	The **CMSProfile** data structure to be converted.
length	The size of exported data, in bytes.
outputData	A buffer of data in the ICC format.

- This is the error code returned by **cmsExportProfile()**:

CMS_OUT_OF_MEMORY
: There is not enough memory to allocate *outputData*.

Deleting an ICC Profile Buffer: cmsFreeProfileExport()

The function **cmsFreeProfileExport()** frees the output data storage created by **cmsExportProfile()**.

- This is the prototype for **cmsFreeProfileExport()**:

```
int32  cmsFreeProfileExport(CMSContext ctxt, void *outputData);
```

- These are the arguments of **cmsFreeProfileExport()**:

ctxt	The context initialized by **cmsOpen()**.
outputData	The buffer of data to be freed.

Importing an ICC Profile from a Buffer: cmsImportProfile()

To create a **CMSProfile** data structure from a data buffer that is in the ICC format, typically obtained from an embedded profile, call **cmsImportProfile()**. You provide a **CMSProfile**, which **cmsImportProfile()** initializes to locate the newly constructed profile data structure.

- This is the prototype for **cmsImportProfile()**:

```
int32  cmsImportProfile(CMSContext ctxt,
                        uint32 length, void *inputData,
                        CMSProfile *profile);
```

- These are the arguments of **cmsImportProfile():**

  ctxt The context initialized by **cmsOpen()**.

  length The size, in bytes, of the input data.

  inputData A buffer of data in the ICC format.

  profile The profile to be initialized.

- This is the error code returned by **cmsImportProfile()**:

  CMS_OUT_OF_MEMORY
  There is not enough memory to create a **CMSProfile** data structure from the profile data.

  CMS_WRONG_DATA
  The data do not conform to the ICC file format.

Chapter 4

Tag Management

This chapter describes Coloratura functions and data structures for access to the data held in profiles: tag data. For example, you could specify the tone reproduction curve for the red channel of a device, or data for an *abstract profile*, which transforms data in a profile connection space. Other uses for tag access are to determine if the profile is appropriate for a transform, or to create or modify a profile.

In contrast to headers, which contain fixed-length data that is frequently accessed and provides generic characterization of a profile, tags contain variable-length and fixed-length data that is less frequently accessed and provides the technical details that define a profile. Because tags hold the detailed information, you need access to tag data for complete control of the color management process. Access to headers is discussed in the sections of Chapter 3 "Examining Headers of Profiles on Disk: cmsGetProfileSpecHeader()" on page 25 and "Creating New Profiles, Getting and Setting Headers, and Saving Edits" on page 28.

The Coloratura CMS does not define tags that are not in the *ICC Profile Format Specification*. You can find a complete description of all publicly defined tags there. The header file *ic.h* provides brief descriptions of tags and their data types.

These are the topics covered in this chapter:

To access tag data, you use two data structures that are defined in the *ICC Profile Format Specification* and declared in *ic.h*:

- **icTagSignature**, which identifies tags
- **icTagTypeSignature**, which identifies a tag's type

Getting Tag Data Sequentially: Tag Iteration

You can iterate through tags to examine a profile more closely and, perhaps, manipulate its contents. You typically use tag iteration as a component of a profile validator, which checks the syntax of a profile or verifies that the contents are meaningful, or a profile examiner, which lists the contents of a profile and helps you determine whether to use it.

Tag Iteration: Pseudocode Example

Tag iteration is similar to profile iteration, which was discussed in the section "Loading Profile Data: cmsOpenProfile()" on page 26. After you open a profile with **cmsOpenProfile()**, you create a **CMSTagIterator** data structure by calling **cmsStartTagIteration()**, and then repeatedly call **cmsNextTagIteration()**.

In contrast to profile iteration, however, tag iteration functions use the Coloratura version of the profile, that is a **CMSProfile** data structure. Thus, you do not obtain information from the disk version of the profile, and may get unsaved changes to tag data resulting from previous Coloratura calls.

You can modify tag data during an iteration with the function **cmsSetTag()**, which is discussed in "Setting Tag Data: cmsSetTag()" on page 41. Changes are not saved to disk until you save the profile with **cmsSaveProfile()** or **cmsSaveProfileAs()**, discussed in "Creating New Profiles, Getting and Setting Headers, and Saving Edits" on page 28.

When there are no more tags in a profile, **cmsNextTagIteration()** returns NULL. You then call **cmsEndTagIteration()**. The Coloratura CMS allocates space to store tag data, so, to free memory, you must call **cmsFreeTagValue()** when you are done with the data.

Here is pseudocode that illustrates how to step through the tags in a profile. In the interest of clarity, the proper error checking is not performed.

```
CMSProfile      prof;
CMSTagIterator  theIterator;

icTagSignature    name;
icTagTypeSignature  type;
uint32             size;
char              *pdata;

cmsStartTagIteration(context, prof, &theIterator);
while (cmsNextTagIteration(context, theIterator, &name,
                           &type, &size, &pdata)!= NULL) {
    /* do whatever with the data from the tag. */
    cmsFreeTagValue(pdata);
}
cmsEndTagIteration(context, theIterator);
```

Data Structure for Tag Iteration: CMSTagIterator

The pointer **CMSTagIterator** refers to an opaque data structure that keeps track of position in a list of tags during an iteration. This is the data type declaration:

```
typedef struct _CMSTagIterator *CMSTagIterator;
```

Starting Tag Iteration: cmsStartTagIteration()

The function **cmsStartTagIteration()** creates a **CMSTagIterator** and starts a tag iteration.

- This is the prototype for **cmsStartTagIteration()**:

```
int32  cmsStartTagIteration(CMSContext ctxt,
                            CMSProfile prof,
                            cmsTagIterator *iterator);
```

- These are the arguments of **cmsStartTagIteration()**:

  ctxt The context initialized by **cmsOpen()**.

  prof The profile in which to find tags.

  iterator The newly generated iterator.

- This is the error code returned by **cmsStartTagIteration()**:

  CMS_OUT_OF_MEMORY

Stepping Through Tags: cmsNextTagIteration()

The function **cmsNextTagIteration()** returns the next tag in an iteration sequence. When the iteration is complete, it sets the tag name to NULL and returns CMS_NO_MORE_TAGS.

To change tag data during an iteration, use the function **cmsSetTag()**, discussed in "Setting Tag Data: cmsSetTag()" on page 41.

When you are finished with a tag, free its allocated memory by calling **cmsFreeTagValue()**, which is discussed in "Deleting Tag Data from a Profile: cmsDeleteTag()" on page 42.

- This is the prototype for **cmsNextTagIteration()**:

```
int32  cmsNextTagIteration(CMSContext ctxt,
                           CMSTagIterator iterator,
                           icTagSignature *name,
                           icTagTypeSignature *type,
                           uint32 *size,
                           void **data);
```

- These are the arguments of **cmsNextTagIteration()**:

  ctxt The context initialized by **cmsOpen()**.

  iterator The tag iterator.

  name The returned tag name.

  type The returned data type.

  size The size, in bytes, of the returned data.

  data The data.

- These are the error codes returned by **cmsNextTagIteration()**:

  CMS_OUT_OF_MEMORY
  There is not enough memory to hold the next tag.

  CMS_NO_MORE_TAGS
  There are no more tags in the profile.

Stopping a Tag Iteration: cmsEndTagIteration()

The function **cmsEndTagIteration()** terminates a tag iteration and deletes the iterator.

- This is the prototype for **cmsEndTagIteration()**:

```
int32 cmsEndTagIteration(CMSContext ctxt, CMSTagIterator iterator);
```

- These are the arguments of **cmsEndTagIteration()**:

  ctxt The context initialized by **cmsOpen()**.

  iterator The tag iterator.

Getting Tag Data Directly: cmsGetTag()

When you have an open profile and you know the tag you want to work with, you can read the tag data by calling **cmsGetTag()** and specifying the tag by name, instead of searching through the list of tags with an iteration.

The function **cmsGetTag()** returns the tag data type, a pointer to the data, and the size of the data structure. Note that the function might return tag data that has been modified and not yet saved to disk, because the tag data it returns comes from a **CMSProfile** data structure. For more information on tag data types, see the header file *ic.h* and the *ICC Profile Format Specification*.

When you no longer need the storage the Coloratura CMS allocates for returned tag data, free it with a call to **cmsFreeTagValue()**, discussed in "Deleting Tag Data from a Profile: cmsDeleteTag()" on page 42.

- This is the prototype for **cmsGetTag()**:

```
int32  cmsGetTag(CMSContext ctxt,
                 CMSProfile prof,
                 icTagSignature name,
                 icTagTypeSignature *type,
                 uint32 *size, void **data);
```

- These are the arguments of **cmsGetTag()**:

ctxt	The context initialized by **cmsOpen()**.
prof	The profile that owns the tag.
name	The name of the tag being queried.
type	The returned value for the tag data type.
size	The size, in bytes, of the returned data.
data	A pointer to the data.

- These are the error codes returned by **cmsGetTag()**:

CMS_TAG_NOT_FOUND
: The tag *name* was not in profile *prof*.

CMS_OUT_OF_MEMORY
: There is not enough memory to store the tag data.

Setting Tag Data: cmsSetTag()

Given a **CMSProfile** and a tag name, the function **cmsSetTag()** sets the tag data to values you supply. If the specified tag doesn't exist, **cmsSetTag()** creates it. The tag data must follow the format given in the *ICC Profile Format Specification*.

Save modified or newly created tags to a file with **cmsSaveProfile()** or **cmsSaveProfileAs()**, which are discussed in "Creating New Profiles, Getting and Setting Headers, and Saving Edits" on page 28. Any tag changes or deletions you make that you do not explicitly save have no effect on the disk image of the profile.

When you pass data to the Coloratura CMS with **cmsSetTag()**, it makes a copy of the data. You should, therefore, free storage for your tag data source after you call **cmsSetTag()**. You free the Coloratura version of the data with **cmsFreeTagValue()**.

- This is the prototype for **cmsSetTag()**:

```
int32  cmsSetTag(CMSContext ctxt,
                 CMSProfile prof,
                 icTagSignature name,
                 uint32 size,
                 void *data);
```

- These are the arguments of **cmsSetTag()**:

ctxt	The context initialized by **cmsOpen()**.
prof	The profile that owns the tag.
name	The name of the tag data.
size	The size, in bytes, of the data.
data	A pointer to the data

- This is the error code returned by **cmsSetTag()**:

  CMS_OUT_OF_MEMORY

Deleting Tag Data from a Profile: cmsDeleteTag()

The function **cmsDeleteTag()** removes tag data from a **CMSProfile** data structure. Do not attempt to remove data by calling **cmsSetTag()** with a data size of zero bytes; this will not remove the data and may confuse the Coloratura CMS.

Remember to save a modified **CMSProfile** structure to a disk file by calling **cmsSaveProfile()** or **cmsSaveProfileAs()** (see "Creating New Profiles, Getting and Setting Headers, and Saving Edits" on page 28). Any tag changes or deletions you make that you do not explicitly save have no effect on the disk image of the profile.

- This is the prototype for **cmsDeleteTag()**:

```
int32  cmsDeleteTag(CMSContext ctxt, CMSProfile prof,
                    icTagSignature tagName);
```

- These are the arguments of **cmsDeleteTag()**:

  ctxt The context initialized by **cmsOpen()**.

  prof The profile that owns the tag.

  tagName The name of the tag to delete.

- These are the error codes returned by **cmsDeleteTag()**:

  CMS_TAG_NOT_FOUND

Freeing Tag Data Storage: cmsFreeTagValue()

The function **cmsFreeTagValue()** frees tag data returned by **cmsNextTagIteration()** or **cmsGetTag().**

In addition to a data pointer, you must pass **cmsFreeTagValue()** the tag name to identify the data type to free, because tag data may have internal structure that affects memory allocation.

- This is the prototype for **cmsFreeTagValue()**:

```
int32  cmsFreeTagValue(CMSContext ctxt,
                       icTagSignature name,
                       void *data);
```

- These are the arguments of **cmsFreeTagValue()**:

ctxt	The context initialized by **cmsOpen()**.
name	The name of the tag.
data	The data to be freed.

Chapter 5

Transform Management

A transform converts pixel data from an input color space to an output color space; it is the central processing step in a color management application. You build a transformation algorithm by specifying a sequence of one or more primitive color manipulations, described by profiles, and a CMM. You then apply the transform to pixel data.

For example, a common two-profile transform converts from an input color space to an output color space; typically the input profile is a monitor profile and the output is a printer profile. A useful three-profile transform simulates on one device the output of another device. The sequence of profiles is first the input, second the simulated device, and third the simulating device: for example, first a scanner, second a printer, and third a monitor.

These are sections of this chapter:

Features of Transform Management Tools

This section introduces the features of the Coloratura transform tools.

Selecting a CMM

The color manipulation module that the Coloratura CMS uses determines the details of the transformation algorithm. Different CMMs can give different results from the same input image and set of profiles, depending on how the CMMs interpret the discrete, and perhaps sparse, data in each profile. When you create a transform, you can select a CMM explicitly, or you can use the default CMM that is included in the Coloratura CMS or, if you have other CMMs, you can have the Coloratura CMS let the profiles determine which CMM to use.

Saving a Transform

You can save a transform, which is convenient if you have a sequence of profiles that you use repeatedly. For example, if you commonly transform data from your monitor to output on a specific printer, you can build a *device-link profile*, which does not represent a particular device, but provides a one-profile characterization of the transformation between devices.

Gamut Checking

As discussed earlier, the set of possible colors for a device is called its color gamut. A central concern for color management is a mismatch between the gamuts of an input and output device. The severity of this mismatch and the distribution of mismatches over an image affect the appearance of your output. A fairly common gamut mismatch is that between a monitor and a printer; any monitor typically has a gamut larger than most printers.

Output devices necessarily have rules to handle out-of-gamut data. If you do not like the results, you can modify the source image data or, conceivably, develop an *abstract profile* to perform a *gamut mapping* to adjust out-of-gamut image data. How you accommodate out-of-gamut image data is a substantial component of an acceptable color conversion, and much of the art of the process.

Before applying a transform, you may want to know how much of the transformed image is out of gamut, rather than observe the effects on the output image. The Coloratura CMS provides gamut-checking tools to give you that information.

Transforming Pixel Data

To create a transform with the Coloratura CMS, you supply a list of profiles and a CMM specifier to the function **cmsCreateTfm()**. You then call **cmsApplyTfm()** to apply the transform to a pixel buffer.

During the process of developing a transform, you probably need to delete a current version. You delete a transform by calling **cmsDeleteTfm()**. To save a transform, use **cmsTFMToLUT()**. The applications mkdevlink and applydevlink in */usr/cms/examples/* illustrate how to use the output of **cmsTFMToLUT()**.

Data Structure for Transforms: CMSTfm

The pointer **CMSTfm** refers to an opaque structure that stores transform data. This is the data type declaration:

```
typedef struct _CMSTfm  *CMSTfm;
```

Data Structure for Pixels: CMSPixelBuffer

Transforms operate on **CMSPixelBuffer** data structures, which hold color image data. The two Coloratura functions that accept a **CMSPixelBuffer** argument are **cmsCheckGamut()** and **cmsApplyTfm()**.

The Coloratura CMS makes the following assumptions about the storage of image data included in a **CMSPixelBuffer**:

- All the channels for a single pixel are stored together.
- Each channel aligns to byte boundaries. Any padding is in the most significant bits.
- The data is encoded according to one of the valid color ICC encodings.

If a pixel holds non-color information, such as the OpenGL opacity parameter *alpha*, then the number of bytes per pixel will be greater than the number of bytes in the image channel data. When implementing transforms, the Coloratura CMS preserves without modification the additional information.

The Coloratura CMS does not store color encoding formats. For the Coloratura CMS to manipulate data with a particular encoding, there must be at least one profile in storage that has that encoding. For example, if no profile has the HSV encoding, the Coloratura CMS cannot process an image in HSV format.

- This is the data type declaration for **CMSPixelBuffer**:

```
typedef struct  _CMSPixelBuffer {
    uint32         width;
    uint32         height;
    uint32         bitsPerChannel;
    uint32         bytesPerPixel;
    uint32         channels;
    uint32         encode;
    unsigned void  *data;
}  CMSPixelBuffer;
```

- These are the fields in the declaration for **CMSPixelBuffer**:

width	The width of the image in pixels.
height	The height of the image in pixels.
bitsPerChannel	The number of bits may range from 1 to 12.
bytesPerPixel	The number of bytes must align on 32-bit boundaries.
channels	The number of color channels. Possible values are: 1, 3, 4, 5, or 6.
encode	These are the possible color space signatures. They correspond to a the values of the enumerated type **icColorSpaceSignature**, which is defined in the header file *ic.h.*
data	A pointer to the image data.

Creating a Transform: cmsCreateTfm()

The function **cmsCreateTfm()** translates a set of profiles into a transformation algorithm. The algorithm is defined by a CMM and a sequence of profiles. The transformation is built from profiles in the sequence in which they are supplied: the first profile defines the first step in the transformation, typically from your input device, and the last profile defines the final step, typically to your output device.

- This is the prototype for **cmsCreateTfm()**:

```
int32 cmsCreateTfm(CMSContext ctxt,
                   int32 profileCount,
                   CMSProfile *profs,
                   icSignature cmm,
                   CMSTfm *ptfm);
```

- These are the arguments of **cmsCreateTfm()**:

ctxt	The context initialized by **cmsOpen()**.
profileCount	The number of profiles in a transform.
profs	An array of profiles for the transform.
cmm	The CMM to use. If you do not directly specify the CMM with a valid **icSignature**, a data type declared in *ic.h*, use one of the following two constants to direct CMM selection: CMS_USE_DEFAULT_CMM selects the default CMM. CMS_USE_PROFILE_CMM prompts the Coloratura CMS to search the CMMs specified by the profiles in the transform until it finds one that can perform the transformation. At least one CMM, the default, can always perform every transform. In searching, the Coloratura CMS examines the profiles from last to first, looking at each profile for a CMM that can perform the entire transformation. The search begins with the last profile because profiles later in the sequence tend to have the greatest effect on output, and a preferred CMM is likely to give the best transform results.
ptfm	The new transform.

- These are the error codes returned by **cmsCreateTfm()**:

  CMS_OUT_OF_MEMORY
  : Not enough memory to create a transform.

  CMS_WRONG_DATA
  : The variable *ctxt* points to something that is not a context.

  CMS_BAD_INPLACE_CONVERT
  : It is not possible to build an in-place conversion for this transform.

  CMS_BAD_ENCODE

  CMS_BAD_CONTEXT

  CMS_TOO_MANY_CHANNELS
  : The CMM cannot support a transform with the requested number of channels.

  CMS_CMM_NOT_AVAILABLE
  : The requested CMM is not available.

Applying a Transform: cmsApplyTfm()

Once you have created a transform, call **cmsApplyTfm()** to apply the transform to pixel data. This function transforms a pixel buffer with one profile to a buffer associated with another profile.

Note that you may not always be able to perform in-place conversions; for example, if the output format requires more space than the input format (RGB to CMYK), or if the CMM does not support in-place conversions. **cmsApplyTfm()** returns an error if you attempt an in-place conversion for one of these cases.

If you are concerned about maintaining interactivity, you may want to transform an image piece-by-piece, to avoid the possibly slow process of transforming a whole image.

- This is the prototype for **cmsApplyTfm()**:

```
int32  cmsApplyTfm(CMSContext ctxt, CMSTfm tfm,
                   CMSPixelBuffer *psrc,
                   CMSPixelBuffer *pdest);
```

- These are the arguments of **cmsApplyTfm()**:

  ctxt The context initialized by **cmsOpen()**.

  tfm The transform to use.

  psrc The pixel buffer to be transformed.

  pdest The resulting output pixel buffer.

- These are the error codes returned by **cmsApplyTfm()**:

  CMS_WRONG_DATA
  Either of the pixel-buffer pointers refers to data in the wrong format for the transform.

  CMS_BAD_TFM
  The argument *tfm* is either not a transform or it was not built with the supplied *ctxt*.

  CMS_BAD_PIXEL_BUF
  Either of the pixel-buffer pointers refers to data in the wrong format for the transform.

  CMS_CONVERT_ERROR
  The transform could not be applied. This error occurs if a transform could not be performed in place.

Saving a Transform as a Look-Up Table: cmsTfmToLUT()

The function **cmsTfmToLUT()** allows you to save transform information in a tag format, the AToB0Tag described in the *ICC Profile Format Specification*. To recover the transform from the tag data produced by **cmsTfmToLUT()**, do the following:

1. Create a profile by calling **cmsCreateProfile()**, which is discussed in "Creating New Profiles, Getting and Setting Headers, and Saving Edits" on page 28. You can specify any rendering intent for the profile when you create the profile header.
2. Set the tag value by calling **cmsSetTag()**, which is discussed in "Setting Tag Data: cmsSetTag()" on page 41. You can save the profile for later use by calling **cmsSaveProfileAs()**, which is discussed in "Saving to a New File on Disk: cmsSaveProfileAs()" on page 32.
3. Create a transform from the profile, by calling **cmsCreateTfm()**.

The application mkdevlink in */usr/cms/examples* illustrates the first two steps. The application applydevlink in the same directory illustrates the last step, and applies the transform to input data.

- This is the prototype for **cmsTfmToLUT()**:

```
int32 cmsTfmToLUT(CMSContext ctxt,
                  CMSTfm tfm,
                  uint32 *psize,
                  void **pdata);
```

- These are the arguments of **cmsTfmToLUT()**:

  tfm The transform to save.

  psize The size of the tag data.

  pdata The tag data.

- This is the error code returned by **cmsTfmToLUT()**:

  CMS_NOT_SUPPORTED
  The CMM does not create a look-up table from a transform.

Deleting a Transform: cmsDeleteTfm()

The function **cmsDeleteTfm()** disposes of all data structures associated with a transform.

- This is the prototype for **cmsDeleteTfm()**:

```
int32  cmsDeleteTfm(CMSContext ctxt, CMSTfm tfm);
```

- These are the arguments of **cmsDeleteTfm()**:

  ctxt The context initialized by **cmsOpen()**.

  tfm The transform.

Checking Gamut Mapping

You can perform a test for which output pixels have data that lies out of gamut, rather than apply a transformation and observe the effects of out-of-gamut image data. This helps you quantify the severity of your gamut mapping problem and to see clearly how out-of-gamut data affect your image. It may help you to develop a gamut mapping strategy.

To examine the gamut mismatch, you create a gamut check with **cmsCreateGamutCheck()**, which takes the same arguments as **cmsCreateTfm()**. You then perform the check with **cmsCheckGamut()**. This function examines the pixel data stored in a **CMSPixelBuffer** data structure, and returns an unsigned **char** array to indicate how each pixel is mapped; zero values indicates the image pixel is in gamut, non-zero indicates out of gamut.

Preparing for a Gamut Map Test: cmsCreateGamutCheck()

The function **cmsCreateGamutCheck()** takes the same set of arguments as **cmsCreateTfm()**; it uses a CMM and a set of profiles to create a gamut checking transform. The gamut check is built from the profiles in the sequence in which they are supplied: the first profile defines the first step in the transformation, typically from your input device, and the last profile defines the final step, typically to your output device.

- This is the prototype for **cmsCreateGamutCheck()**:

```
int32 cmsCreateGamutCheck(CMSContext ctxt,
                          int32 profileCount,
                          CMSProfile *profs,
                          icSignature cmm,
                          CMSTfm *ptfm);
```

- These are the arguments of **cmsCreateGamutCheck()**:

  ctxt The context initialized by **cmsOpen()**.

  profileCount The number of profiles in a transform.

  profs An array of profiles for the transform.

  cmm The CMM to use. If you do not directly specify the CMM with a valid **icSignature**, use one of the following two constants to direct CMM selection:

  CMS_USE_DEFAULT_CMM selects the default CMM.

  CMS_USE_PROFILE_CMM prompts the Coloratura CMS to search the CMMs specified by the profiles in the transform until it finds one that can perform the transformation. See the next section "Creating a Transform: cmsCreateTfm()" on page 49 for more details

  ptfm The gamut-checking transform.

- These are the error codes returned by **cmsCreateGamutCheck()**:

  CMS_OUT_OF_MEMORY
  Not enough memory to create a gamut check.

  CMS_WRONG_DATA
  The *ctxt* variable points to something that is not a context.

  CMS_BAD_ENCODE

  CMS_BAD_CONTEXT

  CMS_TOO_MANY_CHANNELS
  The CMM cannot support a gamut check with the requested number of channels.

  CMS_CMM_NOT_AVAILABLE
  The requested CMM is unavailable.

Checking a Gamut Map: cmsCheckGamut()

Given a transform and set of pixels, the function **cmsCheckGamut()** tests whether the transform maps the pixels within the gamut of the output device. The order of the output bytes follows the order of the input pixels. A value of zero indicates that an output pixel is in gamut; a non-zero value indicates that the pixel is out of gamut.

If you are concerned about maintaining interactivity, you may want to check the gamut mapping of an image piece-by-piece, to avoid the possibly slow process of checking the whole image. The **cmsCheckGamut()** function runs at a rate comparable to that of the transform whose effect it reports.

- This is the prototype for **cmsCheckGamut()**:

```
int32 cmsCheckGamut(CMSContext ctxt, CMSTfm tfm,
                    CMSPixelBuffer *psrc,
                    unsigned char pgamutmap);
```

- These are the arguments of **cmsCheckGamut()**:

  ctxt The context initialized by **cmsOpen()**.

  tfm The transform to check.

  psrc The input pixel buffer.

  pgamutmap The resulting gamut-map buffer.

- These are the error codes returned by **cmsCheckGamut()**:

  CMS_OUT_OF_MEMORY
  There is not sufficient memory available for gamut testing.

  CMS_BAD_PIXEL_BUF
  The variable *psrc* does not point to a valid pixel buffer.

  CMS_BAD_GAMUT_MAP
  The variable *pgamutmap* is not a valid gamut map buffer.

  CMS_BAD_TFM
  The argument *tfm* is either not a transform, or was not built with the supplied *ctxt*.

  CMS_CONVERT_ERROR
  The transform could not be applied.

Chapter 6

Color Manipulation Module Management

The heart of a color management computation is the color manipulation module (CMM), which executes the transformation algorithm.When you create a transformation, you may use one of several CMMs, which are implemented as dynamic shared objects. The Coloratura CMS serves as a dispatcher between your application and the CMMs.

This chapter discusses the tools the Coloratura CMS provides to examine available CMMs. These tools allow you to determine which CMM to specify when you call **cmsCreateTfm()** (see "Creating a Transform: cmsCreateTfm()" on page 49). You may want to examine available CMMs, for example, if you do not want to use the default CMM to perform a transformation, or if you do not want to, or are unable to, use a CMM determined by the profiles in a transformation via CMM_USE_PROFILE_CMM (see "Creating a Transform: cmsCreateTfm()" on page 49).

These are the topics covered in this chapter:

Finding CMMs

The Coloratura CMS provides functions to identify the default CMM and to list all available CMMs. The data structure used to identify CMMs is an **icSignature**, which is declared in *ic.h* and described in the *ICC Profile Format Specification*.

Finding the Default CMM: cmsGetDefaultCmm()

A default CMM ships with the Coloratura CMS. To identify the current default CMM, call **cmsGetDefaultCmm()**, which returns an **icSignature** for the CMM. To obtain information about the CMM, you can call **cmsGetCmmInfo()**, discussed in "Getting Information About a CMM" on page 59.

- This is the prototype for **cmsGetDefaultCmm()**:

```
int32  cmsGetDefaultCmm(CMSContext ctxt, icSignature *cmm);
```

- These are the arguments of **cmsGetDefaultCmm()**:

ctxt	The context initialized by **cmsOpen()**.
cmm	The identifier of the CMM.

Listing the Available CMMs: cmsGetCmmList()

The function **cmsGetCmmList()** supplies a list of available CMMs and the number of CMMs on the list. You may select from the list a preferred CMM to use when creating a transform. Recall that your application cannot use any CMM that you add after you call **cmsOpen()**.

To obtain information about a particular CMM, you call **cmsGetCmmInfo()**, discussed in "Getting CMM Information: cmsGetCmmInfo()" on page 60. Free the list of CMMs by calling **cmsFreeCmmList()**.

- This is the prototype for **cmsGetCmmList()**:

```
int32  cmsGetCmmList(CMSContext ctxt, uint32 *count,
                     icSignature **cmms);
```

- These are the arguments of **cmsGetCmmList()**:

ctxt	The context initialized by **cmsOpen()**.
count	The number of CMMs available.
cmms	The list of identifiers for the CMMs.

Freeing the List: cmsFreeCmmList()

The function **cmsFreeCmmList()** frees the list of available CMMs returned by **cmsGetCmmList()**.

- This is the prototype for **cmsFreeCmmList()**:

```
int32  cmsFreeCmmList(CMSContext ctxt, icSignature *cmms);
```

- These are the arguments of **cmsFreeCmmList()**:

ctxt	The context initialized by **cmsOpen()**.
cmms	The list of identifiers for the CMMs.

Getting Information About a CMM

The Coloratura CMS provides a function, **cmsGetCmmInfo()** to return information about a CMM. The information is held in an enumerated data type, **CMSInfoName**.

CMM Information Data Structure: CMSInfoName

Information about a CMM is held in the enumerated data type **CMSInfoName**.

- This is the data type declaration for **CMSInfoName**:

```
typedef enum {
        CMS_CMM_NAME,
        CMS_CMM_VERSION,
        CMS_FW_VERSION,
        CMS_CAN_DO_IC,
        CMS_MULTIPLE_OK
} CMSInfoName;
```

- The following lists the meanings of the information fields:

CMS_CMM_NAME
: The registered 32-bit CMM name. The name uniquely identifies the CMM and is often interpreted as a 4-character mnemonic.

CMS_CMM_VERSION
: A uint32 uniquely identifying the version of the CMM.

CMS_FW_VERSION
: The version of the Coloratura CMS for which the CMM was programmed. The returned value is a 4-byte string containing the major version and the minor version. Use *CMS_VERSION_MAJOR_MASK* and *CMS_VERSION_MINOR_MASK* to extract these numbers from *CMS_FW_VERSION*.

CMS_CAN_DO_IC
: If TRUE, the CMM supports ICC profiles.

CMS_MULTIPLE_OK
: If TRUE, the CMM supports transformations made from more than two profiles.

Getting CMM Information: cmsGetCmmInfo()

The function **cmsGetCmmInfo()** queries a CMM and returns identifying information.

- This is the prototype for **cmsGetCmmInfo()**:

```
int32  cmsGetCmmInfo(CMSContext ctxt, icSignature cmm,
                     CMSInfoName name, uint32 *value);
```

- These are the arguments of **cmsGetCmmInfo()**:

ctxt
: The context initialized by **cmsOpen()**.

cmm
: The CMM identifier obtained from **cmsGetDefaultCmm()** or **cmsGetCmmList()**.

name
: The name of the information field for the CMM.

value
: The value of the specified field.

- These are the error codes returned by **cmsGetCmmInfo()**:

CMS_BAD_CONTEXT
: The *ctxt* argument is not a valid CMS context.

CMS_CMM_NOT_AVAILABLE
: The *cmm* argument was not a valid name for a CMM.

CMS_MISSING
: The color management system is missing.

Appendix A

Summary of Functions and Data Structures

This appendix provides a quick reference to the Coloratura CMS. It lists all the Coloratura functions, and important data structures and parameters.

The lists of functions are grouped in sections as follows, according to the elements they affect:

1. “Coloratura Access Functions” on page 62
2. “Profile Functions” on page 62
3. “Tag Functions” on page 64
4. “Transform Functions” on page 65
5. “CMM Functions” on page 66 has a subsection “CMM Information Field Parameters.”

The section “Data Structures” on page 67 contains brief descriptions of all of the data structures that are used as arguments of Coloratura functions.

Note: Function names start with “cms”, data type names start with “CMS”, and parameters start with “CMS_”.

Coloratura Access Functions

Function	Description
int32 **cmsClose(**CMSContext *ctxt*)<Function>**cmsClose()**	Frees all allocated memory.
int32 **cmsOpen(**CMSContext **pctxt***)***<Function>cmsOpen()**	Establishes a context for all subsequent calls to the Coloratura CMS.

Profile Functions

Function	Description
int32 **cmsCloseProfile(** CMSContext *ctxt*, CMSProfile *prof*)<Function>**cmsCloseProfile()**	Closes a profile in memory without saving to permanent storage.
int32 **cmsCreateProfile(** CMSContext *ctxt*, CMSProfile **prof*)<Function>**cmsCreateProfile()**	Creates a new, empty profile, with an uninitialized header and no tag data.
int32 **cmsDeleteProfile(** CMSContext *ctxt*, char **name*)<Function>**cmsDeleteProfile()**	Deletes a profile from permanent storage.
int32 **cmsEndProfileIteration(** CMSContext *ctxt*, CMSProfileIterator *profIterator*)<Function>**cmsEndProfileIteration()**	Terminates a profile iteration and disposes of the iterator. Do not use the iterator after disposal.
int32 **cmsExportProfile(** CMSContext *ctxt*, CMSProfile *prof*, uint32 **length*, void ***outputData*)<Function>**cmsExportProfile()**	Generates an external format representation from an open profile.

Function	Description
int32 **cmsFreeProfileExport(** CMSContext *ctxt*, void **outputData*)<Function>**cmsFreeProfileExport()**	Frees storage allocated by **cmsExportProfile()**.
int32 **cmsGetProfileHeader(** CMSContext *ctxt*, cmsProfile *prof*, icHeader **pHeader*)<Function>**cmsGetProfileHeader()**	Gets the header from an open profile.
int32 **cmsGetProfileSpecHeader(** CMSContext *ctxt*, char **spec*, icHeader **pHeader*)<Function>**cmsGetProfileSpecHeader()**	Gets the header from a profile, which need not be open.
int32 **cmsImportProfile(** CMSContext *ctxt*, uint32 *length*, void **inputData*, CMSProfile *prof*)<Function>**cmsImportProfile()**	Generates a new profile from an external format.
int32 **cmsNextProfileIteration(** CMSContext *ctxt*, CMSProfileIterator *profIterator*, char ***spec)*<*Function*>**cmsNextProfileIteration()**	Gets the next profile in an iteration.
int32 **cmsOpenProfile(** CMSContext *ctxt*, char **spec*, CMSProfile **prof*)<Function>**cmsOpenProfile()**	Opens an existing profile for read/write.

Function	Description
int32 **cmsSaveProfile(** CMSContext *ctxt*, CMSProfile *prof*)<Function>**cmsSaveProfile()**	Saves to permanent storage all the modifications since the **cmsOpenProfile()** call. *prof* stays open in memory after a save.
int32 **cmsSaveProfileAs(** CMSContext *ctxt*, CMSProfile *prof*, char **spec*)<Function>**cmsSaveProfileAs()**	Saves a profile to permanent storage under a new name.
int32 **cmsSetProfileHeader(** CMSContext *ctxt*, CMSProfile *prof*, icHeader **pHeader*)<Function>**cmsSetProfileHeader()**	Places a new header in an open profile.
int32 **cmsStartProfileIteration(** CMSContext *ctxt*, CMSProfileIterator **profIterator*)<Function>**cmsStartProfileIteration()**	Creates an iterator to start a profile iteration.

Tag Functions

Function	Description
int32 **cmsDeleteTag(** CMSContext *ctxt*, CMSProfile *prof*, icTagSignature *tagName*)<Function>**cmsDeleteTag()**	Deletes a tag from a profile.
int32 **cmsEndTagIteration(** CMSContext *ctxt*, CMSTagIterator *iterator*)<Function>**cmsEndTagIteration()**	Terminates an iteration and disposes of the iterator.

Function	Description
int32 **cmsFreeTagValue(** CMSContext *ctxt*, icTagSignature *name*, void **data*)<Function>**cmsFreeTagValue()**	Frees tag data returned by **cmsGetTag()**, **cmsGetTagProfileSpec()**, or **cmsNextTagIteration()**.
int32 **cmsGetTag(** CMSContext *ctxt*, CMSProfile *prof*, icTagSignature *name*, icTagTypeSignature **type*, uint32 **size*, void ***data*) <Function>**cmsGetTag()**	Gets data from a profile tag.
int32 **cmsNextTagIteration(** CMSContext *ctxt*, CMSTagIterator *iterator*, icTagSignature **name*, icTagTypeSignature **type*, uint32 **size*, void ***data*)<Function>**cmsNextTagIteration()**	Gets the next tag in an iteration.
int32 **cmsSetTag(** CMSContext *ctxt*, CMSProfile *prof*, icTagSignature *name*, uint32 *size*, void **data*)<Function>**cmsSetTag()**	Sets the tag *name* for the profile *prof*. If *name* doesn't exist in *prof*, creates the tag.
int32 **cmsStartTagIteration(** CMSContext *ctxt*, CMSProfile *prof*, CMSTagIterator **iterator*)<Function>**cmsStartTagIteration()**	Creates an iterator and starts an iteration through all tags in a profile.

Transform Functions

Function	Description
int32 **cmsApplyTfm(** CMSContext *ctxt*, CMSTfm *tfm*, CMSPixelBuffer **psrc*, CMSPixelBuffer **pdst*)<Function>**cmsApplyTfm()**	Applies a transform to convert colors in the pixel buffer *psrc* and place the results in *pdst*.
int32 **cmsCheckGamut(** CMSContext *ctxt*, CMSTfm *tfm*, CMSPixelBuffer **psrc*, unsigned char **pgamutmap)*<Function>**cmsCheckGamut()**	Checks whether transformed pixel colors are in or out of gamut.
int32 **cmsCreateGamutCheck(** CMSContext *ctxt*, int32 *profileCount*, CMSProfile **profs*, icSignature *cmm*, CMSTfm **ptfm*)<Function>**cmsCreateGamutCheck()**	Creates a transform to be used for a gamut check.
int32 **cmsCreateTfm(** CMSContext *ctxt*, int32 *profileCount*, CMSProfile **profs*, icSignature *cmm*, CMSTfm **ptfm*)<Function>**cmsCreateTfm()**	Creates a transform from a set of profiles and a CMM.
int32 **cmsDeleteTfm(** CMSContext *ctxt*, CMSTfm *tfm*)<Function>**cmsDeleteTfm()**	Deletes a transform.

CMM Functions

Function	Description
int32 **cmsFreeCmmList**(CMSContext *ctxt*, icSignature **cmms*)<Function>**cmsFreeCmmList**()	Frees memory allocated for a CMM list.
int32 **cmsGetCmmInfo(** CMSContext *ctxt*, icSignature *cmm*, CMSInfoName *name*, uint32 **value*)<Function>**cmsGetCmmInfo()**	Gets information about a given CMM.
int32 **cmsGetCmmList(** CMSContext *ctxt*, uint32 **count*, icSignature ***cmms*)<Function>**cmsGetCmmList()**	Lists the available CMMs.
int32 **cmsGetDefaultCmm(** CMSContext *ctxt*, icSignature **cmm*)<Function>**cmsGetDefaultCmm()**	Identifies the current default CMM.

CMM Information Field Parameters

cmsGetCmmInfo() uses the CMM information-field names and values in the enumerated type **CMSInfoName**. The values are summarized in the following table.

Field name	Description and Values
CMS_CMM_NAMECMS_CMM_NAME	Registered name uniquely identifying the CMM. The value can be any ICC-registered 32-bit CMM name.
CMS_CMM_VERSIONCMS_CMM_VERSION	Version of the CMM. The value is a **uint32** uniquely distinguishing the version of the CMM from all others

Field name	Description and Values
CMS_FW_VERSIONCMS_FW_VERSION	Version of the Coloratura framework for which the CMM has been programmed. The value is 4 bytes: major version, minor version, revision, 0.
CMS_CAN_DO_ICCMS_CAN_DO_IC	If TRUE, the CMM supports ICC profiles.
CMS_MULTIPLE_OKCMS_MULTIPLE_OK	If TRUE, the CMM supports transformations made from more than two profiles.

Data Structures

Name	Declaration	Description
CMSContext<Function>CMSContext	typedef struct _CMSContext *CMSContext;	Points to an opaque data structure that stores internal working state of the CMS.
CMSInfoName<Function>CMSInfoName	typedef enum { CMS_CMM_NAME, CMS_CMM_VERSION, CMS_FW_VERSION, CMS_CAN_DO_IC, CMS_MULTIPLE_OK } CMSInfoName;	Holds CMM information.
CMSPixelBuffer<Function>CMSPixelBuffer	typedef struct CMSPixelBuffer_s { uint32 *width*; uint32 *height*; uint32 *bitsPerChannel*; uint32 *bytesPerPixel*; uint32 *channels*; uint32 *encode*; unsigned void **data*; } CMSPixelBuffer;	Describes raster pixel data: *width* in pixels *height* in pixels *bitsPerChannel* 1 to 12 *bytesPerPixel* 32-bit boundaries *channels* 1, 3, 4, 5, or 6 *encode* color encoding (see *cms.h*) *data* pointer to data
CMSProfile<Function>CMSProfile	typedef struct _CMSProfile *CMSProfile;	Points to an opaque data structure that identifies profiles in memory.

Name	Declaration	Description
CMSProfileIterator <Function>CMSProfileIterator	typedef struct _CMSProfileIterator *CMSProfileIterator;	Points to an opaque data structure that keeps track of position during profile iterations.
CMSTagIterator<Function>CMSTagIterator	typedef struct _CMSTagIterator *CMSTagIterator;	Points to an opaque data structure that keeps track of position during tag iterations.
CMSTfm<Function>CMSTfm	typedef struct _CMSTfm *CMSTfm;	Points to an opaque data structure that holds transform data.
icHeader<Function>icHeader	This data structure is an enumerated type declared in the file *ic.h*	Lists header entries determined by the *ICC Profile Format Specification*
icSignature<Function>icSignature	This data type is declared in the file *ic.h*	An identifier determined by the *ICC Profile Format Specification* that is typically used to identify CMMs.
icTagSignature<Function>icTagSignature	This data structure is an enumerated type declared in the file *ic.h*	Lists names of tag descriptions determined by the *ICC Profile Format Specification*
icTagTypeSignature <Function>icTagTypeSignature	This data structure is an enumerated type declared in the file *ic.h*	An identifier determined by the *ICC Profile Format Specification*.

Appendix B

Listing of the Application cocoifl

This appendix presents source code for a working color conversion application that uses the Coloratura CMS. A summary of the operations performed appears in "Example Outline of a Color Conversion Program" on page 8. The sections of this appendix correspond to the sections of the summary.

The program, which is called cocoifl and is in */usr/cms/examples*, is written in C++ because it uses the C++ bindings of the Image Format Library (IFL) to manage image files. However, cocoifl does not rely heavily on techniques specific to C++; programmers who know only C can benefit from looking at this example. The IFL also has C bindings, so you could convert cocoifl into a C program. The data flow for cocoifl is the same as that illustrated in Figure 1-2, but without intermediate profiles.

The program cocoifl works; it is not a piece of pseudocode. Therefore, it includes code for manipulating files and error handling, code that is, strictly speaking, irrelevant to the Coloratura CMS.

As you look at the source code, recall that the names of Coloratura objects use the following convention: function names start with "cms", type names start with "CMS", and library constant names start with "CMS_". The names of IFL objects begin with "ifl". In the following discussion, little is said about the IFL objects. For more information about the IFL see the *ImageVision Library Programming Guide* and the reference page IFL(3).

The application takes an input image file and input and destination profiles, and performs a color conversion (hence "coco") on an output image file. The usage is:

cocoifl [-s < src profile> | -a < src profile>] -d <dst profile> -o <outfile> <infile>

With the **-s** option the source profile is always used; with the **-a** option, the source profile is used if the input image does not have an embedded profile. The syntax is similar to that for the color conversion commands supplied by the Coloratura CMS; see the man pages cocogif(1), cocojpeg(1), and cocostiff(1).

The source code for cocoifl contains two functions: an error handler and a main.

Code for Loading Header Files

```
//See page 9
// cocoifl:
//
//    A simple color management program
//    using the Image File Library (IFL)
//

#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <getopt.h>

#include <ifl/iflError.h>
#include <ifl/iflFormat.h>
#include <ifl/iflFile.h>
#include <ifl/iflDataSize.h>
#include <ifl/iflConfig.h>
#include <ifl/iflTileIter.h>
#include <ifl/iflMinMax.h>
#include “ic.h”
#include “cms.h”

// simple function to decode IFL status, print message and exit

void
errorExit(char* prefix, iflStatus err)
{
    char msg[1024];
    iflStatusToString(err, msg, sizeof msg);
    fprintf(stderr, “cocoifl: %s: %s\n”, prefix, msg);
    exit(EXIT_FAILURE);
}
```

Code for Declaring Variables

```
// see page 9
void
main(int argc, char* argv[])
{
    int  c, fPassThru;
    char *nameAnon, *nameSrc, *nameDst, *nameIn, *nameOut;
    CMSProfile<Function>CMSProfile profSrc, profDst, profs[2];
    CMSContext<Function>CMSContext ctxt;
    CMSPixelBuffer<Function>CMSPixelBuffer      pbIn, pbOut;
    CMSTfm<Function>CMSTfm tfm;
    icHeader<Function>icHeader header;
    int32 status;
    iflColorModel cm;
    void* profileData;
    int  profileSize;
    void*  bufOut;
    iflSize pgSizeOut, pgSizeIn;
    fPassThru = FALSE;
```

Code for Opening the Coloratura CMS, the Input Image File, and the Output Profile

```
// see page 9

if( cmsOpen(&ctxt) != CMS_SUCCESS) {<Function>cmsOpen()
       fprintf(stderr, “Can’t open CMS\n”);
       exit(EXIT_FAILURE);
}
profSrc = (CMSProfile) NULL;
while ((c = getopt(argc, argv, “a:d:ho:s:”)) != -1) {
   switch (c) {
   case ‘a’:
   nameAnon = optarg;
   break;

   case ‘d’:
   nameDst = optarg;
   break;

   case ‘o’:
   nameOut = optarg;
   break;
```

```
            case ‘s’:
            nameSrc = optarg;
            break;

            case ‘h’:
            default:
             fprintf(stderr, “Usage: cocoifl <infilename> <outfilename>\n”);
             exit(EXIT_FAILURE);
            }
        }

        iflStatus err;
        iflFile *in;

        if (optind < argc) {
            nameIn = argv[argc-1];
            in = iflFile::open(nameIn, O_RDONLY, &err);
        } else {
             // no filename given, read from stdin
             in = iflFile::open(0, NULL, O_RDONLY, NULL, &err);
        }

        if (in == NULL) errorExit(“Unable to open input file”, err);

        // Set up the output file. Most parameters match the input, but
        // the number of channels and colorModel may change.

        if(cmsOpenProfile(ctxt, nameDst, &profDst) != CMS_SUCCESS) {
            fprintf(stderr, “Can’t open dest profile %s\n”, nameDst);
            exit(EXIT_FAILURE); <Function>cmsOpenProfile()
        }
        cmsGetProfileHeader(ctxt, profDst, &header);
    <Function>cmsGetProfileHeader()
```

Code for Preparing the Output Pixel Buffer and Open an Output Image File

```
// page 10
pbOut.encode = header.colorSpace;
pbOut.channels = 3;
switch (header.colorSpace) {
case icSigXYZData:
case icSigLabData:
case icSigLuvData:
case icSigYCbCrData:
case icSigYxyData:
    cm = iflMultiSpectral;
    break;

case icSigRgbData:
    // this is needed for GIF in particular
    if (in->getColorModel() == iflRGBPalette)
    cm = iflRGBPalette;
    else
    cm = iflRGB;
    break;

case icSigGrayData:
     cm = iflLuminance;
     pbOut.channels = 1;
     break;

case icSigHsvData:
     cm = iflHSV;
     break;

case icSigHlsData:
case icSigCmykData:
     cm = iflCMYK;
     pbOut.channels = 4;
     break;

case icSigCmyData:
     cm = iflCMY;
     pbOut.channels = 3;
     break;
```

```
case icSig2colorData:
     pbOut.channels = 2;
     cm = iflMultiSpectral;
     break;
case icSig15colorData:
    pbOut.channels++;
case icSig14colorData:
    pbOut.channels++;
case icSig13colorData:
    pbOut.channels++;
case icSig12colorData:
    pbOut.channels++;
case icSig11colorData:
    pbOut.channels++;
case icSig10colorData:
    pbOut.channels++;
case icSig9colorData:
    pbOut.channels++;
case icSig8colorData:
    pbOut.channels++;
case icSig7colorData:
    pbOut.channels++;
case icSig6colorData:
    pbOut.channels++;
case icSig5colorData:
    pbOut.channels++;
case icSig4colorData:
    pbOut.channels++;
case icSig3colorData:
    pbOut.channels++;
    cm= iflMultiSpectral;
    break;
}
```

```
iflSize sizeSetup;
in->getSize(sizeSetup, in->getOrientation());
if ( cm != iflRGBPalette)
    sizeSetup.c = pbOut.channels;
iflFileConfig cfgOut = iflFileConfig(&sizeSetup,
                                     in->getDataType(),
                                     in->getOrder(),
                                     cm,
                                     in->getOrientation(),
                                     in->getCompression());
iflFile *out = iflFile::create(nameOut,
                               in,
                               &cfgOut,
                               in->getFormat(),
                               &err);
if (out == NULL) errorExit("Unable to create output file", err);
```

Code for Selecting an Input Profile

```
// see page 10

// Try to open a source profile. Here's the order we search:
// 1) profile specified with -s
// 2) embedded profile
// 3) profile specified with -a
// 4) default profile
// if any of these are specified and fail, we just pass the
// image through unmodified
```

```
if (nameSrc != (char *) NULL) {
    // profile specified with -s is always used
    if(cmsOpenProfile(ctxt, nameSrc, &profSrc) != CMS_SUCCESS) {
       fprintf(stderr, "Can't open source profile %s\n", nameSrc);
       fPassThru = TRUE;<Function>cmsOpenProfile()
     }
} else if (in->getICCProfile(profileSize, profileData) == iflOKAY) {
    // look for embedded profile.
    if (cmsImportProfile(ctxt, (uint32) profileSize, profileData,
                         &profSrc) != CMS_SUCCESS) {
        fprintf(stderr, "Can't open embedded profile");
        fPassThru = TRUE;<Function>cmsImportProfile()
 }
     in->freeICCProfile(profileData);
} else if (nameAnon != (char *)NULL) {
    // look for anonymous profile
    if(cmsOpenProfile(ctxt, nameAnon, &profSrc) != CMS_SUCCESS) {
        fprintf(stderr, "Can't open anonymous profile %s\n", nameAnon);
        fPassThru = TRUE;
     }
} else {
    // look for default profile based on number of image type
    switch(in->getColorModel()) {
    case iflRGB:
             if(cmsOpenProfile(ctxt, CMS_DEFAULT_MONITOR, &profSrc) !=
               CMS_SUCCESS) {
                 fprintf(stderr, "Can't open default profile %s\n",
                   CMS_DEFAULT_MONITOR);
                 fPassThru = TRUE;
             }
             break;
    case iflCMYK:
             if(cmsOpenProfile(ctxt, CMS_DEFAULT_CMYK, &profSrc) !=
               CMS_SUCCESS) {
                 fprintf(stderr, "Can't open default profile %s\n",
                   CMS_DEFAULT_CMYK);
                 fPassThru = TRUE;
             }
             break;
    default:
            fPassThru = TRUE;
            break;
    }
}
```

Code for Creating a Transform and Initializing Buffers

```
// see page 10

if (!fPassThru) {
     profs[0] = profSrc;
     profs[1] = profDst;
     if((status = cmsCreateTfm(ctxt, 2, profs,
         CMS_USE_DEFAULT_CMM, &tfm)) != CMS_SUCCESS) {
           fprintf(stderr, “Can’t create the transform: returned %d\n”,
             status);
           fPassThru = TRUE;<Function>cmsCreateTfm()
     }
}

out->getPageSize(pgSizeOut, out->getOrientation());
pbOut.bitsPerChannel = 8;
pbOut.bytesPerPixel = pbOut.channels;
pbIn.bitsPerChannel = 8;

if (cm == iflRGBPalette) {
    const iflColormap  *cmap;

    in->getColormap(cmap);
    int numChan = cmap->getNumChans();
    iflDataType dataType = cmap->getDataType();
    int length = cmap->getLength();

     if (numChan == 3 && dataType == iflUChar) {
         unsigned char buf[768], *pc;
         unsigned char *pr = (unsigned char *)cmap->getChan(0);
         unsigned char *pg = (unsigned char *)cmap->getChan(1);
         unsigned char *pb = (unsigned char *)cmap->getChan(2);

         // interleave the channels
         pc = buf;
         for (int i = 0; i < length; i++) {
             *pc++ = *pr++;
             *pc++ = *pg++;
             *pc++ = *pb++;
         }
         pbOut.width = length;
         pbOut.height = 1;
         pbOut.data = buf;
```

```
            pbIn.width = length;
            pbIn.height = 1;
            pbIn.channels = 3;
            pbIn.bytesPerPixel = 3;
            pbIn.encode = icSigRgbData;
            pbIn.data = buf;

            if ((status = cmsApplyTfm(ctxt, tfm, &pbIn, &pbOut)) !=
             CMS_SUCCESS){
               fprintf(stderr, "Can't apply tfm: returned %d\n", status);
               exit (EXIT_FAILURE);<Function>cmsApplyTfm()
            }

            unsigned char bufOut[768];
            pr = bufOut;
            pg = pr + 256;
            pb = pg+256;
            pc = buf;
            // repack the channels
            for (i = 0; i < length; i++) {
                 *pr++ = *pc++;
                 *pg++ = *pc++;
                 *pb++ = *pc++;
            }

            iflColormap  cmapOut = iflColormap(bufOut, 3, dataType, 0,
                length -1);
            cmapOut.setData(bufOut);
            out->setColormap(&cmapOut);
        }
        fPassThru = TRUE;
    } else if (!fPassThru) {

        // allocate an output buffer for modified pixels
        int bufsizeOut = iflDataSize(out->getDataType(), pgSizeOut);

        bufOut = new char [bufsizeOut];
        if (bufOut == NULL) {
            fprintf(stderr, "cocoifl: unable to allocate %d bytes\n",
              bufsizeOut);
            exit(EXIT_FAILURE);
        }
        pbOut.data = bufOut;
    }
```

```
// now set up the input
cmsGetProfileHeader(ctxt, profSrc, &header);
<Function>cmsGetProfileHeader()
pbIn.channels = in->getCsize();
pbIn.bytesPerPixel = pbIn.channels; // XXX only 1 byte/channel for now
pbIn.encode = header.colorSpace;

in->getPageSize(pgSizeIn, in->getOrientation());
int bufsizeIn = iflDataSize(in->getDataType(), pgSizeIn);
void* bufIn = new char [bufsizeIn];
if (bufIn == NULL) {
    fprintf(stderr, “cocoifl: unable to allocate %u bytes\n”,
      bufsizeIn);
    exit(EXIT_FAILURE);
}
pbIn.data = bufIn;
if (fPassThru) {
    // We’ll just copy output from input without any color transform
    // on the pixels.  There may already have been a tranform on the
    // colormap.
    pbOut.data = bufIn;
}

int sizeProf;
void *dataProf;
```

Code for Embedding the Output Profile in the Output Image File

```
// see page 11

if (cmsExportProfile(ctxt, profDst, (uint32 *) &sizeProf, (void **)
                     &dataProf) == CMS_SUCCESS)
{<Function>cmsExportProfile()
    (void)  out->setICCProfile(sizeProf, dataProf);
    cmsFreeProfileExport(ctxt,
dataProf);<Function>cmsFreeProfileExport()
}
```

Code for Transforming Pixel Data and Cleaning Up

```
// see page 11
iflSize sizeOut;
```

```
out->getSize(sizeOut, out->getOrientation());
iflConfig config(out->getDataType(), out->getOrder(), sizeOut.c,
                 NULL, 0, out->getOrientation());

iflTileIter iter(iflTile3Dint(0, 0, 0, sizeOut.x, sizeOut.y,
    sizeOut.z), pgSizeOut, sizeOut.c);

while (iter.more()) {
    iflSize rwSize(iflMin(sizeOut.x - iter.x, pgSizeOut.x),
                   iflMin(sizeOut.y - iter.y, pgSizeOut.y),
                   iflMin(sizeOut.z - iter.z, pgSizeOut.z),
                   iflMin(sizeOut.c - iter.c, pgSizeOut.c));

    err = in->getTile(iter.x, iter.y, iter.z,
                      rwSize.x, rwSize.y, rwSize.z,
                      bufIn, &config);
    if (err != iflOKAY) errorExit("Couldn't read page from input",
                                  err);

    if (!fPassThru) {
        // the width and height may change with each tile
        pbOut.width = pbIn.width = rwSize.x;
        pbOut.height = pbIn.height = rwSize.y;
        if ((status = cmsApplyTfm(ctxt, tfm, &pbIn, &pbOut)) !=
          CMS_SUCCESS){
            fprintf(stderr, "Can't apply tfm: returned %d\n", status);
            exit (EXIT_FAILURE);<Function>cmsApplyTfm()
        }
    }

    err = out->setPage(pbOut.data, iter.x, iter.y, iter.z, iter.c,
                       rwSize.x, rwSize.y, rwSize.z, pbOut.channels);
    if (err != iflOKAY) errorExit("Couldn't write page to output",
                                  err);
}
delete[] bufIn;
delete[] bufOut;
err = out->flush();
if (err != iflOKAY) errorExit("Error flushing output file", err);
err = out->close();
if (err != iflOKAY) errorExit("Error closing output file", err);
err = in->close();
if (err != iflOKAY) errorExit("Error closing input file", err);

}
```

Glossary

abstract profile

A profile for making color changes by transforming data within the profile connection space; it does not represent any device. The color space of the input and output data is that of the profile connection space, and so abstract profiles cannot be embedded in an image.

CIE

Commission Internationale de l'Éclairge (International Commission on Illumination).

CMM

See color manipulation module.

CMS

See color management system.

CMYK

Cyan, magenta, yellow, and black primaries, typically used to define color in printers. Black is included to increase the gamut and because it is difficult to get a true black by mixing cyan, magenta, and yellow. Including black also reduces the total amount of ink required and shortens drying times.

color gamut

The range of colors that can be produced by a particular device. When transforming color data from one device to another, the gamuts might not match. This is one source of color distortion. For information about how you can check for this effect, see "Checking Gamut Mapping" on page 53.

color management system

Software to facilitate manipulation of color images and ICC profiles to obtain appropriate transformed images.

color manipulation module

The software that performs the calculations necessary for color transformations.

color space

A system for specifying colors. The human visual system allows the description of all colors with the values of just three parameters; colors thus define an abstract volume. Exactly how coordinate axes are defined in the volume distinguishes color spaces from each other. For example, the set of phosphors and input values for each determines the color space of a monitor. The CIE has defined color spaces that attempt to more accurately reflect human perception of colors; the better known are referred to as CIEXYZ and CIELAB.

device-link profile

A profile that is useful if you repeatedly use a particular series of (device and non-device) profiles that begins and ends with device profiles; it concatenates the series and so defines a one-profile link between devices. Referred to as a DeviceLink profile in the *ICC Profile Format Specification*. Because of its device-specific nature, it does not make sense to embed this profile in an image file.

gamut mapping

If all colors are not mapped within the gamut of the output color space, which is determined by the output profile, a gamut mapping is needed to determine how to transform points in the input color space that would otherwise map out of gamut. Some of the things you can do in response to the need for gamut mapping are: change the input data, change profiles, set a flag.

ICC

International Color Consortium. See *http://www.color.org*, where you can find the *ICC Profile Format Specification*.

primary colors

The colorants that are combined to produce all colors in a device's gamut. Typically three primaries are needed, corresponding to the three dimensions of the color space.

profile

Characterizes a device's color space by specifying a mapping of it into either of two device-independent color spaces, CIELAB or CIEXYZ, developed by the CIE to describe color appearance. The Coloratura CMS uses the profile format described by the *ICC Profile Format Specification*, which can be found at *http://www.color.org*.

profile connection space

A color space based on the human visual system that allows a device independent description of color; it is used to characterize how colors are produced by input and output device color spaces and so allow color translation between devices. A device profile defines a mapping between the device's color space and the profile connection space and is defined by color measurements. The profile connection space uses either CIELAB or CIEXYZ color space. See the *ICC Profile Format Specification*, especially Annex A, for more details.

rendering intent

A tag in a profile to indicate how to reconcile differences between input and output gamuts when image data characterized by the profile are transformed. See the *ICC Profile Format Specification* for more details.

RGB

Red, green, and blue primary colors, typically used to specify colors on monitors or other devices that use additive color.

tag

Subsets of profile data that are defined by the *ICC Profile Format Specification*.

transform

Converts points from one color space to another: typically from the color space of an input device to that of an output device. A transform may include intermediate color adjustments. A transform is defined by a sequence of profiles and the computational algorithm in the CMM.

Index

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3442-001.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389