KCMS Application Developer's Guide

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

© 1995 Sun Microsystems, Inc. 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

Copyright Eastman Kodak Company, 1994. Modified by Sun with permission from Kodak.

All rights reserved. This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® system, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and from the Berkeley 4.3 BSD system, licensed from the University of California. Third-party software, including font technology in this product, is protected by copyright and licensed from Sun's suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19. The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS
Sun, Sun Microsystems, the Sun logo, SunSoft, the SunSoft logo, Solaris, SunOS, OpenWindows, DeskSet, ONC, ONC+, and NFS are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. OPEN LOOK is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc.

All SPARC trademarks are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. SPARCcenter, SPARCcluster, SPARCompiler, SPARCdesign, SPARC811, SPARCengine, SPARCprinter, SPARCserver, SPARCstation, SPARCstorage, SPARCworks, microSPARC, microSPARC-II, and UltraSPARCe licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK® and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a trademark of X Consortium, Inc. Kodak is a trademark of Eastman Kodak Company.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN. THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAMS(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Please
Recycle

Contents

KCMS 1.0 Release Notes xiii

Preface xv

1. Introduction 1

- KCMS Architecture 1
  - Applications 2
  - C API 2
  - KCMS Framework 3
  - Profiles 3
  - Graphics and Imaging Libraries 4
  - Color Management Modules 4
- KCMS File System 5
- Sample Programs 6

2. Profiles 7

- Profile Format 8
- Kinds of Profiles 9

Device Color Profile. 9

Color Space Profile. 9

Effects Color Profile 10

Complete Color Profile 10

Getting and Setting Profile Attributes 10

Loading and Saving Profiles 11

Using Profiles to Convert Color Data 12

Loading Scanner and Monitor Profiles 13

Connecting Scanner to Monitor Profiles 14

Evaluating Color Data Through the Complete Profile 15

Associating Profiles with Devices 16

Using Color Space Profiles 16

Advanced Profile Topics 18

Operation Hints 18

Content Hints 18

Freeing Profiles. 19

Managing Profile Memory 19

Optimizing Profiles 19

Characterizing and Calibrating Profiles 20

3. Data Structures 23

Macros 23

Constants 24

Data Types 24

4. Functions 59

5. KCMS Profile Attributes and ICC Tags 99

Tags 99

Attribute Value 100

Required and Optional Attributes 100

KCMS Framework Tags. 100

`icSigHeaderTag` 100

`icSigNumTag` 100

`icSigListTag` 100

Example 101

Required ICC Tags 102

Tag Types. 107

Constants 108

Signatures 109

Other Enums 114

Arrays of Numbers 117

Tag Type Definitions 125

KCMS-Specific Tag Definitions 131

6. Warning and Error Messages 135

Warnings 135

Errors 136

Localizing Status Messages 141

Glossary 143

Bibliography. 157

Index 159

Figures

Figure 1-1 KCMS Architecture 2

Figure 2-1 Converting Color Data From a Scanner to a Monitor 12

Figure 2-2 Building a CCP From Two DCPs 14

Figure 2-3 Profile Load Hint Operations 15

Figure 3-1 Bit Positions and Masks for Load Hints 41

Figure 3-2 24-bit Color Component-Interleaved Data for RGB Pixel Image 52

Tables

Table 1-1 Optional Imaging and Graphics Libraries 4

Table 1-2 KCMS Directories 5

Table 2-1 KCMS and ICC Profile Format Equivalents 8

Table 3-1 `KcsFunction` Bit Constants 39

Table 3-2 Bit Mask Values for Load Hints 42

Table 3-3 `KcsSampleType` Constants 57

Table 6-1 Warning Codes 135

Table 6-2 Error Codes 136

Code Samples

Code Example 2-1 Simple Color Data Conversion 13
Code Example 2-2 Connecting a DCP and CSP. 17
Code Example 3-1 `KcsAttributeValue`. 28
Code Example 3-2 `KcsCallbackFunction` 33
Code Example 3-3 Load Hint Bit Mask Combinations. 43
Code Example 4-1 `KcsConnectProfiles()` 62
Code Example 4-2 `KcsCreateProfile()`. 65
Code Example 4-3 `KcsEvaluate()`. 67
Code Example 4-4 `KcsFreeProfile()` 68
Code Example 4-5 `KcsGetAttribute()`. 69
Code Example 4-6 `KcsGetLastError()`. 73
Code Example 4-7 `KcsLoadProfile()` 76
Code Example 4-8 `KcsModifyLoadHints()` 79
Code Example 4-9 `KcsOptimizeProfile()` 82
Code Example 4-10 `KcsSaveProfile()` 84
Code Example 4-11 `KcsSetAttribute()`. 87

Code Example 4-12 `KcsSetCallback()` 91
Code Example 4-13 `KcsUpdateProfile()`. 95
Code Example 5-1 `icSigNumTag` and `icSigListTag` 101

KCMS 1.0 Release Notes

See the on-line SUNWrdm packages for information on bugs and issues, engineering news, and patches. For Solaris installation bugs and for late-breaking bugs, news, and patch information, see the *Solaris Installation Notes* (SPARC™ or x86).

For SPARC systems, consult any updates your hardware manufacturer provides.

Multithread Unsafe

In this release, KCMS does not support multithread programs; it is multithread unsafe (MT-unsafe). If your application uses multithread capabilities you must put locks around KCMS library calls.

Preface

The *KCMS Application Developer's Guide* describes the Kodak Color Management System (KCMS) framework application programming interface. The purpose of the KCMS framework is to enable the accurate reproduction, and improve the appearance of, digital color images on desktop computers and associated peripherals. With this C API, you can write applications that perform correct color conversions and manipulations.

Who Should Use This Book

The intended audience of this manual is the professional programmer who is fluent in the C programming language and writing an application that:

- Uses color data
- Prints images
- Is an imaging tool
- Uses PhotoCD

Before You Read This Book

Check the following manuals for any corrections or updates to the information in this manual:

- *Solaris 2.5 Software Developer Kit Introduction*
- *Solaris 2.5 Software Developer Kit Installation Guide*

See the on-line SUNWrdm packages for information on bugs and issues, engineering news, and patches. For Solaris installation bugs and for late-breaking bugs, news, and patch information, see the *Solaris Installation Notes* (SPARC or x86).

For SPARC systems, consult any updates your hardware manufacturer provided.

Although you do not have to be a color scientist to write applications with the KCMS API, a certain amount of color literacy is helpful. Table P-1 lists two white papers that contain some basic information on color and KCMS. The files are located online in the `/usr/openwin/demo/kcms/docs/` directory.

Table P-1 KCMS White Papers

File Name	Title
`kcms-wp.ps`	*An Introduction to the Kodak Color Management System*
`kcms-wp-solaris.ps`	*Kodak Color Management System*

The KCMS framework this manual describes uses the International Color Consortium (ICC) format as the default format for color manipulation. For details on ICC, you should read the *International Color Consortium Profile Format Specification*. It is located by default in the `icc.ps` file in the `/opt/SUNWsdk/kcms/doc` directory. This manual refers to that document as the ICC specification.

Related Manuals

The following manuals will help you further understand the Driver Developer Kit (DDK) portion of the KCMS software product. These manuals are located in the DDK AnswerBook.

- *KCMS CMM Developer's Guide*
- *KCMS CMM Reference Manual*

The following manuals will help you further understand the Calibrator Tool portion of the KCMS software product.

- *Solaris Advanced User's Guide*

  In Chapter 10, "Customizing Your Environment," there is a section called "Calibrating Your Monitor." The section tells you how to adjust your viewing environment and how to calibrate your monitor with Calibrator Tool. This manual is in the Solaris 2.5 User AnswerBook.

- *KCMS Calibrator Tool Loadable Interface Guide*

  This manual will help you further understand the API to the Calibrator Tool. You can tailor the Calibrator Tool for your specific calibrator hardware and software with this API. This manual is in the KCMS AnswerBook.

Suggested Reading

It is highly recommended that you be familiar with, or have access to, the following manuals and manual pages to help you with topics discussed in this manual:

- `setlocale(3c)`
- *Solaris Developer's Guide to Internationalization*

How This Book Is Organized

This document consists of the following chapters and appendix:

- Chapter 1, "Introduction" explains the KCMS architecture and programming environment. In addition, it introduces you to several online sample programs that demonstrate the use of the KCMS API.

- Chapter 2, "Profiles" explains profiles, which are the focus of your programming efforts with the KCMS framework.

- Chapter 3, "Data Structures" describes the data structures of the KCMS framework.

- Chapter 4, "Functions" details each API function.

- Chapter 5, "KCMS Profile Attributes and ICC Tags" details each profile attribute and ICC tag.

- Chapter 6, "Warning and Error Messages" describes status codes (error and warning messages) returned by the KCMS framework functions.

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

Table P-2 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

API Naming Conventions

The naming conventions shown in Table P-3 are used throughout the KCMS framework and this guide.

Table P-3 API Naming Conventions

Item	Convention	Examples
Attribute names	ICC profile format attribute names begin with "`ic`"—`ic`<*AttributeName*>	`icHeader`
Data structures Typedefs Constants	ICC profile format data structures begin with "`ic`". All other data structures, typedefs, and constants are KCMS specific and begin with "Kcs"—`Kcs`<*TypeDefName*>	`icTextDescription` `KcsCalibrationData`

Table P-3 API Naming Conventions *(Continued)*

Item	Convention	Examples
Functions	Each significant word in a function name is capitalized. Intervening spaces are removed—`Kcs`<*FunctionName*>()	`KcsConnectProfiles()`
Macros	Macros are KCMS specific and are capitalized—`KCS_`<*MACRO_NAME*>	`KCS_DEFAULT_ATTRIB_COUNT`
Status codes	All status codes are capitalized and have the format `KCS_`<*STATUS_CODE*>	`KCS_PROF_ID_BAD`

Note – Historically KCMS was referred to by the acronym *KCS* (or *Kcs*). This acronym has been carried forward as the prefix in KCMS data type names, for example, KcsCalibrationData.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Table P-4 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

Introduction

1

This chapter introduces you to the Kodak Color Management System (KCMS) product. It describes each of the components of the KCMS architecture and tells you about programming requirements and hints when writing your KCMS application.

KCMS Architecture

The KCMS architecture provides a way to encapsulate specific color management functions in color profiles. Figure 1-1 illustrates the architecture of the KCMS environment. Each segment filled with gray is supplied by SunSoft; these are the default components. The other segments, filled with white, are components that you can add to your development environment.

Each component is discussed further in the following sections.

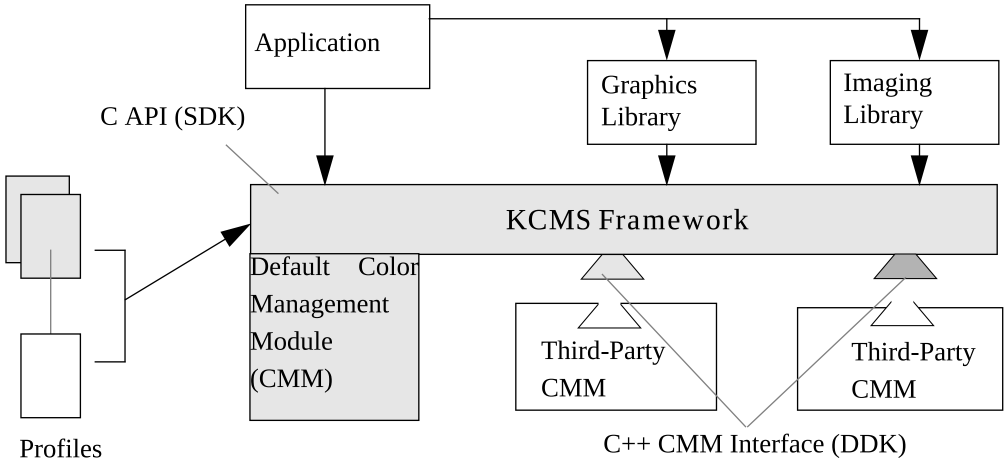


Figure 1-1 KCMS Architecture

Applications

At the top of the hierarchy are applications. With the KCMS framework you can write an application that:

- Uses color data
- Prints
- Is an imaging tool
- Uses PhotoCD

Applications connect color profiles to provide a variety of new forms, thus minimizing the task of predefining all possibilities. With the 14 available KCMS API functions, your application can load, create and update profiles, connect and optimize profiles, and then process data through the result.

C API

The KCMS C API provides functions for your application to communicate with the KCMS framework and color management modules (CMMs). The C API is a portable programming interface that allows applications to manipulate color profiles and to use them to correct color data.

The C API consists of:

- A set of callable functions
- Header files
- A shared library and dynamically loaded code modules required for Solaris

KCMS Framework

The KCMS framework loads and saves profiles, gets and sets KCMS profile attributes, and directs requests for color management to the right CMM at the right time. It is particularly vital in calls that involve more than one CMM. The KCMS framework also maintains attributes and executes certain default behaviors and functionality.

Color management is performed by the framework and the CMMs. You can concentrate on dealing with profiles because the KCMS framework makes color management details transparent to the caller.

Profiles

Profiles are files that tell the KCMS framework how to convert input color data to the appropriate *color-corrected* output color data. They will be the focus of your programming efforts. For example, your application might load profiles, read profile attributes, connect profiles, optimize profiles, and apply profiles to color data.

See Chapter 2, “Profiles,” for detailed information.

Graphics and Imaging Libraries

Table 1-1 lists some of the imaging and graphics libraries available to use with the KCMS framework.

Table 1-1 Optional Imaging and Graphics Libraries

Library	**Description**
PEXlib	PHIGS Extensions to the X Library
XGL	Solaris 3D Graphics Foundation Library
XIElib	X Imaging Extension Library
XIL	Solaris Foundation Imaging Library
Xlib	X11 Window System Library

You can mix KCMS calls with any calls from these libraries. If the library you choose supports color management, your application may not need to make direct calls to the KCMS framework; the other library may already make those direct KCMS calls. See the documentation with the imaging and graphics library of your choice to see if that library already supports color management.

Color Management Modules

A color management module (CMM) is the component that ultimately does the color correction. Different CMMs use different techniques for evaluating color data, which can result in differences in quality, size, and speed of color correction.

Because CMMs are loaded at run-time and CMM interfaces are extendable, you can take advantage of the improvements in existing technologies and the latest color-correction technology, along with hardware acceleration, (without changing your code or rebuilding your application) by changing or adding new CMMs, or profiles, or both.

A Kodak-supplied CMM is the default CMM. You can write your own CMM (third-party CMM) or override portions of the default CMM. To write your own CMM you must purchase the Solaris Device Developer's Kit (DDK) that includes the following KCMS CMM manuals:

- *KCMS CMM Developer's Guide*
- *KCMS CMM Reference Manual*

KCMS File System

The software product's directory structure indicates the types and locations of files. Table 1-2 shows you the top-level directories.

Table 1-2 KCMS Directories

Directory	**Subdirectory**	**Content**
`/usr/openwin`	`bin`	Configuration and networking binaries
	`demo/kcms`	KCMS demonstration programs
	`demo/kcms/images/tiff`	Sample TIFF images
	`demo/kcms/docs`	KCMS user white papers
	`lib`	`libkcs.so`; main KCMS library
	`share/etc/gpiutils`	CMM libraries
	`share/etc/devhandlers`	Dynamically loadable modules and third-party CMMs
	`share/etc/devdata/profiles`	Device profiles provided with KCMS
	`include/kcms`	Various library header files
	`man/man1`	KCMS command / utility manual pages
	`man/man6`	KCMS demo manual pages
`SUNWsdk/kcms`	`demo`	Sample programs
	`doc`	ICC specification
	`man/man3`	KCMS API manual pages
	`man/man6`	KCMS demo manual pages
	`src`	Sample source code
	`xi_lib`	XIL-based library to read and write TIFF files

Sample Programs

Several sample programs demonstrate how to use the API described in this manual. These programs are available online in the `SUNWsdk/kcms/demo` directory. The programs show you how to

- Check profile calibration (`kcms_update.c`)
- Test the loading of a scanner profile and a monitor profile, and correct the color image data (`kcstest.c`)
- Print header attributes in a profile (`print_attributes.c`)

The `/demo` directory also provides files used in the sample programs. These include

- `kcms_create.c`
- `kcmstest_tiff.c`
- `kcms_timer.c`
- `kcms_utils.c`
- `kcms_utils.h`
- `print_header.c`
- `print_montbls.c`

Check the `README_SDK` file for additional information.

Profiles

2

Profiles (also called color profiles) provide the KCMS framework with information on how to convert input color data to the appropriate color-corrected output color data. They are the focus of your programming efforts. A typical application loads profiles, reads profile attributes, connects profiles, optimizes profiles, and applies profiles to color data. You will probably combine or connect existing profiles to create profiles, rather than generate new ones.

Profiles include the following information:

- Color spaces in which the input and output data appears (for example, RGB, CMYK, or XYZ).
- Specific parameters of the color spaces (for example, the chromaticities of the primary colors and the tables to correct for the response of each channel).
- Specific conditions in which the colors are expected to be viewed (for example, the lighting conditions and type of media that will be used).
- Tables of data or parameters of equations that a CMM uses to transform color data. Each profile is owned by a specific CMM. Although all profiles have common, public information, some of an individual profile’s format can be CMM-specific. (You do not need to understand the profile file format to write applications.)

KCMS, by default, uses the International Color Consortium (ICC) profile format. The ICC format is an emerging default defacto standard supported by a wide range of computer and color device vendors. This is extremely advantageous for users, as a single profile will work over multiple platforms.

Note – The ICC format is currently endorsed by the following companies: Adobe Systems Inc., Agfa-Gevaert N.V., Apple Computer Inc., Eastman Kodak Company, FOGRA (Honorary), Microsoft Corporation, Silicon Graphics, Inc., Sun Microsystems Inc., and Taligent Inc.

Profile Format

The KCMS framework uses the ICC format as the default profile format. For details on the ICC profile format, see the ICC specification. By default, it is located online in the `SUNWsdk/kcms/doc` directory.

There are some terminology differences between ICC profiles and the KCMS framework. These differences are mostly historical. Table 2-1 lists the equivalent ICC name if it is different.

Table 2-1 KCMS and ICC Profile Format Equivalents

KCMS Profile Format	**ICC Equivalent**
Device Color Profile	Any Input, Display, or Output Profile
Color Space Profile	Color Space Conversion Profile
Effects Color Profile	Abstract Profile
Complete Color Profile	Device Link Profile

Each color profile is owned by, or associated with a specific CMM. In general, you do not need to know which CMM owns the profile. In the case where a profile's CMM is not present and the profile is a valid ICC profile, the default CMM will provide the functionality necessary to use that profile. However, each CMM does have different ways of performing its color-correction technology. For example, each CMM has a unique way to calibrate its profiles. In addition, you may occasionally receive error codes that are CMM-specific.

For more information on CMMs, see the DDK document, *KCMS CMM Developer's Guide*.

Kinds of Profiles

The KCMS framework supports several kinds of color profiles.

Device Color Profile

A *Device Color Profile* (DCP) represents the behavior of a specific digital color device, such as a flatbed or film scanner, a computer monitor, or a printer. Each DCP specifies device color appearance under a specific set of conditions (for example, lighting type, media type, and so on). Because device behavior tends to change over time, calibration software may adjust a DCP whenever its device is calibrated. *Calibration* refers to fine tuning a specific device's color response. Typically it changes the profile data so that it can be color managed to produce the same color response as other devices of the same make and model. In other cases, depending on the device's method of calibration, the device itself is changed to match the profile.

The ICC specification separates DCPs into three categories: input, output and display. This separation can be confusing when a device, such as a printer includes input device data. The data can be considered an input profile, an output profile, or both. This occurs in print simulation where the printer is an input device to a display or other output device.

Conceptually, it may be easier to separate profiles into these three categories only in terms of how data can and cannot be sent from and to the *profile connection space* (PCS). The PCS is the common junction where profiles are connected together.

KCMS does not make this syntactical separation. Rather it considers all input, output, and display profiles as device profiles and makes no assumptions about what profiles can and cannot be connected together. The connection of the profiles is then evaluated at connection time based on the data contained within the profile.

Color Space Profile

A *Color Space Profile* (CSP) defines a color space. Colors are defined in terms directly related to spectral response. A CSP does not depend on the behavior of a particular color device. CSPs contain information about assumed viewing conditions in the data expressed for that color space. Typically, the color space can be relative to CIEXYZ values, defined by the Commission Internationale de l'Eclairage (CIE). The equivalent ICC term for Color Space Profile is *Color Space Conversion Profile*. (See Table 2-1.)

Effects Color Profile

An *Effects Color Profile* (ECP) represents a condition that changes the appearance of colors, such as a specific kind of lighting or a simulated anomalous color vision (color blindness). In addition, an ECP can be applied for artistic purposes, such as making colors appear lighter or darker. The equivalent ICC term for Effects Color Profile is *Abstract* profile. (See Table 2-1.)

Complete Color Profile

The preceding three profile types do not contain enough information for the KCMS framework to convert color data from one form to another. Useful color transformations can only happen when your application uses the KCMS API to connect two or more profiles together to form a *Complete Color Profile* (CCP). A CCP is a connected sequence of profiles with a DCP or a CSP at either end, and possibly one or more ECPs or DCPs in between. The equivalent ICC term for Complete Color Profile is D*evice Link Profile*. (See Table 2-1.)

Getting and Setting Profile Attributes

The KCMS API provides a way to get profile information by examining the profile's *attribute set*. Each attribute has a value, which is data associated with the attribute. The C API provides the following attribute calls:

- `KcsGetAttribute()`—gets a specific attribute value associated with a profile. See "KcsGetAttribute()" on page 69 for detailed information.
- `KcsSetAttribute()`—modifies an attribute. (This is not always possible because some attributes are read-only.) See "KcsSetAttribute()" on page 86 for detailed information.

For more information on profile attributes, see Chapter 5, "KCMS Profile Attributes and ICC Tags."

Loading and Saving Profiles

Profiles are typically stored as files on disks, although they can be imbedded in an image located across a network or in read-only memory in a printer.

Profiles are loaded with the `KcsLoadProfile()` function (see page 74) and are saved with the `KcsSaveProfile()` function (see page 83). `KcsLoadProfile()` takes the three arguments listed below. `KcsSaveProfile()` takes the first two arguments listed.

- A profile identifier
- A profile description
- Hints about loading the profile

The *profile identifier* is returned to the calling program from `KcsLoadProfile()` for use with other API functions. In the case of `KcsSaveProfile()`, the identifier is passed back into the KCMS framework library to indicate the profile to be saved.

The *profile description* is a union of many different types, each of which represents a way to supply a location where the profile data should be stored. The `type` and the associated fields in the union are required to complete a profile description. The `type` field indicates which of the union's fields to use.

A calling program can request that the KCMS framework load only specific parts of a profile, (for example, just its attributes). The caller uses the `KcsModifyLoadHints()` function to provide these *load hints*, which change the load status of the profile. Hints are described by the `KcsLoadHints` data type discussed on page 40. Load hints that request specific operations and specific content be loaded for a profile are described in “Operation Hints” on page 18.

Using Profiles to Convert Color Data

The example in Figure 2-1 shows how color data is converted between a scanner device and a monitor device.

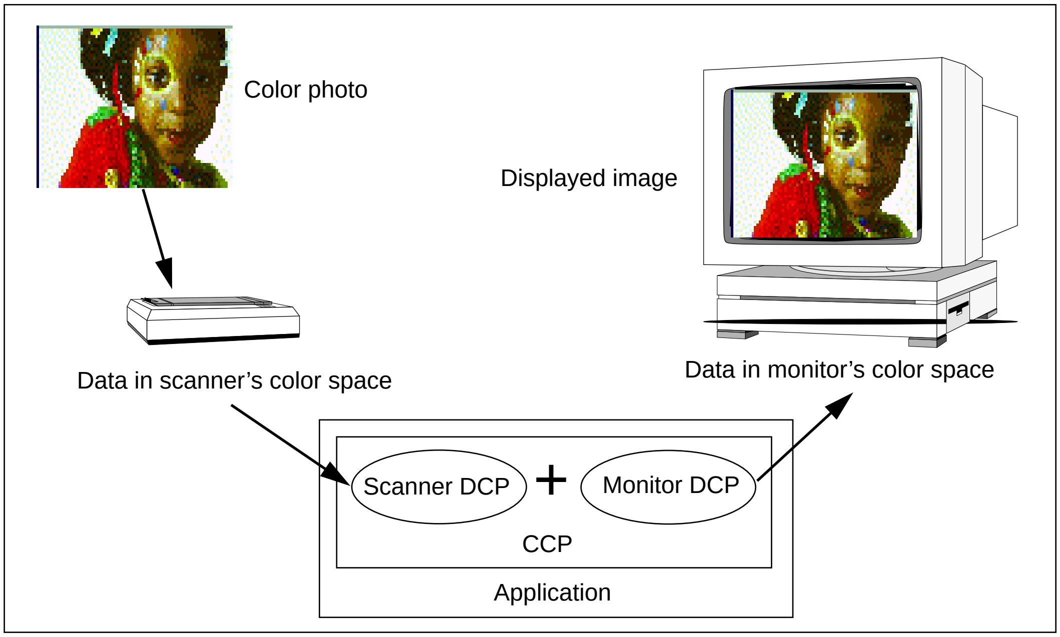


Figure 2-1 Converting Color Data From a Scanner to a Monitor

In this example, these devices do not perform their own color correction; therefore, the color data must be converted from the form provided by the scanner (Scanner DCP) to a form appropriate for display on the monitor (Monitor DCP). Follow these steps to convert the color data:

1. Load scanner and monitor profiles.
   See "Loading Scanner and Monitor Profiles" on page 13."
2. Connect a scanner profile to amonitor profile to get a complete profile.
   See "Connecting Scanner to Monitor Profiles" on page 14.
3. Evaluate color data through the complete profile.
   See "Evaluating Color Data Through the Complete Profile" on page 15.

The sequence of calls that perform this conversion is shown in Code Example 2-1. For more information on the `KcsConnectProfiles()` function, see "Using Color Space Profiles" on page 16 and the detailed function description on page 61.

Code Example 2-1 Simple Color Data Conversion

```
/* Load the scanner's DCP.*/
KcsLoadProfile(&inProfile, &scannerDescription, KcsLoadAllNow);

/* Load the monitor's DCP. */
KcsLoadProfile(&outProfile, &monitorDescription, KcsLoadAllNow);

/* Connect two DCPs to form a CCP */
profileSequence[0] = inProfile;
profileSequence[1] = outProfile;
KcsConnectProfiles(&completeProfile, 2, profileSequence,
       KcsLoadAllNow, &failedProfileIndex);

/* Apply the CCP to input color data. */
KcsEvaluate(completeProfile, KcsOperationForward, &inbufLayout,
       &outbufLayout);
```

Loading Scanner and Monitor Profiles

The `KcsLoadProfile()` function loads the profile associated with a specific device, effect, partial or complete profile. It allocates any system resources required by the profile. For a detailed description of the `KcsLoadProfile()` function, see page 74.

Connecting Scanner to Monitor Profiles

As shown in Code Example 2-1 and Figure 2-2, the next stage is to connect a pair of DCPs to form a CCP. `KcsConnectProfiles()` provides this functionality. Continuing with the example illustrated in Figure 2-1, a CCP is built by connecting the scanner's DCP to the monitor's DCP. The resulting CCP converts scanner data to monitor data.

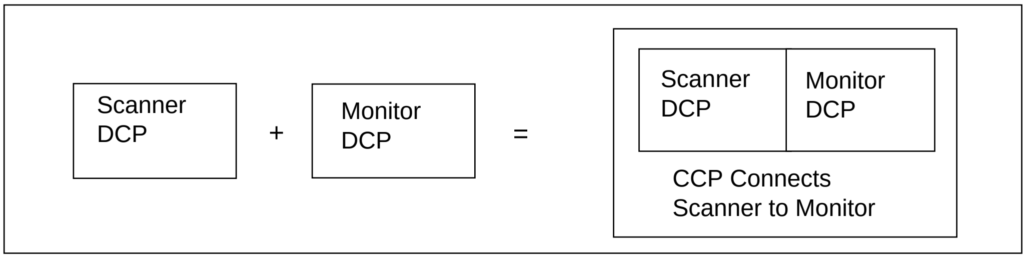


Figure 2-2 Building a CCP From Two DCPs

Evaluating Color Data Through the Complete Profile

Next the `KcsEvaluate()` function is used to apply a color transformation based on the supplied CCP. One of the following operations is associated with the evaluation. These operations are illustrated in Figure 2-3.

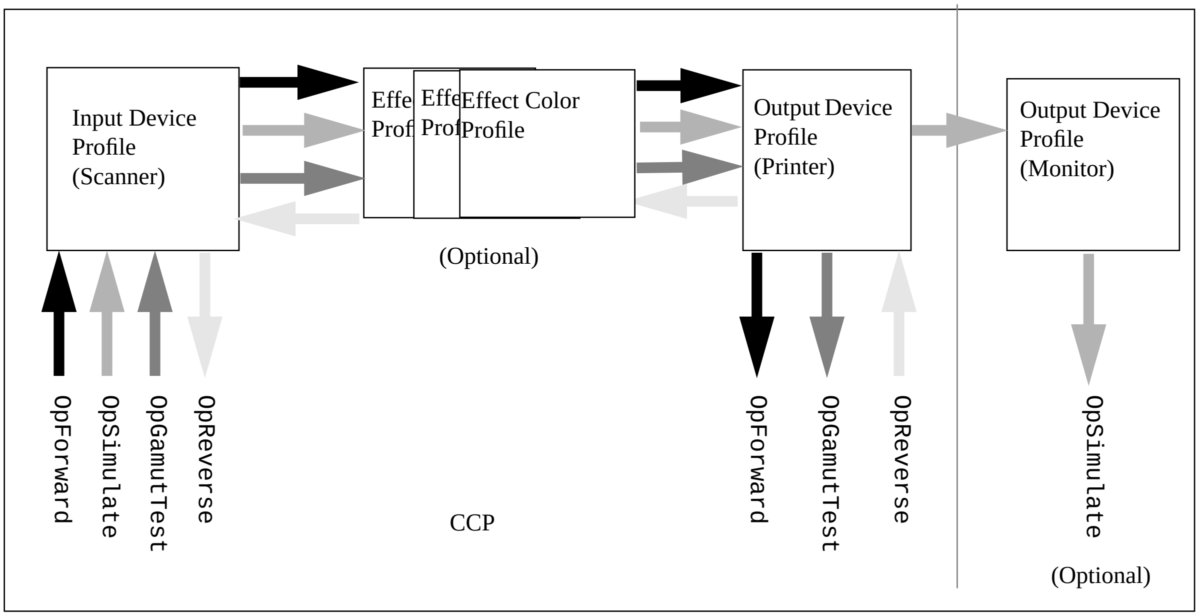


Figure 2-3 Profile Load Hint Operations

- `OpForward`—The forward operation is used to transform color from the scanner form to the monitor form.
- `OpReverse`—The reverse operation is used to transform color from the monitor form to the scanner form. This is useful if your application modifies some colors in monitor space, to keep the greatest number of colors that can be converted back and stored in the scanner’s color space.

  A more familiar use of the reverse operation is to transform the color from printer to monitor form to see what the data looks like from the printer.

- `OpSimulate`—The simulate operation is used to simulate the effect of running color data through a CCP, but retaining it in the form of the last device profile. For example, the simulate operation can produce monitor data that simulates the result of printed data.
- `OpGamutTest`—The gamut-test operation is used to determine if each color in the source data is within the gamut of the destination device. Physical devices have a range of colors they can produce. This range of colors is known as the gamut of the device.

`KcsEvaluate()` can take a long time to execute, especially if the input image or graphic contains millions of pixels. Therefore, you can provide a callback function using `KcsSetCallback()`, which `KcsEvaluate()` calls when necessary. The callback function can, for example, provide feedback to request that processing be cancelled. If the callback returns a non-KCS_SUCCESS status, the processing stops.

Associating Profiles with Devices

The `KcsSaveProfile()` function, when supplied a `KcsProfileDesc` structure, associates that color profile with the supplied structure. Typically, a configuration or calibration program calls `KcsSaveProfile()`. The profile associated with the `KcsDescription` structure represents the last calibrated condition of the device. For more information about the `KcsSaveProfile()` function, see page 83.

Many events can change the condition of a device. For example, as room lighting changes, so does the viewer's perception of a monitor's colors. As another example, consider a color printer; when different kinds of paper are used in the printer, the printer's color condition changes. When conditions change, a user may associate a different profile with the device.

Using Color Space Profiles

Another possible use of `KcsConnectProfiles()` is to connect a DCP and a CSP, creating a new CCP. Refer to Figure 2-1 on page 12. If the scanner DCP in this figure is connected to the CSP (instead of the Monitor DCP shown that converts for the CIE XYZ color space), the resulting CCP will convert color data produced by the scanner into CIE XYZ format.

Code Example 2-2 shows the sequence of calls that creates and applies the CCP. Note that this example is very similar to Code Example 2-1 on page 13. The difference is the second call to `KcsLoadProfile()`. In Code Example 2-2, `KcsLoadProfile()` loads the CIE XYZ profile description instead of the monitor description.

Code Example 2-2 Connecting a DCP and CSP

```
/*Load scanner’s DCP. */
KcsLoadProfile(&inProfile, &scannerDescription, KcsLoadAllNow);
if(status != KCS_SUCCESS) {
    status = KcsGetLastError(&errDesc);
    KcsFreeProfile(profileid);
    exit(1);
}

/*Load CSP for CIE XYZ color space. */
KcsLoadProfile(&outProfile,&CIEXYZdescription, KcsLoadAllNow);
if(status != KCS_SUCCESS) {
    status = KcsGetLastError(&errDesc);
    KcsFreeProfile(profileid);
    exit(1);
}

/*Connect two profiles to form a CCP.*/
profileSequence[0] = inProfile;
profileSequence[1] = outProfile;
KcsConnectProfiles(&completeProfile,2, profileSequence,
        KcsLoadAllNow, &failedProfileIndex);
if(status != KCS_SUCCESS) {
    status = KcsGetLastError(&errDesc);
    KcsFreeProfile(profileid);
    exit(1);
}

/*Apply the CCP to input color data.*/
KcsEvaluate(completeProfile, KcsOperationForward,
        &inbufLayout, &outbufLayout);
```

Advanced Profile Topics

Operation Hints

`KcsEvaluate()` takes an additional argument that describes the operation to be performed on the profile; it is an operation hint. For example, you can tell `KcsEvaluate()` to convert data in the forward direction (`KcsOpForward`), such as from the scanner to the printer. You may also convert data in the reverse direction, such as from the monitor to the scanner. The reverse operation (`KcsOpReverse`), when it is available in a profile, inverts the function performed by `KcsOpForward`. However, it rarely performs an exact inverse because information is lost when color data is transformed. In other words, if you perform a `KcsOpForward` and then a `KcsOpReverse` of a profile on the same buffer, the result is almost equivalent to what you started with before `KcsOpForward`; some quality may be lost.

Only one of these operation hint bits can be set at one time for `KcsEvaluate()`, unlike general load hints, for which any combination can be set at the same time. As part of the `KcsLoadHints` data type, the operation hints signify the required set of operations available to use with the profile. By contrast, `KcsEvaluate()` uses only the single operation that the application wants to perform.

See "Operation Hint Constants" on page 43 for more information on operation hints.

Content Hints

You can also specify hints about the content of the data being processed; for example, photographic image data or computer-generated graphic image data. A CMM can use these hints to do a better job of converting the data; for example, the CMM can use these hints to adjust the gamut-mapping technique.

See "Content Hint Constants" on page 44 for more information on content hints.

Freeing Profiles

After creating a complete profile, you can use it more than once; for example, to convert images page-by-page during printing and to process individual rasters or tiles in a large image. Once your program no longer needs the profile, use `KcsFreeProfile()` to free the profile's resources. The profiles in the profile sequence used to create a CCP can be freed without affecting the CCP.

Managing Profile Memory

The C API expects the application to allocate memory required for the data returned by the KCMS framework. In general, the application allocates a C structure and passes a pointer to that structure into the KCMS framework.

The one exception to this is the profile; the KCMS framework returns and accepts an identifier only. Memory allocated for the identifier must be managed by your application. Call `KcsFreeProfile()` in your application to inform the KCMS framework to release the memory associated with the profile identifier.

Optimizing Profiles

Once a color profile has been loaded, a CMM may be able to optimize it. With `KcsOptimizeProfile()`, you can optimize a profile (an individual profile or a CCP) in two ways:

- First, you can optimize a profile to make it more accurate (by eliminating intermediate round-off errors, for instance), smaller (by merging sequences of look-up tables, for instance), or faster (by precomputing some results). The application specifies whether size, speed, accuracy, or some combination is more important.
- Second, by using load hints to limit a profile's operations, you may also affect its optimization. This is valuable, for instance, if you want to write color data with a DCP that will be used later to read the data. The size of the DCP can be significantly reduced (depending on the CMM in use) by restricting the profile to the forward operation only.

Because optimization can take a long time, the application can provide a callback, similar to the one used with `KcsEvaluate()`.

After optimizing a profile, call `KcsSaveProfile()` to save it for future use. Then use this profile with `KcsLoadProfile()` to avoid the slow performance of `KcsOptimizeProfile()`.

There are some potential implications when saving an optimized profile. The optimization may indirectly affect future operations on the profile. For example, if the profile is optimized for size, portions of the profile needed only for highest accuracy may be discarded, resulting in compromised accuracy.

Note – Another way to *optimize* a profile is to use the operation flags supplied when using `KcsConnectProfiles()`. `KcsConnectProfiles()` allows you to specify which operations and content hints should be supported when you connect a sequence of profiles. Reducing the set of operations by supplying the content hint can make `KcsConnectProfiles()` run faster and make the resulting profile smaller.

Characterizing and Calibrating Profiles

Characterization establishes a norm for a particular device across a range of samples of the device. This form of profile is typically supplied by a Profile vendor. To obtain an optimally accurate DCP for a particular device, calibration is required.

Calibration makes measurements of an individual device and applies them to the base DCP. The updated DCP represents the actual color device the customer is using.

The KCMS API provides two API functions, `KcsCreateProfile()` and `KcsUpdateProfile()`, to create new blank profiles and then update them with characterization data or calibration data.

The first step in building a new profile is to create an empty profile using `KcsCreateProfile()`. Fill the empty profile with `KcsSetAttribute()` to describe the device being characterized; for example, supply monitor chromaticities and white-point values. Measurement data is required for `KcsUpdateProfile()` to complete the creation of the new profile. Once updated, save the profile with `KcsSaveProfile()` to the desired `KcsProfileDesc` location.

Updating profiles is typically a CMM-dependent operation. The use of measurement data at the KCMS framework interface level frees you from details of the profile format and the process by which the CMM turns the measurement data into its methodology for color correction.

The default CMM supports characterization and calibration of monitors and scanners.

Data Structures

3

This chapter details data structures in the C API that are common to many functions. These data structures are categorized by macros, constants, and data type definitions. Data structures are listed alphabetically and defined in the `kcs.h`, `kcstypes.h`, and `kcsstats.h` header files.

Note – The `kcstypes.h` header file includes a color space addition for gray scale profiles. See the format of the `KcsAttrSpace` structure on page 30 for details.

Data structures relevant only to attributes are defined in Chapter 5, "KCMS Profile Attributes and ICC Tags."

Macros

The following macros are used in the C API:

```
#define KCS_DEFAULT_ATTRIB_COUNT(data_type
    ((sizeof (KcsAttributeValue) -
    sizeof (KcsAttributeBase)) / sizeof (data_type))
```

Constants

The following constants are used in the C API:

```
#define KcsAttrStrLength         256
#define KcsExtendableArray       4
#define KcsExtendablePixelLayout 4
#define KcsExtendableMeasSet     4
#define KcsForceAlign            0x7FFFFFFF
#define KcsMaxSamples            4
#define KcsMaxPatches            8
```

Data Types

`KcsAttributeBase`

```
typedef struct KcsAttributeBase_s {
    KcsAttributeType   type;
    unsigned long      countSupplied;
    unsigned long      countAvailable;
    unsigned long      sizeOfType;
    char               strVal[KcsAttrStrLength];
} KcsAttributeBase;
```

The `KcsAttributeBase` structure defines a common subset of information in the `KcsAttributeValue` structure. Nothing in `KcsAttributeBase` is extendable.

The `type` field determines the data type in which the attribute value is stored. It is the `icSigxxxType` as defined in the `icc.h` and `kcstypes.h` header files.

The `countSupplied` field specifies the number of allocated elements in the array. For example, if `type` is set to `KcsDoubleValue` and `countSupplied` is set to `2`, the attribute value is large enough to hold two doubles, which are stored in the first two elements of the `doubleVal` array of `KcsAttributeValue` (see page 27).

When the `type` field is set to `KcsString`, `KcsDateTimeStamp`, or an ic type defined in the header file `icc.h`, the `countSupplied` field must be set to `1` because strings are treated as a single token.

Note – `KcsDateTimeStamp`, `KcsDoubleValue`, and `KcsString` are equated to ic types in the header.

To determine how many values of a particular data type that can fit in a `KcsAttributeValue` structure, use the `KCS_DEFAULT_ATTRIB_COUNT` macro. It returns the number of values of the specified data type that will fit in the structure. You must set the `countSupplied` field of the `KcsAttributeBase` structure to the number of values to get or set before you call `KcsGetAttribute()` or `KcsSetAttribute()`. Upon return of `KcsGetAttribute()`, the `countAvailable` field specifies the number of values in the profile.

The `sizeOfType` field is the value, array or structure indicated by `type`:

```
attrValuePtr->base.type = icSigHeaderType;
attrValuePtr->base.sizeOfType = sizeof(icHeader);
```

OR

```
attrValuePtr->base.type = icSigMeasurementType;
attrValuePtr->base.sizeOfType = sizeof(icMeasurement);
```

The `KcsAttrStrLength` field is defined in the `kcstypes.h` header file as the maximum string length of 256.

KcsAttributeName

```
typedef long KcsAttributeName;
```

`KcsAttributeName` is used in several functions as the tag argument.

KcsAttributeType

```
typedef enum KcsAttributeType_s {
    /* InterColor types map to KcsTypes... */
    KcsString           = 2,    /* Original; different than ictext! */
    KcsDateTimeStamp    = 9,    /* Original. Different from ‘dtim’*/
    KcsUByte            = icSigUInt8ArrayType,    /* ‘ui08’ */
    KcsUShort           = icSigUInt16ArrayType,   /* ‘ui16’ */
    KcsULong            = icSigUInt32ArrayType,   /* ‘ui32’ */
    /* Signed types follow the InterColor convention... */
    KcsByte             = icSigSInt8ArrayType,    /* ‘si08’ */
    KcsShort            = icSigSInt16ArrayType,   /* ‘si16’ */
    KcsLong             = icSigSInt32ArrayType,   /* ‘si32’ */
    KcsDouble           = icSigSFlt64ArrayType,   /* ‘sf64’ */
    /* A few KCMS-specific */
    KcsPixelLayoutSupported = icSigPixelLayoutSType, /* ‘play’ */
    KcsAlias            = icSigAliasType,         /* ‘lias’ */

    /* To avoid conflict with the icTagTypeSignature enum in */
    /* icc.h, the following list of enums is commented out.*/
    /* They do represent valid KcsAttributeType enums. */
    .
    .
    .
    /* Old pre-ICC types. */
    .
    .
    .
    KcsAttrTypeMax      = KcsForceAlign
} KcsAttributeType;
```

`KcsAttributeType` is the data type of one field in the `KcsAttributeBase` structure. It is the name of the data type for a particular attribute’s value. It is an enumerated type. See “KcsAttributeBase” on page 24 for more information.

KcsAttributeValue

```
typedef struct KcsAttributeValue_s {
    KcsAttributeBase      base;
    union KcsAttributeValueValue_s {
        struct tm          dateTimeVal;
        long               longVal[KcsExtendableArray];
        double             doubleVal[KcsExtendableArray];
        char               byteVal[KcsExtendableArray];
        unsigned char      uByteVal[KcsExtendableArray];
        short              shortVal[KcsExtendableArray];
        unsigned short     uShortVal[KcsExtendableArray];
        unsigned long      uLongVal[KcsExtendableArray];
        KcsPixelLayoutSpeeds layoutVal[KcsExtendablePixelLayout];
        /* ICC 3.0 values */
        icText             icText;
        icData             icData;
        icCurve            icCurve;
        icUcrBg            icUcrBg;
        icNamedColor       icNamedColor;
        icScreening        icScreening;
        icSignature        icSignature;
        icMeasurement      icMeasurement;
        icDateTimeNumber   icDateTime;
        icViewingCondition icViewingCondition;
        icTextDescription  icTextDescription;
        icProfileSequenceDesc icProfileSequenceDescription;
        icXYZArray         icXYZ;
        icInt8Array        icInt8Array;
        icInt16Array       icInt16Array;
        icInt32Array       icInt32Array;
        icInt64Array       icInt64Array;
        icUInt8Array       icUInt8Array;
        icUInt16Array      icUInt16Array;
        icUInt32Array      icUInt32Array;
        icUInt64Array      icUInt64Array;
        icS15Fixed16Array  icS15Fixed16Array;
        icU16Fixed16Array  icU16Fixed16Array
        icHeader           icHeader
    } KcsAttributeValueValue;
} KcsAttributeValue;
```

Note – The `KcsAttributeValueValue` data type is included in this type definition.

The `KcsAttributeValue` structure is the data type of one argument in:

- `KcsGetAttribute()`
- `KcsSetAttribute()`

A variable of data type `KcsAttributeValue` holds the value of an attribute. An attribute's value fits in a normal `KcsAttributeValue` structure. However, you may have to extend the `KcsAttributeValue` structure if the number of values an attribute contains is greater than the number in the default size of the structure. The C API macro `KCS_DEFAULT_ATTRIB_COUNT` returns the values that a variable of this type can hold. (For more information on `KCS_DEFAULT_ATTRIB_COUNT`, see the description of `KcsAttributeBase` on page 24.) For example, to have more values in an attribute than the value returned from the macro, you can extend the structure by allocating more memory and then casting it as a pointer to a `KcsAttributeValue` structure. Because it is specified as an array at the end of the structure, and C does not check array bounds, you can allocate a piece of memory larger than `KcsAttributeValue` and treat the extra memory as an extension of the `val` arrays. This allows you to access the values by using the array operator (`myAttributeValuePtr->val.doubleVal[i]`).

For example, the following code shows you how to get the colorant from a profile:

Code Example 3-1 `KcsAttributeValue`

```
/* Get the colorants */
/* Red */
KcsAttributeValue*attrValuePtr;

attrValuePtr = (KcsAttributeValue *)malloc(sizeof(KcsAttributeBase) +
        sizeof(icXYZNumber) );
attrValuePtr->base.type = icSigXYZArrayType;
attrValuePtr->base.countSupplied = 1;
status = KcsGetAttribute(profileid, icSigRedColorantTag, attrValuePtr);
if(status != KCS_SUCCESS) {
    status = KcsGetLastError(&errDesc);
    printf(“GetAttribute error: %s\n”, errDesc.desc);
    KcsFreeProfile(profileid);
    exit(1);
```

Code Example 3-1 `KcsAttributeValue` *(Continued)*

```
}

XYZval = (icXYZNumber *)attrValuePtr->val.icXYZ.data;
printf(“Red X=%f Y=%f Z=%f\n”, icfixed2double(XYZval->X, icSigS15Fixed16ArrayType),
       icfixed2double(XYZval->Y, icSigS15Fixed16ArrayType), icfixed2double(XYZval->Z,
       icSigS15Fixed16ArrayType));
/* Green */
status = KcsGetAttribute(profileid, icSigGreenColorantTag, attrValuePtr);
if(status != KCS_SUCCESS) {
   status = KcsGetLastError(&errDesc);
   printf(“SetAttribute error: %s\n”, errDesc.desc);
   KcsFreeProfile(profileid);
   exit(1);
}

XYZval = (icXYZNumber *)attrValuePtr->val.icXYZ.data;
printf(“Green X=%f Y=%f Z=%f\n”, icfixed2double(XYZval->X, icSigS15Fixed16ArrayType),
       icfixed2double(XYZval->Y, icSigS15Fixed16ArrayType), icfixed2double(XYZval->Z,
       icSigS15Fixed16ArrayType));

/* Blue */
status = KcsGetAttribute(profileid, icSigBlueColorantTag, attrValuePtr);
if(status != KCS_SUCCESS) {
   status = KcsGetLastError(&errDesc);
   printf(“SetAttribute error: %s\n”, errDesc.desc);
   KcsFreeProfile(profileid);
   exit(1);
}

XYZval = (icXYZNumber *)attrValuePtr->val.icXYZ.data;
printf(“Blue X=%f Y=%f Z=%f\n”, icfixed2double(XYZval->X, icSigS15Fixed16ArrayType),
       icfixed2double(XYZval->Y, icSigS15Fixed16ArrayType), icfixed2double(XYZval->Z,
       icSigS15Fixed16ArrayType));
free(attrValuePtr);
```

If an attribute returns just one `long` value, use the following code fragment:

```
KcsAttributeValue myAttributeValue;
myAttributeValue.base.countSupplied = 1;
KcsGetAttribute(myProfile, myAttributeName, &myAttributeValue);
```

KcsAttrSpace

```
typedef enum {
    KcsSpaceUnknown,        /* Unknown* /
    KcsRGB,                 /* RGB */
    KcsPhotoCDYcc,          /* Photo CD Ycc */
    KcsUVLStar,             /* uvL */
    KcsCMY,                 /* CMY */
    KcsCMYK,                /* CMYK */
    KcsRCS,                 /* RCS */
    KcsGray,                /* Gray scale*/
    KcsCIEXYZ,              /* CIE XYZ */
    KcsCIELAB,              /* CIE LAB */
    KcsCIELUV,              /* CIE LUV */
    KcsLogExp,              /* Log Exposure interchange space */
    KcsAttrEnd,
    KcsAttrSpaceMax = KcsForceAlign
}KcsAttrSpace;
```

`KcsAttrSpace` defines the `inputSpace` and `outputSpace` fields of the `KcsMeasurementBase` structure. (See the format of this structure on page 46.)

KcsCalibrationData

```
typedef struct KcsCalibrationData_s {
    KcsMeasurementBase aBase;
    union {         /* Place holder */
         long Pad;
    } oBase;
    union {
        KcsMeasurementSample patch[KcsExtendableMeasSet];
    } val;
} KcsCalibrationData;
```

`KcsCalibrationData` holds a set of data used by `KcsUpdateProfile` to update a profile that has been calibrated or, in the case of scanners, characterized. (For more information on calibration and characterization, see “Characterizing and Calibrating Profiles” on page 20. Also see the description of the `KcsUpdateProfile()` function on page 93.)

The `KcsCalibrationData` structure contains `aBase`, `oBase` (currently not used) and `val`.

The field `aBase` is a `KcsMeasurementBase` structure. It contains fields that apply to all the calibration measurements.

The field `val` is a union that may contain a `KcsMeasurementSample` extendable structure, or some other measurement structure that another CMM may require. The `KcsMeasurementSample` structure is expected by the default KCMS CMM. (See the detailed description of KcsMeasurementSample on page 46.) When allocating memory for a `KcsCalibrationData` structure, allocate sufficient memory to extend the `KcsMeasurementSample` structure so that it can contain the number of measurements corresponding to the field `countSupplied` in the `KcsMeasurementBase` structure. In addition, the color space of these measurements must correspond to the enumerated values in the `inputSpace` and `outputSpace` fields of the `KcsMeasurementBase` structure. These spaces and the expected range of values for the measurements are defined in Chapter 4, “Functions.”

KcsCallbackFunction

```
typedef KCS_CALLBK (KcsStatusId) (KCS_PTR KcsCallbackFunction)
(KcsProfileId  profile,
    unsigned long  current,
    unsigned long  final,
    KcsFunction    callingFunc,
    void KCS_PTR   userDefinedData);
```

`KcsCallbackFunction` is the data type of one argument to `KcsSetCallback`. It is a pointer to a function returning `KcsStatusId`.

Note – The `profile` field is currently undefined.

A `KcsCallbackFunction` variable holds a pointer to a callback that you supply, not the C API. The callback tells you how far certain lengthy operations (such as `KcsEvaluate()` and `KcsOptimizeProfile()`) have progressed. If these operations are too slow, you can provide a way to terminate them. Use `KcsSetCallback()` in your application for each function for which a callback is needed.

Code Example 3-2 on page 33 demonstrates a callback to the potentially time-consuming `KcsOptimizeProfile()` function. In the example, `KcsSetCallback` sets `myCallbackFunc`, a variable of type `KcsCallbackFunction`, as the callback that `KcsOptimizeProfile()` calls. While executing, `KcsOptimizeProfile()` periodically calls `myCallbackFunc`, passing it the following arguments:

- `profile`—a reference to the profile.
- `current`—an integer value that tells you how many times (minus one) `KcsOptimizeProfile()` has called `myCallbackFunc`. The first time `myCallbackFunc` is called, `KcsOptimizeProfile()` sets the value of `current` to `0`; the second time it sets `current` to `1`, and so on.
- `final`—a positive integer that indicates the number of times (plus one) `myCallbackFunc` will ultimately be called (assuming you do not cancel the operation before completion). You can set this argument if you know how many times you want `myCallbackFunc` to be called. Use `final` to get a percent complete number or an indication of an endless loop. When `current` = `final`, the optimization is terminated.
- `callingFunc`—the identity of the function currently executing.
- `userDefinedData`—a pointer that can be any user-definable item.

Code Example 3-2 `KcsCallbackFunction`

```
main()
{
    KcsCallbackFunction myCallbackFunc;
    ...
    status=KcsSetCallback(KcsOptimizeFunc, myCallbackFunc,
            userDefinedData);
    status=KcsOptimizeProfile(profile, optimizationType, loadHint);
    ...
}

/* KcsOptimizeProfile will call myCallbackFunc periodically. This is a
 * simple progress monitoring function; your own progress monitoring
 * function will probably be far more sophisticated. */
KcsStatusId myCallbackFunc (KcsProfileId profile,
        unsigned long current, unsigned long final,
        KcsCallbackFunction CallingFunc, void* userDefinedData);
{
    printf(“The call is %d percent complete.\n”, (current*100)/final);
    return(KCS_SUCCESS);
}
```

If the application returns `KCS_SUCCESS` from the callback function, the C API allows the operation in progress to continue. If the callback function returns any other `KcsStatusId` value, the operation terminates, returning the status value returned from the callback function as its own status. The C API provides a status value, `KCS_OPERATION_CANCELLED`, that the callback function can use to indicate that the operation was terminated by the user.

KcsCharacterizationData

```
typedef struct KcsCharacterizationData_s {
    KcsMeasurementBaseaBase;
    union {         /* Place holder */
         long pad;
    } oBase;
    union {
        KcsMeasurementSample patch[KcsExtendableArray];
    } Val;
} KcsCharacterizationData;
```

`KcsUpdateProfile()` uses data in `KcsCharacterizationData` to update a recharacterized profile. Note that monitor device profiles do not require a `KcsCharacterizationData` structure to be recalibrated by the default KCMS CMM because the profiles use white-point and colorants. However, scanner device profiles do require one. Another CMM may require that this structure be defined for updating a monitor profile.

The description of fields of this structure are the same as the `KcsCalibrationData` structure.

KcsColorSample

```
typedef enum {
    KcsBlack,
    KcsWhite,
    KcsNeutral,
    KcsFluorescent,
    KcsChromatic,
    KcsSampleTypeEnd = KcsForceAlign
} KcsColorSample;
```

`KcsColorSample` defines the `sampleType` field in `KcsMeasurementSample`. (For the format of `the KcsMeasurementSample` structure, see page 46.)

KcsComponent

```
typedef struct KcsComponent_s {
    char             *addr;
    KcsSampleType    compType;
    unsigned long    compDepth;
    long             bitOffset;
    long             rowOffset;
    long             colOffset;
    unsigned long    maxRow;
    unsigned long    maxCol;
    double           rangeStart;
    double           rangeEnd;
} KcsComponent;
```

`KcsComponent` describes the data structure used in `KcsPixelLayout` for a channel or component of color. There is one `KcsComponent` for each channel. For example, three of these structures are required to describe RGB data; four are required to describe CMYK data.

The `addr` field defines the actual memory address of the first pixel of the channel or component.

The `compType` field defines the data type of a channel. For example, given RGB data in which each of the three channels of the input data is represented as an unsigned 8-bit number, you specify `KcsCompUFixed` with a component depth of 8.

The `compDepth` field specifies the number of bits used to represent the component. With respect to memory layout, neither the range of values represented nor the data encoding is relevant. The memory layout determines how the data is accessed; interpreting the data is a higher-level operation.

The `bitOffset` field, if set to `0`, signifies that the component is byte-aligned. If it is not set to 0, then non-byte-based components are described. This allows, for example, a 5-5-5 RGB pixel encoding (that is, 5 bits for each channel).

The `rowOffset` field is the offset between the beginning of a component for one pixel and the beginning of the same component for the pixel in the same column of the next row. It is expressed in units of bits or, if `compDepth` is a multiple of 8, in bytes.

Similarly, the `colOffset` field is the offset between the beginning of a component for one pixel and the beginning of the same component for the pixel in the next column of the same row. The pixels need not be contiguous in memory. The offset is expressed in units of bits or, if `compDepth` is a multiple of 8, in bytes.

The `maxRow` and `maxCol` fields specify the number of rows and columns to process. If you want to apply the profile to the entire bitmap, specify the number of rows and columns (y-size and x-size) of the entire bitmap.

The `rangeStart` and `rangeEnd` fields specify values representing minimum and maximum intensities.

See “KcsPixelLayout” on page 49 and Figure 3-2 on page 52 for more information on how component data is stored in memory.

KcsCreationDesc

```
typedef struct KcsCreationDesc_s {
    KcsCreationType    type;
    KcsProfileDesc     KCS_PTR profileDesc;
    union {
        struct id_f {
            KcsIdent    cmmId;
            KcsIdent    cmmVersionId;
            KcsIdent    profileId;
            KcsIdent    profileVersionId;
        } id;
        long pad[4];    /* maximum size of union */
    } desc;
} KcsCreationDesc;
```

This structure is used as an argument to the `KcsCreateProfile()` function. It contains all of the necessary information to describe the CMM and the profile format used when creating the empty profile and the location of that profile.

`type` indicates which member of the `desc` union to use in creating the profile. This union is intended to be extendible for future use.

`profileDesc` is a pointer to a `KcsProfileDesc` structure describing the source from which the profile is created. If this entry is `NULL,` the profile is created internally and a `KcsProfileDesc` must be supplied if the profile is to be saved to an external store.

The members of the `id` structure are all 4-byte signatures that specify the identification (`cmmId`) and version (`cmmVersionId`) of the CMM to be used. The members also specify the identification (`profileId`) and version (`profileVersionId`) of profile format to be used.

If the `id` structure field members are not available or are set to 0, the default profile format and default CMM are used.

KcsCreationType

```
typedef enum {
    KcsIdentifierSpec      = 0x49640000, /* Id */
    KcsCreationTypeEnd     = 0x7FFFFFFF,
    KcsCreationTypeMax     = KcsForceAlign
} KcsCreationType
```

This enumerated type is used to indicate which member of the `KcsCreationDesc` union to use in creating a profile.

KcsErrDesc

```
typedef struct KcsErrDesc_s {
    KcsStatusId    statId;
    long           sysErrNo;
    char           desc[256];
} KcsErrDesc;
```

`KcsErrDesc` contains useful information about an error.

The `statId` field contains the `KcsStatusId`. If the error was an I/O error, the `sysErrNo` field of `KcsErrDesc` contains the error number returned by the operating system. The `desc` field contains the description for the particular `statId`, for example, “Internal Color Processor Error.” or “No description for this status id number.”

KcsEvalSpeed

```
typedef long KcsEvalSpeed;
```

`KcsEvalSpeed` is a metric in `KcsPixelLayoutSpeeds` that estimates how fast a CMM performs evaluations for a particular pixel layout on a standard machine for the given platform. The metric is measured in pixels per second, where a pixel is comprised of all channels of data. For example, a pixel is 24 bits for an 8-bit RGB and 32 bits for an 8-bit CMYK.

KcsFileId

```
typedef int KcsFileId;
```

`KcsFileId` is a field of the `KcsProfileDesc` data structure (see page 54). It identifies an open file to read with `KcsLoadProfile()`, or to write with `KcsSaveProfile()`.

To get a `KcsFileId`, use the `open(2)` system call.

If the load hints specify anything other than `KcsLoadNow`, or if you intend to save the profile, the file associated with `KcsFileId` must be left open.

KcsFunction

```
typedef unsigned long KcsFunction;
```

`KcsFunction` is the data type of one argument in the signature of a callback function ("KcsCallbackFunction" on page 31) and a data type of one argument in `KcsSetCallback()`. A variable of this data type indicates the function currently executing.

The bits in this integer have particular meanings, as listed in Table 3-1.

Table 3-1 `KcsFunction` Bit Constants

Definition	**Function**
`#define KcsEvalFunc (1<<0)`	`KcsEvaluate()`
`#define KcsFreeFunc (1<<1)`	`KcsFreeProfile()`
`#define KcsGetAttrFunc (1<<2)`	`KcsGetAttribute()`
`#define KcsLoadFunc (1<<3)`	`KcsLoadProfile()`
`#define KcsConnectFunc (1<<4)`	`KcsConnectProfiles()`
`#define KcsOptFunc (1<<5)`	`KcsOptimizeProfile()`
`#define KcsModLoadHintsFunc(1<<6)`	`KcsModifyLoadHints()`
`#define KcsSaveFunc (1<<7)`	`KcsSaveProfile()`
`#define KcsSetAttrFunc(1<<8)`	`KcsSetAttribute()`
`#define KcsUpdateFunc (1<<9)`	`KcsUpdateProfile()`
`#define KcsCreateFunc (1<<10)`	`KcsCreateProfile()`
`#define KcsAllFunc (0xFFFFFFFF)`	All Function Calls

`KcsIdent`

```
typedef long KcsIdent;
```

`KcsIdent` is a type used throughout the C API. A `KcsIdent` variable holds identifiers and version numbers used by the KCMS framework and CMMs. It is typically encoded as four bytes in the readable ASCII range. For example, a KCMS CMM might be identified by `0x4B434D53` (a `long`) or `KCMS` (a `char`). This is identical to the ICC `typedef icSig` defined in the `icc.h` header file.

KcsLoadHints

```
typedef unsigned long KcsLoadHints;
```

`KcsLoadHints` is a data type of one argument in the following functions:

- `KcsConnectProfiles()`
- `KcsCreateProfile()`
- `KcsOptimizeProfile()`
- `KcsLoadProfile()`
- `KcsModifyLoadHints()`

`KcsLoadHints` gives the KCMS framework a hint as to how a profile's allocated resources should be managed. It lets the caller supply information to the KCMS framework about what, how, when, and where to load and unload the profile. It consists of a set of bit definitions that allow the application to supply more than one option. `KcsLoadHints` also lets the application mix the operation hints and content hints for greater flexibility.

Figure 3-1 shows the bits positions (31–0) of an unsigned `long` representing `KcsLoadHints` and `KcsOperationType`. See Table 3-1 and the bits each mask occupies for more information.

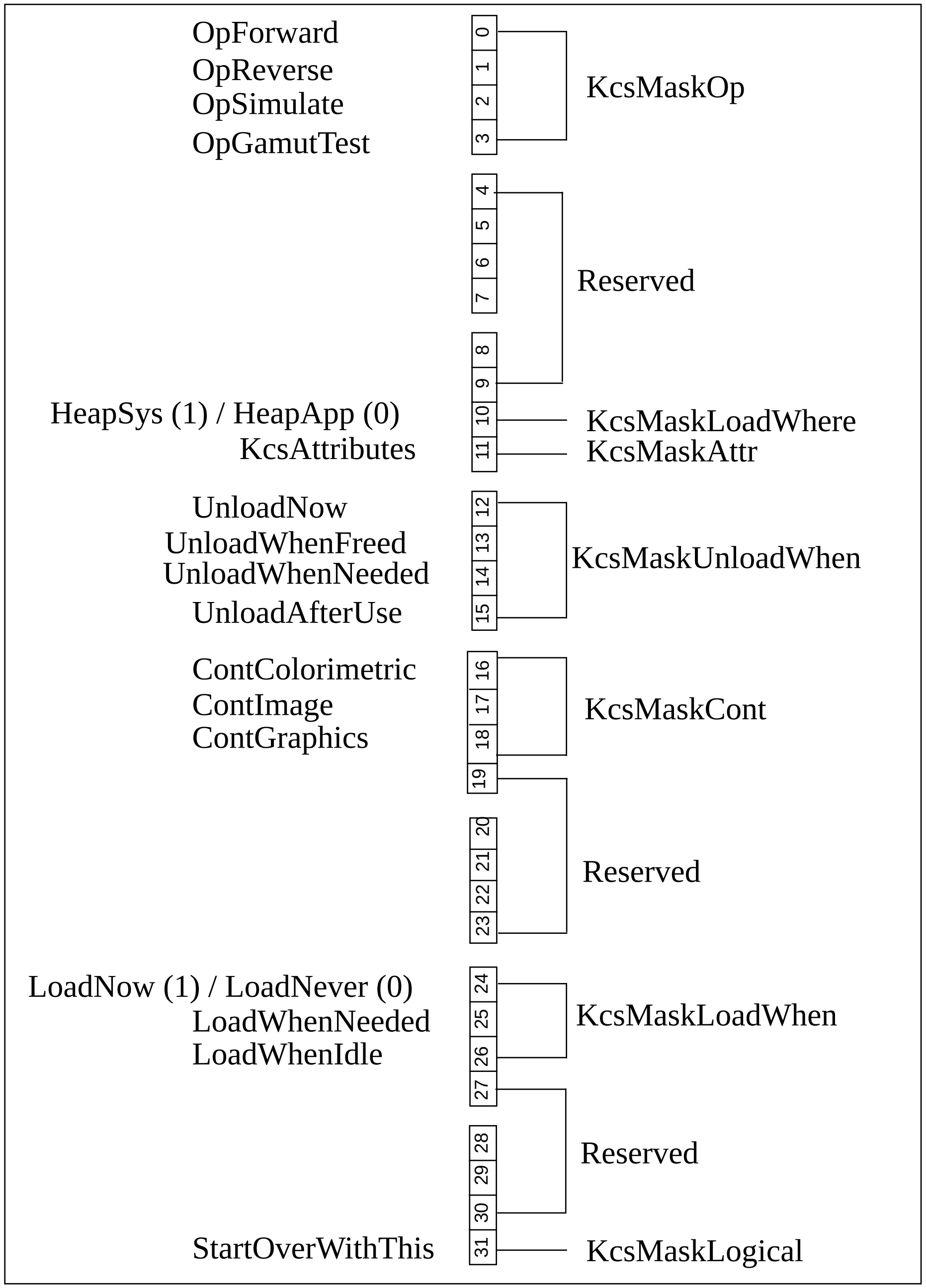


Figure 3-1 Bit Positions and Masks for Load Hints

Table 3-2 lists the values for the load hint bit masks.

Table 3-2 Bit Mask Values for Load Hints

Load Hint Bit Masks	Values	Description

`KcsMaskOp`	`#define KcsOpForward (0x00000001)`
	`#define KcsOpReverse (0x00000002)`
	`#define KcsOpSimulate (0x00000004)`
	`#define KcsOpGamutTest (0x00000008)`
	`#define KcsOpAll (0x000003FF)`
		See "Operation Hint Constants" on page 43.

`KcsMaskEffect`	`#define KcsEffect (0x00000200)`	
`KcsMaskLoadWhere`	`#define KcsHeapApp (0)`	Load it into application heap.
	`#define KcsHeapSys (0x00000400)`	Load it into system heap.
`KcsMaskAttr`	`#define KcsAttributes (0x00000800)`	Load attributes.
`KcsMaskUnloadWhen`	`#define KcsUnloadNow (0x00001000)`	Unload it now.
	`#define KcsUnloadWhenFreed (0x00002000)`	Unload it during a call to `KcsFreeProfile`.
	`#define KcsUnloadWhenNeeded (0x00004000)`	Unload it when the CMM needs the memory for something else.
	`#define KcsUnloadAfterUse (0x00008000)`	Unload it just after the CMM needs to reference it.

`KcsMaskCont`	`#define KcsContUnknown (0x00000000)`
	`#define KcsContGraphics (0x00010000)`
	`#define KcsContImage (0x00020000)`
	`#define KcsContColorimetric (0x00040000)`
	`#define KcsContAll (0x00FF0000)`
		See "Content Hint Constants" on page 44.

`KcsMaskLoadWhen`	`#define KcsLoadNever (0x00000000)`	Never load it.
	`#define KcsLoadNow (0x01000000)`	Load it now.
	`#define KcsLoadWhenNeeded (0x02000000)`	Load it just before CMM needs to reference it.
	`#define KcsLoadWhenIdle (0x04000000)`	Load it when the system has a free moment.
`KcsMaskLogical`	`#define KcsStartOverWithThis (0x10000000)`	Get rid of the previous Hints and start with this one.
	`#define KcsAddToCurrentHints (0x00000000)`	Logically add this Hints with the others already set.

Code Example 3-3 shows some combinations of the masks.

Code Example 3-3 Load Hint Bit Mask Combinations

```
#define KcsLoadAllNow
(KcsAll|KcsLoadNow|KcsUnloadWhenFreed|KcsStartOverWithThis)
#define KcsLoadAllWhenNeeded
(KcsAll|KcsLoadWhenNeeded|KcsUnloadWhenFreed|KcsStartOverWithThis)
#define KcsLoadAttributesNow
(KcsAttributes|KcsLoadNow|KcsUnloadWhenFreed|KcsStartOverWithThis)
#define KcsLoadMinimalMemory
(KcsAll|KcsLoadWhenNeeded|KcsUnloadAfterUse|KcsStartOverWithThis)
#define KcsPurgeMemoryNow
(KcsAll|KcsLoadWhenNeeded|KcsUnloadNow|KcsStartOverWithThis)
```

Operation Hint Constants

The four *operation hint constants* describe the operations in Figure 3-1 on page 41 that can be performed on CCPs to transform color data: forward, reverse, simulate, and gamut-test.

Ordinarily, the application transforms data in the forward direction, for example, from a scanner to a printer. You specify `KcsOpForward` to achieve this.

You also may be able to convert the data in the reverse direction, for example, from a monitor to a scanner. Specify `KcsOpReverse` for this. The reverse direction can be useful if, for instance, you are given colors in the monitor device color space and you want to transform the data back to the original scanner color space.

`KcsOpSimulate` lets you simulate the effect of running data through a complete profile, but leaves it in the color space of the last device profile in the connected sequence of profiles. For instance, suppose you have a CCP consisting of scanner → printer→ monitor profiles. You can use the CCP with the simulate operation on monitor data to produce monitor data that simulates the result of printing the data. For this to work, you must have connected a destination device to a source → destination combination. In this situation, the scanner is the source device, the printer is the first destination device, and the monitor is the connected destination device.

Note – A typical color monitor can display colors that a printer cannot print. Similarly, many printers are capable of printing colors that cannot be displayed on a color monitor. `KcsOpSimulate` lets users preview what a graphic or image will look like (approximately) when printed.

`KcsOpGamutTest` lets you determine if each source color is in the gamut of the destination device.

Because of constraints in the CMM or in the specific profile, not all of the above operations may be supported. Also, some CMMs may offer additional custom operations. You can use `KcsGetAttribute` and supply the `KcsAttrSupportedOperations` attribute to determine which operations are supported by a given profile.

If you specify `KcsOpAll` when loading or making a profile, the resultant profile will have the full range of operations available to it. If you do not, the resultant profile will be restricted to the operations supplied by the function.

You cannot specify `KcsOpAll` as an argument to `KcsEvaluate()`.

Content Hint Constants

The *content hint* constants let you specify hints about what kind of data is being processed. A CMM can use these hints to better convert the data as you requested. For instance, these hints may be used to adjust the gamut-mapping technique (the approach used to map the colors falling outside a device's capability to colors that the device can produce).

The C API defines the following constants:

- `KcsContImage` describes photographic data, photorealistic data, or some 3-dimensional rendering schemes. In this kind of data, fine gradations of luminance and relative color differences are important.
- `KcsContGraphics` describes computer-generated color data, which is likely to have large flat regions of highly saturated colors. In graphics data, an attempt is made to maintain the brightness and distinctness of the colors.
- `KcsContColorimetric` describes colors in terms of CIE specifications intended to be reproduced without modification. This is important when specific spot colors have been selected.

- `KcsContUnknown` describes color data content that is not known by the application. The CMM provides a general default for this case.

Note – ICC content hints are called *rendering hints*. Currently, the following rendering hints defined are:
`icPerceptual = KcsContImage`
`icRelativeColorimetric = KcsContColormetric`
`icSaturation = KcsContGraphics`
`icAbsoluteColorimetric` = <no equivalent>

If you have input color data that matches more than one of these content hints (for example, a complicated page layout), you can specify `KcsContUnknown` to produce adequate results. For best results, your application may have to divide color data into different parts (for example, separate graphics and images parts). After dividing, your application can process each part separately, applying the appropriate content hint to each part.

If you specify `KcsContAll` as an argument to `KcsConnectProfiles()`, the resultant profile has the full range of content hints available to it. If you do not, the resultant profile is restricted to the content hints supplied by the function.

CMMs can define additional custom content hints, such as the following examples:

- Indicate what kind of output is being produced, such as a photograph or a computer-generated graphic.
- Indicate that speed is more important than color image quality; therefore, compromised color is acceptable.

KcsMeasurementBase

```
typedef struct KcsMeasurementBase_s {
    unsigned long  countSupplied;
    KcsAttrSpace   inputSpace;
    KcsAttrSpace   outputSpace;
    unsigned long  numInComp;
    unsigned long  numOutComp;
    unsigned long  pad;
} KcsMeasurementBase;
```

`KcsMeasurementBase` defines a common subset of information in the `KcsCharacterizationData` and `KcsCalibrationData` structures. Nothing in `KcsMeasurementBase` is extendable.

The `countSupplied` field represents the number of allocated color patches, or samples in the measurement set.

The `inputSpace` and `outputSpace` fields represent the input and output color spaces, respectively, for the measurement set.

The `numInComp` and `numOutComp` fields represent the number of input components (such as 3 for RGB) and the number of output components, respectively.

KcsMeasurementSample

```
typedef struct KcsMeasurementSample_s {
    float          weight;
    float          standardDeviation;
    KcsColorSample sampleType;
    float          input[KcsMaxSamples];
    float          output[KcsMaxSamples];
} KcsMeasurementSample;
```

`KcsMeasurementSample` holds a single measurement. Both the `KcsCalibrationData` and the `KcsCharacterization` data structures contain extendable arrays of `KcsMeasurementSample` structures. Each measurement has an input, an output, a measurement weight, standard deviation and sample type. The input and output color spaces are specified by fields in the `KcsMeasurementBase` structure, which is part of both the `KcsCalibration` and `KcsCharacterization` structures.

The `weight` field should contain a value greater than 0.0 and less than or equal to 1.0. This is to provide information about the importance of this color measurement. The `KcsUpdateProfile()` function may or may not use this field when performing the steps needed to update the profile. Hence, it is to be considered a hint. The default setting should be the value 1.0.

The `standardDeviation` field is used to record this value when the sample is the result of statistical averaging of multiple measurements.

The `sampleType` field is to be used to indicate that a sample is from a black, white, neutral, chromatic, or fluorescent color. The default value is chromatic.

To calibrate or characterize device profiles, the default KCMS CMM needs color measurements that contain both input and output values. The `input` and `output` fields hold the input and output values of a color measurement. For RGB monitors, the input values are a series of RGB values and the output values are measured luminants of the RGB value.

`KcsMaxSamples` equals `4,` which allows up to four components of color to be stored in a measurement, for example, a CMYK color value. However, a three-component color value such as RGB or XYZ also can be stored. In such a case leave `input[3]` or `output[3]` undefined.

KcsOperationType

```
typedef unsigned long KcsOperationType;
```

`KcsOperationType` specifies the set of operations possible on a profile and the contents of the data on which the profile acts. It is an argument in these functions:

- `KcsConnectProfiles()`
- `KcsOptimizeProfile()`
- `KcsEvaluate()`

When used in `KcsConnectProfiles()` and `KcsOptimizeProfile()`, `KcsOperationType` limits the range of operations in a profile, thereby potentially speeding performance and reducing profile size. The operation hints and content hints are assigned positions in the load hints that let the application limit what resources are used from the initial loading of the profile.

When used in `KcsEvaluate()`, `KcsOperationType` indicates which kind of evaluation operation to perform. In this case, the operation type can specify only one operation; for example, you cannot evaluate in the forward and simulate directions at the same time.

To help you set the operation hints and content hints, the C API provides the following constants:

```
#define KcsOpForward          (0x00000001)
#define KcsOpReverse          (0x00000002)
#define KcsOpSimulate         (0x00000004)
#define KcsOpGamutTest        (0x00000008)
#define KcsOpAll              (0x000003FF)
#define KcsContUnknown        (0x00000000)
#define KcsContGraphics       (0x00010000)
#define KcsContImage          (0x00020000)
#define KcsContColorimetric   (0x00040000)
#define KcsContAll            (0x00FF0000)
```

KcsOptimizationType

```
typedef unsigned long KcsOptimizationType;
```

`KcsOptimizationType` is the data type of one of the arguments to the `KcsOptimizeProfile()` function.

`KcsOptimizationType` indicates the types of optimization that should be performed on a profile. It can have any of the following values, alone or in combination. Note that these are only hints.

```
#define KcsOptNone        (0)
#define KcsOptAccuracy    (1<<0)
#define KcsOptSpeed       (1<<1)
#define KcsOptSize        (1<<2)
```

- `KcsOptAccuracy`—profile produces more accurate output colors when it is input to the `KcsEvaluate()` function.
- `KcsOptSpeed`—profile runs faster when it is input to the `KcsEvaluate()` function.
- `KcsOptSize`—profile uses as little space as possible.

KcsPixelLayout

```
typedef struct KcsPixelLayout_s {
    unsigned long  numbOfComp;
    KcsComponent   component[KcsExtendablePixelLayout];
} KcsPixelLayout;
```

The `KcsPixelLayout` structure describes both the source data buffer (the layout of the data to be converted) and the destination data buffer (the receptacle of the converted data) used by `KcsEvaluate()`.

`KcsPixelLayout` describes a wide variety of pixel layouts in memory including:

- *Component-interleaved* data—components of a pixel (for example, the red, green, and blue components of an RGB image) are stored in consecutive memory addresses. (This is also called *pixel-interleaved* data.) See Figure 3-2 on page 52 for a detailed diagram of this pixel layout.
- *Row-interleaved* data—image data is stored by row and, within each row, by sub-rows for each component.

- *Planar* or *band-interleaved* data—image data is stored by component, allowing the components to be stored in independently contiguous memory areas.

`KcsPixelLayout` can also hold *palette color*, or a *colormap* by allowing the application to describe the palette instead of the data itself, as well as allowing the application to describe a single pixel.

If an application stores its image data in a form that is not representable using the `KcsPixelLayout` structure, the application must convert the data into one of the representable forms before calling the `KcsEvaluate` function.

The `numbOfComp` field specifies the number of components (channels). For example, you specify the value 3 for RGB data or 4 for CMYK data.

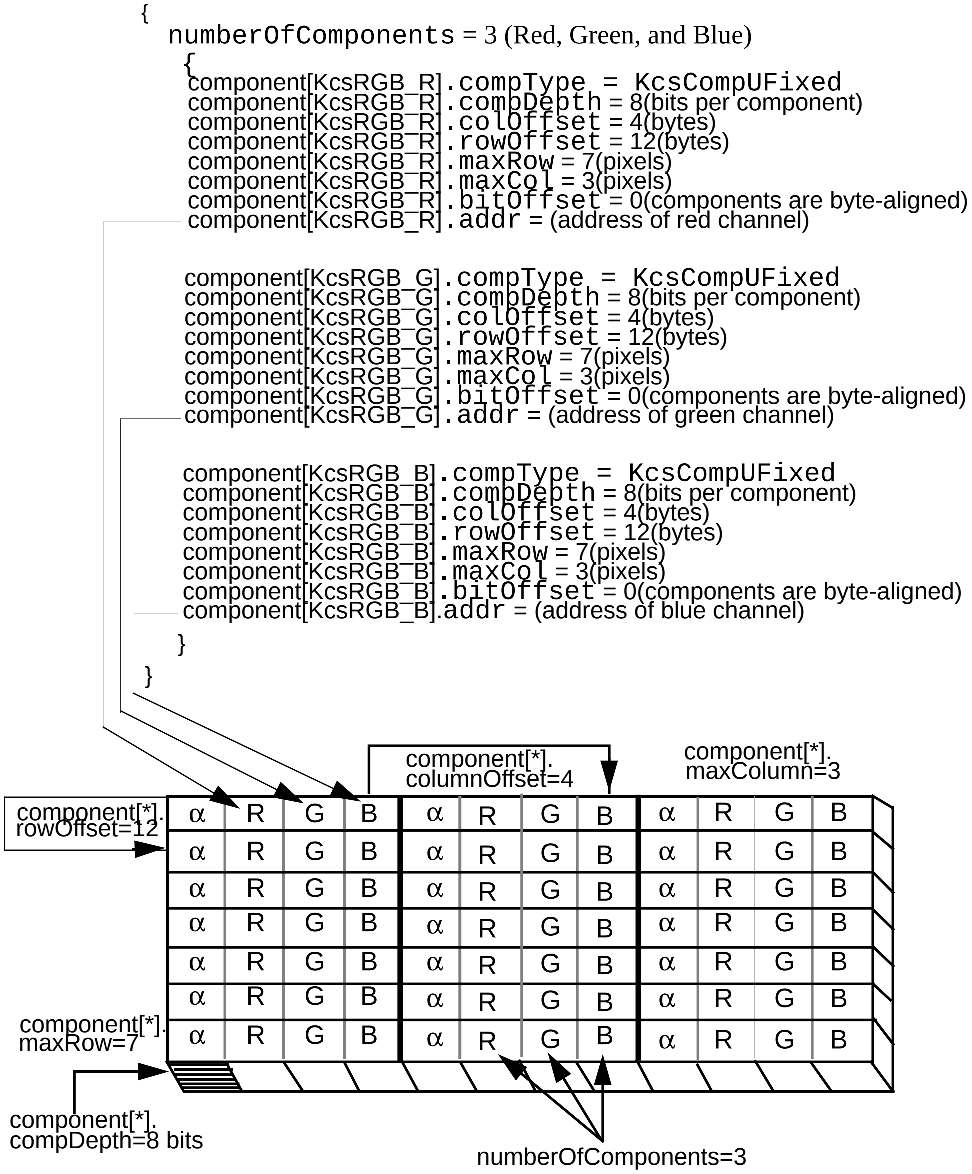
The `component` field is an array of base type `KcsComponent`. It holds the information needed to describe a component (see page 35 for more information). The `KcsExtendableArray` constant equals 4 by default. For ease of use, 4 was chosen because it can accommodate most applications, such as CMYK and RGB. It holds the upper limit. Having the open-ended array at the end of the structure allows you to allocate a larger structure and to extend it past 4, if needed.

Use the following definitions to index the component array:

RGB	#define KcsRGB_R	0
	#define KcsRGB_G	1
	#define KcsRGB_B	2
CMY[K]	#define KcsCMYK_C	0
	#define KcsCMYK_M	1
	#define KcsCMYK_Y	2
	#define KcsCMYK_K	3
YCC	#define KcsYCbC_Y	0
	#define KcsYCbC_Cb	1
	#define KcsYCbC_Cy	2
XYZ	#define KcsCIEXYZ_X	0
	#define KcsCIEXYZ_Y	1
	#define KcsCIEXYZ_Z	2
xyY	#define KcsCIExyY_x	1
	#define KcsCIExyY_y	2
	#define KcsCIExyY_Y	0
CIEuvL	#define KcsCIEuvL_u	1
	#define KcsCIEuvL_v	2
	#define KcsCIEuvL_L	0
CIEL*u*v	#define KcsCIELuv_L	0
	#define KcsCIELuv_u	1
	#define KcsCIELuv_v	2
CIEL*a*b*	#define KcsCIELab_L	0
	#define KcsCIELab_a	1
	#define KcsCIELab_b	2
HSV	#define KcsHSV_H	0
	#define KcsHSV_S	1
	#define KcsHSV_V	2
HLS	#define KcsHSV_H	0
	#define KcsHSV_L	1
	#define KcsHSV_S	2
GRAY	#define KcsGRAY_K	0

Two structures of type `KcsPixelLayout` are needed to describe the source data and destination data. Source and destination structures can point to the same data. If the CMM in use does not support this, or if there is some other mismatch between the CMM and the layout structures, `KcsEvaluate()` returns KCS_LAYOUT_UNSUPPORTED. For example, a CMM may not be able to support the way the source data and the destination data overlap in memory.

You can use a pixel layout structure to define any rectangular region of a larger image. Figure 3-2 illustrates the component-interleaved, 3-by-7 layout supported in the C API.



```
{
  numberOfComponents = 3 (Red, Green, and Blue)
   {
   component[KcsRGB_R].compType = KcsCompUFixed
   component[KcsRGB_R].compDepth = 8(bits per component)
   component[KcsRGB_R].colOffset = 4(bytes)
   component[KcsRGB_R].rowOffset = 12(bytes)
   component[KcsRGB_R].maxRow = 7(pixels)
   component[KcsRGB_R].maxCol = 3(pixels)
   component[KcsRGB_R].bitOffset = 0(components are byte-aligned)
   component[KcsRGB_R].addr = (address of red channel)

   component[KcsRGB_G].compType = KcsCompUFixed
   component[KcsRGB_G].compDepth = 8(bits per component)
   component[KcsRGB_G].colOffset = 4(bytes)
   component[KcsRGB_G].rowOffset = 12(bytes)
   component[KcsRGB_G].maxRow = 7(pixels)
   component[KcsRGB_G].maxCol = 3(pixels)
   component[KcsRGB_G].bitOffset = 0(components are byte-aligned)
   component[KcsRGB_G].addr = (address of green channel)

   component[KcsRGB_B].compType = KcsCompUFixed
   component[KcsRGB_B].compDepth = 8(bits per component)
   component[KcsRGB_B].colOffset = 4(bytes)
   component[KcsRGB_B].rowOffset = 12(bytes)
   component[KcsRGB_B].maxRow = 7(pixels)
   component[KcsRGB_B].maxCol = 3(pixels)
   component[KcsRGB_B].bitOffset = 0(components are byte-aligned)
   component[KcsRGB_B].addr = (address of blue channel)
   }
}
```

Figure 3-2 24-bit Color Component-Interleaved Data for RGB Pixel Image

KcsPixelLayoutSpeeds

```
typedef struct KcsPixelLayoutSpeeds_s {
    KcsPixelLayout supportedLayout;
    KcsEvalSpeed   speed;
}KcsPixelLayoutSpeeds;
```

`KcsPixelLayoutSpeeds`, used in the `KcsAttributeValue` structure, defines the relationship between a CMM’s support of a pixel layout and how efficiently it uses that layout. Some CMMs are optimized for certain layouts; this allows the application to maximize a CMM’s performance based on the information returned by `KcsPixelLayoutSpeeds`.

KcsProfileDesc

```
typedef struct KcsProfileDesc_s {
    KcsProfileType     type;
    union {
        struct file_f {
            long        offset;    /* Offset into the file */
            KcsFileId   openFileId;/* File descriptor */
        } file;
        struct memPtr_f {
            void        *memPtr;   /* Pointer to start of memory */
            long        offset;    /* Offset to the profile */
            long        size;      /* Size of the profile */
        } memPtr;
#ifdef KCS_ON_SOLARIS
        struct solarisFile_f {
            char        *fileName; /* Name of the file */
            char        *hostName; /* Host name */
            int         oflag;     /* How to open it, see open(2) */
            mode_t      mode;      /* This is a u_long, see open(2) */
        } solarisFile;
        struct xWindow_f {
            Display     *dpy;          /* Display pointer */
            int         screen;        /* Screen number */
            Visual      *visual;       /* Pointer to windows visual */
            long        reserved;      /* Reserved for KCMS internal use */
        } xwin;
#endif KCS_ON_SOLARIS
        long pad[4];/* Maximum size of union */
    } desc;
} KcsProfileDesc;
```

`KcsProfileDesc` is a data structure that describes a profile and the kind of mechanism in which to load and save that profile. The mechanism is platform independent. A profile can reside in the file system, on a remote network device, in a piece of hardware or its device driver, in a contiguous piece of memory, and so on. `KcsProfileDesc` is a union to minimize space and to allow for future flexibility. Thus, the actual definition can be augmented to provide additional locations where a profile may reside in the system.

The types of profiles supported by each `type` are summarized below. See “KcsProfileType” on page 56 for more information on these profiles.

`KcsFileProfile`

The calling application opens the file and passes the KCMS framework a `KcsFileId`, `openFileId`, and an `offset` from the start of the file to the start of the profile data. This profile type is most likely used for profiles embedded in other files, such as TIFF.

`KcsMemoryProfile`

The calling application has loaded the profile into program memory. The `offset` value determines where the profile data starts relative to `memPtr`. The `size` value is the profile's size in bytes.

`KcsSolarisProfile`

The calling application supplies the name of a file, `fileName`, and its location, `hostName`. The `KcsSolarisProfile` loadable module searches for the name supplied in `fileName`. It searches the following directories in the order listed:

1. The current directory
2. Directories listed by the `KCMS_PROFILES` environment variable, which is a colon-separated list of directories
3. `/etc/openwin/devdata/profiles`
4. `/usr/openwin/etc/devdata/profiles`

If `hostName` is non-NULL, the `KcsSolarisProfile` loadable module first checks if the name supplied is the name of the current machine. If it is not the the current machine's name, the `KcsSolarisProfile` loadable module opens a connection to the RPC daemon, `kcms_server(1)` and tries to locate the profile on a remote machine. The RPC daemon searches only in the last two directories for the profile (#3 and #4), and only reads remote profiles.

The application does not need to supply the full name of the file; the `KcsSolarisProfile` loadable module automatically adds the following suffixes.

`.mon`	Monitor
`.spc`	Color space
`.inp`	Input (scanner)
`.out`	Output (printer)

KcsWindowProfile

The calling application supplies X11 window system information and then the `KcsWindowProfile` loadable module matches a corresponding profile with the `Display*`, `screen` number and `Visual*`.

Remote display capabilities are handled using the RPC daemon `kcms_server(1)`. The location and name of the host is derived from the X11 display pointer. Remote profiles have read-only permissions.

KcsProfileId

```
typedef long KcsProfileId;
```

`KcsProfileId` is a data type used in all API functions, except `KcsSetCallback()`. A `KcsProfileId` variable identifies a particular loaded profile in the KCMS framework. It is an opaque data type; therefore, direct manipulation of it has undetermined results.

KcsProfileType

```
typedef enum {
    KcsFileProfile    = 0x46696C65,/* File */
    KcsMemoryProfile  = 0x4D426C00,/* MBl */
#ifdef KCS_ON_SOLARIS
    KcsWindowProfile  = 0x7877696E,/* xwin */
    KcsSolarisProfile = 0x736F6C66,/* solf */
#else
    KcsWindowProfile  = 0x57696E64, /* Wind */
#endif KCS_ON_SOLARIS
    KcsProfileTypeEnd = 0x7FFFFFFF,
    KcsProfileTypeMax = KcsForceAlign
} KcsProfileType;
```

Each `KcsProfileType` entry is a 4-byte hexadecimal value that is translated into a 4-byte ASCII string. This string is used as a key to determine which KCMS CMM module to use when loading or saving the profile into KCMS.

`KcsFileProfile` and `KcsMemoryProfile` are always included with KCMS; `KcsSolarisProfile` and `KcsWindowProfile` are dynamically loaded when needed.

See “KcsProfileDesc” on page 54 for details on using each type.

The type of color measurements depends on the specific device type. The default KCMS CMM supports scanner and monitor profile updates. For each of these devices, the color measurements are different. See Chapter 4, “Functions,” for a complete specification of the measurements passed to `KcsUpdateProfile` for each device type.

KcsSampleType

```
typedef unsigned long KcsSampleType;
```

`KcsSampleType` is the data type of a field in the `KcsComponent` structure. It is an enumerated constant with any of the values shown in Table 3-3. A variable of type `KcsSampleType` holds the data type of samples of each color channel.

The C API uses the `KcsSampleType` value with the `compDepth` field of `KcsComponent`. The `compDepth` field specifies the number of bits for each channel. For example, an RGB color space has three channels. If each represents its color in eight fixed-point bits, the value of `KcsSampleType` is `KcsCompUFixed`.

Table 3-3 `KcsSampleType` Constants

Enumerated Constant	Data Type of Channel
`#define KcsCompFixed 1`	Signed fixed-point sample.
`#define KcsCompUFixed 2`	Unsigned fixed-point sample.
`#define KcsCompFloat 3`	Floating point.
`#define KcsCompName 4`	A named color space component.

`KcsStatusId`

Every function in the C API returns a status code that indicates success or the reason for failure. A status code is an error or warning message. The `KcsStatusId` enumerated type is a list of all available status codes. KcsStatusId is defined in `kcsstats.h`.

See Chapter 6, “Warning and Error Messages,” for a complete list of all the enumerated constants and their meanings.

Functions 4

This chapter describes in detail each C API function you can use in applications. It describes each function's signature, use, arguments, and return values. For several functions, the chapter provides code examples. The functions are defined in the `kcs.h` header file and are presented in alphabetical order.

All constants, definitions, macros, and data types are defined in Chapter 3, "Data Structures," Chapter 5, "KCMS Profile Attributes and ICC Tags" and in the ICC specification. By default, the ICC specification is located online in the `SUNWsdk/kcms/doc` directory.

These API functions support error and warning messages returned by the operating system. See Chapter 6, "Warning and Error Messages," for all error and warning messages returned by these functions.

`KcsAvailable()`

```
KcsStatusId
KcsAvailable(long *response)
```

Purpose	The `KcsAvailable()` function determines if the KCMS framework has been installed on the system. This function is provided primarily for cross-platform compatibility.
Arguments	`response` is a pointer to a `long` for temporary use in the `KcsAvailable` function.
Returns	`KCS_SUCCESS` is always returned in the Solaris environment.

`KcsConnectProfiles()`

```
KcsStatusId
KcsConnectProfiles(KcsProfileId *resultProfileId,
       unsigned long profileCount,
       KcsProfileId *profileSequence,
       KcsOperationType operationLoadSet,
       unsigned long *failedProfileIndex)
```

Purpose Use `KcsConnectProfiles()` to combine several existing profiles into a new complete profile, or to restrict the functionality of a single existing profile to make it more efficient.

If `KcsConnectProfiles()` returns successfully, it generates a new profile from the sequence of existing profiles. The reference to this new profile is stored in the `resultProfileId` argument. With this reference, you can free the resources of the existing profiles in `profileSequence` if they are no longer required. Use `KcsFreeProfile()` to release the resources.

Note – If you have minimized a profile's load operation or state with `operationLoadSet` or with `KcsOptimizeProfile()` (page 81), only that load operation or state is saved with `KcsSaveProfile()`. Therefore, operations not included in the profile are not available the next time the profile is loaded.

Arguments `resultProfileId`

The profile returned if the function executes successfully.

`profileCount`

The number of profiles to be connected.

`profileSequence`

An array of the identifiers of the profiles to be connected.

`operationLoadSet`

One or more flags symbolizing the kind of information in the resultant profile. It also describes what, how, when, and where to load and unload the resulting `resultProfileId`. See "KcsLoadHints" on page 40 for more information.

`failedProfileIndex`

`KcsConnectProfiles()` returns an integer in `failedProfileIndex`. This value has meaning only when `KcsConnectProfiles()` returns a value other than `KCS_SUCCESS`. If the function fails, this index helps you identify which input profile caused the failure. If the index = `0`, the first profile in `profileSequence` failed; if it = `1`, the second profile in `profileSequence` failed, and so on. A common problem when making the resultant profile is that the profiles specified in `profileSequence` could not be connected. In this case, the index returns an integer symbolizing the latter profile in a failed connection pair. For example, if the first profile and second profile in the sequence were mismatched, the index contains 1 (for the second profile).

Returns

`KCS_SUCCESS`
`KCS_PROF_ID_BAD`
`KCS_MEM_ALLOC_ERROR`
`KCS_CONNECT_PRECISION_UNACCEPTABLE`
`KCS_MISMATCHED_COLORSPACES`
`KCS_CONNECT_OPT_FORCED_DATA_LOSS`

Example

Code Example 4-1 `KcsConnectProfiles()`

```
KcsProfileDesc scannerDesc, monitorDesc, completeDesc;
KcsProfileId   scannerProfile, monitorProfile;
KcsProfileId   profileSequence[2], completeProfile;
KcsStatusId    status;
KcsErrDesc     errDesc;
u_long         failedProfileNum;
KcsOperationType=(KcsOpForward+KcsContImage);
/*file names input a program arguments */

scannerDesc.type = KcsSolarisProfile;
```

Code Example 4-1 `KcsConnectProfiles()` *(Continued)*

```
scannerDesc.desc.solarisFile.fileName = argv[1];
scannerDesc.desc.solarisFile.hostName = NULL;
scannerDesc.desc.solarisFile.oflag = O_RDONLY;
scannerDesc.desc.solarisFile.mode = 0;

monitorDesc.type = KcsSolarisProfile;
monitorDesc.desc.solarisFile.fileName = argv[2];
monitorDesc.desc.solarisFile.hostName = NULL;
monitorDesc.desc.solarisFile.oflag = O_RDONLY;
monitorDesc.desc.solarisFile.mode = 0;

status = KcsLoadProfile(&scannerProfile, &scannerDesc, KcsLoadAllNow);

if(status != KCS_SUCCESS) {
    KcsGetLastError(&errDesc);
    printf(“Scanner LoadProfile error: %s\n”, errDesc.desc);
    exit(1);
}

status = KcsLoadProfile(&monitorProfile, &monitorDesc, KcsLoadAllNow);

if(status != KCS_SUCCESS) {
    KcsGetLastError(&errDesc);
    printf(“Monitor LoadProfile error: %s\n”, errDesc.desc);
    exit(1);
}

/* See if we can combine them */
profileSequence[0] = scannerProfile;
profileSequence[1] = monitorProfile;

status = KcsConnectProfiles(&completeProfile, 2, profileSequence, op,
        &failedProfileNum);

if(status != KCS_SUCCESS) {
    KcsGetLastError(&errDesc);
    printf(“ConnectProfile error: %s\n”, errDesc.desc);
    fprintf(stderr, “Failed in profile number %d\n”, failedProfileNum);
    exit(1);
}
```

KcsCreateProfile()

```
KcsStatusId
KcsCreateProfile(KcsProfileId *resultProfileId,
       KcsCreationDesc *desc)
```

Purpose Use `KcsCreateProfile()` to create an empty profile. The profile will contain neither attributes nor CMM-specific data.

Note – Currently, you cannot call `KcsGetAttribute()` for a list of the installed and available CMMs. The workaround is to load all available profiles and do a `KcsGetAttribute()` for each individual CMM type.

Arguments `resultProfileId`

The reference to the resultant profile, returned if the function executes successfully.

`desc`

This is a pointer to a `KcsCreationDesc` (see page 36) structure that describes the static store used to save the profile and an extendable union of profile information used to create the profile. The `id` member of the union describes which CMM and version to use, and the profile format and version to use.

If `desc` is `NULL` the default CMM and profile format are used.

Returns `KCS_SUCCESS`
`KCS_MEM_ALLOC_ERROR`

Example

Code Example 4-2 `KcsCreateProfile()`

```
KcsProfileDesc     desc;
KcsCreationDesc    c_desc;
KcsProfileId       profileid;
KcsStatusId        status;
KcsErrDesc         errDesc;
/* The filename is a command line argument */
/* Create a new profile with the default CMM */

desc.type = KcsSolarisProfile;
desc.desc.solarisFile.fileName = argv[1];
desc.desc.solarisFile.hostName = NULL;
desc.desc.solarisFile.oflag = O_RDWR|O_CREAT|O_TRUNC;
desc.desc.solarisFile.mode = 0666;
c_desc.profileDesc = &desc;
c_desc.desc.id.cmmId = 0;
c_desc.desc.id.cmmVersionId = 0;
c_desc.desc.id.profileId = 0;
c_desc.desc.id.profileVersionId = 0;
status = KcsCreateProfile(&profileid, &c_desc);
if(status != KCS_SUCCESS) {
    KcsGetLastError(&errDesc);
    printf(“CreateProfile error: %s\n”, errDesc.desc);
}
```

Note – Other required fields in the profile must be set with `KcsSetAttribute()`.

KcsEvaluate()

```
KcsStatusId
KcsEvaluate(
        KcsProfileId profile,
        KcsOperationType operation,
        KcsPixelLayout *srcData,
        KcsPixelLayout *destData)
```

Purpose Use `KcsEvaluate()` to apply a color profile to input color data to produce color-corrected output data.

See “KcsPixelLayout” on page 49 for more information about using pixel layouts in this context.

Arguments `profile`

The ID of the profile to be applied to the input data. If the operation specified when the profile was created in `KcsConnectProfiles()` does not match the operation specified in `KcsEvaluate()`, the `statusId` `KCS_EVAL_ONLY_ONE_OP_ALLOWED` is returned. If, for example, you wanted to evaluate forward (specified `KcsOpForward` in `KcsEvaluate()`) with a profile you created with `KcsConnectProfiles()` to simulate (used `KcsOpSimulate` in `KcsConnectProfiles()`), the above `statusId` would be returned.

`operation`

The kind of data to be operated on, and the kind of profile evaluation to be performed on, the data. (See “Operation Hint Constants” on page 43 and “Content Hint Constants” on page 44 more information.) Note that only one bit can be set for `KcsEvaluate()`.

`srcData`

The description of the source color data to be transformed by the profile.

`destData`

The description of the area (the destination) to which the transformed data is written.

Returns

```
KCS_SUCCESS
KCS_OPERATION_CANCELLED
KCS_PROF_ID_BAD
KCS_MEM_ALLOC_ERROR
KCS_EVAL_ONLY_ONE_OP_ALLOWED
KCS_EVAL_TOO_MANY_CHANNELS
KCS_EVAL_BUFFER_OVERFLOW
KCS_LAYOUT_INVALID
KCS_LAYOUT_UNSUPPORTED
KCS_LAYOUT_MISMATCH
```

Example

Code Example 4-3 `KcsEvaluate()`

```
int                 op;
KcsPixelLayout      pixelLayoutIn, pixelLayoutOut;
KcsProfileId        scannerProfile, monitorProfile;
KcsProfileId        profileSequence[2], completeProfile;

/* Load and connect profiles. */
/* Load input and output pixel layout structures with appropriate data. */

status = KcsEvaluate(completeProfile, op, &pixelLayoutIn,
            &pixelLayoutOut);
```

KcsFreeProfile()

```
KcsStatusId
KcsFreeProfile(
        KcsProfileId    profile)
```

Purpose Use `KcsFreeProfile()` to release all resources a loaded profile is using. A loaded profile uses memory and additional types of resources.

The KCMS framework does not automatically save profile changes when your application terminates. To save profile changes, you must call `KcsSaveProfile()`.

Note – If the application passes a `KcsFileProfile` type of `KcsProfileDesc` as an argument, `KcsFreeProfile()` does not close the `KcsFileId` contained in the file entry of the `KcsProfileDesc` union.

Arguments `profile`

The identifier of a loaded profile.

Returns `KCS_SUCCESS`
`KCS_PROF_ID_BAD`

Example

Code Example 4-4 `KcsFreeProfile()`

```
KcsProfileId   profile;

/* Complete all processing. */

KcsFreeProfile(profile);
```

KcsGetAttribute()

```
KcsStatusId
KcsGetAttribute(KcsProfileId profile, KcsAttributeName name,
       KcsAttributeValue *value)
```

Purpose Use `KcsGetAttribute()` to find the value of a particular attribute of the given profile. (See Chapter 5, “KCMS Profile Attributes and ICC Tags” for more information on attributes.)

Arguments `profile`

The identifier of the loaded profile.

`name`

ICC tag name.

`value`

A pointer to the structure to hold the value of the profile’s attribute. You need to set the `countSupplied` field in the `value` argument. If you do not set it, you may see the warning `KCS_ATTR_LARGE_CT_SUPPLIED` or the error `KCS_ATTR_CT_ZERO_OR_NEG` returned.

Returns `KCS_SUCCESS`
`KCS_PROF_ID_BAD`
`KCS_ATTR_NAME_OUT_OF_RANGE`
`KCS_ATTR_CT_ZERO_OR_NEG`
`KCS_ATTR_LARGE_CT_SUPPLIED` (warning)

Example

Code Example 4-5 `KcsGetAttribute()`

```
#include “./kcms_utils.h”

KcsProfileId       profileid;
KcsAttributeValue  *attrValue;
int                size;
void print_header(icHeader *hdr);

size = sizeof(KcsAttributeBase) + sizeof(icHeader);
```

Code Example 4-5 `KcsGetAttribute()` *(Continued)*

```
attrValue = (KcsAttributeValue *)malloc(size);

/* Get the header */
attrValue->base.type = icSigHeaderType;
attrValue->base.sizeOfType = sizeof(icHeader);
attrValue->base.countSupplied = 1;
KcsGetAttribute(profileid, icSigHeaderTag, attrValue);
...
print_header(&attrValue->val.icHeader);
...

void
print_header(icHeader *hdr)
{
    char    charstring[5];

    printf(“Size in bytes = %d\n”, hdr->size);
    printf(“CMM Id = 0x%x\n”, hdr->cmmId);
    printf(“Major version number = 0x%x\n”, hdr->version>>24);
    printf(“Minor version number = 0x%x\n”, (hdr->version&0x00FF0000)>>16);

    switch(hdr->deviceClass) {
    case icSigInputClass :
        printf(“deviceClass = input\n”);
        break;
    case icSigDisplayClass :
        printf(“deviceClass = display\n”);
        break;
    case icSigOutputClass :
        printf(“deviceClass = output\n”);
        break;
    case icSigLinkClass :
        printf(“deviceClass = link\n”);
        break;
    case icSigAbstractClass :
        printf(“deviceClass = abstract\n”);
        break;
    case icSigColorSpaceClass :
        printf(“deviceClass = colorspace\n”);
        break;
    default :
        printf(“Unknown\n”);
        break;
```

Code Example 4-5 `KcsGetAttribute()` *(Continued)*

```
    }

    memset(charstring, 0 ,5);
    memcpy(charstring, &hdr->colorSpace, 4);
    printf(“colorspace = %s\n”, charstring);

    memset(charstring, 0 ,5);
    memcpy(charstring, &hdr->pcs, 4);
    printf(“profile connection space = %s\n”, charstring);

    printf(“date = %d/%d/%d,  “, hdr->date.day,hdr->date.month,
           hdr->date.year);
    printf(“time = %d:%d:%d\n”, hdr->date.hours,hdr->date.minutes,
           hdr->date.seconds);

    memset(charstring, 0 ,5);
    memcpy(charstring, &hdr->magic, 4);
    printf(“magic number = %s\n”, charstring);

    switch(hdr->platform) {
    case icSigMacintosh :
       printf(“platform = Macintosh\n”);
       break;
    case icSigMicrosoft :
       printf(“platform = Microsoft\n”);
       break;
    case icSigSolaris :
       printf(“platform = Solaris\n”);
       break;
    case icSigSGI :
       printf(“platform = SGI\n”);
       break;
    case icSigTaligent :
       printf(“platform = Taligent\n”);
       break;
    default :
       printf(“Unknown\n”);
       break;
    }

    if(hdr->flags && icEmbeddedProfileTrue)
       printf(“Embedded profile.\n”);
    else
       printf(“Non-embedded profile\n”);
```

Code Example 4-5 `KcsGetAttribute()` *(Continued)*

```
    if(hdr->flags && icUseWithEmbeddedDataOnly)
       printf(“If this profile is embedded, it is not allowed to strip
              it out and use it independently.\n”);
    else
       printf(“OK to strip embedded profile out and use
              independently\n”);

    memset(charstring, 0 ,5);
    memcpy(charstring, &hdr->manufacturer, 4);
    printf(“manufacturer = %s\n”, charstring);

    printf(“model number = %d\n”, hdr->model);

    printf(“device attributes = %d%d\n”, hdr->attributes[0],
           hdr->attributes[1]);

    switch (hdr->renderingIntent) {
    case 0 :
       printf(“rendering intent = Perceptual\n”);
       break;
    case 1 :
       printf(“rendering intent = Relative Colorimetric\n”);
       break;
    case 2 :
       printf(“rendering intent = Saturation\n”);
       break;
    case 3 :
       printf(“rendering intent = Absolute Colorimetric\n”);
       break;
    default :
       printf(“Unknown\n”);
       break;
    }

    printf(“Illuminat X=%f Y=%f X=%f\n”,
       icfixed2double(hdr->illuminant.X, icSigS15Fixed16ArrayType),
       icfixed2double(hdr->illuminant.Y, icSigS15Fixed16ArrayType),
       icfixed2double(hdr->illuminant.Z, icSigS15Fixed16ArrayType));
}
```

KcsGetLastError()

```
KcsStatusId
KcsGetLastError (KcsErrDesc *errDesc)
```

Purpose Use `KcsGetLastError()` to find information about the most recent error.

Arguments `errDesc`

A pointer to the structure holding information about the last error.

If an operating-system-defined error occurs, the `sysErrNo` field is set.

The `desc` field contains the description of the particular `statId`. This is either a string in Table 6-1 or Table 6-2 on page 136, or the literal string “Internal Color Processor Error” or “No description for this status id number”. See Chapter 6, “Warning and Error Messages” for information on using `KcsGetLastError()` to localize `KcsStatusId`.

Returns `KCS_SUCCESS`

Example

Code Example 4-6 `KcsGetLastError()`

```
KcsErrDesc errDesc;

status = KcsLoadProfile(&profile, &profileDesc,
        KcsLoadAttributesNow);
if (status != KCS_SUCCESS) {
    status = KcsGetLastError(&errDesc);
    fprintf(stderr,”%s KcsLoadProfile failed error = %s\n”,
            errDesc.desc);
    exit(1);
}
```

`KcsLoadProfile()`

```
KcsStatusId
KcsLoadProfile(KcsProfileId *profile,
       KcsProfileDesc *desc, KcsLoadHints loadHints)
```

Purpose Use `KcsLoadProfile()` to load a profile and all of its resources into the system.

The function uses `desc` to determine where to get the data to generate the profile's resources in the system. (See page 54 for an in-depth description of `KcsProfileDesc`.) It uses `profile` to return a reference to the loaded profile; this reference is needed by other API functions.

You can determine the length of the data read from the file by calling `KcsGetAttribute()` and supplying the `icHeader` attribute. The value of `size` in `icHeader` is the size of the profile. (For the format of the icHeader structure, see "icHeader" on page 132.)

With the `loadHints` argument, `KcsLoadProfile()` allows the application to suggest how the KCMS framework manages the memory and other resources associated with a loaded profile. Although this is a flexible mechanism, these caveats apply:

- The load hints are merely hints, which means the KCMS framework can ignore them. However, because the functionalities of various CMMs loaded by the KCMS framework cannot always be determined, your application should supply the load hints anyway. Furthermore, even if a CMM loaded by the KCMS framework does not support a particular load hint in its current release, it may support it in future releases.
- If the application supplies a hint that indicates that the profile is to be loaded at a time other than now, it must keep the described mechanism open to allow for data access at a future and somewhat arbitrary time. For example, if the application specifies `KcsLoadWhenNecessary` and the `desc` argument describes a file, and the application uses a `KcsFileId`, it

cannot close the file until it first frees the profile. This allows the KCMS framework to read any necessary data to load the profile at any time.

After you are finished with the profile, call `KcsFreeProfile()` to release the resources allocated by this profile.

Note – If you use the `KcsFileId` entry in the file part of the `KcsProfileDesc` union, `KcsFileId` marks the *current position* within an open file. After a call to `KcsLoadProfile()`, the current position is undefined. The application must reset the pointer before doing any other I/O.

Arguments `profile`

The identifier of the profile returned after the profile is loaded into memory. This value serves as an argument to all other functions, such as `KcsEvaluate()`.

`desc`

The location of the profile's static storage, needed to obtain the data required to generate the profile's resources. It is specified as a union of independent static storage mechanisms. The `KcsProfileDesc` structure has a field that identifies which storage mechanism to use.

`loadHints`

The set of bits describing what, how, when and where to load and unload `profile`. See page 40 for more information on the `KcsLoadHints` data type.

Returns

`KCS_SUCCESS`
`KCS_MEM_ALLOC_ERROR`
`KCS_IO_READ_ERR`
`KCS_IO_SEEK_ERR`
`KCS_SOLARIS_FILE_NOT_OPENED`
`KCS_SOLARIS_FILE_RO`
`KCS_SOLARIS_FILE_LOCKED`
`KCS_SOLARIS_FILE_NAME_NULL`
`KCS_X11_DATA_NULL`
`KCS_X11_PROFILE_NOT_LOADED`
`KCS_X11_PROFILE_RO`

Example

Code Example 4-7 `KcsLoadProfile()`

```
#pragma ident “@(#)kcstest.c1.8  11/28/94”
/*
 * Copyright (c) 1994, by Sun Microsystems, Inc.

    KcsFileIdscannerFd, monitorFd, completeFd;
    KcsProfileDescscannerDesc, monitorDesc, completeDesc;
    KcsProfileId scannerProfile, monitorProfile;
    KcsProfileIdprofileSequence[2], completeProfile;
    KcsStatusIdstatus;
    KcsAttributeValueattrValue;
    KcsAttributeNamei;
    KcsOperationTypeop = (KcsOpForward+KcsContImage);
    u_longfailedProfileNum;
    extern voidkcs_timer(int);

    if (argc > 4) {
    fprintf(stderr,”Usage : kcstest profile_1 profile_2 [save_profile]\n”);
    exit(1);
    }

#ifdef FILE_DESC
    /* Open up the files from disk */
    scannerDesc.type = KcsFileProfile;
    scannerFd = open(argv[1], O_RDONLY);
    if (scannerFd == -1) {
    perror(“Failed to open scanner profile”);
    exit(1);
    }
    scannerDesc.desc.file.openFileId = scannerFd;
```

Code Example 4-7 `KcsLoadProfile()`

```
    scannerDesc.desc.file.offset = 0;

    monitorDesc.type = KcsFileProfile;
    monitorFd = open(argv[2], O_RDONLY);
    if (monitorFd == -1) {
    perror(“Failed to open monitor profile”);
    exit(1);
    }
    monitorDesc.desc.file.openFileId = monitorFd;
    monitorDesc.desc.file.offset = 0;
#endif

#ifdef FILE_NAME
    scannerDesc.type = KcsSolarisProfile;
    scannerDesc.desc.solarisFile.fileName = argv[1];
    scannerDesc.desc.solarisFile.hostName = NULL;
    scannerDesc.desc.solarisFile.oflag = O_RDONLY;
    scannerDesc.desc.solarisFile.mode = 0;

    monitorDesc.type = KcsSolarisProfile;
    monitorDesc.desc.solarisFile.fileName = argv[2];
    monitorDesc.desc.solarisFile.hostName = NULL;
    monitorDesc.desc.solarisFile.oflag = O_RDONLY;
    monitorDesc.desc.solarisFile.mode = 0;
#endif

    /* Load the profiles */
    printf(“Load scanner profile\n”);
    kcs_timer(START);
    status = KcsLoadProfile(&scannerProfile, &scannerDesc, KcsLoadAllNow);
    kcs_timer(STOP);
    if (status != KCS_SUCCESS) {
    fprintf(stderr,”Scanner KcsLoadProfile failed error = 0x%x\n”, status);
#ifdef FILE_DESC
    close(scannerFd);
    close(monitorFd);
#endif
    exit(1);
    }

    printf(“Load monitor profile\n”);
    kcs_timer(START);
    status = KcsLoadProfile(&monitorProfile, &monitorDesc, KcsLoadAllNow);
    kcs_timer(STOP);
```

Code Example 4-7 `KcsLoadProfile()`

```
    if (status != KCS_SUCCESS) {
    fprintf(stderr,”MonitoKcsLoadProfile failed error = 0x%x\n”, status);
#ifdef FILE_DESC
    close(scannerFd);
    close(monitorFd);
#endif
    exit(1);
    }
```

KcsModifyLoadHints()

```
KcsStatusId
KcsModifyLoadHints(KcsProfileId profile,
       KcsLoadHints newHints)
```

Purpose `KcsModifyLoadHints()` applies a new set of load hints to a profile already loaded. If, for example, you no longer need to simulate a profile and available memory is limited, you can use this function to unload the simulation portion of the profile immediately, making more memory available for the application.

Note – Remember that the load hints are just that—hints to the KCMS framework. Although the KCMS framework tries to accomplish what is specified, and typically does, it cannot guarantee everything exactly as hinted.

Arguments `profile`

The identifier of a loaded profile.

`newHints`

The set of bits describing what, how, when, and where to load and unload `profile`. See “KcsLoadHints” on page 40 for more information.

Returns

`KCS_SUCCESS`
`KCS_PROF_ID_BAD`
`KCS_MEM_ALLOC_ERROR`

Example

Code Example 4-8 `KcsModifyLoadHints()`

```
KcsProfileId   profileid;
KcsErrDesc     errDesc;
KcsProfileDesc profileDesc;
KcsProfileId   profile;
KcsStatusId    status;
KcsLoadHints   newhints;

/* profile name is a command line argument */
```

Code Example 4-8 `KcsModifyLoadHints()` *(Continued)*

```
profileDesc.type = KcsSolarisProfile;
profileDesc.desc.solarisFile.fileName = argv[1];
profileDesc.desc.solarisFile.hostName = NULL;
profileDesc.desc.solarisFile.mode = 0;
profileDesc.desc.solarisFile.oflag = NULL;

status = KcsLoadProfile(&profile, &profileDesc, KcsLoadAttributesNow);
if (status != KCS_SUCCESS) {
    status = KcsGetLastError(&errDesc);
    fprintf(stderr,”%s KcsLoadProfile failed error = %s\n”,
        argv[optind], errDesc.desc);
    exit(1);
}

/* suppose it was determined that this is the profile we want to *
 * use for evaluating data. We want to load it all in now. */

newhints = KcsLoadAllNow;
status = KcsModifyLoadHints(profile, newhints);
if (status != KCS_SUCCESS) {
    status = KcsGetLastError(&errDesc);
    fprintf(stderr,” ModifyHints failed error = %s\n”, errDesc.desc);
    exit(1);
}
```

KcsOptimizeProfile()

```
KcsStatusId
KcsOptimizeProfile(KcsProfileId profile,
       KcsOptimizationType optimizationType,
       KcsLoadHints operationLoadSet)
```

Purpose Use `KcsOptimizeProfile()` to optimize the profile by:

- Reducing the profile’s size
- Increasing the profile’s speed
- Increasing the profile’s accuracy

Optimization is CMM dependent. The CMM always interprets the load hints in terms of the particular situation.

Note – If you have minimized a profile’s load operation or state with `operationLoadSet` or with `KcsOptimizeProfile()`, only that load operation or state is saved with `KcsSaveProfile()`. Therefore, operations not included in the profile are not available the next time the profile is loaded.

Arguments `profile`

The identifier of the profile.

`optimizationType`

The kinds of optimization (size, speed, and accuracy) you want to perform on the profile. (See “KcsOptimizationType” on page 48 for more information.) When a combination of values is specified, it is up to the CMM to determine which value is more important.

`operationLoadSet`

One or more flags symbolizing the kind of information wanted in `profile`. It also describes what, how, when, and where to load and unload `profile`. See “KcsLoadHints” on page 40 for more information.

Returns `KCS_SUCCESS`
`KCS_OPERATION_CANCELLED`
`KCS_MEM_ALLOC_ERROR`
`KCS_PROF_ID_BAD`

Example

Code Example 4-9 `KcsOptimizeProfile()`

```
KcsProfileId   monitorProfile, scannerProfile, completeProfile;
KcsStatusId    status;
KcsErrDesc     errDesc;

/* The monitor profile and scanner profile have been loaded and connected *
 * to become a complete profile, now optimize. */

status = KcsOptimizeProfile(completeProfile, KcsOptSpeed, KcsLoadAllNow);
if (status != KCS_SUCCESS) {
    status = KcsGetLastError(&errDesc);
    fprintf(stderr,”KcsOptimizeProfile failed error = %s\n”, errDesc.desc);
    KcsFreeProfile(monitorProfile);
    KcsFreeProfile(scannerProfile);
    return(-1);
}
```

`KcsSaveProfile()`

```
KcsStatusId
KcsSaveProfile (KcsProfileId profile, KcsProfileDesc *desc)
```

Purpose

Use `KcsSaveProfile()` to save a loaded profile, and any changes to its attributes or profile data, to the mechanism described by `desc`.

If supported by the mechanism, a profile's state can be saved at an offset. For example, if the mechanism indicates a file, the following two situations are applicable:

- Create a file containing only one profile. In this case most typically the offset would be 0.
- Create a file containing one profile plus some application data (like a TIFF file). You must ensure that the profile fits into the file format and does not overwrite data nor is itself overwritten. You can determine the length of the data read from the file by calling `KcsGetAttribute()` and supplying the `icHeader` attribute. The value of `size` in `icHeader` is the size of the profile.

`KcsSaveProfile()` writes information, but does not free the profile. Even after saving the profile, the application can continue to use it. In fact, the application must call `KcsFreeProfile()` to free all resources associated with the profile.

Arguments

`profile`

The identifier of the loaded profile. Typically, your application obtains this value when it calls `KcsLoadProfile()` or `KcsConnectProfiles()`.

`desc`

The location of the profile's static storage mechanism, needed to obtain the data required to generate the profile's resources. It is specified as a union of independent static storage mechanisms. This argument has a field that identifies which storage mechanism to use. If this field is `NULL`, the profile is saved through the same mechanism from which it was loaded. (See "KcsProfileId" on page 56 for more information.)

Returns

`KCS_SUCCESS`
`KCS_IO_WRITE_ERR`
`KCS_IO_READ_ERR`
`KCS_IO_SEEK_ERR`
`KCS_SOLARIS_FILE_NOT_OPENED`
`KCS_SOLARIS_FILE_RO`
`KCS_SOLARIS_FILE_LOCKED`
`KCS_SOLARIS_FILE_NAME_NULL`
`KCS_X11_DATA_NULL`
`KCS_X11_PROFILE_NOT_LOADED`
`KCS_X11_PROFILE_RO`

Example

Code Example 4-10 `KcsSaveProfile()`

```
KcsProfileDesc desc;
KcsProfileId   profileid;
KcsStatusId    status;
KcsErrDesc     errDesc;

/*see example kcms_update.c for a full example code */

desc.type = KcsSolarisProfile;
desc.desc.solarisFile.fileName = argv[1];
desc.desc.solarisFile.hostName = NULL;
desc.desc.solarisFile.mode = 0;
desc.desc.solarisFile.oflag = O_RDWR
status = KcsSaveProfile(profileid, &desc);
if(status != KCS_SUCCESS) {
    status = KcsGetLastError(&errDesc);
    fprintf(stderr,"KcsSaveProfile failed error = %s\n", errDesc.desc);
}
KcsFreeProfile(profileid);
```

Note – If you are saving a *new* profile, use the following assignments instead of the assignments in Code Example 4-10:

```
desc.desc.solarisFile.mode = 0666;
desc.desc.solarisFile.oflag = O_RDWR | O_CREAT | O_TRUNC;
```

KcsSetAttribute()

```
KcsStatusId
KcsSetAttribute(KcsProfileId profile,
       KcsAttributeName name,
       KcsAttributeValue *value)
```

Purpose Use `KcsSetAttribute()` to create, to modify, or to delete a specific attribute in a profile. See Chapter 5, “KCMS Profile Attributes and ICC Tags” for details on attributes.

Arguments `profile`

The identifier to the profile.

`name`

The attribute to be created, modified, or deleted. If this attribute is already used in the profile, this function overwrites its value. If this attribute does not already exist, the function creates it. See the attribute tables in Chapter 5, “KCMS Profile Attributes and ICC Tags” for more information.

`value`

A pointer to the value for the attribute. If the attribute already exists, then `value` becomes the attribute’s new value. If the attribute does not already exist, this function creates it and sets its original value to `value`. To delete an existing attribute, set `value` to `NULL`.

Note – For this function to execute correctly, you must check what needs to be set in the `KcsAttributeBase` structure (part of the `KcsAttributeValue` structure). A valid type and number of tokens found in the attribute must be set.

Returns `KCS_SUCCESS`
`KCS_MEM_ALLOC_ERROR`
`KCS_PROF_ID_BAD`
`KCS_ATTR_NAME_OUT_OF_RANGE`

`KCS_ATTR_TYPE_UNKNOWN`
`KCS_ATTR_NEG_CT_SUPPLIED`
`KCS_ATTR_LARGE_CT_SUPPLIED`

Example

Code Example 4-11 `KcsSetAttribute()`

```
#include “kcms_utils.h”
#define SAMPLE_WORDS “A profile created using kcms_create”

KcsProfileId      profileid;
KcsStatusId       status;
KcsAttributeValue attrValue;
KcsAttributeValue *attrValue2;
KcsAttributeValue *attrValuePtr;
KcsErrDesc        errDesc;
int               sizemeas, size, nvalues, i, j;
time_t            clocktime;
struct tm         *datetime;
size_t            rc;
char              *description;
char              attr[256];
double            test_double[3];

/* Fill out the measurement structures - The illuminant must be D50 */
test_double[0] = 0.9642;
test_double[1] = 1.0;
test_double[2] = 0.8249;

/*  open or create a profile, then set some attributes */
if ((description = (char *)malloc(strlen(SAMPLE_WORDS) + 1)) == NULL) {
    perror(“malloc failed : “);
    KcsFreeProfile(profileid);
    exit(1);
}
memset(description, 0, strlen(SAMPLE_WORDS) + 1);
strcpy(description, SAMPLE_WORDS);
/* the function used below can be found in kcms_utils.c in appendix */
if ((attrValue2 = string2icTextAttrValue(description)) == NULL) {
    fprintf(stderr, “conversion to AttrValue failed \n”);
    KcsFreeProfile(profileid);
    exit(1);
}
if (KcsSetAttribute(profileid, icSigProfileDescriptionTag, attrValue2)
    != KCS_SUCCESS) {
```

Code Example 4-11 `KcsSetAttribute()` *(Continued)*

```
    KcsGetLastError(&errDesc);
    printf(“Set Attribute error: %s\n”, errDesc.desc);
    exit(1);
}
free(attrValue2);
free(description);
size = sizeof(KcsAttributeBase) + sizeof(icHeader);
attrValuePtr = (KcsAttributeValue *)malloc(size);

/* Build the header */
attrValuePtr->base.type = icSigHeaderType;
attrValuePtr->base.sizeOfType = sizeof(icHeader);
attrValuePtr->base.countSupplied = 1;
KcsGetAttribute(profileid, icSigHeaderTag, attrValuePtr);
attrValuePtr->val.icHeader.size = 0;

/* The following three values do not have to be set if you do a
 * GetAttribute on the header, since the Create should set them for you.
 * If you do not do a GetAttribute of the header, you must set these:
 *   attrValuePtr->val.icHeader.cmmId = 0x4b434d53;
 *   attrValuePtr->val.icHeader.version =icVersionNumber;
 *   attrValuePtr->val.icHeader.magic = icMagicNumber;
*/
attrValuePtr->val.icHeader.deviceClass = icSigDisplayClass;
attrValuePtr->val.icHeader.colorSpace = icSigRgbData;
attrValuePtr->val.icHeader.pcs = icSigXYZData;

/* Get the time from the system */
clocktime = time(NULL);
datetime = localtime(&clocktime);

attrValuePtr->val.icHeader.date.seconds =
        (icUInt16Number)datetime->tm_sec;
attrValuePtr->val.icHeader.date.minutes =
        (icUInt16Number)datetime->tm_min;
attrValuePtr->val.icHeader.date.hours =
        (icUInt16Number)datetime->tm_hour;
attrValuePtr->val.icHeader.date.day =
        (icUInt16Number)datetime->tm_mday;
attrValuePtr->val.icHeader.date.month =
        (icUInt16Number)datetime->tm_mon + 1;
attrValuePtr->val.icHeader.date.year =
        (icUInt16Number)datetime->tm_year;
attrValuePtr->val.icHeader.platform = icSigSolaris;
```

Code Example 4-11 `KcsSetAttribute()` *(Continued)*

```
attrValuePtr->val.icHeader.flags =
        icEmbeddedProfileFalse || icUseAnywhere;
strcpy(description,”SUNW “);
memcpy(&attrValuePtr->val.icHeader.manufacturer, description, 4);
attrValuePtr->val.icHeader.model = 0;
attrValuePtr->val.icHeader.attributes[0] = 0;
attrValuePtr->val.icHeader.attributes[1] = 0;
attrValuePtr->val.icHeader.renderingIntent = icPerceptual;
attrValuePtr->val.icHeader.illuminant.X =
        double2icfixed(test_double[0], icSigS15Fixed16ArrayType);
attrValuePtr->val.icHeader.illuminant.Y =
        double2icfixed(test_double[1], icSigS15Fixed16ArrayType);
attrValuePtr->val.icHeader.illuminant.Z =
        double2icfixed(test_double[2], icSigS15Fixed16ArrayType);
rc = KcsSetAttribute(profileid, icSigHeaderTag, attrValuePtr);
if(rc != KCS_SUCCESS) {
    rc =KcsGetLastError(&errDesc);
    fprintf(stderr, “unable to set header: %s\n”, errDesc.desc);
    KcsFreeProfile(profileid);
    return(-1);
}
free(attrValuePtr);

/* set white point and colorants with dummy values to show it works*/
attrValue.base.countSupplied = 1;
attrValue.base.type = icSigXYZType;
attrValue.base.sizeOfType = sizeof(icXYZNumber);
attrValue.val.icXYZ.data[0].X = double2icfixed(test_double[0],
        icSigS15Fixed16ArrayType);
attrValue.val.icXYZ.data[0].Y = double2icfixed(test_double[1],
        icSigS15Fixed16ArrayType);
attrValue.val.icXYZ.data[0].Z = double2icfixed(test_double[2],
        icSigS15Fixed16ArrayType);
rc = KcsSetAttribute(profileid, icSigMediaWhitePointTag, &attrValue);
if(rc != KCS_SUCCESS) {
    KcsGetLastError(&errDesc);
    fprintf(stderr, “unable to set whitepoint: %s\n”, errDesc.desc);
    KcsFreeProfile(profileid);
    return(-1);
}
test_double[0] = 0.572586;
test_double[1] = 0.337198;
test_double[2] = 0.026291;
```

Code Example 4-11 `KcsSetAttribute()` *(Continued)*

```
attrValue.val.icXYZ.data[0].X = double2icfixed(test_double[0],
        icSigS15Fixed16ArrayType);
attrValue.val.icXYZ.data[0].Y =double2icfixed(test_double[1],
        icSigS15Fixed16ArrayType);
attrValue.val.icXYZ.data[0].Z =double2icfixed(test_double[2],
        icSigS15Fixed16ArrayType);
rc = KcsSetAttribute(profileid, icSigRedColorantTag, &attrValue);
if(rc != KCS_SUCCESS) {
    KcsGetLastError(&errDesc);
    fprintf(stderr, “unable to set red primaries: %s\n”, errDesc.desc);
    KcsFreeProfile(profileid);
    return(-1);
}
```

KcsSetCallback()

```
KcsStatusId
KcsSetCallback (KcsFunction function,
       KcsCallbackFunction callback, void *userDefinedData)
```

Purpose

Use `KcsSetCallback()` to associate a callback function with any set of API functions that support callbacks. Those functions are listed in `KcsFunction` (see page 38). If `KcsSetCallback()` is not called for particular values of `KcsFunction`, no callback is issued.

This function allocates resources. To release those resources, set all callback functions to NULL:

`KcsSetCallback(KcsAllFunc, NULL, NULL);`

Arguments

`function`

A set of API functions. See “KcsFunction” on page 38 for the list of functions.

`callback`

The application-supplied function to be called when the variable `function` needs to report progress.

`userDefinedData`

Any user-defined data.

Returns

`KCS_SUCCESS`
`KCS_MEM_ALLOC_ERROR`

Example

Code Example 4-12 `KcsSetCallback()`

```
/* template function declaration */

int myProgressCallback(KcsProfileId profileid, unsigned long
       current, unsigned long total, KcsFunction
       operation, void *userDefinedData);

KcsProfileId   completeProfile;
```

Code Example 4-12 `KcsSetCallback()` *(Continued)*

```
KcsPixelLayout pixelLayoutIn;

/* the profiles have been loaded and connected, now set up the
* callback to be active for both the optimize and evaluate
* functions */

status = KcsSetCallback(KcsOptFunc + KcsEvalFunc,
       (KcsCallbackFunction)myProgressCallback, NULL );
if (status != KCS_SUCCESS) {
    fprintf(stderr, “Callback function call failed\n”);
}

printf(“Optimizing the complete profile \n”);
status = KcsOptimizeProfile(completeProfile, KcsOptSpeed, KcsLoadAllNow);
/* check status here*/
/* set up the pixel layout */
status = KcsEvaluate(completeProfile, op, &pixelLayoutIn, &pixelLayoutIn);
/* check status here*/

/* This is my callback function */

int myProgressCallback(KcsProfileId profileid, unsigned long current,
       unsigned long total, KcsFunction operation, void *userDefinedData)

{
    int     pcent;

    pcent = (int) (((float)current/ (float)total) *100.0);
    fprintf(stderr,”Optimize+Evaluate is %3d percent complete\n”, pcent);
    fflush(stderr);
    return(KCS_SUCCESS);
/* Free callback resources*/
KcsSetCallback (KcsOptFunc+KcsEvalFunc, NULL, NULL);
}
```

KcsUpdateProfile()

```
KcsStatusId
KcsUpdateProfile(KcsProfileId profile,
       KcsCharacterizationData *charact,
       KcsCalibrationData *calib, void *CMMSpecificData)
```

Purpose

Use `KcsUpdateProfile()` to change the profile data in the loaded profile according to the supplied measurement data.

The data supplied to this call depends on the type of device the profile represents. The default CMM currently supports scanners and monitors; printer profiles are *not* currently supported. The C AP also will be used for printers, when implemented by the default or alternative CMMs. The data required for this call depends on whether the profile is *calibrated* or *characterized*.

Characterization refers to defining the generic color response of all devices of the same make and model (normally by making measurements on a number of sample devices to find an average response). Characterization requires colorimetric measurements. Code Example 4-13 on page 95 shows how these measurements are used to update a profile.

Calibration refers to fine-tuning a specific device's color response. It changes the profile data so that it can be color managed to produce the same color response as other devices of the same make and model.

Arguments

`profile`

The identifier of the profile to be updated.

`*charact`

A set of color sample measurements where sample is a color patch on a test target.

For a scanner, this is a target that is scanned. In this case, each sample measurement consists of an input that is the CIE XYZ value of the patch, as measured. The sample output is the RGB value that the scanner produced when scanning the patch. In addition, each color sample contains fields for the sample weight, standard deviation, and sample type. The weight is a hint indicating the importance of the sample color. The default should equal 1.0. The standard deviation is used to indicate the statistics of a set of measurements of the sample color that have been reduced to a single sample. The sample type is used to indicate that a color sample represents either black, white, other, neutral, or chromatic color. For best results, the sample type field should be correctly set for each color sample. For example, the `KcsFluorescent` sample type can be used to tag special color samples with this property. The sample type is a hint passed by the KCMS framework to the CMM.

Note that CIE XYZ values are to be scaled in the range `0.0` to `100.0` and that RGB values are to be scaled in range `0.0` to `1.0`. For additional details, see “KcsCharacterizationData” on page 34.

Monitors do not use the `charact` argument. Pointer `*charact` should be set to `NULL` when `KcsUpdateProfile()` is called for a monitor profile. For a monitor, characterization data consists of the profile attributes `icSigRedColorantTag`, `icSigGreenColorantTag`, `icSigBlueColorantTag` and `icSigMediaWhitePointTag`. These attributes must be set and valid prior to calling `KcsUpdateProfile()`. Use `KcsSetAttribute()` to set these attributes.

`*calib`

The linearization tables needed to calibrate the profile. These tables are required to calibrate all device types. They are also required when calling `KcsUpdateProfile()` to characterize a scanner or monitor. Both the input and output spaces are `KcsRGB` for a scanner and monitor. The RGB samples are scaled in the range of `0.0` to `1.0`.

`*CMMSpecificData`

A pointer to any additional data needed by a specific CMM to update the profile. Refer to the CMM documentation for any specific data required. For use with the default CMM, set this argument to `NULL`.

Returns	`KCS_SUCCESS`
	`KCS_MEM_ALLOC_ERROR`
	`KCS_CC_UPDATE_NEEDS_MORE_DATA`
	`KCS_CC_UPDATE_INVALID_DATA`

Example

To call `KcsUpdateProfile()` successfully, the profile must contain a small number of attributes that identify the type of device the profile represents. It is assumed that the profile already contains these attributes.

An example is given of how to allocate and fill out the arguments required to call `KcsUpdateProfile()`.

Code Example 4-13 `KcsUpdateProfile()`

```
#pragma ident “@(#) kcms_update.c”
/* kcs_update.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <math.h>
#include <kcms/kcs.h>
#include <kcms/kcstypes.h>
#include <kcms/kcsattrb.h>

float   Luminance_float_out[3][256];

/* test code to check profile calibration * /
main(int argc, char **argv)
{
KcsCalibrationData    *calData;
KcsProfileDesc        x_desc, desc;
KcsProfileId          profileid;
KcsStatusId           status;
KcsAttributeValue     attrValue;
KcsErrDesc            errDesc;
int         levels = 256, channels = 3;
int         sizemeas, nvalues, i, j;
FILE        *simfile;
float       input_val;
size_t      rc;

/* Read in the measured calibration data from a file */
/* file lum_out should be located in demo directory with this program */
```

Code Example 4-13 `KcsUpdateProfile()` *(Continued)*

```
if ((simfile = fopen(“lum_out”, “r”)) == NULL) {
    fprintf(stderr,”cannot open output luminance file\n”);
    exit(1);
    }

for (i=0; i<channels; i++)
    for (j=0; j<levels; j++)
        Luminance_float_out[i][j] = 0.0;
nvalues = levels * channels;
rc = fread(Luminance_float_out, sizeof(float), nvalues, simfile);
fclose(simfile);

/* Fill out the measurement structures */
sizemeas = (int) (sizeof(KcsMeasurementBase) + sizeof(long) + levels);

calData = (KcsCalibrationData *) malloc(sizemeas);

calData->base.countSupplied = levels;
calData->base.numInComp = 3;
calData->base.numOutComp = 3;
calData->base.inputSpace = KcsRGB;
calData->base.outputSpace = KcsRGB;
for (i=0; i< levels; i++) {
    calData->val.patch[i].weight = 1.0;
    calData->val.patch[i].standardDeviation = 0.0;
    calData->val.patch[i].sampleType = KcsChromatic;

    calData->val.patch[i].input[KcsRGB_R] = (float)i/255;
    calData->val.patch[i].input[KcsRGB_G] = (float)i/255;
    calData->val.patch[i].input[KcsRGB_B] = (float)i/255;
    calData->val.patch[i].input[3] = 0.0;

    calData->val.patch[i].output[KcsRGB_R] = Luminance_float_out[0][i];
    calData->val.patch[i].output[KcsRGB_G] = Luminance_float_out[1][i];
    calData->val.patch[i].output[KcsRGB_B] = Luminance_float_out[2][i];
    calData->val.patch[i].output[3] = 0.0;
    }

calData->val.patch[0].sampleType = KcsBlack;
calData->val.patch[255].sampleType = KcsWhite;

if (!argv[1]) {
    fprintf(stderr, “Usage kcms_update profile_in [profile_out]\n”);
```

Code Example 4-13 `KcsUpdateProfile()` *(Continued)*

```
    exit(1);
    }
 /* Let the library open the file */
x_desc.type = KcsSolarisProfile;
x_desc.desc.solarisFile.fileName = argv[optind];
x_desc.desc.solarisFile.hostName = NULL;
x_desc.desc.solarisFile.oflag = O_RDWR;
x_desc.desc.solarisFile.mode = 0;

status = KcsLoadProfile(&profileid, &x_desc, KcsLoadAllNow);
if(status != KCS_SUCCESS) {
    status = KcsGetLastError(&errDesc);
    printf("LoadProfile error: %s\n", errDesc.desc);
    }

status = KcsUpdateProfile(profileid, NULL, calData, NULL);
if(status != KCS_SUCCESS) {
    status = KcsGetLastError(&errDesc);
    printf("UpdateProfile error: %s\n", errDesc.desc);
    KcsFreeProfile(profileid);
    exit(1);
    }

if (argv[2]) {
    /* Save to an output file */
    desc.type = KcsSolarisProfile;
    desc.desc.solarisFile.fileName = argv[2];
    desc.desc.solarisFile.hostName = NULL;
    desc.desc.solarisFile.oflag = O_RDWR|O_CREAT|O_TRUNC;
    desc.desc.solarisFile.mode = 0666;
    status = KcsSaveProfile(profileid, &desc);
    if(status != KCS_SUCCESS) {
        status = KcsGetLastError(&errDesc);
        printf("SaveProfile error: %s\n", errDesc.desc);
        }
    KcsFreeProfile(profileid);

    } else {
        /* Just save to the same description */
        status = KcsSaveProfile(profileid, NULL);
        if(status != KCS_SUCCESS) {
            status = KcsGetLastError(&errDesc);
            printf("SaveProfile error: %s\n", errDesc.desc);
        }
```

Code Example 4-13 `KcsUpdateProfile()` *(Continued)*

```
    KcsFreeProfile(profileid);
    }
exit(0);
}
```

KCMS Profile Attributes and ICC Tags

5

Every profile contains a group of attributes (or tags) that describes the characteristics of the profile. Attributes and tags are specified by name, value, and status (whether they are required or optional). Attributes and tags are identical; the term attributes existed before the ICC tags came into use. Attributes are KCMS-framework specific. Tags are primarily defined in the ICC specification (and the `icc.h` header file), except for a few that are KCMS-framework specific. These tags are defined in the `kcstypes.h` header file and are registered with the ICC. Tags are discussed in this chapter.

Several API functions create and modify tags, while others define what is stored in a tag; see Chapter 4, "Functions" for detailed descriptions of all functions.

Tags

A tag is defined with an enumerated constant listed in `icc.h`. The enumerated constant, `icTagSignature` is a list of all available tags. Use the attribute name as an argument to the API calls `KcsGetAttribute()` and `KcsSetAttribute()`.

Attribute Value

An attribute’s value is defined in the `val` field of the `KcsAttributeValue` data structure (see page 27). Since there are many possible data types for `val`, you need some way of interpreting the value as the correct data type. The `KcsAttributeType` data type provides this interface (see page 26).

Required and Optional Attributes

Attributes are either required or optional for all profiles. The software that creates the profile must assign required attributes.

KCMS Framework Tags

The following tags are KCMS-framework specific. They are not defined in the ICC specification (or `icc.h`); they are defined in the `kcstypes.h` header file. These tags are never stored in a profile. They are used to access portions of an ICC profile that are not covered by tags.

icSigHeaderTag

```
#define icSigHeaderTag (0x69636864UL) /* ‘ichd’ */
```

This data structure is an ICC header. The header file contains useful attribute information.

icSigNumTag

```
#define icSigNumTag (0x6E746167UL) /* ‘ntag’ */
```

This data structure returns a `KcsULong` value that is the number of ICC profile attributes and tags in a file. This is a read-only attribute; it cannot be set. The count includes the `icSigHeaderTag`, `icSigNumTag` and `icSigListTag` entries.

icSigListTag

```
#define icSigListTag (0x6C746167UL) /* ‘ltag’ */
```

This data structure is a list of the ICC attributes and tags in a profile.

Example

The following code sample shows you how to use the `icSigNumTag` and `icSigListTag` data structures.

Code Example 5-1 `icSigNumTag` and `icSigListTag`

```
#include <kcms/kcs.h>
KcsAttributeValue  attrValue, *attrPtr;
int                i;
char               *tmp;

/* Set the value of countSupplied */
attrValue.base.countSupplied = 1;
attrValue.base.type = KcsULong;

/* Get the number of attributes in the profile */
status = KcsGetAttribute(profile, icSigNumTag, &attrValue);
if (status != KCS_SUCCESS) {
    KcsFreeProfile(profile);
    exit(1);
}

/* Make space to get a list of all tags */
size = sizeof(KcsAttributeBase) + sizeof(long)*attrValue.val.uLongVal[0];
if ((attrPtr = (KcsAttributeValue *)malloc(size)) == NULL) {
    perror(“malloc failed : “);
    KcsFreeProfile(profile);
    exit(1);
}

/* Get the list of tags */
attrPtr->base.type = KcsULong;
attrPtr->base.sizeOfType = sizeof(long);
attrPtr->base.countSupplied = attrValue.val.uLongVal[0];
status = KcsGetAttribute(profile, icSigListTag, attrPtr);
if (status != KCS_SUCCESS) {
    KcsFreeProfile(profile);
    free (attrPtr);
    exit(1);
}

/* Print the list */
printf(“Number of tags = %d\n”, attrPtr->base.countSupplied);
for (i=0; i<attrPtr->base.countSupplied; i++) {
    tmp = (char *)&attrPtr->val.uLongVal[i];
```

Code Example 5-1 `icSigNumTag` and `icSigListTag` *(Continued)*

```
    printf(“Tag # = %d, Tag Hex = 0x%x, Tag Ascii = %c%c%c%c\n”, i,
       attrPtr->val.uLongVal[i]; *tmp, *(tmp+1), *(tmp+2), *(tmp+3));
}

KcsFreeProfile(profile);
free (attrPtr);
```

Required ICC Tags

Some tags in the profile structure are required by the default CMM. These tags provide the necessary data for the CMM to translate the color information.

The following `ic*` tag names are defined in the `icc.h` header file. The `ic*` data structures are defined starting on page 107 and are in the `icc.h` header file. See the ICC specification for more detailed definitions of device profiles, tag names and tag types (or data structures). ICC specification section titles are referenced in each profile class section below. (By default, the ICC specification is located online in the `SUNWsdk/kcms/doc` directory.)

Note – Lut8 and Lut16 tables are considered transforms by the KCMS framework and are not available as attributes.

The following tags are are required depending on the profile type and interpretation. The first five tags listed cannot be set using `KcsGetAttribute()` and `KcsSetAttribute()`. Instead they must be set with `KcsUpdateProfile()`. The `grayTRCTags` fully support reading and writing. They are required for input profiles only.

Tags Required Depending on Interpretation

Profile	Tag Name	Interpretation
Input Profile	AToB0Tag	None
Display Profile	AToB0Tag	None
Output Profile	BToA0Tag	Perceptual rendering
Output Profile	BToA1Tag	Colorimetric rendering
Output Profile	BToA2Tag	Saturation rendering

Tags Required Depending on Interpretation

Profile	Tag Name	Interpretation
Input Profile	grayTRCTag	Depends on intent
Display Profile	grayTRCTag	Additive
Output Profile	grayTRCTag	Subtractive

Input Profile

The following tags are for input devices such as scanners. See “Input File” in the ICC specification for more information.

Monochrome Input Profiles

Tag Name	Tag Type
`icSigHeaderTag`	`icHeader`
`icSigProfileDescriptionTag`	`icTextDescription`
`icSigGrayTRCTag`	`icCurve`
`icSigMediaWhitePointTag`	`icXYZArray`
`icSigCopyrightTag`	`icText`

RGB Input Profiles

Tag Name	Tag Type
`icSigHeaderTag`	`icHeader`
`icSigProfileDescriptionTag`	`icTextDescription`
`icSigRedColorantTag`	`icXYZArray`
`icSigGreeColorantTag`	`icXYZArray`
`icSigBlueColorantTag`	`icXYZArray`
`icSigRedTRCTag`	`icCurve`
`icSigGreenTRCTag`	`icCurve`
`icSigBlueTRCTag`	`icCurve`
`icSigMediaWhitePointTag`	`icXYZArray`
`icSigCopyrightTag`	`icText`

CMYK Input Profiles

Tag Name	Tag Type
`icSigHeaderTag`	`icHeader`
`icSigProfileDescriptionTag`	`icTextDescription`
`icSigMediaWhitePointTag`	`icXYZArray`
`icSigCopyrightTag`	`icText`

Output Profile

The following tags are required for output devices such as printers. See “Output Profile” in the ICC specification for more information.

Monochrome Output Profiles

Tag Name	**Tag Type**
`icSigHeaderTag`	`icHeader`
`icSigProfileDescriptionTag`	`icTextDescription`
`icSigGrayTRCTag`	`icCurve`
`icSigMediaWhitePointTag`	`icXYZArray`
`icSigCopyrightTag`	`icText`

RGB and CMYK Output Profiles

Tag Name	**Tag Type**
`icSigHeaderTag`	`icHeader`
`icSigProfileDescriptionTag`	`icTextDescription`
`icSigMediaWhitePointTag`	`icXYZArray`
`icSigCopyrightTag`	`icText`

Device Link Profile

The device link profile is for a link or connection between devices. The following tags are for device link profiles. See “DeviceLink Profile” in the ICC specification for more information.

Tag Name	**Tag Type**
`icSigHeaderTag`	`icHeader`
`icSigProfileDescriptionTag`	`icTextDescription`
`icSigProfileSequenceDescTag`	`icProfileSequenceDesc`
`icSigCopyrightTag`	`icText`

Color Space Conversion Profile

The color space conversion profile is for color space transformation between non-device color spaces and the profile connection space (PCS). The following tags are for color space conversion profiles. See “ColorSpaceConversion Profile” in the ICC specification for more information.

Tag Name	**Tag Type**
`icSigHeaderTag`	`icHeader`
`icSigProfileDescriptionTag`	`icTextDescription`
`icSigMediaWhitePointTag`	`icXYZArray`
`icSigCopyrightTag`	`icText`

Abstract Profile

The abstract profile is for color transformations between PCS and PCS. The following tags are for abstract profiles. See “Abstract Profile” in the ICC specification for more information.

Tag Name	**Tag Type**
`icSigHeaderTag`	`icHeader`
`icSigProfileDescriptionTag`	`icTextDescription`
`icSigMediaWhitePointTag`	`icXYZArray`
`icSigCopyrightTag`	`icText`

List of All Tags

Tag Name	**Tag Type**
`icSigHeaderTag`	`icHeader`
`icSigBlueColorantTag`	`icXYZArray`
`icSigBlueTRCTag`	`icCurve`
`icSigCalibrationDateTimeTag`	`icSigDateTimeType`
`icSigCharTargetTag`	`icText`
`icSigCopyrightTag`	`icText`
`icSigDeviceMfgDescTag`	`icTextDescription`
`icSigDeviceModelDescTag`	`icTextDescription`
`icSigGrayTRCTag`	`icCurve`
`icSigGreenColorantTag`	`icXYZArray`
`icSigGreenTRCTag`	`icCurve`
`icSigLuminanceTag`	`icXYZArray`
`icSigMeasurementTag`	`icMeasurement`
`icSigMediaBlackPointTag`	`icXYZArray`
`icSigMediaWhitePointTag`	`icXYZArray`
`icSigNamedColorTag`	`icNamedColor`
`icSigProfileDescriptionTag`	`icTextDescription`
`icSigProfileSequenceDescTag`	`icProfileSequenceDesc`
`icSigPs2CRD0Tag`	`icData`
`icSigPs2CRD1Tag`	`icData`
`icSigPs2CRD2Tag`	`icData`
`icSigPs2CRD3Tag`	`icData`
`icSigPs2CSATag`	`icData`
`icSigPs2RenderingIntentTag`	`icData`
`icSigRedColorantTag`	`icXYZArray`
`icSigRedTRCTag`	`icSigCurve`
`icSigScreeningDescTag`	`icTextDescription`
`icSigScreeningTag`	`icScreening`
`icSigTechnologyTag`	`icSignature`
`icSigUcrBgTag`	`icUcrBg`
`icSigViewingCondDescTag`	`icTextDescription`
`icSigViewingConditionsTag`	`icViewingConditions`

Tag Types

The following data structures are used only with tags (or attributes) and are defined in the `icc.h` header file. All other data structures in the KCMS framework API are defined in Chapter 3, “Data Structures” and in the `kcstypes.h` header file.

Constants

```
#define icMagicNumber 0x61637370L /* 'acsp' */
#define icVersionNumber 0x02000000L /* 2.0, BCD */
```

Screen Encodings

```
#define icPrtrDefaultScreensFalse 0x00000000L /* Bit position 0 */
#define icPrtrDefaultScreensTrue  0x00000001L /* Bit position 0 */
#define icLinesPerInch            0x00000002L /* Bit position 1 */
#define icLinesPerCm              0x00000000L /* Bit position 1 */
```

Device Attributes

The defined values correspond to the low four bytes of the eight-byte attribute quantity; see `icc.h` for their location.

```
#define icReflective              0x00000000L /* Bit position 0 */
#define icTransparency            0x00000001L /* Bit position 0 */
#define icGlossy                  0x00000000L /* Bit position 1 */
#define icMatte                   0x00000002L /* Bit position 1 */
```

Profile Header Flags

The low 16 bits are reserved for the ICC.

```
#define icEmbeddedProfileFalse    0x00000000L /* Bit position 0 */
#define icEmbeddedProfileTrue     0x00000001L /* Bit position 0 */
#define icUseAnywhere             0x00000000L /* Bit position 1 */
#define icUseWithEmbeddedDataOnly 0x00000002L /* Bit position 1 */
```

ASCII or Binary Data

```
#define icAsciiData               0x00000000L /* Used in dataType */
#define icBinaryData              0x00000001L
```

Variable-Length Array

```
#define icAny                    1
```

Signatures

Signatures are 4-byte identifiers used to translate platform definitions to `ic*` form and to differentiate between tags and other items in the profile format. Set `icSignature` as appropriate for your operating system.

`icSignature`

This `icSignature` is for the Solaris operating system. Note the number definitions.

```
#if defined(sun) || defined(__sun)
typedef long icSignature;

/* Number Definitions */

/* Unsigned Integer Numbers */
typedef unsigned char icUInt8Number;
typedef unsigned shorticUInt16Number;
typedef unsigned long icUInt32Number;
typedef unsigned long icUInt64Number[2];

/* Signed Integer Numbers */
typedef char          icInt8Number;
typedef short         icInt16Number;
typedef long          icInt32Number;
typedef long          icInt64Number[2];

/* Fixed Numbers */
typedef long          icS15Fixed16Number;
typedef unsigned long     icU16Fixed16Number;
#endif /* 32-bit Solaris, SunOS */
```

`icTagSignature`

```
typedef enum {
    icSigAToB0Tag      = 0x41324230L,/* 'A2B0' */
    icSigAToB1Tag      = 0x41324231L,/* 'A2B1' */
    icSigAToB2Tag      = 0x41324232L,/* 'A2B2' */
    icSigBlueColorantTag  = 0x6258595AL,/* 'bXYZ' */
    icSigBlueTRCTag    = 0x62545243L,/* 'bTRC' */
    icSigBToA0Tag      = 0x42324130L,/* 'B2A0' */
    icSigBToA1Tag      = 0x42324131L,/* 'B2A1' */
    icSigBToA2Tag      = 0x42324132L,/* 'B2A2' */
    icSigCalibrationDateTimeTag   = 0x63616C74L,/* 'calt' */
    icSigCharTargetTag    = 0x74617267L,/* 'targ' */
    icSigCopyrightTag     = 0x63707274L,/* 'cprt' */
    icSigDeviceMfgDescTag = 0x646D6E64L,/* 'dmnd' */
    icSigDeviceModelDescTag   = 0x646D6464L,/* 'dmdd' */
    icSigGamutTag         = 0x676d7420L,/* 'gmt ' */
    icSigGrayTRCTag       = 0x6b545243L,/* 'kTRC' */
    icSigGreenColorantTag = 0x6758595AL,/* 'gXYZ' */
    icSigGreenTRCTag      = 0x67545243L,/* 'gTRC' */
    icSigLuminanceTag     = 0x6C756d69L,/* 'lumi' */
    icSigMeasurementTag   = 0x6D656173L,/* 'meas' */
    icSigMediaBlackPointTag   = 0x626B7074L,/* 'bkpt' */
    icSigMediaWhitePointTag   = 0x77747074L,/* 'wtpt' */
    icSigNamedColorTag    = 0x6E636f6CL,/* 'ncol' */
    icSigPreview0Tag      = 0x70726530L,/* 'pre0' */
    icSigPreview1Tag      = 0x70726531L,/* 'pre1' */
    icSigPreview2Tag      = 0x70726532L,/* 'pre2' */
    icSigProfileDescriptionTag    = 0x64657363L,/* 'desc' */
    icSigProfileSequenceDescTag   = 0x70736571L,/* 'pseq' */
    icSigPs2CRD0Tag    = 0x70736430L,/* 'psd0' */
    icSigPs2CRD1Tag    = 0x70736431L,/* 'psd1' */
    icSigPs2CRD2Tag    = 0x70736432L,/* 'psd2' */
    icSigPs2CRD3Tag    = 0x70736433L,/* 'psd3' */
    icSigPs2CSATag     = 0x70733273L,/* 'ps2s' */
```

```
    icSigPs2RenderingIntentTag    = 0x70733269L,/* 'ps2i' */
    icSigRedColorantTag           = 0x7258595AL,/* 'rXYZ' */
    icSigRedTRCTag                = 0x72545243L,/* 'rTRC' */
    icSigScreeningDescTag         = 0x73637264L,/* 'scrd' */
    icSigScreeningTag             = 0x7363726EL,/* 'scrn' */
    icSigTechnologyTag            = 0x74656368L,/* 'tech' */
    icSigUcrBgTag                 = 0x62666420L,/* 'bfd ' */
    icSigViewingCondDescTag       = 0x76756564L,/* 'vued' */
    icSigViewingConditionsTag = 0x76696577L,/* 'view' */
    icMaxEnumTag              = 0xFFFFFFFFL/* enum = 4 bytes max */
} icTagSignature;
```

icTechnologySignature

```
typedef enum {
    icSigFilmScanner          = 0x6673636EL,     /* 'fscn' */
    icSigReflectiveScanner= 0x7273636EL,     /* 'rscn' */
    icSigInkJetPrinter        = 0x696A6574L,     /* 'ijet' */
    icSigThermalWaxPrinter= 0x74776178L,     /* 'twax' */
    icSigElectrophotographicPrinter= 0x6570686FL,    /* 'epho' */
    icSigElectrostaticPrinter     = 0x65737461L,     /* 'esta' */
    icSigDyeSublimationPrinter    = 0x64737562L,     /* 'dsub' */
    icSigPhotographicPaperPrinter = 0x7270686FL,     /* 'rpho' */
    icSigFilmWriter           = 0x6670726EL,     /* 'fprn' */
    icSigVideoMonitor         = 0x7669646DL,     /* 'vidm' */
    icSigVideoCamera          = 0x76696463L,     /* 'vidc' */
    icSigProjectionTelevision     = 0x706A7476L,     /* 'pjtv' */
    icSigCRTDisplay           = 0x43525420L,     /* 'CRT ' */
    icSigPMDisplay            = 0x504D4420L,     /* 'PMD ' */
    icSigAMDisplay            = 0x414D4420L,     /* 'AMD ' */
    icSigPhotoCD              = 0x4B504344L,     /* 'KPCD' */
    icSigPhotoImageSetter     = 0x696D6773L,     /* 'imgs' */
    icSigGravure              = 0x67726176L,     /* 'grav' */
    icSigOffsetLithography= 0x6F666673L,     /* 'offs' */
    icSigSilkscreen           = 0x73696C6BL,     /* 'silk' */
    icSigFlexography          = 0x666C6578L,     /* 'flex' */
    icMaxEnumTechnology       = 0xFFFFFFFFL/* enum = 4 bytes max */
} icTechnologySignature;
```

icTagTypeSignature

```
typedef enum {
    icSigCurveType        = 0x63757276L,/* 'curv' */
    icSigDataType         = 0x64617461L,/* 'data' */
    icSigDateTimeType     = 0x6474696DL,/* 'dtim' */
    icSigLut16Type        = 0x6d667432L,/* 'mft2' */
    icSigLut8Type         = 0x6d667431L,/* 'mft1' */
    icSigMeasurementType  = 0x6D656173L,/* 'meas' */
    icSigNamedColorType   = 0x6E636f6CL,/* 'ncol' */
    icSigProfileSequenceDescType= 0x70736571L,/* 'pseq' */
    icSigS15Fixed16ArrayType  = 0x73663332L,/* 'sf32' */
    icSigScreeningType    = 0x7363726EL,/* 'scrn' */
    icSigSignatureType    = 0x73696720L,/* 'sig ' */
    icSigTextType         = 0x74657874L,/* 'text' */
    icSigTextDescriptionType  = 0x64657363L,/* 'desc' */
    icSigU16Fixed16ArrayType  = 0x75663332L,/* 'uf32' */
    icSigUcrBgType        = 0x62666420L,/* 'bfd ' */
    icSigUInt16ArrayType  = 0x75693136L,/* 'ui16' */
    icSigUInt32ArrayType  = 0x75693332L,/* 'ui32' */
    icSigUInt64ArrayType  = 0x75693634L,/* 'ui64' */
    icSigUInt8ArrayType   = 0x75693038L,/* 'ui08' */
    icSigViewingConditionsType= 0x76696577L,/* 'view' */
    icSigXYZType          = 0x58595A20L,/* 'XYZ ' */
    icSigXYZArrayType     = 0x58595A20L,/* 'XYZ ' */
    icMaxEnumType         = 0xFFFFFFFFL/* enum = 4 bytes max */
} icTagTypeSignature;
```

Color Space Signature

icColorSpaceSignature

`icColorSpaceSignature` is used in the `icHeader` structure.

```
typedef enum {
    icSigXYZData   = 0x58595A20L,/* 'XYZ ' */
    icSigLabData   = 0x4C616220L,/* 'Lab ' */
    icSigLuvData   = 0x4C757620L,/* 'Luv ' */
    icSigYCbCrData = 0x59436272L,/* 'YCbr' */
    icSigYxyData   = 0x59787920L,/* 'Yxy ' */
    icSigRgbData   = 0x52474220L,/* 'RGB ' */
    icSigGrayData  = 0x47524159L,/* 'GRAY' */
    icSigHsvData   = 0x48535620L,/* 'HSV ' */
    icSigHlsData   = 0x484C5320L,/* 'HLS ' */
    icSigCmykData  = 0x434D594BL,/* 'CMYK' */
    icSigCmyData   = 0x434D5920L,/* 'CMY ' */
    icMaxEnumData  = 0xFFFFFFFFL/* enum = 4 bytes max */
} icColorSpaceSignature;
```

Note – Currently, only `icSigXYZData` and `icSigLabData` are valid profile connection spaces (PCSs).

icProfileClassSignature

`icProfileClassSignature` is used in the `icHeader` structure.

```
/* profileClass enumerations */
typedef enum {
    icSigInputClass    = 0x73636E72L,/* 'scnr' */
    icSigDisplayClass  = 0x6D6E7472L,/* 'mntr' */
    icSigOutputClass   = 0x70727472L,/* 'prtr' */
    icSigLinkClass     = 0x6C696E6BL,/* 'link' */
    icSigAbstractClass = 0x61627374L,/* 'abst' */
    icSigColorSpaceClass   = 0x73706163L,/* 'spac' */
    icMaxEnumClass     = 0xFFFFFFFFL/* enum = 4 bytes max */
} icProfileClassSignature;
```

`icPlatformSignature`

`icPlatformSignature` is used in the `icHeader` structure.

```
/* Platform Signatures */
typedef enum {
    icSigMacintosh     = 0x4150504CL,/* 'APPL' */
    icSigMicrosoft     = 0x4D534654L,/* 'MSFT' */
    icSigSolaris       = 0x53554E57L,/* 'SUNW' */
    icSigSGI           = 0x53474920L,/* 'SGI ' */
    icSigTaligent      = 0x54474E54L,/* 'TGNT' */
    icMaxEnumPlatform  = 0xFFFFFFFFL/* enum = 4 bytes max */
} icPlatformSignature;
```

Other Enums

`icMeasurementFlare`

`icMeasurementFlare` is used in the `icMeasurement` structure.

```
/* Measurement Flare, used in the measurmentType tag */
typedef enum {
    icFlare0       = 0x00000000L,     /* 0% flare */
    icFlare100     = 0x00000001L,     /* 100% flare */
    icMaxFlare     = 0xFFFFFFFFL      /* enum = 4 bytes max */
} icMeasurementFlare;
```

`icMeasurementGeometry`

`icMeasurementGeometry` is used in the `icMeasurement` structure.

```
/* Measurement Geometry, used in the measurmentType tag */
typedef enum {
    icGeometryUnknown      = 0x00000000L,/* Unknown geometry */
    icGeometry045or450     = 0x00000001L,/* 0/45 or 45/0 */
    icGeometry0dord0       = 0x00000002L,/* 0/d or d/0 */
    icMaxGeometry          = 0xFFFFFFFFL/* enum = 4 bytes max */
} icMeasurementGeometry;
```

icRenderingIntent

`icRenderingIntent` is used in the `icHeader` structure.

```
/* Rendering Intents, used in the profile header */
typedef enum {
    icPerceptual            = 0,
    icRelativeColorimetric= 1,
    icSaturation            = 2,
    icAbsoluteColorimetric= 3,
    icMaxEnumIntent         = 0xFFFFFFFFL/* enum = 4 bytes max */
} icRenderingIntent;
```

icSpotShape

`icSpotShape` is used in the `icScreening` structure.

```
/* Different Spot Shapes currently defined, used for
screeningType */
typedef enum {
    icSpotShapeUnknown        = 0,
    icSpotShapePrinterDefault = 1,
    icSpotShapeRound          = 2,
    icSpotShapeDiamond        = 3,
    icSpotShapeEllipse        = 4,
    icSpotShapeLine           = 5,
    icSpotShapeSquare         = 6,
    icSpotShapeCross          = 7,
    icMaxEnumSpot           = 0xFFFFFFFFL/* enum = 4 bytes max */
} icSpotShape;
```

icStandardObserver

`icStandardObserver` is used in the `Measurement` structure.

```
/* Standard Observer, used in the measurementType tag */
typedef enum {
    icStdObsUnknown        = 0x00000000L, /* Unknown observer */
    icStdObs1931TwoDegrees= 0x00000001L, /* 1931 two degrees */
    icStdObs1964TenDegrees= 0x00000002L, /* 1961 ten degrees */
    icMaxStdObs            = 0xFFFFFFFFL/* enum = 4 bytes max */
} icStandardObserver;
```

`icIlluminant`

`icIlluminant` is used in the `icMeasurement` structure.

```
/* Pre-defined illuminants, used in measurement and viewing
 * conditions type */
typedef enum {
    icIlluminantUnknown    = 0x00000000L,
    icIlluminantD50        = 0x00000001L,
    icIlluminantD65        = 0x00000002L,
    icIlluminantD93        = 0x00000003L,
    icIlluminantF2         = 0x00000004L,
    icIlluminantD55        = 0x00000005L,
    icIlluminantA          = 0x00000006L,
    icIlluminantEquiPowerE= 0x00000007L,/* Equi-Power (E) */
    icIlluminantF8         = 0x00000008L,
    icMaxEnumIluminant     = 0xFFFFFFFFL/* enum = 4 bytes max */
} icIlluminant;
```

Arrays of Numbers

These arrays are variable in length and type. They are implemented with the `icAny` constant instead of pointers. The `icAny` constant is a single-byte array that allows you to extend the data structure by allocating more data.

icInt8Number

```
typedef struct {
    icInt8Number   data[icAny];
} icInt8Array;
```

icUInt8Number

```
typedef struct {
    icUInt8Number  data[icAny];
} icUInt8Array;
```

icUInt16Number

```
typedef struct {
    icUInt16Number     data[icAny];
} icUInt16Array;
```

icInt16Array

```
typedef struct {
    icInt16Number  data[icAny];
} icInt16Array;
```

icUInt32Number

```
typedef struct {
    icUInt32Number     data[icAny];
} icUInt32Array;
```

`icInt32Array`

```
typedef struct {
    icInt32Number  data[icAny];
} icInt32Array;
```

`icUInt64Number`

```
typedef struct {
    icUInt64Number     data[icAny];
} icUInt64Array;
```

`icInt64Number`

```
typedef struct {
    icInt64Number  data[icAny];
} icInt64Array;
```

`icU16Fixed16Number`

```
typedef struct {
    icU16Fixed16Number data[icAny];
} icU16Fixed16Array;
```

`icS15Fixed16Number`

```
typedef struct {
    icS15Fixed16Number data[icAny];
} icS15Fixed16Array;
```

icDateTimeNumber

```
/* The base date time number */
typedef struct {
    icUInt16Number    year;
    icUInt16Number    month;
    icUInt16Number    day;
    icUInt16Number    hours;
    icUInt16Number    minutes;
    icUInt16Number    seconds;
} icDateTimeNumber;
```

icXYZNumber

```
typedef struct {
    icS15Fixed16Number    X;
    icS15Fixed16Number    Y;
    icS15Fixed16Number    Z;
} icXYZNumber;
```

icXYZArray

```
typedef struct {
    icXYZNumberdata[icAny]; /* Variable array of XYZ numbers */
} icXYZArray;
```

icCurve

```
typedef struct {
    icUInt32Numbercount;      /* Number of entries */
    icUInt16Numberdata[icAny];/* The actual table data, real
                               * number is determined by count.
                               * Interpretation depends on data
                               * use and tag. */
} icCurve;
```

`icData`

```
typedef struct {
    icUInt32Number dataFlag;  /* 0 = ascii, 1 = binary */
    icInt8Number   data[icAny];/*Data,size determined from tag */
} icData;
```

Note – Lut8 (`icLut8`) and Lut16 (`icLut16`) tables are considered transforms by the KCMS framework and are not available as attributes.

---

icMeasurement

```
typedef struct {
    icStandardObserver    stdObserver;/* Standard observer */
    icXYZNumber           backing;   /* XYZ for backing material */
    icMeasurementGeometry geometry;  /* Measurement geometry */
    icMeasurementFlare    flare;     /* Measurement flare */
    icIlluminant          illuminant;/* Illuminant */
} icMeasurement;
```

icDescStruct

```
typedef struct {
    icSignature       deviceMfg;     /* Device Manufacturer */
    icSignature       deviceModel;   /* Decvice Model */
    icUInt64Number    attributes;    /* Device attributes */
    icTechnologySignature technology;/* Technology signature */
    icInt8Number   data[icAny];   /* Descriptions text follows */

/*  Data that follows is of this form, this is an icInt8Number
 *  to avoid problems with a compiler generating  bad code as
 *  these arrays are variable in length.
 * icTextDescription  deviceMfgDesc; * Manufacturer text
 * icTextDescription  modelDesc;     * Model text */

} icDescStruct;
```

icProfileSequenceDesc

```
typedef struct {
    icUInt32Number count;     /* Number of descriptions */
    icUInt8Number  data[icAny];/* Array of description struct */
} icProfileSequenceDesc;
```

icTextDescription

```
typedef struct {
    icUInt32Number count;        /* Description length */
    icInt8Number   data[icAny];  /* Descriptions follow */

/* Data that follows is of this form
 * icInt8Number    desc[count]    * NULL terminated ascii string
 * icUInt32Number  ucLangCode;    * UniCode language code
 * icUInt32Number  ucCount;       * UniCode description length
 * icInt16Number   ucDesc[ucCount];* The UniCode description
 * icUInt16Number  scCode;        * ScriptCode code
 * icUInt8Number   scCount;       * ScriptCode count
 * icInt8Number    scDesc[67];    * ScriptCode Description */

} icTextDescription;
```

icScreeningData

```
typedef struct {
    icS15Fixed16Number    frequency; /* Frequency */
    icS15Fixed16Number    angle;     /* Screen angle */
    icSpotShape           spotShape; /* Spot Shape encodings */
} icScreeningData;
```

icScreening

```
typedef struct {
    icUInt32Number        screeningFlag; /* Screening flag */
    icUInt32Number        channels;      /* Number of channels */
    icScreeningData       data[icAny];   /* Array of screening data
*/
} icScreening;
```

icText

```
typedef struct {
    icInt8Numberdata[icAny];/* Variable array of chars */
} icText;
```

icUcrBgCurve

```
/* Structure describing either a UCR or BG curve */
typedef struct {
    icUInt32Number    count;         /* Curve length */
    icUInt16Number    curve[icAny];/* The array of curve values */
} icUcrBgCurve;
```

icUcrBg

```
/* Under color removal, black generation */
typedef struct {
    icInt8Number   data[icAny];/* The Ucr BG data */
/* Data that follows is of this form. icUcrBg is an icInt8Number
 * for a compiler and variable-length arrays.
 * icUcrBgCurve    ucr;    * Ucr curve
 * icUcrBgCurve    bg;     * Bg curve
 * icInt8Number string[]; * Description string */
} icUcrBg;
```

icViewingCondition

```
typedef struct {
    icXYZNumber    illuminant;/* In candelas per metre sq'd */
    icXYZNumber    surround;  /* In candelas per metre sq'd */
    icIlluminant   stdIluminant;/* See icIlluminant defines */
} icViewingCondition;
```

Tag Type Definitions

The following tag type definitions are in the `icc.h` header file.

icTagBase

```
typedef struct {
    icTagTypeSignature sig;          /* Signature */
    icInt8Number       reserved[4];  /* Reserved, set to 0 */
} icTagBase;
```

icCurveType

```
typedef struct {
    icTagBase      base;          /* “curv” signature */
    icCurve        curve;         /* curve data */
} icCurveType;
```

icDataType

```
typedef struct {
    icTagBase      base;          /* “data” signature */
    icData         data;          /* data structure */
} icDataType;
```

icDateTimeType

```
typedef struct {
    icTagBase      base;          /* “dtim” signature */
    icData         data;          /* date */
} icDateTimeType;
```

icLut16Type

```
typedef struct {
    icTagBase      base;          /* “mft2” signature */
    icLut16        lut;           /* Lut16 data*/
} icLut16Type;
```

Note – Lut16 (`icLut16`) tables are considered transforms by the KCMS framework and are not available as attributes. You cannot use this with `KcsGetAttribute()` and `KcsSetAttribute()`.

`icLut8Type`

```
typedef struct {
    icTagBase      base;          /* “mft1” signature */
    icLut8         lut;           /* Lut8 data*/
} icLut8Type;
```

Note – Lut8 (`icLut8`) tables are considered transforms by the KCMS framework and are not available as attributes. You cannot use this with `KcsGetAttribute()` and `KcsSetAttribute()`.

icMeasurementType

```
typedef struct {
    icTagBase      base;          /* “meas” signature */
    icMeasurement  measurement;   /* measurement data*/
} icMeasurementType;
```

icNamedColorType

```
typedef struct {
    icTagBase      base;          /* “ncol” signature */
    icNamedColor   ncolor;        /* named color data*/
} icNamedColorType;
```

icProfileSequenceType

```
typedef struct {
    icTagBase      base;          /* “pseq” signature */
    icProfileSequence  desc;      /* seq description data*/
} icProfileSequenceType;
```

icTextDescriptionType

```
typedef struct {
    icTagBase      base;          /* “desc” signature */
    icTextDescription  desc;      /* description data*/
} icTextDescriptionType;
```

icS15Fixed16ArrayType

```
typedef struct {
    icTagBase          base;      /* “sf32” signature */
    icS15Fixed16Array  data;      /* array of values */
} icS15Fixed16ArrayType;
```

icScreeningType

```
typedef struct {
    icTagBase      base;      /* “scrn” signature */
    icScreening    screen;    /* screening structure */
} icScreeningType;
```

icSignatureType

```
typedef struct {
    icTagBase      base;      /* “sig” signature */
    icSignature    signature; /* signature data */
} icSignatureType;
```

icTextType

```
typedef struct {
    icTagBase      base;          /* “text” signature */
    icText         data;      /* variable array of chars */
} icTextType;
```

icU16Fixed16ArrayType

```
typedef struct {
    icTagBase          base;      /* “uf32” signature */
    icU16Fixed16Array  data;      /* variable array of values */
} icU16Fixed16ArrayType;
```

icUcrBgType

```
typedef struct {
    icTagBase          base;      /* “bfd” signature */
    icUcrBg            data;      /* ucrBg structure*/
} icUcrBgType;
```

icUInt16ArrayType

```
typedef struct {
    icTagBase          base;      /* “ui16” signature */
    icUInt16Array      data;      /* variable array of values */
} icUInt16ArrayType;
```

icUInt32ArrayType

```
typedef struct {
    icTagBase          base;      /* “ui32” signature */
    icUInt32Array      data;      /* variable array of values */
} icUInt32ArrayType;
```

icUInt64ArrayType

```
typedef struct {
    icTagBase          base;      /* “ui64” signature */
    icUInt64Array      data;      /* variable array of values */
} icUInt64ArrayType;
```

icUInt8ArrayType

```
typedef struct {
    icTagBase          base;      /* “ui08” signature */
    icUInt8Array       data;      /* variable array of values */
} icUInt8ArrayType;
```

icViewingConditionType

```
typedef struct {
    icTagBase          base;      /* “view” signature */
    icViewingCondition view;      /* viewing conditions*/
} icViewingConditionType;
```

`icXYZType`

```
typedef struct {
    icTagBase          base;   /* “XYZ” signature */
    icXYZArray         data;   /* variable array of XYZ numbers */
} icXYZType;
```

KCMS-Specific Tag Definitions

The following definitions are KCMS-specific and in the `icc.h`. These definitions are registered with the ICC.

icTag

```
typedef struct {
    icTagSignature sig;    /* tag signature */
    icUInt32Number offset; /* start of tag relative to start of
                            * header, See ICC spec, sect 8 */
    icUInt32Number size;   /* size in bytes */
} icTag;
```

icTagList

```
typedef struct {
    icUInt32Number count;          /* number of tags in profile */
    icTag          tags[icAny];    /* variable array of tags */
} icTagList;
```

icHeader

```
typedef struct {
    icUInt32Number             size;       /* Profile size in bytes */
    icSignature                cmmId;      /* CMM for this profile */
    icUInt32Number             version;    /* Format version number */
    icProfileClassSignature    deviceClass;/* Type of profile */
    icColorSpaceSignature      colorSpace;/* Color space of data*/
    icColorSpaceSignature      pcs;        /* PCS, XYZ or Lab only */
    icDateTimeNumber           date;       /* Date profile was created */
    icSignature                magic;      /* icMagicNumber */
    icPlatformSignature        platform;   /* Primary Platform */
    icUInt32Number             flags;      /* Various bit settings */
    icSignature                manufacturer;   /* Device manufacturer */
    icUInt32Number             model;      /* Device model number */
    icUInt64Number             attributes;/* Device attributes */
    icUInt32Number             renderingIntent;    /* Rendering intent */
    icXYZNumber                illuminant;/* Profile illuminant */
    icInt8Number               reserved[48];       /* Reserved for future */
} icHeader;
```

icProfile

```
typedef struct {
    icHeader       header;     /* header */
    icUInt32Number count;  /* number of tags in profile */
    icInt8Number   data[icAny];/* tagTable and tagData */

/* Data the follows is of this form:
 * icTag           tagTable[icAny];   * tag table
 * icInt8Number    tagData[icAny];    * tag data
 */

} icProfile;
```

Warning and Error Messages 6

Every KCMS C API function returns warning and error messages in a status code (in `KcsStatusId`) to indicate whether it executed successfully or, if it did not, why it failed. If a function successfully executes, it returns the `KCS_SUCCESS` status code. If a function is cancelled before its completion, it returns the `KCS_OPERATION_CANCELLED` status code. Any other returned status code indicates a problem. This chapter describes each warning and error message and provides information on localizing the messages.

The status codes are defined in `/usr/openwin/include/kcms/kcsstats.h`.

Warnings

A returned status code in the range `KCS_WARNINGS_START` to `KCS_WARNINGS_END` indicates a warning. Table 6-1 describes the warning constants that the C API functions return.

Table 6-1 Warning Codes

Enumerated Warning Constant	**Description**
`KCS_WARNINGS_START`	The beginning of the defined warnings.
`KCS_ATTR_LARGE_CT_SUPPLIED`	Attribute count supplied field was unexpectedly large.
`KCS_CANNOT_DEOPTIMIZE`	Original data not available so optimization cannot be changed.

Table 6-1 Warning Codes

Enumerated Warning Constant	Description
`KCS_CANNOT_OPTIMIZE`	This profile cannot be optimized.
`KCS_OPERATION_CANCELLED`	This operation was cancelled by the application's user.
`KCS_SPEC_CMM_NOT_FOUND`	Specified CMM was not found.
`KCS_TRUNCATED`	The buffer you supplied was too small. Therefore, the data in it was truncated.
`KCS_WARNINGS_END`	Marks end of `KcsStatusId` warnings currently defined.

Errors

A returned status code in the range `KCS_ERRORS_START` to `KCS_ERRORS_END` indicates a call error. Table 6-2 describes the error messages returned by the C API.

Table 6-2 Error Codes

Enumerated Error Constant	Description
General Failures:	
`KCS_ERRORS_START`	Beginning of errors.
`KCS_NOT_AVAILABLE`	KCMS has not been installed or is not available.
Memory:	
`KCS_MEM_ALLOC_ERR`	Memory allocation error.
OS:	
`KCS_OS_ERR`	General OS error.
IO:	
`KCS_IO_READ_ERR`	Read error.
`KCS_IO_WRITE_ERR`	Write error.
`KCS_IO_SEEK_ERR`	Seek error.

Table 6-2 Error Codes *(Continued)*

Enumerated Error Constant	Description
`KCS_IO_UNKNOWN_TYPE_ERR`	An unknown `KcsProfileDesc` type entry was found.
Solaris File	
`KCS_SOLARIS_FILE_NOT_OPENED`	Cannot open profile.
`KCS_SOLARIS_FILE_RO`	Cannot open profile for writing.
`KCS_SOLARIS_FILE_LOCKED`	Profile is locked by another process.
`KCS_SOLARIS_FILE_NAME_NULL`	Filename pointer is `NULL`.
X11 Profile:	
`KCS_X11_DATA_NULL`	Display or visual pointer is `NULL`.
`KCS_X11_PROFILE_NOT_LOADED`	Cannot load profile; may be locked or does not exist.
`KCS_X11_PROFILE_RO`	Remote X11 profiles are read only.
Profile:	
`KCS_PROF_ID_BAD`	Invalid profile ID.
`KCS_PROF_FORMAT_BAD`	Profile format error.
`KCS_PROF_CT_EXCEEDS_PROF_LIST`	Number of profiles on list is smaller than argument count.
`KCS_PROF_INCOMPLETE`	Incomplete profile specified.
`KCS_PROF_NO_DATA_SUPPORT_4_REQUEST`	
`KCS_PROF_REQ_ATTRS_INCOMPLETE`	
Attributes:	
`KCS_ATTR_NAME_OUT_OF_RANGE`	Specified attribute is out of range.
`KCS_ATTR_TYPE_UNKNOWN`	Attribute type supplied by user is not known.
`KCS_ATTR_LOAD_FORMAT_INCORRECT`	The format of the attribute does not match specifications upon loading.
`KCS_ATTR_LOAD_FLOAT_ERR`	Error interpreting a float upon loading.

Table 6-2 Error Codes *(Continued)*

Enumerated Error Constant	**Description**
`KCS_ATTR_LOAD_INT_ERR`	Error interpreting an integer upon loading.
`KCS_ATTR_DATE_TIME_FORMAT`	The format of the date time stamp does not match specifications.
`KCS_ATTR_CT_ZERO_OR_NEG`	The count supplied in `KcsAttributeValue` was zero or negative.
`KCS_ATTR_READ_ONLY`	Attempting to set an attribute that is read only.
Connection:	
`KCS_CONNECT_FAILED`	Pair of profiles could not be connected.
`KCS_CONNECT_PRECISION_UNACCEPTABLE`	Profile connect will result in unacceptable precision.
`KCS_CONNECT_OPT_FORCED_DATA_LOSS`	The last optimization forced the KCMS framework to remove some data necessary for this operation.
`KCS_CONNECT_PROFILES_CT_ERR`	The operation requires a different number of profiles in the list than supplied.
`KCS_CONNECT_QUANT_MISMATCH`	Mismatch between the quantization of a pair of profiles.
`KCS_CONNECT_UNIMP_OP`	Connect operation is unimplemented.
Validation:	
`KCS_MISMATCHED_WHITEPOINTS`	Profile white points did not match during validation.
`KCS_MISMATCHED_BLACKPOINTS`	Profile black points did not match during validation.
`KCS_MISMATCHED_COLORSPACES`	Profile color spaces did not match during validation.

Table 6-2 Error Codes *(Continued)*

Enumerated Error Constant	Description
`KCS_MISMATCHED_DIMENSIONS`	Profile dimensions did not match during validation.
`KCS_MISMATCHED_VERSIONS`	Profile versions did not match during validation.
Layout:	
`KCS_LAYOUT_INVALID`	Invalid pixel layout.
`KCS_LAYOUT_UNSUPPORTED`	Unsupported pixel layout.
`KCS_LAYOUT_MISMATCH`	Pixel layouts do not match profile input and output specifications.
Evaluation:	
`KCS_EVAL_TOO_MANY_CHANNELS`	More channels specified in the pixel layout structure than the profile supports.
`KCS_EVAL_BUFFER_OVERFLOW`	Caller’s buffer too small.
`KCS_EVAL_ONLY_ONE_OP_ALLOWED`	KcsEvaluate only supports one operation at a time, (KcsForward).
Characterization/Calibration:	
`KCS_CC_UPDATE_NEEDS_MORE_DATA`	Data supplied is inadequate.
`KCS_CC_UPDATE_INVALID_DATA`	Data supplied is invalid.
`KCS_CC_INCORRECT_COLOR_SPACE`	Characterization/calibration data contains incorrect color space.
`KCS_CC_NUM_COMPS_OUT_OF_RANGE`	Characterization/calibration data contains incorrect number of I/O components.
`KCS_CC_TOO_FEW_MEASUREMENTS`	Not enough measurements to support calibrating or characterizing this device.
`KCS_CC_TABLE_DATA_BAD`	Table data is out of range.
`KCS_CC_INCORRECT_DEV_TYPE`	KcsAttributeDevType is incorrect.
`KCS_CC_INCORRECT_ATTR_CLASS`	KcsAttributeClass is incorrect.
`KCS_CC_CANNOT_CAL_DEV_TYPE`	Device type cannot be calibrated.

Table 6-2 Error Codes *(Continued)*

Enumerated Error Constant	Description
`KCS_CC_CANNOT_CHAR_DEV_TYPE`	Device type cannot be characterized.
`KCS_CC_INPUT_NOT_RAMP`	Currently data must be a ramp.
Color Management Module:	
`KCS_CMM_RTLOAD_FAILED`	Runtime loading of CMM failed.
`KCS_CMM_MAJOR_VERSION_MISMATCH`	Incompatible CMM major version number.
`KCS_CMM_MINOR_VERSION_MISMATCH`	Incompatible CMM minor version number.
`KCS_CMM_UNKNOWN_TECHNOLOGY`	CMM requested could not be found.
`KCS_CMM_UNKNOWN_RUNTIME_TYPE`	CMM associated with this profile could not be found.
`KCS_CMM_UNSUPPORTED_OP`	Operation not supported by this CMM.
Unimplemented Features:	
`KCS_UNIMP_NESTED_CONNECTIONS`	Currently, KCMS cannot handle nested connections.
`KCS_UNIMP_TOO_MANY_PROFILES`	Profile array contains too many profiles.
`KCS_UNIMP_ILLEGAL_TECHNOLOGY`	When connecting profiles, one CMM technology is incompatible with another CMM technology. (Very rare with standard ICC profile format.)
Internal:	
`KCS_INTERNAL_CLASS_CORRUPTED`	Internal error related to one of the KCMS classes.
`KCS_INTERNAL_DATA_CORRUPTED`	Internal error related to one of the KCMS data.
IO:	
`KCS_HOSTNAME_ERROR`	Host name unknown (not local or remote).

Localizing Status Messages

The KCMS library warning and error codes are internationalized. You can convert `KcsStatusId` into a text string with the `KcsGetLastError()` function (defined on page 73). Call the appropriate setup functions to convert a message to the appropriate language. A translation table must also exist. The translatable KCMS `.po` files are `kcs_strings.po` and `kcssolmsg_strings.po` located in `/openwin/lib/locale/C/LC_MESSAGES`.

See the following documentation for further information on accessing the translated message file:

- `setlocale`(3c)
- *Solaris Developer's Guide to Internationalization*

Glossary

absorbed light	Light that enters a material and is trapped (neither reflected nor transmitted).
achromatic	Having no hue; white, gray, or black.
adaptation	Process by which the visual mechanism adjusts to the conditions under which the eyes are exposed to radiant energy. See *chromatic adaptation*.
additive color primaries	Red, green, and blue light that produces white light when mixed together in the proper proportions.
ambient lighting	Environmental lighting condition for a particular location.
attribute	A synonym for tag. See *tag*.
bitmap	A digital representation of an image in which all dots or pixels making up the image are rendered in a rectangular grid and correspond to specifically assigned bits in memory.
brightness	Attribute of a visual sensation according to which an area appears to exhibit more or less light.

bit plane	Level of intensity of each electron gun for each primary color in a CRT, controlled by the depth or number of bits describing a pixel. In a simple one-bit monochromatic display, the pixel is either black or white (on or off). In a three-bit image, eight possible colors can be displayed (2^3). This allows eight gray shades in a monochrome display; in a simple three-bit color CRT, the eight colors are red, green, blue, cyan, magenta, yellow, white, and black.
calibration	Procedure for correcting any deviation from a standard.
characterization	Process that defines what colors are produced by (or, when scanning, ought to produce) a given set of numbers by measuring a sample population of devices. Characterization is a description of a device’s color gamut, operation, dynamic range, interaction of colors, color data transfer characteristics, and so forth, which is used as an average operating model for the device.
chroma	Strength of a color, how far it departs from neutral gray.
chromatic	Having a hue; not white, gray, or black.
chromatic adaptation	Adjustment of the visual mechanism in response to the overall color of a stimulus to which the eyes are exposed.
CIE	Commission Internationale de l’Eclairage (International Commission on Illumination), an international organization that establishes and maintains standards of light and color. Its system of describing color is based on standardization of illuminants and observers, not physical samples.
CIEXYZ	Term used when referring to the CIE standard for tristimulus values X, Y, and Z. The system represents all visible colors with positive tristimulus values. Two colors match when their tristimulus values are the same and they are viewed under identical conditions.

CLUT	Color look-up table. An area in computer memory where a set of values is used to index another set of values. Since the table of pixel color information is stored, the information does not have to be recomputed each time it is called up.
CMY/CMYK	Abbreviation for cyan (C), magenta (M), yellow (Y), and black (K) process colors used in printing and other imaging technologies. Cyan, magenta, and yellow are subtractive primaries as well as secondary colors in the additive color system. Black is sometimes added to enhance color and to produce a true black.
CMY/CMYK color space	Color-order model of subtractive primaries cyan (C), magenta (M), yellow (Y), and sometimes black (K), used by printing technologies.
color	Visual sensation that occurs through a combination of physical, physiological, and psychological events involving light, objects, and the visual system.
colorant	A dye, pigment, or ink used in the process of coloring material.
colorimetry	A branch of color science concerned with the measurement and specification of color stimuli.
color laser printer	A printer that uses a laser to xerographically generate the image to be reproduced. Each page is run through the color-application process four times, each time with a different CYMK toner.
color order system	A system used for arranging and describing color, based on physical samples, specific devices, or colorimatric quantities.
color profile	See *device color profile (DCP)*.
***c*alibrator**	A physical device that calibrates the monitor attached to a computer.

color management module (CMM)

That component of a color manager that actually processes color data being input and output to the system in addition to the information about the devices stored in the device color profiles (DCPs).

color space

See *color order system*.

color temperature

A measure that defines the color of a light source relative to the spectral distribution of the light radiated by a theoretically perfect radiator, or black body, heated until it emits visible light. See *correlated color temperature*.

color wheel

Circle with primary colors (red, green and blue) and secondary colors (cyan, magenta, and yellow) located equidistant from each other. A color wheel may also show intermediate hues.

complementary colors

Particular wavelengths of light that, when added together, create white light. The subtractive primaries (cyan, magenta, and yellow) are complementary to the additive primaries (red, green, and blue). For example, blue (an additive primary) and its complementary yellow (a subtractive primary), a secondary color on the additive color wheel, can be added together to produce white light. In the visual arts, complementary colors are diametrically opposite one another on any color wheel.

cones

Visual color-receptor cells of the retina. There are three different types of cone-shaped cells, each thought to have a different photosensitive pigment. Under normal and bright lights, cones produce the sensation necessary for color vision. See *rods*.

contrast

Tonal gradation between the highlights, middle tones and shadows of images.

correlated color temperature

Temperature of a black body (Planckian) radiator whose perceived color most closely matches a given stimulus seen at the same brightness and under specified viewing conditions.

D50	A CIE designation for a white-light spectrum and its associated colorimetric coordinates. It represents a yellower daylight than D65. This is the “daylight” that is specified by the graphics industry for viewing color prints and transparencies. D indicates “daylight” and 5000, the correlated color temperature in degrees Kelvin.
D65	A CIE designation for a white-light spectrum and its associated colorimetric coordinates. It represents a standard daylight for general use. This “daylight” is commonly used in colorimetry, and it is becoming a “standard” for monitor white point. D indicates “daylight” and 6500 the correlated color temperature in degrees Kelvin.
device color profile (DCP)	Device-specific color information for devices.
display	Representation of a data item in visible form, for example, output to a CRT. Visual representation of the output of an electronic device. See *monitor.*
dithering	The technique of making adjacent pixels different colors to give the illusion of an intermediate color. Dithering can produce the effect of shades of gray on a black-and-white display, or simulate a greater number of colors on a color display than the display is capable of producing.
dither cell	Grouping of pixels into a super pixel for the purpose of creating halftones on the computer. Also called *halftone cell.*
dpi (dots per inch)	Measure of resolution level of raster imaging output devices such as laser printers, monitors and photo or laser typesetters (imagesetters).
dynamic range	Extent of minimum and maximum operational characteristics. For example, the difference between lowest and highest intensity (for a monitor), or the lowest and highest density (for prints and transparencies).

electromagnetic radiation

Combination of electrical and magnetic vibrations called *waves* that constitute the electromagnetic spectrum. The human eye sees only a small range of electromagnetic waveforms, or wavelengths, from approximately 400 nm (violet) to 700 nm (red) in the area designated *visible light*.

gamma

For a CRT device, the slope of the line relating the logarithm of the light output to the logarithm of the applied voltage.

gamut

The limits on a set of colors. Ordinarily the gamut is imposed by the limitations of a physical capture, display, or output device. In a computer screen, colors that cannot be displayed are called *out-of-gamut colors*.

gamut adjustment

Ability to account for device capabilities and limitations by regulating colors through compression or expansion techniques. In *gamut compression*, colors that are beyond the capabilities of a device are mapped into colors that the device can actually produce.

halftone

A color or black-and-white continuous tone image reproduced by changing the image into dots through the use of halftone screens. Because printing presses are not able to print true continuous tone images, a halftone allows tone gradation, in which the dots are perceived as a whole, depending on the halftone screen used, quality of the original image, and so forth. In computers, electronic algorithms can create digital halftone representations.

hue

Attribute of a visual sensation according to which an area appears to be similar to one, or to proportions of two, of the visible colors, red, yellow, green, cyan, blue, and magenta. Hue is part of the HSV (hue, saturation, and value) and HLS (hue, lightness, and saturation) color models.

ICC

International Color Consortium.

illuminant

A light defined by its spectral power distribution. An illuminant may or may not be physically realizable as a source. Several standard illuminants have been defined by the CIE for use in colorimetric computations. See *source*.

ink-jet printer	A printer that uses finely directed sprays of ink to produce the character image. Color printout is achieved in one pass and colors are based on the CMYK or CYM color model. Technologies for this category of color output printers include *drop-on-demand*, which can be subdivided into *bubble jet* (or *thermal ink-jet*) and *piezoelectric; continuous ink-jet;* and *phase-change ink jet*. Phase-change ink jet technique requires solid ink while the others take liquid ink.
light source	See *illuminant* and *source*.
memory colors	Colors seen regularly that people tend to remember best and agree on the appearance of, such as green grass and blue sky.
metameric colors	A pair of colors that match visually under some lighting conditions, but not under others.
metamerism	Visual phenomenon where the colors of two spectrally different objects appear to match under a specific set of conditions. The term *observer metamerism* is used when two objects appear to some observers (or instruments) to have the same color, but to other observers the same objects do not match.
moiré	In printing, undesirable patterns caused by misalignment of halftone dots. In imaging devices: visual patterns formed by interference between two sets of regular divisions, such as the combination of a TV raster with a striped object in the scene; can be caused by any beating between frequencies.
monitor	Device for computer generated display; video display terminal.
monitor calibration	Process that measures the performance of a display and compensates for its variations.
monitor RGB	See *RGB color space*.

monitor white point	Color specification of a monitor’s white, when all three phosphors are lit to maximum level.
Munsell chroma	The quality that describes the extent to which a color differs from a gray of the same value.
Munsell hue	The quality of color described by the words red, yellow, blue, and so forth. The principal hues of the Munsell system are red, yellow, green, blue and purple.
Munsell system	A color-order system established by A.H. Munsell in 1905. Based on visual perception, this system provides a description of a color, using a collection of samples as well as a color notation system. See *Munsell chroma, Munsell hue,* and *Munsell value.*
Munsell value	The quality of a color described by the words light, dark, and so forth, relating the color to a gray of similar lightness.
nanometer	Preferred nomenclature for describing measurement of wavelengths of light. One nanometer equals $1x10^{-6}$ millimeter. The abbreviation is nm.
observer metamerism	See *metamerism*.
palette	The set of colors (ranging from four to more than 16 million) that a particular computer graphics program is using. Many display adapters have a limited palette. The set of colors may be in a table.
peripherals	The devices that hook up to the desktop computer (color monitor, printer, scanner, and so forth).
phosphor	The phosphorescent coating on the interior of the front surface of a cathode ray tube (CRT) that emits light of one of the three additive primary colors (red, green, or blue) when a carefully controlled beam of electrons strikes the material. Depending on the type of color tube, the pattern of the phosphors can be dot, brick-like, or stripe.

Photo CD	A photographic compact disc (CD) made using a Kodak imaging system. The system scans in photographic images (negatives, slides, and prints), processes the data to optimize its quality for digital imaging, compresses the data, and then writes it on a compact disk.
pigment	Finely ground, natural or synthetic, inorganic or organic, insoluble particles (powder) that, when dispersed in a liquid vehicle, give color to paints, printing inks, and other materials by reflecting and absorbing light.
pixel	P*icture element*. Smallest addressable point of a bitmapped screen that can be independently assigned color and intensity.
pixel depth	Number of bits describing a pixel. Syn. *bit depth.* See *bitplane.*
PMS (Pantone Matching System)	A printing industry standard for specifying spot color.
pre-press	Term used to describe the process or components of the process of preparing information for printing or alternative media output after the writing and design concept stages. In desktop publishing, it is the process of all of the elements on any page to produce the master copy.
primary colors	Three basic colors used to make other colors by mixture, either additive mixture of lights or subtractive mixture of colorants. The additive primaries are red, green, and blue; the subtractive primaries are cyan, magenta, and yellow. See *additive color primaries, subtractive primaries,* and *secondary color.*
printer	Computer-driven device that deposits images on paper or film. See *ink-jet printer, thermal wax printer* and *color laser printer.*
process colors	Cyan, magenta, yellow, and black used in color printing. See *CMY/CMYK.*
profile connection space	The common junction where profiles for different devices are connected together.

reflected light	Light that bounces back from the object that it strikes.
registration	In printing: accuracy with which printing images are positioned or combined so that they align exactly. In multi-color printing each color must be precisely aligned one over the other for accurate reproduction. In color monitors: alignment of the electron guns to produce correct color.
resolution	The degree of sharpness of an image displayed on a computer screen, or quality of printed output from a laser printer or photo or laser typesetter; expressed in dots per inch (dpi). Resolution can also refer to the number of bits per pixel. In printing, resolution refers to the space between dots in a halftone screen; expressed as lines per inch (lpi).
RGB	Abbreviation for red, green and blue primaries of the additive color system. Used in reference to color computer graphics and video technology.
RGB color space	A color-order model that may be based on either the light-emitting phosphors (red, green, and blue) of an actual device or on a set of hypothetical RGB primaries.
rods	Photoreceptor cells in the retina that respond to low levels of light. They are not thought to contribute to color vision. See *cones*.
saturation	The amount of hue in a color sample compared to the amount of achromatic light it reflects or transmits.
scanner	An electronic device that digitizes and converts photographs, slides, paper images, or other two-dimensional images into bitmapped images.
scanner calibration	A feature that measures the performance of a scanner and compensates for its variations.

secondary color	Color made by mixing two primary colors. In the additive color system, the secondary colors are cyan, magenta, and yellow; in the subtractive color system, the secondary colors are red, green, and blue.
service bureau	A company that provides pre-press and other computer output in a variety of forms, such as film separations, slides and other transparencies, and color proofs. A service bureau may specialize or can be a full-service operation that offers a wide range of services, including printing.
simulation	Used to represent an image on a display. It is a feature that changes the display colors to match the input or output colors in a way that corresponds to a defined device, medium, viewing environment, and so forth.
source	A physically realizable light, whose spectral power distribution can be experimentally determined. Several standard sources have been defined by the CIE for use in colorimetry. Also a computer term for *origin* of data.
spectral response	Using the example of the human eye, the spectral response curves map the wavelength of light against the fraction of light absorbed by each type of eye cone (red, green, and blue sensitive cones). It is the sensitivity of the eye or a device to different wavelengths of light.
spot color	Color printed in pure color (ink straight out of the container), as opposed to four-color process, where colors are composed of percentages of cyan, magenta, yellow, and black. Spot color separations for printing involve one plate for each color on the page, unlike process color, which requires four separate plates.
standard illuminant	See *illuminant*.
standard observer	The CIE specification for a hypothetical observer whose spectral responsivities represent those of the average human population with normal color vision.
standard source	See *source*.

subtractive primaries	Cyan, magenta, and yellow. The three colors that, when superimposed in register, produce black. Also known as *process colors* because cyan, magenta, and yellow are used in printing. See *CMY/CMYK*.
surround effect	A perceptual phenomenon where the appearance of a color is influenced by the color or colors surrounding it.
system monitor	The monitor that is physically attached to a computer system to be used when displaying images.
tag	Attribute of a color profile that provides information for a CMM to translate color information between the profile connection space and the native device space. Tags are specified by name, value, and status (required or optional).
target	A physical paper target with a reference image used for determining the color response of a scanner.
thermal dye transfer printer	A type of thermal-transfer printer that produces a high resolution continuous tone image. This technology mixes percentages of cyan, magenta, and yellow, and adjusts the density of each printed dot, thereby eliminating the need for halftoning and dithering to produce different colors. Specially coated paper reacts with the dye causing the dye to diffuse into the paper. Also referred to as *dye-diffusion printer, dye-sublimation printer,* and *sublimal-dye printer.*
thermal wax printer	A printer that uses colored wax or plastic, dye, dyed ribbons, or some other material that can be heat-flowed onto paper or transparency film. Other names for this category: *thermal-transfer printer* and *thermal-wax transfer* printer.
transmitted light	Light that passes through an object.
transparency	Image formed on a clear or translucent base by means of a photographic, printing, chemical, or other process, generally viewed by transmitting light through the image.

tristimulus values	Intensities or amounts of each of a set of three primary colors required to match a given color stimulus. See *CIEXYZ*.
value	See *Munsell value.*
visible spectrum	The portion of electromagnetic radiation, from approximately 400 nm to 700 nm, that is seen as visible light. The colors of the spectrum from 400 to 700 nm are violet, blue, green, yellow, orange, and red.
wavelength	Distance between successive corresponding points in electromagnetic and other forms of waves. See *nanometer.*
white point	See *monitor white point*.
XYZ	See *CIEXYZ*.

Bibliography

Hunt, R. W. G, Dr. *The Reproductions of Colour In Photography, Printing and Television*. 4th ed. England: Fountain Press, 1987.

Verbum Magazine. *The Desktop Color Book*. California: Verbum, Inc., 1992.

Wyszecki, Gunter and W. S. Stiles. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. 2nd ed. New York: John Wiley & Sons, Inc., 1982.

Index

Numerics

icInt64Number, 109
icUInt64Number, 109

A

abstract profile, 106
accuracy, optimizing for, 81
architecture, 1 to 2
architecture diagram, 2
attribute, 86
  data structures, 99 to 133
    arrays of numbers, 117
    ASCII data, 108
    ASCII data, variable-length array, 109
    binary data, 108
    binary data, variable-length array, 109
    device attributes, 108
    enums, other, 114
    number definitions, 109
    profile header flags, 108
    screening encodings, 108
    signatures, 109
    signatures, color space, 113
    signatures, color space valid PCSs note, 113
  error messages, 137
  Lut8 and Lut16 transform note, 102, 121, 127
  required and optional, 100
  value, 100

B

band-interleaved data, 50
bibliography, 157

C

calibration, 93
calibration, definition of
  See also profile, 20
CCP (complete color profile), definition of, 10
characterization, 93
  error messages, 139

characterization, definition of
  See also profile, 20
chromaticity, 7
CIE (Commission Internationale de l'Eclairage), 10
CMM (Color Management Module)
  error messages, 140
CMM (color management module)
  in KCMS product overview, 4
  profile, assocation with, 8
CMYK input profile (ICC), 104
CMYK output profile (ICC), 105
color blindness, 10
color profiles (See profiles)
color space conversion profile, 105
color spaces, 7
color-corrected, 3
colorimetric data, 44
colormap, 50
component array defines, 51
component-interleaved data, 49
computer-generated color data, 44
constants, 24
  operation hint, 43
content hints (See hints), 18
CSP (color space profile), definition of, 9

D

DCP (device color profile), definition of, 9
demonstration programs, 6
device attributes, 108
device link profile, 105

E

ECP (effects color profile), definition of, 10
error format, 58
error messages, 136 to 140
  attributes, 137
  characterization, 139
  CMM, 140
  connection, 138
  evaluation, 139
  general failure, 136
  internal, 140
  IO, 136
  memory, 136
  pixel layout, 139
  profile, 137
  unimplemented features, 140
  validation, 138
  X11 profile, 137

F

forward operation hints (See hints), 18

H

hints
  content, 18, 44
  load, 19, 74, 79
    bit mask code example, 43
    bit mask values table, 42
    bit positions and masks, 41
  operation
    forward, 43
    reverse, 43
  operation, forward, 15, 18
  operation, gamut-test, 16

  - operation, reverse, 15, 18
  - operation, simulate, 16
- hints, load, 11

I

- icAny, 109
- icAsciiData, 108
- icBinaryData, 108
- ICC content hints, 45
- ICC specification
  - device link profiles, 105
  - input profile, 103
    - CMYK, 104
    - monochrome, 104
    - RGB, 104
  - output profile, 104
    - CMYK, 105
    - monochrome, 105
    - RGB, 105
- ICC tag, See tag, 102
- icColorSpaceSignature, 113
- icCurve, 120
- icCurveType, 126
- icData, 121
- icDataType, 126
- icDateTimeNumber, 120
- icDateTimeType, 126
- icDescStruct, 122
- icEmbeddedProfileFalse, 108
- icEmbeddedProfileTrue, 108
- icGlossy, 108
- icHeader, 132
- icIlluminant, 116
- icInt16Array, 118
- icInt16Number, 109
- icInt32Array, 119
- icInt32Number, 109
- icInt64Number, 119
- icInt8Number, 109, 118
- icLinesPerCm, 108
- icLinesPerInch, 108
- icLut16Type, 126
- icLut8Type, 127
- icMagicNumber, 108
- icMatte, 108
- icMeasurement, 122
- icMeasurementFlare, 114
- icMeasurementGeometry, 114
- icMeasurementType, 128
- icNamedColorType, 128
- icPlatformSignature, 114
- icProfile, 133
- icProfileClassSignature, 113
- icProfileSequenceDesc, 122
- icProfileSequenceType, 128
- icPrtrDefaultScreensFalse, 108
- icPrtrDefaultScreensTrue, 108
- icReflective, 108
- icRenderingIntent, 115
- icS15Fixed16ArrayType, 128
- icS15Fixed16Number, 119
- icScreening, 123
- icScreeningData, 123
- icScreeningType, 129
- icSigHeaderTag, 100
- icSigLabData, 113

icSigListTag, 100
icSignature, 109
icSignatureType, 129
icSigNumTag, 100
icSigXYZData, 113
icSpotShape, 115
icStandardObserver, 115
icTag, 132
icTagBase, 126
icTagList, 132
icTagSignature, 110
icTagTypeSignature, 112
icTechnologySignature, 111
icText, 124
icTextDescription, 123
icTextDescriptionType, 128
icTextType, 129
icTransparency, 108
icU16Fixed16ArrayType, 129
icU16Fixed16Number, 119
icUcrBg, 124
icUcrBgCurve, 124
icUcrBgType, 129
icUInt16ArrayType, 130
icUInt16Number, 109, 118
icUInt32ArrayType, 130
icUInt32Number, 109, 118
icUInt64ArrayType, 130
icUInt64Number, 119
icUInt8ArrayType, 130
icUInt8Number, 109, 118
icUseAnywhere, 108
icUseWithEmbeddedDataOnly, 108
icVersionNumber, 108
icViewingCondition, 124
icViewingConditionType, 130
icXYZArray, 120
icXYZNumber, 120
icXYZType, 131
interleaved data
    band, 50
    component, 49
        pixel layout diagram, 52
    planar, 50
    row, 49

K

kcms, 6
KCMS product overview, 1 to 6
    applications, 2
    architecture, 1
    architecture diagram, 2
    C API, 2
    CMM, 4
    KCMS file system, 5
    KCMS framework, 3
    libraries, graphics and imaging, 4
    profile, 3
kcms_create.c, 6
kcms_timer.c, 6
kcms_update.c, 6
kcms_utils.c, 6
kcmstest, 6
kcmstest_tiff.c, 6
KcsAddToCurrentHints, 42
KcsAllFunc, 39

KcsAttributeBase, declaration of, 24
KcsAttributeName
  in KcsGetAttribute(), 69
  in KcsSetAttribute(), 86
KcsAttributes, 42
KcsAttributeType, declaration of, 26
KcsAttributeValue
  in KcsGetAttribute(), 69
  in KcsSetAttribute(), 86
KcsAttributeValue, declaration of, 28
KcsAttrStrLength, declaration of, 24
KcsAvailable()
  declaration, 60
  use of, 60
KcsCalibrationData
  in KcsUpdateProfile(), 93
KcsCallbackFunction
  in KcsSetCallback(), 91
KcsCallbackFunction, declaration of, 31
KcsCharacterizationData
  in KcsUpdateProfile(), 93
KcsComponent, declaration of, 35
KcsConnectFunc, 39
KcsConnectProfiles()
  declaration, 61
  use of, 14, 16, 20, 48, 61
KcsContAll, 42, 45, 48
KcsContColorimetric, 42, 44, 48
KcsContGraphics, 42, 44, 48
KcsContImage, 42, 44, 48
KcsContUnknown, 42, 45, 48
KcsCreateProfile()
  use of, 20, 36, 64
KcsCreationDesc, declaration of, 36
KcsCreationType, declaration of, 37
KcsEffect, 42
KcsErrDesc, declaration of, 37
KcsEvaluate()
  declaration, 66
  use of, 15, 18, 19, 31, 48, 51, 66
KcsEvaluateFunc, 39
KcsEvaluationSpeed, declaration of, 38
KcsExtendableArray, declaration of, 24
KcsExtendableMeasSet, declaration of, 24
KcsExtendablePixelLayout, declaration of, 24
KcsFileId, declaration of, 38
KcsFileProfile, 55
KcsFreeFunc, 39
KcsFreeProfile()
  declaration, 68
  use of, 19, 68
KcsFunction
  in KcsSetCallback(), 91
KcsFunction, declaration of, 38
KcsGetAttribFunc, 39
KcsGetAttribute()
  declaration, 69
  get CMM list note, 64
  use of, 10, 25, 44, 69
KcsGetLastError()
  declaration, 73
KcsHeapApp, 42
KcsHeapSys, 42
KcsIdentifier, 39
KcsLoadFunc, 39
KcsLoadHints, 40
  in KcsLoadProfile(), 74
  in KcsModifyLoadHints(), 79

  in KcsOptimizeProfile(), 81
  use of, 11
KcsLoadHints, bit mask code example, 43
KcsLoadHints, bit mask values table, 42
KcsLoadHints, bit positions and masks diagram, 41
KcsLoadNever, 42
KcsLoadNow, 42
KcsLoadNow, use of, 38
KcsLoadProfile()
  declaration, 74
  memory management, 74
  use of, 11, 20, 74
KcsLoadWhenIdle, 42
KcsLoadWhenNeeded, 42
KcsMaskAttr, 42
KcsMaskCont, 42
KcsMaskEffect, 42
KcsMaskLoadWhen, 42
KcsMaskLoadWhere, 42
KcsMaskLogical, 42
KcsMaskOp, 42
KcsMaskUnloadWhen, 42
KcsMeasurementBase, 46
KcsMeasurementSample, 46
KcsMemoryProfile, 55
KcsModifyLoadHints()
  declaration, 79
  use of, 11, 79
KcsModifyLoadHintsFunc, 39
KcsOpAll, 42, 44, 48
KcsOperationType, 40, 47
  in KcsConnectProfiles(), 61
  in KcsEvaluate(), 66
KcsOpForward, 42, 43, 48
KcsOpGamutTest, 42, 44, 48
KcsOpReverse, 42, 43, 48
KcsOpSimulate, 42, 48
  use of, 43
KcsOpSimulate, preview printer output note, 44
KcsOptAccuracy, 49
KcsOptimizationType, 48
  in KcsOptimizeProfile(), 81
KcsOptimizeFunc, 39
KcsOptimizeProfile()
  declaration, 81
  use of, 19, 20, 31, 48, 81
KcsOptNone, 49
KcsOptSize, 49
KcsOptSpeed, 49
KcsPixelLayout, 49 to 52
  component array defines, 51
  component interleaved data, pixel layout diagram, 52
  in KcsEvaluate(), 66
KcsPixelLayoutSpeeds, 53
KcsProfileDesc, 54
  in KcsLoadProfile(), 74
  in KcsSaveProfile(), 83
  use of, 20
KcsProfileId, 56
  in KcsConnectProfiles(), 61
  in KcsEvaluate(), 66
  in KcsFreeProfile(), 68
  in KcsGetAttribute(), 69
  in KcsLoadProfile(), 74
  in KcsModifyLoadHints(), 79

  - in KcsOptimizeProfile(), 81
  - in KcsSaveProfile(), 83
  - in KcsSetAttribute(), 86
  - in KcsUpdateProfile(), 93
- KcsProfileType, 56
- KcsSampleType, 57
- KcsSampleType constants, 57
- KcsSaveFunc, 39
- KcsSaveProfile()
  - declaration, 83
  - use of, 11, 16, 20, 83
- KcsSetAttribFunc, 39
- KcsSetAttribute()
  - declaration, 86
  - use of, 10, 20, 25, 86
- KcsSetCallback()
  - declaration, 91
  - use of, 16, 31, 91
- KcsSolarisFile, 55
- KcsStartOverWithThis, 42
- KcsStatusId, 58
- kcstest, 6
- kcstest.c, 6
- KcsUnloadAfterUse, 42
- KcsUnloadNow, 42
- KcsUnloadWhenFreed, 42
- KcsUnloadWhenNeeded, 42
- KcsUpdateProfile(), 93 to 98
  - declaration, 93
  - use of, 20, 47

L

- libraries
  - graphics and imaging, 4
- lighting conditions, 10, 16
- linearization tables, 94
- load hints (See hints), 11
- Localizing Status Messages, 141

M

- macro
  - KCS_DEFAULT_ATTRIB_COUNT, 25
- macros, 23
- monitors
  - effect of lighting on, 16
- monochrome input profile (ICC), 104
- monochrome output profile (ICC), 105

N

- naming conventions used, xviii

O

- OpForward, 15
- OpGamutTest, 16
- OpReverse, 15
- OpSimulate, 16

P

- palette color data, 50
- photographic input data, 44
- pixel layout
  - error messages, 139
- planar data, 50
- print, 6
- print_attributes.c, 6
- print_header.c, 6
- print_montbls.c, 6

profile, 7 to 21
  abstract, 106
  assocation with CMMs, 8
  calibration, definition of, 20
  CCP, 16
    creating, 14
  CCP code example, 17
  CCP, definition of, 10
  characterization, definition of, 20
  color space conversion, 105
  connecting, 14
  converting data diagram, 12
  CSP, 16
  CSP, definition of, 9
  DCP, 16
  DCP, definition of, 9
  description, 11
  device link, 105
  devices, associating with, 16
  devices, associating with diagram, 12
  ECP, definition of, 10
  error messages, 137
  evaluating, 15
  freeing, 19
  header flags, 108
  identifier, 11
  in KCMS product overview, 3
  input
    CMYK (ICC), 104
    monochrome (ICC), 104
    RGB (ICC), 104
  loading, 11
  memory management, 19
  monitor, converting to, 12
  operations diagram, 15
  optimizing, 19, 81
    accuracy, 19, 49
    callback function, 19
    size, 19, 49
    speed, 19, 49
  output
    CMYK (ICC), 105
    monochrome (ICC), 105
    RGB (ICC), 105
  saving, 11
  scanner, converting from, 12
  simple color data conversion code example, 13
  simulated execution, 43
  using to convert color data, 12 to 17

R

README, 6
readme file, 6
rendering hints, 45
reverse operation hints (See hints), 18
RGB input profile (ICC), 104
RGB output profile (ICC), 105
row-interleaved data, 49

S

sample programs, 6
screening encodings, 108
signatures (ICC), 109
size, optimizing for, 81
speed, optimizing for, 81

T

tag
  definition of all, 107

  name, 99
  required, 102
  types, 107 to 124

V

visual impairment, 10

W

warning messages, 135

Copyright 1995 Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, Californie 94043-1100 U.S.A.

Copyright Eastman Kodak Company, 1994. Modifié par avec permission de Kodak.

Tous droits réservés. Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l’utilisation, la copie, et la décompliation. Aucune partie de ce produit ou de sa documentation associée ne peuvent Être reproduits sous aucune forme, par quelque moyen que ce soit sans l’autorisation préalable et écrite de Sun et de ses bailleurs de licence, s’il en a.

Des parties de ce produit pourront etre derivees du système UNIX®, licencié par UNIX System Laboratories, Inc., filiale entierement detenue par Novell, Inc., ainsi que par le système 4.3. de Berkeley, licencié par l’Université de Californie. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fourmisseurs de Sun.

LEGENDE RELATIVE AUX DROITS RESTREINTS: l’utilisation, la duplication ou la divulgation par l’administration americaine sont soumises aux restrictions visées a l’alinéa (c)(1)(ii) de la clause relative aux droits des données techniques et aux logiciels informatiques du DFARS 252.227-7013 et FAR 52.227-19. Le produit décrit dans ce manuel peut Être protege par un ou plusieurs brevet(s) americain(s), etranger(s) ou par des demandes en cours d’enregistrement.

MARQUES

Sun, Sun Microsystems, le logo Sun, SunSoft, le logo SunSoft, Solaris, SunOS, OpenWindows, DeskSet, ONC, ONC+, KCMS et NFS sont des marques deposées ou enregistrées par Sun Microsystems, Inc. aux Etats-Unis et dans d’autres pays. UNIX est une marque enregistrée aux Etats- Unis et dans d’autres pays, et exclusivement licenciée par X/Open Company Ltd. OPEN LOOK est une marque enregistrée de Novell, Inc. PostScript et Display PostScript sont des marques d’Adobe Systems, Inc.

Toutes les marques SPARC sont des marques deposées ou enregitrées de SPARC International, Inc. aux Etats-Unis et dans d’autres pays. SPARCcenter, SPARCcluster, SPARCompiler, SPARCdesign, SPARC811, SPARCengine, SPARCprinter, SPARCserver, SPARCstation, SPARCstorage, SPARCworks, microSPARC, microSPARC-II, et UltraSPARC sont exclusivement licenciées a Sun Microsystems, Inc. Les produits portant les marques sont basés sur une architecture développée par Sun Microsystems, Inc.

Les utilisateurs d’interfaces graphiques OPEN LOOK® et Sun™ ont été développés par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnait les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d’utilisation visuelle ou graphique pour l’industrie de l’informatique. Sun détient une licence non exclusive de Xerox sur l’interface d’utilisation graphique, cette licence couvrant aussi les licenciés de Sun qui mettent en place OPEN LOOK GUIs et qui en outre se conforment aux licences écrites de Sun.

Le système X Window est un produit du X Consortium, Inc. Kodak est une marque de Eastman Kodak Company.

CETTE PUBLICATION EST FOURNIE “EN L’ETAT” SANS GARANTIE D’AUCUNE SORTE, NI EXPRESSE NI IMPLICITE, Y COMPRIS, ET SANS QUE CETTE LISTE NE SOIT LIMITATIVE, DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L’APTITUDE DES PRODUITS A REPONDRE A UNE UTILISATION PARTICULIERE OU LE FAIT QU’ILS NE SOIENT PAS CONTREFAISANTS DE PRODUITS DE TIERS.

CETTE PUBLICATION PEUT CONTENIR DES MENTIONS TECHNIQUES ERRONEES OU DES ERREURS TYPOGRAPHIQUES. DES CHANGEMENTS SONT PERIODIQUEMENT APPORTES AUX INFORMATIONS CONTENUES AUX PRESENTES. CES CHANGEMENTS SERONT INCORPORES AUX NOUVELLES EDITIONS DE LA PUBLICATION. SUN MICROSYSTEMS INC. PEUT REALISER DES AMELIORATIONS ET/OU DES CHANGEMENTS DANS LE(S) PRODUIT(S) ET/OU LE(S) PROGRAMME(S) DECRITS DANS DETTE PUBLICATION A TOUS MOMENTS.

Adobe PostScript