KCMS CMM Developer's Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 805-3933–10
October 1998

Copyright 1998 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303-4900 U.S.A. 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. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, SunDocs, Java, the Java Coffee Cup logo, and Solaris are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was 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.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227–14(g)(2)(6/87) and FAR 52.227–19(6/87), or DFAR 252.227–7015(b)(6/95) and DFAR 227.7202–3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 1998 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, Californie 94303-4900 Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite 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 y en a. 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 fournisseurs de Sun.

Des parties de ce produit pourront être dérivées du système Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, SunDocs, Java, le logo Java Coffee Cup, et Solaris sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît 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 Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

Please
Recycle

Contents

Preface ix

New Features xv

1. Class Descriptions 1

In This Chapter 1

KCMS Class Hierarchy 2

`KcsShareable` Class 3

`KcsLoadable` Class 3

UIDs and Sharing 4

Example 5

Derivatives 5

`KcsIO` Class 6

`KcsFile` Class 6

`KcsMemoryBlock` Class 7

`KcsSolarisFile` Class 7

`KcsXWindow` Class 7

`KcsChunkSet` Class 7

`KcsProfile` Class 9

`KcsProfileFormat` Class 10

`KcsAttributeSet` Class 10

Using a KcsAttributeSet Object 11
KcsXform Class 12
KcsXformSeq Class 13
KcsStatus Class 13
KcsSwapObj Class 13
2. CMM : A Runtime Derivative 15
In This Chapter 15
Development Environment Requirements 16
Requirements For Creating a CMM 16
Why You Might Derive From or Extend a KCMS Class 17
KcsIO 18
KcsProfile 18
KcsProfileFormat 18
KcsXform 19
KcsStatus 19
Deriving Classes at Runtime 19
Runtime Derivation Coding Requirements 20
Runtime Derivation Code Examples 20
Wrapper Functions 20
External Entry Points 21
Instantiation 23
Initialization and Cleanup 24
Configuration Requirements 25
CMM Filename Convention 25
CMM Makefile 26
Creating OWconfig File Entries 27
Updating the OWconfig File 31
Version Numbering 33

Profiles 33

ICC Profile Header 34

Naming and Installing Profiles 35

3. KCMS Framework Operations 39

In This Chapter 39

KCMS Framework Architecture 39

`KcsProfile` 40

`KcsProfileFormat` 41

`KcsAttributeSet` 41

`KcsXform` 42

KCMS Framework Flow Examples 42

KCMS Framework Primary Operations 44

Loading a Profile From the Solaris File System 44

Loading an X11 Window System Profile 49

Connecting Two Loaded Profiles 50

Evaluating Data Without Optimization 51

Evaluating Data With Optimization 52

Freeing a Profile 52

Attributes 52

Characterization and Calibration 54

Saving a Profile to the Same Description 56

Saving a Profile to a Different Description 57

4. `KcsIO` Derivative 59

In This Chapter 59

External Entry Points 60

Mandatory 60

Optional 60

Example 60

Member Function Override Rules 61
Examples To Help You Create Your KcsIO Derivitave 62
5. `KcsProfile` **Derivative 65**
In This Chapter 65
External Entry Points 66
Mandatory 66
Optional 66
Example 66
Member Function Override Rules 67
Attribute Sets 69
`KcsProfileFormat` Instance 70
Transformations 70
Constructors and Destructors 72
Creation Methods 73
Save Methods 73
Using `connect()()` 74
Examples 76
Characterization and Calibration 76
6. `KcsProfileFormat` **Derivative 79**
In This Chapter 79
External Entry Points 80
Mandatory 80
Optional 80
Examples 81
Member Function Override Rules 82
Attributes 83
Transformations 84
Loading 84

Error Protocols 84
Protected Derivatives 85
Base Class Support 85
Retrievable Objects 86
7. `KcsXform` **Derivative 87**
In This Chapter 87
External Entry Points 88
Mandatory 88
Optional 88
Example 88
Member Function Override Rules 89
Technology 91
`KcsXform` Attributes 91
Optimization 92
Loading 92
Save Types 93
Universal 93
Private 94
Example 94
Composition 95
Evaluation 95
Evaluation Helper Methods 96
`KcsXformSeq` Derivatives 97
Constructs and Destructors 97
Saving 97
Loading and Constructing the List 98
Connections 98
Optimization 98

Composition 98

Evaluation 99

Validation 99

The List 99

8. `KcsStatus` Extension 101

Example 102

Header File 102

Localizing Messages 103

Application Module 103

Developer 104

A. Supported Devices 105

Supported Devices 105

Preface

The *KCMS CMM Developer's Guide* describes how to create a Kodak Color Management System (KCMS™) color management module (CMM). It provides information on how to use the KCMS foundation library, which is a graphics porting interface (GPI) implemented in C++. These C++ interfaces link the device-independent layer of the KCMS library with the CMM and enable the flow of data from the application to the CMM.

Use this manual with the *KCMS CMM Reference Manual*, which provides detailed information on all C++ classes in the KCMS foundation library.

Who Should Use This Guide

Use this guide if you are a C++ programmer interested in:

- Writing your own color management module (CMM)
- Creating your own profile format
- Adding attributes or tags to the ICC profile format
- Overriding various class methods

Before You Read This Guide

Check all of the following for any KCMS-specific or system release-specific information that you might need:

- You should be familiar with the Kodak Color Management System (KCMS) API, which is part of the Software Developer's Kit (SDK). See the following manual:
  - *KCMS Application Developer's Guide*
- You should also have an understanding of C++ and Solaris™ dynamic loading technology. Solaris dynamic loading is discussed in the *Linker and Libraries Guide*and in the following manual pages:
  - `ld()`(1)
  - dlopen(3)
  - `dlclose()`(3)
  - `dlerror()`(3)
  - `dlsym()`(3)
- A basic understanding of color science is also assumed. Color science references are included in the Bibliography of the *KCMS Application Developer's 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 7 (SPARC Platform Edition) Installation Library* and the *Solaris 7 (Intel Platform Edition) Installation Library* manuals.
- For SPARC systems, consult the updates your hardware manufacturer may have provided.

How This Guide Is Organized

Chapter 1 briefly describes each of the relevant classes in the KCMS CMM class hierarchy.

Chapter 2 describes how to create a CMM that is a runtime derivative. It also discusses each of the KCMS classes from which you can derive or extend.

Chapter 3 provides examples of how some of the C++ methods interface with the KCMS framework API.

Chapter 4 describes how to derive from the `KcsIO` base class.

Chapter 5 describes how to derive from the `KcsProfile` base class.

Chapter 6 describes how to derive from the `KcsProfileFormat` base class.

Chapter 7 describes how to derive from the `KcsXform` base class.

Chapter 8 describes how to extend the `KcsStatus` base class.

Appendix A describes how to name and install your own profile.

Related Books

The following is a list of recommended books that can help you accomplish the tasks described in this guide:

- *KCMS CMM Reference Manual* (part of DDK)
- *KCMS Application Developer's Guide*
- *KCMS Calibrator Tool Loadable Interface Guide*(part of SDK)
- *ICC Profile Format Specification* (located on-line in `/usr/openwin/demo/kcms/docs/icc.ps`). For the most current version of the ICC specification, see the web site at `http://www.color.org`.

Ordering Sun Documents

The SunDocs[SM] program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress[TM] Internet site at `http://www.sun.com/sunexpress`.

Note - The term "x86" refers to the Intel 8086 family of microprocessor chips, including Pentium and Pentium Pro processors and compatible microprocessor chips made by AMD and Cyrix. In this document, the term "x86" refers to the overall platform architecture, whereas "*Intel Platform Edition*" appears in the product name.

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this guide:

TABLE P–1 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. `system% You have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`system%`
		`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.
Code samples are included in boxes and may display the following:		
`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`

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–2 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	`#`

Equivalent Terms In This Guide

For historic reasons, this guide uses several equivalent Kodak and ICC terms. The terms evolved at different times. Development of the ICC specification introduced new ICC terms with meanings the same as (or similar to) already existing Kodak terms.

You should be familiar with the terms listed in the table below, as you will encounter them in the ICC specification and KCMS color management documentation, as well as in the KCMS header files and example programs. The terms are defined as they are introduced in this guide.

TABLE P–3 Equivalent ICC and Kodak Terms

Kodak Term	ICC Term
attribute	tag
device color profile (DCP)	input, display, or output profile
effects color profile (ECP)	abstract profile
complete color profile (CCP)	device link profile
profile format Id or magic number	profile file signature
reference color space (RCS)	profile connection space (PCS)

Note - The text in this guide uses the term *attribute* instead of *tag*, (but code examples and header files may use *tag* for the historic reasons previously mentioned.

New Features

The following information is about features provided in this release of the KCMS product.

KCMS is Multithread Safe

In this release, KCMS supports multithreaded programs.

`OWconfig` File Modification

The procedure for updating the `OWconfig` file has changed. Using the interactive program called `OWconfig_sample`, you can insert and remove configuration entries in the `OWconfig` file.

CHAPTER 1

Class Descriptions

In This Chapter

This chapter introduces you to the KCMS framework classes. The chapter assumes your familiarity with the KCMS architecture. See Figure 1–1 .

As a brief review, the KCMS architecture consists of components supplied by SunSoft (shown in gray in the figure) and other components (shown in white) that the developer can add. Towards the top of the figure, you see the KCMS "C" application programming interface (API). It consists of a group of functions that allow an application to communicate with the KCMS framework to manipulate profiles in a device-independent manner. The KCMS "C"API is described in detail in the SDK manual *KCMS Application Developer's Guide*. This manual assumes that you are familiar with the API functions.

The color management module (CMM), shown below the KCMS framework, is the component that ultimately does the color management. The default CMM includes all the classes described in this chapter.

Other CMMs use different techniques for evaluating color data, which can result in differences in quality, profile size, and speed of color manipulations. To add a new third-party CMM to the framework, you can only derive from four of the classes described in this chapter and you can extend one. Collectively, these classes are referred to as the C++ CMM interface in the figure. Although you can only derive from a few classes, you need to understand the inner workings of all the classes to understand what is being inherited in the implementation of the CMM you write.

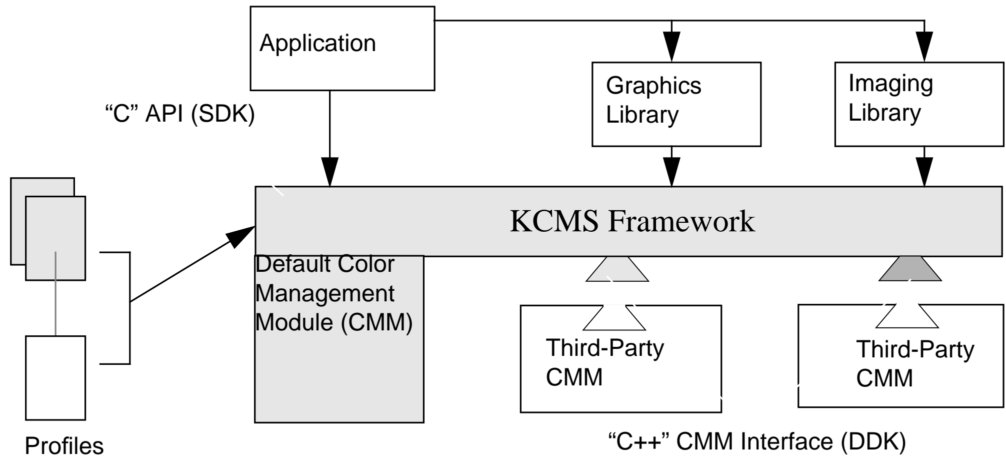

Figure 1–1 KCMS Architecture

KCMS Class Hierarchy

Figure 1-2 shows all relevant classes in the KCMS framework. Each class is described in this chapter.

For descriptions of the enumerations and protected and public members of each class, see the *KCMS CMM Reference Manual*

Note that, when you write a CMM, you can only derive from the `KcsIO`, `KcsProfile`, `KcsProfileFormat`, and `KcsXform` classes, and you can extend `KcsStatus`. How you can use these particular classes in the design of your CMM is described in greater detail in subsequent chapters in this manual. The remainder of the classes your CMM uses are the default classes.

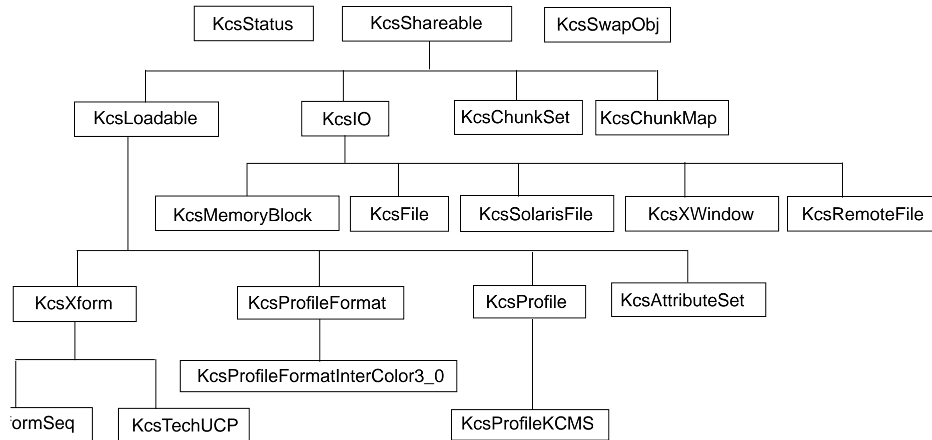

Figure 1–2 KCMS Class Hierarchy

`KcsShareable` Class

The `KcsShareable` class allows derivatives to be shared by other objects in the system. This class uses reference counting. It follows all of the typical C++ semantics, except you should use the `dettach()()` method instead of calling the destructor `~KcsShareable()()`. The `dettach()()` method calls the destructor only if it is the last object sharing the derivative.

Using a shareable derivative is similar to using a non-shareable objects with the following exceptions:

- When you want to use a shareable object with another instance, you must use `attach()()` rather than the constructor.
- Use the `dettach()()` method instead of the `delete()()` method to delete a sharable object.

The abstraction provided by this class is simple yet powerful. With only a few methods you can share objects. Every time you want to share an object, the usage count is incremented. Any time a shared object is detached, the usage count is decremented.

`KcsLoadable` Class

The `KcsLoadable` class allows derivatives to be saved and generated from a static store and possibly minimized and regenerated from that original store at a later time.

If a class cannot regenerate itself at runtime, it must generate itself fully on construction. With KcsLoadable class derivatives, you must allocate and deallocate loadable objects if those objects require regeneration *and* are not supported by the contained class.

All derived objects return KCS_NOT_RUNTIME_LOADABLE whenever regeneration is unsupported. KCS_NOT_RUNTIME_LOADABLE indicates that you must use the constructor and destructor methods to regenerate at runtime.

To relax the requirements on a derivative, assume it is loadable and do not provide any special generation support. That is, allocate the object, load it when necessary, unload it when it is not needed and assume everything worked. In this case, ignore the KCS_NOT_RUNTIME_LOADABLE status message returned by the load()() and unload()() methods.

If the object does not support regeneration it returns KCS_NOT_RUNTIME_LOADABLE when issued a load()() or unload()() command. The object remains loaded in memory so it is not necessary to observe this protocol unless more flexibility is required for performance reasons.

UIDs and Sharing

All KcsLoadable classes have unique identifiers (UIDs). The combination of a chunkSet and a chunkId allows you to save the state of KcsLoadable derivatives for later use. To do this, either minimize and regenerate by calling unload()() and load()(), or save the UID of the instance and reallocate the instance with the UID-based constructor.

Because classes that contain other loadable objects use the same chunkSet, you must save the chunkId within your own data store. To explain this further, an example with an KcsXform class is used; see Chapter 7 "for more information. For example, a sequence transformation saves its array of transform chunkIds in the same chunkset as it does its own state. The KcsXformSeq class has an array of pointers to KcsXforms when it is allocated in memory.

Since all of these transforms have unique identifiers, the KcsXformSeq class places the UID of each transform in an array and saves it. Once this sequence is constructed and told to load, (the chunkId is passed into the constructor) it gets the chunk and, for each transform chunkId, it calls the KcsXform::createXform(uid)() constructor. This constructor allocates the transform associated with that chunkId.

All loadable derivatives should support construction based on this chunkSet and chunkId combination. Loadable objects are shared by using a UID map table kept in the static KcsLoadable data member. When a new loadable object is created, this UID map table is searched first to see if an object with a particular ChunkSet and ChunkId has already been instantiated. If so, the pointer to that object is returned; if not, a new object is created and entered into the table.

Example

```
*aStat
= KcsLoadable::LoadCreator(Kcs2Id('P', 'f', 'm', 't'), Kcs2Id('K', 'C', 'M',
'S'), Kcs2Id('0', '1', '\0', '\0'), Kcs2Id('B', 'l', 'n', 'k'), (void *
(**)())&sCreateFunction,
&sDLHandle);
```

This `LoadCreator()()` example returns a function pointer in `sCreateFunction` that is cast and called with the arguments as follows:

```
KcsProfileFormat::KcsProfileFormat(KcsStatus
*aStat,  KcsId aCmmId, KcsVersion aCmmVersion, KcsId aProfileId,  KcsVersion
aProfVersion)
```

`KcsEkPfmticc30.so.1` is a `KcsProfileFormat` derivative whose method's object code is contained in the file mapped from the arguments to the `LoadCreator()()` method. The `B`, `l`, `n`, and `k` arguments are qualifiers for the constructor to use. In this case it is the Id-based constructor that generates a blank profile format. This call makes runtime loadabilty of derivatives platform independent. If `aDerivId` is a non-ASCII printable character, it is treated as BCD for these reasons: the ICC identifies their versions in BCD, and the runtime derivatives naming conventions need to conform to file naming conventions. Therefore, the operating system cannot use anything nonprintable to name files. If available, the method calls the initialization entry points upon the first load of the sharable. Currently, when an object is loaded, it is not unloaded until the program exits. The dlopen(3x) call returns a pointer to the same handle when it is opened.

Derivatives

Use a `KcsIO` class derivative for a static store. These derivatives can be memory based, disk based, and network based. The object does not care where the information is actually stored. The `KcsIO` base class has a file-like interface. Loadable derivatives also use the `KcsChunkSet` abstraction. This provides a random access bit bucket that is built on top of the `KcsIO` hierarchy.

Once a loadable object is minimized (or unloaded), a derived object regenerates or reloads itself in the manner described below.

The derivation implements the `unload()()` method to minimize the state of the object in memory. Then in the `load()()` method, it restores the state of the object to

that described by the `chunkSet` and `iChunkId` member fields of the loadable base class. If in the unloaded state, it returns an error to signal that a `load()()` call in this state is not adequate for all methods. Additionally, for methods that need access to the state, it loads the minimum state according to the specific derivative's load hints. Then it continues the original method's functionality.

It is assumed that if hints are allowed for loading, the derivative overloads the `load()()` method to allow the hints to be passed to its contained object's `load()()` method.

`KcsIO` Class

The `KcsIO` class provides a generic input/output (I/O) interface to access data in a static store like files on a disk or in memory. The `KcsIO` class provides a common interface for device-, platform-, and transport-independent I/O operations such as read and write. It is a derivative of the `KcsShareable` class. The `KcsFile`, `KcsMemoryBlock`, `KcsSolarisFile`, and `KcsXwindow` are derivatives of the `KcsIO` class that provide I/O for more specific types of data storage.

The `KcsIO` class is primarily provided for operating system vendors to access profiles in devices that cannot be created by deriving from other classes in the system. For example, you may require access to profiles in a printer that have properties not accessible in other classes—if you could not mmap(2) the printer memory.

Note - You must derive from the `KcsIO` class if you require device-, platform- or transport-dependent I/O operations.

See Chapter 4 ”for detailed information.

`KcsFile` Class

The `KcsFile` class is a `KcsIO` base class derivative that allows an implementation of the I/O interface to store its data on a physical disk mounted on the platform in use. It takes an open file as the argument for its constructor and allows sequential and random access file manipulation.

`KcsFile` is useful for embedding profiles in other files.

KcsMemoryBlock Class

The KcsMemoryBlock class is a KcsIO base class derivative that allows you to read from and write to a block of memory.

KcsSolarisFile Class

The KcsSolarisFile class is a KcsIO base class derivative that supports searching for profiles located in known directories and accessing files located on remote machines. It also loads and saves profiles. This class contains a pointer to a KcsIO object of type KcsFile or KcsRemoteFile. This pointer is then used in all of the I/O methods for the class.

KcsSolarisFile cannot be used for embedding profiles and is dynamically loaded at runtime.

KcsXWindow Class

The KcsXWindow class is a KcsIO base class derivative that provides the interface between X11 Window System visuals and corresponding profile data. This class takes as arguments a pointer to a display structure, a pointer to a visual structure, and a screen number. It translates this information into either a local or remote display and creates a KcsFile or KcsRemoteFile pointer. The I/O pointer is then used in all of the derived I/O methods for the class.

KcsXWindow is dynamically loaded at runtime.

KcsChunkSet Class

The KcsChunkSet class provides an interface to access *chunks* (or blocks) of data in a static store (such as a file on disk).

Chunks are separated blocks of data that contain any type of data. The KcsChunkSet class does not know what the data is in the blocks. It provides functions to manipulate the blocks, such as arranging and resizing them.

A chunk set has two components: a chunk map and the chunks. As shown in Figure 1–3 , the chunk map is a table containing an array of descriptions of each chunk. Each chunk map entry contains the chunk Id (a unique identifier for that block), the offset, and the chunk size.

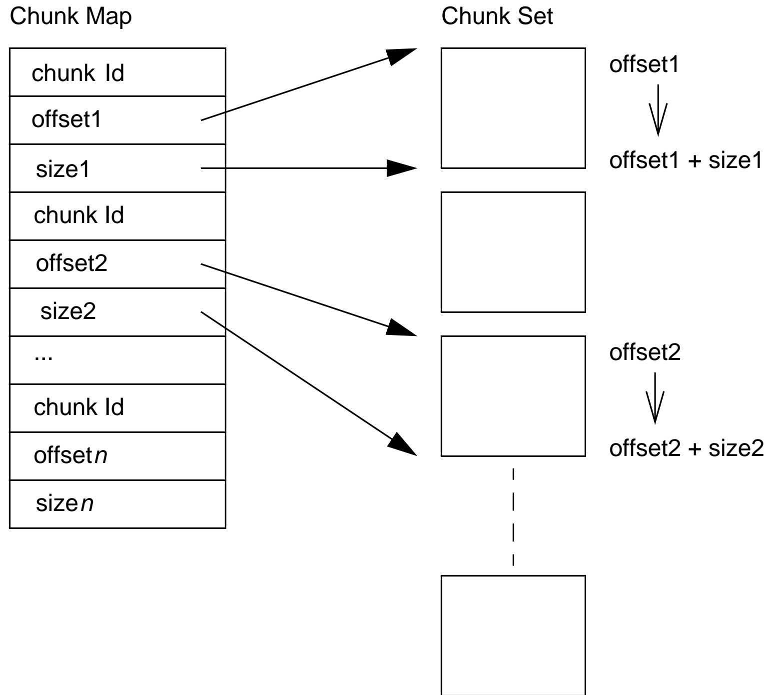

Figure 1–3 Chunk Set Layout

The ICC profile format is directly analogous to the `KcsChunkSet`.

Some `KcsChunkSet` class features are:

- It identifies each chunk by a unique `chunkId`.
- All objects based on `KcsChunkSet`s can be uniquely identified with a combination of `KcsChunkSet` and `chunkId`.
- It uses a `ChunkMap` object to keep track of each chunk's size and the offset of the chunk within the static store.
- It knows nothing about the contents of a chunk.
- It uses an I/O object and tells its I/O object the offset and number of bytes to read or write. Then the I/O object does the actual reading or writing.
- If the size of one chunk changes, it adjusts the location of other chunks in static store as necessary to accommodate the change.
- It relieves other classes from keeping track of specific offsets within the static store.
- It regenerates loadable objects from the static store.

The KcsChunkSet method can be used for various reasons. For example, you need the chunk Id(s) to access data directly. Use KcsChunkSet to read or write a particular chunk Id. You do not need the specific offsets within the static store, but you do need to know the chunk Id(s). You can also specify to write a chunk at a specific static store location. You may want to do this for format conventions that require specific data be stored at a specific location within the static store. In this case, KcsChunkSet moves other chunks to accommodate this request.

KcsProfile Class

The KcsProfile class is a base class that represents a color profile. It is a set of attributes that describe the profile and a set of transformations that allow it to perform the appropriate color changes.

The KcsProfile class is hierarchically derived from the KcsLoadable and KcsShareable classes. This means that profiles can be shared by other objects, and are loadable.

The hierarchy below the base KcsProfile class represents different types of profiles in terms of their techniques, rather than their type. For example, both of the different profile types—Effects Color Profile (ECP) and Device Color Profile (DCP)—can be represented by the same derivative. However, a KCMS profile that uses multi-channel linear interpolation must be a different derivative than an XYZ profile that uses XYZ-based transformations and techniques. Profile types can easily be differentiated by the combination and actual values of the attributes contained within the data. The KcsProfile class determines which transformation technologies a specific profile needs and instantiates the appropriate KcsXform derivatives. For a list of attributes and their possible values see the SDK manual ***KCMS Application Developer's Guide*** and the ICC specification located on-line in /opt/SUNWsdk/kcms/doc/icc.ps. For the most current version of the ICC specification, see the web site at http://www.color.org

The KcsProfile class provides data and necessary KcsXforms to describe, characterize, and calibrate a color-managed input and output device or any point-processible special effect, such as an image filter. It coordinates and determines the loading, saving, and execution of the transformation for all profile types.

Note - You must derive from the KcsProfile class if you want your ICC profiles containing your CMM Id to be used as a loadable module instead of the default profile format.

See Chapter 5 "for detailed information.

`KcsProfileFormat` Class

The `KcsProfileFormat` class allows any of its derivatives to map a profile from a static store into the traditional pieces that make up a profile. All of these pieces are presented to its users as objects in the KCMS framework. Therefore, you can load, set, and get these profile-based objects without regard to the actual format of the data in the store.

Note - You can define your own profile format with this class. If you are using ICC profiles, it is recommended that you use the `KcsProfileFormatInterColor3_0` class, because it deals with ICC profiles.

See Chapter 6 ”for more information.

`KcsAttributeSet` Class

Note - `KcsAttributeSet` is an alias to the `KcsTags` class as indicated in `kcstags.h`. This is for historical reasons only.

The `KcsAttributeSet` class provides a general-purpose interface for an attribute-value pair array. You can associate attributes with different structures.

This object is an associative array—a way of mapping unique identifiers to a variety of data structures. A `KcsAttributeSet` object stores and deletes attributes. Attributes are identifiers and associated data. For a complete discussion of attributes and their properties, see the SDK document *KCMS Application Developer’s Guide* and the ICC Profile Format Specification.

The `KcsAttributeSet` class is a subclass of the `KcsLoadable` class.

The `KcsAttributeSet` class does not override any functionality provided by its parent, but it does provide additional functionality. All access to a `KcsAttributeSet` object is controlled through a set of public methods of the `KcsAttributeSet` class.

The `KcsAttributeSet` class contains a pointer to a `ChunkSet` object that stores the `KcsAttributeSet` data. The `KcsAttributeSet` object uses its `ChunkSet`, if one is supplied when a `KcsAttributeSet` object is created, to read and write data to whatever static store is being accessed by the supplied `ChunkSet`.

Using a KcsAttributeSet Object

A KcsAttributeSet object is created when you need to map identifiers to variable data structures such as ICC tags (that is, integers, floats, strings, and dates). There are two ways to create a non-empty KcsAttributeSet object. The method you choose depends on the origin of the data used to populate the KcsAttributeSet object. The origin can be a supplied chunk or character buffer.

If you do not want to create a KcsAttributeSet object with data from a chunk, you can create a KcsAttributeSet object using a character buffer (the KcsAttributeSet object contains a null chunk set). The only issue you must be aware of in this case is that, in order to save the KcsAttributeSet object, you need to have set the internal chunk set pointer of that KcsAttributeSet object to a valid chunk set and gotten a chunk Id from that chunk set. If the chunk has not been set, then attempting to save the KcsAttributeSet object results in a KCS_UNINITIALIZED_CHUNKSET error.

Using a KcsAttributeSet object, you can perform the following operations:

- Insert new data
- Remove data
- Update data

All three operations performed on the KcsAttributeSet data are accomplished by calling setAttribute()(), a public method of the KcsAttributeSet class. The operation performed is decided by the parameters supplied to the setAttribute()() method and the state of the KcsAttributeSet object when the method is called. Conceptually, only two parameters to the method are important: an identifier and a structure used to contain the variable data associated with that identifier.

To insert new data into a KcsAttributeSet object call setAttribute()() with an identifier not currently used and information stored within the KcsAttributeSet object. For example, assume the structure contains the character string "today is my birthday" as variable data and that the identifier equals 30. After successful completion of a call to setAttribute()(), an association between "today is my birthday" and 30 is stored within the object.

To remove data from a KcsAttributeSet object call setAttribute()() with the identifier of the data you want to delete and assign the information structure parameter to NULL.

To update data in a KcsAttributeSet object call setAttribute()() with an identifier currently used and new information stored within the information structure. For example, assume that the information structure contains the integer data "100 200 300" as variable data and that the identifier is set to 30. After successful completion of a call to setAttribute()(), the association of 30 with "today is my birthday" would be replaced with the association of 30 with "100 200 300" in the object.

Several methods give you information about the `KcsAttributeSet` data as a whole, as well as information about specific associations that make up the `KcsAttributeSet` data. The `returnCurrentNumberOfAttributes()()` method provides the number of associations currently stored within a `KcsAttributeSet` object. The `getAttribute()()` method provides information associated with a specific identifier. The `getTag()()` method returns the nth identifier stored within the data. The `setChunkSet()()` method allows the chunk pointer associated with an instance of a `KcsAttributeSet` object to be reassigned to a new or different chunk; this method is needed to save `KcsAttributeSet` data for a `KcsAttributeSet` object with which no chunk has been supplied. The `getAttributeInfo()()` method provides detailed information associated with an identifier such as the type of data (for example, string, integer, float) and the number of tokens found within the variable data.

`KcsAttributeSet` data is loaded into a `KcsAttributeSet` object when a non-empty `KcsAttributeSet` object is constructed. The `save()()` method is used to store the `KcsAttributeSet` data. As mentioned earlier, a `KcsAttributeSet` object must have a valid chunk in order for the `KcsAttributeSet` data to be saved.

See the *KCMS CMM Reference Manual* for detailed information on all of the `KcsAttributeSet` class member functions.

`KcsXform` Class

The `KcsXform` class represents a set of classes that perform n->m component transformations. These transformations do not need to conform to any single type of transformation. The implementation of a `KcsXform` derivative is irrelevant as long as the derivative transforms in compliance with the base class interface. Some of the most helpful methods are:

- `connect()()`
- `compose()()`
- `optimize()()`
- `save()()`
- `evaluate()()`

All transformations have a number of properties and methods. When using a transform derivative, you can: construct it, load it, save it, associate and inquire storage information, set and retrieve attributes and information about the transform, compose another transformation from it, and most importantly evaluate or transform data.

Note - You must derive from the `KcsXform` class to augment color data processing on the KCMS framework.

See Chapter 7 ”for more information.

`KcsXformSeq` Class

The `KcsXformSeq` class is a `KcsXform` base class that allows other incompatible `KcsXform` derivatives to connect for serial evaluations. It has methods to `append()()` and `insert()()` transformations into an existing sequence and constructors that can instantiate from an array of `KcsXform` pointers. The derivation from the `KcsXform` base class allows a sequence of `KcsXform`s to act like a single `KcsXform` from the perspective of the rest of the architecture.

`KcsStatus` Class

The `KcsStatus` class provides communication of status codes, errors, and customizable textual descriptions of the state. You can dynamically add your own error messages with internationalized text strings associated with them.

You can extend methods in this class to add your own error messages. See Chapter 8 ,” for information on how to add your own error messages.

`KcsSwapObj` Class

The `KcsSwapObj` class provides an interface to swap data between BIG_ENDIAN and LITTLE_ENDIAN hardware architectures. Use this interface for cross-platform compatibility.

CHAPTER **2**

CMM : A Runtime Derivative

In This Chapter

This chapter provides an overview of the requirements necessary to create your CMM as a runtime derivative. It briefly discusses the classes from which you can derive to create your CMM, namely

- `KcsIO`
- `KcsProfile`
- `KcsProfileFormat`
- `KcsXform`

In addition, the chapter discusses the extendable class, `KcsStatus` class.

The chapter starts by identifying the requirements for creating a CMM. Then, it suggests reasons you might derive from (or extend) each of the aforementioned classes to customize your CMM. The chapter approaches runtime derivation prerequisites from the perspective of program coding as well as configuration requirements. The chapter concludes with a discussion of profile requirements, since the primary reason for writing a custom CMM is to create profiles that it can manipulate.

Subsequent chapters detail how to create class derivatives.

Development Environment Requirements

The KCMS packages are automatically placed in a protected directory when you load them with the `pkgadd(3)`command. Copy the packages to a writable directory for development use.

To compile programs, you must use version 4.2 of the Sun™ Visual Workshop™ C++ compiler, which is included with Sun Visual Workshop C++ 3.0.

Requirements For Creating a CMM

A CMM is defined as:

- Color management techniques
- Data structures
- Profiles

The following steps summarize the requirements for creating a CMM that is a runtime derivative:

1. **Understand the KCMS framework, its general principles, and the SDK "C" interface.**

   This manual assumes your understanding of the KCMS framework "C" API functions described in the SDK manual *KCMS Application Developer's Guide*. Chapter 3 ," in this manual, shows you how the KCMS classes implement the framework.

2. **Determine your color management requirements and whether you need to derive from or extend any of the KCMS framework classes to meet those requirements.**

   This chapter suggests ways you can use each of the derivable KCMS classes to meet the special requirements of your CMM. See "Why You Might Derive From or Extend a KCMS Class " on page 17 ," for details.

3. **Understand the ICC profile format.**

   If you want to use the ICC format with different color manipulations, you should familiarize yourself with the ICC profile format. For further discussion, see "Profiles" on page 33 .

4. **Understand the runtime mechanism for derivatives.**

   See “Deriving Classes at Runtime ” on page 19 .

5. **Understand the CMM naming conventions and the `OWconfig` file.**

   See “Configuration Requirements” on page 25 .

6. **Implement your KCMS framework runtime extensions.**

   This manual as well as the *KCMS CMM Reference Manual* describe the foundation library interfaces. You may also find it helpful to refer to the SDK manual *KCMS Application Developer’s Guide* for information on the KCMS framework API.

7. **Test your CMM.**

   In addition to testing your CMM, you can verify that it *adheres* to the KCMS framework. See the DDK manual *KCMS Test Suite User’s Guide* for details on an optional facility to do this.

8. **Follow registration requirements for your CMM.**

   If your profiles are ICC compliant and you intend to make them and/or your CMM available to the public, you must register your CMM Id with the ICC. If your CMM creates attributes for public use, these also must be registered (you don’t need to register private attributes). And finally, if you derive from the `KcsXform` class (see “`KcsIO` Example” on page 27), you must register transform Ids. For details on CMM registration requirements and who to contact, see the ICC profile format specification. The specification to which this version of KCMS conforms is a PostScript file located on-line in `/usr/openwin/demo/kcms/docs/icc.ps`. Check the web site at `http://www.color.org` for the most up-to-date version of the specification.

Why You Might Derive From or Extend a KCMS Class

The following sections give helpful information on why you might derive from or extend a particular KCMS class.

KcsIO

If you have special I/O considerations, you might want to create a `KcsIO` class derivative. It is a simple I/O protocol that most devices support. For example, this version of KCMS includes an X11 Window System derivative (`kcsSUNWIOxwin.so.1`) and a Solaris file derivative (`kcsSUNWIOsolf.so.1`). The code for the latter is included in the `/opt/SUNWddk/kcms/src` directory.

The framework supports file-, memory-, and network-based derivatives. Objects use a static store to read from or write to data; a common type of static store is a file on disk. A *static store* is a hardware- or platform-independent mechanism for generation and regeneration. *Generation* is the first time data is read from a static store and an object is instantiated from that data; the data is constructed from the saved state. *Regeneration*, or loading occurs when a derivative brings back all of its state and functionality, after it has been minimized, from its static store. With minimization and regeneration the object is already instantiated. A minimized object contains sufficient information to generate itself from a static store [typically just its unique identifier(UID)].

See Chapter 4 ," for information on creating a `KcsIO` class derivative.

KcsProfile

Derive a new `KcsProfile` class for characterization and calibration, additional functionality, or new transformation derivatives. Usually, this involves overriding one of the update methods to actually produce a new `KcsXform` derivative. Once the transformation is saved to the static store, the runtime load mechanism automatically handles it from then on, as long as the CMM is installed on the loading system. Since you can supply profiles directly that contain new `KcsXform` derivatives, the only derivative necessary to supply is the one derived from the `KcsXform` class. However, if the profile used to contain these new `KcsXform` derivatives is the `KcsProfile` derivative, it overwrites the new `KcsXform` type with one of its own when calibrated.

See Chapter 5 ," for information on creating a `KcsProfile` class derivative.

KcsProfileFormat

You can create a `KcsProfileFormat` class derivative to support an existing non-ICC profile format, a new profile format, or possibly a set of data that is not an ICC profile (for example, a tag encoded TIFF file). To build a `KcsAttributeSet` instance within a `KcsProfileFormat` instance and a set of `KcsXform` derivatives, enough information in a properly tagged TIFF image might exist.

The CMM Id and version must be in a known location in the profile header. This is always the case with the ICC profile format. See "ICC Profile Header " on page 34 ,"

for details. Other profile formats must be formatted to conform to this requirement so that the KCMS framework can form the keys to locate the format's runtime loadable module.

See Chapter 6 ," for information on creating a `KcsProfileFormat` class derivative.

KcsXform

Typically you create a `KcsXform` class derivative when you create a CMM. The `KcsXform` is the most common derivative, since most color management suppliers have their own type of transformation technology. Most color technology involves manipulating matrixes and transforms. This class allows you to define a transform and its methods.

See Chapter 7 ," for information on creating a `KcsXform` class derivative.

KcsStatus

The `KcsStatus` class represents a consistent object-oriented way of returning results from all the KCMS methods. It enables each of the following to be represented:

- Error and warning values
- Error text descriptions
- Error conversions to and from a `KcsStatusId`
- Error comparisons
- External mappings through the KCMS C API
- Message extraction for language localization

You do not actually derive from the `KcsStatus` class: you extend it. You override an error and warning message function to provide your own error and warning messages. It is recommended that you override `KcsStatus` functions with message extraction for language localization. You are not required to provide your own messages. The KCMS-framework error and warning messages may be sufficient.

See Chapter 8 ," for information on extending the `KcsStatus` class.

Deriving Classes at Runtime

The KCMS framework uses a model that allows derivation of classes at runtime. This model changes and augments the default functionality in the pre-compiled shared library. The KCMS framework uses the Solaris runtime-loader interface. See the

dlopen(3X) and dlsym(3X) man pages for more information. The runtime derivation model uses C-based routines to load, unload, initialize, terminate, and allocate at runtime.

To enable runtime derivation, you must code your CMM according to special requirements and configure each derived class it contains. Each of these topics is addressed in the sections below. See "Runtime Derivation Coding Requirements " on page 20 and "Configuration Requirements" on page 25 .

Runtime Derivation Coding Requirements

This section describes what you need to do in the code of each of your CMM derivatives so that the derivative can be executed at runtime.

Runtime Derivation Code Examples

For code examples showing how to use the wrapper functions and entry points described in the sections below, see the chapter describing that derived class. The chapters are:

- Chapter 4 "
- Chapter 5 "
- Chapter 6 "
- Chapter 7 "

In addition, see the sample programs in /opt/SUNWddk/kcms/src. The directory contains brief sample programs that illustrate all the coding requirements described in this section.

Wrapper Functions

To allocate an object at runtime, you use wrapper functions. ***Wrapper functions*** are implemented in C and perform a C++-to-C conversion. The allocation routines return a pointer to the base class object that indicates to the C++ compiler what is returned, but not the definitions. As in typical C, you can reference symbols in a sharable library because the functions are defined as `extern C {}()`.

These functions are written in C++ and call `new()()` (or its equivalent alternative). Since the shareable object code has all of the header information from the base class,

the derivative is constructed properly and has the same structure as statically-linked code.

External Entry Points

You need to provide the KCMS framework with an external entry point to load each of your derivatives as an executable. The symbols are loaded and the derivative is called by the framework to access your derivative's functionality.

The types of entry points for a runtime derivative are:

- mandatory
- optional
- base-class specific

In the paragraphs that follow, *XXXX* refers to the base class identifier from which it is being derived. *XXXX* can *only* have the following values:

- `IO`
- `Prof`
- `Pfm`
- `Xfrm`
- `Stat`

Mandatory

For each runtime derivative, you must supply a C-based variable and an external entry point. Respectively, these are:

- `KcsDLOpen()`*XXXX*`Count()()`
- `KcsCreate()`*XXXX*`()()`

Note - The `KcsDLOpenStatusCount()()`variable is the only requirement for extending the `KcsStatus` class.

`KcsDLOpen()`*XXXX*`Count()`

( )

```
extern

long

KcsDLOpen()XXXXCount();
```

KcsDLOpen()*XXXX*Count()()is the number of times the shareable object was opened. It is equivalent to the number of times the shareable object is being shared. The CMM should not set this variable. It is controlled by the KCMS framework.

KcsCreate()*XXXX*()
( )

```
KcsXXXX

*KcsCreateXXXX(KcsStatus *,

XXXX creation

args);
```

KcsCreate()*XXXX*()() is one of possibly many creation entry points. This entry point maps directly to the static create()*XXXX*()() methods of the base class from which it is being derived. The arguments following KcsStatus * are specific to the base class and are described in the appropriate class chapter. See the chapters describing the KcsIO, KcsProfile, KcsProfileFormat, and KcsXform classes (Chapter 4 through Chapter 7 , respectively). The CMM must support all declared create()*XXXX*()() methods; otherwise, applications receive CMM errors from calls to load()().

Optional

Runtime derivatives can supply the following external entry points:

- KcsInit()*XXXX*()()
- KcsCleanup()*XXXX*()()

Note - It is highly recommended that you use the KcsInit()*XXXX*()() entry point to verify the version of the versions so that your CMM can use the profile data properly.

`KcsInit()`*XXXX*`()`
`()`

```
KcsStatus
KcsInit()XXXX(long libMajor,
long libMinor,   long *myMajor, long
*myMinor);
```

If you supply the `KcsInit()`*XXXX*`()()` entry point, the KCMS framework calls it when the shareable object is loaded for the first time. This initializes and derives private allocations before any creation method is called. `KcsInit()`*XXXX*`()()` checks for minor version numbering. See “Configuration Requirements” on page 25 for more information.

`KcsCleanup()`*XXXX*`()`
`()`

```
KcsStatus
KcsCleanup()XXXX()();
```

If you supply the `KcsCleanup()`*XXXX*`()()` entry point, the KCMS framework calls it when the shareable object is unloaded for the last time (when `KcsDLOpen()`*XXXX*`Count = 0()`). This cleans up shareable objects when they are no longer needed.

Base-Class Specific

Each base class also provides additional necessary and optional entry points. See the chapters describing the `KcsIO`, `KcsProfile`, `KcsProfileFormat`, and `KcsXform` classes (Chapter 4 through Chapter 7 , respectively), for detailed explanations.

Instantiation

You instantiate KCMS framework objects or any runtime derivations of that object with the following methods:

- `createXXXX()()`
- `attach()()`
- `new()()`

`create()`*XXXX*`()()`

You allocate an object with the `create()`*XXXX*`()()` method. This method combines sharing of the object with runtime derivative support. With chunk set-based objects, this function searches for a match through allocated objects. If it finds a match, it attaches to that object and returns its address. If it does not find a match or the object is not chunk set based, it searches for a match through objects in the runtime-loadable object files.

To maximize the runtime nature of the KCMS framework, it is recommended that you use the `create()`*XXXX*`()()` method whenever possible within your CMM derivative. It enables derivatives statically linked into an application or included directly in the KCMS framework's shared object libraries (such as, `libkcs`) to use the correct and latest version of your CMM derivative.

Note that the `KcsStatus` class extension is an exception to this recommendation. It passes back a status string rather than a pointer to a derivative, and only two C functions are written.

`attach()()`

You use the `attach()()` method to share an object. If an object already exists, it can be shared in memory with this method. You can share the object with other users of that object. Any changes in the object are applied to objects that share it. If you share an object, make sure that object does not change while your derivative is attached to it.

`new()()`

To get a new object of a specified type or a KCMS framework derivative, you use the `new()()` method. This allows a runtime derivative to actually override a built-in type after it has been released.

Initialization and Cleanup

The `KcsLoadable` class loads a runtime derivative's binaries when a `create()()` method is used. It generates the shared object's configuration file keywords based on class, derivative, and version identifiers. It retrieves the module name and loads the library. See "Creating `OWconfig` File Entries" on page 27 for further information.

The `KcsLoadable` class then locates the `KcsDLOpen()`*XXXX*`Count()` variable. If `KcsDLOpen()`*XXXX*`Count = 0`, it locates and loads the `KcsInit()`*XXXX*`()()` entry point and, if available, calls it. Then the `KcsCreate()`*XXXX*`()()` entry point is located and loaded. If everything is successful, the `KcsCreate()`*XXXX*`()()`entry point is called.

When the last of a specific derivative type is deallocated and the `KcsCleanup()`*XXXX*`()()` entry point is available, it is located, loaded, and called.

Configuration Requirements

This section tells you what you need to know to load your CMM. It explains how to name derived classes and how to update the `OWconfig` file for each derived class in your CMM.

CMM Filename Convention

A module (or CMM) name should follow this convention:

```
kcs<STOCK
SYMBOL><CLASS><unique
identifier>.so.<version>
```

Table 2–1 describes each field in the CMM filename.

TABLE 2–1 CMM Filename Description

Filename Field	Description
`kcs`	Color management framework.
stock symbol	Short mnemonic used by the stock market or a unique identifier.
class	Class from which the module is derived (`IO`, `Prof`, `Pfmt`, `Xform`, or `Stat`).
unique identifier	Four-character identifier that distinguishes multiple modules derived from the same class.

TABLE 2–1 CMM Filename Description *(continued)*

Filename Field	Description
`.so`	Shared object library.
version	Number compared to the KCS_MAJOR_VERSION number (incremented by SunSoft for every major release; for bundling CMMs only).

Note - The version number in the `#define` and the version number in the module name *must* match. See `icc.h` for an example.

Table 2–2 lists a few KCMS CMM filenames.

TABLE 2–2 KCMS CMM Filenames

CMM Filename	Description
`kcsSUNWIOsolf.so.1`	Solaris File CMM component
`kcsSUNWIOxwin.so.1`	X11 Window System CMM component
`kcsSUNWStatsolm.so.1`	Solaris Message CMM component, which accompanies each of the above CMM files for messages

CMM Makefile

You must install your CMM in `/usr/openwin/etc/devhandlers`. To copy the file into this directory, you must be superuser and you must set the file permissions to 755 so that it is executable.

See the sample makefile in `/opt/SUNWddk/kcms/src` for an illustration of how CMMs are compiled and installed, and how the CMM library names are associated with the CMM modules.

Creating OWconfig File Entries

You must include OWconfig library in the CMM linking. To link in this library, enter -lowconfig on the link command line . The OWconfig library is bundled with Solaris in /usr/openwin/lib/libowconfig.so. It provides routines to access the OWconfig file, which gets the name of the CMM class derivitive you want to dynamically load.

To advertise its existence of your CMM class derivative to the KCMS framework, you must add the name of the derivitive to the OWconfig file.

A generic OWconfig entry looks like this:

```
class=''KCS_IO''

name=''solf''

 kcsLoadableModule=''kcsSUNWIOsolf.so.1'';
```

Table 2–3 describes each field.

TABLE 2–3 OWconfig File Entry Description

OWconfig **File Entry**	**Description**
class	KCS_<*class name*>.
name	Four- or eight-character identifier that matches the identifier in your code.
kcsLoadableModule	Entire module name.

The above is just an example of the OWconfig file structure. You need to add an OWconfig file entry for each class you derive from or extend in your CMM. Examples of these entries are shown in the following paragraphs.

KcsIO Example

If you derive from the KcsIO class, you need to provide a KcsIO class entry such as the example entries below:

```
#KcsIO

class, Solaris profiles class=''KCS_IO'' name=''solf''

 kcsLoadableModule=''KcsSUNWIOsolf.so.1''; #KcsIO class, X11 window

system profiles class=''KCS_IO'' name=''xwin''

 kcsLoadableModule=''kcsSUNWIOxwin.so.1'';
```

Note the `name` strings in the above examples. The `KcsProfileType` enumeration in `kcstypes.h` contains a type field that is a 4-character array described in hexadecimal form as a `long`, for example:

```
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;
```

The `OWconfig` library turns the `type` field back into a string corresponding to the `name` field entry and searches all of the appropriate `OWconfig` class entries for that string.

`KcsProfile` Example

If you derive from the `KcsProfile` class, you need to provide a `KcsProfile` class entry such as the example entries below:

```
#KcsProfile

Class, Solaris default is KCMS #Default profile class, CMM Id == Profile

Format class=''KCS_Prof'' name=''dflt''

 kcsLoadableModule=''kcsEkProfkcms.so.1''; #KCMS profile, CMM Id ==

Profile Format class=''KCS_Prof'' name=''KCMS''

 kcsLoadableModule=''kcsEKProfkcms.so.1'';
```

The key to loading a new version is the CMM Id (also called CMM Type), which is contained in bytes 4 through 7 in the ICC profile header). (See “ICC Profile Header ” on page 34 for details.) If there is not a match, the default entry `dflt` is used. You must load the proper CMM Id into the new profile’s CMM Id attribute field for recognition of the module.

The default loadable `KcsProfile` module is the Solaris-supplied default.

The `KcsProfile` class is the base class that can contain transformations (`KcsXform` class) and a profile format (`KcsProfileFormat` class). Since the Kodak `KcsProfile` class is built into the library, you can use this mechanism to extend the calibration and characterization interface.

`KcsProfileFormat` Example

If you derive from the `KcsProfileFormat` class, you need to provide a `KcsProfileFormat` class entry such as the example entries below:

```
#Profile

format class, default is ICC #Default profile format, ICC, default CMM

class=''KCS_Pfmt'' name=''acspdflt''

 kcsLoadableModule=''kcsEKPfmticc30.so.1''; #ICC profile format,

KCMS CMM class=''KCS_Pfmt'' name=''acspKCMS''

 kcsLoadableModule=''kcsEKPfmticc30.so.1'';
```

The profile format is determined from the profile file signature (also known as the *magic number*), which is contained in bytes 36-39 in the ICC profile header), and CMM Id (bytes 4-7 in the header). (See “ICC Profile Header ” on page 34 .) A check is performed to ensure that an ICC profile uses the magic number of the file. If another format is used, the magic number is used to load the module.

All profiles that are ICC profile format files should have a magic number equal to acsp, and they must have the ICC header included. The CMM Id is used to match the profile format with the correct derivative. If no match is found, the default entry (dflt) is used; therefore, you can use the supplied default profile format class for ICC profiles.

The name field syntax is: <***Profile magic number***><***CMM Id***>

The OWconfig file entry must match the resulting name. This also gives color management vendors the opportunity to support pre-ICC format profiles, provided they include the ICC header.

KcsXform Example

If you derive from the KcsXForm class, you need to provide a KcsXForm class entry such as the example entries below:

```
#ICC

interpolation table 8 bit, default CMM class=''KCS_Xfrm''

name=''mft1dflt''

 kcsLoadableModule=''kcsEKXfrmucp.so.1''; #ICC interpolation table

16 bit, default CMM class=''KCS_Xfrm'' name=''mft2dflt''

 kcsLoadableModule=''kcsEKXfrmucp.so.1''; #ICC interpolation table

8 bit, default CMM class=''KCS_Xfrm'' name=''mft1KCMS''

 kcsLoadableModule=''kcsEKXfrmucp.so.1''; #ICC interpolation table

16 bit, default CMM class=''KCS_Xfrm'' name=''mft2KCMS''

 kcsLoadableModule=''kcsEKXfrmucp.so.1''; #KCMS universal color

processor table class=''KCS_Xfrm'' name=''ucpKCMS''

 kcsLoadableModule=''kcsEKXfrmucp.so.1'';
```

The name field is a combination of a unique 4-character transform identifier and the CMM Id. (The transform identifier must be registered with the ICC if the profile and CMM are to be made available to the public.) The library turns name back into a string and searches all of the appropriate OWconfig class entries.

Inside an ICC profile, the type of transform is defined by a type identifier that indicates whether it is an 8- or 16-multi function table, indicated by the signature element of either the Lut8Type (mft1) or Lut16Type (mft2). Default values have been supplied for these cases: mft1dflt and mft2dflt.

KcsStatus Example

If you extend the KcsStatus class, you need to provide a KcsStatus class entry such as the example below:

```
#Extending

error messages class=''KCS_STAT'' name-''solm''

 kcsLoadableModule=''kcsSUNWSTATsolm.so.1'';
```

The name field is a 4-character string that uniquely identifies your set of error and warning messages.

To add your own error messages, supply a single “C” routine that translates your error value into an error string. Also supply a messages.po file for localization purposes. See Chapter 8 ”for detailed information.

If an OwnerId variable is set with the status message, the KcsStatus class dynamically loads the matching OwnerId that was set by the dynamically loaded class. The OwnerID is described in Chapter 8 .

Updating the OWconfig File

Using the interactive program called OWconfig_sample, which is provided with the DDK, you can insert and remove configuration entries (class, name, and kcsLoadableModule) in the OWconfig file.

To update OWconfig file entries, you must be root. If you are not root or the /etc/openwin/server/etc path does not exist, the following error is generated:

```
OWconfile

file not created/updated. Check that you are root and /etc/openwin/server/etc

exists.
```

Start the OWconfig_sample program as follows:

```
example%

su example# ./OWconfig_sample
```

Inserting Entries

The following is an example of how to insert a configuration entry into the `OWconfig` file. Sample user responses are enclosed in brackets (`[ ]`).

CODE EXAMPLE 2–1 Inserting an OWconfig File Entry

```
ATTENTION:You must be

root to update the OWconfig file. Are you inserting an OWconfig entry? y/n

[y]  You will be asked to supply a class, a name and a kcsLoadableModule to

create an entry such as the following: class = ‘‘SUN_Xfrm’’ name =

‘‘acspdndp’’ kcsLoadable Module = ‘‘sunSUNWxfrmdndp.so.1’’

Please see the KCMS CMM Developer’s Guide for information  Enter the

profile class name in the form of XXX_IO, XXX_Prof, XXX_Prfmt, XXX_Xfrm, or

XXX_Stat. XXX is your unique cmm identifier, which must match the class name

of your derived class.  [KCS_Prof]  Enter the name of your cmm - it must match

the name in your derived class.  [dndp]  Enter the name of your dynamically

loadable module.  [kcsSUNWProfdndp.so.1]  This is your OWconfig entry. OK? y/n

 class = KCS_Prof name = dndp kcsLoadableModule = kcsSunWProfdndp.so.1  [y]

Do you have more entries to create? y/n  [n]
```

The `OWconfig_sample` program above appends the entry to the `OWconfig` file.

Try inserting the entry shown in the above example:

1. **Run the `OWconfig_sample` program. Be sure you are root.**
   See “Updating the `OWconfig` File ” on page 31 .

2. **Insert the user responses shown in Code Example 2–1 .**

3. **Check for the entry at the end of the `OWconfig` file.**
   The new configuration entry is appended to the end of the `/usr/openwin/server/etc/OWconfig` file. For local machine use only, the `/etc/openwin/server/etc/OWconfig` file is updated.

Removing Entries

The following is an example of how to remove a configuration entry from the `OWconfig` file. Sample user responses are enclosed in brackets (`[ ]`).

CODE EXAMPLE 2–2 Removing An OWconfig File Entry

```
ATTENTION:You must be
root to update the OWconfig file. Are you inserting an OWconfig entry? y/n
[n]  To remove an OWconfig entry:  Enter the unique class name for your module
for removal from the OWconfig file. The class name would be in the form
XXX_IO, XXX_Prof, XXX_Pfrmt, or XXX_Stat  [KCS_Prof]  Enter the unique cmm
name for your module for removal from the OWconfig file.  [dndp]  This is the
OWconfig entry to remove. OK? y/n  class = KCS_Prof name = dndp  [y]  Do you
have more entries to remove? y/n  [n]
```

The `OWconfig_sample` program removes the last entry in the `OWconfig` file.

Try removing the entry you inserted following Code Example 2–1 :

1. **Run the `OWconfig_sample` program again. Be sure you are root.**
   See “Updating the `OWconfig` File ” on page 31 .

2. **Fill in the user responses shown in Code Example 2–2 .**

3. **Check the `OWconfig` file.**
   The entry should no longer appear at the end of the file.

Version Numbering

Once `OWconfigGetAttribute()()` returns the module name, the version number is parsed out of the module name and compared to the global library version number located in `kcsos.h` to determine if this version can be executed.

Note - The major version number in the module name *must* match the global variable located in the header file `kcsrpc.h` for an example.

Profiles

This chapter would not be complete without a discussion on profiles. You write a CMM to create profile(s) that meet your special requirements. This section describes the KCMS profile standard. It makes recommendations regarding the portability of

the profiles your CMM creates. Finally, it explains how to set up the profiles so that your CMM can use them.

KCMS uses the ICC profile format as the default profile format. The ICC profile format specification to which this version of KCMS conforms is located on-line in `/usr/openwin/demo/kcms/docs/icc.ps`. (For the most current version of the ICC specification, check the web site at http://`www.color.org`.) By supporting the ICC specification, the profiles your CMM creates can be used on all participating vendors' color management systems.

Much of the work in processing and handling ICC format profiles is included by default in the framework. To speed your development cycle, use as much of this default technology as possible. You can develop your own profile format within the framework.

Note - It is strongly recommended that you extend the ICC profile format rather than develop your own. If you do not use the ICC profile format your profiles will not be easily ported to other platforms.

ICC Profile Header

Table 2–4 shows the ICC profile header. For details on the profile header fields, see the ICC profile format specification. The KCMS-specific header is defined in `icc.h`, and KCMS attributes are registered with the ICC. (The profile header and all attribute structures and definitions that pertain to KCMS also are described in Chapter 5, "Profile Attributes," in the SDK manual *KCMS Application Developer's Guide*.)

Note that the CMM Id must in a fixed place (bytes 4-7) in the header .

TABLE 2–4 ICC Profile Header Format

Byte offset	Content
0-3	Profile size
4-7	CMM type (CMM Id)
8-11	Profile version number
12-15	Profile/device class
16-19	Color space of data
20-23	Profile connection space (PCS) (CIEXYZ or CIELAB only)
24-35	Date and time this profile was first created

TABLE 2–4 ICC Profile Header Format *(continued)*

Byte offset	Content
36-39	'acsp' (61637370h) profile file signature (magic number)
40-43	Primary platform target for the profile
44-47	Flags indicating various options for CMM such as distributed processing and caching
48-51	Device manufacturer of the device for which this profile is created
52-55	Device model number of the device for which this profile is created
56-63	Device attributes unique to the particular device setup such as media type
64-67	Rendering intent
68-79	XYZ values of the illuminant of the PCS.
80-83	Profile creator identifier
84-127	Reserved for future use

Naming and Installing Profiles

Any profile you want to include in the KCMS library must be named according to specified conventions to avoid name clashes and promote portability. The paragraphs below explain how to name and install your profile so that it can be automatically used in the KCMS framework.

Naming Profiles

The KCMS profile name is a filename with the following naming convention:

<CMM

ID><stock

symbol><device>.<type>

Table 2–5 describes the fields in the profile filename:

TABLE 2–5 Profile Filename Description

Profile Filename Field	Description
CMM ID	A mnemonic. Solaris-supplied profiles use `kcms` as the CMM ID. Choose your own mnemonic for profiles you create.
stock symbol	Short mnemonic used by the stock market for your company or a unique identifier.
device	Unique string identifying the device or color space. See Table A–1 for devices supported by Solaris.
type	ICC profile format standard filename suffixes. See Table 2–6 .

Profile Filename Suffixes

Table 2–6 describes the various filename suffixes for profiles for the Solaris environment.

TABLE 2–6 Profile Filename Suffixes

Filename Suffix (`type`)	Description
`inp`	Input devices (scanners, digital cameras and Photo CDs)
`mon`	Display devices (CRTs and LCDs)
`out`	Output devices such as printers
`spc`	Color space conversion transformations
`link`	Device link transformations
`abst`	Abstract transformations for special color effects

Installing Profiles

If you use the file type `KcsSolarisFile` in the `KcsLoadProfile()()` function, you must install profiles where the KCMS framework can locate them . The KCMS framework searches for profiles in the following directories in the order listed below:

1. Local current directory

   Typically a CMM creates profiles in the directory in which it is being run.

2. Directories specified by the `KCMS_PROFILES` environment variable

   `KCMS_PROFILES` is a colon-separated list of directory paths to profiles. You can set this variable on a per-user or work-group basis. You may want to use this variable to set the path to temporary profiles your CMM creates.

3. `/etc/openwin/devdata/profiles`

   This directory contains the local or machine-specific copies of configured profiles, for example, X Window System visual profiles.

4. `/usr/openwin/etc/devdata/profiles`

   This directory always contains read-only files.

Note - All profiles for distribution (whether you create them or they are supplied with the Solaris operating system) should be written as superuser and read only to protect them from being overwritten.

Note - If you use the file type `KcsFile` in the `KcsLoadProfile()()` function, when the application opens the profile, you can name it anything you want.

Supported Devices

For a list of the devices supported by the KCMS framework, see Appendix A .

CHAPTER 3

KCMS Framework Operations

In This Chapter

The Kodak Color Management System (KCMS) is a flexible and powerful framework for developing color management technology. You can add attributes to the current list and incorporate new color processing technology.

This chapter provides an overview of the KCMS framework architecture. It introduces you to how the framework works by showing excerpts form KCMS "C" API programs and comparing these excerpts to the underlying framework actions.

KCMS Framework Architecture

Figure 3–1 illustrates the KCMS framework architecture. It provides an overview of how the KCMS classes are used to implement the KCMS framework. Basically, the framework is implemented by manipulating an array of `KcsProfile` objects within a set of "C" wrapper functions. The "C" wrapper functions are C-to-C++ calls that make up the KCMS API. Use this figure as a general reference as you read this chapter.

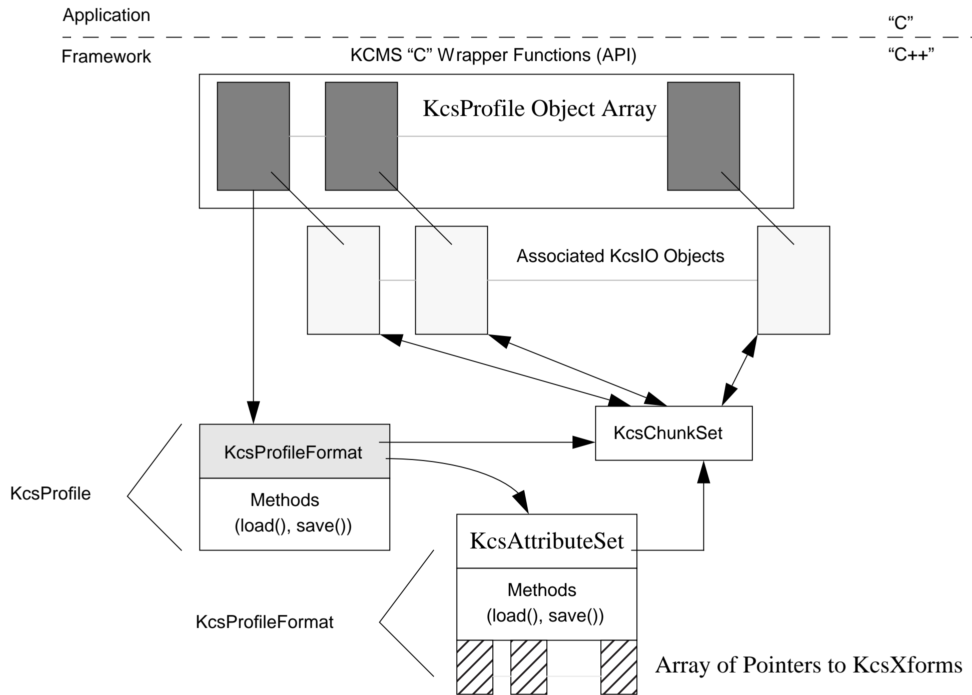

Figure 3–1 KCMS Framework Architecture

KcsProfile

`KcsProfile` objects are created from a ***static store*** which is a `KcsIO` object. `KcsProfile` objects are described using one of the types in the `KcsProfileDesc` structure which is defined in the `kcstypes.h` header file. Objects can read from and write to data in a static store. Examples of a static store include a file and memory. `KcsProfile` objects generated internally by the framework use a `KcsMemoryBlock` object.

The `KcsProfile` class static member function, `createProfile()()` reads the CMM Id from the static store and generates a pointer to the `KcsProfile` derivative. The CMM Id is located at byte 4 in the ICC profile format. If the CMM Id has no associated runtime derivative, the default `KcsProfile` derivative, `KcsProfileKCMS`, is used.

Note - The CMM Id must be in a set location in the file that is the same location as used by the ICC profile format. For details, see "ICC Profile Header " on page 34 .

The `KcsProfile` class contains a set of public member functions that correspond to the KCMS "C" API functions shown in the following table.

TABLE 3–1 Mapping of API Functions to `KcsProfile` Class Member Functions

KCMS API Function	`KcsProfile` **Member Functions**
`KcsLoadProfile()()`	`load()()`
`KcsSaveProfile()()`	`save()()`
`KcsSetAttribute()()`	`setAttribute()()`
`KcsGetAttribute()()`	`getAttribute()()`
`KcsConnectProfiles()()`	`connect()()`
`KcsEvaluate()()`	`evaluate()()`
`KcsUpdateProfile()()`	`updateXforms()()`

`KcsProfileFormat`

Each `KcsProfile` base class contains a pointer to a `KcsProfileFormat` object. This allows the architecture to link different profile formats and keep the `KcsProfile` class independent of the actual profile format. The `KcsProfileFormat` object is created based on the profile format Id (also called ***profile file signature***) and profile version number. The ICC profile format Id is `acsp`, located at byte 36 in the profile header. (See “ICC Profile Header ” on page 34 .) The version number is derived from the profile version number; ICC profile byte 8. The framework uses the version number with the profile format Id so that it can handle different versions of profile formats. For non-ICC profile formats the format Id and version number must be at the same byte location in the static store.

`KcsAttributeSet`

Each `KcsProfileFormat` base class contains a pointer to a `KcsAttributeSet` object and handles all of the functionality for attributes.Using the `KcsIO` class associated with the parent `KcsProfile`, the `KcsAttributeSet` object can load itself from the static store. `KcsAttributeSet` does not use the `KcsIO` class directly;

it uses the `KcsChunkSet` utility class to access the static store. `KcsChunkSet` knows how to handle the mapping from desired information blocks to its actual location in the static store. `KcsChunkSet` and `KcsIO` have no knowledge of the contents of the data. That is left to the calling class.

KcsXform

The `KcsXform` base class contains an array of pointers to `KcsXform`s. The primary function of `KcsXform` (or transformation) is to manipulate color data. `KcsXform` also uses the `KcsChunkSet` class to load from and save to static store.

KCMS Framework Flow Examples

The following examples will help you better understand the flow of control and data between the KCMS "C" API and the KCMS framework. Use Figure 3–1 as a reference.

Loading a Profile

This example explains what the KCMS framework does when an application makes a KCMS "C" API call to load a profile.

1. Using the `KcsIO` derivative, the framework determines the CMM Id of the profile.
2. The framework calls the `KcsProfile::createProfile()()` static method and loading starts. It uses the CMM Id of the profile as a key to determine the particular `KcsProfile` derivative to load. The CMM Id is associated with the dynamically loadable module using entries in the `OWconfig` file.

   Once dynamically loaded, the module returns a pointer to a `KcsProfile` object. If the particular CMM Id has no match in the `OWconfig` file, the framework uses the default `KcsProfile` derivative, `KcsProfileKCMS`. There is a special CMM Id key `dflt` entry in the `OWconfig` file, so that your CMM can override the default `KcsProfile` class. (See "`KcsProfile` Example" on page 28 for details.) If you want your CMM to override the default `KcsProfile` class, it *must* duplicate all of the functionality that the default class handles.
3. The framework calls the `load()()` method on the `KcsProfile` object pointer that was created in step 2. This causes a `KcsProfileFormat` object pointer to be created using an entry in the `OWconfig` file. Then the `KcsProfileFormat` object loads itself.

   The profile format Id (also called the profile signature or magic number, which starts at byte 36 of the ICC profile format) is used as the key to this entry in the `OWconfig` file. (For details on the ICC profile format, see "ICC Profile Header " on page 34 .)

4. The `KcsProfileFormat` object contains pointers to a `KcsAttributeSet` object and an array of pointers to `KcsXform` objects. The framework also creates these objects and calls their `load()()` methods so they load themselves from the static store.

   Your CMM can derive directly from the `KcsAttributeSet` object, since it is statically linked into the KCMS framework. The `KcsXform` array has an `OWconfig` entry that uses a 4-byte identifier as a key. For ICC-based profiles, use the 8- and 16-bit LUT tags, `mft1` and `mft2`. (See "`KcsXform` Example" on page 30 for details.)

5. If all pieces of the profile load successfully, the framework returns a `KCS_SUCCESS` status to the calling application.

Getting Attributes

Once a profile is loaded (see "Loading a Profile" on page 42), an application can call the `KcsGetAttribute()()` "C" API function to retrieve the profile's attributes. The following outlines the flow of control at the framework level to obtain the profile's attributes (see Figure 3–2):

1. For the appropriate object in the `KcsProfile` object array , the framework calls its `getAttribute()()` method.
2. The `KcsProfile()`*XXXX*`::getAttribute ()`calls its `KcsProfileFormat::getAttribute()()` method.
3. This in turn calls its `KcsAttributeSet::getAttribute()()` method.
4. The `KcsAttributeSet::getAttribute()()` method gets the attribute and returns it back up the chain to the API layer.

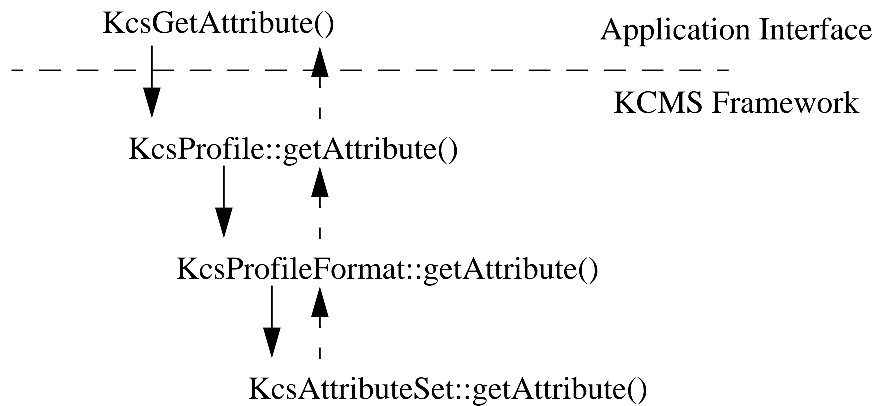

Figure 3–2 `KcsGetAttribute()()` Flow Example

A similar flow of control is true for the other KCMS "C" API calls.

KCMS Framework Primary Operations

The remainder of this chapter describes how the KCMS framework operates from the perspective of the KCMS "C" API. Code examples show KCMS "C" API calls that perform the following tasks:

- Loading profiles
- Connecting profiles
- Evaluating data
- Freeing profiles
- Getting and setting attributes
- Accessing characterization and calibration data
- Saving profiles

Within the primary framework, events are illustrated and described in sequence to explain what actually takes place when each "C" API call is made.

Loading a Profile From the Solaris File System

The framework must have a profile with which to operate. Code Example 3–1 is a KCMS "C" API code excerpt that loads a scanner profile with a file name.

CODE EXAMPLE 3–1 Loading a Profile from the Solaris File System

```
KcsProfileId

scannerProfile; KcsProfileDesc      scannerDesc; KcsStatusId      status;

char   *in_prof= ``kcmsEKmtk600zs'';  scannerDesc.type =

KcsSolarisProfile; scannerDesc.desc.solarisFile.fileName = in_prof;

scannerDesc.desc.solarisFile.hostName = NULL;

scannerDesc.desc.solarisFile.oflag = O_RDONLY;

scannerDesc.desc.solarisFile.mode = 0;  /* Load the scanner profiles */ status

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

(status != KCS_SUCCESS) {  fprintf(stderr,''scanner KcsLoadProfile failed

error =     0x%x\n'', status);  return(-1); }
```

In the example, the KCMS API layer calls `KcsLoadProfile()( )` to inform the KCMS framework that a profile description of type `KcsSolarisProfile` is to be

loaded. The name of the profile and the options for opening that file are also specified using the `solarisFile` entry in the `KcsProfileDesc` structure.

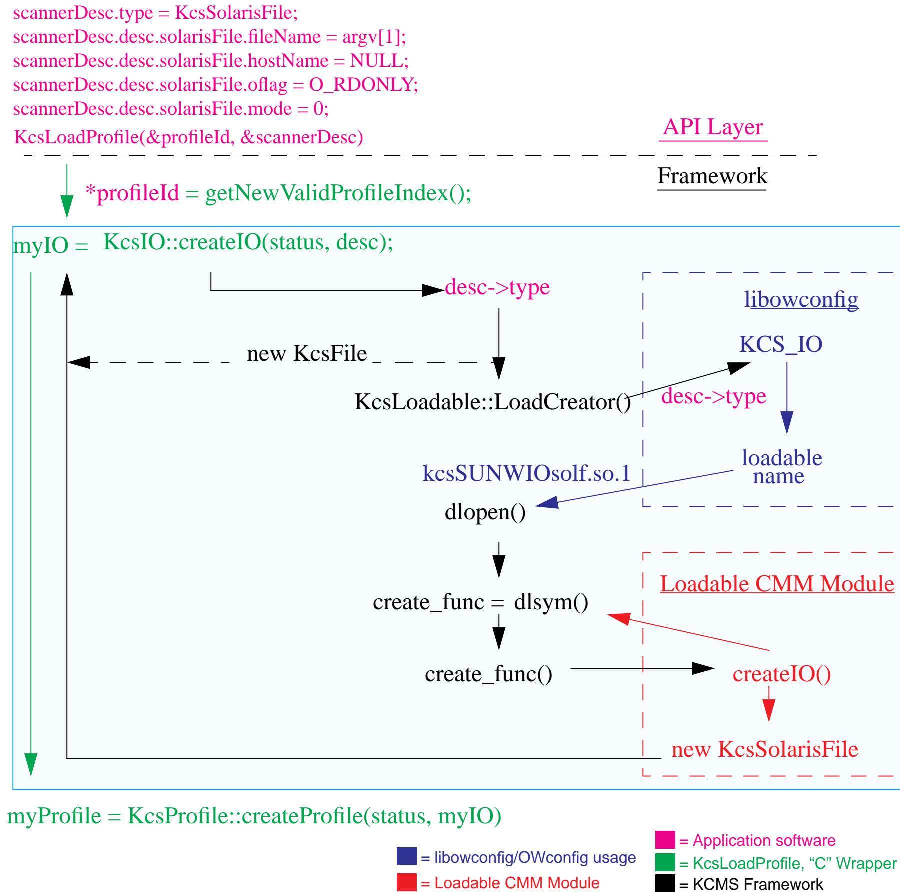

Figure 3–3 Creating a `KcsIO` Object

Creating a `KcsIO` Object

As a result of the call to `KcsLoadProfile()()` the framework creates a `KcsIO` object. Figure 3–3 illustrates how the `KcsLoadProfile()()` API call is implemented. The legend indicates the source of each call (KCMS "C" API layer, framework, "C" wrapper). In addition, the legend shows where the `OWconfig` file and the loadable CMM module fit into the overall scheme of loading the profile to creating a `KcsIO` object.

Looking ahead for a moment, Figure 3–4 , Figure 3–5 , and Figure 3–6 show the progression of framework calls that ultimately load the profile as chunks of data and return the profile Id to the calling application.

Returning now to Figure 3-3, the KCMS framework and the dynamic loading mechanism perform the following task sequence when `KcsLoadProfile()()` is called:

1. The framework gets a new profile Id. The framework maintains a dynamically allocated global array of profiles. The `getNewValidProfileIndex()()` method allocates a new profile entry.
2. The framework creates a `KcsIO` pointer. (All profiles access their data using the `KcsIO` independent access mechanism.) The `KcsIO` pointer is created based on the type field of the `KcsProfileDesc` structure pointer passed in from `KcsLoadProfile()()`.

   Two externally available types are built into the `libkcs` library, `KcsFile` and `KcsMemoryBlock`. A third derivative, `KcsRemoteFile`, is used with the `KcsSolarisFile` and `KcsXWindow` classes. In this particular example, `KcsSolarisFile` is not built into the `libkcs` library, so the dynamic loading mechanism creates one.
3. The dynamic loading mechanism turns the `KcsProfileDesc->type` structure pointer field into a 4-character string and searches entries in the `OWconfig` file for the entries that correspond to loadable `KcsIO` classes. If it finds a match, it dynamically loads the `KcsIO` module. This action supplies the framework with a shared object to load. (See "`KcsIO` Example" on page 27 for details.)

   The `KcsIO` module contains calls to a list of known function names. The framework uses dlsym(3X) to bring these functions into the framework to create and load a pointer to a `KcsIO` derivative.
4. Once the `KcsSolarisFile` object pointer is loaded, the framework uses the `fileName`, `hostName`, and open(2) arguments to search for the profile. First, it checks the `hostName` to see if the file is on a local or remote machine. Depending on the location, the `KcsSolarisFile` object reuses the existing `KcsIO` class derivatives.

   If the file is on a local machine, the `fileName` is opened using open(2), and a `KcsFile` object pointer is created. If, however, the file is located on a remote machine, the `fileName` and `hostName` are passed to `KcsRemoteFile` and a `KcsRemoteFile` object pointer is created.

   As shown in Code Example 3–2 , the `KcsFile` or `KcsRemoteFile` pointer that the `KcsSolarisFile` file object contains is then used to override the `KcsIO` methods of the same name.

CODE EXAMPLE 3–2 Overriding `KcsIO` Methods With `KcsSolarisFile`

```
// Just call myIO

   version of the call KcsStatus KcsSolarisFile::relRead(const long aBytesWanted,

   void *aBuffer, const char *aCallersName) {     KcsStatus status;      status =

   myIO->relRead(aBytesWanted, aBuffer, aCallersName);     return (status);
```

```
}
```

Creating a `KcsProfile` Object

Once a `KcsIO` object has been created, the profile can be loaded. Figure 3–4 illustrates creating a `KcsProfile` object.

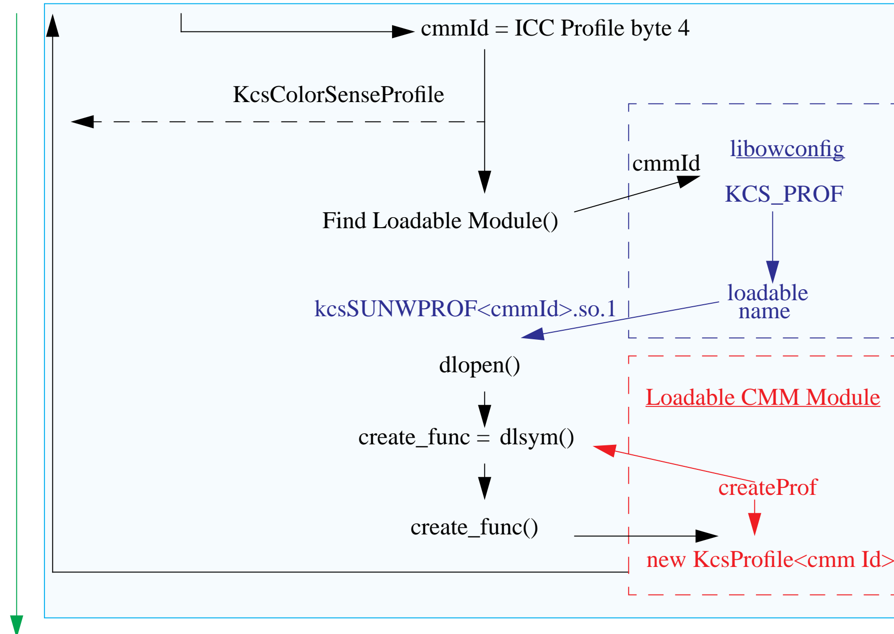

Figure 3–4 Creating a `KcsProfile` Object

In Figure 3-4, the first step to loading the profile is to create a new `KcsProfile` object with the `createProfile()()` static `KcsProfile` method. This method uses the CMM Id of the profile, which is located in a fixed place (bytes 4-7) in the profile header. (See “ICC Profile Header ” on page 34 for details.) The CMM Id determines the `KcsProfile` derivative to be created. If the CMM Id has no corresponding entry in the `OWconfig` file, the default `KcsProfile` class is created.

Creating a `KcsProfileFormat` Object

Once a `KcsProfile` object has been created, you *(???your CMM)*can ask it to load itself using the generated `KcsIO`. Figure 3–5 illustrates the process.

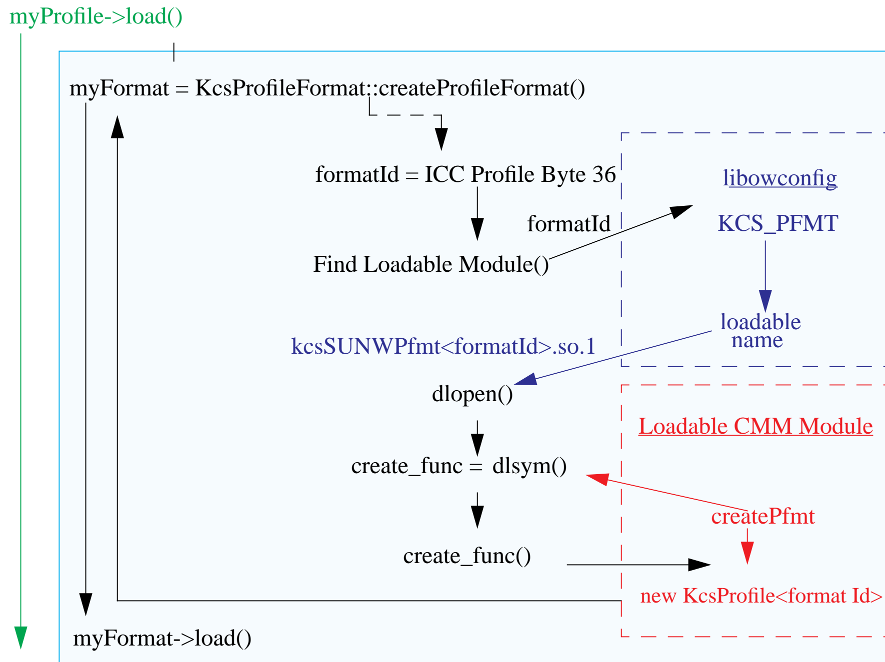

Figure 3–5 Creating a `KcsProfileFormat` Object

In Figure 3–5 , the `KcsProfile` object creates a `KcsProfileFormat` object pointer using `createProfileFormat()()`. Then `createProfileFormat()()` searches the `OWconfig` file for loadable entries based on the profile format Id. For ICC profiles, the profile format Id (also called the profile file signature) is always `acsp`. (See "ICC Profile Header " on page 34 for details.) Once the `KcsProfileFormat` object is created, the library generates a `KcsAttributeSet` object and an array of pointers to `KcsXform` objects.

Loading a `KcsProfileFormat` Object

The pointers to objects contained within the `KcsProfileFormat` object load themselves using the `KcsChunkSet` class. Figure 3–6 illustrates the process.

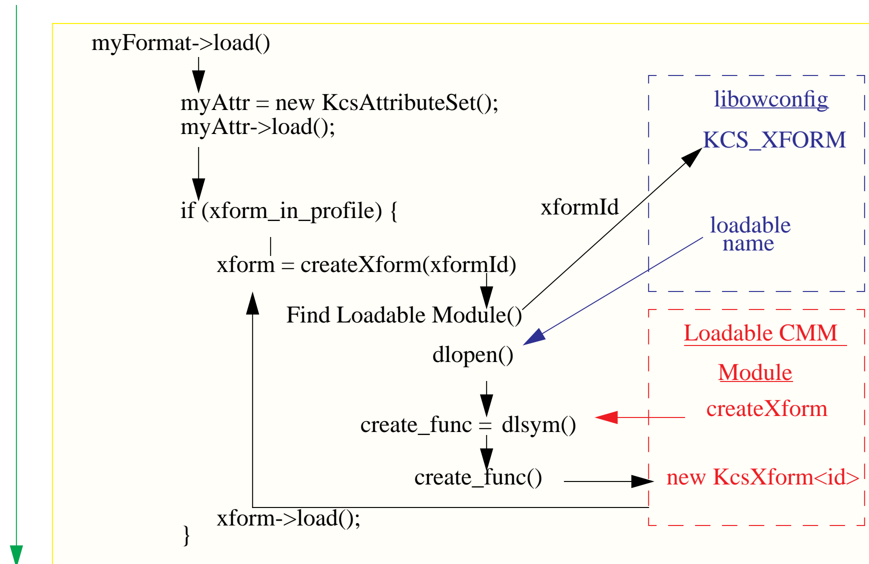

Figure 3–6 Loading a `KcsProfileFormat` Object

In Figure 3–6 , the `KcsChunkSet` class returns the blocks of data from the file, which were requested by the `KcsAttributeSet` and `KcsXform` objects. These objects interpret the block of data, turning it into tables for processing color data or sets of attributes. The `KcsIO` and `KcsChunkSet` classes do not interpret the data.

If the Solaris file system profile is successfully loaded, the framework increments the number of entries in the global profile array, and the profile Id is returned to the application.

Loading an X11 Window System Profile

In the next example, the framework loads a profile associated with a particular X11 Window System visual. The `KcsXWindow` object converts the display, visual, and screen information into a profile loaded into the KCMS framework. Code Example 3–3 is a KCMS "C" API code excerpt that shows this.

CODE EXAMPLE 3–3 Loading an X11 Window System Profile

```
if ((dpy =

XOpenDisplay(hostname)) == NULL) {  fprintf(stderr, ''Couldn't open

the X display \n'');  exit(1); }  profileDesc.type = KcsWindowProfile;

profileDesc.desc.xwin.dpy = dpy; profileDesc.desc.xwin.visual =

DefaultVisual(dpy,     DefaultScreen(dpy)); profileDesc.desc.xwin.screen =
```

```
DefaultScreen(dpy);  status = KcsLoadProfile(&profile, &profileDesc,

   KcsLoadAttributesNow); if (status != KCS_SUCCESS) {  status =

KcsGetLastError(&errDesc);  fprintf(stderr,''KcsLoadProfile failed

error = %s\n'',     errDesc.desc);  exit(1); }
```

The only difference between this example and Code Example 3–2 , is the type of `KcsIO` class loaded. That example showed how to load a `KcsSolarisFile` object rather than a `KcsXWindow` object.

Connecting Two Loaded Profiles

Code Example 3–4 is a "C" API code excerpt that shows how to connect two profiles together once they have been loaded.

CODE EXAMPLE 3–4 Connecting Two Loaded Profiles

```
profileSequence[0] =

scannerProfile; profileSequence[1] = monitorProfile; status =

KcsConnectProfiles(&completeProfile, 2, profileSequence, op,

&failedProfileNum); if (status != KCS_SUCCESS) {  fprintf(stderr,

''Connect Profiles failed in profile number     %d\n'',

failedProfileNum);  KcsFreeProfile(monitorProfile);

 KcsFreeProfile(scannerProfile);  return(-1); }
```

The KCMS framework implements the `K()csConnectProfiles()()` API call as follows:

1. It calls `getNewValidProfileIndex()()`method to get a new valid index for the connected profile from the dynamically allocated global array of profiles.
2. The new connected profile needs a `KcsIO` class to handle its I/O. Currently, this is stored in memory only, so it creates a `KcsMemoryBlock` object.
3. It creates a `KcsProfile` object that can link together sequences of profiles.
4. It attaches each profile in the sequence with `attach()()` to the newly created `KcsProfile` object. The `attach()()`method reference counts the objects. (Note that all classes are reference counted through inheritance from the `KcsShareable` class.)
5. Once attached, the `KcsAttributeSet` object composes the attributes from the two `KcsProfile` members in the array into a single set of attributes for the newly created profile object.

6. It links the `KcsXform` array so that the "into" and "out of" profile connection space (PCS) transforms of each `KcsProfile` object (profile) can be connected. When color data is processed through this sequence, it moves from input profile to PCS and from PCS to output profile.
7. Once connected, it returns the new profile Id to the calling application for later reference, and generally cleans up the classes.

Evaluating Data Without Optimization

The evaluation path of data is different for unoptimized and optimized sequences. Figure 3–7 shows both paths.

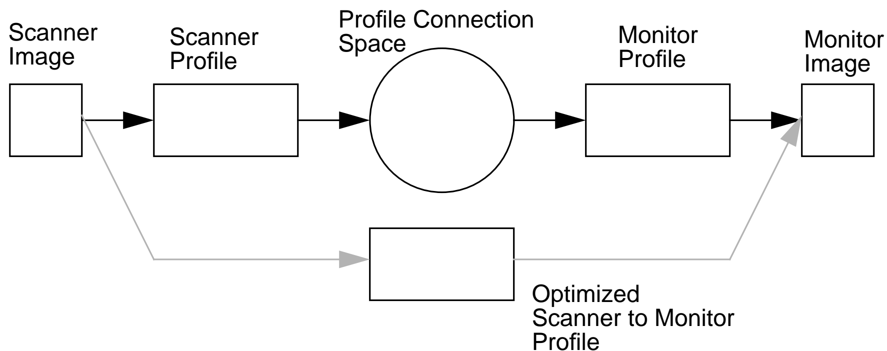

Figure 3–7 Optimized And Unoptimized Evaluation

In the unoptimized case, when `evaluate()()` is called, the color data is moved from input space to PCS and from PCS to output space. This is achieved by passing the data through the appropriate `KcsXform` object in the `KcsXform` object array. The KCMS "C" API code excerpt shown in Code Example 3–5 evaluates data without optimization.

CODE EXAMPLE 3–5 Evaluating Data Without Optimization

```
/* set up the pixel

layout and color correct the image */ if (depth == 24)

 setupPixelLayout24(&pixelLayoutIn, image_in); else

 setupPixelLayout8(&pixelLayoutIn, red, green, blue,     maplength);

status = KcsEvaluate(completeProfile, op, &pixelLayoutIn,

   &pixelLayoutIn); if (status != KCS_SUCCESS) {  fprintf(stderr,

``EvaluateProfile failed\n'');  KcsFreeProfile(monitorProfile);

 KcsFreeProfile(scannerProfile);  KcsFreeProfile(completeProfile);

 return(-1); }
```

Evaluating Data With Optimization

When a profile sequence is optimized for speed, a set of tables is generated that does not require the color data to be passed through the PCS. As a result, the connected profile contains a composed `KcsXform` object that moves data directly from input space to output space. (***Composition*** reduces multiple transforms into a single transform.) The KCMS "C" API code excerpt shown in Code Example 3–6 evaluates data with optimization for speed.

CODE EXAMPLE 3–6 Evaluating Data With Optimization for Speed

```
status =

KcsOptimizeProfile(completeProfile, KcsOptSpeed,     KcsLoadAllNow); if

(status != KCS_SUCCESS) {   fprintf(stderr, ``OptimizeProfile

failed\n'');   KcsFreeProfile(monitorProfile);

KcsFreeProfile(scannerProfile);   return(-1); }  /* set up the pixel layout

and color correct the image */ setupPixelLayout24(&pixelLayoutIn,

image_in);  status = KcsEvaluate(completeProfile, op, &pixelLayoutIn,

   &pixelLayoutIn);  if (status != KCS_SUCCESS) {  fprintf(stderr,

``EvaluateProfile failed\n'');  KcsFreeProfile(monitorProfile);

 KcsFreeProfile(scannerProfile);  KcsFreeProfile(completeProfile);

 return(-1); }
```

Freeing a Profile

Freeing a profile causes each of the objects pointed to by the profile Id in the framework's global array to release all of its associated data. If a given object is a shared or reference-counted object, the memory is released only if the reference count drops to zero.

Freeing a profile loaded via `KcsSolarisProfile` or `KcsXWindowProfile` closes the associated file descriptor or remote procedure call (RPC) connection if the file is located on a remote machine. Use the `KcsFreeProfile(profile)()` KCMS "C" API call to free a profile.

Attributes

The examples below show how to get and set attributes.

Setting an Attribute

When setting an attribute, the `KcsProfile` object in the `KcsProfile` object array passes the setting of the attribute to the `KcsAttributeSet` object contained in its `KcsProfileFormat` object. This is illustrated in Figure 3–2 and in the KCMS API code excerpt shown in Code Example 3–7 .

CODE EXAMPLE 3–7 Setting an Attribute

```
/* double2icFixed

converts a double float to a signed 15 16 fixed point   * number */ /* Set

white point */ test_double[] = 0.2556; test_double[1] = 0.600189;

test_double[2] = 0.097794; attrValue.base.countSupplied = 1

attrValue.base.type = icSigXYZType; attrValue.base.sizeof(icXYZNumber);

attrValue.val.icXYZ.[0].X = double2icfixed(test_double[0],

  icSigS15Fixed16ArrayType); attrValue.val.icXYZ.[0].Y =

double2icfixed(test_double[1],    icSigS15Fixed16ArrayType);

attrValue.val.icXYZ.[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); }
```

Getting an Attribute

When getting an attribute, the `KcsProfile` object in the array passes the getting of the attribute to the `KcsAttributeSet` object contained in its `KcsProfileFormat` object (replacing `set` with `get`). This is illustrated in Figure 3–2 and in the KCMS API code excerpt shown in Code Example 3–8 .

CODE EXAMPLE 3–8 Getting an Attribute

```
/* Get the colorants */

/* icfixed2double converts signed 15.16 fixed point number to a double   *

float */ /*Red */ 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); }  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),
```

Characterization and Calibration

Characterization and calibration are accessed using the following KCMS "C" API calls:

- `KcsCreateProfile()()`
- `KcsUpdateProfile()()`
- `KcsSetAttribute()()`
- `KcsSaveProfile()()`

See the SDK manual *KCMS Application Developer's Guide* for more information on these calls.

The `KcsProfile` base class contains virtual methods to characterize and calibrate two types of devices: scanners and monitors. You must decide whether to override the base functionality to take characterization and calibration data and turn it into the appropriate `KcsXform` data.

Note - Currently, the default CMM supports monitor and scanner characterization and calibration only. It does not support printer characterization and calibration. However enabling hooks exist in the source so you can write a CMM that supports printers.

Attributes are set using the normal mechanisms. The KCMS "C" API code excerpt in Code Example 3–9 shows characterization and calibration.

CODE EXAMPLE 3–9 Characterization and Calibration

```
KcsCalibrationData

       *calData;     KcsCharacterizationData              *charData;

float             Luminance_float_out[3][256];

double             test_double[3];   /* this is a test which does not use
```

```
real data - just a gamma curve for the *  calibration structure and the same
curve *.75 for the characterization curve. */      /*create luminance tables
with a gamma = 2.22 */   for (j=0; j<levels; j++) {    input_val = j *
(1.0/255.0);    Luminance_float_out[0][j] = pow(input_val, 2.22);
   Luminance_float_out[1][j] = pow(input_val, 2.22);
   Luminance_float_out[2][j] = pow(input_val, 2.22);     }     /* 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;     sizemeas =
(int) (sizeof(KcsMeasurementBase) + sizeof(long) +
sizeof(KcsMeasurementSample) * levels);     charData =
(KcsCharacterizationData *) malloc(sizemeas);
charData->base.countSupplied = levels;     charData->base.numInComp = 3;
    charData->base.numOutComp = 3;     charData->base.inputSpace =
KcsCIEXYZ;     charData->base.outputSpace = KcsRGB;     for (i=0; i<
levels; i++) {    charData->val.patch[i].weight = 1.0;
   charData->val.patch[i].standardDeviation = 0.0;
   charData->val.patch[i].sampleType = KcsChromatic;
   charData->val.patch[i].input[KcsRGB_R] = (float)i/255;
   charData->val.patch[i].input[KcsRGB_G] = (float)i/255;
   charData->val.patch[i].input[KcsRGB_B] = (float)i/255;
   charData->val.patch[i].input[3] = 0.0;
   charData->val.patch[i].output[KcsRGB_R] =
(Luminance_float_out[0][i])/0.75;
   charData->val.patch[i].output[KcsRGB_G] =
(Luminance_float_out[1][i])/0.75;
   charData->val.patch[i].output[KcsRGB_B] =
(Luminance_float_out[2][i])/0.75;    charData->val.patch[i].output[3] =
0.0;     }     charData->val.patch[0].sampleType = KcsBlack;
charData->val.patch[255].sampleType = KcsWhite;      sizemeas = (int)
(sizeof(KcsMeasurementBase) + sizeof(long) +
```

```
sizeof(KcsMeasurementSample) * 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;      printf(``Update a
profile with characterization and calibration data.\n'');     rc =
KcsUpdateProfile(profileid, charData, calData, NULL);     if(rc !=
KCS_SUCCESS) {         KcsGetLastError(&errDesc);         fprintf(stderr,
               ``unable to update profile: %s\n'', errDesc.desc);
     KcsFreeProfile(profileid);         return(-1);     }
```

Saving a Profile to the Same Description

Saving a profile to the same description is the same as loading in reverse. Each object pointed to or contained within the `KcsProfile` object is instructed, with its own save mechanisms, to write the data needed to reconstruct itself out to static store. In this case, the description is identical to that used to load the profile, so the current `KcsIO` associated with the profile is used. Code Example 3–10 is a KCMS "C" API code excerpt that saves a profile to the same description.

CODE EXAMPLE 3–10 Saving a Profile to the Same Description

```
status

= KcsSaveProfile(profileid, NULL); if(status != KCS_SUCCESS) {  status =

KcsGetLastError(&errDesc);  printf(``SaveProfile error: %s\n'',

errDesc.desc); }
```

Saving a Profile to a Different Description

To save a profile to a different description, load a new `KcsIO` so that the `KcsProfile` object can save itself. This is accomplished with the same mechanism as that described in steps 2 to 5 of "Creating a `KcsIO` Object" on page 45 . Code Example 3–11 is a KCMS "C" API code excerpt that saves a profile to a different description.

CODE EXAMPLE 3–11 Saving a Profile to a Different Description

```
/* Application opens the

file */ if ((sfd = open(argv[2], O_RDWR|O_CREAT, 0666)) == -1) {  perror

(``save open failed'');  exit (1); }  desc.type = KcsFileProfile;

desc.desc.file.openFileId = sfd; desc.desc.file.offset = 0; status =

KcsSaveProfile(profileid, &desc); if(status != KCS_SUCCESS) {  status =

KcsGetLastError(&errDesc);  printf(``SaveProfile error: %s\n'',

errDesc.desc); }
```

CHAPTER **4**

KcsIO Derivative

In This Chapter

This chapter discusses the following topics to help you create a KcsIO class derivative that is dynamically loadable at runtime:

- External entry points with an example
- Member function override rules
- Pointer to the KcsSolarisFile class source code to use as an example of a KcsIO derivative

Figure 4–1 shows the relationship of the KcsIO class to the parent classes in the KCMS class hierarchy. See Figure 1–2 for an illustration of all the relevant KCMS classes.

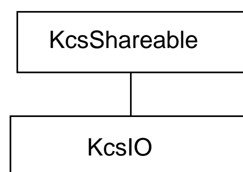

Figure 4–1 KcsIO Derivative

External Entry Points

The KCMS framework uses external entry points to load your derivative as an executable. The mandatory and optional entry points are described.

Mandatory

When you derive from a `KcsIO` class, the mandatory external entry points are:

```
extern

long KcsDLOpenIOCount; KcsIO *KcsCreateIO(KcsStatus *aStat,  const

KcsProfileDesc *aDesc);
```

The `KcsCreateIO()()` method creates an instance of a `KcsIO` derivative. The instance is determined by `aDesc->type`, which contains the derivative portion of the class identifier (see "Creating `OWconfig` File Entries" on page 27).

Optional

When you derive from a `KcsIO` class, the optional external entry points are:

```
KcsStatusId

KcsInitIO(); KcsStatusId

KcsCleanupIO();
```

Example

The following example shows you how to use the entry points when creating a `KcsIO` derivative.

CODE EXAMPLE 4–1 `KcsIO` Class Entry Points Example

```
/* External loadable

interface */ extern ''C''  extern long    KcsDLOpenIOCount;

 KcsStatus    KcsInitIO();  KcsIO    *KcsCreateIO(KcsStatus *aStatus,

      const KcsProfileDesc *aDesc);  KcsStatus    KcsCleanupIO(); //Loadable
```

```
stuff //external DL open count to support runtime derivation extern long
KcsDLOpenIOCount = 0;  /* Runtime derivable routine */ KcsIO *
KcsCreateIO(KcsStatus *aStat, const KcsProfileDesc *Desc) {  //Create the new
derivative  return(new KcsSolarisFile(aStat,
aDesc->desc.solarisFile.fileName,    aDesc->desc.solarisFile.hostName,
  aDesc->desc.solarisFile.oflag, aDesc->desc.solarisFile.mode); }
KcsStatus KcsInitIO(long libMajor, long libMinor, long *myMajor, long
*myMinor) {  // Set up the return values  *myMajor = KCS_MAJOR_VERSION;
 *myMinor = KCS_MINOR_VERSION;   //Check the major version  if (libMajor !=
KCS_MAJOR_VERSION)   return (KCS_CMM_MAJOR_VERSION_MISMATCH);   //Currently,
if minor version of library is less than the KCMS   // minor version, return
an error.  if (libMinor != KCS_MINOR_VERSION)   return
(KCS_CMM_MINOR_VERSION_MISMATCH);   //Library guarantees if your minor version
number is greater than  //KCMS minor version number, it will handle it. No
more init.  return(KCS_SUCCESS); }  KcsStatus KcsCleanupIO() {  /* Clean up is
performed in the destructor */  return (KCS_SUCCESS);
}
```

Member Function Override Rules

The following table tells you which `KcsIO` member functions you must override and can override when deriving from this class. The member functions indicated with an "X" in the Must column are required to successfully derive from this base class. All of these member functions are defined in the `kcsio.h` header file and the *KCMS CMM Reference Manual*.

TABLE 4–1 `KcsIO` Member Function Override Rules

Member Function	Override Rules: Must	Override Rules: Can
`getEOF()()`	X	
`getType()()`	X	
`isEqual()()`	X	
`KcsIO()()`	X	
`~KcsIO()()`		X
`relRead()()`	X	
`relWrite()()`	X	
`setCursorPos()()`	X	
`setEOF()()`	X	
`setOffset()()`		X

Examples To Help You Create Your KcsIO Derivitave

The `KcsSolarisFile` class is a derivative of the `KcsIO` class. You can use any of the `KcsSolarisFile` files (or any of the other `KcsIO` derivatives) as good sources of example code for creating your `KcsIO` derivative. You can find the files on-line in `/usr/opt/SUNWddk/kcms/src`. The `kcssolfi.cc` and `kcssolfi.h` files in this directory are actual SunSoft source code that supports enhanced file access on Solaris. This source code is directly tied into the `kcstypes.h` header file. The

`kcssolfitest.h` header file explains how to include this derivative without changing the KCMS header file.

CHAPTER 5

KcsProfile Derivative

In This Chapter

This chapter discusses the following topics to help you create a KcsProfile class derivative that is dynamically loadable at runtime:

- External entry points with an example
- Member function override rules
- Helpful information on attributes and the KcsProfileFormat instance

Figure 5–1 shows the relationship of the KcsProfile class to the parent classes in the KCMS class hierarchy. See Figure 1–2 for an illustration of all the relevant KCMS classes.

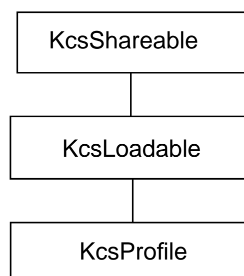

Figure 5–1 KcsProfile Derivative

External Entry Points

The KCMS framework uses external entry points to load your derivative as an executable. The mandatory and optional entry points are described.

Mandatory

When you derive from a `KcsProfile` class and create a `KcsProfile` instance you must provide these mandatory external entry points:

```
extern

long KcsDLOpen ProfCount; KcsProfile * KcsCreateProf(KcsStatus *sStat, KcsIO

*aIO); KcsProfile * KcsCreateProBlnk(KcsStatus *aStat, KcsId aCmmID,

 KcsVersion aCmmVersion, KcsId aProfId,  KcsVersion

aProfVersion);
```

The `KcsCreateProf()()` entry point creates an instance of a `KcsProfile` derivative that is determined by the profile's CMM Id within `aIO`.

The `KcsCreateProfBlnk()()` entry point creates an instance of a `KcsProfile` derivative that is determined by `aCmmID` and `aCmmVersion`. This is how an empty profile is created from scratch. The `aProfId` argument specifies which `KcsProfileFormat` derivative to use.

Optional

When you derive from a `KcsProfile` class, the optional entry points are:

```
KcsStatusId

KcsInitProf(); KcsStatusId

KcsCleanupProf();
```

Example

Code Example 5–1 shows you how to use the entry points when creating a `KcsProfile` instance.

CODE EXAMPLE 5–1 `KcsProfile` Class Entry Points Example

```
/* Required entry points
*/ extern long KcsDLOpenProfCount = 0;  /* Construct a profile object using
KcsIO */ KcsProfile * KcsCreateProf(KcsStatus *aStat, KcsIO *aIO) {  //Create
the new derivative  return (new KcsProfileKCMS(aStat, aIO)); } /* Construct an
in-memory profile object using the ids */ KcsProfile *
KcsCreateProfBlnk(KcsStatus *aStat, KcsId aCmmId,   KcsVersion aCmmVersion,
KcsId aProfId,   KcsVersion aProfVersion) {  //Create the new derivative
 return(new KcsProfileKCMS (aStat, aCmmId, aCmmVersion,    aProfId,
aProfVersion)); } /* Optional entry points */ KcsStatus KcsInitProf(long
libMajor, long libMinor, long *myMajor,    long *myMinor) {  // Set up the
return values  *myMajor = KCS_MAJOR_VERSION;  *myMinor = KCS_MINOR_VERSION;
 //Check the major version  if (libMajor != KCS_MAJOR_VERSION)   return
(KCS_CMM_MAJOR_VERSION_MISMATCH);   //Currently, if minor version of library
is less than the KCMS   // minor version, return an error.  if (libMinor !=
KCS_MINOR_VERSION)   return (KCS_CMM_MINOR_VERSION_MISMATCH);   //Library
guarantees if your minor version number is greater than  //KCMS minor version
number, it will handle it. No more init.  return(KCS_SUCCESS); }  KcsStatus
KcsCleanupProf() {  /* Clean up is performed in the destructor */  return;
}
```

Member Function Override Rules

The following table tells you which `KcsProfile` member functions you must override and can override when deriving from this class. The member functions indicated with an “X” in the Must column are required to successfully derive from this base class. All of these member functions are defined in the `kcsprofi.h` header file and the *KCMS CMM Reference Manual*.

TABLE 5–1 `KcsProfile` Member Function Override Rules

Member Function	Override Rules		
	Must	Can	Do Not
`connect()()`		X	
`createEmptyProfile()()`		X	
`evaluate()()`		X	
`getAttribute()()`		X	
`initDataMember()()`		X	
`isColorSenseCMM()()`		X	
`KcsProfile()()`	X		
`~KcsProfile()()`		X	
`load()()`		X	
`optimize()()`		X	
`propagateAttributes2Xforms()()`		X	
`save()()`		X	
`save()()`		X	
`setAttribute()()`		X	
`setOpAndCont()()`		X	
`setTimeAttribute()()`		X	

TABLE 5–1 `KcsProfile` Member Function Override Rules *(continued)*

Member Function	Override Rules		
	Must	Can	Do Not
`setXform()()`		X	
`unload()()`		X	
`updateMonitorXforms()()`		X	
`updatePrinterXforms()()`		X	
`updateScannerXforms()()`		X	
`updateXforms()()`		X	
`XformIsNOP()()`		X	

Attribute Sets

Attributes can include the following information:

- Profile's manufacturer name
- Input and output color spaces
- Calibration data
- Device's colorimetric information (for example, monitor's white point, 8- or 16-bit lookup table data)

The attribute set is represented by the `KcsAttrAttributesSet` object. The `KcsProfile` class `getAttribute()()` and `setAttribute()()` methods map directly to the `KcsAttrAttributesSet` object's `getAttribute()()` and `setAttribute()()` methods.

Before performing any operation, the `KcsProfile` base class loads what is necessary for that operation. For example, the `getAttribute()()` method always

loads the attribute set before it accesses the `KcsAttributeSet` instance. It also tries to unload data based on the unload hints you supplied.

The base class overrides the following list of attribute set values with the `getAttribute()()` and `setAttribute()()` methods:

- Attribute number
- Attribute set
- Profile length
- Pixel layout supported
- Supported operations
- CMM version
- ICC profile version

If `getAttribute()()` or `setAttribute()()` intercepts one of these attributes, it does not use the `KcsAttributeSet` class. Instead it uses a `KcsProfile` class derivative; otherwise, it is passed to the `KcsAttrAttributesSet` object.

`KcsProfileFormat` Instance

In addition to overriding attribute set values, the `KcsProfile` base class uses a `KcsProfileFormat` instance. (See the protected `KcsProfile` pointer `iFormat` in the `kcsprofi.h` header file.) `KcsProfile` uses the `KcsProfileFormat` object to:

- Make the version number of profile data in static store transparent
- Provide support for special profile formats a CMM might need
- Allow compatibility with new and old supported profile formats

The `KcsProfileFormat` object supports a consistent interface to attributes and transformations as objects. For example, if the profile format is the ICC format, the derivative of the `KcsProfile` class can use the `KcsProfileFormat` derivation supplied with the KCMS framework.

Transformations

Transformations are represented by an array of pointers to instances of the `KcsXform` class hierarchy. This array is indexed by the `enum` type `KcsXformType`.

The logical transformation types are listed in Table 5–2 .

Note - In Table 5-2 , RCS refers to *reference color space*. This Kodak term is equivalent to profile connection space (PCS).

TABLE 5–2 Logical Transformation Types

KcsXformType **Values**	**Logical Transformation Type**	**Description**
KcsSftIntoRCS = 0	into-RCS	Input color space to the output color space which can be one of the standard references.
KcsXftOutofRCS = 1	outof-RCS	Output color space (possibly one of the standards) to the input color space.
KcsSftFwdEffect = 2	forward-RCS-effect	An effect that goes from a color space to that same color space.
KcsXftRvsEffect = 3	reverse-RCS-effect	Inverse of forward-RCS-effect.
KcsXftFwdSimulate = 4	simulation-RCS	Special processing done to device's output, if simulation is desired on another device. If there is not a simulation-RCS transformation, the KCMS framework defaults to connecting the outof-RCS to the into-RCS transformations, which generates an RCS-to-RCS transformation that approximates the simulation. This results in a clip close to the simulation normally seen on devices. Currently, profiles do not supply simulation-RCS transformations by default. This connection technique fails on profiles that perform gamut-mapping.
KcsXftFwdGamut = 5	gamut-test-RCS	Provides all gamut testing for the device. The gamut-test-RCS transformation output is a set of 8-bit values representing how much this particular data is out of gamut for all ***n*** components.
KcsXftFwdComplete = 6	complete-forward transform	Profile's transformation from a source device to a destination device. It includes any intermediate effects connected to the chain.
KcsXftRvsComplete = 7	complete-reverse transform	Goes from the destination device backwards through any of the inverse effects and then into the color space of the input device.

TABLE 5–2 Logical Transformation Types *(continued)*

`KcsXformType` **Values**	**Logical Transformation Type**	**Description**
`KcsXftRcsSimulate = 8`	complete-simulate transform	Maps the pixels from the source device through the destination device, its simulationtransformation, and ultimately to the destination device (the device on which you are viewing the data).
`KcsXftRCSGamut = 9`	complete-gamut transform	Gamut test for whole chain.

The set of types in Table 5–2 refer to RCS. However, the KCMS framework supports a non-RCS model where there is no RCS. The only mandate is that CMMs support the standard references (CIEXYZ and CIELAB) and that the color spaces match between connections. (Even this mandate is not strictly applied since the `KcsProfile` base class `connect()()` method automatically inserts profiles if that creates a match).

Instead of referring to the into-RCS transformation as going from a non-RCS into a RCS, the ICC specification describes a transformation from an input color space to an output color space. The output may be one of the standard references if it is a device profile.

Transformation Type Methods

The methods supplied to the `evaluate()()` method of the derived class choose which `KcsXform` instance to use. The `KcsForward()()`, `KcsReverse()()`, `KcsSimuate()()`, and `KcsGamutTest()()` methods map directly to the corresponding complete transformation types.

Constructors and Destructors

`KcsProfile` includes two types of constructors: an *I/O-based* constructor and an *identifier-based* constructor.

The I/O-based constructor takes something that is out in static store and instantiates the profile based on the data contained within it. That I/O object can represent file, memory, network, or any other I/O derivative. This relates back to the save methods where the state is saved through an I/O object. This constructor generates a `KcsProfile` derivative from a saved state.

The identifier-based constructor indicates the CMM Id, CMM version, `ProfileId`, and the profile version. This constructor allows creation of an empty profile and

determines which CMM to use, which profile format to use, and which CMM version to use. They are defaulted to create the latest ICC CMM, with the latest KCMS profile format version.

Both constructors allocate or create a profile format object. Then they take the `ChunkSet` of that profile format object and use that to set their own `ChunkSet`. This is how `KcsProfile` and `KcsProfileFormat` objects link their `KcsChunkSet` objects during construction. This happens in the base class, so derivatives do not need to do this unless they have special requirements such as requiring a special derivation of the `ChunkSet` object.

Creation Methods

Each profile constructor corresponds to two `createProfile()()` methods. Both methods use the runtime mechanism in the `KcsLoadable` class to dynamically load themselves at runtime. During creation of an identifier-based profile, a `createProfile()()` method automatically loads the runtime module, which allocates the correct derivative. It then calls `createEmptyProfile()()` for initialization.

Save Methods

`KcsProfile` includes two types of `save()()` methods: a ***blind*** save and an ***I/O-based***save.

The blind `save()()` method saves the profile to the current location. No arguments are required and the `timelastsaved` attribute is set. The `iFormat` `save()()` is called. (`iFormat` is a pointer to the `KcsProfileFormat` instance in the KcsProfile base class. See “`KcsProfileFormat` Instance” on page 70 for details.) The `KcsProfileFormat` class saves the profile (with `iFormat`) because only that class knows the content of the data.

The I/O-based `save()()` method constructs a new chunk set by

1. Replacing the one currently there
2. Doing a blind save to the new chunk set
3. Resetting everything back

This save method also creates an empty profile by calling the `createEmptyProfile()()` method.

Since the I/O-based save means save the data to something different, or something new, `save()()` must reset all the default data to a loadable empty profile with no attributes and no transformations.

All `load()()` and `save()()` methods are based on chunk sets. All chunk sets are based on I/O objects. Therefore, indirectly, `save()()` uses the I/O object to get its data from static store. See Chapter 4 ," for details.

The I/O-based `save()()` is not virtual because it just wraps around the virtual blind `save()()` method.

Using `connect()()`

The `connect()()` method is one of the most demanding methods of the `KcsProfile` class. It uses all of the profiles that are in the list `sequence`, as well as the `opAndHints` operation, to determine which transformations to generate and how to generate them. It also checks a number of internal rules to ensure those connections are possible. For example, it checks to see that color spaces are compatible. The KCMS profile model does not include a specific reference. Some color management solutions do, but the KCMS framework checks the consistency of the connections. As illustrated in Figure 5–2 , the connected sequence shares the Xforms (through the class) of those in the list.

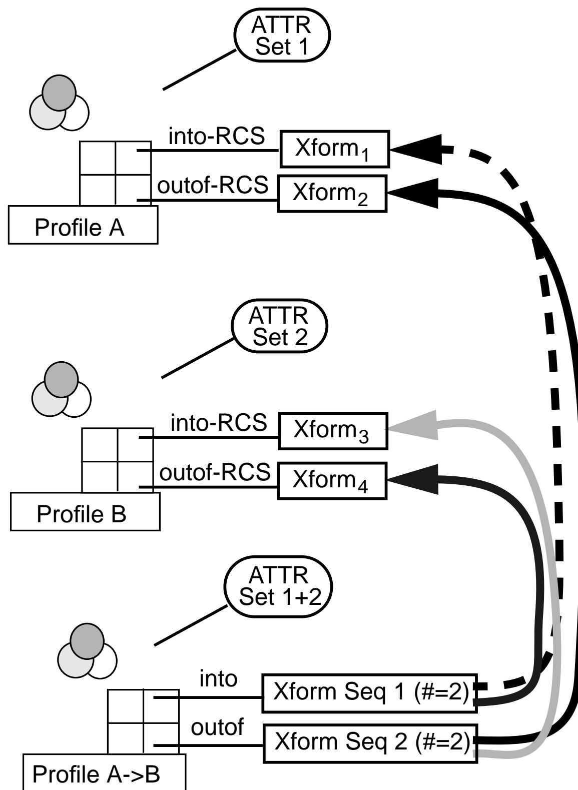

Figure 5–2 Sequences Sharing Xforms

The KCMS framework also does automatic insertion of profiles if those color spaces do not match properly. For example, if you want to connect a KCMS profile that uses KCMS RGB as its connection space with an XYZ profile that uses CIE Lab as its connection space, `connect()()` looks for a KCMS RGB-to-CIE Lab profile and automatically inserts it into the list. The rest of the connection proceeds as if that profile was in the list when called.

The `connect()()` method searches through all KCMS profiles so more can be installed to add to the flexibility of this mechanism. opAndHints allows you to trim result to contain only the operations wanted for future use. For example, if you only want to perform a forward operation, then supply only `KcsForward`–even if there is enough information to connect and create a reverse operation. The default behavior in the base class is to not create the `KcsReverseOp` transformation. The method only creates what you tell it to create. The only exception is attributes. All profiles need attributes; otherwise, `result` is useless. It is recommended that derivatives keep consistent with this policy.

When the base `KcsProfile` connects profiles and creates a new one, it does not create a connection of profiles. One profile is generated with the `KcsProfile` elements: attributes, transformations, and a format object. The `connect()()` method uses sequences of transformations to fill in the `results` transformation array. The sequences are generated based on the content of the profile in the list `sequence`.

As shown in Figure 5–2 , if you give a list consisting of an input device with an output device as the only profiles listed, the `connect()()` method takes the into-RCS transformation of the input device, connects it with the outof-RCS transformation of the output device, and creates a sequence. The method then assigns that sequence to the complete-Forward index of a `result`. If `KcsForwardOp` is the only operation specified in `opAndHints`, that is the only sequence generated. Figure 5–2 also illustrates `KcsReverseOp`.

The `connect()()` method also goes through a composition of all the attributes in all the profiles in the `sequence` list. A set of attribute rules, a `composition` method in the `KcsAttributeSet` class, and the base class `connect()()` method feed the list of attributes from profile objects in the list to the `KcsAttributeSet` `composition` method.

By default, when connecting the simulation transformations for the resulting profile, the `connect()()` method looks for the simulation-RCS transformation to accomplish the simulation part of the chain. If `connect()()` doesn't find one and the outof-RCS and into-RCS transformations of the simulated device are available, it makes a simulation sequence that contains these transformations in place of the simulation-RCS transformation. This results in a clipping close to the simulation normally seen on devices.

Examples

Use Figure 5–3 with the two examples described below to better understand the `connect()()` method and RCS simulation.

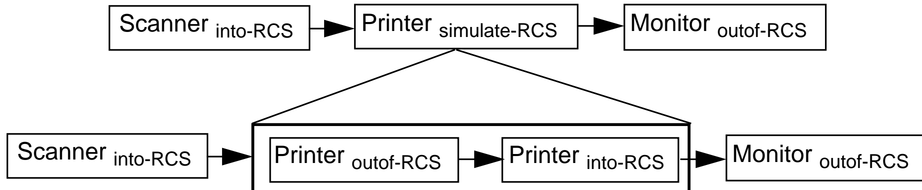

Figure 5–3 Into- and Out-of-RCS Transformations

With Printer RCS Transformation

Three profiles exist in the `sequence` list: a scanner, a printer, and a monitor. A value of (`KcsForwardOp|KcsSimulateOp`) for `opAndHints` indicates to the `KcsProfile::connect()()` method that data in the scanner color space is ready to go into the printer color space and then transform so it can be previewed on the monitor supplied. The complete simulation transformation is a sequence of the into-RCS transformation of the scanner profile, the simulate-RCS transformation of the printer profile, and the outof-RCS transformation of the monitor profile.

Without Printer RCS Transformation

If RCS simulation is not available in the printer profile, the `connect()()` method connects the transformations by first connecting the into-RCS transformation of the scanner profile to the outof-RCS transformation of the printer profile, then to the into-RCS transformation of the printer profile, and then to the outof-RCS transformation in the monitor profile. Note that for the printer simulation transformation, the simulate-RCS is replaced with the combination outof-RCS and into-RCS transformations. This clips color to the simulated device.

If the simulated profile has neither the simulate-RCS nor the into-RCS and outof-RCS combination, `connect()()` returns a `KCS_NOT_ENOUGH_DATA_4_OP` error.

Characterization and Calibration

Characterization and calibration are handled through the update methods of the `KcsProfile` class, namely

- `updateMonitorXforms()()`
- `updatePrinterXforms()()`

- `updateScannerXforms()()`
- `updateXforms()()`

These methods either characterize or calibrate, depending on the content of those tables.

The base class implementation of `updateMonitorXforms()()`, `updatePrinterXforms()()`, `and()` `updateScannerXforms()()` returns errors indicating that this particular profile does not support calibration or characterization, depending upon which data are supplied. The `KcsProfile` base class implements `updateXforms()()` to provide some general-purpose device typing, yet executes the device-specific update mechanism in the derivations. This allows you to put your characterization and calibration techniques in your `KcsProfile` derivative while using the base class to determine which type of device is actually being updated.

CHAPTER **6**

KcsProfileFormat Derivative

In This Chapter

This chapter discusses the following to help you create a KcsProfileFormat class derivative that is dynamically loadable at runtime:

- External entry points with an example
- Member function override rules
- Helpful information on attributes, transformations, loading and what you need to consider with a protected KcsProfileFormat derivative

Figure 6–1 shows the relationship of the KcsProfileFormat class to the parent classes in the KCMS class hierarchy. See Figure 1–2 for an illustration of all the relevant KCMS classes.

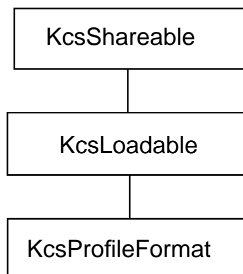

Figure 6–1 KcsProfileFormat Derivative

External Entry Points

The KCMS framework uses external entry-points to load your derivative as an executable. The mandatory and optional entry points are described.

Mandatory

When you derive from a `KcsProfileFormat` class, the mandatory external entry points are:

```
extern

long KcsDLOpenPfmtCount; KcsProfileFormat *KcsCreatePfmt(KcsStatus *sStat,

KcsIO *aIO); KcsProfileFormat *KcsCreatePfmtBlnk(KcsStatus *aStat,  KcsId

aCmmId, KcsVersion aCmmVersion, KcsId aProfId,  KcsVersion

aProfVersion);
```

`KcsCreatePfmt()()` creates an instance of a `KcsProfileFormat` derivative. The derivative is determined by the version information contained within the data represented by `aIO`. This corresponds to the `KcsLoadProfile()()` call.

`KcsCreatePfmtBlnk()()` creates an instance of a `KcsProfileFormat` derivative that is determined by `aProfId`. This creates a blank profile version instance with no objects associated with the returned instance. It initializes the CMM type identifier, the CMM version, and the profile version from `aCmmId`, `aCmmVersion`, and `aProfVersion`, respectively.

Optional

When you derive from a `KcsProfileFormat` class, the optional "C" based entry points are:

```
KcsStatus

KcsInitPfmt(); KcsStatus

KcsCleanupPfmt();
```

Examples

The following example shows you how to use the entry points when creating a `KcsProfileFormat` derivative.

CODE EXAMPLE 6–1 `KcsProfileFormat` Class Entry Points Example

```
extern long

KcsDLOpenPfmtCount = 0;  /* Global initialization - constructor can be used */

KcsStatus KcsInitPfmt(long libMajor, long libMinor, long *myMajor, long

*myMinor) {  // Set up the return values  *myMajor = KCS_MAJOR_VERSION;

 *myMinor = KCS_MINOR_VERSION;   //Check the major version  if (libMajor !=

KCS_MAJOR_VERSION)   return (KCS_CMM_MAJOR_VERSION_MISMATCH);   //Currently,

if minor version of library is less than the KCMS   // minor version, return

an error.  if (libMinor != KCS_MINOR_VERSION)   return

(KCS_CMM_MINOR_VERSION_MISMATCH);   //Library guarantees if your minor version

number is greater than  //KCMS minor version number, it will handle it. No

more init.  return(KCS_SUCCESS); }  /* Clean up global initialization */

KcsStatus KcsCleanupPfmt() {  KcsStatus sStat;  return(KCS_SUCCESS); }  /*

Create a profile format derivative based on information passed in,  * there is

profile data associated with it. Corresponds to the  * KcsCreateProfile() API

call. */ KcsProfileFormat * KcsCreatePfmtBlnk(KcsStatus *aStat, KcsId aCmmId,

 KcsVersion aCmmVersion, KcsId aProfId, KcsVersion aProfileVersion) {

 //Create the new derivative  return(new KcsProfileFormatInterColor3_0(aStat,

aCmmId,    aCmmVersion, aProfId, aProfileVersion)); }  /* Create a profile

format derivative using the supplied IO.  * Corresponds to the

KcsLoadProfile() API call. */ KcsProfileFormat * KcsCreatePfmt(KcsStatus

*aStat, KcsIO *aIO) {  //Create the new derivative  return(new

KcsProfileFormatInterColor3_0(aStat, aIO));

}
```

Member Function Override Rules

The following table tells you which `KcsProfileFormat` member functions you must override and can override when deriving from this class. The member functions indicated with an "X" in the Must column are requiredto successfully derive from this base class. All of these member functions are defined in the `kcspfmt.h` header file and the *KCMS CMM Reference Manual*.

TABLE 6–1 `KcsProfileFormat` Member Function Override Rules

	Override Rules	
Member Function	**Must**	**Can**
`deleteXform()()`		X
`dirtyAttrCache()()`		X
`dirtyXformCache()()`		X
`generateLoadWhat()()`		X
`generateXformAttributes()()`		X
`getObject()()`		X
`getObject()()`		X
`getSaveSize()()`		X
`initEmptyFormat()()`		X
`isSupported()()`		X
`KcsProfileFormat()()`	X	
`~KcsProfileFormat()()`		X

TABLE 6–1 `KcsProfileFormat` Member Function Override Rules *(continued)*

	Override Rules	
Member Function	**Must**	**Can**
`load()()`		X
`loadObjectMap()()`		X
`postAttrCompose()()`		X
`save()()`		X
`saveNew()()`		X
`saveObjectMap()()`		X
`setCmmId()()`		X
`setObject()()`		X
`unload()()`		X
`unloadWhenMatch()()`		X

Attributes

All attributes of a profile are in the `KcsAttributeSet` object returned from the `getObject()()` method. (Note that `KcsProfileFormat` includes two public `getObject()()` methods. The method discussed here gets the `KcsAttributeSet`: the other, discussed in "Transformations " on page 84 "below, returns a `KcsXform`.) This attribute instance includes public attributes from all of the formats, regardless of where they reside in the data store. The returned attributes' object of this class contains all of the attributes that describe this profile–including all attributes in the public sections of the various profile formats as well as any private attributes.

Do not confuse these attributes with those associated with individual transformations. Attributes associated with individual transformations are stored in the profile's data store but are not added to the attributes object returned from the getObject()() method of this class. Since this class uses the KcsChunkSet object, it can separate out these attributes from the transformation attributes. The KcsProfile base class informs the KcsXforms it loads about any attributes needed to construct themselves.

Transformations

As many transformation slots exist as there are valid operations for a profile. Use the getObject(KcsXformType)() overloaded method to retrieve the correct KcsXform*. (KcsProfileFormat includes two public getObject()() methods. Use the one that returns the specified KcsXform.)

Like attributes, it does not matter whether the KcsXform wanted is in a public header or in a private part of the profile. This class abstracts out the placement and type. A profile format with a mixture of public and private parts for transformation representation is supported.

Loading

Like most loadable derivatives, use the load()() method to force a specific load state. It takes the hints supplied and loads the instance based on those hints. The KcsProfileFormat class loads those objects and any supporting data returned from the getObject() methods.

If a request is made that requires the loading of unloaded state, the instance goes to static store and loads what is required to accomplish the request. It is up to you whether the objects loaded stay loaded between getObject()() calls.

You can cache any object returned from the getObject()() methods. This means that the load()() and save()() methods must be propagated to the cached object. Since this class keeps its own cache of objects and it is expected to be optimized, let KcsProfileFormat handle all caching and call getObject()() before that particular object is needed.

Error Protocols

The load()() method can take a loadhint with multiple bits set. The following error protocols are used:

- If the only error the derivative receives during load is KCS_PFMT_NO_DATA_SUPPORT_4_REQUEST and everything loads successfully, return KCS_SUCCESS.

  Say, for example, you ask for a forward complete and the derivative also tries to load the reverse complete (for optimization). If the reverse receives an error but forward succeeds, KCS_SUCCESS is returned.

- If nothing requested is available, return KCS_PFMT_NO_DATA_SUPPORT_4_REQUEST.

- Any other errors that occur should be returned, and the object should unload itself before exiting. A failure during load()( ) (other than KCS_PFMT_NO_DATA_SUPPORT_4_REQUEST) results in an object for which the loadhints requested are unloaded.

Protected Derivatives

Differences in physical profile formats is hidden by abstracting the physical pieces of all profile parts into a standard set of objects that represent them. This can be a problem when a new profile format contains a new part that cannot be represented by any of the objects. It is neither a transformation nor an attribute.

If the new derivation can support the existing objects, you can use a new derivation with getObject()( ). If you need a new object type, see if the derivation supports the new object. Any new profile format that supports this new object is derived from that new format derivative instead of the base class.

If you use the KcsChunkSet class appropriately in your new derivative, implemention is minimal. The base class allows the minimum derivative to only override the save()( ) method by having the derivative assign chunks with hard-coded offsets for the pieces of a profile during save()( ). Then load()( ) automatically loads the pieces from the hard-coded offsets by using the chunk set mechanism. If, however, your derivative's pieces are split into smaller pieces, you must override load()( ) to gather the smaller pieces into the original objects.

Base Class Support

The base class supports the caching objects and transformation map saving. It contains the KcsXform * array, an KcsAttribute * and supports the getObject()( ) and setObject()( ) overloads. Most derivative profile formats implement these virtuals: initEmptyFormat()( ), isSupported()( ), load()( ), and save()( ).

A derivative can define and use the base class data elements protected during load()(), and the base class passes them to you.

Retrievable Objects

To find out if a part of an instance is supported, the derivative needs to support the pure virtual method isSupported(KcsLoadHint). This method returns KCS_SUCCESS for only its loadable parts and, if the request is not supported. It takes a loadhint that indicates what can be returned from all getObject()() overloads; this includes whether the forward xform is supported as well as any new parts.

An unsupported object is represented by a NULL in one of the object pointers (attributes or xforms) and it returns KCS_PFMT_NO_DATA_SUPPORT_4_REQUEST.

CHAPTER **7**

KcsXform Derivative

In This Chapter

This chapter discusses the following topics to help you create a KcsXform class derivative that is dynamically loadable at runtime:

- External entry points with an example
- Member function override rules
- Helpful information and hints on using many of the KcsXform member functions
- KcsXformSeq derivative

Figure 7–1 shows the relationship of the KcsXform class to the parent classes in the KCMS class hierarchy. See Figure 1–2 for an illustration of all the relevant KCMS classes.

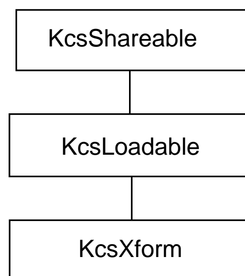

Figure 7–1 KcsXform Derivative

External Entry Points

The KCMS framework uses external entry points to load your derivative as an executable. The mandatory and optional entry points are described.

Mandatory

When you derive from a `KcsXform` class, the mandatory external entry points are:

```
extern

long KcsDLOpenXfrmCount; KcsXform * KcsCreateXfrm(KcsStatus *aStat,

 KcsChunkSet *aChunkSet, KcsChunkId aChunkId,  KcsAttributeSet

*aAttrSet);
```

`KcsCreateXfrm()()` creates an instance of a `KcsXform` derivative.

Optional

When you derive from a `KcsXform` class, the optional entry points are:

```
KcsStatus

KcsInitXfrm(); KcsStatus

KcsCleanupXfrm();
```

Example

The following example shows you how to use the entry points when creating a `KcsXform` derivative.

CODE EXAMPLE 7–1 `KcsXform` Class Entry Points Example

```
extern long

KcsDLOpenXfrmCount = 0;  /* Global initialization */ KcsStatus

KcsInitXfrm(long libMajor, long libMinor, long *myMajor, long *myMinor) {  //

Set up the return values  *myMajor = KCS_MAJOR_VERSION;  *myMinor =
```

```
KCS_MINOR_VERSION;   //Check the major version  if (libMajor !=
KCS_MAJOR_VERSION)   return (KCS_CMM_MAJOR_VERSION_MISMATCH);   //Currently,
if minor version of library is less than the KCMS   // minor version, return
an error.  if (libMinor < KCS_MINOR_VERSION)   return
(KCS_CMM_MINOR_VERSION_MISMATCH);   //Library guarantees if your minor version
number is greater than  //KCMS minor version number, it will handle it. No
more init.  return(KCS_SUCCESS); }  KcsXform * KcsCreateXfrm(KcsStatus *aStat,
KcsChunkSet *aCS,   KcsChunkId aChunkId, KcsAttributeSet *aAttrSet) {
 //Create the new derivative  return(new KcsTechUCP(aStat, KcsLoadAllow, aCS,
aChunkId,    aAttrSet)); }  /* Global clean up */ KcsStatus KcsCleanupXfrm() {
 KcsStatus sStat;  return(KCS_SUCCESS);
}
```

Member Function Override Rules

The following table tells you which `KcsXform` member functions you must override and can override when deriving from this class. The member functions indicated with an “X” in the Must column are required to successfully derive from this base class. All of these member functions are defined in the `kcsxform.h` header file and the *KCMS CMM Reference Manual*.

TABLE 7–1 `Kcs`Xform Member Function Override Rules

Member Function	Override Rules	
	Must	**Can**
`compose()()`		X
`connect()()`		X
`connectSink()()`		X

TABLE 7–1 `Kcs`Xform Member Function Override Rules *(continued)*

	Override Rules	
Member Function	**Must**	**Can**
`connectSource()()`		X
`connectXform()()`		X
`convertXform()()`		X
`eval()()`	X	
`getAttrSet()()`		X
`getLoadOrder()()`		X
`getSaveOrder()()`		X
`KcsXform()()`	X	
`~KcsXform()()`		X
`loadU()()`		X
`numberOfCallbacks()()`		X
`optimize()()`		X
`saveU()()`		X
`setAttrSet()()`		X
`setCallbackInterval()()`		X
`setComponentDepth()()`		X

TABLE 7–1 KcsXform Member Function Override Rules *(continued)*

Member Function	Override Rules: Must	Override Rules: Can
setDefaultAttributes()()		X
setNumComponents()()		X
validateLayouts()()		X

Technology

In the KCMS framework environment, the term ***technology*** means algorithms, code, and data used to implement a specific method of color transformations. All supported technologies must supply certain uniform functionality. You can do this in C++ by having a KcsXform base class with pure virtual methods. Each technology is implemented in a derived class that ***must*** implement the required virtual methods.

With transformation conversion, a technology or base class can default to a specific derivative with the functionality that best meets that technology. For example, the KcsXform base class is aware of only one type of KcsXform derivative that can save universally. Therefore, the default saveU()() method converts whatever technology it has into a KcsTechUCP. Then it asks the converted KcsXform to do the saveU()().

KcsXform Attributes

KcsXforms contain their own KcsAttributeSets. They are passed in through all constructors and default to NULL. The KcsAttributeSets are copied and are not shared by default: they are set by their constructor callers. All derivative constructors are updated.

The KcsProfile base class copies some standard attributes to the appropriate KcsXform. Access is through methods that set and get the attribute set; therefore, all access to these attributes are equal to the interface to KcsAttributeSets.

Optimization

Transformation optimization includes one or more compositions, but this is not always the case. That is why `optimize()()` is separate from composition. Generally stated, optimizing an object makes it smaller, faster, more precise, or some combination of the above. It is up to the derivative to figure out what is best for its situation. For example, if your derivative contains a resource such as extra tables for quality purposes and the derivative is requested to optimize for space and speed, it may very well throw away those extra tables.

During a save, this same derivative may not want to get rid of these extra tables. Instead, it either can use the hierarchical method described in save or reread the tables back into memory and save them again. It is up to the derivative. The choice might depend on the size of the table or some error constraint on the transform. See "`KcsXformSeq` Derivatives" on page 97 for information on how a derivative always composes and keeps that transformation for evaluation. Also note that it keeps the original transformations in the list unless it is also told to optimize for size, after which it will get rid of them.

If your derivative discards a resource in the process of optimizing and subsequently attempts to retrieve it, that resource may no longer be available. Ultimately it is up to the derivative to decide when and how to make that determination. It may (and probably will) change between releases of that derivative as well.

Optimization must be defined by the derivative if that functionality is needed. Only the derivative instances understand how best to optimize. The derivative can refuse any optimization request. It also can prioritize the types of optimization if more than one bit is set. For example, if the instance is told to optimize for space and speed and speed means to add space, then if you consider it appropriate, have your derivative add the space to support the speed increase.

Loading

Defer some loading functionality to other objects in the KCMS framework, because the objects can minimize and load more efficiently. With the `KcsXform` class, the object does not need to implement the load in all derivatives for the first time. This means that the profile instance has the objects (in this case the `KcsXform` derivatives) load and unload themselves, but it still has to load and minimize objects through construction and destruction to make up for those `KcsXform` derivatives that do not load and minimize.

The `KcsXform` base class provides the default load virtual methods that return the `KCS_NOT_RUNTIME_LOADABLE` error. This error allows the `KcsProfile` class, or

any other KcsXform container, to check for this error condition and to use another approach if necessary.

Note - Currently, not all technologies provide their own loading mechanism; use the base class functionality.

Save Types

Since there is more than one way to save, derivatives can specify the order in which its pieces get saved. The save types consist of bit sets and are:

- Universal

  KcsXform derivative saves and loads an industry-standard format—ICC 3.2.

- Private

  KcsXform derivative saves and loads the standard framework through an unformatted chunk of data.

- Universal as Private

  KcsXform derivative saves and loads data in a chunk, but uses the universal format. This type allows the KcsProfileFormat derivative to map a chunk set's chunk Id to the data in universal format so that the KcsXform class can use the getChunk()() method. This save type sets its own bit and the universal bit, since it saves universally.

These choices are available with an extensible protocol in which:

- Derivatives are passed what the caller has as possibilities.
- Derivatives only save in the order they care about.
- The caller is obligated to the order returned by the derivative.

  Obligation can be broken if the caller supplies a new set of possibilities, or the caller never saves.

- Private pieces are always supported.

  This implementation is necessary for backward compatibility.

Universal

The KcsXform base class supports saving in the universal format. The save()() method converts the object to ICC 3.0 to icLutX form. You need to provide allocation of the *aLut argument. When complete, the converted date is copied to

the *aLut variable. This method is used by other objects during `save()()`. The ICC 3.2 profile format derivative calls `KcsXform` during its save to convert the `KcsXform` object into the appropriate ICC transformation attribute. If not overridden, the `KcsXform` base class converts the transformation into a `KcsXform` derivative that supports the `save()()` method and returns its conversion. If a derivative needs more control over this type of save, then it must override this method.

Private

Private saving uses the chunk set and chunk Id associated with the instance to save. The derivative only needs to package all of its data into a contiguous piece of memory and pass the address and its chunk Id to the object's chunk set. If this is too limiting, you can split the derivative's pieces into different chunks, each with its own chunk Id. The only caveat is that the instance must then place all of those chunk Ids into one chunk, which is ultimately saved as the top of the object.

This approach is appropriate when the object has many data structures that it does not want to store into one contiguous memory block. It also helps with loading if all the pieces are not needed all the time. This is the overall approach taken by the KCMS framework where the `KcsProfile` class has a table of `chunkIds`, one of which is the attribute `chunkId` for this profile. When loading attributes only, it is faster to use `getchunk()()` and load just the attribute object than it is to use `getchunk()()` and load the entire set of objects that represent a profile.

Example

ICC has both universal and private places for transformation data. The `InterColorProfileFormat` asks for the load order and gives a list of universal plus private. The Universal Color Processor (UCP) derivative responds with `universalAsPrivate`. Since the derivative knows that UCPs can do this, it asks any `KcsXform` derivative that does not save in the universal format to convert itself into a UCP. This follows the second way to break an obligation, since the `InterColorProfileFormat` actually converts the transformation to another kind and saves the converted one. It never saves the original.

The `typdefs` are as follows:

```
typedef

long KcsLoadSaveSet; #define KcsNoParts

((KcsLoadSaveSet)0x00000000) #define KcsPrivatePart

((KcsLoadSaveSet)0x00000001) #define KcsUniversalPart

((KcsLoadSaveSet)0x00000002)

 #define KcsUniversalisPrivate

 ((KcsLoadSaveSet)((0x80000000)|KcsUniversalPart|KcsPrivatePart))
```

Composition

Some technologies convert from another technology (`Xform *`). For example, CS1.0 logTech can generate an instance of itself from any other (`KcsXform *`) derivative. It does this by calling the `compose()()` method, which takes a (`KcsXform *`) and returns a (`KcsXform **`). To use this technique, you should supply a callback function because it can be a slow operation.

The KCMS framework uses this protocol to implement a `sequence KcsXform` derivative that can take many transformations and treat them as one by sequentially evaluating the chain. Since the `KcsXformSeq` class is a `KcsXform` derivative, one `LogTech` can be generated that represents the complete connection. This has tremendous speed and quality advantages.

The `KcsXform` base class performs composition using the default CMM.

Evaluation

When a `KcsXform` is instantiated, it is ready to transform `n->m` component data (unless it is in the process of being built). Since it can handle many different data formats, the KCMS framework encapsulates the description of the data to be transformed into a data structure called `KcsPixelLayout`. This structure is an array of component descriptions. See the ***KCMS Application Developer's Guide*** for more information on `KcsPixelLayout`.

The `KcsPixelLayout` structure is used by the `eval()()` methods to describe the information to be transformed. When using an `eval()method()`, supply a source

PixelLayout, a destination PixelLayout, and a callback function. The eval()()method takes the data described by the source layout, transforms it, and puts it into the buffer described by the destination layout. If the evaluation is going to take a long time, the callback function is repeatedly called until evaluation is complete.

The layouts can describe the same buffer (a technique called *in-place transformation*). In this case, the eval()() method detects it is the same buffer and optimizes for performance. The layouts can also specify different buffers, in which case the data is moved as well as transformed. You can even supply two layouts which differ in composition (for example, planar RGB and chunky CMYK), and the data is moved and transformed from RGB to CMYK as well as has its composition transformed from planar to chunky. The evaluation methods accomplish this with minimal steps. Evaluation is most efficient when given large buffers of data to transform.

If the transformation is not compatible with the layout(s), it returns an error. For example, if an RGB->CMYK KcsXform * is given two 3-component descriptions, it would return an error if the destination is expecting 4-component data.

If your data can be represented in different formats, get the value of the attribute KcsAttrPixelLayoutSupported to see what the most efficient pixel layout is for that KcsXform derivative.

Evaluation Helper Methods

KcsXform includes only one pure virtual eval()() method. It is the one with two pixel layouts and a callback as arguments. Other eval()() methods are overloaded in the base KcsXform class to allow other data types to fit a long on each side of the xform, for example,

```
(long *) -> (long

*)
```

and

```
aRGB->aR'G'B'
```

The base class takes all the overloaded methods and creates a pixel layout for each method. Then it calls the pure virtual method. Therefore, derivatives only need to implement one eval()() method.

When starting an evaluation, the derivative can use any of the helper functions provided for pixel layout usage. The convertLayouts()() method takes any layout and transforms it into any other. If a derivative can only handle chunky, the derivative may want to convert a non-chunky derivative to chunky before the evaluation is started.

KcsXformSeq Derivatives

The KcsXformSeq class is a KcsXform derivative that allows a list or concatenation of transformations to act as one KcsXform. It is an alias to an ordered transformation collection that allows all normal list management in addition to all of the required KcsXform protocols. It also allows a hierarchy of KcsXform instances by providing the ability to sequence the list. Evaluating through a sequence of KcsXforms is like serially running each transform, with successive transformations taking input from the output of its predecessor and ending with the last one putting its output into the destination location.

Constructs and Destructors

You can construct a KcsXformSeq with any of the following:

- An empty constructor

  Like all constructors, this one has a status object passed by reference to simplify constructor failure recovery.

- A chunk set/chunk Id constructor

  This constructor also provides the required load hints.

- A special sequence constructor

  This constructor takes a status object and load hints just like all KcsXforms, but it also accepts an array of KcsXform *s and a count (numXforms) so that you can generate sequences from scratch.

Saving

Saving trickles down throughout the whole connected hierarchy. Any change to any transformation in the sequence is saved when the sequence is saved. This happens because the sequence shares the transformations passed to it. The instance also gets the chunk Ids from each transformation in the list. It then packs these and other state information into a memory block and does a setChunk()() to allow lookup of this transformation list upon a load request to the sequence.

When requested to save in universal format (see “Universal” on page 93 for details), the sequence does a composition that generates one transformation that is saved in this format.

Loading and Constructing the List

A `KcsXformSeq` instance saves its transformations as a list of chunk Ids to later instantiate when needed. For every chunk Id in its own chunk, `getChunkXform()()` takes the current chunk set and chunk Ids and, through the chunk set protocol of `createXform()()`, allocates the transformation represented by that unique combination.

Connections

`KcsXformSeq` is the only class in the KCMS framework that supports connection (and connection is the only reason the `KcsXformSeq` class exists). The base `KcsXform` class uses a sequence derivative in its `connect()()` method.

To make a connection, you can call either of two `KcsXformSeq` constructors (or use a combination of the two): one constructor takes a list of transformation pointers; the other creates a sequence of 0s. Then edit the transformations list with the `list()()` methods. See “Validation” on page 99 for additional information on the connection method.

Optimization

When a sequence is told to optimize itself, first it optimizes each transformation in the chain individually. Then it composes all the transformations into one `KcsTechUCP` transformation. Finally it uses that composed `KcsTechUCP` to do future evaluations. Overall optimization is provided with optimization and composition of the individual transformations in the list.

The `KcsXformSeq` class performs composition by asking each transformation in the list to compose. If none comply, it uses the base class method to compose. It attempts to compose from the rightmost to leftmost. By doing so, the harder-to-model devices (typically printers, which are on the right) get composed first.

If you request to optimize for size, `KcsXformSeq` detaches all of the original list. After optimizing for size, the only way to regenerate the original list is to build it again.

Composition

The `KcsXformSeq` class uses the `compose()()` method to implement optimization. Since the `KcsXformSeq` class is a `KcsXform` derivative, you can generate one `KcsTechUCP` that represents the complete connection. This offers performance and quality advantages.

Evaluation

Evaluation of a KcsXformSeq instance is done with either the optimized or non-optimized technique.

Optimized evaluation uses the composed transformation it constructed when told to optimize. It keeps a pointer to that optimized transformation in its private section. When asked to evaluate, it passes the information down to the optimized transformation.

Unoptimized evaluation is used when the sequence is not optimized. This implementation evaluates the data through the list of transformations sequentially. Between transformations, a buffer is used to hold the temporary calculations. The first step evaluates from the source buffer, while the last step evaluates into the destination buffer.

Up to two different extra buffers are used between non-endpoint transformations, depending on the layout of the data. They are swapped between eval()()s. If the composition of the transformations is different (for example, chunky and planar), two buffers are needed. If the implementation did not use this technique, the data from one complete pixel (or component set) overrides a different (set of) pixel. The eval()() method always alternates between two buffer pointers. Both buffer pointers point to the same buffer if an output buffer for a transformation is compatible with the input buffor for the next transformation. This can be optimized further if all buffer layouts describe a buffer that is compatible with the destination buffer supplied by the caller. In this case, the buffer pointers point to the destination buffer described. And if the caller is using the same buffer for source and destination, everything ultimately uses one buffer. Such buffers are KcsMemoryBlocks that can be resized.

Validation

Each time a connection is made, it is validated against a set of rules defined in thisKcsXformSeq class. The rules use the current set of attributes as well as the current state of all of the transformations in the connection.

If the sequence rules pass, the sequence passes itself down to all the validation methods of each KcsXform in the list. In this way, all KcsXforms are allowed to determine if a connection can be made. If an error occurs in any single KcsXform, the connection is refused.

The List

The list of transformations is represented by a memory block of pointers to KcsXforms. The size of the block is incremented by a constant each time the current block fills with pointers. A few methods access and edit the list.

Note that a NULL parent starts the list based on this sequence. You must pass the last parent found into the next call to getNextXform()() and use the same object for invocations of this method. getNextXform()() returns KCS_END_OF_XFORMS when it reaches the end of the transformations in the sequence. All getNextXform()() calls are sequential. Any sharing of an object must take this into account. Otherwise, if the calls to getNextXform()() are not synchronized, two different results may occur. getNextXform()() works correctly when called on a sequence that is a part of another sequence: it runs through that subsequence only.

For example, given sequence A (a->B->e) and sequence B (c->d) where a, c, d, and e are primitive transformation types: A->GetNext()(). If GetNextXform()() is called (starting with a NULL parent **) until it returns KCS_END_OF_SEQUENCE, it returns transformations in the following order: a, c, d, e, B->GetNext()(). If called (starting with a NULL parent **) until it returns KCS_END_OF_SEQUENCE, it returns transformations in the following order: c, d. It also skips over all sequences of 0 transformations as if they are not even there.

CHAPTER **8**

KcsStatus Extension

Every API function returns a KcsStatusId to inform the application when it has executed successfully or, if it has not, why it has failed. If a function successfully executes, it returns the status code KCS_SUCCESS. If the application's user cancels a function before its completion, the function returns the status code KCS_OPERATION_CANCELLED. API calls can also return warning messages. See the SDK manual *KCMS Application Developer's Guide* for more details.

The KcsStatus extension returns a status message string. Provide a maximum of two functions depending upon whether or not you want custom errors and warnings in your software.

```
extern

long KcsDLOpenStatCount; char * findErrDesc(KcsStatus statId); char *

findWarningDesc(KcsStatus statId);
```

findErrDesc()() creates an instance of the function connecting the custom error codes with your string descriptions.

findWarningDesc()() creates an instance of the function connecting the custom warning codes with your string descriptions.

You can add your own findErrDesc()() and findWarningDesc()() functions and provide a header file with your own error and warning numbers and strings. Use custom status codes in your software and identify them with an OwnerId value so that your dynamically loadable status module is used for messages rather than the KCMS default messages. The OwnerId is a long that you set in your loadable module to identify your error and warning messages.

Example

Use these on-line files as a reference for this example:

/opt/SUNWddk/kcms/src/kcssolmsg.cp
/opt/SUNWddk/kcms/src/kcssolmsg.h

The following is an excerpt from the `kcssolmsg.cc` file. Use it as a template when extending the `KcsStatus` class.

CODE EXAMPLE 8–1 `KcsStatus` Class Example

```
... char *

findErrDesc(KcsStatusId statId) { #ifndef KCSSOLMSG_STRINGS #define

KCSSOLMSG_STRINGS  setlocale (LC_MESSAGES,'''');

 bindtextdomain(``kcssolmsg.strings'',''/usr/lib/locale'');

#endif  switch (statId)  {   case KCS_SOLARIS_FILE_NOT_FOUND:

   return(dgettext(``kcssolmsg_strings'',''Could not find

Solaris file type \       profile''));   case KCS_X11_WINDOW_PROF_ERROR:

   return(dgettext(``kcssolmsg_strings'',''Error in X11 window

profile'')); ...
```

Header File

In addition to the two functions, `findErrDesc()()` and `findWarningDesc()()`, you need to provide a header file to incorporate status messages into your code. The header file should contain the following:

```
#define

<identifier> <status

number>
```

This `define` links a status identifier (for example, `KCS_SOLARIS_FILE_NAME_NULL`) with a hexadecimal status identification number

(for example, 0x4203). You can assign numbers to your own status numbers that are not used by the KCMS library and only in the following specified ranges:

- Warning range: 0x1007 - 0x3ffe
- Error range: 0x4122 - 0x6ffe

The header file should also contain your OwnerID (for example, SolMsgOwner) by which these messages are distinguished from the KCMS default messages.

The setId( ) function is one of the KcsStatus class methods.

```
status->setId(KCS_SOLARIS_FILE_NOT_FOUND,

SolMsgOwner);
```

It sets an ID (OwnerID) that tells the KcsStatus class function, getDescription()( ), that it is not a KCMS library error and to search the OWconfig file for a dynamically loadable module containing the matching error descriptions.

This example also contains code that prepares the message strings for language localization. You must setlocale(3C) and bindtextdomain(3I) once, so choose a unique define with which to ifdef these functions. Also choose a message file name for the message extraction script, xgettext(3I), in the makefile.

Localizing Messages

Note that the message strings are arguments in the dgettext(3I) function that marks these strings for inclusion in the kcssolmsg_strings.po file upon running xgettext()( ) on this code file.

These are very terse notes on what the application or library should contain and what you should run to create a file of messages ready to be translated into the local language.

See setlocale(3C), bindtextdomain(3I), gettext(3I), dgettext(3I),and msgfmt(1) for information on internationalization and localization.

Application Module

The application, or library module must include the following code:

```
#include

<locale.h> #include <libintl.h>

setlocale(``LC_MESSAGES'',

``<language>'');

bindtestdomain(``string_file_name'',''directory'');

dgettext(``string_file_name'',

``message'');
```

where

language is one of the language locale directories in `/usr/lib/locale`

directory is the location of the installed translated message file, *string_file_name*

message is the message string to translate

Developer

As the CMM developer, you must do the following tasks to create a file of messages to translate into the local language:

1. Use the `-lintl` linker option when building.
2. Run `xgettext` on files using the `dgettext()` function.
3. Edit the resulting `.po` file to translate the messages into the appropriate language.
4. Run `msgfmt` on the `.po` file to create a `.mo` file.
5. Move the `.mo` file to the appropriate directory, such as `/usr/openwin/lib/locale/<language>/LC_MESSAGES`.

The application should then pick up the translated messages.

APPENDIX **A**

Supported Devices

Supported Devices

Table A–1 lists the types of devices by manufacturer and model and profile filename. All of these devices are supported by the KCMS framework.

Note - Most of the color space conversion profiles listed in Table A–1 support KCMS integration into Sun's XIL™ Imaging Library.

TABLE A–1 Supported Devices

Device Type	Manufacturer and Model	Profile Filename
SPARC Monitors	Sony Multiscan 16 or 19 inch	`kcmsEKsony16.mon`
	Sony 20 inch P4	`kcmsEKsony.mon`
	Sony 17 inch N1	`kcmsEKsony17.mon`
	Sony 16 inch P3	`kcmsEKsony16.mon`
	Nokia 15 inch	`kcmsEKnokia15.mon`
x86 Monitor	ViewSonic 17 inch	`kcmsEKvs17.mon`

TABLE A–1 Supported Devices *(continued)*

Device Type	Manufacturer and Model	Profile Filename
Other Monitors	Apple 13 inch	kcmsEKappl13.mon
	Generic EBU 1.8 Gamma	kcmsEKebu18.mon
	Generic P22 monitor, 1.8 gamma, dim ambient	kcmsEKp22g187d.mon
Input	Hewlett Packard ScanJet IIc	kcmsEKhpsjtw.inp
	Kodak PhotoCD Color Negative	kcmsEKphcdcn.inp
	Kodak PhotoCD Ektachrome	kcmsEKphcdek.inp
	Epson ES-800C Scanner	kcmsEKepsn1p.inp
	Epson ES-800C Scanner	kcmsEKepsn3p.inp
	Kodak RFS 2035 Scanner	kcmsEKk2035.inp
	Nikon LS-3510 AF Scanner	kcmsEKls3510.inp
	UMAX PowerLook Scanner	kcmsEKumax_a.inp
	Hewlett-Packard ScanJet IICX	kcmsEKhpsjt35b.inp
	Microtek MT600Z	kcmsEKmt600zek.inp
Output	Sun SunPics NeWSprint CL+	kcmsEKsunnws.out
	Canon BJC-800 Printer	kcmsEKbjc800.out
Output	Kodak PS 1 Printer	kcmsEKcewps1.out
	Hewlett Packard DeskJet/ DeskWriter 550C Printer	kcmsEKhp550c.out
	Tektronix Phaser III PXi Printer	kcmsEKtpiiic0.out

TABLE A–1 Supported Devices *(continued)*

Device Type	Manufacturer and Model	Profile Filename
	Kodak XL7700/XL7720 Printer	kcmsEKxl7700.out
	Kodak XKS8300 Printer	kcmsEKxls830.out
	Fargo Primera Printer	kcmsEKprimer.out
	QMS ColorScript 100 Model 30/30i Printer	kcmsEKqms30i.out
Color Space Conversions	Kodak RGB to CIELAB	kcmsEKRGB709.spc
	Sun YCC601 to CIELAB	kcmsSUNWYCC601.spc
	Sun YCC709 to CIELAB	kcmsSUNWYCC709.spc
	Sun YCC601 linear to CIELAB	kcmsSUNWYCC601L.spc
	Sun YCC709 linear to CIELAB	kcmsSUNWYCC709L.spc
	Sun RGB linear to CIELAB	kcmsSUNWRGBL.spc
	Sun Photo YCC to CIELAB	kcmsSUNWPhotoYCC.spc
	Sun CMYK to CIELAB	kcmsSUNWCMYK.spc
	Sun CMY to CIELAB	kcmsSUNWCMY.spc
	Sun Y linear to CIELAB	kcmsSUNWYlinear.spc