KCMS CMM Developer's Guide

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

SunSoft
A Sun Microsystems, Inc. Business

Copyright 1996 Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043-1100 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.

Portions of this product may be derived from the UNIX® system, licensed from Novell, Inc., and from the Berkeley 4.3 BSD system, licensed from the University of California. UNIX is a registered trademark in the United States and other countries and is exclusively licensed by X/Open Company Ltd. Third-party software, including font technology in this product, is protected by copyright and licensed from Sun's suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

Sun, Sun Microsystems, the Sun logo, Solaris, SunSoft, the SunSoft logo, and AnswerBook are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. OPEN LOOK is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc., which may be registered in certain jurisdictions.

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

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

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

Please
Recycle

Adobe PostScript

Contents

Figures

Tables

Code Samples

Feature Notes

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

KCMS is Multithread Unsafe

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

Preface

The *KCMS 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 Book

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 Book

Check all of the following for any KCMS-specific or release 2.5-specific information that you might need:

- You should be familiar with the Kodak Color Management System (KCMS) API which is part of the Solaris Software Developer's Kit (SDK) in the KCMS AnswerBook™ on-line documentation; 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*.
- Check the following manuals for any corrections or updates to the information in this manual:
  - *Solaris 2.5 Driver Developer Kit Introduction*
  - *Solaris 2.5 Driver Developer Kit Installation Guide*
- See the on-line SUNWrdm packages for information on bugs and issues, engineering news, and patches. For Solaris installation bugs and for late-breaking bugs, news, and patch information, see the *Solaris Installation Notes* (SPARC™ or x86).
- For SPARC systems, consult the updates your hardware manufacturer may have provided.

How This Book Is Organized

Chapter 1, "Class Descriptions," briefly describes each of the classes in the KCMS CMM class hierarchy.

Chapter 2, “CMM Runtime Derivative,” 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 , “KCMS Framework Operations,” provides examples of how some of the C++ methods interface with the KCMS framework API.

Chapter 4, “KcsIO Derivative,” describes how to derive from the `KcsIO` base class.

Chapter 5, “KcsProfile Derivative,” describes how to derive from the `KcsProfile` base class.

Chapter 6, “KcsProfileFormat Derivative,” describes how to derive from the `KcsProfileFormat` base class.

Chapter 7, “KcsXform Derivative,” describes how to derive from the `KcsXform` base class.

Chapter 8, “KcsStatus Extension,” describes how to extend the `KcsStatus` base class.

Appendix A, “Naming and Installing Profiles,” 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 manual:

- *KCMS CMM Reference Manual* (part of DDK)
- *KCMS Application Developer’s Guide* (part of SDK in the KCMS AnswerBook on-line documentation)
- *KCMS Calibrator Tool Loadable Interface Guide* (part of SDK in the KCMS AnswerBook on-line documentation)
- *ICC Profile Format Specification* (located on-line in `/usr/openwin/demo/kcms/docs/icc.ps`)

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

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#`

Class Descriptions

1

This chapter briefly describes the KCMS framework classes. Although you can only derive from four of these classes (`KcsIO`, `KcsProfile`, `KcsProfileFormat`, and `KcsXform`) to add new loadable CMMs, you need to know about all of the classes you may use in the implementation of your CMM.

See the *KCMS CMM Reference Manual* for descriptions of the enumerations and protected and public members of each class.

KCMS Class Hierarchy

All relevant classes in the KCMS framework are shown in the following diagram. Each class is discussed in the following sections.

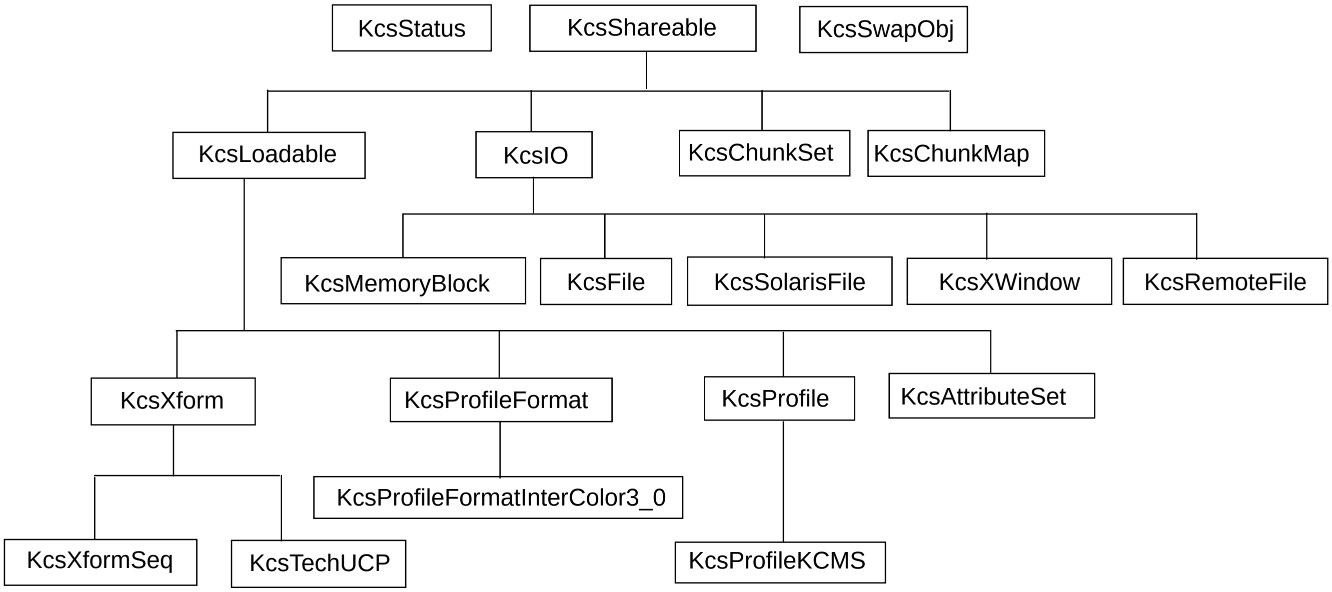

Figure 1-1 KCMS Class Hierarchy

`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()`. `dettach()` 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 `Xform` class is used; see Chapter 7, “KcsXform Derivative” for more information. For example, a sequence xform saves its array of transform `chunkId`s in the same `chunkset` as it does its own state. The `KcsXformSeq` class has an array of pointers to xforms when it is allocated in memory.

Since all of these xforms have unique identifiers, the `KcsXformSeq` class places the UID of each xform 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 xform `chunkId`, it calls the `KcsXform::createXform(uid)` constructor. This constructor allocates the xform 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:

> The derivation implements the `unload()` method to minimize the state of the object in memory. Then in the `load()` method, restore the state of the object to that described by the loadable base class' `chunkSet` and `iChunkId` member fields. If in the unloaded state, return an error to signal that a `load()` call in the state is not adequate for all methods. Additionally, for methods that need access to the state, load 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 I/O (input/output) 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 OS 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, "KcsIO Derivative" 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 construcor 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.

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 1-2, 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.

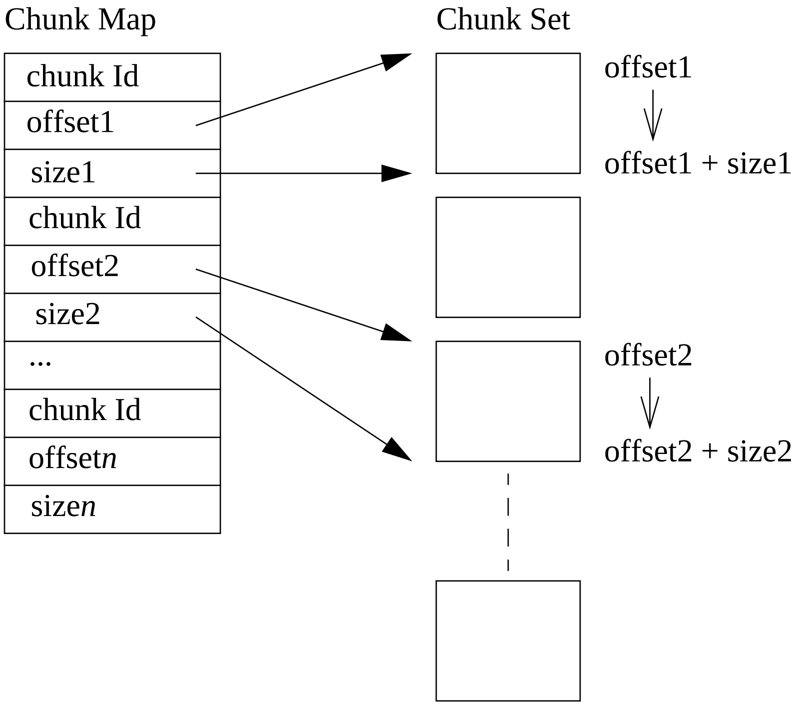

Figure 1-2 Chunk Set Layout

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 `KcsChunkSets` 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 do not need the chunk Id(s) to access data directly. Use `KcsChunkSet` to read or write a particular chunk Id. You also 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. It 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`.

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, “KcsProfile Derivative” 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, “KcsProfileFormat Derivative” 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 `KcsAttributeSet` 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 `KcsTags` and `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, “KcsXform Derivative” 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, “KcsStatus Extension,” 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.

CMM Runtime Derivative

2

This chapter briefly discusses how to create a CMM that is a runtime derivative. It defines a CMM and explains the steps required to write a CMM. It explains the Solaris runtime derivation mechanism and how to name your CMM and make entries in the `OWconfig` file to make your CMM known to the system at runtime. It also discusses each of the KCMS classes from which you can derive or extend:

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

Subsequent chapters detail how to create class derivatives.

Creating a CMM

A CMM is defined as:

- Color management techniques
- Data structures
- Profiles

Follow these steps to create a CMM that is a runtime derivative:

1. **Understand the KCMS framework, its general principles and the SDK "C" interface.**
   See Chapter 3, "KCMS Framework Operations" for more information.
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.**
   Knowing more about the derivable KCMS classes will help you with this decision; see "Derivable Classes."
3. **Understand the runtime mechanism for derivatives.**
   See "Runtime Derivation Mechanism" on page 18.
4. **Understand the CMM naming conventions and the `OWconfig` file.**
   See "Loading CMMs" on page 22.
5. **Implement your KCMS framework runtime extensions.**
   The information in this manual as well as the *KCMS CMM Reference Manual* will help you understand the foundation library interfaces. You may also find it helpful to use the *KCMS Application Developer's Guide* SDK manual for information on the KCMS framework API.
6. **Test your CMM.**

Derivable Classes

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 (`kcsSUNWIOsolfi.so.1`).

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 `UID` (unique identifier)).

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

KcsProfile

Derive a new `KcsProfile` class for characterization and calibration, additional functionality, or new transform derivatives. Usually, this involves overriding one of the update methods to actually produce a new `KcsXform`. Once the transformation is saved to the static store, the runtime load mechanism deals with it automatically 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 `KcsXform`. 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, "KcsProfile Derivative," for information on creating a `KcsProfile` class derivative.

KcsProfileFormat

You can create a `KcsProfileFormat` class derivative to support an existing 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 should be in a known location in the profile header. This is always the case with the ICC profile format; see the ICC specification for this location. 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, "KcsProfileFormat Derivative," for information on creating a `KcsProfileFormat` class derivative.

`KcsXform`

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

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

`KcsStatus`

The `KcsStatus` class represents a consistent object-oriented way of returning results from all of the KCMS methods. It allows a representation of error and warning values, error text descriptions, error conversions to and from a `KcsStatusId`, error comparisons, external mappings through the C API, and 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, "KcsStatus Extension," for information on extending the `KcsStatus` class.

Runtime Derivation Mechanism

The KCMS framework uses a model that allows derivation of classes at runtime. This derivation changes and adds to 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 has C-based routines that load, unload, initialize, terminate, and allocate at runtime.

Wrapper Functions

The KCMS framework uses *wrapper functions* to allocate an object at runtime. 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 informs 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

The KCMS framework uses external entry points to load your derivative 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

Mandatory

Each runtime derivative must supply the following C-based external entry points and variables. In the paragraphs below,"XXXX" refers to the base class identifier from which it is being derived.

Note – XXXX can *only* have the following values: `IO`, `Prof`, `Pfmt`, `Xfrm`, and `Stat`.

`KcsDLOpenXXXXCount()`

```
extern long KcsDLOpenXXXXCount;
```

`KcsDLOpenXXXXCount()`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.

`KcsCreateXXXX()`

```
KcsXXXX *KcsCreateXXXX(KcsStatus *, XXXX creation args);
```

`KcsCreateXXXX()` is one of possibly many creation entry points. This maps directly to the static `createXXXX()` 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 that class chapter. The CMM must support all declared `createXXXX()` methods; otherwise, applications receive CMM errors from calls to `load()`.

Optional

Runtime derivatives can supply the following external entry points.

`KcsInitXXXX()`

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

If you supply `KcsInitXXXX()`, 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. The method checks for minor version numbering; see “Loading CMMs” on page 22 for more information.

`KcsCleanupXXXX()`

```
KcsStatus KcsCleanupXXXX();
```

If you supply `KcsCleanupXXXX()` the KCMS framework calls it when the shareable object is unloaded for the last time (when `KcsDLOpenXXXXCount` = `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 `KcsProfile`, `KcsProfileFormat`, `KcsXform`, and `KcsStatus` classes (Chapter 5 through Chapter 8, respectively), for detailed explanations.

Instantiation

Instantiate KCMS framework objects or any runtime derivations of that object with the `createXXXX()`, `attach()`, and `new()`methods.

`createXXXX()`

Allocate an object with the `createXXXX()` 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.

`attach()`

Share an object with the `attach()` method. 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 objects that share it. If you share an object, make sure that object does not change while attached to it.

`new()`

Get a new object of a specified type or a KCMS framework derivative with 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 "OWconfig File Structure" on page 24 for further information.

The `KcsLoadable` class then locates the `KcsDLOpenXXXXCount` external entry point. If `KcsDLOpenXXXXCount` = `0`, it locates and loads the `KcsInitXXXX()` entry point and, if available, calls it. Then the `KcsCreateXXXX()` entry point is located and loaded. If everything is successful, the create entry point is called.

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

Maximizing Extensibility to Runtime Loadability

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

Loading CMMs

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

CMM Filename Convention

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

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

The following table 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.
class	Class from which the module is derived.
`.so`	Shared object library.
unique identifier	Four-character identifier that distinguishes multiple modules derived from the same class.
version	Number compared to the `KCS_MAJOR_VERSION` number (incremented by SunSoft for every major release).

Note – The version number in the `#define` and the version number in the module name *must* match.

A few KCMS CMM filenames are:

• `kcsSUNWIOsolf.so.1`	Solaris File CMM
• `kcsSUNWIOxwin.so.1`	X11 Window System CMM
• `kcsSUNWStatsolm.so.1`	Solaris Message CMM

CMM Makefile

The CMM must be installed in `/usr/openwin/etc/devhandlers`.

A sample makefile in `/opt/SUNWddk/kcms/src` illustrates how the CMMs are compiled and installed, and how the library names are associated with the modules.

`OWconfig` *File Structure*

The `OWconfig` library must be included in the CMM linking. It is bundled with Solaris in `/usr/openwin/lib/libowconfig.so`. The `OWconfig` library provides routines to access the `OWconfig` file that gets the name of the CMM you want to dynamically load. You must add the name of your CMM to the `OWconfig` file to advertise its existence to the KCMS framework.

A generic `OWconfig` entry looks like this:

```
class=”KCS_IO” name=”solf”
    kcsLoadableModule=”kcsSUNWIOsolf.so.1”;
```

The following table describes each field.

Table 2-2 `OWconfig` File Entry Description

`OWconfig` **File Entry**	**Description**
`class`	`KCS_`<*class name*>.
`name`	Four- or eight-character identifier.
`kcsLoadableModule`	Entire module name.

`KcsIO` *Example*

```
#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”;
```

The `KcsProfileType` enumeration in `kcstypes.h` contains a type field that is a four-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 and searches all of the appropriate `OWconfig` class entries.

`KcsProfile` *Example*

```
#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 (bytes 4 through 7 in the ICC profile format). 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 module is the Solaris-supplied default.

This is the base `KcsProfile` class that can contain transforms (`KcsXform` class) and a profile format (`KcsProfileFormat` class). Since the Kodak class is built into the library, this is the mechanism by which the calibration and characterization interface can be extended.

`KcsProfileFormat` *Example*

```
#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 type and CMM Id in the ICC profile header. 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 should be ICC profile format files with a magic number equal to `acsp` and 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*

```
#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 four-character transform identifier that must be registered with the ICC and the CMM Id. 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*

```
#Extending error messages
class=”KCS_STAT” name-”solm”
    kcsLoadableModule=”kcsSUNWSTATsolm.so.1”;
```

You define the `name` field as 4 characters uniquely identifying 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, “KcsStatus Extension” for detailed information.

The `KcsStatus` class will, if an `OwnerId` variable is set with the status message, dynamically load the matching `OwnerId` set by the dynamically loaded class. The `OwnerID` is described in Chapter 8, “KcsStatus Extension.”

Updating the `OWconfig` *File*

You can insert and remove configuration entries (`class`, `name`, and `kcsLoadableModule`) in the `OWconfig` file. The supplied source program `OWconfig_sample.c` must be edited at the indicated places. Substitute your own unique names where the program comments tell you to “choose a unique name here,” for example

```
#define package “SUNWkcsdnd” /* choose a unique name here */
```

Then compile the with the `makefile.owconfig`.

```
%make -f makefile.owconfig
```

Refer to the comments at the beginning of the `OWconfig_sample.c` program for more information.

Inserting Entries

To insert configuration entries into the `OWconfig` file for your new module, run the `OWconfig_sample` program with the `insert` option as follows:

```
example% su
example# ./OWconfig_sample -i
```

The new configuration entries will be appended to the end of the `/usr/openwin/server/etc/OWconfig` file. For local machine use only, the `/etc/openwin/server/etc/OWconfig` file will be updated.

Removing Entries

To remove configuration entries from the `OWconfig` file, run the `OWconfig_sample` program with the `insert` option as follows:

```
example% su
example# ./OWconfig_sample -r
```

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.

KCMS Framework Operations 3

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 contains the following information:

- A description of the profile format
- An overview of the KCMS framework architecture
- An introduction to how the framework works using sample API programs and the corresponding framework action

Profile Format

KCMS uses the ICC profile format as the default profile format. The profile format specification is a PostScript file located on-line in `/usr/openwin/demo/kcms/docs/icc.ps`. By supporting this specification, profiles 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.

KCMS Framework Architecture

Figure 3-1 illustrates the KCMS framework architecture and how the KCMS classes are used to implement the KCMS framework. 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. The following sections help explain this diagram.

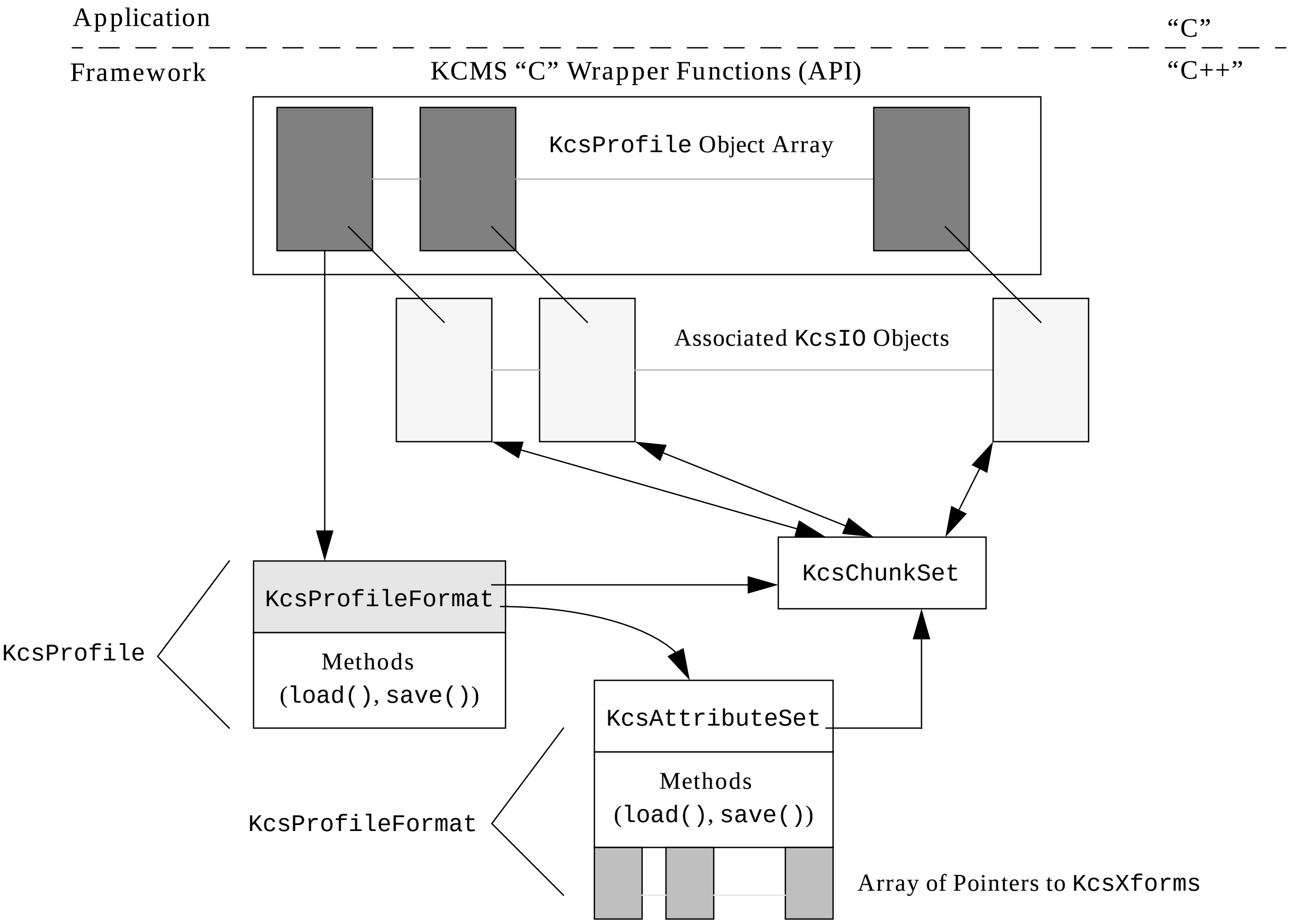

Figure 3-1 KCMS Framework Architecture

`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.

The `KcsProfile` class contains a set of public member functions that correspond to the KCMS 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 and profile version number. The ICC profile format Id is `acsp`, located at byte 36. 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 `KcsXform`s. The primary function of `KcsXform` (or transforms) 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 KCMS architecture and the flow of control and data between the KCMS API and the KCMS framework. Use Figure 3-1 on page 32 as a reference.

Loading a Profile

The example explains how a profile is loaded.

1. Using the `KcsIO` derivative, the CMM Id of the profile is determined.

2. The `KcsProfile::createProfile()` static method is called and loading starts. The CMM Id of the profile is used as a key to determine the particular `KcsProfile` derivative to load. The association of the CMM Id with dynamically loadable module is made 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 default `KcsProfile` derivative, `KcsProfileKCMS`, is used. There is a special CMM Id key `dflt` entry in the `OWconfig` file, so that you can override the default `KcsProfile` class. If you do this then you *must* duplicate all of the functionality handled in the default class.

3. The `load()` method is then called 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 and then loads itself.

   The profile format Id, byte 36, of the ICC profile format is used as the key to this entry in the `OWconfig` file.

4. The `KcsProfileFormat` object contains pointers to a `KcsAttributeSet` object and an array of `KcsXforms`. These objects are also created and their `load()` methods called to load themselves from the static store.

   The `KcsAttributeSet` object can be derived from directly since it is statically linked into the KCMS framework. The `KcsXform` array has an `OWconfig` entry that uses as a key the 4-byte identifier. For ICC-based profiles, use the 8- and 16-bit LUT tags, `mft1` and `mft2`.

5. If all pieces of the profile are loaded, a `KCS_SUCCESS` status is returned.

Getting Attributes

This example shows you how to get a profile's attributes with `KcsGetAttribute()`, once the profile is loaded.

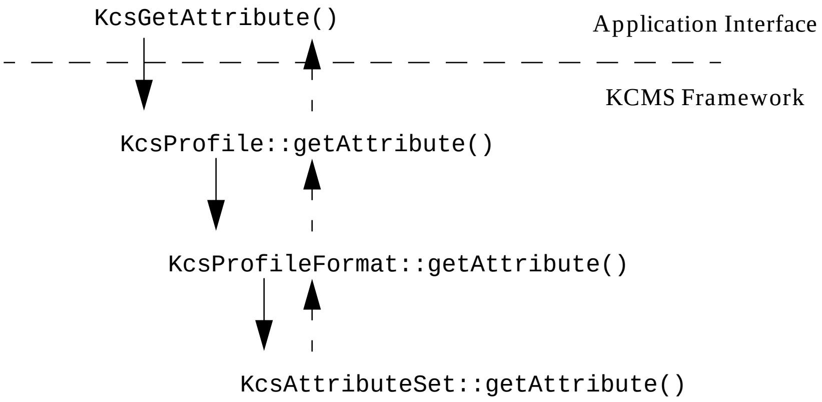

Figure 3-2 `KcsGetAttribute()` Flow Example

1. For the appropriate `KcsProfile` object in the array, call its `getAttribute()` method.
2. The `KcsProfileXXXX::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.

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

KCMS Framework Primary Operations

The following examples describe how the framework operates from the perspective of the KCMS "C" API. These examples illustrate sequences of operations in the primary framework, attributes, and calibration and characterization.

Loading a Profile From the Solaris File System

The framework must have a profile with which to operate. The following API code sample 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);
}
```

Creating a `KcsIO` Object

In this example the KCMS framework is informed from the API layer that a profile description of type `KcsSolarisProfile` is to be loaded. It uses `KcsLoadProfile()`. The name of the profile and the options for opening that file are also specified using the `solarisFile` entry in the `KcsProfileDesc` structure.

Use Figure 3-3 on page 38 to illustrate the following implementation of the `KcsLoadProfile()` API call.

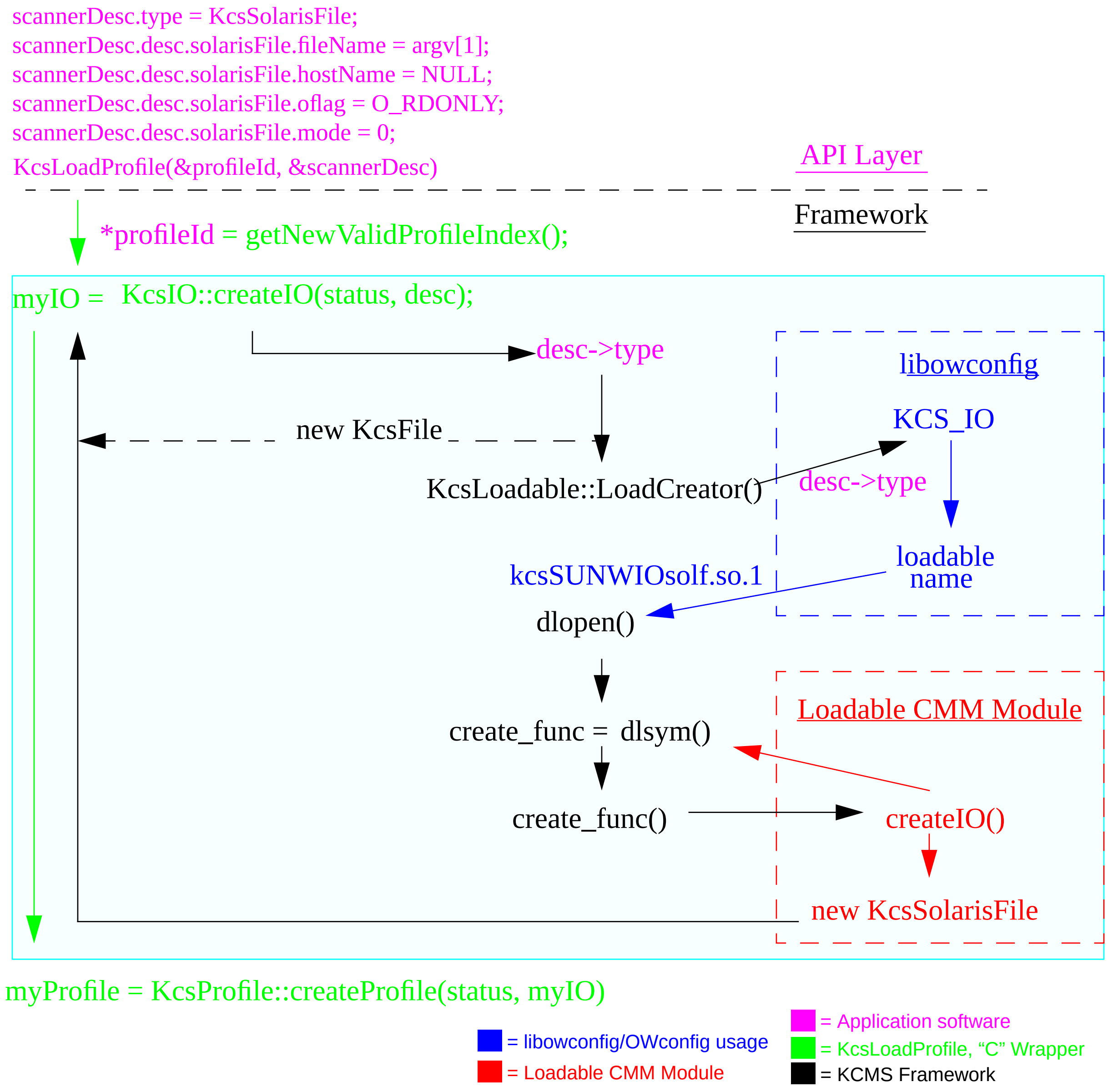

Figure 3-3 Creating a `KcsIO` Object

1. Get a new profile Id. The framework maintains a dynamically allocated global array of profiles. The `getNewValidProfileIndex()` method allocates a new profile entry.

2. All profiles access their data using the independent access mechanism `KcsIO`. A `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`. There is a third derivative, `KcsRemoteFile`, to be used with the classes `KcsSolarisFile` and `KcsXWindow` classes. In this 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 four-character string and searches entries in the `OWconfig` file for the entries that correspond to loadable `KcsIO` classes. If a match is found, the `KcsIO` module is dynamically loaded. This supplies the framework with a shared object to load.

   The `KcsIO` module contains calls to a list of known function names and the framework uses `dlsym`(3X) to bring these functions into the framework to create and load a pointer to a `KcsIO` derivative.

   See Chapter 2, “CMM Runtime Derivative” for more details.

4. Once the `KcsSolarisFile` object pointer is loaded, the `fileName`, `hostName` and `open`(2) arguments are used to search for the profile. The `hostName` is first checked to see if the file is on a local or remote machine. Depending on the location, the `KcsSolarisFile` 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 the file is on a remote machine the `fileName` and `hostName` are passed to `KcsRemoteFile` and an object pointer is created.

As shown in Code Example 3-2, the `KcsFile` or `KcsRemoteFile` pointer which the `KcsSolarisFile` file object contains is then used to override the `KcsIO` methods.

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` has been created the profile can be loaded. The following diagram illustrates the process.

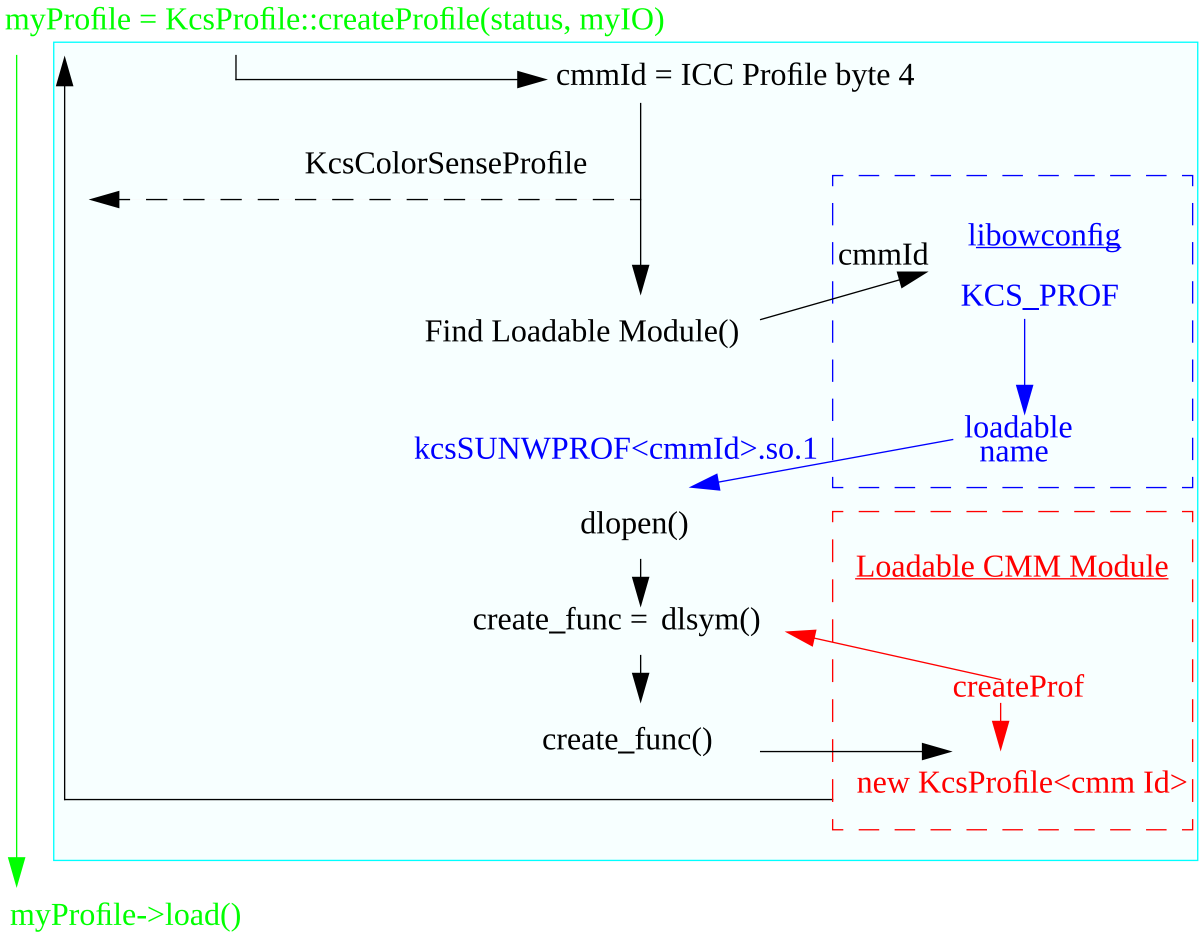

Figure 3-4 Creating a `KcsProfile` Object

The first step 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 in the profile. 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` has been created you can ask it to load itself using the generated `KcsIO`. The following diagram illustrates the process.

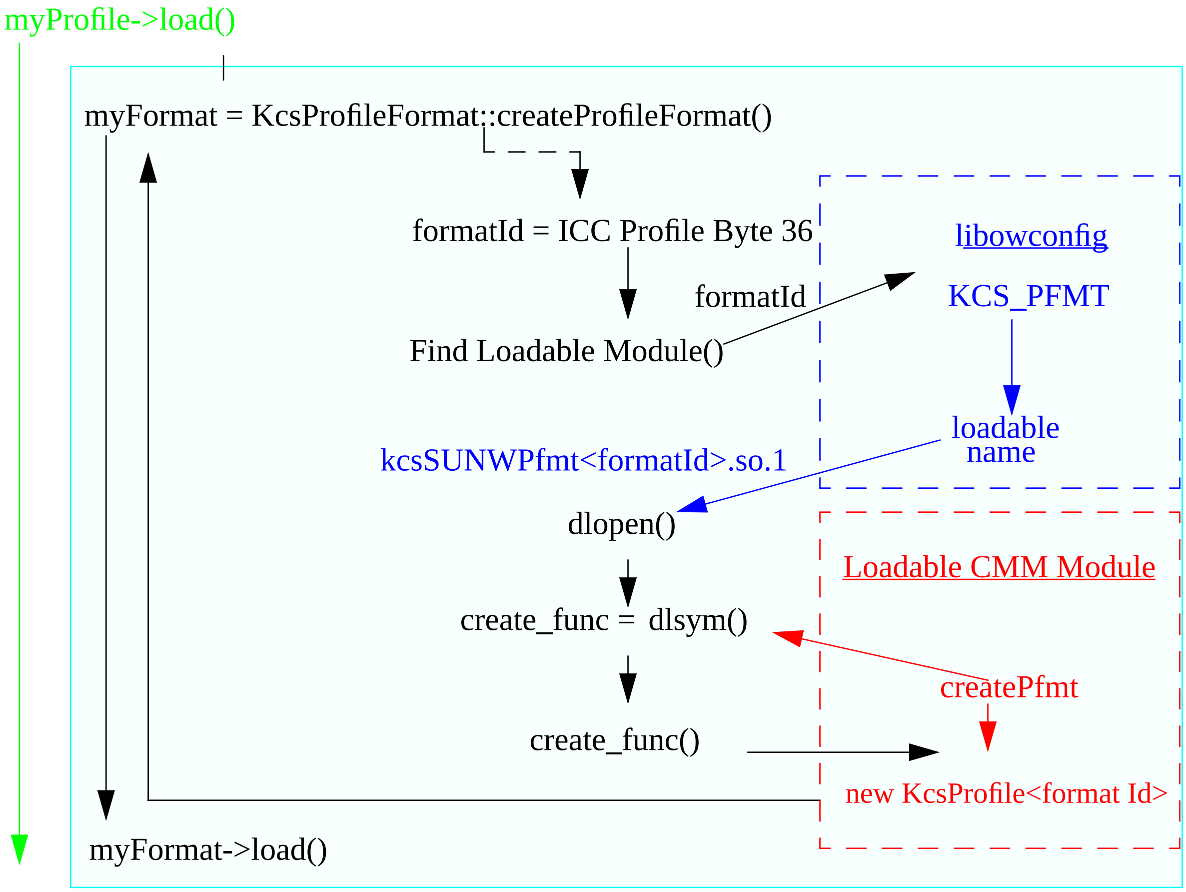

Figure 3-5 Creating a `KcsProfileFormat` Object

The `KcsProfile` object creates a `KcsProfileFormat` object pointer using `createProfileFormat()` which searches the `OWconfig` file for loadable entries based on the profile format bytes. For ICC profiles this is always `acsp`. Once the `KcsProfileFormat` object is created, the library generates a `KcsAttributeSet` object and an array of `KcsXform` objects.

Loading a `KcsProfileFormat` *Object*

The pointers to objects contained within the `KcsProfileFormat` object load themselves using the `KcsChunkSet` class. The following diagram illustrates the process.

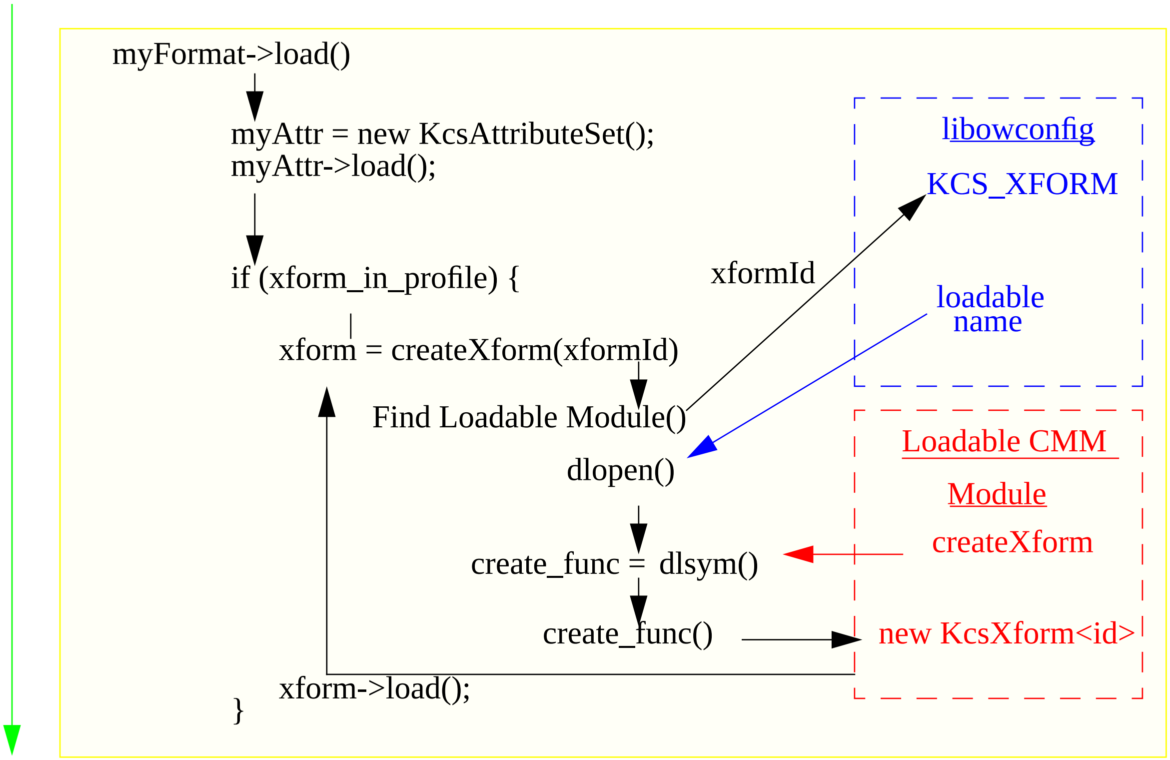

Figure 3-6 Loading a `KcsProfileFormat` Object

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 profile is successfully loaded, the number of entries in the global profile array is incremented, and the profile Id is returned to the application.

Loading an X11 Window System Profile

In this 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 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 on page 40, 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

The following API code example shows you how 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 `KcsConnectProfiles()` API call is implemented as follows:

1. Get a new valid index with `getNewValidProfileIndex()` for the connected profile.
2. The new connected profile needs a `KcsIO` class to handle its input and output. This is currently only stored in memory, so a `KcsMemoryBlock` object is created.
3. A `KcsProfile` object is created that can link together sequences of profiles.
4. Each profile in the sequence is then attached with `attach()` to the newly created `KcsProfile` object. `attach()` reference counts the objects. All classes are reference counted through inheritance from the `KcsShareable` class.
5. Once attached, the attributes of the two profiles are composed into a single set of attributes. The `KcsAttributeSet` object composes the attributes from the two `KcsProfile` members in the array into the newly created profile object.
6. The `KcsXform` array is linked so that the “into” and “out of” profile connection space (PCS) xforms of each `KcsProfile` 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, the new profile Id is returned to the calling application for later reference and the classes are generally cleaned up.

Evaluating Data Without Optimization

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

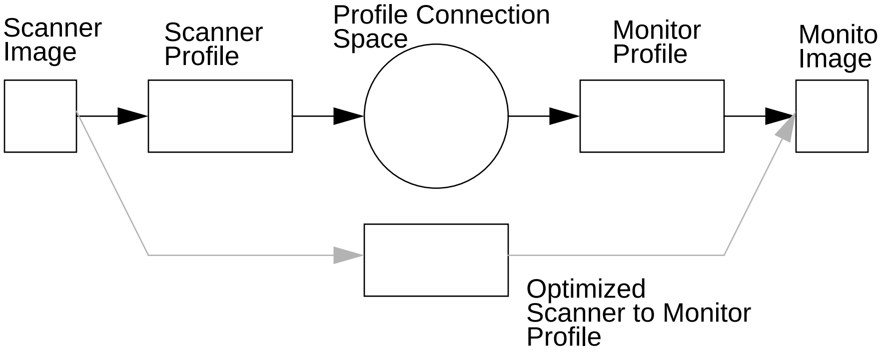

Figure 3-7 Optimized Versus Unoptimized Evaluation

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 API code excerpt below 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”);
```

Code Example 3-5 Evaluating Data Without Optimization *(Continued)*

```
    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 is the process of reducing multiple transforms into a single transform. The KCMS API code excerpt below 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, only if the reference count drops to zero will the memory be released.

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

Attributes

The following examples show you how to get and set attributes.

Setting an Attribute

When setting an attribute, the `KcsProfile` in the global array passes the setting of the attribute to the `KcsAttributeSet` object contained in its `KcsProfileFormat` object. This is illustrated in Figure 3-2 on page 36 and in the following KCMS API code excerpt.

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);
```

Code Example 3-7 Setting an Attribute

```
    fprintf(stderr, “unable to set whitepoint: %s\n”, errDesc.desc);
    KcsFreeProfile(profileid);
    return (-1);
}
```

Getting an Attribute

When getting an attribute, the `KcsProfile` in the array passes the getting of the attribute to the `KcsAttributeSet` object, replacing `set` with `get`, contained in its `KcsProfileFormat` object. This is illustrated in Figure 3-2 on page 36 and in the following KCMS API code excerpt.

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 API calls: `KcsCreateProfile()`, `KcsUpdateProfile()`, `KcsSetAttribute()`, and `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 three types of devices: scanners, monitors, and printers. It is your decision 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.

Attributes are set using the normal mechanisms. The following is a KCMS API code excerpt showing characterization and calibration.

Code Example 3-9 Characterization and Calibration

```
/* Fill out the measurement structures with dummy data*/
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;/*sample data in Luminance_flat_out array */
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;

status = KcsUpdateProfile(profileid, NULL, calData, NULL);
if (status != KCS_SUCCESS) {
```

Code Example 3-9 Characterization and Calibration *(Continued)*

```
    status = KcsGetLastError(&errDesc);
    printf(“UpdateProfile error: %s\n”, errDesc.desc);
    KcsFreeProfile(profileid);
    exit(1);
}
free(calData);
```

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 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. You do this with the same mechanism as that described in steps 2 to 5 of “Loading a Profile” on page 34.

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) {
```

Code Example 3-11 Saving a Profile to a Different Description

```
        status = KcsGetLastError(&errDesc);
        printf(“SaveProfile error: %s\n”, errDesc.desc);
    }
```

`KcsIO` *Derivative* 4

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

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 “OWconfig File Structure” on page 24).

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;
    if (libMinor < KCS_MINOR_VERSION)
```

Code Example 4-1 `KcsIO` Class Entry Points Example *(Continued)*

```
    //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;
}
    if (libMinor < KCS_MINOR_VERSION)
```

Member Function Override Rules

The following table tells you which `KcsIO` member functions you must override, can override, and should not 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

	Override Rules		
Member Function	**Must**	**Can**	**Do Not**
`absRead()`			X
`absWrite()`			X
`copyData()`			X
`createIO()`			X
`getEOF()`	X		

Table 4-1 `KcsIO` Member Function Override Rules *(Continued)*

	Override Rules		
Member Function	**Must**	**Can**	**Do Not**
`getOffset()`			X
`getType()`	X		
`isEqual()`	X		
`KcsIO()`	X		
`setCursorPos()`	X		
`setEOF()`	X		
`~KcsIO()`		X	
`relRead()`	X		
`relWrite()`	X		
`replaceData()`			X
`setOffset()`		X	

The `KcsSolarisFile` *Derivative as an Example*

The `KcsSolarisFile` class is a derivative of the `KcsIO` class. Any `KcsSolarisFile` files (or any of the other `KcsIO` derivatives) are good sources of example code for creating a `KcsIO` derivative. The on-line files `/usr/opt/SUNWddk/kcms/src/kcssolfi.cc` and `kcssolfi.h` 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.

`KcsProfile` *Derivative* 5

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

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

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

Code Example 5-1 `KcsProfile` Class Entry Points Example

```
/* Sample entry points for a new profile derivative */
extern “C” {
    extern long    KcsDLOpenProfCount;
    KcsProfile *   KcsCreateProf(KcsStatus *aStat, KcsIO *aIO);
    KcsProfile *   KcsCreateProfBlnk(KcsStatus *aStat,
                       KcsId aCmmId, KcsVersion aCmmVersion,
                       KcsId aProfId, KcsVersion aProfVersion);
    KcsStatus      KcsCleanupProf();
};
/* 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)
{
```

Code Example 5-1 `KcsProfile` Class Entry Points Example *(Continued)*

```
    //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, can override, and should not 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 `kcsprofi.h` header file and the *KCMS CMM Reference Manual*.

Table 5-1 `KcsProfile` Member Function Override Rules

	Override Rules		
Member Function	**Must**	**Can**	**Do Not**
`connect()`		X	
`createEmptyProfile()`		X	
`createProfile()`		X	
`evaluate()`		X	
`getAttribute()`		X	
`getFormat()`			X
`getOpAndCont()`			X
`getXform()`			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	
`setXform()`		X	
`unload()`		X	
`updateMonitorXforms()`		X	
`updatePrinterXforms()`		X	

Table 5-1 `KcsProfile` Member Function Override Rules *(Continued)*

	Override Rules		
Member Function	**Must**	**Can**	**Do Not**
`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)

The attribute set is represented by the `KcsAttrAttributesSet` object. There are `getAttribute()` and `setAttribute()` methods in the `KcsProfile` class that map directly to `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 following list of attribute set values are overridden by the base class 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. It uses a `KcsProfile` class derivative; otherwise, it is passed to the `KcsAttrAttributesSet` object.

`KcsProfileFormat` *Instance*

In addition to attributes and transforms, the `KcsProfile` base class uses a `KcsProfileFormat` instance for:

- Making the version number of profile data in static store transparent
- Special formats that a CMM might need
- Compatibility with new and old supported profile formats

The `KcsProfileFormat` object supports a consistent interface to attributes and transforms 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.

Transforms

Transforms are represented by an array of pointers to instances of the `KcsXform` class hierarchy. This array is indexed by `KcsXformTypes`. The logical transform types are listed in Table 5-2.

The ICC specification defines the profile color space (PCS) which is equivalent to RCS.

Table 5-2 Logical Transform Types

`XformType` **Values**	**Logical Transform 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.

Table 5-2 Logical Transform Types *(Continued)*

`XformType` **Values**	**Logical Transform Type**	**Description**
`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 transform, the KCMS framework defaults to connecting the outof-RCS to the into-RCS transformations, which generates an RCS-to-RCS transform that approximates the simulation. This results in a clip close to the simulation normally seen on devices. Currently, profiles do not supply simulation-RCS transforms by default. This connection technique fails on profiles that perform gamut-mapping; therefore, profiles that do gamut-map, should include one of these transforms.
`KcsXftFwdGamut = 5`	gamut-test-RCS	Provides all gamut testing for the device. The gamut-test-RCS transform 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.
`KcsXftRcsSimulate` = 8	complete-simulate transform	Maps the pixels from the source device through the destination device, its simulate transform, 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. The ICC profile format describes two sets of types: RCS or PCS. However, the KCMS framework supports a non-RCS model where there is no RCS. The only mandate is that CMMs support a number of standard references and that the color spaces match between connections. (Even this mandate is not strictly applied since the base class `connect()` method automatically inserts profiles if that creates a match).

Instead of referring to the into-RCS transform as going from a non-RCS into a RCS, the ICC 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.

Transform 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

There are two types of constructors: the *I/O-based constructor* and the *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.

Creators

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

Save Methods

There are two types of save methods: the *blind save* and the *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. 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 transforms.

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, “KcsIO Derivative,” 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 transforms 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 5-1, the connected sequence shares the Xforms (through the class) of those in the list.

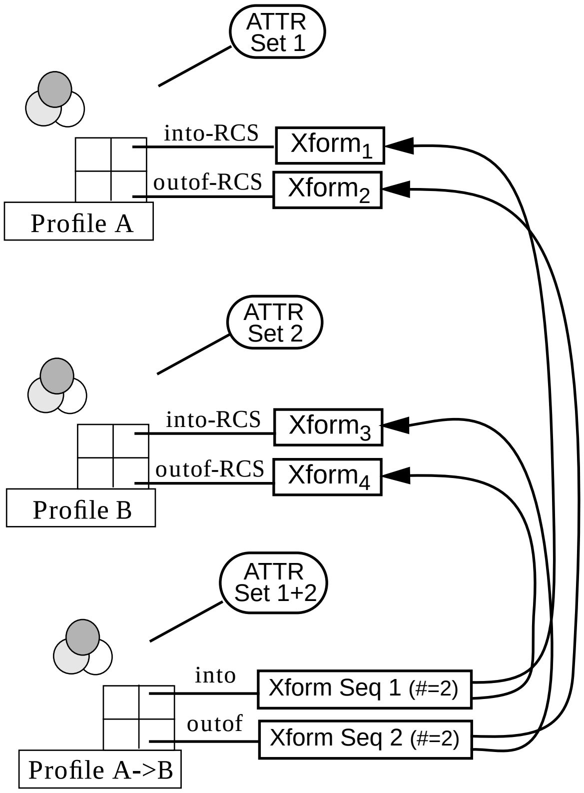

Figure 5-1 Sequences Sharing Xforms

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 method uses sequences of transforms to fill in the `results` transform array. The sequences are generated based on the content of the profile in the list `sequence`. As shown in Figure 5-1, if you gave a list consisting of an input device with an output device as the only profiles listed, the method takes the into-RCS transform of the input device, connects it with the outof-RCS transform 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 5-1 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 transform to accomplish the simulation part of the chain. If `connect()` doesn't find one and the outof-RCS and into-RCS transforms of the simulated device are available, it makes a simulation sequence that contains these transforms in place of the simulation-RCS transform.

Examples

Use Figure 5-2 with the two examples to better understand the `connect()` method and RCS simulation.

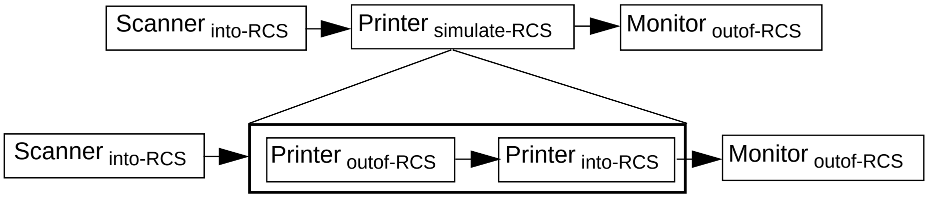

Figure 5-2 Into- and Out-of-RCS Transformations

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

The base class implementation of `updateScannerXforms()`, `updateMonitorXforms()`, and `updatePrinterXforms()` returns errors indicating that this particular profile does not support calibration or characterization, depending upon which data is 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.

`KcsProfileFormat` *Derivative*

6

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, transforms, loading and what you need to consider with a protected `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

```
// External entry points for runtime derivation
extern “C” {
    extern long       KcsDLOpenPfmtCount;
    KcsProfileFormat* KcsCreatePfmt(KcsStatus *aStat, KcsIO *aIO);
    KcsProfileFormat* KcsCreatePfmtBlnk(KcsStatus, *aStat,
                          KcsId aCmmId, KcsVersion aCmmVersion,
                          KcsId aProfId, KcsVersion aProfVersion);
    KcsStatus         KcsInitPfmt();
    KcsStatus         KcsCleanupPfmt();
}

extern long KcsDLOpenPfmtCount = 0;

/* Global initialization - constructor can be used */
KcsStatus
KcsInitPfmt(long libMajor, long libMinor, long *myMajor, long *myMinor)
{
```

Code Example 6-1 `KcsProfileFormat` Class Entry Points Example *(Continued)*

```
    // 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)
{
```

Code Example 6-1 `KcsProfileFormat` Class Entry Points Example *(Continued)*

```
    //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, can override, and should not 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**	**Do Not**
`deleteXform()`		X	
`dirtyAttrCache()`		X	
`dirtyXformCache()`		X	
`generateLoadWhat()`		X	
`generateXformAttributes()`		X	
`getCMMid()`			X
`getObject()`		X	
`getObject()`		X	
`getSaveSize()`		X	
`getTheCMMId`		X	
`getTheCMMVersion()`		X	
`getTheInfo()`		X	
`getTheProfileFormat()`		X	
`getTheProfileVersion()`		X	
`initEmptyFormat()`		X	
`isSupported()`		X	

Table 6-1 `KcsProfileFormat` Member Function Override Rules *(Continued)*

	Override Rules		
Member Function	**Must**	**Can**	**Do Not**
`KcsProfileFormat()`	X		
`~KcsProfileFormat()`		X	
`load()`		X	
`loadObjectMap()`		X	
`postAttrCompose()`		X	
`save()`		X	
`saveNew()`		X	
`saveObjectMap()`		X	
`setCmmId()`		X	
`setObject()`		X	
`setObject()`		X	
`unload()`		X	
`unloadWhenMatch()`		X	

Attributes

All attributes of a profile are in the `KcsAttributeSet` object returned from the `getObject()` method. 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 transforms. Attributes associated with individual transforms 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 those of the transform’s. The `KcsProfile` base class informs the xforms it loads about any attributes needed to construct themselves.

Transforms

As many transform slots exist as there are valid operations for a profile. Use the `getObject(KcsXformType)` overloaded method to retrieve the correct `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 transform 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 transform 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 transform 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 `KCS_PFMT_NO_DATA_SUPPORT_4_REQUEST`, 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`.

`KcsXform` *Derivative* 7

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

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

```
//External entry points for runtime derivation
extern “C”{
    extern long    KcsDLOpenXfrmCount;
    KcsXform*      KcsCreateXfrm(KcsStatus *aStat,
                       KcsChunkSet *aChunkSet,
                       KcsChunkId aChunkId,
                       KcsAttributeSet *aAttrSet);
    KcsStatus      KcsInitXfrm();
    KcsStatus      KcsCleanupXfrm();
}
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);
```

Code Example 7-1 `KcsXform` Class Entry Points Example *(Continued)*

```
    //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, can override, and should not 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 `KcsXform` Member Function Override Rules

	Override Rules		
Member Function	**Must**	**Can**	**Do Not**
`compose()`		X	
`connect()`		X	
`connectSink()`		X	
`connectSource()`		X	
`convertXform()`		X	

Table 7-1 `KcsXform` Member Function Override Rules *(Continued)*

	Override Rules		
Member Function	**Must**	**Can**	**Do Not**
`eval()`	X		
`getAttrSet()`		X	
`getComponentDepth()`			X
`getLoadOrder()`		X	
`getNumComponents()`			X
`getSaveOrder()`		X	
`KcsXform()`	X		
`~KcsXform()`		X	
`kindOfTransform()`			X
`loadU()`		X	
`numberOfCallbacks()`		X	
`optimize()`		X	
`saveU()`		X	
`setAttrSet()`		X	
`setCallbackInterval()`		X	
`setComponentDepth()`		X	
`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 a certain uniform set of functionality. You can do this in C++ by having a transform base class with pure virtual methods. Each technology is implemented in a derived class that *must* implement the required virtual methods.

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

Xform Attributes

`Xforms` contain their own `KcsAttributeSets`. They are passed in through all constructors and default to `NULL`. The `KcsAttributeSets` are copied and 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 xform. 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

Transform 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 extra tables for quality purposes and 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 88 for information on how a derivative always composes and keeps that xform for evaluation. Also note that it keeps the original xforms in the list unless it is also told to optimize for size, after which it will get rid of them.

If you optimize something away and then subsequently save it, that pruned data may not be replacable. 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 derivation as well.

Optimization must be defined by the derivative if that functionality is needed. Only the derivative instances understand how best to optimize. The derivation 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 feel it is appropriate, 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 xform) 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 xform 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 bitsets and are:

- Universal

  Xform saves and loads an industry-standard format—ICC 3.0.

- Private

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

- Universal as Private

Xform 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 type sets its own bit and the Universal bit since it saves universally.

Three choices are available with an extensible protocol in which:

- Derivatives are passed what the caller has as possibilities.
- Derivatives only reply with 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 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 3.0 profile format derivative calls `KcsXform` during its save to convert the xform into the appropriate ICC transform tag. If not overridden, the `KcsXform` base class converts the xform 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 is when the chunk set and chunk Id associated with the instance are used to save. The derivative only needs to package all of it's data into a contiguous piece of memory and pass the address and its `chunkId` 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 `chunkId`. The only caveat is that the instance must then place all of those `chunkIds` 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 of the pieces are not needed all of the time. This is the overall approach taken by the KCMS framework where the `KcsProfile` class has a table of `chunkId`s, 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 xform data. The `InterColorProfileFormat` asks for the load order and gives a list of universal plus private. The UCP derivative responds with `universalAsPrivate`. Since the derivative knows that UCPs can do this, it asks anything that does not do universal to convert themselves into a UCP. This follows the second way to break an obligation since the `InterColorProfileFormat` actually converts the xform to another kind and saves the converted one. It never saves the original.

The `typdef`s 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 one of itself from any other (`KcsXform` *) derivative. This is done by calling the `compose()` method that takes a (`KcsXform` *) and returns a (`KcsXform **`). Supply a callback function because this can be a slow operation.

The KCMS framework uses this protocol to implement a `sequence` xform derivative that can take many xforms 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 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 has encapsulated 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 (called in-place transformations). In this case, the `eval()` method detects it is the same buffer and optimizes for performance. These 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 will be moved and transformed from RGB to CMYK as well as have its composition transformed from planar to chunky. The evaluation methods will do so with minimal steps. The evaluation is most efficient when given large buffers of data to transform.

If the transform is not compatible with the layout(s), it returns an error. For example, if an RGB->CMYK `KcsXform` * was given two three-component descriptions, it would return an error since the destination was expecting four-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 xform derivative.

Evaluation Helper Methods

There is only one `eval()` method that is pure virtual. It is the one with two pixel layouts and a callback. Other `eval()` methods are overloaded in the base 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 and then 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, then 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 `KcsXforms` to act as one `KcsXform`. It is an alias to an ordered transform 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 transforms 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:

- 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 `xforms`, but it also accepts an array of `KcsXform` *s and `count` so that you can generate sequences from scratch.

Saving

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

When requested to save in universal format, the sequence does a composition that generates one xform that is saved in universal format.

Loading and Constructing the List

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

Connections

Connection is the only reason the `KcsXformSeq` class exists. No other support in the KCMS framework for connections exist except the `KcsXformSeq` class. The base `KcsXform` class uses a sequence derivative in its `connect()` method.

To make a connection, call the constructor that takes a list of transform pointers, or create a sequence of 0s and edit the transforms list with the `list()` methods (or use a combination of the two). See “Validation” on page 91 for a description of the connection method.

Optimization

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

The `KcsXformSeq` class performs composition by asking each transform 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, one `KcsTechUCP` can be generated 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 transform it constructed when told to optimize. It keeps a pointer to that optimized transform in its private section. When asked to evaluate, it passes the information down to the optimized transform.

Unoptimized evaluation is used when the sequence is not optimized. This implementation evaluates the data through the list of xforms sequentially. Between transforms 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 transforms, depending on the layout of the data. They are swapped between `eval()`s. If the composition of the transforms 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 a transform's output buffer is compatible with the next transform's input buffer. This can be optimized further if all buffer layouts describe a buffer that is compatible with the destination buffer supplied by the caller. If this is the case, the buffer pointers point to the destination buffer described. And if the caller has its source and destination buffer the same, everything ultimately uses one buffer. Such buffers are `KcsMemoryBlock`s that can be resized.

Validation

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

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

The List

The list of xforms 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 and you must pass the last parent found into the next call of this function and use the same object for invocations of this method. It returns `KCS_END_OF_XFORMS` when it reaches the end of the xforms in the sequence. All `getNextXform()` calls are sequential. Any sharing of an object must take this into account; otherwise, two different results may occur if the gets are not synchronized. It 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 xform types: `A->GetNext()`. If called (starting with a `NULL` parent `**`) until it returns `KCS_END_OF_SEQUENCE`, it returns xforms 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 xforms in the following order: `c`, `d`. It also skips over all sequences of 0 xforms as if they are not even there.

`KcsStatus` *Extension*

8

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)`, `msgfmt(1)`, and the *Solaris 2.4 Developer's Guide to Internationalization* for detailed 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.

Naming and Installing Profiles A

Any profile you want to include in the KCMS library must be named according to specified conventions to avoid name clashes and promote portability. This appendix tells you to how name and install your profile so that it can be 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>*

The following table describes the fields in the profile filename:

Table A-1 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.
device	Unique string identifying the device or color space. See Table A-2 for devices supported by Solaris.
type	ICC profile format standard filename suffixes; see Table A-3.

Supported Device

The following table lists the types of devices by manufacturer and model and profile filename. All of these devices are supported by the KCMS framework.

Table A-2 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`
Input	UMAX PowerLook Scanner	`kcmsEKumax_a.inp`
	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`
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`
	Kodak XL7700/XL7720 Printer	`kcmsEKx17700.out`
	Kodak XKS8300 Printer	`kcmsEKxls830.out`

Profile Filename Suffixes

The following table describes the various filename suffixes for profiles.

Table A-3 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

To install profiles, you must first locate them. Search for profiles in the following order:

1. Local current directory
2. Directories in `KCMS_PROFILES` environment variable

   A colon-separated list of directories where profiles reside. You can do this on a per-user or work-group basis.

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

   Local or machine-specific copies of *configured profiles*, for example, X Window System visual profiles.

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

Note – All profiles for distribution (whether you create it or it is supplied with Solaris) should be written as superuser and read only.

Index

Copyright 1996 Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, Californie 94043-1100, U.S.A.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 de sa documentation associée 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.

Des parties de ce produit pourront être dérivées du système UNIX® licencié par Novell, Inc. et du système Berkeley 4.3 BSD licencié par l'Université de Californie. UNIX est une marque enregistrée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd. 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.

Sun, Sun Microsystems, le logo Sun, Solaris, SunSoft, le logo SunSoft, et AnswerBook sont des marques déposées ou enregistrées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC, utilisées sous licence, sont des marques déposées ou enregistrées 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. OPEN LOOK est une marque enregistrée de Novell, Inc. PostScript et Display PostScript sont des marques d'Adobe Systems, Inc., lesquelles pourront être enregistrées dans des juridictions compétentes.

Les interfaces d'utilisation graphique OPEN LOOK® et Sun™ ont été développées 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 aussi 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.

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

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