KCMS CMM Reference Manual

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

Part No: 806–1519
May 6 1999

Contents

Preface

The *KCMS CMM Reference Manual* provides detailed descriptions of the Kodak Color Management System (KCMS™) foundation library. This library is a graphics porting interface (GPI) implemented in C++ for creating KCMS color modules. A set of C++ classes are supplied that can be derived from and extended. You can add attributes to the current list, incorporate new color processing technology, or support alternate profile formats.

Use this book with the *KCMS CMM Developer's Guide* which provides an an in-depth view of the KCMS framework and how the API works with this GPI, how to derive from each C++ class, how to create a dynamically loadable CMM, and how to add profiles to the system.

Who Should Use This Book

Use this book if you are interested in:

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

Before You Read This Book

You should be familiar with the Kodak Color Management System (KCMS) API which is part of the SDK; see the *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 in man Pages(1): User Commands and man Pages(3): Library Routines:

- ld(1)
- dlopen(3)
- dlclose(3)
- dlerror(3)
- dlsym(3)
- OWconfigInit(3)
- OWconfigGetAttribute(3)
- OWconfigFreeAttribute(3)
- OWconfigClose(3)

A basic understanding of color science is also assumed; references are included in the Bibliography of the *KCMS Application Developer's Guide.*

See the on-line SUNWrdm packages for information on bugs and issues, engineering news, and patches. For Solaris installation bugs and for late-breaking bugs, news, and patch information, see the *Solaris 2.6 Installation Instructions (SPARC Platform Edition)* and the *Solaris 2.6 Installation Instructions (Intel Platform Edition)* manuals.

For SPARC systems, consult the updates your hardware manufacturer may have provided also.

How This Book Is Organized

Note - Each chapter in this book describes relevant classes in the KCMS architecture. Although the DDK header files may include additional information (private and methods and other internal interfaces) for each class, be aware that the chapters in this book present all the methods you need to be concerned about to write your CMM.

The chapters are organized as follows:

Chapter 1 describes the `KcsShareable` class.

Chapter 2 describes these I/O classes: `KcsIO`, `KcsFile`, `KcsMemoryBlock`, `KcsSolarisFile`, and `KcsXWindow`.

Chapter 3 describes in detail the `KcsChunkSet` class.

Chapter 4 describes in detail the `KcsLoadable` class.

Chapter 5 describes in detail the KcsProfile class.

Chapter 6 describes in detail the `KcsProfileFormat` class.

Chapter 7 describes in detail the `KcsAttributeSet` class.

Chapter 8 describes the member functions in the `KcsXform` class.

Chapter 9 describes the member functions in the `KcsXformSeq` class.

describes in the `KcsStatus` class.

Related Books

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

- *KCMS Application Developer's Guide*
- *ICC Profile Format Specification* (located on-line in `/opt/SUNWsdk/kcms/doc/icc.ps`). For the most current version of the ICC specification, see the web site at `http://www.color.org`.

Ordering Sun Documents

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

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

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

What Typographic Changes and Symbols Mean

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

Shell Prompts in Command Examples

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

TABLE P–2 Shell Prompts

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

`KcsId` Naming Convention

Each class contains a `KcsId` that uniquely identifies that class. Most `KcsId`s are defined in the `kcsids.h` header file. The naming conventions shown in the following table are used for the `KcsId` for each class in the KCMS framework. The `#defines` are helpful in `switch` statements.

TABLE P–3 `KcsId` Naming Conventions

Item	Convention	Examples
`const`	`Kcs`*<Base Class Id><Derived Class Id>*`Id`	`KcsSharIOId`
`#define`	`Kcs`*<Base Class Id><Derived Class Id>*`Idd`	`KcsSharIOIdd`

Equivalent Terms In This Book

For historic reasons, this book uses several equivalent Kodak and International Color Consortium (ICC) terms. The terms evolved at different times. Development of the

ICC specification introduced new ICC terms with meanings the same as (or similar to) already existing Kodak terms.

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

TABLE P–4 Equivalent ICC and Kodak Terms

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

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

New Features

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

KCMS is Multithread Safe

In this release, KCMS supports multithreaded programs.

`OWconfig` File Modification

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

CHAPTER 1

KcsShareable Class

This chapter describes the KCMS framework's primary base class, the KcsShareable class. This class is at the top of the KCMS class hierarchy. The KcsShareable class allows any of its derivatives to be shared.

As you read this chapter, you will find it helpful to have access to the kcsshare.h header file.

The constant and #define identifiers for this class are defined in the kcsids.h header file as:

```
const KcsId KcsRlocSharLoadId = {(0x53686172UL)}; /* 'Shar' */
#define KcsRlocSharLoadIdd (0x53686172UL) /* 'Shar' */
```

The protected and public members are described.

Protected Members

The KcsShareable class provides the following protected members.

TABLE 1–1 KcsShareable Protected Members

Protected Member	Description
`virtual ~KcsShareable();`	Destructor. A protected member.

Public Members

The KcsShareable class provides the following public members.

TABLE 1–2 KcsShareable Public Members

Public Member	Description
`virtual KcsShareable *attach( long howMany = 1, KcsAttachType aAttachFlag = KcsAttMem, KcsStatus *aStatus = NULL);`	Use this method when you want to share an object already allocated. It returns a pointer to the shared object. All shared objects *must* `dettach()` instead of using delete. `aAttachFlag` and `aStatus` are only used with the `KcsLoadable` class override.
`void dettach(long howMany = 1, KcsAttachType aDettachFlag = KcsAttMem, KcsStatus *aStatus = NULL);`	Deletes an object. The actual deallocation only happens when no other object is sharing this object. `aDettachFlag` and `aStatus` are only used with the `KcsLoadable` class override.
`static long getGlobalCount();`	Returns the total number of objects sharing any other objects in the system. Use for debugging.
`long getUseCount() {return(useCount);};`	Returns the current number of objects sharing this object. Use for debugging.
`KcsShareable(KcsStatus *status, long nUse = 1);`	Constructor.

CHAPTER 2

I/O Classes

This chapter describes the following KCMS input/output (I/O) classes:

- KcsIO
- KcsFile
- KcsMemoryBlock
- KcsSolarisFile
- KcsXWindow

The KcsIO class provides a common interface for I/O operations such as read and write. The KcsIO class is a derivative of the KcsShareable class (see Chapter 1"). The KcsFile, KcsMemoryBlock, KcsSolarisFile, and KcsXWindow are derivatives of the KcsIO class. These derivatives provide I/O for more specific types of data storage.

As you read this chapter, you will find it helpful to have access to the following header files:

- kcsio.h, kcsfile.h, kcsmblk.h, kcssolfi.h, and kcsxwin.h
- kcsshare.h and kcsids.h

KcsIO Class

With a common interface, the KcsIO class maintains device-, platform-, and transport-independent I/O functionality for all derivatives.

The header file for the class is kcsio.h. The constant and #define identifiers for this class are defined in the kcsids.h header file as:

```
const KcsId KcsSharIOId = {(0x494F0000UL)}; /* 'IO' */
```

```
#define KcsSharIOIdd (0x494F0000UL) /* 'IO' */
```

The enumerations and protected and public members are described, as well as the member function override rules when deriving from this class.

Enumerations

The KcsIO class provides the following enumerations.

TABLE 2–1 KcsIO Enumerations

Enumeration	Description
`enum KcsIOPosition {KCS_OFS, KCS_BOO, KCS_CUR};`	Used for calls to `setCursorPos( )`. KCS_OFS is relative to beginning of I/O object+baseoffset. KCS_BOO is relative to beginning of the I/O object. KCS_CUR is relative to present I/O cursor.

Protected Members

The KcsIO class provides the following protected members.

TABLE 2–2 KcsIO Protected Members

Protected Member	Description
`KcsStatus aSysError(const char *callersName, const char *sysName, KcsStatus stat, const int sysErrCode);`	A protected member that returns a KcsStatus object that contains the OS error in the causingError data member. Use callersName for debugging; it is recommended that you change to a non-NULL value.

Public Members

The KcsIO class provides the following public members.

TABLE 2–3 KcsIO Public Members

Member Function	Description

`virtual`
`KcsStatus absRead(const long index,`
`  const long bytesWanted,`
`  void *buffer,`
`  const char *callersName = NULL);`
	Absolute read an I/O object already opened or allocated. Supply a count of number of bytes to read and a buffer.

`virtual`
`KcsStatus absWrite(const long index,`
`  const long numBytes2Write,`
`  const void *buffer,`
`  const char *callersName = NULL);`
	Absolute write to some number of bytes in `numBytes2Write` from the buffer to the data store at the current position of the cursor

`virtual` `KcsStatus copyData(KcsIO *anotherIO);`	Copies all data in I/O object to `anotherIO` object.
`static KcsIO * createIO(KcsStatus *aStatus,` `  const KcsProfileDesc *aDesc);`	Static method that creates an I/O object, by calling either a `KcsIO` derivative constructor within the KCMS library or a run-time loadable constructor.
`virtual` `KcsStatus getEOF(long *theEOF) = 0;`	Returns the end-of-file (EOF) position.
`virtual` `long getOffset();`	Returns the permanent offset into the I/O object..
`virtual` `KcsIOType getType() = 0;`	Returns the type of I/O object.
`virtual` `int isEqual(KcsIO *anotherIO) = 0;`	Determines if this I/O object and another I/O object are working on the same data stores.The base offsets must also be the same for them to return true.
`KcsIO(KcsStatus *status,` `  const unsigned long absBaseOffset = 0);`	Constructor that initializes the `baseOffset` data member with the values passed in.
`virtual ~KcsIO();`	Destructor.

`virtual`
`KcsStatus relRead(const long bytesWanted,`
`  void *buffer,`
`  const char *callersName = NULL) = 0;`
	Reads `bytesWanted` from the I/O object from the current position of the cursor. Positions the cursor after the last byte read.

`virtual`
`KcsStatus relWrite(const long numBytes2Write,`
`  const void *buffer,`
`  const char *callersName = NULL) = 0;`
	Relative write of the number of bytes in `numBytes2Write` from the `buffer` to the data store at the current position of the cursor. Positions the cursor after the last byte written.

TABLE 2–3 KcsIO Public Members *(continued)*

Member Function	Description

`virtual`
`KcsStatus replaceData(const unsigned long offset,`
`  const unsigned long oldSize,`
`  const void *buffer,`
`  const unsigned long newSize,`
`  const char *callersName = NULL);`
	Replaces bytes of different lengths in an I/O object. Specifies where to start writing and size of old and new data. If the old data is longer than the new data, the I/O object is compressed. If the new data is longer than the old data, everything after the old data is moved to the end of the I/O object. If an error occurs, the cursor is where it was before the error occurred.

`virtual`
`KcsStatus setCursorPos(long position,`
`  const KcsIOPosition mode= KCS_OFS,`
`  const char *callersName = NULL) = 0;`
	Sets the I/O object cursor to a caller-supplied `position`.
	Positions the I/O object cursor to a specific spot in the object. Mode is defined by the enum `KcsFilePosition`.

`virtual` `KcsStatus setEOF(long theEOF) = 0;`	Sets the EOF to a caller-supplied position. If the new EOF is greater than the old EOF, the storage for the new space is undefined.
`virtual` `void setOffset(long theOffset);`	Sets the base offset to a specified position.

Member Function Override Rules

The following table tells you which `KcsIO` member functions you must override and can override. The member functions indicated with an “X” in the Must column are required to derive successfully from this base class. Others can be used and not overridden.

TABLE 2–4 KcsIO Member Function Override Rules

Member Function	Override Rules	
	Must	**Can**
`getEOF()`	X	
`getType()`	X	

TABLE 2–4 KcsIO Member Function Override Rules *(continued)*

Member Function	Override Rules	
	Must	**Can**
isEqual( )	X	
KcsIO( )	X	
~KcsIO( )		X
relRead( )	X	
relWrite( )	X	
setCursorPos( )	X	
setEOF( )	X	
setOffset( )		X

Example

The following code shows you how to use a KcsIO derivative as a data member or as an object to pass into a function.

CODE EXAMPLE 2–1 KcsIO Example

```
//Read 4 bytes from KcsIO starting at byte 8.
 long getTheSecondLong(KcsStatus *aStat, KcsIO *aIO)
 {
  long badNumber = -1;
  long sSecondLong;

  if (aIO == NULL)
   return(badNumber);
  if ((*aStat = aIO->absRead(8, 4, &sSecondLong)) == KCS_SUCCESS)
   return(sSecondLong);
```

(continued)

```
  else
   return(badNumber);
 }
```

KcsMemoryBlock Class

The KcsMemoryBlock class is a memory-based I/O derivative of the KcsIO class. It provides a way to manipulate blocks of memory by creating a KcsIO object. You can *use* the protected and public member functions in this class in the implementation of your CMM; you cannot override any member function in this class.

The header file for the class is kcsmblk.h.

Note - It is highly recommended that you do *not* use any of the variables and functions for handle-based memory in the kcsmblk.h header file. Handle-based memory is not required on the Solaris system.

The constant and #define identifiers for this class are defined in the kcsids.h header file as:

```
const KcsId KcsIOMBlkId = {(0x4D426C6BUL)}; /* 'MBlk' */
#define KcsIOMBlkIdd (0x4D426C6BUL) /* 'MBlk' */
```

In addition to the KcsIO virtual member functions that must be overridden for a minimal KcsIO implementation, the KcsMemoryBlock class has member functions for manipulating blocks of memory. See Table 2–4 for a list of the member functions minimally required to derive from the KcsIO class.

The enumerations and protected and public members are described.

Enumerations

The KcsMemboryBlock class provides the following enumerations.

TABLE 2–5 KcsMemoryBlock Enumerations

Enumeration	Description
`enum KcsMemoryKind { KCS_APPLICATION_HEAP, KCS_SYSTEM_HEAP };`	`enum` used in `setCursorPos( )`. The object is in the application or system heap.

Protected Members

The `KcsMemboryBlock` class provides the following protected members

TABLE 2–6 KcsMemoryBlock Protected Members

Protected Member	Description
`long allocMe;`	Determines whether memory block is allocated.
`char* curPtr();`	Returns the current address of the memory block + position.
`long numElements;`	Number of elements if memory block is an array.
`long pos;`	Current position in memory block.
`long realEOF;`	Number of bytes actually written to memory file.
`long size;`	Number of bytes allocated to contain memory file.
`char *startPtr;`	Start of char-pointer-based memory block.

Public Members

The `KcsmemoryBlock` class provides the following public members.

TABLE 2–7 KcsMemoryBlock Public Members

Public Member	Description
`KcsStatus get(char* buf, long nbytes);`	Gets `nbytes` of data starting at the current cursor position. Post-increments the cursor position by `nbytes`.
`long getNumElements() {return numElements;}`	Returns the number of elements in the `KcsMemoryBlock` object as a `long`.
`long getSize();`	Returns the size of the `KcsMemoryBlock` object.
`KcsMemoryBlock(KcsStatus *status, char *start, long size, long numElements = 1, const unsigned long absBaseOffset = 0);`	Constructor. Specifies a character pointer in `start`, block size in `size`, and number of elements in `numElements`.
`KcsMemoryBlock(KcsStatus *status, long size, long numElements = 1, const unsigned long absBaseOffset = 0, const KcsMemoryKind kind = KCS_APPLICATION_HEAP);`	Constructor. Allocates and deallocates the memory.
`virtual ~KcsMemoryBlock();`	Destructor.
`long position() {return pos;}`	Returns the current cursor position.
`void position(long posit) {pos=posit;}`	Sets the current cursor position.
`KcsStatus put(char* buf, long nbytes);`	Puts `nbytes` of data into the `KcsMemoryBlock` object at the current cursor position. Post-increments the cursor position by `nbytes`.

Example

This example shows you how to change the size of memory with the `KcsMemoryBlock` class.

CODE EXAMPLE 2–2 Resizing Memory with `KcsMemoryBlock`

```
KcsStatus resizeIt()
 {
```

(continued)

```
  unsigned long sNewSize = 10;
  KcsStatus sStatus;
  KcsMemoryBlock *memBlock;
  char * buffer = {'a','b','c','d'};

  // create a new KcsMemoryBlock object
  memBlock = new KcsMemoryBlock(&sStatus,4);
  if (sStatus != KCS_SUCCESS)
    return (sStatus);
  // put the four bytes above into the new KcsMemoryBlock
  sStatus = memBlock->put(buffer,4);
  if (sStatus != KCS_SUCCESS)
    return (sStatus);
  // resize the KcsMemoryBlock from 4 to 10
  sStatus = memBlock->reSize(sNewSize);
  //Finished with the data
  delete memBlock;
  return (sStatus);
 }
```

KcsFile Class

The KcsFile class is a file I/O derivative of the KcsIO class. It provides a way to manipulate files by creating a KcsIO object. You can *use* the protected and public member functions in this class in the implementation of your CMM; you cannot override any member function in this class.

The header file for the class is kcsfile.h.

Note - It is highly recommended that you do *not* use any of the variables and functions for handle-based memory in the kcsmblk.h header file. Handle-based memory is not required on the Solaris system.

The constant and #define identifiers of this class as defined in the kcsids.h header file are:

```
const KcsId KcsIOFileId = {(0x46696C65UL)}; /* 'File' */
#define KcsIOFileIdd (0x46696C65UL) /* 'File' */
```

In addition to the KcsIO virtual functions that must be overridden for a minimal KcsIO implementation, the KcsFile class has member functions for reading from

and writing to a file. See Table 2–4 for detailed information on the virtual functions that are minimally required to derive from the KcsIO class.

This class does not have any protected members; the public members are described.

Public Members

The KcsFile class provides the following public members.

TABLE 2–8 KcsFile Public Members

Member Function	Description
`virtual long getFref();`	Gets the file description being used.
`KcsFile(KcsStatus *status);`	Constructor. Creates a file object without a file and offset. You must use `setFref()` to establish a link to a file before using any other methods.

`KcsFile(KcsStatus *status,`
`  const KcsFileId anFref,`
`  const unsigned long absBaseOffset = 0);`
	Constructor. Creates a file object with an open file and offset.

`virtual ~KcsFile();`	Destructor.
`virtual void setFref(long theFref);`	Sets the file to use and the offset.

Examples

The following examples show you how to use the KcsFile class.

Reading a File From a Specified Offset

This example shows you how to read a file from a specified offset with absRead(). See the kcsio.h header file for a description of absRead().

CODE EXAMPLE 2–3 Reading a File From a Specified Offset

```
KcsStatus readIt()
 {
  KcsStatus sStatus;
```

(Continuation)

```
  KcsFileId sFileRef;
  long index = 32;
  long number;

  // open the file, put the fileRef into sFileRef
  sFileRef = open (''Profile'', O_RDWR);
  if (sFileRef == -1)
    return (KCS_IO_ERROR);
  // create a file object
  file = new KcsFile(&sStatus, sFileRef, 0);
  if (sStatus != KCS_SUCCESS)
    return(sStatus);

  // using the file object, read from the file into a buffer.
  sStatus = file->absRead(index, sizeof(long), &number);
  delete file;
  close(sFileRef);

  return (sStatus);
 }
```

Writing to a File From the Last Cursor Position

This example shows you how to write to a file with `relWrite()`. See Table 2–3 for a full definition of `relWrite()`.

CODE EXAMPLE 2–4 Writing to a File From the Last Cursor Position

```
KcsStatus writeIt()
 {
  KcsStatus sStatus;
  KcsFileId sFileRef;
  long nbytes;
  char *buffer;

  // open the file, get a fileRef
  sFileRef = open (''Profile'', O_RDWR);
  if (sFileRef == -1)
    return (KCS_IO_ERROR);

  // create a file object
  file = new KcsFile(&sStatus, sFileRef);
  if (sStatus != KCS_SUCCESS)
```

(continued)

(Continuation)

```
    return(sStatus);

  // Allocate memory for the buffer, fill it with data.
  // Set nbytes to the length of the buffer.
  if ((buffer = (char*) malloc(nbytes)) == NULL)
    return (KCS_IO_ERROR);
  delete file;
  close(sFileRef);

  // using the file object, write the buffer to the file.
  sStatus = file->relWrite(nbytes, buffer);

  // Free the buffer's memory.
  free (buffer);
  delete file;
  close(sFileRef);

  return (sStatus);
}
```

KcsSolarisFile Class

The KcsSolarisFile class is a derivative of the KcsIO class. It is a Solaris-specific KcsIO class that provides member functions that:

- Open a file with a partial name
- Search through a list of known directories
- Check for string endings for filename suffixes (inp, mon, out, and spc)
- Access files on a remote machine

Note - The KCMS daemon, kcms_server, must be running to access remote files. Remote access is read only. See the kcms_server(1) man page.

The KcsSolarisFile class creates a pointer to a KcsFile or KcsRemoteFile object depending on the host location. The derived public methods (relWrite( ), relRead( ), getEOF( ), and setEOF( )) then call the KcsIO pointer to do the actual operation.

The header file for this class is `kcssolfi.h`.

The `const` and `#define` for this class are defined in the `kcsids.h` header file as:

```
const KcsId KcsIOsolfId = {(0x736f6c66UL)}; /* 'solf' */

#define KcsIOsolfIdd (0x736f6c66UL) /* 'solf' */
```

This class does not have any protected members; the public members are described.

Public Members

The `KcsSolarisFile` class provides the following public members.

TABLE 2–9 `KcsSolarisFile` Public Members

Public Member	Description
`virtual KcsIO* getIO();`	Returns the I/O pointer.
`KcsSolarisFile(KcsStatus *status, const char *filename, const char *hostname, const int oflag, const mode_t mode);`	Constructs a new I/O object pointer to a file (full path and suffix not needed) either a remote or local machine. The file is opened with the specified permissions. See the open(2) man page for information on oflag and mode.
`virtual ~KcsSolarisFile();`	Destructor.

`KcsXWindow` Class

The `KcsXWindow` class is a derivative of the `KcsIO` class. It provides an interface for the X11 Window System connection. It turns X11 information into filenames for access at known directories either on a local or remote system. The KCMS daemon, kcms_server(1) must be running to access remote files. Remote access is read only.

The `KcsXWindow` class creates a pointer to a `KcsFile` or `KcsRemoteFile` object depending on the host location which is derived from the X11 Window System information. The derived public members (`relWrite( )`, `relRead( )`, `getEOF( )`, and `setEOF( )`) then call the `KcsIO` pointer to do the actual operation.

The header file for the class is `kcsxwin.h`.

The const and #define for this class are defined in the kcsids.h header file as:

```
const KcsId KcsIOxwinId = {(0x7877696EUL)}; /* 'xwin' */
#define KcsIOxwinIdd (0x7877696EUL) /* 'xwin' */
```

In addition to the KcsIO methods overridden by this class, there are methods for creating filenames remotely or locally with X Window System information. See Table 2–4 for detailed information on the virtual functions that are minimally required to derive from the KcsIO class.

This class does not have any protected members; the public members are described.

Public Members

The KcsXWindow class has the following public members.

TABLE 2–10 KcsXWindow Public Members

Public Member	Description
`virtual KcsIO* getIO();`	Returns the I/O pointer.
`KcsXWindow(KcsStatus *status, const Display *dpy, const int screen, const Visual *visual, const long caller)`	Constructs a new IO object pointer to a profile connected to the machine and display. The specific X Window System profile name is constructed. The location is either a known local directory or a path specified by the KCMS_XTERMINAL_PROFILES environment variable.
`virtual ~KcsXWindow()`	Destructor.

Constructing a KcsXWindow Profile Name

The X Window System profiles are created with the KCMS configuration program kcms_configure; see the kcms_configure(1) on-line man page for more information. The KCMS Calibrator Tool (kcms_calibrate) supplies monitor calibration, as well as configuration of the X profiles. See the kcms_calibrate(1) man page for more information.

X Window System visual profiles follow this naming convention:

```
<Visual Class><Visual ID in Hex>:<Display #>.<Screen #>
```

For example, for the PseudoColor visual on display 0, screen 0, with Visual ID 0x20, has the following profile name: `PseudoColor0x20:0.0.`

The `Visual ID` is provided with the visual argument as well as an indicator to one of the visual names ("StaticGray", "GrayScale", "StaticColor", "PseudoColor", "TrueColor", or "DirectColor").

Similar entries exist for the other visuals. X11 Window System visual profiles are overwritten when a system is recalibrated after setup. The base uncalibrated monitor profiles are in the `/usr/openwin/etc/profiles` directory; therefore, you can always reset the system, if for some reason one of the per-machine profiles is corrupted.

CHAPTER 3

KcsChunkSet Class

This chapter describes the KCMS framework KcsChunkSet class. You can manage blocks (or chunks) of data arranged in tagged file format with Chunk classes. The ICC profile format is directly analogous to the KcsChunkSet. The KcsChunkSet class is derived from the KcsShareable class. See Chapter 1," for a description of the KcsShareable class.

As you read this chapter, you will find it helpful to have access to the following header files:

- kcsshare.h
- kcsio.h and kcsmblk.h
- kcschu.h and kcschunk.h

Note - It is highly recommended that you do *not* use any of the variables and functions for handle-based memory in the these header files. Handle-based memory is not required on the Solaris system.

The header file for the class is kcschunk.h. The constant and #define identifiers for this class are defined in the kcsids.h header file as:

```
const KcsId KcsSharChkSId = {(0x43686B53UL)}; /* 'ChkS' */
#define KcsSharChkSIdd (0x43686B53UL) /* 'ChkS' */
```

In addition to the KcsShareable methods overridden by this class, there are methods for managing chunks of data. The protected and public members are described.

Protected Members

The KcsChunkSet class has the following protected members.

TABLE 3–1 KcsChunkSet Protected Members

Protected Member	Description
`virtual KcsStatus readChunkBuffer(const KcsChunkId aChunkId, void *aChunk,unsigned long *aSizeRead, unsigned long aSizeWanted = 0);`	Reads a chunk of bytes indicated by a chunk Id. KCS_CHUNK_ID_ERR is returned if no chunk Ids match in the profile. Assumes the aChunk buffer is allocated. Get the buffer size with getChunkSize( ) which is defined in Table 3–2.
`virtual KcsStatus writeChunkBuffer(const KcsChunkId aChunkId, void *aChunk, const unsigned long aSize, KcsCompressBoolean aCompress = KcsCompress, KcsLocationEnum aLocationEnum = KcsLocationUnspecified, long aOffset = KcsNoOffsetSpecified);`	Writes a chunk of bytes to a new chunk in the profile, given a void *. Returns a new chunk Id. If a particular location or offset is specified, other chunks are moved to write this chunk at the specified location.

Public Members

The KcsChunkSet class has the following public members.

TABLE 3–2 KcsChunkSet Public Members

Public Member	Description
`virtual KcsStatus createChunkId(KcsChunkId *aNewChunkId, unsigned long aChunkSize = 0, KcsLocationEnum aLocationEnum = aKcsLocationUnspecified, long aChunkOffset = KcsNoOffsetSpecified, KcsRegisterBoolean aRegister = KcsIgnoreIdSpecified);`	Allocates `aNewChunkId` for a chunk introduced to the data type. Puts a new entry into the chunk map. The size of the chunk can be specified, or by default equals `0`. The chunk location can be specified in `aLocationEnum` and `aChunkOffset` or defaults can be used. Creates specific chunk ids with `aRegister`.
`virtual KcsStatus deleteChunk(const long aChunkId);`	Deletes the chunk (and its chunk map entry) identified by a `chunkId`. You are required to check whether any other objects are using this chunk.
`virtual unsigned long getChunkSetLength(KcsStatus *aStatus);`	Returns the length of the domain of the chunk set.
`virtual long getChunkSetOffset(KcsStatus *aStatus);`	Returns the offset of the beginning of the domain of the chunk set.
`virtual unsigned long getChunkSize(KcsStatus *aStatus, KcsChunkId aChunkId);`	Returns the chunk size identified by `aChunkId`. If the chunk is compressed, returns the uncompressed chunk size.
`virtual KcsIO * getIO();`	Returns the chunk set's I/O object.
`virtual KcsChunkId getTopChunkId(KcsStatus *aStatus, unsigned long aSize = 0, long aOffset = KcsNoOffsetSpecified);`	Returns the chunk Id of the top chunk which stores information about other chunks. Read this chunk first because it tells what is contained in the other chunks. Or you can specify the size of the chunk and the chunk's offset; by default, `aSize` = `0` and `aOffset` = `KcsNoOffsetSpecified`. If no top chunk exists , call `createChunkId( )`.
`virtual Boolean isEqual(KcsChunkSet *aNotherCS);`	Checks if this chunk set is the same as another chunk set.
`KcsChunkSet(KcsStatus *aStatus);`	Constructor without an I/O object.

TABLE 3–2 KcsChunkSet Public Members *(continued)*

Public Member	Description

`KcsChunkSet(KcsStatus *aStatus,`
`  KcsIO *aIoObj,`
`  unsigned long aChunkSetLength = KcsUseEOF,`
`  KcsSaveBoolean aSaveMapping =`
`  KcsSaveChunkSet,`
`  int Format = 0);`
	Constructor of a chunk set instance based on an I/O object.
	
	`aChunkSetLength` can be set to the length of the static store managed by the chunk set if not equal to the length managed by the I/O object. Otherwise, by default, `aChunkSetLength` is set to the end of the static store managed by the I/O object.
	
	When a particular file format does not allow writing out the chunk map, `aSaveMapping` is `False`; otherwise, `aSaveMapping` is, by default, `True` indicating that the chunk containing the chunk map should be saved.

`~KcsChunkSet();`	Destructor.

`KcsStatus`
`readChunk(const KcsChunkId aChunkId, void *aChunk,`
`  unsigned long *aSizeRead,`
`  unsigned long aSizeWanted = 0);`
	Reads a chunk set.

`virtual KcsStatus` `setChunkSetOffset(long aOffset);`	Sets `aOffset` to the beginning of the domain of the chunk set.
`virtual KcsStatus` `setChunkSetPrivateOffset(const long aOffset)`	Sets the chunk set's private offset.
`virtual KcsStatus` `setIO(KcsIO *aIoObj);`	Sets the chunk set's I/O object.

`virtual unsigned long`
`updateChunkUseCount(KcsStatus *aStatus,`
`  KcsChunkId aChunkId, long aDelta,`
`  unsigned long aComparisonCount);`
	Calls the `ChunkMap` object to change the use count of a particular chunk. Compares the use count with an expected value, `aComparisonCount`.

`KcsStatus`
`writeChunk(KcsChunkId aChunkId,`
`  void *aChunk,`
`  const unsigned long aSize,`
`  KcsCompressBoolean aCompress = KcsCompress,`
`  KcsLocationEnum aLocationEnum =`
`  KcsLocationUnspecified,`
`  long aOffset = KcsNoOffsetSpecified);`
	Writes a chunk set.

Example

The following code shows you some examples of how to use these `KcsChunkSet` class member functions: `getChunkSet( )`, `getChunkSize( )`, `readChunk( )`, and `writeChunk( )`.

CODE EXAMPLE 3–1 Getting, Reading and Writing a Chunk

```
KcsChunkId    myChunkId;
 KcsStatus    aStat;
 KcsChunkSet*    chunkSet;
 char*    mftData;
 u_long    mftSize;
 u_long    mftWanted;

 // Get the chunk data
 chunkSet = getChunkSet();
 if (chunkSet == NULL)
  return (KCS_INTERNAL_CLASS_CORRUPTED);

 //Get the size
 mftSize = chunkSet->getChunkSize(&aStat, myChunkId);
 if (aStat != KCS_SUCCESS)
  return(aStat);

 //Make space for the data
 mftData = NULL;
 if ((mftData = malloc((u_int)mftSize)) == NULL) {
  return (KCS_MEM_ALLOC_ERR);
 }

 //Read it
 aStat = chunkSet->readChunk(myChunkId, mftData, &mftWanted,
     mftSize);
 if (aStat != KCS_SUCCESS)
  return (aStat);

 //Zero it out and write it back
 memset(mftData, 0, mftSize);
 aStat = chunkSet->writeChunk(myChunkId, mftData, mftSize);
 if (aStat != KCS_SUCCESS)
  return (aStat);
```

CHAPTER 4

KcsLoadable Class

This chapter describes the KcsLoadable class. This base class provides the basic functionality required to create a dynamically loadable CMM. The classes derived from this class (KcsProfile, KcsProfileFormat, and KcsXform) provide functionality to create your own color profiles. The KcsLoadable class is derived from the KcsShareable class. See Chapter 1," for a description of the KcsShareable class.

As you read this chapter, you will find it helpful to have access to the following header files:

- kcsload.h (the KcsLoadable class header file)
- kcsshare.h and kcsswap.h
- kcschunk.h and kcsuidmp.h

Note - It is highly recommended that you do *not* use any of the variables and functions for handle-based memory in these header files. Handle-based memory is not required on the Solaris system.

The constant and #define identifiers for this class are defined in the kcsload.h header file as:

```
const KcsId KcsRlocLoadId = {(0x4C6f6164UL)}; /* 'Load' */
#define KcsRlocLoadIdd (0x4C6f6164UL) /* 'Load' */
```

In addition to the KcsShareable methods overridden by this class, there are methods for dynamically loading your CMM. The protected and public members are described.

Public Members

The KcsLoadable class provides the following public members.

TABLE 4–1 KcsLoadable Public Members

Public Member	Description
`typedef KcsStatus (*KCS_FUNC_INIT_PTR) (long, long, long *, long *);`	
`virtual KcsShareable * attach(long aHowMany = 1, KcsAttachType aAttachFlag = KcsAttMem, KcsStatus *aStatus = NULL);`	Calls `changePermanentUseCount( )`. If `aAttachflag` = `KcsMemFile`, also calls `KcsShareable::attach( )`. Returns pointer to the object.
`virtual long changePermanentUseCount(KcsStatus *aStatus, long aDelta);`	Increments or decrements permanent use count of a particular chunk set/chunk Id entry in the unique identifier (UID) map table. Calls `KcsChunkSet::updateChunkUseCount( )` to update the chunk map. Returns new use count.
`static KcsStatus deleteChunkSetsUIDMapEntries(KcsChunkSet *aCS);`	Deletes all the entries in the UID map table associated with a particular chunk set.
`virtual void dettach(long aHowMany = 1, KcsAttachType aDettachFlag = KcsAttMem, KcsStatus *aStatus = NULL);`	If a chunk Id is illegal, searches the UID map table for the chunk Id. Changes permanent use count. If permanent use count == 0, deletes chunk from file. Calls `KcsShareable::dettach( )`.
`virtual KcsChunkId getChunkId(KcsStatus *aStatus, KcsChunkSet *aCS);`	Looks in the UID map table and returns the chunk Id associated with this object and `aCS`. If no entry is found in the UID map table, it returns `KCS_OBJMAP_ENTRY_NOT_FOUND` and `KcsIllegalChunkId`.
`virtual KcsChunkId getChunkId();`	Very useful function; returns the chunk Id portion of the UID associated with this object.
`virtual KcsChunkSet *getChunkSet();`	Very useful function; returns the chunk set portion of the UID associated with this object.
`static KcsLoadable * getObjFromUIDMap(KcsStatus *aStatus, KcsChunkSet *aCS, KcsChunkId aChunkId);`	Checks if object identified by this chunk set and chunk Id is instantiated. If object is in UID map table, it returns a pointer to the object; else it returns `NULL`.

TABLE 4–1 KcsLoadable Public Members *(continued)*

Public Member	Description
`virtual KcsStatus isLoadable();`	Returns `KcsSuccess` if the loadable object can load and regenerate itself.
`virtual long isLoaded();`	Returns non-zero if the state of the loadable object is loaded.
`KcsLoadable(KcsStatus *);`	Constructor.
`KcsLoadable(KcsChunkSet *, KcsChunkId, KcsStatus *);`	Constructor that instantiates a loadable based on a UID that is a combination of `KcsChunkSet` and `KcsChunkId`.
`virtual ~KcsLoadable();`	Destructor. Deallocates all resources associated with specified instance.
`virtual KcsStatus load(const KcsLoadHints aLoadHints = KcsLoadAllNow, KcsCallbackFunction aCallback = NULL);`	Regenerates all necessary state from the static store associated with the `aLoadHints`.
`KcsStatus putObjIntoUIDMap(KcsChunkSet *aCS, KcsChunkId aChunkId);`	Puts the object identified by this chunk set and chunk Id in the UID map table. Returns a pointer to the object.
`virtual KcsStatus save() {return(KCS_SUCCESS);};`	Saves all object state information to its static store.
`KcsStatus save(KcsChunkSet *, KcsChunkId);`	Saves all object state information to the supplied static store.
`KcsStatus setChunkId(KcsChunkId);`	Sets the chunk set portion of the object's UID. Changing the value of ChunkId changes the position of this objects regeneration data within the object's static store.
`KcsStatus setChunkSet(KcsChunkSet *);`	Sets the chunk set portion of the object's UID. Changing the value of the chunk set changes the object's static store.

TABLE 4–1 KcsLoadable Public Members *(continued)*

Public Member	Description
`virtual KcsStatus setUID(KcsChunkSet *aNewChunkSet);`	Tries to get a chunk Id corresponding to itself and a chunk set from the UID map table. If no entry, it calls `createChunkId( )` to get a new chunk Id, does a permanent attach, and then calls `setChunkSet( )` and `setChunkId( )`.
`virtual KcsStatus unLoad(KcsLoadHints aloadHints = (KcsPurgeMemoryNow));`	Minimizes the memory requirements of the object by releasing all unnecessary state reclaimed from the static store.

CHAPTER 5

KcsProfile Class

This chapter describes the primary KcsProfile class. This base class provides basic functionality for using and creating color profiles. It was used to create the KCMS framework API .

As you read this chapter, you will find it helpful to have access to the following header files:

- kcsprofi.h
- kcspfmt.h and kcsxform.h
- kcsattr.h, kcsio.h, and icc.h

The header file for the class is kcsprofi.h. The constant and #define identifiers for this class are defined in the kcsids.h header file as:

```
const KcsId KcsLoadProfId = {(0x50726F66UL)}; /* 'Prof' */
#define KcsLoadProfIdd (0x50726F66UL) /* 'Prof' */
```

The protected and public members are described, as well as the member function override rules when deriving from this class.

Protected Members

The KcsProfile class provides the following protected members.

TABLE 5–1 KcsProfile Protected Members

Protected Member	Description
`KcsProfileFormat *iFormat;`	Pointer to a `KcsProfileFormat` instance.
`KcsOperationType *iOpAndCont;`	Indicates which operations and content types are supported by the profile. The profile can support the logical `OR` of any combination of these flags.
`virtual KcsStatus createEmptyProfile();`	Fill in the minimum amount of data such that when saved and reloaded, no `KcsXform` instances, and only the minimal tags are needed to load it.
`void initDataMembers(void);`	Sets all member data to a known state. Allows each constructor to initialize all data members to the empty or initial state.
`static long isColorSenseCMM(KcsId);`	Do not use this method in new CMMs. It is documented for historic reasons (for existing CMM only).
`KcsProfile(KcsStatus *aStat, KcsIO *aIO);`	Constructs a `KcsProfile` based on the static store defined by `aIO`.
`KcsProfile(KcsStatus *aStat, KcsId aCmmId, KcsVersion aCmmVersion, KcsId aProfId, KcsVersion aProfVersion);`	Constructs a `KcsProfile` of the type `aCmmId` with the version `aCmmVersion` using a profile format determined by `aProfVersion`.
`KcsStatus setTimeAttribute( KcsAttributeName aAttrName);`	Sets attribute specified by `aAttrName` to current time.
`virtual KcsStatus updateMonitorXforms( KcsCharacterizationData *aChar, KcsCalibrationData *aCal, void *CMMSpecificData, KcsCallbackFunction aCallback);`	Updates the monitor transformations based on `aChar` and `aCal` using the CMM's techniques. Returns either `KCS_CALIBRATION_UNSUPPORTED` or `KCS_CHARACTERIZATION_UNSUPPORTED`, unless overridden. Some derivatives require specific data in `CMMSpecificData`. All CMMs do a generic update if `CMMSpecificData` is set to `NULL`. Some CMMs callback into `aCallback` if set to non `NULL`.

TABLE 5–1 KcsProfile Protected Members *(continued)*

Protected Member	Description
virtual KcsStatus updatePrinterXforms(KcsCharacterizationData *aChar, KcsCalibrationData *aCal, void *CMMSpecificData, KcsCallbackFunction aCallback);	Updates the printer transformations based on aChar and aCal using the CMM's techniques. Returns either KCS_CALIBRATION_UNSUPPORTED or KCS_CHARACTERIZATION_UNSUPPORTED, unless overridden. Some derivatives require specific data in CMMSpecificData. All CMMs do a generic update if CMMSpecificData is set to NULL. Some CMMs callback into aCallback if set to non NULL. Note that this function currently is not supported in the KCMS default CMM.
virtual KcsStatus updateScannerXforms(KcsCharacterizationData *aChar, KcsCalibrationData *aCal, void *CMMSpecificData, KcsCallbackFunction aCallback);	Updates the scanner transformations based on aChar and aCal using the CMM's technique. Returns either KCS_CALIBRATION_UNSUPPORTED or KCS_CHARACTERIZATION_UNSUPPORTED, unless overridden. Most scanners need both aCal and aChar set to generate valid transforms. Some derivatives require specific data in CMMSpecificData . All CMMs update if CMMSpecificData is set to NULL. Some CMMs callback into aCallback if set to non NULL.

Public Members

The KcsProfile class provides the following public members.

TABLE 5–2 KcsProfile Public Members

Public Member	Description
`virtual KcsStatus connect(const long count, KcsProfile **sequence, const KcsOperationType opAndHints, KcsProfile **result, long *failingProfileIndex);`	Connects the list of KcsProfile's pointers passed in by the caller via the sequence parameter. Returns the new KcsProfile object result based on that list. failingProfileIndex indicates the number of the profile in the input list that failed. If element 0 in the array caused the connection to fail, 1 is returned.
`static KcsProfile * createProfile(KcsStatus *aStat, aIO *aIO);`	Constructs the correct runtime-loadable or internal KcsProfile based on the static store defined by aIO.
`static KcsProfile * createProfile(KcsStatus *Stat, KcsId aCmmId = KcsProfKCMSId, KcsVersion aCmmVersion = KcsProfKCMSVersionId, KcsId aProfId = Kcs2Id(icMagicNumber), KcsVersion aProfVersion = Kcs2Id(icVersionNumber);`	Constructs the correct runtime-loadable or internal KcsProfile of aCmmId type with the aCmmVersion version using a aProfVersion profile format.
`virtual KcsStatus evaluate(const KcsOperationType opAndHints, KcsPixelLayout source, KcsPixelLayout dest, KcsCallbackFunction progress);`	Transforms the caller's data. Use opAndHints to indicate which transform and content type your 'CMM is providing.
`virtual KcsStatus getAttribute(KcsAttributeName aName, KcsAttributeValue *aValue);`	Sets aValue to the attribute's aName value.
`virtual KcsStatus getAttribute(KcsAttributeName aName, void *data);`	Sets aData to the attribute's aName value.
`virtual KcsProfileFormat * getFormat();`	Gets the profile format.
`virtual KcsOperationType getOpandCont() {return(iOpAndCont);};`	Returns the supported operations and content hints.
`virtual KcsXform * getXform(KcsStatus *status, const KcsXformType XfType);`	Returns the transform of the XfType type.
`virtual KcsStatus isLoadable();`	Returns KCS_SUCCESS if meets all loadability requirements.
`KcsProfile(KcsStatus *aStat);`	Constructor.

TABLE 5–2 KcsProfile Public Members *(continued)*

Public Member	Description
`virtual ~KcsProfile(void);`	Destructor. Frees up accummulated memory by calling all member object's destructors (with `delete( )`).
`virtual KcsStatus optimize(const KcsOptimizationType type, KcsCallbackFunction progress);`	Makes profile as fast and small as possible. Specify optimization for speed or accuracy with the `type` parameter.
`virtual KcsStatus propagateAttributes2Xforms();`	Propagates all the expected attributes from the profile attribute set to the attribute sets of the transforms.
`KcsStatus save(KcsIO *);`	Saves to the static store indicated by `aIO`.
`virtual KcsStatus setAttribute(KcsAttributeName aName, KcsAttributeType aType, const char *aValue);`	Sets the value of `aName` to `aValue`. `aValue` is interpreted as the type `aType`.
`virtual KcsStatus setAttribute(KcsAttributeName aName, const KcsAttributeValue *aValue);`	Sets the value of `aName` to `aValue`. `aValue` is interpreted as the default type of `aName`.
`virtual KcsStatus setAttribute(KcsAttributeName aName, void *data);`	Sets the value of `aName` to `data` interpreted as the default type of `aName`.
`virtual KcsStatus setOpAndCont(KcsOperationType aOpAndCont);`	Sets the support operations and content hints to `aOpAndCont`.
`virtual KcsStatus setXform(KcsXformType aXformId, KcsXform *aXform);`	Sets the `aXformId` to `aXform`. If the transform is `null`, removes attachment to that particular `KcsXform` instance, if one exists. Directly implemented by calling `KcsProfileFormat::setObject( )`.

TABLE 5–2 KcsProfile Public Members *(continued)*

Public Member	Description
`virtual KcsStatus updateXforms( KcsCharacterizationData *aChar, KcsCalibrationData *aCal, void *aCMMSpecificData = NULL, KcsCallbackFunction aCallback = NULL);`	Takes all data supplied and information contained within its `KcsAttributeSet` instance to determine which type of device to update. Passes `aChar`, `aCal`, `aCMMSpecificData` and `aCallback` to the appropriate `update()`*Device*`Xforms()` call (where *Device* is `Scanner`|`Monitor`|`Printer`). (SeeTable 5–1 for descriptions of these protected members.)
`long xformIsNOP(const KcsXformType);`	Indicates whether the `KcsXformType` acts on the data passed to it. If no xform is available, returns `true`.

Member Function Override Rules

The following table tells you which `KcsProfile` member functions you must override and can override when deriving from this class. The member functions indicated with an “X” in the Must column are required to derive successfully from this base class. Others may be used but not overridden.

TABLE 5–3 KcsProfile Member Function Override Rules

Member Function	Override Rules	
	Must	**Can**
`connect()`		X
`createEmptyProfile()`		X
`evaluate()`		X
`getAttribute()`		X

TABLE 5–3 `KcsProfile` Member Function Override Rules *(continued)*

Member Function	Override Rules	
	Must	**Can**
`initDataMember()`		X
`isColorSenseCMM()`		X
`KcsProfile()`	X	
`~KcsProfile()`		X
`load()`		X
`optimize()`		X
`propagateAttributes2Xforms()`		X
`save()`		X
`setAttribute()`		X
`setOpAndCont()`		X
`setTimeAttribute()`		X
`setXform()`		X
`unload()`		X
`updateMonitorXforms()`		X
`updatePrinterXforms()`		X
`updateScannerXforms()`		X

TABLE 5–3 KcsProfile Member Function Override Rules *(continued)*

Member Function	Override Rules	
	Must	**Can**
updateXforms( )		X
XformIsNOP( )		X

Examples

The following code sample shows you how to interface with the KcsProfile class and its derivatives using the KcsGetAttribute( ) KCMS framework API wrapper function.

CODE EXAMPLE 5–1 KcsProfile Class and KcsGetAttribute ( )

```
KcsStatusId
 KcsGetAttribute(KcsProfileId profile, KcsAttributeName name,
    KcsAttributeValue *value)
 {
  VirtualWorld vWorld(true, true);
  KcsProfile * *profiles = 0;

  if (gProfileArray == NULL)
  // no profiles have been loaded or user has not handled a load
  // error correctly
  return(KCS_BAD_PROFILE_ID);
  KcsStatusId stat;
  profiles = (KcsProfile * *)KcsLockHandle(gProfileArray);
  if (isValid(profile, profiles))
   stat = profiles[profile]->getAttribute(name, value);
  else
   stat = KCS_BAD_PROFILE_ID;
  KcsUnlockHandle(gProfileArray);
  return(stat);
 }
```

The following code sample shows you how to interface with the KcsProfile class and its derivatives using the KcsConnectProfiles( ) KCMS framework API wrapper function.

CODE EXAMPLE 5–2 KcsProfile Class and and KcsConnectProfiles( )

```
KcsStatusId
 KcsConnectProfiles(KcsProfileId *resultProfileId,
    unsigned long profileCount, KcsProfileId *profileSequence,
    KcsOperationType operationLoadSet,
    unsigned long *failedProfileIndex)
 {
  VirtualWorld vWorld(true, true);
  KcsProfile * *profiles = 0;
  KcsStatus result;
  long i;

  if (gProfileArray == NULL)
    // no profiles have been loaded or user hasn't handled a load
    // error correctly
    return(KCS_BAD_PROFILE_ID);

  result = getNewValidIndex(resultProfileId);
  if (result != KCS_SUCCESS)
    return(result);
  profiles = (KcsProfile * *)KcsLockHandle(gProfileArray);
  KcsMemoryBlock * memblk = new KcsMemoryBlock(&result,
    sizeof(KcsProfile * ), profileCount);
  if (memblk == NULL)
    return(KCS_MEM_ALLOC_ERROR);
  KcsProfile * *moreProfiles = (KcsProfile * *)memblk->lock();
  KcsProfile * madeProfile = NULL;
  long failingNum = 0;
  for (i = 0; i < profileCount; i++) (
    if (!isValid(profileSequence[i], profiles)) {
     *failedProfileIndex = i;
     *resultProfileId = BAD_PROFILE_ID;
     memblk->unlock();
     memblk->dettach();
     KcsUnlockHandle(gProfileArray);
     // also have to do a dettach here !!
     return(KCS_BAD_PROFILE_ID);
     }
    else
     moreProfiles[i] =
       (KcsProfile *)profiles[profileSequence[i]]->attach();
    }
  result = profiles[profileSequence[0]]->connect(profileCount,
    moreProfiles, operationLoadSet, madeProfile, &failingNum);
  *failedProfileIndex = failingNum;
  if (result == KCS_SUCCESS)
  profiles[*resultProfileId] = madeProfile;

  for (i = 0; i < profileCount; i++) {
    moreProfiles[i]->dettach();
    }
```

(continued)

(Continuation)

```
 memblk->unlock();
 memblk->dettach();
 KcsUnlockHandle(gProfileArray);

gNumProfilesAllocated++;
return(result);
}
```

CHAPTER 6

KcsProfileFormat Class

This chapter describes the KCMS framework KcsProfileFormat class. With the KcsProfileFormat class interface you can map a profile from a static store into the traditional objects that make up a profile. The KcsProfileFormat class is derived from the KcsLoadable class. See Chapter 4," for a description of the KcsLoadable class.

As you read this chapter, you will find it helpful to have access to the following header files:

- kcspfmt.h
- kcsload.h and kcsxform.h
- kcsattr.h

The header file for the class is kcspfmt.h. The constant and #define identifiers for this class are defined in the kcsids.h header file as:

```
const KcsId KcsLoadPfmtId = {(0x50666D74UL)}; /* 'Pfmt' */
#define KcsLoadPfmtIdd (0x50666D74UL) /* 'Pfmt' */
```

In addition to the KcsLoadable methods overridden by this class, there are methods for mapping data in static store into profiles comprised of objects. The protected and public members are described, as well as the member function override rules when deriving from this class.

Protected Members

The KcsProfileFormat class has the following protected members.

TABLE 6–1 KcsProfileFormat Protected Members

Protected Member	Description
`virtual KcsStatus deleteXform(KcsXformType aXfType);`	Deletes the transforms.
`KcsProfileFormat(KcsStatus *aStat, KcsIO *KcsIO, int Format = 0);`	Constructor that takes a KcsIO object to construct the appropriate profile format.
`KcsProfileFormat(KcsStatus *aStat, KcsId aCmmId,KcsVersion aCmmVersion, KcsId aProfId, KcsVersion aProfVersion);`	Constructor. Takes a profile version and creates a blank profile format.
`virtual KcsStatus loadObjectMap(const KcsLoadHints aHints = KcsLoadAllNow, KcsCallbackFunction aCallback = NULL);`	Loads the profile format's objects mapped to chunk Ids.
`virtual KcsStatus saveObjectMap();`	Saves profile format's objects mapped to chunk Ids.

Public Members

The KcsProfileFormat class has the following public members.

TABLE 6–2 KcsProfileFormat Public Members

Public Member	Description
`static KcsProfileFormat * createProfileFormat(KcsStatus *aStat, KcsIO *aIO);`	Creates a profile format object from a KcsIO object.
`static KcsProfileFormat * createProfileFormat(KcsStatus *aStat, KcsId aCmmId, KcsVersion aCmmVersion, KcsId aProfId, KcsVersion aProfVersion);`	Creates a profile format object from CMM and profile information.
`virtual void dirtyAttrCache();`	Caches an attribute that has changed.

TABLE 6–2 KcsProfileFormat Public Members *(continued)*

Public Member	Description
`virtual void dirtyXformCache(KcsXformType aXformId);`	Caches an Xform that has changed.
`KcsLoadHints generateLoadWhat( const KcsXformType aXfType) const;`	Returns the load hint associated with this KcsXformType.
`virtual KcsAttributeSet * generateXformAttributes(KcsStatus *aStat, KcsXformType aXfType);`	Generates attributes associated with a KcsXform.
`virtual KcsStatus getCmmId(KcsId *);`	Returns a CMM Id.
`virtual KcsStatus getObject(KcsAttributeSet **aAttr, KcsCallbackFunction aFunc = NULL);`	Gets the KcsAttributeSet of the format.
`virtual KcsStatus getObject(KcsXform **aXform, KcsXformType aXformId, KcsCallbackFunction aFunc = NULL);`	Returns the KcsXform that represents the aXformID operation.
`virtual long getSaveSize();`	Returns the number of bytes to be saved.
`static KcsId getTheCMMId(KcsStatus *aStat, KcsIO *aIO);`	Gets the CMM identifier for the KcsIO object.
`static KcsVersion getTheCMMVersion(KcsStatus *aStat, KcsIO *aIO);`	Gets the CMM version for the KcsIO object.
`static KcsId getTheProfileFormat(KcsStatus *aStat, KcsIO *aIO);`	Gets the profile format using the KcsIO object.
`static KcsVersion getTheProfileVersion(KcsStatus *aStat, KcsIO *aIO);`	Gets the profile version using the KcsIO object.
`virtual long getSaveSize();`	Returns the number of bytes needed to save the profile.
`virtual KcsStatus initEmptyFormat(KcsIdent aCMMId = KcsKodakColorSenseCMM);`	Initializes profile's static store to an initial state.
`virtual KcsStatus isSupported(KcsLoadHint aHints);`	Returns whether the object(s) associated with aHints is supported by this profile instance.
`KcsProfileFormat(KcsStatus *aStat);`	Constructor.

TABLE 6–2 KcsProfileFormat Public Members *(continued)*

Public Member	Description
`virtual ~KcsProfileFormat(void);`	Destructor.
`virtual KcsStatus postAttrCompose(KcsTags *aOwningAttrSet, const long aCount, KcsTags **aOrigSequence);`	Builds ICC profile sequence description tag.
`virtual KcsStatus save();`	Saves all objects associated with this profile format.
`virtual KcsStatus saveNew(KcsChunkSet *);`	Saves using a different chunk set .
`virtual KcsStatus setCmmId(KcsId);`	Sets the CMM Id.
`virtual KcsStatus setObject(KcsAttribute *aAttr);`	Sets `KcsAttribute` to the set of attributes to save in this profile's format.
`virtual KcsStatus setObject(KcsXformType aXformId, KcsXform *aXform);`	Sets the `KcsXformType` to the `KcsXform` class `aXform`.
`virtual KcsStatus unloadWhenMatch(KcsLoadHints aMatchHints = KcsUnloadAfterUse);`	Unloads the profile objects associated with `aMatchHints`, if they match the last hints supplied to `unload( )`. Unloads the correct data automatically. Helps recover memory (call it with `KcsUnloadWhenNecessary( )`).

Member Function Override Rules

The following table tells you which `KcsProfileFormat` member functions you must override and can override when deriving from this class. The member functions indicated with an "X" in the Must column are required to successfully derive from this base class. Others can be used but not overridden.

TABLE 6–3 KcsProfileFormat Member Function Override Rules

Member Function	Override Rules	
	Must	**Can**
deleteXform( )		X
dirtyAttrCache( )		X
dirtyXformCache( )		X
generateLoadWhat( )		X
generateXformAttributes( )		X
getObject( )		X
getSaveSize( )		X
initEmptyFormat( )		X
isSupported( )		X
KcsProfileFormat( )	X	
~KcsProfileFormat( )		X
load( )		X
loadObjectMap( )		X
postAttrCompose( )		X
save( )		X
saveNew( )		X

TABLE 6–3 KcsProfileFormat Member Function Override Rules *(continued)*

Member Function	Override Rules	
	Must	**Can**
saveObjectMap( )		X
setCmmId( )		X
setObject( )		X
unload( )		X
unloadWhenMatch( )		X

External Loadable Interface

All KcsProfileFormat derivatives support runtime derivability. The base class supports the static createX(UID)( ) method like many of the other classes in the KCMS framework. It also supports createProfileFormat(KcsVersion)( ) for creation of new profile formats.

```
static KcsProfileFormat *createProfileFormat(
   KcsStatus *aStat, KcsIO *aIO);
 // Create a profile format given a chunkSet that has
 // already been created.
 static KcsProfileFormat *createProfileFormat(
   KcsStatus *aStat, KcsId aCmmId, KcsVersion aCmmVersion,
   KcsId aProfId, KcsVersion aProfVersion);
 // Create a new profile format of a given version.
```

See the *KCMS CMM Developer's Guide* for more information on creating a KcsProfileFormat runtime derivative.

CHAPTER 7

KcsTags Class

This chapter describes the KcsTags class of the KCMS framework. This class provides numerous data members used throughout the KCMS framework and for use in your CMM.

Note - KcsAttributeSet is an alias to the KcsTags class. This is for historical reasons only.

As you read this chapter, you will find it helpful to have access to the kcstags.h header file.

Note - It is highly recommended that you do *not* use any of the variables and functions for handle-based memory in the these header files. Handle-based memory is not required on the Solaris system.

The #define identifiers for this class are defined in the kcstags.h header file as:

```
#define KcsAttributeSet KcsTags
```

In addition to the KcsLoadable methods overridden by this class, this class includes data members used throughout the KCMS framework and for use in your CMM. There are no protected members. Public members are described in detail.

Public Members

The KcsTags class provides the following public members.

TABLE 7–1 KcsTags Public Members

Public Member	Description
`KcsAttributeSet *copy(KcsStatus *status);`	A non-shared copy of the object.
`static KcsTags * createAttributeSet(KcsStatus *status);`	Creates a blank attribute set.
`static KcsTags * createAttributeSet(KcsChunkSet *ChunkSet, KcsStatus *status);`	Creates an attribute set from a chunk set.
`static KcsTags * createAttributeSet(KcsStatus *aStat, KcsChunkSet *aCS, KcsChunkId aChunkId);`	Creates an attributes set from a chunk set and chunk Id.
`KcsStatus getAttribute(KcsAttributeName tag, KcsAttributeValue *value, KcsAliasUsage aAliasUse = KcsIndirectThroughAlias);`	Given a valid `attribute`, loads the `value`.
`static KcsStatus getAttributeInfo(KcsAttributeName aTag, KcsAttributeType *aType, unsigned long *count, unsigned long *sizeOfType, KcsTags *tags = NULL);`	Given a valid `aTag`, populates the supplied parameters with the type and the count (the number of tokens that make up the attribute) of the attribute associated with the supplied `aTag`.
`static unsigned long getAttrSize(KcsAttributeType aTagType);`	Returns the size in bytes of the attribute type.
`static unsigned long getAttrSize(KcsAttributeValue *aVal);`	Returns the attribute size.
`static long getICProfSeqDescInfo( icProfileSequenceDesc *aDesc, icDescStruct **aDescPtrs, long nPtrs);`	Returns information about an ICC profile sequence description.

TABLE 7–1 KcsTags Public Members *(continued)*

Public Member	Description

`static long getICTextDescInfo(`
 	
`icTextDescription *aDesc,`
`long *aASCIILength = NULL,`
`icUInt8Number **aASCIIPtr = NULL,`
`icInt32Number *aUniCodeCode = NULL,`
`long *aUniCodeLength = NULL,`
`icInt8Number **aUniCodePtr = NULL,`
`icInt16Number *aScriptCodeCode =`
`NULL,`
`long *aScriptCodeLength = NULL,`
`icInt8Number **aScriptCodePtr =`
`NULL);`
	Given an ICC text description, returns the parameters contained in it.

`static long`
`KcsTags::getICTextDescSize(`
`long aASCIILength,`
`long aUniCodeLength,`
`long aScriptCodeLength);`
	Returns the size of an ICC text description attribute.

`static long`
`getICUcrBgCounts(icUcrBg *aUcrBg,`
`icUInt32Number *aUcrCount = NULL,`
`icUInt32Number *aBgCount = NULL);`
	Returns the number of Ucrdatasets in an ICC Ucr Bg curve.

`unsigned long getLargestAttrValSize();`	Returns the size of the largest attribute.

`KcsStatus`
`getTag(long ordinal,`
`KcsAttributeName *aTag);`
	Sets the parameter `aName` to position `ordinal` in the internal attribute array pointed to by the member handle `tagHandle`.

`virtual Boolean` `isReadOnly(KcsAttributeName name);`	Returns `True` if an attribute is read only. Read-only attributes are: `icSigNumTag`, `icSigListTag`, `KcsAttrNumber`, `KcsAttrAttributesSet`, and `KcsPixelLayoutSupported`.

`KcsStatus`
`KcsAttrCompose(const long count,`
`KcsTags **sequence);`
	Given a count and a sequence of `KcsAttributeSet` objects, combines the attributes according to the rules in the current `KcsAttributeSet` object and populates its attributes accordingly. See the ICC profile format specification for the profile sequence attribute.

`typedef enum {`
`/* ... */`
`} KcsAttribute;`
	ICC attributes.

TABLE 7–1 KcsTags Public Members *(continued)*

Public Member	Description
`KcsStatus KcsClassify(const long count, KcsTags **sequence, KcsAttributeName tag);`	Attribute composition rule. Classifies the attributes.
`KcsStatus KcsCommon(const long count, KcsTags **sequence, KcsAttributeName tag);`	Attribute composition rule. Returns status on whether there are common attributes in two profiles.
`KcsStatus KcsConcatenate(const long count, KcsTags **sequence, KcsAttributeName tag);`	Attribute composition rule. Concatenates attributes.
`KcsStatus KcsNever(const long count, KcsTags **sequence, KcsAttributeName tag);`	Attribute composition rule. Attribute composition never propagates.
`KcsTags(KcsChunkSet *ChunkSet, KcsChunkId chunkId, KcsStatus *status));`	Allows you to create an attribute's object from a chunk. A chunk constructor.
`KcsTags(char *buffer, unsigned long sizeOfBuf, KcsStatus *status);`	Allows a user to create an attribute's object from data found in the character buffer supplied as an input parameter. A character buffer constructor.
`KcsTags(KcsStatus *status);`	Constructor that allows a user to create an empty attributes object.
`~KcsTags();`	Destructor.
`KcsStatus KcsUseCSInRight(const long count, KcsTags **sequence, KcsAttributeName tag);`	Composition rule.
`KcsStatus KcsUseLeft(const long count, KcsTags **sequence, KcsAttributeName tag);`	Composition rule.
`KcsStatus KcsUseRight(const long count, KcsTags **sequence, KcsAttributeName tag);`	Composition rule.

TABLE 7–1 KcsTags Public Members *(continued)*

Public Member	Description
`long returnCurrentNumberOfAttributes(void);`	Returns the current number of attributes stored in the attribute array.
`KcsStatus save();`	Saves the attributes.
`KcsStatus saveTags();`	Saves the attributes.
`KcsStatus setAttribute(KcsAttributeName aName, void *data);`	Stores an association of `name`-to-attribute in the attribute array.
`KcsStatus setAttribute(KcsAttributeName tag, KcsAttributeType aType, void *data);`	Stores an association of `tag`-to-attribute in the attribute array.
`KcsStatus setAttribute(KcsAttributeName tag, KcsAttributeType aType, const char *value);`	Stores an association of `tag`-to-attribute in the attribute array. The information is supplied in a character string.
`KcsStatus setAttribute(KcsAttributeName tag, const KcsAttributeValue *value, KcsAliasUsage aAliasUse = KcsIndirectThroughAlias);`	Stores an association of `tag`-to-attribute in the attribute array. If you use the `KcsAttributeValue` structure as a parameter and pass in `NULL` as its `value`, the attribute will be deleted if it is stored; otherwise an error is returned.
`static long KcsTags::setICTextDesc(icTextDescription *aDesc, long aSize, icUInt32Number aASCIILength = 1, icUInt8Number *aASCIIPtr = NULL, icUInt32Number aUniCodeCode = 0, icUInt32Number aUniCodeLength = 1, icUInt8Number *aUniCodePtr = NULL, icUInt16Number aScriptCodeCode = 0, icUInt8Number aScriptCodeLength = 1, icUInt8Number *aScriptCodePtr = NULL);`	Puts the various arguments into an ICC text description structure.

CHAPTER 8

KcsXform Class

This chapter describes the KCMS framework KcsXform class. This base class provides an interface for component transformations of different transformation types; for example, matrix and grid-table based.

As you read this chapter, you will find it helpful to have access to the following header files:

- kcsxform.h
- kcschunk.h and kcsload.h
- kcsattr.h and icc.h

The header file for the class is kcsxform.h. The constant and #define identifiers for this class are defined in the kcsxform.h header file as:

```
const KcsId KcsLoadXfrmId = {(0x5866726dUL)}; /* 'Xfrm' */
#define KcsLoadXfrmIdd (0x5866726dUL) /* 'Xfrm' */
```

In addition to the KcsLoadable methods overridden by this class, this class includes methods for component transformations of different transformation types. The protected and public members are described, as well as the member function override rules when deriving from this class.

Enumerations

The KcsXform class has the following enumerations.

TABLE 8–1 KcsXform Enumerations

Enumeration	Description
`enum KcsInOut {In, Out};`	Enumeration used in `getNumComponents( )`, `getComponentDepth( )`, `setNumComponents( )`, and `setComponentDepth( )`.

Protected Members

The KcsXform class has the following protected members.

TABLE 8–2 KcsXform Enumerations

Enumeration	Description
`long myChunkId;`	Current chunk Id.
`KcsOperationType myOpsAndHints;`	Current operations and hints.
`KcsTransformKind myKindOf;`	Current transform.
`int callbackInterval;`	Rows per callback.

Public Members

The `KcsXform` class has the following public members.

TABLE 8–3 KcsXform Public Members

Public Member	Description
`int areLayoutsCloseEnough(KcsPixelLayout *in1, KcsPixelLayout *in2, KcsPixelLayout *out);`	Returns nonzero if the *n*th pixel in `in1` and `in2` uses the pixel in `out` without corrupting the (n+1)th pixel of `out`, (for example, `planar<->chunky` fails while `chunky<->chunky` does not); else returns zero.
`int areLayoutsEqual(KcsPixelLayout *l1, KcsPixelLayout *l2);`	Returns non zero if the pixel layout structs in `l1` and `l2` are identical (especially the pointers to the data buffers); else returns zero.
`virtual KcsStatus compose(KcsXformSeq *xformSeq, KcsXform **newXform, KcsCallbackFunction progress);`	Generates a new Xform from `xformSeq` supplied. Connects `numXforms` transforms supplied in a `KcsXformSeq` and returns a `KcsXform` in `newXform` that has the same effect as the list of Xforms in the order supplied. An error in connecting from techs[*n*] to techs[*n*+1] is reflected in errors[*n*]. A `NULL KcsXform` is returned on error.
`virtual KcsStatus connect(const KcsOperationType opsAndHints, const long numXforms, KcsXform **technologies, KcsXform **newXform, KcsCallbackFunction progress, long *numberThatFailed);`	Connects `numXforms` transforms supplied in `technologies` and returns a `KcsXform` that has the same effect as the list of Xforms in the order supplied. If an error is found, it is reported. An error in connecting from techs[*n*] to techs[*n*+1] is reflected in errors[*n*]. A `NULL KcsXform` is returned on error.
`virtual KcsStatus connectSink(KcsPixelLayout *sinkLayout);`	Associates `sinkLayout` as the default layout buffer for output from this transform.
`virtual KcsStatus connectSource(KcsPixelLayout *sourceLayout);`	Associates `sourceLayout` as the default layout buffer for input to this transform.
`virtual KcsXform *convertXform(KcsStatus *aStat, KcsId aXformType);`	Converts the instance into a derivative of type `aXformType` and returns a pointer to the new transform. Does automatic xform conversion when needed.
`static KcsXform * createXform(KcsStatus *status, KcsChunkSet *aChunkSet, KcsChunkId chunkId, KcsAttributeSet *aAttrSet=NULL);`	Creates a `KcsXform` from a chunk set.
`KcsStatus eval();`	Evaluates the data described in the local `inBuffer` set by `connectSource( )`through this transform into the buffer described by the `outBuffer` set by `connectSink( )`. The operation is from `myOptType`. This provides no progress.

TABLE 8–3 KcsXform Public Members *(continued)*

Public Member	Description
`KcsStatus eval(const KcsOperationType opsAndHints, long *in32, long *out32);`	Evaluates the data described in one 32-bit integer.
`KcsStatus eval(const KcsOperationType opsAndHints, KcsPixelLayout *inBuffer, KcsPixelLayout *outBuffer, KcsCallbackFunction progress) = 0;`	Evaluates the data described in `inBuffer` through the Xform described by this instance into the buffer described by `outBuffer`.
`KcsStatus eval(const KcsOperationType opsAndHints, const float **inComp, const float **outComp, const long *inStride, const long *outStride, const long num, KcsCallbackFunction progress);`	Evaluates the data described in the float-based buffer supplied through this transform into the float-based buffer supplied.
`KcsStatus eval(const KcsOperationType opsAndHints, const unsigned char **inComp, const unsigned char **outComp, const long *inStride, const long *outStride, const long num, KcsCallbackFunction progress);`	Evaluates the data described in the byte-based buffer supplied through this transform into the byte-based buffer supplied.
`virtual KcsAttributeSet *getAttrSet(KcsStatus *aStat = NULL);`	Returns the set of attributes associated with this `KcsXform` instance. If `NULL`, creates an empty one.
`virtual long getComponentDepth(KcsInOut whichOne);`	Retrieves the component depth.
`virtual KcsStatus getLoadOrder(long aNumTypes, KcsLoadSaveSet aAvailableTypes, KcsLoadSaveSet *aOrderOfTypes);`	Given `aAvailableTypes`, returns the order to load the types in an array `aOrderOfTypes[aNumTypes]`.
`virtual long getNumComponents(KcsInOut whichOne);`	Retrieves the number of components.
`KcsOperationType getOpsAndHints();`	Retrieves the ops and hints.
`virtual KcsStatus getSaveOrder(long aNumTypes, KcsLoadSaveSet aAvailableTypes, KcsLoadSaveSet *aOrderOfTypes);`	Given `aAvailableTypes`, returns the order to save the types in an array `aOrderOfTypes[aNumTypes]`.

TABLE 8–3 KcsXform Public Members *(continued)*

Public Member	Description
`static KcsId getXformType(KcsChunkSet *aChunkSet, KcsChunkId chunkId, KcsStatus *stat, int *InComp, int *OutComp);`	Gets the Xform type.
`typedef long KcsLoadSaveSet;`	Used in `getLoadOrder()` and `getSaveOrder()`.
`typedef unsigned long KcsTransformKind;`	Used in `kindOfTransform()`.
`KcsXform(KcsStatus *status, KcsAttributeSet *aAttrSet=NULL);`	Constructor.
`KcsXform(KcsStatus *status, KcsOperationType opsAndHints, KcsChunkSet *aChunkSet, KcsChunkId chunkId, KcsAttributeSet *aAttrSet=NULL);`	Constructor.
`virtual ~KcsXform();`	Destructor.
`virtual KcsTransformKind kindOfTransform();`	Retrieves the kind of transform.
`virtual KcsStatus loadU(KcsMemoryBlock *aXform, KcsCallbackFunction aCallback = NULL);`	Loads the Xform from the universal buffer supplied.
`virtual int numberOfCallbacks(KcsPixelLayout *aIn, KcsPixelLayout *aOut);`	If the Xform evaluated data with the supplied pixel layouts and a callback was supplied, tells how many times the callback would be called. The default is once for every four scan lines of `aIn`.
`KcsStatus optimize();`	Optimizes the Xform.
`virtual KcsStatus optimize(const KcsOperationType opsAndHints, const KcsOptimizationType optimization, KcsCallbackFunction progress);`	Optimizes the Xform.
`virtual KcsStatus saveU(KcsMemoryBlock *aXform, KcsCallbackFunction aCallback=NULL);`	Saves the Xform into the universal buffer supplied.
`virtual KcsStatus setAttrSet(KcsAttributeSet *aAttrSet);`	Sets the set of attributes associated with this `KcsXform` instance.

TABLE 8–3 KcsXform Public Members *(continued)*

Public Member	Description
`virtual KcsStatus setCallbackInterval(int callbackInt);`	Sets the callback interval.
`virtual KcsStatus setComponentDepth(KcsInOut whichOne, long Depth = 8);`	Sets the component depth.
`virtual KcsStatus setDefaultAttributes(void);`	Sets up the default set of attributes for Xforms.
`virtual KcsStatus setNumComponents(KcsInOut whichOne, long numComp = 3);`	Sets the number of components.
`void setOpsAndHints(KcsOperationType);`	Sets the ops and hints.
`KcsStatus validateLayouts( KcsPixelLayout *sourceLayout, KcsPixelLayout *sinkLayout);`	Makes sure the source layout and sink layout are compatible with each other and with transform. The color space of the buffer is validated once that field is added to the `KcsPixelLayout` structure.

External Loadable Interface

Use these `KcsXform` external entry points to *load your derivatives at runtime*. See the DDK manual *KCMS CMM Developer's Guide* for more information on creating a `KcsXform` derivative as a runtime loadable CMM.

TABLE 8–4 KcsXform External Loadable Interface

Extern "C"	Description
`extern long KcsDLOpenXfrmCount;`	Holds a counter for the number of times this dynamically loadable module has been loaded.
`KcsStatus KcsInitXfrm(long libMajor, long libMinor, long *myMajor, long *myMinor);`	Returns `KCS_SUCCESS`.

TABLE 8–4 KcsXform External Loadable Interface *(continued)*

Extern “C”	Description
`KcsXfrm * KcsCreateXfrm(KcsStatus *aStat, KcsChunkSet *aChunkSet, KcsChunkId aChunkId , KcsAttributeSet *aAttrSet);`	Returns a `KcsXform` object by invoking the `KcsXform` constructor.
`KcsStatus KcsCleanupXfrm();`	Returns `KCS_SUCCESS`.

Member Function Override Rules

The following table tells you which `KcsXform` member functions you must override and can override when deriving from this class. The member functions indicated with an “X” in the Must column are required to derive successfully from this base class. Others can be used and not overridden.

TABLE 8–5 KcsXform Member Function Override Rules

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

TABLE 8–5 KcsXform Member Function Override Rules *(continued)*

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

CHAPTER 9

KcsXformSeq Class

This chapter describes the KCMS framework KcsXformSeq class. This class provides an interface to manipulate a list or concatenation of KcsXforms in a sequence as one KcsXform instance.

As you read this chapter, you will find it helpful to have access to the following header files:

kcsxfseq.h and kcsxform.h

Note - It is highly recommended that you do *not* use any of the variables and functions for handle-based memory in the these header files. Handle-based memory is not required on the Solaris system.

The header file for the class is kcsxfseq.h. The constant and #define identifiers for this class are defined in the kcsxfseq.h header file as:

```
const KcsId KcsXfrmSeqId = {0x53657120}; /* 'Seq ' */
#define KcsXfrmSeqIdd (0x53657120) /* 'Seq ' */
```

In addition to the KcsLoadable and KcsXform methods overridden by this class, there are methods to manipulate a list or concatenation of KcsXforms. The protected and public members are described.

Protected Members

The KcsXformSeq class has the following protected members.

TABLE 9–1 KcsXformSeq Protected Members

Protected Member	Description
`virtual KcsStatus evalPrep(unsigned long *seqCount, KcsPixelLayout *inBuffer, KcsPixelLayout *outBuffer, KcsPixelLayout *genIn, KcsPixelLayout *genOut, KcsPixelLayout *remIn, KcsPixelLayout *remOut);`	Prepares the pixel layouts for evaluation.

Public Members

The KcsXformSeq class has the following public members.

TABLE 9–2 KcsXformSeq Public Members

Public Member	Description
`virtual KcsStatus addAsParent(KcsXformSeq *parent);`	Adds a parent to my list of parents.
`virtual KcsStatus delAsParent(KcsXformSeq *parent);`	Deletes a parent from my list of parents. Decrements the parent use count and unshares the parent.
`virtual KcsStatus deOptimize();`	Deoptimizes this sequence. If the original data is not available, returns KCS_CANNOT_DEOPTIMIZE.
`virtual KcsStatus evalSegment(const KcsOperationType opsAndHints, KcsPixelLayout *inBuffer, KcsPixelLayout *outBUffer, KcsCallbackFunction progress);`	Evaluates the inBuffer data through the sequence of Xforms described by this instance into outBuffer.
`virtual KcsStatus getLeftmostXform(long *n, KcsXformSeq **pnextXform, KcsXform **nextXform);`	Gets the leftmost basic Xform in the sequence. Returns the parent of the xform(pnextXform) and the number of the Xform within that sequence. Recurses to the true leftmost.

TABLE 9–2 KcsXformSeq Public Members *(continued)*

Public Member	Description
`virtual KcsStatus getNextXform(KcsXformSeq **pnextXform, KcsXform **nextXform);`	Gets the next basic Xform in the sequence. Returns the parent of the xform(`pnextXform`). Unrolls a sequence into a list of its most basic Xforms. Returns `KCS_END_OF_XFORMS` when it reaches the last Xform in the sequence.
`virtual long getOrigNumber() const {return(numOriginalXforms);};;`	Returns the number of `KcsXforms` in the sequence.
`virtual KcsStatus getRightmostXform(long *n, KcsXformSeq **pnextXform, KcsXform **nextXform);`	Gets the rightmost basic Xform in the sequence. Returns the parent of the xform(`pnextXform`) and the number of the Xform within that sequence. Recurses to the true rightmost.
`virtual KcsStatus getXform(long n, KcsXform **anXform);`	Gets the `n`th Xform in the list. Applies only to this sequence. It does not recurse through sequences of sequences.
`virtual KcsStatus insertXform(long n, KcsXform *anXform);`	Adds the Xform in the `n`th position in the list. It is not recursive.
`virtual KcsOptimizationType isOptimized();`	Indicates if the sequence is optimized.
`KcsXformSeq(KcsStatus *status, KcsAttributeSet *aAttrSet = NULL);`	Constructors from a sequence of 0 Xforms.
`KcsXformSeq(KcsStatus *status, const KcsOperationType opsAndHints, KcsChunkSet *aChunkSet, KcsChunkId chunkId, KcsAttributeSet *aAttrSet = NULL);`	Constructors from a chunk set object.
`KcsXformSeq(const KcsOperationType opsAndHints, const long numXforms, KcsXform **technologies, KcsCallbackFunction progress, long *numberThatFailed, KcsStatus *stat, KcsAttributeSet *aAttrSet = NULL);`	Constructors based on the list of Xforms supplied.
`virtual ~KcsXformSeq();`	Destructor.
`virtual KcsOptimizationType isOptimized();`	Queries whether this sequence has been optimized.

TABLE 9–2 KcsXformSeq Public Members *(continued)*

Public Member	Description
`virtual KcsStatus optimizeLineage(const KcsOperationType opsAndHints, const KcsOptimizationType optimization, KcsCallbackFunction prog);`	Optimizes this Xform and all of its lineages. Does not optimize parents since a parent use count is kept.
`virtual KcsStatus removeXform(long n);`	Deletes the nth Xform in the list. It is not recursive.
`virtual KcsStatus replaceXform(long n, KcsXform *anXform);`	Replaces the Xform in the nth position in the list with an Xform. It is not recursive.