Data Structures 3

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

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

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

Macros

The following macros are used in the C API:

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

3

Constants

The following constants are used in the C API:

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

Data Types

KcsAttributeBase

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

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

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

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

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

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

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

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

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

OR

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

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

KcsAttributeName

```
typedef long KcsAttributeName;
```

KcsAttributeName is used in several functions as the tag argument.

KcsAttributeType

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

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

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

KcsAttributeValue

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

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

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

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

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

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

Code Example 3-1 `KcsAttributeValue`

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

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

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

```
}

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

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

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

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

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

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

KcsAttrSpace

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

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

KcsCalibrationData

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

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

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

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

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

KcsCallbackFunction

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

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

Note – The profile field is currently undefined.

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

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

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

Code Example 3-2 KcsCallbackFunction

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

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

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

KcsCharacterizationData

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

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

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

KcsColorSample

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

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

KcsComponent

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

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

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

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

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

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

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

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

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

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

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

KcsCreationDesc

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

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

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

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

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

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

KcsCreationType

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

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

KcsErrDesc

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

KcsErrDesc contains useful information about an error.

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

KcsEvalSpeed

```
typedef long KcsEvalSpeed;
```

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

KcsFileId

```
typedef int KcsFileId;
```

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

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

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

KcsFunction

```
typedef unsigned long KcsFunction;
```

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

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

Table 3-1 `KcsFunction` Bit Constants

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

KcsIdent

```
typedef long KcsIdent;
```

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

3

KcsLoadHints

```
typedef unsigned long KcsLoadHints;
```

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

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

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

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

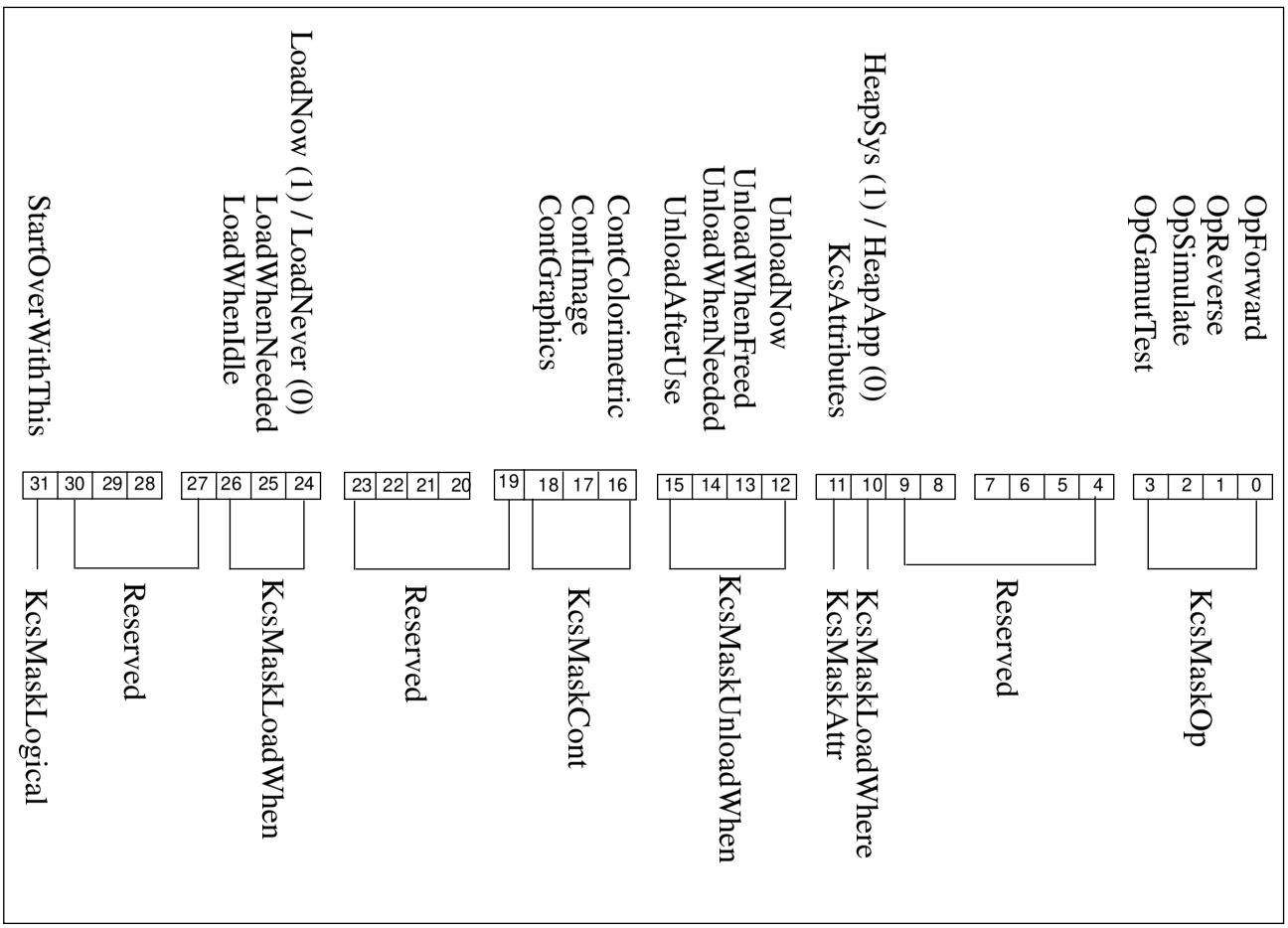

Figure 3-1 Bit Positions and Masks for Load Hints

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

Table 3-2 Bit Mask Values for Load Hints

Load Hint Bit Masks	Values	Description
KcsMaskOp	#define KcsOpForward (0x00000001) #define KcsOpReverse (0x00000002) #define KcsOpSimulate (0x00000004) #define KcsOpGamutTest (0x00000008) #define KcsOpAll (0x000003FF)	See "Operation Hint Constants" on page 43.
KcsMaskEffect	#define KcsEffect (0x00000200)	
KcsMaskLoadWhere	#define KcsHeapApp (0) #define KcsHeapSys (0x00000400)	Load it into application heap. Load it into system heap.
KcsMaskAttr	#define KcsAttributes (0x00000800)	Load attributes.
KcsMaskUnloadWhen	#define KcsUnloadNow (0x00001000) #define KcsUnloadWhenFreed (0x00002000) #define KcsUnloadWhenNeeded (0x00004000) #define KcsUnloadAfterUse (0x00008000)	Unload it now. Unload it during a call to KcsFreeProfile. Unload it when the CMM needs the memory for something else. Unload it just after the CMM needs to reference it.
KcsMaskCont	#define KcsContUnknown (0x00000000) #define KcsContGraphics (0x00010000) #define KcsContImage (0x00020000) #define KcsContColorimetric (0x00040000) #define KcsContAll (0x00FF0000)	See "Content Hint Constants" on page 44.
KcsMaskLoadWhen	#define KcsLoadNever (0x00000000) #define KcsLoadNow (0x01000000) #define KcsLoadWhenNeeded (0x02000000) #define KcsLoadWhenIdle (0x04000000)	Never load it. Load it now. Load it just before CMM needs to reference it. Load it when the system has a free moment.
KcsMaskLogical	#define KcsStartOverWithThis (0x10000000) #define KcsAddToCurrentHints (0x00000000)	Get rid of the previous Hints and start with this one. Logically add this Hints with the others already set.

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

Code Example 3-3 Load Hint Bit Mask Combinations

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

Operation Hint Constants

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

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

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

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

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

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

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

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

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

Content Hint Constants

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

The C API defines the following constants:

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

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

Note – ICC content hints are called *rendering hints*. Currently, the following rendering hints defined are:

```
icPerceptual = KcsContImage
icRelativeColorimetric = KcsContColormetric
icSaturation = KcsContGraphics
icAbsoluteColorimetric = <no equivalent>
```

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

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

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

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

KcsMeasurementBase

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

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

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

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

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

KcsMeasurementSample

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

`KcsMeasurementSample` holds a single measurement. Both the `KcsCalibrationData` and the `KcsCharacterization` data structures contain extendable arrays of `KcsMeasurementSample` structures. Each measurement has an input, an output, a measurement weight, standard

deviation and sample type. The input and output color spaces are specified by fields in the `KcsMeasurementBase` structure, which is part of both the `KcsCalibration` and `KcsCharacterization` structures.

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

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

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

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

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

KcsOperationType

```
typedef unsigned long KcsOperationType;
```

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

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

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

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

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

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

KcsOptimizationType

```
typedef unsigned long KcsOptimizationType;
```

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

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

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

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

KcsPixelLayout

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

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

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

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

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

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

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

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

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

Use the following definitions to index the component array:

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

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

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

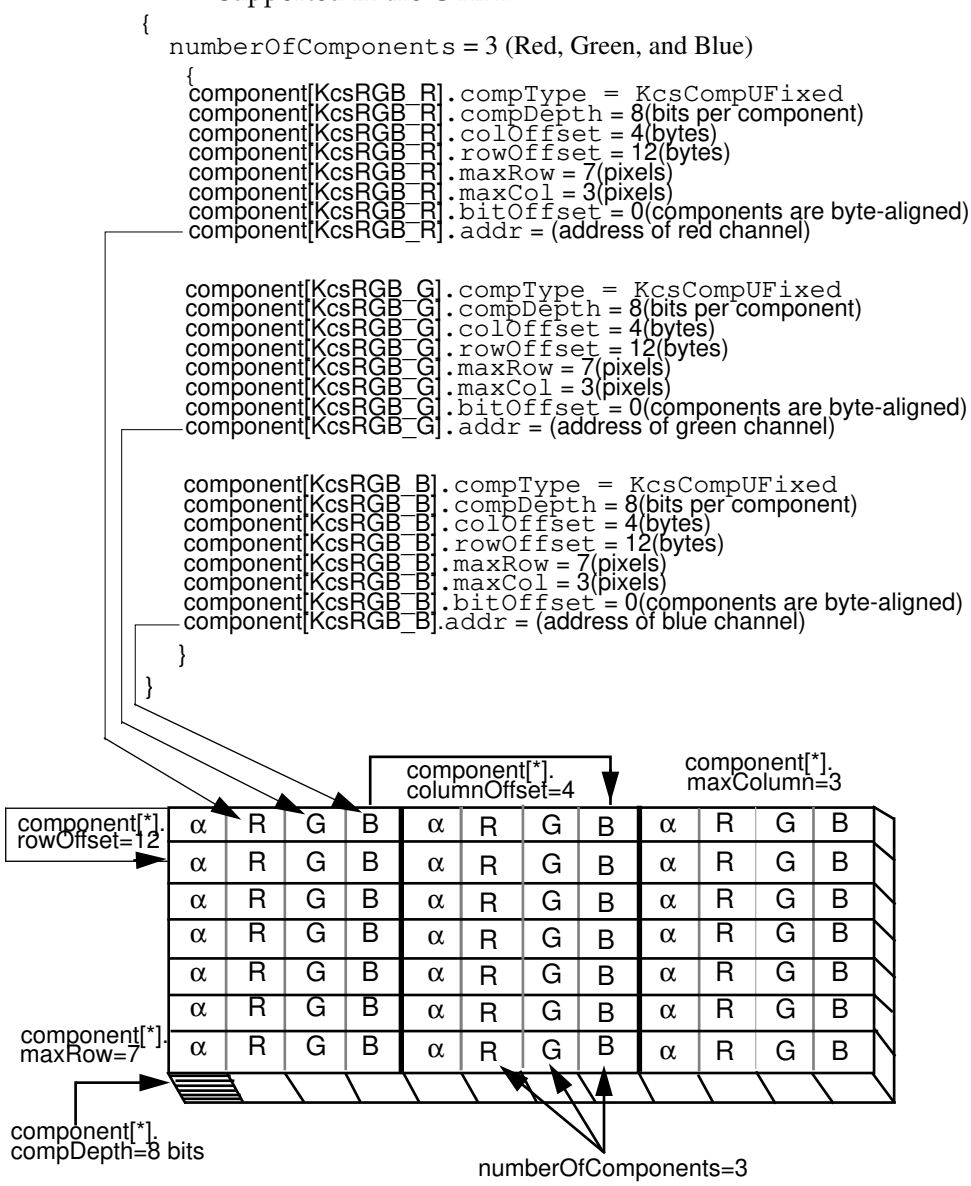

Figure 3-2 24-bit Color Component-Interleaved Data for RGB Pixel Image

KcsPixelLayoutSpeeds

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

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

3

KcsProfileDesc

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

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

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

KcsFileProfile

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

KcsMemoryProfile

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

KcsSolarisProfile

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

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

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

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

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

KcsWindowProfile

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

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

KcsProfileId

```
typedef long KcsProfileId;
```

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

KcsProfileType

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

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

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

See "KcsProfileDesc" on page 54 for details on using each type.

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

KcsSampleType

```
typedef unsigned long KcsSampleType;
```

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

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

Table 3-3 KcsSampleType Constants

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

3

KcsStatusId

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

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