Little

CMS2

Engine API 2.18

https://www.littlecms.com

Contents

Requirements

Little CMS 2 requires a C99 compliant compiler. gcc 3.2 and above, Intel compiler, clang, llvm and Borland 5.5 does support C99 standard. In addition, Microsoft® Visual C++ 2017, 2019 and 2022 are supported as well.

Dependencies

If you plan to compile the tifficc and jpgicc utilities, you need to have following libraries installed. Please refer to documentation of each library for installation instructions.

tifficc	Lib TIFF	http://www.simplesystems.org/libtiff/
jpgicc	Independent JPEG Group	http://www.ijg.org/

Installation

Linux/unices

Unpack & untar the tarball, cd to the newly created directory and type:

```
./configure
make
make check
```

This latter will run the testbed program as well. If you want to install the package, type:

```
sudo make install
```

This does copy API include files into /usr/local/include and libraries into /usr/local/lib. You can change the installation directory by using the –prefix option

There are additional targets on the Makefile:

install: Iinstalls packages (needs sudo)

check: Builds and executes testbed program

clean: Deletes object & binary files

distclean: Deletes any file not present in the distribution package

dist: Creates the distribution files

uninstall: removes pakage from system (needs sudo)

In the case you have autotools installed (autoconf, automake and so) and would like to regenerate the configure script, you can use the provided script:

```
./autogen.sh
```

To erase all files not needed for “configure” generation, type

```
./autogen.sh --distclean
```

Note that doing so will left the project without a way to be configured. You need to run the autogen.sh script again to generate the needed files.

There are optional plug-ins that comes in the standard distribution. Since the license for those plug-ins is not same as the core library, the confgure script does NOT enable plug-ins by default. You can add the plug-ins into your libraries by typing adequate toggles to configure script. i.e.:

```
./configure –with-fastfloat
./configure –with-threaded
```

MESON Build

Since 2.15 LittleCMS and plug-ins can also be built by Meson.
https://en.wikipedia.org/wiki/Meson_(software)

There are several ways to compile LittleCMS with meson, the most usual one is:

```
meson setup build/
meson test -C build
sudo meson install -C build
```

plug-ins and samples compilation are selectable. Use meson configure to see the exact settings.

Note: meson build for Cygwin and MinGW/MSYS2 are NOT supported. Use auto-tools or Windows Subsystem for Linux (WSL) instead.

Windows® MS Visual Studio

There are projects for most popular environments in the 'Projects' folder. Just locate which one you want to use.

Windows® Borland C++ 5.5

BC 5.5 is partially supported. It compiles Little CMS as a DLL with some limitations. There is a BorlandC_5.5 folder in Projects that contains the necessary scripts. Run mklcmsdll.bat to get the DLL compiled.

Apple® Mac

There is an X-Code project in the ‘Projects’ folder. In addition, you can use the procedure described in Linux/unices section. Using this latter, Little CMS has been tested to work in Catalina x64 and Sequoia on Apple silicon M1 and M3 arm64.

Other

For Solaris and other, you could try the procedure described Linux/unices section. Autotools scripts does work in a multitude of different environments. If this doesn’t work, any C99 compliant compiler should be able to deal with the code. I have checked on embedded Linux kernels like STM32, Cortex-M, Cortex-A and it works ok with gcc. Please let me know if you experiment issues when porting the code.

Note: Make sure to instruct your compiler to use C99 convention. In gcc, you can add:

```
-std=gnu99
```

Without that, ULLONG_MAX wouldn’t be defined in some situations.

Configuration toggles

lcms2.h is coded in a way that tries to automatically detect the better configuration for the current compiler. However, in some situations (unchecked compilers/environments) it may need some "manual override". To do so, comment/uncomment following symbols in ***lcms2.h***, the test bed program may hint to manually change some of those flags.

CMS_DLL	*Define this if you are **using** this package as a DLL (windows only)*
CMS_DLL_BUILD	*Define this if you are **compiling** this package as a DLL (windows only)*
CMS_USE_BIG_ ENDIAN	*Uncomment this symbol if you are using non-supported big endian machines and the test bed hints to do so.*
CMS_DONT_USE_INT64	*Uncomment this symbol if your compiler/machine does NOT support the "long long" type. This is automatically detected on most cases*
CMS_DONT_USE_FAST_FLOOR	*Uncomment this if your compiler doesn't work with fast floor function. The test bed will hint to do so if necessary.*
CMS_USE_PROFILE_BLACK_POINT_TAG	*Uncomment this line if you want lcms2 to use the black point tag in profile, if commented, lcms2 will compute the black point by its own. Important note: It is safer to leave it commented out, as black point detection feature will work even for missing or wrong black point tags.*
CMS_BASIC_TYPES_ALREADY_DEFINED	*Define this one if you want to define the basic types elsewhere, and want **lcms2.h** to reuse those types.*
CMS_STRICT_CGATS	*Define this one if you want strict CGATS.13 parsing. By default, Little CMS is tolerant to some issues, like missing "KEYWORD" definitions. If you want errors raised on such situations, define this symbol.*
CMS_NO_PTHREADS	*Uncomment to get rid of pthreads/windows dependency. Without pthreads only cmsDoTransform is reentrant.*

CMS_RELY_ON_WINDOWS_STATIC_MUTEX_INIT	*For pre Windows XP compatibility. See lcms2_internal.h*
CMS_NO_REGISTER_KEYWORD	*Uncomment this to remove the "register" storage class* ***NOTE**: Some compilers exhibit a weird behavior on the use of register in parameters. C++17 compilers may warn about register keyword being deprecated. I found this happens only if the file with the register keyword is not placed in a system include folder. Be careful when using this option with a shared object like any *.so or *.dll, since ABI may be broken due to interface change. Checking compatibility is left to user's discretion.*
CMS_NO_VISIBILITY	*Uncomment this to remove visibility attribute when building shared objects.*

Table 1

DLL COMPILATION and use (Windows® only)

To ***use*** *Little CMS* as DLL, you need to define the symbol CMS_DLL when compiling ***lcms2.h***, this is easily done by using the toggle -DCMS_DLL on gcc, other compilers may use different syntax.

Similarly, to compile *Little CMS* to ***produce*** a DLL, you need to define the symbol CMS_DLL_BUILD. On Visual Studio, you can define this symbol on Properties, C/C++, Preprocessor, Preprocessor definitions. There is a project that builds such DLL in the Projects folder.

Asserting

Internally, *Little CMS* uses an internal assert function to catch run-time errors. This macro is not exposed to *Little CMS* API and is enabled only in debug builds. You can disable this functionality by editing lcms2_internal.h, although is highly recommended to leave untouched as a checking feature. In Release builds, no code is generated.

```
_cmsAssert(a)
```

Parameters:

a: logical expression

Returns:

None

Included files (dependencies)

*Used by **lcms2.h***

```
#include <stdio.h>
#include <limits.h>
#include <time.h>
#include <stddef.h>
```

*Used by **lcms2_plugin.h***

```
#include <stdlib.h>
#include <math.h>
#include <stdarg.h>
#include <memory.h>
#include <string.h>
```

Used Internally

```
#include <ctype.h>
```

Generic types

Basic types are automatically detected and defined by ***lcms2.h*** You can override them by defining **CMS_BASIC_TYPES_ALREADY_DEFINED**. In this case, you must define such types before including ***lcms2.h***

Basic Types	Bits	Signed	Comment
cmsUInt8Number	8	*No*	*Byte*
cmsInt8Number	8	*Yes*	
cmsUInt16Number	16	*No*	*Word*
cmsInt16Number	16	*Yes*	
cmsUInt32Number	32	*No*	*Double word*
cmsInt32Number	32	*Yes*	*Native int on most 32-bit architectures*
cmsUInt64Number	64	*No*	
cmsInt64Number	64	*Yes*	
cmsFloat32Number	32	*Yes*	*IEEE float*
cmsFloat64Number	64	*Yes*	*IEEE cmsFloat64Number*
cmsBool	?	*No*	*TRUE, FALSE Boolean type, which will be using the native integer*

Table 2

Derivative Types	Bits	Signed	Comment
cmsSignature	32	*No*	*Base type for ICC signatures*
cmsU8Fixed8Number	8.8 = 16	*No*	
cmsS15Fixed16Number	15.16 = 32	*Yes*	*Fixed point*
cmsU16Fixed16Number	16.16 = 32	*No*	

Table 3

Handles	Comment
cmsHANDLE	Generic handle=void*
cmsHPROFILE	Handle to a profile
cmsHTRANSFORM	Handle to a color transform

Table 4

Opaque typedefs	Comment
cmsContext	Pointer to undisclosed _cms_context_struct
cmsToneCurve	Pointer to undisclosed _cms_curve_struct
cmsMLU	Pointer to undisclosed _cms_MLU_struct
cmsIOHANDLER	Pointer to undisclosed _cms_io_handler
cmsNAMEDCOLORLIST	Pointer to undisclosed _cms_NAMEDCOLORLIST_struct

Table 5

Common constants and version retrival

Those are utility constants defined in ***lcms2.h***

Version/release

```
LCMS_VERSION    2170
```

Maximum number of chars in a path

```
cmsMAX_PATH     256
```

Maximum number of channels in ICC profiles

```
cmsMAXCHANNELS  16
```

Magic number to identify an ICC profile

cmsMagicNumber	0x61637370 'acsp'

Little CMS signature

lcmsSignature	0x6c636d73 'lcms'

2.8

```
int  cmsGetEncodedCMMversion(void);
```

Returns the value of LCMS_VERSION. This function is here to help applications to prevent mixing lcms versions on header and shared objects. A safety check can be used to prevent unwanted version mixing. i.e. assert(LCMS_VERSION == cmsGetEncodedCMMversion());

Parameters:

none

Returns:

the value of LCMS_VERSION.

Contexts

There are situations where several instances of Little CMS engine have to coexist but on different conditions. For example, when the library is used as a DLL or a shared object, diverse applications may want to use different plug-ins. Another example is when multiple threads are being used in same task and the user wants to pass thread-dependent information to the memory allocators or the logging system. For all this use, Little CMS 2.6 and above implements context handling functions. The type cmsContext is a pointer to an internal structure that keeps track of all plug-ins and static data needed by the THR corresponding function. A context-aware app could allocate a new context by calling *cmsCreateContext()* or duplicate a yet-existing one by using cmsDupContext(). Each context can hold different plug-ins, defined by the *Plugin* parameter when creating the context or later by calling cmsPluginTHR(). The context can also hold loggers, defined by using cmsSetLogErrorHandlerTHR() and other settings. To free context resources, cmsDeleteContext() does the job. Users may associate private data across a void pointer when creating the context, and can retrieve this pointer by using cmsGetContextUserData(). Context ID of 0 is a special case that holds the global context, for non-THR functions.

Important Note: Prior to 2.6, cmsContext was just a void pointer to user data. 2.6 redefined the meaning of contexts and therefore the binary backwards compatibility in the absolute sense was broken. However, the library tries to guess whatever the context is being used in the old way, and behave in consequence. Any cmsContext created by cmsCreateContext() or cmsDupContext() behaves in the new way. Otherwise it is assumed a void pointer to user data. Users are strongly encouraged to use cmsCreateContext() function instead of passing raw user data.

2.6

```
cmsContext   cmsCreateContext(void* Plugin,  void* UserData);
```

Creates a new context with optional associated plug-ins. Caller may specify an optional pointer to user-defined data that will be forwarded to plug-ins and logger.

Parameters:

Plugin: Pointer to plug-in collection. Set to NULL for no plug-ins.
UserData: optional pointer to user-defined data that will be forwarded to plug-ins and logger. Set to NULL for none.

Returns:

A valid cmsContext on success, or NULL on error.

***Note**: All memory used by this context is allocated by using the memory plugin, if present, this includes the block for the context itself.*

2.6

```
cmsContext  cmsDupContext(cmsContext ContextID,  void* NewUserData);
```

Duplicates a context with all associated plug-ins. Caller may specify an optional pointer to user-defined data that will be forwarded to plug-ins and logger.

Parameters:

UserData: optional pointer to user-defined data that will be forwarded to plug-ins and logger. Set to NULL for using user defined pointer from the source context.

Returns:

A valid cmsContext on success, or NULL on error.

2.6

```
void  cmsDeleteContext(cmsContext ContextID);
```

Frees any resources associated with the given context, and destroys the context placeholder. The ContextID can no longer be used in any THR operation.

Parameters:

ContextID: Handle to user-defined context.

Returns:

None

Notes:

The system context, ContextID = NULL cannot be used, the function does nothing in this case.

2.6

```
void*  cmsGetContextUserData(cmsContext ContextID);
```

Returns the user data associated to the given ContextID, or NULL if no user data was attached on context creation

Parameters:

ContextID: Handle to user-defined context.

Returns:

Pointer to a user-defined data or NULL if no data.

Notes:

The system context, ContextID = NULL cannot be used in this function.

2.0

```
cmsContext  cmsGetProfileContextID(cmsHPROFILE hProfile);
```

Returns the ContextID associated with a given profile.

Parameters:

hProfile: Handle to a profile object

Returns:

Pointer to a user-defined context cargo or NULL if no context

2.0

```
cmsContext  cmsGetTransformContextID(cmsHTRANSFORM hTransform);
```

Returns the ContextID associated with a given transform.

Parameters:

*hTransform**:** Handle to a color transform object.*

Returns:

Pointer to a user-defined context cargo or NULL if no context.

2.13

```
cmsContext  cmsGetStageContextID(const cmsStage* mpe);
```

Returns the ContextID associated with a given stage object

Parameters:

mpe: a pointer to a stage object.

Returns:

The context of a given stage object

Plug-Ins

By using plug-ins you can use the normal API to access customized functionality. Licensing is another compelling reason; you can move all your intellectual property into plug-ins and still be able to upgrade the core Little CMS library and still stay in the open source side. See the Plug-in API documentation for further information. The suggested way to use plug-ins is across contexts, but you can first allocate a context with no plug-ins and then invoke the cmsPluginTHR function. The easiest one works on the global context.

2.0

```
cmsBool cmsPlugin(void* Plugin);
```

Declares external extensions to the core engine **in the global context**. The "Plugin" parameter may hold one or several plug-ins, as defined by the plug-in developer.

Parameters:

Plugin: Pointer to plug-in collection.

Returns:

TRUE on success FALSE on error.

2.0

```
void  cmsUnregisterPlugins(void);
```

This function returns Little CMS **global context** to its default pristine state, as no plug-ins were declared. There is no way to unregister a single plug-in, as a single call to cmsPlugin() function may register many different plug-ins simultaneously, then there is no way to identify which plug-in to unregister.

Parameters:

None

Returns:

None

2.6

```
cmsBool cmsPluginTHR(cmsContext  ContextID, void* Plugin);
```

Installs a plug-in bundle in the given context.

Parameters:

ContextID: Handle to user-defined context.

Plugin: Pointer to plug-in bundle.

Returns:

TRUE on success FALSE on error.

2.6

```
void cmsUnregisterPluginsTHR(cmsContext ContextID);
```

This function returns the given context its default pristine state, as no plug-ins were declared. There is no way to unregister a single plug-in, as a single call to cmsPluginTHR() function may register many different plug-ins simultaneously, then there is no way to identify which plug-in to unregister.

Parameters:

ContextID: Handle to user-defined context.

Returns:

None

Error logging

When a function fails, it returns proper value. For example, all create functions does return NULL on failure. Other may return FALSE. It may be interesting, for the developer, to know why the function is failing, for that reason, Little CMS offers a logging function. This function will get an **english** string with some clues on what is going wrong. You can show this info to the end user if you wish so, or just create some sort of log on disk.

The logging function should NOT terminate the program, as this obviously can leave leaked resources. It is the programmer's responsability to check each function return code to make sure it didn't fail. The default logger does nothing.

Error family	Defined as
cmsERROR_UNDEFINED	0
cmsERROR_FILE	1
cmsERROR_RANGE	2
cmsERROR_INTERNAL	3
cmsERROR_NULL	4
cmsERROR_READ	5
cmsERROR_SEEK	6
cmsERROR_WRITE	7
cmsERROR_UNKNOWN_EXTENSION	8
cmsERROR_COLORSPACE_CHECK	9
cmsERROR_ALREADY_DEFINED	10
cmsERROR_BAD_SIGNATURE	11
cmsERROR_CORRUPTION_DETECTED	12
cmsERROR_NOT_SUITABLE	13

Table 6

Error logger is called with the ContextID when a message is raised. This gives the chance to know which thread is responsible of the warning and any environment associated with it. Non-contexted applications may ignore this parameter. Please note that, by default ContextID is 0 (the global context).

```
typedef void (* cmsLogErrorHandlerFunction)(cmsContext ContextID,
                                            cmsUInt32Number ErrorCode,
                                            const char *Text);
```

Definition of error logging callback.

2.0

```
void cmsSetLogErrorHandler(cmsLogErrorHandlerFunction Fn);
```

Allows user to set any specific logger on global context. Each time this function is called, the previous logger is replaced. Calling this functin with NULL as parameter, does reset the logger to the default Little CMS logger. The default Little CMS logger does nothing.

Parameters:

Fn: Callback to the logger (user defined function), or NULL to reset Little CMS to its default logger.

Returns:

None

2.6

```
void cmsSetLogErrorHandlerTHR(cmsContext ContextID,
                                             cmsLogErrorHandlerFunction Fn);
```

Allows user to set any specific logger for the given context. Each time this function is called, the previous logger is replaced. Calling this functin with NULL as parameter, does reset the logger to the default *Little CMS* logger. The default *Little CMS* logger does nothing.

Parameters:

ContextID: Handle to user-defined context, or NULL for the global context

Fn: Callback to the logger (user defined function), or NULL to reset Little CMS to its default logger.

Returns:

None

IO handlers

IO handlers are abstractions used to deal with files or streams. All reading/writing of ICC profiles and PostScript resources are done by using IO handlers. IO handlers do support random access. Advanced users may want to write their own IO handlers, see the plug-in API documentation for further details.

2.0

```
cmsIOHANDLER* cmsOpenIOhandlerFromFile(cmsContext ContextID,
                                       const char* FileName,
                                       const char* AccessMode);
```

Creates an IO handler object from a disk-based file. Note filename is limited to UTF-8 in this function.

Parameters:

ContextID: Pointer to a user-defined context cargo.
FileName: Full path of file resource
AccessMode: “r” to read, “w” to write.

Returns:

A pointer to an iohandler object on success, NULL on error.

2.0

```
cmsIOHANDLER* cmsOpenIOhandlerFromStream(cmsContext ContextID,
                                         FILE* Stream);
```

Creates an IO handler object from an already open stream.

Parameters:

ContextID: Pointer to a user-defined context cargo.

Returns:

A pointer to an iohandler object on success, NULL on error.

2.0

```
cmsIOHANDLER* cmsOpenIOhandlerFromMem(cmsContext ContextID,
                                      void *Buffer,
                                      cmsUInt32Number size,
                                      const char* AccessMode);
```

Creates an IO handler object from a memory block. Limited to 4Gb.

Parameters:

ContextID: Pointer to a user-defined context cargo.
Buffer: Points to a block of contiguous memory containing the data
size: Buffer's size measured in bytes.
AccessMode: "r" to read, "w" to write.

Returns:

A pointer to an iohandler object on success, NULL on error.

2.0

```
cmsIOHANDLER* cmsOpenIOhandlerFromNULL(cmsContext ContextID);
```

Creates a void iohandler object (similar to a file iohandler on /dev/null). All read operations returns 0 bytes and sets the EOF flag. All write operations discards the given data.

Parameters:

ContextID: Pointer to a user-defined context cargo.

Returns:

A pointer to an iohandler object on success, NULL on error.

2.0

```
cmsBool   cmsCloseIOhandler(cmsIOHANDLER* io);
```

Closes the iohandler object, freeing any associated resources.

Parameters:

io: A pointer to an iohandler object.

Returns:

TRUE on success, FALSE on error. Note that on file write operations, the real flushing to disk may happen on closing the iohandler, so it is important to check the return code.

2.8

```
cmsIOHANDLER* cmsGetProfileIOhandler(cmsHPROFILE hProfile);
```

Returns the *iohandler* used by a given profile object.

Parameters:

> *hProfile: Handle to a profile object*

Returns:

> *On success, a pointer to the iohandler object used by the profile. NULL on error.*

Profile access functions

These are the basic functions on opening profiles. For simpler operation, you must open two profiles using *cmsOpenProfileFromFile*, and then create a transform with these open profiles with *cmsCreateTransform*. Using this transform you can color correct your bitmaps by *cmsDoTransform*. When you are done you must free the transform AND the profiles by *cmsDeleteTransform* and *cmsCloseProfile*.

2.0

```
cmsHPROFILE  cmsOpenProfileFromFile(const char *ICCProfile,
                                              const char *sAccess);
```

Opens a file-based ICC profile returning a handle to it.

Parameters:

ICCProfile: File name w/ full path.

sAccess: "r" for normal operation, "w" for profile creation

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE  cmsOpenProfileFromFileTHR(cmsContext ContextID,
                                                 const char *ICCProfile,
                                                 const char *sAccess);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.

ICCProfile: File name w/ full path.

sAccess: "r" for normal operation, "w" for profile creation

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE   cmsOpenProfileFromStream(FILE* ICCProfile, const char* sAccess);
```

Opens a stream-based ICC profile returning a handle to it.

Parameters:

ICCProfile: stream holding the ICC profile.
sAccess: "r" for normal operation, "w" for profile creation

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE  cmsOpenProfileFromStreamTHR(cmsContext ContextID,
                                          FILE* ICCProfile,
                                          const char* sAccess);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE   cmsOpenProfileFromMem(const void * MemPtr,
                                    cmsUInt32Number dwSize);
```

Opens an ICC profile which is entirely contained in a memory block. Useful for accessing embedded profiles. MemPtr must point to a buffer of at least dwSize bytes. This buffer must hold a full profile image. Memory must be contiguous.

Parameters:

MemPtr: Points to a block of contiguous memory containing the profile
dwSize: Profile's size measured in bytes.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE  cmsOpenProfileFromMemTHR(cmsContext ContextID,
                                       const void * MemPtr, cmsUInt32Number dwSize);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.
MemPtr: Points to a block of contiguous memory containing the profile
dwSize: Profile's size measured in bytes.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE  cmsOpenProfileFromIOhandlerTHR(cmsContext ContextID,
                                             cmsIOHANDLER* io);
```

Opens a profile, returning a handle to it. The profile access is described by an IOHANDLER. See IO handlers section for further details.

Parameters:

ContextID: Pointer to a user-defined context cargo.
Io: Pointer to a serialization object.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.6

```
cmsHPROFILE cmsOpenProfileFromIOhandler2THR(cmsContext ContextID,
                                            cmsIOHANDLER* io
                                            cmsBool write);
```

Opens a profile, returning a handle to it. The profile access is described by an IOHANDLER. See IO handlers section for further details. This function allows to specify write access as well

Parameters:

ContextID: Pointer to a user-defined context cargo.
Io: Pointer to a serialization object.
write: TRUE to grant write access, FALSE to open the IOHANDLER as read only

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsBool  cmsCloseProfile(cmsHPROFILE hProfile);
```

Closes a profile handle and frees any associated resource. Can return error when creating disk profiles, as this function flushes the data to disk.

Parameters:

hProfile: Handle to a profile object.

Returns:

TRUE on success, FALSE on error

2.0

```
cmsBool  cmsSaveProfileToFile(cmsHPROFILE hProfile, const char* FileName);
```

Saves the contents of a profile to a given filename.

Parameters:

hProfile: Handle to a profile object
ICCProfile: File name w/ full path.

Returns:

TRUE on success, FALSE on error.

2.0

```
cmsBool   cmsSaveProfileToStream(cmsHPROFILE hProfile, FILE* Stream);
```

Saves the contents of a profile to a given stream.

Parameters:

hProfile: Handle to a profile object

Returns:

TRUE on success, FALSE on error.

2.0

```
cmsBool  cmsSaveProfileToMem(cmsHPROFILE hProfile,
                                       void *MemPtr,  cmsUInt32Number* BytesNeeded);
```

Same as anterior, but for memory blocks. In this case, a NULL as MemPtr means to calculate needed space only.

Parameters:

hProfile: Handle to a profile object.

MemPtr: Points to a block of contiguous memory with enough space to contain the profile

BytesNeeded: points to a cmsUInt32Number, where the function will store profile's size measured in bytes.

Returns:

TRUE on success, FALSE on error.

2.0

```
cmsUInt32Number  cmsSaveProfileToIOhandler(cmsHPROFILE hProfile,
                                           cmsIOHANDLER* io);
```

Low-level save to IOHANDLER. It returns the number of bytes used to store the profile, or zero on error. io may be NULL and in this case no data is written--only sizes are calculated.

Parameters:

hProfile: Handle to a profile object

Io: Pointer to a serialization object.

Returns:

The number of bytes used to store the profile, or zero on error.

Predefined virtual profiles

2.0

```
cmsHPROFILE  cmsCreateProfilePlaceholder(cmsContext ContextID);
```

Creates an empty profile object, ready to be populated by the programmer.

WARNING: The obtained profile without adding any information is not directly useable.

Parameters:

ContextID: Pointer to a user-defined context cargo.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE cmsCreateRGBProfile(const cmsCIExyY* WhitePoint,
                                const cmsCIExyYTRIPLE* Primaries,
                                cmsToneCurve* const TransferFunction[3]);
```

This function creates a display RGB profile based on White point, primaries and transfer functions. It populates following tags; this conform a standard RGB Display Profile, and then adds (As per addendum II of ICC spec) chromaticity tag.

1	cmsSigProfileDescriptionTag
2	cmsSigMediaWhitePointTag
3	cmsSigRedColorantTag
4	cmsSigGreenColorantTag
5	cmsSigBlueColorantTag
6	cmsSigRedTRCTag
7	cmsSigGreenTRCTag
8	cmsSigBlueTRCTag
9	Chromatic adaptation Tag
10	cmsSigChromaticityTag

Parameters:

WhitePoint: The white point of the RGB device or space.

Primaries: The primaries in xyY of the device or space.

TransferFunction[]: 3 tone curves describing the device or space gamma.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE    cmsCreateRGBProfileTHR(cmsContext ContextID,
                                      const cmsCIExyY* WhitePoint,
                                      const cmsCIExyYTRIPLE* Primaries,
                                      cmsToneCurve* const TransferFunction[3]);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.

WhitePoint: The white point of the RGB device or space.

Primaries: The primaries in xyY of the device or space.

TransferFunction[]: 3 tone curves describing the device or space gamma.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE    cmsCreateGrayProfile(const cmsCIExyY* WhitePoint,
                                    const cmsToneCurve* TransferFunction);
```

This function creates a gray profile based on White point and transfer function. It populates following tags; this conform a standard gray display profile.

1	cmsSigProfileDescriptionTag
2	cmsSigMediaWhitePointTag
3	cmsSigGrayTRCTag

Parameters:

WhitePoint: The white point of the gray device or space.

TransferFunction: tone curve describing the device or space gamma.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE    cmsCreateGrayProfileTHR(cmsContext ContextID,
                                        const cmsCIExyY* WhitePoint,
                                        const cmsToneCurve* TransferFunction);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.

WhitePoint: The white point of the gray device or space.

TransferFunction: tone curve describing the device or space gamma.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE   cmsCreateLinearizationDeviceLink(cmsColorSpaceSignature Space,
                                               cmsToneCurve* const TransferFunctions[]);
```

This is a devicelink operating in the target colorspace with as many transfer functions as components.

Parameters:

Space: any cmsColorSpaceSignature from Table 10

TransferFunction[]: tone curves describing the device or space linearization.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE    cmsCreateLinearizationDeviceLinkTHR(cmsContext ContextID,
                                               cmsColorSpaceSignature ColorSpace,
                                               cmsToneCurve* const TransferFunctions[]);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.

ColorSpace: any cmsColorSpaceSignature from Table 10

TransferFunction[]: tone curves describing the device or space linearization.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE cmsCreateInkLimitingDeviceLink(cmsColorSpaceSignature Space,
                                            cmsFloat64Number Limit);
```

This is a devicelink operating in CMYK for ink-limiting.

Ink-limiting algorithm:

```
Sum = C + M + Y + K
If Sum > InkLimit
    Ratio= 1 - (Sum - InkLimit) / (C + M + Y)
    if Ratio <0
        Ratio=0
    endif
Else
  Ratio=1
endif

C = Ratio * C
M = Ratio * M
Y = Ratio * Y
K: Does not change
```

Parameters:

Space: any cmsColorSpaceSignature from Table 10. Currently only cmsSigCmykData is supported.

Limit: Amount of ink limiting in % (0..400%)

Returns:

A handle to an ICC profile object on success, NULL on error.

2.16

```
cmsHPROFILE cmsCreateDeviceLinkFromCubeFile (const char*  cFileName);
```

Imports a 3D LUT file as a devicelink profile operating in the RGB color model. The file format corresponds to .CUBE file format, defined by Adobe in document cube-lut-specification-1.0.pdf

Some .cube files may contains negative numbers, LittleCMS can only deal with those numbers on unbounded mode, so you need to use floating point transforms to get negative numbers.

Parameters:

cFileName: The .CUBE file to import.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.16

```
cmsHPROFILE cmsCreateDeviceLinkFromCubeFileTHR (cmsContext ContextID,
                                                const char*  cFileName);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.
cFileName: The .CUBE file to import.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE cmsCreateInkLimitingDeviceLinkTHR(cmsContext ContextID,
                                               cmsColorSpaceSignature Space,
                                               cmsFloat64Number Limit);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.
Space: any cmsColorSpaceSignature from Table 10. Currently only cmsSigCmykData is supported.
Limit: Amount of ink limiting in % (0..400%)

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE  cmsCreateLab2Profile(const cmsCIExyY* WhitePoint);
```

Creates a Lab → Lab identity, marking it as v2 ICC profile. Adjustments for accomodating PCS endoing shall be done by *Little CMS* when using this profile.

Parameters:

WhitePoint: Lab reference white. NULL for D50.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE  cmsCreateLab2ProfileTHR(cmsContext ContextID,
                                      const cmsCIExyY* WhitePoint);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.
WhitePoint: Lab reference white. NULL for D50.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE    cmsCreateLab4Profile(const cmsCIExyY* WhitePoint);
```

Creates a Lab → Lab identity, marking it as v4 ICC profile.

Parameters:

WhitePoint: Lab reference white. NULL for D50.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE    cmsCreateLab4ProfileTHR(cmsContext ContextID,
                                        const cmsCIExyY* WhitePoint);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.
WhitePoint: Lab reference white. NULL for D50.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE    cmsCreateXYZProfile(void);
```

Creates a XYZ → XYZ identity, marking it as v4 ICC profile. WhitePoint used in Absolute colorimetric intent is D50.

Parameters:

None

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE    cmsCreateXYZProfileTHR(cmsContext ContextID);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE    cmsCreate_sRGBProfile(void);
```

Create an ICC virtual profile for sRGB space. sRGB is a standard RGB color space created cooperatively by HP and Microsoft in 1996 for use on monitors, printers, and the Internet.

sRGB white point is D65.	
xyY	0.3127, 0.3291, 1.0

Primaries are ITU-R BT.709-5 (xYY)	
R	0.6400, 0.3300, 1.0
G	0.3000, 0.6000, 1.0
B	0.1500, 0.0600, 1.0

sRGB transfer functions are defined by:

```
If $R'_{sRGB}, G'_{sRGB}, B'_{sRGB} < 0.04045$

    $R = R'_{sRGB} / 12.92$
    $G = G'_{sRGB} / 12.92$
    $B = B'_{sRGB} / 12.92$

else if $R'_{sRGB}, G'_{sRGB}, B'_{sRGB} >= 0.04045$

    $R = ((R'_{sRGB} + 0.055) / 1.055)^{2.4}$
    $G = ((G'_{sRGB} + 0.055) / 1.055)^{2.4}$
    $B = ((B'_{sRGB} + 0.055) / 1.055)^{2.4}$
```

Parameters:

None

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE    cmsCreate_sRGBProfileTHR(cmsContext ContextID);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE    cmsCreateNULLProfile(void);
```

Creates a fake NULL profile. This profile returns 1 channel as always 0. Is useful only for gamut checking tricks.

Parameters:

None

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE    cmsCreateNULLProfileTHR(cmsContext ContextID);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE  cmsCreateBCHSWabstractProfile(int nLUTPoints,
                                           cmsFloat64Number Bright,
                                           cmsFloat64Number Contrast,
                                           cmsFloat64Number Hue,
                                           cmsFloat64Number Saturation,
                                           int TempSrc,
                                           int TempDest);
```

Creates an abstract devicelink operating in Lab for Bright/Contrast/Hue/Saturation and white point translation. White points are specified as temperatures ºK

Parameters B,C,H,S are applied in the L*C*h space as follows:

```
   LChOut.L = LChIn.L * bchsw ->Contrast + bchsw ->Brightness;
   LChOut.C = LChIn.C + bchsw ->Saturation;
   LChOut.h = LChIn.h + bchsw ->Hue;
```

- contrast should be close to 1.0, this is the L* slope.
- Brightness goes from -100 to 100, this is the L* offset
- Hue comes in degrees, and is the offset applied

Parameters:

nLUTPoints : Resulting color map resolution
Bright: Bright increment. May be negative
Contrast : Contrast increment. May be negative.
Hue : Hue displacement in degree.
Saturation: Saturation increment. May be negative
TempSrc: Source white point temperature
TempDest: Destination white point temperature.

Returns:

A handle to an ICC profile object on success, NULL on error.

Notes:

To prevent white point adjustment, set TempSrc = TempDest = 0

2.0

```
cmsHPROFILE    cmsCreateBCHSWabstractProfileTHR(cmsContext ContextID,
                                                int nLUTPoints,
                                                cmsFloat64Number Bright,
                                                cmsFloat64Number Contrast,
                                                cmsFloat64Number Hue,
                                                cmsFloat64Number Saturation,
                                                int TempSrc,
                                                int TempDest);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.
nLUTPoints : Resulting colormap resolution
Bright: Bright increment. May be negative
Contrast : Contrast increment. May be negative.
Hue : Hue displacement in degree.
Saturation: Saturation increment. May be negative
TempSrc, TempDest: Source, Destination white point temperatures

Returns:

A handle to an ICC profile object on success, NULL on error.

2.0

```
cmsHPROFILE    cmsTransform2DeviceLink(cmsHTRANSFORM hTransform,
                                       cmsFloat64Number Version,
                                       cmsUInt32Number dwFlags);
```

Generates a device-link profile from a given color transform. This profile can then be used by any other function accepting profile handle. Depending on the specified version number, the implementation of the devicelink may vary. Accepted versions are in range 1.0…4.4

Parameters:

*hTransform**:** Handle to a color transform object.*
Version: The target devicelink version number.
dwFlags: A combination of bit-field constants described in Table 42.

Returns:

A handle to an ICC profile object on success, NULL on error.

2.16

```
cmsHPROFILE  cmsCreate_OkLabProfile(cmsContext ctx)
```

Generates a profile for OkLab color space . The predefined type **TYPE_OKLAB_DBL** can be used for this colorspace. The profile works in both directions.

See https://bottosson.github.io/posts/oklab/

WARNING: This profile cannot be saved it uses features not supported in the because the ICC format , it only works as a virtual profile.

Parameters:

ContextID: Handle to user-defined context, or NULL for the global context

Returns:

A handle to an ICC profile object on success, NULL on error.

Obtaining localized info from profiles

In versions prior to 4.0, the ICC format defined a required tag 'desc' which stored ASCII, Unicode, and Script Code versions of the profile description for display purposes. However, this structure allowed the profile to be localized for one language only through Unicode or Script Code. Profile vendors had to ship many localized versions to different countries. It also created problems when a document with localized profiles embedded in it was shipped to a system using a different language. With the adoption of V4 spec as basis, Little CMS solves all those issues honoring a new tag type: 'mluc' and multi localized Unicode. There is a full part of the API to deal with this stuff, but if you don't care about the details and all you want is to display the right string, *Little CMS* provides a simplified interface for that purpose.

Note that ASCII is strictly 7 bits, so you need to use wide chars if you want to preserve the information in the profile. The localization trick is done by using the lenguage and country codes, which you are supposed to supply. Those are two or three ASCII letters. A list of codes may be found here:

Language Code: http://lcweb.loc.gov/standards/iso639-2/iso639jac.html

Country Codes: http://www.iso.ch/iso/en/prods-services/iso3166ma/index.html

In practice, **"en"** for "english" and **"US"** for "united states" are implemented in most profiles. It is Ok to set a language and a country even if the profile does not implement such specific language and country. *Little CMS* will search for a proper match.

If you don't care and want just to take the first string in the profile, you can use:

For the language:

```
cmsNoLanguage
```

For the country:

```
cmsNoCountry
```

This will force to get the very first string, without any searching. A note of warning on that: you will get an string, but the language would be any, and probably that is not what you want. It is better to specify a default for language, and let LittleCMS to choose any other country (or language!) if what you ask for is not available.

```
typedef enum {
        cmsInfoDescription   = 0,
        cmsInfoManufacturer= 1,
        cmsInfoModel         = 2,
        cmsInfoCopyright     = 3
} cmsInfoType;
```

Reading the unicode variant on V2 profiles

Since 2.16, a special setting for the lenguage and country allows to access the unicode variant on V2 profiles.

For the language:

```
cmsV2Unicode
```

For the country:

```
cmsV2Unicode
```

Many V2 profiles have this field empty or filled with bogus values. Previous versions of Little CMS were ignoring it, but with this additional setting, correct V2 profiles with two variants can be honored now. By default, the ASCII variant is returned on V2 profiles unless you specify this special setting. If you decide to use it, check the result for empty strings and if this is the case, repeat reading by using the normal path.

2.0

```
cmsUInt32Number cmsGetProfileInfo(cmsHPROFILE hProfile,
                                                   cmsInfoType Info,
                                                   const char LanguageCode[3],
                                                   const char CountryCode[3],
                                                   wchar_t* Buffer,
                                                   cmsUInt32Number BufferSize);
```

Gets several information strings from the profile, dealing with localization. Strings are returned as wide chars.

Parameters:

hProfile: Handle to a profile object
Info: A selector of which info to return
Language Code: first name language code from ISO-639/2.
Country Code: first name region code from ISO-3166.
Buffer: pointer to a memory block to get the result. NULL to calculate size only
BufferSize: Amount of byes allocated in Buffer, or 0 to calculate size only.

Returns:

Number of required bytes to hold the result. 0 on error.

2.0

```
cmsUInt32Number cmsGetProfileInfoASCII(cmsHPROFILE hProfile,
                                                        cmsInfoType Info,
                                                        const char LanguageCode[3],
                                                        const char CountryCode[3],
                                                        char* Buffer,
                                                        cmsUInt32Number BufferSize);
```

Gets several information strings from the profile, dealing with localization. Strings are returned as ASCII.

Parameters:

hProfile: Handle to a profile object
Info: A selector of which info to return
Language Code: first name language code from ISO-639/2.
Country Code: first name region code from ISO-3166.
Buffer: pointer to a memory block to get the result. NULL to calculate size only
BufferSize: Amount of byes allocated in Buffer, or 0 to calculate size only.

Returns:

Number of required bytes to hold the result. 0 on error.

Profile feature detection

2.0

```
cmsBool   cmsDetectBlackPoint(cmsCIEXYZ* BlackPoint,
                              cmsHPROFILE hProfile,
                              cmsUInt32Number Intent,
                              cmsUInt32Number dwFlags);
```

Estimate the black point of a given profile. Used by black point compensation algorithm.

Parameters:

BlackPoint: Pointer to cmsCIEXYZ *object to receive the detected black point.*
hProfile: Handle to a profile object
Intent: A cmsUInt32Number *holding the intent code, as described in* Intents *section.*
dwFlags: reserved (unused). Set it to 0

Returns:

TRUE on success, FALSE on error

2.8

```
cmsBool   cmsDetectDestinationBlackPoint(cmsCIEXYZ* BlackPoint,
                              cmsHPROFILE hProfile,
                              cmsUInt32Number Intent,
                              cmsUInt32Number dwFlags);
```

Estimate the black point of a given destination profile by using the Black point compensation ICC algorithm.

Parameters:

BlackPoint: Pointer to cmsCIEXYZ *object to receive the detected black point.*
hProfile: Handle to a profile object
Intent: A cmsUInt32Number *holding the intent code, as described in* Intents *section.*
dwFlags: reserved (unused). Set it to 0

Returns:

TRUE on success, FALSE on error

2.0

```
cmsFloat64Number   cmsDetectTAC(cmsHPROFILE hProfile);
```

When several colors are printed on top of each other, there is a limit to the amount of ink that can be put on paper. This maximum total dot percentage is referred to as either TIC (Total Ink Coverage) or TAC (Total Area Coverage). This function does estimate total area coverage for a given profile in %. Only works on output profiles. On RGB profiles, 400% is returned. TAC is detected by subsampling Lab color space on 6x74x74 points.

Parameters:

hProfile: Handle to a profile object

Returns:

Estimated area coverage in % on success, 0 on error.

2.13

```
cmsFloat64Number   cmsDetectRGBProfileGamma (cmsHPROFILE hProfile,
                                             cmsFloat64Number   thereshold);
```

Detect whatever a given ICC profile works in linear (gamma 1.0) space. Actually, doing that "well" is quite hard, because each component may behave completely different. Since the true point of this function is to detect suitable optimizations, I am imposing some requirements that simplifies things: only RGB, and only profiles that can got in both directions. The algorithm obtains Y from a synthetic gray R=G=B. Then a least squares fitting algorithm is then used to estimate gamma. For gamma close to 1.0, RGB is linear. On profiles not supported, -1 is returned.

Parameters:

hProfile: Handle to a profile object

threshold: The standard deviation above that gamma is returned.

Returns:

Estimated gamma of the RGB space on success, -1 on error.

Accessing profiler header

2.0

```
cmsBool  cmsGetHeaderCreationDateTime(cmsHPROFILE hProfile, struct tm *Dest);
```

Returns the date and time when profile was created. This is a field stored in profile header.

Parameters:

hProfile: Handle to a profile object

Dest: pointer to struct tm object to hold the result.

Returns:

TRUE on success, FALSE on error

2.0

```
cmsUInt32Number   cmsGetHeaderFlags(cmsHPROFILE hProfile);
```

Get header flags of given ICC profile object. The profile flags field does contain flags to indicate various hints for the CMM such as distributed processing and caching options. The least-significant 16 bits are reserved for the ICC. Flags in bit positions 0 and 1 shall be used as indicated in Table 7.

Position	**Field Length (bits)**	**Field Contents**
0	1	Embedded Profile (cmsEmbeddedProfileFalse if not embedded, cmsEmbeddedProfileTrue if embedded in file)
1	1	Profile cannot be used independently from the embedded color data (set to cmsUseWithEmbeddedDataOnly if true, cmsUseAnywhere if false)

Table 7

Parameters:

hProfile: Handle to a profile object

Returns:

Flags field of profile header.

2.0

```
void  cmsSetHeaderFlags(cmsHPROFILE hProfile, cmsUInt32Number Flags);
```

Sets header flags of given ICC profile object. Valid flags are defined in Table 7.

Parameters:

hProfile: Handle to a profile object.
Flags: Flags field of profile header.

Returns:

None

2.0

```
cmsUInt32Number   cmsGetHeaderManufacturer(cmsHPROFILE hProfile);
```

Returns the manufacturer signature as described in the header. This funcionality is widely superseded by the manufaturer tag. Of use only in elder profiles.

Parameters:

hProfile: Handle to a profile object

Returns:

The profile manufacturer signature stored in the header.

2.0

```
void  cmsSetHeaderManufacturer(cmsHPROFILE hProfile,
                                             cmsUInt32Number manufacturer);
```

Sets the manufacturer signature in the header. This funcionality is widely superseded by the manufaturer tag. Of use only in elder profiles.

Parameters:

hProfile: Handle to a profile object.
Manufacturer: The profile manufacturer signature to store in the header.

Returns:

None

2.0

```
cmsUInt32Number   cmsGetHeaderModel(cmsHPROFILE hProfile);
```

Returns the model signature as described in the header. This funcionality is widely superseded by the model tag. Of use only in elder profiles.

Parameters:

hProfile: Handle to a profile object

Returns:

The profile model signature stored in the header.

2.0

```
void  cmsSetHeaderModel(cmsHPROFILE hProfile, cmsUInt32Number model);
```

Sets the model signature in the profile header. This funcionality is widely superseded by the model tag. Of use only in elder profiles.

Parameters:

hProfile: Handle to a profile object
model: The profile model signature to store in the header.

Returns:

None

Device attributes

currently defined values correspond to the low 4 bytes of the 8 byte attribute quantity.

cmsReflective	cmsTransparency
cmsGlossy	cmsMatte

Table 8

2.0

```
void   cmsGetHeaderAttributes(cmsHPROFILE hProfile , cmsUInt64Number* Flags );
```

Gets the attribute flags as described in Table 8.

Parameters:

hProfile: Handle to a profile object

Flags: a pointer to a cmsUInt64Number to receive the flags.

Returns:

None

2.0

```
void  cmsSetHeaderAttributes(cmsHPROFILE hProfile, cmsUInt64Number Flags);
```

Sets the attribute flags in the profile header. Flags are enumerated in Table 8.

Parameters:

hProfile: Handle to a profile object

Flags: The flags to be set.

Returns:

None

Profile classes

Device Class (cmsProfileClassSignature)	
cmsSigInputClass	0x73636E72 'scnr'
cmsSigDisplayClass	0x6D6E7472 'mntr'
cmsSigOutputClass	0x70727472 'prtr'
cmsSigLinkClass	0x6C696E6B 'link'
cmsSigAbstractClass	0x61627374 'abst'
cmsSigColorSpaceClass	0x73706163 'spac'
cmsSigNamedColorClass	0x6e6d636c 'nmcl'

Table 9

2.0

```
cmsProfileClassSignature  cmsGetDeviceClass(cmsHPROFILE hProfile);
```

Gets the device class signature from profile header.

Parameters:

hProfile: Handle to a profile object

Returns:

Device class of profile as described in Table 9

2.0

```
void    cmsSetDeviceClass(cmsHPROFILE hProfile, cmsProfileClassSignature sig);
```

Sets the device class signature in profile header.

Parameters:

hProfile: Handle to a profile object
sig: Device class of profile as described in Table 9

Returns:

None

Profile versioning

2.0

```
void    cmsSetProfileVersion(cmsHPROFILE hProfile, cmsFloat64Number Version);
```

Sets the ICC version in profile header. The version is given to this function as a float n.m

Parameters:

hProfile: Handle to a profile object

Version: Profile version in readable floating point format.

Returns:

None

2.0

```
cmsFloat64Number    cmsGetProfileVersion(cmsHPROFILE hProfile);
```

Returns the profile ICC version. The version is decoded to readable floating point format.

Parameters:

hProfile: Handle to a profile object

Returns:

The profile ICC version, in readable floating point format.

2.0

```
cmsUInt32Number   cmsGetEncodedICCversion(cmsHPROFILE hProfile);
```

Returns the profile ICC version in the same format as it is stored in the header.

Parameters:

hProfile: Handle to a profile object

Returns:

The encoded ICC profile version.

2.0

```
void   cmsSetEncodedICCversion(cmsHPROFILE hProfile,
                                                   cmsUInt32Number Version);
```

Sets the ICC version in profile header, without any decoding.

Parameters:

hProfile: Handle to a profile object

Version: Profile version in the same format as it will be stored in profile header.

Returns:

None

Info on profile implementation

2.0

```
cmsBool  cmsIsMatrixShaper(cmsHPROFILE hProfile);
```

Returns whatever a matrix-shaper is present in the profile. Note that a profile may hold matrix-shaper and CLUT as well.

Parameters:

hProfile: Handle to a profile object

Returns:

TRUE if the profile holds a matrix-shaper, FALSE otherwise.

2.1

```
cmsBool   cmsIsCLUT(cmsHPROFILE hProfile,
                    cmsUInt32Number Intent,
                    cmsUInt32Number UsedDirection);
```

Returns whatever a CLUT is present in the profile for the given intent and direction.

Parameters:

hProfile: Handle to a profile object

Intent: A *cmsUInt32Number holding the intent code, as described in Intents section.*

Direction: any of following values:

```
#define LCMS_USED_AS_INPUT      0
#define LCMS_USED_AS_OUTPUT     1
#define LCMS_USED_AS_PROOF      2
```

Returns:

TRUE if a CLUT is present for given intent and direction, FALSE otherwise.

Color spaces

cmsColorSpaceSignature	
cmsSigXYZData	0x58595A20 'XYZ '
cmsSigLabData	0x4C616220 'Lab '
cmsSigLuvData	0x4C757620 'Luv '
cmsSigYCbCrData	0x59436272 'YCbr'
cmsSigYxyData	0x59787920 'Yxy '
cmsSigRgbData	0x52474220 'RGB '
cmsSigGrayData	0x47524159 'GRAY'
cmsSigHsvData	0x48535620 'HSV '
cmsSigHlsData	0x484C5320 'HLS '
cmsSigCmykData	0x434D594B 'CMYK'
cmsSigCmyData	0x434D5920 'CMY '
cmsSigMCH1Data	0x4D434831 'MCH1'
cmsSigMCH2Data	0x4D434832 'MCH2'
cmsSigMCH3Data	0x4D434833 'MCH3'
cmsSigMCH4Data	0x4D434834 'MCH4'
cmsSigMCH5Data	0x4D434835 'MCH5'
cmsSigMCH6Data	0x4D434836 'MCH6'
cmsSigMCH7Data	0x4D434837 'MCH7'
cmsSigMCH8Data	0x4D434838 'MCH8'
cmsSigMCH9Data	0x4D434839 'MCH9'
cmsSigMCHAData	0x4D43483A 'MCHA'
cmsSigMCHBData	0x4D43483B 'MCHB'
cmsSigMCHCData	0x4D43483C 'MCHC'
cmsSigMCHDData	0x4D43483D 'MCHD'
cmsSigMCHEData	0x4D43483E 'MCHE'
cmsSigMCHFData	0x4D43483F 'MCHF'
cmsSigNamedData	0x6e6d636c 'nmcl'
cmsSig1colorData	0x31434C52 '1CLR'
cmsSig2colorData	0x32434C52 '2CLR'
cmsSig3colorData	0x33434C52 '3CLR'
cmsSig4colorData	0x34434C52 '4CLR'
cmsSig5colorData	0x35434C52 '5CLR'
cmsSig6colorData	0x36434C52 '6CLR'
cmsSig7colorData	0x37434C52 '7CLR'
cmsSig8colorData	0x38434C52 '8CLR'
cmsSig9colorData	0x39434C52 '9CLR'
cmsSig10colorData	0x41434C52 'ACLR'
cmsSig11colorData	0x42434C52 'BCLR'
cmsSig12colorData	0x43434C52 'CCLR'
cmsSig13colorData	0x44434C52 'DCLR'
cmsSig14colorData	0x45434C52 'ECLR'
cmsSig15colorData	0x46434C52 'FCLR'
cmsSigLuvKData	0x4C75764B 'LuvK'

Table 10

2.0

```
cmsUInt32Number   cmsChannelsOf(cmsColorSpaceSignature ColorSpace);
```

Returns channel count for a given color space.

Parameters:

ColorSpace: any cmsColorSpaceSignature from Table 10

Returns:

Number of channels, or 3 on error.

2.0

```
cmsColorSpaceSignature    cmsGetPCS(cmsHPROFILE hProfile);
```

Gets the profile connection space used by the given profile, using the ICC convention.

Parameters:

hProfile: Handle to a profile object

Returns:

Obtained cmsColorSpaceSignature (Table 10).

2.0

```
void   cmsSetPCS(cmsHPROFILE hProfile, cmsColorSpaceSignature pcs);
```

Sets the profile connection space signature in profile header, using ICC convention.

Parameters:

hProfile: Handle to a profile object
pcs: any cmsColorSpaceSignature from Table 10

Returns:

None

2.0

```
cmsColorSpaceSignature    cmsGetColorSpace(cmsHPROFILE hProfile);
```

Gets the color space used by the given profile, using the ICC convention.

Parameters:

hProfile: Handle to a profile object

Returns:

Obtained cmsColorSpaceSignature (Table 10).

2.0

```
void    cmsSetColorSpace(cmsHPROFILE hProfile, cmsColorSpaceSignature sig);
```

Sets the color space signature in profile header, using ICC convention.

Parameters:

hProfile: Handle to a profile object
sig: any cmsColorSpaceSignature from Table 10

Returns:

None

Containers in floating point format

cmsCIEXYZ	
cmsFloat64Number	X;
cmsFloat64Number	Y;
cmsFloat64Number	Z;

Table 11

cmsCIExyY	
cmsFloat64Number	x;
cmsFloat64Number	y;
cmsFloat64Number	Y;

Table 12

cmsCIELab	
cmsFloat64Number	L;
cmsFloat64Number	a;
cmsFloat64Number	b;

Table 13

cmsCIELCh	
cmsFloat64Number	L;
cmsFloat64Number	C;
cmsFloat64Number	h;

Table 14

cmsJCh	
cmsFloat64Number	J;
cmsFloat64Number	C;
cmsFloat64Number	h;

Table 15

cmsCIEXYZTRIPLE	
cmsCIEXYZ	Red;
cmsCIEXYZ	Green;
cmsCIEXYZ	Blue;

Table 16

cmsCIExyYTRIPLE	
cmsCIExyY	Red;
cmsCIExyY	Green;
cmsCIExyY	Blue;

Table 17

Colorspace conversions

D50 XYZ normalized to Y=1.0	
cmsD50X	0.9642
cmsD50Y	1.0
cmsD50Z	0.8249

Table 18

V4 perceptual black	
cmsPERCEPTUAL_BLACK_X	0.00336
cmsPERCEPTUAL_BLACK_Y	0.0034731
cmsPERCEPTUAL_BLACK_Z	0.00287

Table 19

2.0

```
const cmsCIEXYZ* cmsD50_XYZ(void);
const cmsCIExyY* cmsD50_xyY(void);
```

Returns pointer to constant structures.

Parameters:

None

Returns:

Pointers to constant D50 white point in XYZ and xyY spaces.

2.0

```
void cmsXYZ2xyY(cmsCIExyY* Dest, const cmsCIEXYZ* Source);
void cmsxyY2XYZ(cmsCIEXYZ* Dest, const cmsCIExyY* Source);
```

Colorimetric space conversions.

Parameters:

Source, Dest: Source and destination values.

Returns:

None

2.0

```
void cmsXYZ2Lab(const cmsCIEXYZ* WhitePoint,
                        cmsCIELab* Lab,
                        const cmsCIEXYZ* xyz);

void cmsLab2XYZ(const cmsCIEXYZ* WhitePoint,
                        cmsCIEXYZ* xyz,
                        const cmsCIELab* Lab);
```

Colorimetric space conversions. Setting WhitePoint to NULL forces D50 as white point.

Parameters:

Lab: Pointer to a cmsCIELab value as described in Table 13
xyz: Pointer to a cmsCIEXYZ value as described in Table 11

Returns:

None

2.0

```
void cmsLab2LCh(cmsCIELCh*LCh, const cmsCIELab* Lab);
void cmsLCh2Lab(cmsCIELab* Lab, const cmsCIELCh* LCh);
```

Colorimetric space conversions.

Parameters:

Lab: Pointer to a cmsCIELab value as described in Table 13
LCh: Pointer to a cmsCIELCh value as described in Table 14

Returns:

None

Encoding /Decoding on PCS

2.0

```
void cmsLabEncoded2Float(cmsCIELab* Lab, const cmsUInt16Number wLab[3]);
```

Decodes a Lab value, encoded on ICC v4 convention to *a cmsCIELab value as described in Table 13*

Parameters:

> *Lab: Pointer to a cmsCIELab value as described in Table 13*
> *wLab[] : Array of 3 cmsUInt16Number holding the encoded values.*

Returns:

> **None**

2.0

```
void cmsLabEncoded2FloatV2(cmsCIELab* Lab, const cmsUInt16Number wLab[3]);
```

Decodes a Lab value, encoded on ICC v2 convention to a cmsCIELab value *as described in Table 13*

Parameters:

> *Lab: Pointer to a cmsCIELab value as described in Table 13*
> *wLab[] : Array of 3 cmsUInt16Number holding the encoded values.*

Returns:

> **None**

2.0

```
void cmsFloat2LabEncoded(cmsUInt16Number wLab[3], const cmsCIELab* Lab);
```

Encodes a Lab value, *from a cmsCIELab value as described in Table 13*, to ICC v4 convention.

Parameters:

> *Lab: Pointer to a cmsCIELab value as described in Table 13*
> *wLab[] : Array of 3 cmsUInt16Number to hold the encoded values.*

Returns:

> **None**

2.0

```
void cmsFloat2LabEncodedV2(cmsUInt16Number wLab[3], const cmsCIELab* Lab);
```

Encodes a Lab value, *from a cmsCIELab value as described in Table 13*, to ICC v2 convention.

Parameters:

Lab: Pointer to a cmsCIELab value as described in Table 13
wLab[] : Array of 3 cmsUInt16Number to hold the encoded values.

Returns:

None

2.0

```
void cmsXYZEncoded2Float(cmsCIEXYZ* fxyz, const cmsUInt16Number XYZ[3]);
```

Decodes a XYZ value, encoded on ICC convention to a cmsCIEXYZ value *as described in* Table 11

Parameters:

fxyz: Pointer to a cmsCIEXYZ value as described in Table 11
XYZ[] : Array of 3 cmsUInt16Number holding the encoded values.

Returns:

None

2.0

```
void cmsFloat2XYZEncoded(cmsUInt16Number XYZ[3], const cmsCIEXYZ* fXYZ);
```

Encodes a XYZ value, *from a cmsCIELab value as described in* Table 11, to ICC convention.

Parameters:

XYZ[] : Array of 3 cmsUInt16Number to hold the encoded values.
fxyz: Pointer to a cmsCIEXYZ value as described in Table 11

Returns:

None

Accessing tags

Tag types

Those are the predefined tag types. You can add more types by using tag type plug-ins. See the plug-in API reference for further details.

Base type definitions (cmsTagTypeSignature)	
cmsSigChromaticityType	0x6368726D 'chrm'
cmsSigcicpType	0x63696370, 'cicp'
cmsSigColorantOrderType	0x636C726F 'clro'
cmsSigColorantTableType	0x636C7274 'clrt'
cmsSigCrdInfoType	0x63726469 'crdi'
cmsSigCurveType	0x63757276 'curv'
cmsSigDataType	0x64617461 'data'
cmsSigDateTimeType	0x6474696D 'dtim'
cmsSigDeviceSettingsType	0x64657673 'devs'
cmsSigLut16Type	0x6d667432 'mft2'
cmsSigLut8Type	0x6d667431 'mft1'
cmsSigLutAtoBType	0x6d414220 'mAB '
cmsSigLutBtoAType	0x6d424120 'mBA '
cmsSigMeasurementType	0x6D656173 'meas'
cmsSigMultiLocalizedUnicodeType	0x6D6C7563 'mluc'
cmsSigMultiProcessElementType	0x6D706574 'mpet'
cmsSigNamedColorType	0x6E636f6C 'ncol'
cmsSigNamedColor2Type	0x6E636C32 'ncl2'
cmsSigParametricCurveType	0x70617261 'para'
cmsSigProfileSequenceDescType	0x70736571 'pseq'
cmsSigProfileSequenceIdType	0x70736964 'psid'
cmsSigResponseCurveSet16Type	0x72637332 'rcs2'
cmsSigS15Fixed16ArrayType	0x73663332 'sf32'
cmsSigScreeningType	0x7363726E 'scrn'
cmsSigSignatureType	0x73696720 'sig '
cmsSigTextType	0x74657874 'text'
cmsSigTextDescriptionType	0x64657363 'desc'
cmsSigU16Fixed16ArrayType	0x75663332 'uf32'
cmsSigUcrBgType	0x62666420 'bfd '
cmsSigUInt16ArrayType	0x75693136 'ui16'
cmsSigUInt32ArrayType	0x75693332 'ui32'
cmsSigUInt64ArrayType	0x75693634 'ui64'
cmsSigUInt8ArrayType	0x75693038 'ui08'
cmsSigViewingConditionsType	0x76696577 'view'
cmsSigXYZType	0x58595A20 'XYZ '

Table 20

Tags

Those are the predefined tags. You can add more tags by using tag plug-ins. See the plug-in API reference for further details. On the right there is the lcms type representation for cmsReadTag and cmsWriteTag.

Base tag definitions (cmsTagSignature)		lcms type
cmsSigAToB0Tag	0x41324230 'A2B0'	cmsPipeline
cmsSigAToB1Tag	0x41324231 'A2B1'	cmsPipeline
cmsSigAToB2Tag	0x41324232 'A2B2'	cmsPipeline
cmsSigBlueColorantTag	0x6258595A 'bXYZ'	cmsCIEXYZ
cmsSigBlueMatrixColumnTag	0x6258595A 'bXYZ'	cmsCIEXYZ
cmsSigBlueTRCTag	0x62545243 'bTRC'	cmsToneCurve
cmsSigBToA0Tag	0x42324130 'B2A0'	cmsPipeline
cmsSigBToA1Tag	0x42324131 'B2A1'	cmsPipeline
cmsSigBToA2Tag	0x42324132 'B2A2'	cmsPipeline
cmsSigCalibrationDateTimeTag	0x63616C74 'calt'	struct tm
cmsSigCharTargetTag	0x74617267 'targ'	cmsMLU
cmsSigChromaticAdaptationTag	0x63686164 'chad'	cmsCIEXYZ [3]
cmsSigChromaticityTag	0x6368726D 'chrm'	cmsCIExyYTRIPLE
cmsSigColorantOrderTag	0x636C726F 'clro'	cmsUInt8Number [16]
cmsSigColorantTableTag	0x636C7274 'clrt'	cmsNAMEDCOLORLIST
cmsSigColorantTableOutTag	0x636C6F74 'clot'	cmsNAMEDCOLORLIST
cmsSigColorimetricIntentImageStateTag	0x63696973 'ciis'	cmsSignature
cmsSigCopyrightTag	0x63707274 'cprt'	cmsMLU
cmsSigCrdInfoTag*	0x63726469 'crdi'	cmsNAMEDCOLORLIST
cmsSigDataTag	0x64617461 'data'	cmsICCData
cmsSigDateTimeTag	0x6474696D 'dtim'	struct tm
cmsSigDeviceMfgDescTag	0x646D6E64 'dmnd'	cmsMLU
cmsSigDeviceModelDescTag	0x646D6464 'dmdd'	cmsMLU
cmsSigDeviceSettingsTag	0x64657673 'devs'	Not supported*
cmsSigDToB0Tag	0x44324230 'D2B0'	cmsPipeline
cmsSigDToB1Tag	0x44324231 'D2B1'	cmsPipeline
cmsSigDToB2Tag	0x44324232 'D2B2'	cmsPipeline
cmsSigDToB3Tag	0x44324233 'D2B3'	cmsPipeline
cmsSigBToD0Tag	0x42324430 'B2D0'	cmsPipeline
cmsSigBToD1Tag	0x42324431 'B2D1'	cmsPipeline
cmsSigBToD2Tag	0x42324432 'B2D2'	cmsPipeline
cmsSigBToD3Tag	0x42324433 'B2D3'	cmsPipeline
cmsSigGamutTag	0x67616D74 'gamt'	cmsPipeline
cmsSigGrayTRCTag	0x6b545243 'kTRC'	cmsToneCurve
cmsSigGreenColorantTag	0x6758595A 'gXYZ'	cmsCIEXYZ
cmsSigGreenMatrixColumnTag	0x6758595A 'gXYZ'	cmsCIEXYZ
cmsSigGreenTRCTag	0x67545243 'gTRC'	cmsToneCurve
cmsSigLuminanceTag	0x6C756d69 'lumi'	cmsCIEXYZ
cmsSigMeasurementTag	0x6D656173 'meas'	cmsICCMeasurementConditions
cmsSigMediaBlackPointTag	0x626B7074 'bkpt'	cmsCIEXYZ
cmsSigMediaWhitePointTag	0x77747074 'wtpt'	cmsCIEXYZ

cmsSigNamedColorTag	0x6E636f6C 'ncol'	Not supported*
cmsSigNamedColor2Tag	0x6E636C32 'ncl2'	cmsNAMEDCOLORLIST
cmsSigOutputResponseTag	0x72657370 'resp'	Not supported*
cmsSigPerceptualRenderingIntentGamutTag	0x72696730 'rig0'	cmsSignature
cmsSigPreview0Tag	0x70726530 'pre0'	cmsPipeline
cmsSigPreview1Tag	0x70726531 'pre1'	cmsPipeline
cmsSigPreview2Tag	0x70726532 'pre2'	cmsPipeline
cmsSigProfileDescriptionTag	0x64657363 'desc'	cmsMLU
cmsSigProfileSequenceDescTag	0x70736571 'pseq'	cmsSEQ
cmsSigProfileSequenceIdTag	0x70736964 'psid'	cmsSEQ
cmsSigPs2CRD0Tag	0x70736430 'psd0'	cmsICCData
cmsSigPs2CRD1Tag	0x70736431 'psd1'	cmsICCData
cmsSigPs2CRD2Tag	0x70736432 'psd2'	cmsICCData
cmsSigPs2CRD3Tag	0x70736433 'psd3'	cmsICCData
cmsSigPs2CSATag	0x70733273 'ps2s'	cmsICCData
cmsSigPs2RenderingIntentTag	0x70733269 'ps2i'	cmsICCData
cmsSigRedColorantTag	0x7258595A 'rXYZ'	cmsCIEXYZ
cmsSigRedMatrixColumnTag	0x7258595A 'rXYZ'	cmsCIEXYZ
cmsSigRedTRCTag	0x72545243 'rTRC'	cmsToneCurve
cmsSigSaturationRenderingIntentGamutTag	0x72696732 'rig2'	cmsSignature
cmsSigScreeningDescTag	0x73637264 'scrd'	cmsMLU
cmsSigScreeningTag	0x7363726E 'scrn'	cmsScreening
cmsSigTechnologyTag	0x74656368 'tech'	cmsSignature
cmsSigUcrBgTag	0x62666420 'bfd '	cmsUcrBg
cmsSigViewingCondDescTag	0x76756564 'vued'	cmsMLU
cmsSigViewingConditionsTag	0x76696577 'view'	cmsICCViewingConditions
cmsSigMetaTag	0x6D657461 'meta'	cmsHANDLE (DICT)
cmsSigcicpTag	0x63696370 'cicp'	cmsVideoSignalType
cmsSigArgyllArtsTag	0x61727473 'arts'	cmsS15Fixed16Number Array [9]
cmsSigMHC2Tag	0x4D484332 'MHC2'	cmsMHC2Type

Table 21

***cmsSigCrdInfoTag**: This type contains the PostScript product name to which this profile corresponds and the names of the companion CRDs. A single profile can generate multiple CRDs. It is implemented as a MLU being the language code "PS" and then country varies for each element:

- nm: PostScript product name
- #0: Rendering intent 0 CRD name
- #1: Rendering intent 1 CRD name
- #2: Rendering intent 2 CRD name
- #3: Rendering intent 3 CRD name

There are several tags not supported, they are listed below with an explanation on why are not supported.

Not supported	**Why**
cmsSigOutputResponseTag	*POSSIBLE PATENT ON THIS SUBJECT!*
cmsSigNamedColorTag	*Deprecated*
cmsSigDataTag	*Ancient, unused*
cmsSigDeviceSettingsTag	*Deprecated, useless*

2.0

```
cmsInt32Number   cmsGetTagCount(cmsHPROFILE hProfile);
```

Returns the number of tags present in a given profile.

Parameters:

hProfile: Handle to a profile object

Returns:

Number of tags on success, -1 on error.

2.0

```
cmsTagSignature  cmsGetTagSignature(cmsHPROFILE hProfile,
                                                   cmsUInt32Number n);
```

Returns the signature of a tag located in n position being n a 0-based index: i.e., first tag is indexed with n=0.

Parameters:

hProfile: Handle to a profile object
n: index to a tag position (0-based)

Returns:

The tag signature on success, 0 on error.

2.0

```
cmsBool cmsIsTag(cmsHPROFILE hProfile, cmsTagSignature sig);
```

Returns TRUE if a tag with signature sig is found on the profile. Useful to check if a profile contains a given tag.

Parameters:

hProfile: Handle to a profile object.
sig: Tag signature, as stated in Table 21

Returns:

TRUE if the tag is found, FALSE otherwise.

2.0

```
void*  cmsReadTag(cmsHPROFILE hProfile,  cmsTagSignature sig);
```

Reads an existing tag with signature sig, parses it and returns a pointer to an object owned by the profile object holding a representation of tag contents.

Little CMS will return (if found) a pointer to a structure holding the tag. The obtained structure is not the raw contents of the tag, but the result of *parsing* the tag. For example, reading a cmsSigAToB0 tag results as a Pipeline structure ready to be used by all the cmsPipeline functions. The memory belongs to the profile and is set free on closing the profile. In this way, there are no memory duplicates and you can safely re-use the same tag as many times as you wish. Anything coming from cmsReadTag should be treated as const. Otherwise you are modifying structures that are owned by the profile, when the profile is set free, it tries to free those structures. If you have modified the internal pointers, it can get corrupted.

Parameters:

hProfile: Handle to a profile object.
sig: Tag signature, as stated in Table 21

Returns:

A pointer to a profile-owned object holding tag contents, or NULL if the signature is not found. Type of object does vary. See Table 21 *for a list of returned types.*

2.0

```
cmsBool  cmsWriteTag(cmsHPROFILE hProfile,
                                  cmsTagSignature sig,
                                  const void* data);
```

Writes an object to an ICC profile tag, doing all necessary serialization. The obtained tag depends on ICC version number used when creating the profile.

Writing tags is almost the same as read them, you just specify a pointer to the structure and the tag name and *Little CMS* will do all serialization for you. Process under the hood may be very complex, if you realize v2 and v4 of the ICC spec are using different representations of same structures.

Parameters:

hProfile: Handle to a profile object
sig: Tag signature, as stated in Table 21
data: A pointer to an object holding tag contents. Type of object does vary. See Table 21 *for a list of required types.*

Returns:

TRUE on success, FALSE on error

2.0

```
cmsBool cmsLinkTag(cmsHPROFILE hProfile,
                               cmsTagSignature sig,
                               cmsTagSignature dest);
```

Creates a directory entry on tag *sig* that points to same location as tag *dest.* Using this function you can collapse several tag entries to the same block in the profile. For example, point perceptual, rel.col and saturation BtoAxx tags to same implementation.

Parameters:

hProfile: Handle to a profile object
sig: Signature of linking tag
dest: Signature of linked tag

Returns:

TRUE on success, FALSE on error

2.1

```
cmsTagSignature   cmsTagLinkedTo(cmsHPROFILE hProfile,  cmsTagSignature sig);
```

Returns the tag linked to sig, in the case two tags are sharing same resource, or NULL if the tag is not linked to any other tag.

Parameters:

hProfile: Handle to a profile object
sig: Signature of linking tag

Returns:

Signature of linked tag, or NULL if no tag is linked

Accessing tags as raw data

Those functions allows to read/write directly to the ICC profile any data, without checking anything. As a rule, mixing Raw with cooked doesn't work, so writting a tag as raw and then reading it as cooked without serializing does result into an error. If that is what you want, you will need to dump the profile to memory or disk and then reopen it.

2.0

```
cmsInt32Number    cmsReadRawTag(cmsHPROFILE hProfile,
                                          cmsTagSignature sig,
                                          void* Buffer, cmsUInt32Number BufferSize);
```

Similar to cmsReadTag, but different in two important aspects. 1st, the memory is not owned by the profile, but by you, so you have to allocate the necessary amount of memory. To know the size in advance, use NULL as buffer and 0 as buffer size. The function then returns the number of needed bytes without writing them.

The second important point is, this is raw data. No processing is performed, so you can effectively read wrong or broken profiles with this function. Obviously, it is up to you to interpret all those bytes!

Parameters:

hProfile: Handle to a profile object
sig: Signature of tag to be read
Buffer: Points to a memory block to hold the result.
BufferSize: Size of memory buffer in bytes

Returns:

Number of bytes read.

2.0

```
cmsBool cmsWriteRawTag(cmsHPROFILE hProfile,
                                        cmsTagSignature sig,
                                        const void* data, cmsUInt32Number Size);
```

The RAW version does the same as cmsWriteTag but without any interpretation of the data. Please note it is fair easy to deal with "cooked" structures, since there are primitives for allocating, deleting and modifying data. For RAW data you are responsible of everything. If you want to deal with a private tag, you may want to write a plug-in instead of messing up with raw data.

Parameters:

hProfile: Handle to a profile object
sig: Signature of tag to be written
Buffer: Points to a memory block holding the data.
BufferSize: Size of data in bytes

Returns:

TRUE on success, FALSE on error

Profile structures

ICC profile internal base types. Strictly, shouldn't be declared in this header, but maybe somebody wants to use this info for accessing profile header directly, so here it is. Data is 32-bit aligned.

cmsICCHeader		
cmsUInt32Number	size;	Profile size in bytes
cmsSignature	cmmId;	CMM for this profile
cmsUInt32Number	version;	Format version number
cmsProfileClassSignature	deviceClass;	Type of profile
cmsColorSpaceSignature	colorSpace;	Color space of data
cmsColorSpaceSignature	pcs;	PCS, XYZ or Lab only
cmsDateTimeNumber	date;	Date it was created
cmsSignature	magic;	Identify ICC profile
cmsPlatformSignature	platform;	Primary Platform
cmsUInt32Number	flags;	Various bit settings
cmsSignature	manufacturer;	Device manufacturer
cmsUInt32Number	model;	Device model number
cmsUInt64Number	attributes;	Device attributes
cmsUInt32Number	renderingIntent;	Rendering intent
cmsEncodedXYZNumber	illuminant;	Profile illuminant
cmsSignature	creator;	Profile creator
cmsProfileID	profileID;	Profile ID using MD5
cmsInt8Number	reserved[28];	Reserved

Table 22

cmsICCData	
cmsUInt32Number	len;
cmsUInt32Number	flag;
cmsUInt8Number	data[1];

Table 23

cmsDateTimeNumber (ICC date time)	
cmsUInt16Number	year;
cmsUInt16Number	month;
cmsUInt16Number	day;
cmsUInt16Number	hours;
cmsUInt16Number	minutes;
cmsUInt16Number	seconds;

Table 24

cmsEncodedXYZNumber (ICC XYZ)	
cmsS15Fixed16Number	X;
cmsS15Fixed16Number	Y;
cmsS15Fixed16Number	Z;

Table 25

cmsICCMeasurementConditions		
cmsUInt32Number	Observer;	0 = unknown, 1=CIE 1931, 2=CIE 1964
cmsCIEXYZ	Backing;	Value of backing
cmsUInt32Number	Geometry;	0=unknown, 1=45/0, 0/45 2=0d, d/0
cmsFloat64Number	Flare;	0..1.0
cmsUInt32Number	IlluminantType;	

Table 26

cmsICCViewingConditions		
cmsCIEXYZ	IlluminantXYZ;	Not the same struct as CAM02,
cmsCIEXYZ	SurroundXYZ;	This is for storing the tag
cmsUInt32Number	IlluminantType;	viewing condition

Table 27

Platforms

Platforms(cmsPlatformSignature)	
cmsSigMacintosh	0x4150504C 'APPL'
cmsSigMicrosoft	0x4D534654 'MSFT'
cmsSigSolaris	0x53554E57 'SUNW'
cmsSigSGI	0x53474920 'SGI '
cmsSigTaligent	0x54474E54 'TGNT'
cmsSigUnices	0x2A6E6978 '*nix'

Table 28

Reference gamut

cmsSigPerceptualReferenceMediumGamut	0x70726d67 'prmg'

Table 29

Image State

For cmsSigColorimetricIntentImageStateTag	
cmsSigSceneColorimetryEstimates	0x73636F65 'scoe'
cmsSigSceneAppearanceEstimates	0x73617065 'sape'
cmsSigFocalPlaneColorimetryEstimates	0x66706365 'fpce'
cmsSigReflectionHardcopyOriginalColorimetry	0x72686F63 'rhoc'
cmsSigReflectionPrintOutputColorimetry	0x72706F63 'rpoc'

Table 30

Pipeline Stages (Multi processing elements)

Stage types (cmsStageSignature)	
cmsSigCurveSetElemType	0x63767374 'cvst'
cmsSigMatrixElemType	0x6D617466 'matf'
cmsSigCLutElemType	0x636C7574 'clut'
cmsSigBAcsElemType	0x62414353 'bACS'
cmsSigEAcsElemType	0x65414353 'eACS'
Private extensions	
cmsSigXYZ2LabElemType	0x6C327820 'l2x '
cmsSigLab2XYZElemType	0x78326C20 'x2l '
cmsSigNamedColorElemType	0x6E636C20 'ncl '
cmsSigLabV2toV4	0x32203420 '2 4 '
cmsSigLabV4toV2	0x34203220 '4 2 '
cmsSigIdentityElemType	0x69646E20 'idn '

Table 31

Types of CurveElements	
cmsSigFormulaCurveSeg	0x70617266 'parf'
cmsSigSampledCurveSeg	0x73616D66 'samf'
cmsSigSegmentedCurve	0x63757266 'curf'

Table 32

Used in ResponseCurveType		
cmsSigStatusA	0x53746141 'StaA'	Status A: ISO 5-3 densitometer response. This is the accepted standard for reflection densitometers for measuring photographic colour prints.
cmsSigStatusE	0x53746145 'StaE'	Status E: ISO 5-3 densitometer response which is the accepted standard in Europe for colour reflection densitometers.
cmsSigStatusI	0x53746149 'StaI'	Status I: ISO 5-3 densitometer response commonly referred to as narrow band or interference-type response.
cmsSigStatusT	0x53746154 'StaT'	Status T: ISO 5-3 wide band colour reflection densitometer response which is the accepted standard in the United States for colour reflection densitometers.
cmsSigStatusM	0x5374614D 'StaM'	Status M: ISO 5-3 densitometer response for measuring colour negatives.
cmsSigDN	0x444E2020 'DN '	DIN E: DIN 16536-2 densitometer response, with no polarising filter.
cmsSigDNP	0x444E2050 'DN P'	DIN E: DIN 16536-2 densitometer response, with polarising filter.
cmsSigDNN	0x444E4E20 'DNN '	DIN I: DIN 16536-2 narrow band densitometer response, with no polarising filter.
cmsSigDNNP	0x444E4E50 'DNNP'	DIN I: DIN 16536-2 narrow band densitometer response, with polarising filter.

Table 33

Technology enumeration for cmsTechnologySignature, used on header and profile sequence description.

Technology tag (cmsTechnologySignature)	
cmsSigDigitalCamera	0x6463616D 'dcam'
cmsSigFilmScanner	0x6673636E 'fscn'
cmsSigReflectiveScanner	0x7273636E 'rscn'
cmsSigInkJetPrinter	0x696A6574 'ijet'
cmsSigThermalWaxPrinter	0x74776178 'twax'
cmsSigElectrophotographicPrinter	0x6570686F 'epho'
cmsSigElectrostaticPrinter	0x65737461 'esta'
cmsSigDyeSublimationPrinter	0x64737562 'dsub'
cmsSigPhotographicPaperPrinter	0x7270686F 'rpho'
cmsSigFilmWriter	0x6670726E 'fprn'
cmsSigVideoMonitor	0x7669646D 'vidm'
cmsSigVideoCamera	0x76696463 'vidc'
cmsSigProjectionTelevision	0x706A7476 'pjtv'
cmsSigCRTDisplay	0x43525420 'CRT '
cmsSigPMDisplay	0x504D4420 'PMD '
cmsSigAMDisplay	0x414D4420 'AMD '
cmsSigPhotoCD	0x4B504344 'KPCD'
cmsSigPhotoImageSetter	0x696D6773 'imgs'
cmsSigGravure	0x67726176 'grav'
cmsSigOffsetLithography	0x6F666673 'offs'
cmsSigSilkscreen	0x73696C6B 'silk'
cmsSigFlexography	0x666C6578 'flex'
cmsSigMotionPictureFilmScanner	0x6D706673 'mpfs'
cmsSigMotionPictureFilmRecorder	0x6D706672 'mpfr'
cmsSigDigitalMotionPictureCamera	0x646D7063 'dmpc'
cmsSigDigitalCinemaProjector	0x64636A70 'dcpj'

Table 34

Formatters

Formatters are used to describe how bitmap buffers are organized. Format of pixel is defined by a cmsUInt32Number, using bit fields as follows:

M A O TTTTT U Y F P X S EEE CCCC BBB

M	Premultiplied alpha (only works when extra samples is 1)
A	Floating point. With this flag we can differentiate, for example, 16 bits as float or as int
O	Optimized previous optimization already returns the final 8-bit value (*internal use only*)
T	Pixeltype (see table below)
F	Flavor 0=MinIsBlack(Chocolate) 1=MinIsWhite(Vanilla)
P	Planar? 0=Chunky, 1=Planar
X	swap 16 bps endianess?
S	Do swap? ie, BGR, KYMC
E	Extra samples
C	Channels (Samples per pixel)
B	Bytes per sample
Y	Swap first channel - changes ABGR to BGRA and KCMY to CMYK

Table 35

Macros to build formatters

```
#define PREMUL_SH(m)        ((m) << 23)
#define FLOAT_SH(a)         ((a) << 22)
#define OPTIMIZED_SH(s)     ((s) << 21)
#define COLORSPACE_SH(s)    ((s) << 16)
#define SWAPFIRST_SH(s)     ((s) << 14)
#define FLAVOR_SH(s)        ((s) << 13)
#define PLANAR_SH(p)        ((p) << 12)
#define ENDIAN16_SH(e)      ((e) << 11)
#define DOSWAP_SH(e)        ((e) << 10)
#define EXTRA_SH(e)         ((e) << 7)
#define CHANNELS_SH(c)      ((c) << 3)
#define BYTES_SH(b)         (b)
```

Macros to extract information from formatters

```
#define T_PREMUL(m)         (((m)>>23)&1)
#define T_FLOAT(a)          (((a)>>22)&1)
#define T_OPTIMIZED(o)      (((o)>>21)&1)
#define T_COLORSPACE(s)     (((s)>>16)&31)
#define T_SWAPFIRST(s)      (((s)>>14)&1)
#define T_FLAVOR(s)         (((s)>>13)&1)
#define T_PLANAR(p)         (((p)>>12)&1)
#define T_ENDIAN16(e)       (((e)>>11)&1)
#define T_DOSWAP(e)         (((e)>>10)&1)
#define T_EXTRA(e)          (((e)>>7)&7)
#define T_CHANNELS(c)       (((c)>>3)&15)
#define T_BYTES(b)          ((b)&7)
```

Color spaces in Little CMS notation

Used in formatters as a double check of color space. Each color space has an implicit range.

Pixel types		
PT_ANY	0	Don't check colorspace
reserved	1	Reseved for future ampliations (bilevel)
reserved	2	Reserved for future ampliations (palette)
PT_GRAY	3	Gray scale
PT_RGB	4	Red Green Blue
PT_CMY	5	Cyan Magenta Yellow
PT_CMYK	6	Cyan Magenta Yellow blacK
PT_YCbCr	7	Y Cb Cr
PT_YUV	8	L u'v'
PT_XYZ	9	CIE XYZ
PT_Lab	10	CIE L*a*b
PT_YUVK	11	L u'v' K
PT_HSV	12	H S V
PT_HLS	13	H L S
PT_Yxy	14	Y x y
PT_MCH1	15	1 unspecified channel(s)
PT_MCH2	16	2 unspecified channel(s)
PT_MCH3	17	3 unspecified channel(s)
PT_MCH4	18	4 unspecified channel(s)
PT_MCH5	19	5 unspecified channel(s)
PT_MCH6	20	6 unspecified channel(s)
PT_MCH7	21	7 unspecified channel(s)
PT_MCH8	22	8 unspecified channel(s)
PT_MCH9	23	9 unspecified channel(s)
PT_MCH10	24	10 unspecified channel(s)
PT_MCH11	25	11 unspecified channel(s)
PT_MCH12	26	12 unspecified channel(s)
PT_MCH13	27	13 unspecified channel(s)
PT_MCH14	28	14 unspecified channel(s)
PT_MCH15	29	15 unspecified channel(s)
PT_LabV2	30	Identical to PT_Lab, but using the V2 old encoding

Table 36

Translate color space from/to Little CMS notation to ICC

With those functions you can convert ICC color space enumeration (*cmsColorSpaceSignature, Table 10*) to the *Little CMS* integers used in formatters *(Table 36)* .

2.0

```
cmsColorSpaceSignature  _cmsICCcolorSpace(int OurNotation);
```

Converts from *Little CMS* color space notation *(Table 36)* to ICC color space notation (*Table 10*).

Parameters:

OurNotation: any value from Table 36

Returns:

Corresponding cmsColorSpaceSignature (Table 10) or -1 on error.

2.0

```
int  _cmsLCMScolorSpace(cmsColorSpaceSignature ProfileSpace);
```

Converts from ICC color space notation (*Table 10*) to *Little CMS* color space notation *(Table 36)*.

Parameters:

ProfileSpace: any cmsColorSpaceSignature from Table 10

Returns:

Corresponding Little CMS value (Table 36) or -1 on error.

Predefined formatters

Already defined in ***lcms2.h***, each formatter identifies a particular buffer organization. More formatters can be defined by using the macros listed in lcms2.h.

The macro names are built by using following convention:

- "TYPE_" literal
- Color Space
- A if alpha
- Bits per component
- Modifiers (one or more)
    - PLANAR,
    - SE (swap endian),
    - REV (reverse)
    - PREMUL (premultiplied alpha)
    - HALF_FLT for 16-bit floats
    - FLT for 32-bit floats
    - DBL for 64-bit doubles

Examples

TYPE_GRAY_8	Grayscale 8 bits
TYPE_GRAY_8_REV	Grayscale 8 bits, reversed
TYPE_GRAY_16	Grayscale 16 bits
TYPE_GRAY_16_REV	Grayscale 16 bits, reversed
TYPE_GRAY_16_SE	Grayscale 16 bits, swapped-endian
TYPE_GRAYA_8	Grayscale + alpha 8 bits
TYPE_GRAYA_8_PREMUL	Grayscale with premultiplied alpha
TYPE_GRAYA_16	Grayscale + alpha 16 bits
TYPE_GRAYA_16_SE	Grayscale + alpha 16 bits, swapped endian
TYPE_GRAYA_8_PLANAR	Grayscale 8 bits, single plane
TYPE_GRAYA_16_PLANAR	Grayscale 16 bits, single plane
TYPE_RGB_8	RGB 8 bits
TYPE_RGB_8_PLANAR	RGB 8 bits, stored as contiguous planes
TYPE_BGR_8	BGR 8 bits
TYPE_BGR_8_PLANAR	BGR 8 bits, stored as contiguous planes
TYPE_RGB_16	BGR 16 bits
TYPE_RGB_16_PLANAR	RGB 16 bits, stored as contiguous planes
TYPE_RGB_16_SE	RGB 16 bits, swapped endian
TYPE_BGR_16	BGR 16 bits
TYPE_BGR_16_PLANAR	BGR 16 bits, stored as contiguous planes
TYPE_BGR_16_SE	BGR 16 bits, with swapped endinaness
TYPE_RGBA_8	RGB 8 bits plus an Alpha channel
TYPE_RGBA_8_PLANAR	RGBA 8 bits, stored as contiguous planes
TYPE_RGBA_16	RGB 16 bits plus an Alpha channel
TYPE_RGBA_16_PLANAR	RGBA 16 bits, stored as contiguous planes
TYPE_RGBA_16_SE	RGBA 16 bits, with swapped endinaness

TYPE_ARGB_8	An alpha channel plus RGB in 8 bits
TYPE_ARGB_8_PLANAR	
TYPE_ARGB_16	An alpha channel plus RGB in 16 bits
TYPE_ABGR_8	An alpha channel plus BGR in 8 bits
TYPE_ABGR_8_PLANAR	An alpha channel plus BGR in seprate 8 bit planes
TYPE_ABGR_16	An alpha channel plus BGR in 16 bits
TYPE_ABGR_16_PLANAR	An alpha channel plus BGR in separate 16 bit planes
TYPE_ABGR_16_SE	
TYPE_BGRA_8	
TYPE_BGRA_8_PLANAR	
TYPE_BGRA_16	
TYPE_BGRA_16_SE	
TYPE_CMY_8	
TYPE_CMY_8_PLANAR	
TYPE_CMY_16	
TYPE_CMY_16_PLANAR	
TYPE_CMY_16_SE	
TYPE_CMYK_8	
TYPE_CMYKA_8	
TYPE_CMYK_8_REV	
TYPE_YUVK_8	
TYPE_CMYK_8_PLANAR	
TYPE_CMYK_16	
TYPE_CMYK_16_REV	
TYPE_YUVK_16	
TYPE_CMYK_16_PLANAR	
TYPE_CMYK_16_SE	
TYPE_KYMC_8	
TYPE_KYMC_16	
TYPE_KYMC_16_SE	
TYPE_KCMY_8	
TYPE_KCMY_8_REV	
TYPE_KCMY_16	
TYPE_KCMY_16_REV	
TYPE_KCMY_16_SE	
TYPE_CMYK5_8	
TYPE_CMYK5_16	
TYPE_CMYK5_16_SE	
TYPE_KYMC5_8	
TYPE_KYMC5_16	
TYPE_KYMC5_16_SE	
TYPE_CMYKcm_8	
TYPE_CMYKcm_8_PLANAR	
TYPE_CMYKcm_16	
TYPE_CMYKcm_16_PLANAR	
TYPE_CMYKcm_16_SE	
TYPE_CMYK7_8	
TYPE_CMYK7_16	

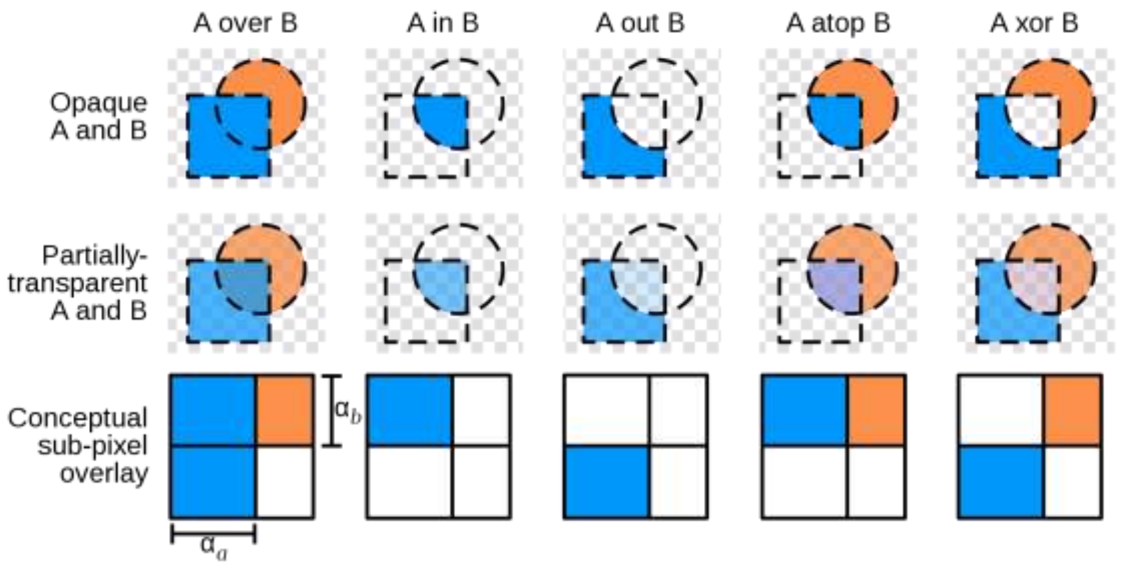
TYPE_CMYK7_16_SE	
TYPE_KYMC7_8	
TYPE_KYMC7_16	
TYPE_KYMC7_16_SE	
TYPE_CMYK8_8	
TYPE_CMYK8_16	
TYPE_CMYK8_16_SE	
TYPE_KYMC8_8	
TYPE_KYMC8_16	
TYPE_KYMC8_16_SE	
TYPE_CMYK9_8	
TYPE_CMYK9_16	
TYPE_CMYK9_16_SE	
TYPE_KYMC9_8	
TYPE_KYMC9_16	
TYPE_KYMC9_16_SE	
TYPE_CMYK10_8	
TYPE_CMYK10_16	
TYPE_CMYK10_16_SE	
TYPE_KYMC10_8	
TYPE_KYMC10_16	
TYPE_KYMC10_16_SE	
TYPE_CMYK11_8	
TYPE_CMYK11_16	
TYPE_CMYK11_16_SE	
TYPE_KYMC11_8	
TYPE_KYMC11_16	
TYPE_KYMC11_16_SE	
TYPE_CMYK12_8	
TYPE_CMYK12_16	
TYPE_CMYK12_16_SE	
TYPE_KYMC12_8	
TYPE_KYMC12_16	
TYPE_KYMC12_16_SE	
TYPE_XYZ_16	
TYPE_Lab_8	
TYPE_ALab_8	
TYPE_Lab_16	
TYPE_Yxy_16	
TYPE_YCbCr_8	
TYPE_YCbCr_8_PLANAR	
TYPE_YCbCr_16	
TYPE_YCbCr_16_PLANAR	
TYPE_YCbCr_16_SE	
TYPE_YUV_8	
TYPE_YUV_8_PLANAR	
TYPE_YUV_16	
TYPE_YUV_16_PLANAR	
TYPE_YUV_16_SE	

TYPE_HLS_8	
TYPE_HLS_8_PLANAR	
TYPE_HLS_16	
TYPE_HLS_16_PLANAR	
TYPE_HLS_16_SE	
TYPE_HSV_8	
TYPE_HSV_8_PLANAR	
TYPE_HSV_16	
TYPE_HSV_16_PLANAR	
TYPE_HSV_16_SE	
Floating point	
TYPE_XYZ_FLT	
TYPE_Lab_FLT	
TYPE_GRAY_FLT	
TYPE_RGB_FLT	
TYPE_CMYK_FLT	
TYPE_XYZ_DBL	
TYPE_Lab_DBL	
TYPE_GRAY_DBL	
TYPE_RGB_DBL	
TYPE_CMYK_DBL	
TYPE_LabV2_8	
TYPE_ALabV2_8	
TYPE_LabV2_16	
TYPE_OKLAB_DBL	
Takes directly the floating-point structs	
TYPE_XYZ_FLT	
TYPE_Lab_FLT	
TYPE_GRAY_FLT	
TYPE_RGB_FLT	
TYPE_CMYK_FLT	
TYPE_XYZ_DBL	
TYPE_Lab_DBL	
TYPE_GRAY_DBL	
TYPE_RGB_DBL	
TYPE_CMYK_DBL	
TYPE_XYZA_FLT	
TYPE_LabA_FLT	
TYPE_RGBA_FLT	

Table 37

Alpha channel

The alpha channel is a color component that represents the degree of transparency (or opacity) of a color. It is used to determine how a pixel is rendered when blended with another. when an image is overlaid onto another image, the alpha value of the source color is used to determine the resulting color. If the alpha value is opaque, the source color overwrites the destination color; if transparent, the source color is invisible, allowing the background color to show through. If the value is in between, the resulting color has a varying degree of transparency/opacity, which creates a translucent effect.



There are two common representations that are available: unassociated alpha and premultiplied alpha. With unassociated alpha, the RGB components represent the color of the object or pixel, disregarding its opacity. With premultiplied alpha, the RGB components represent the emission of the object or pixel, and the alpha represents the occlusion.

An obvious advantage of this is that, in certain situations, it can save a multiplication However, the most significant advantages of using premultiplied alpha are for correctness and simplicity rather than performance: premultiplied alpha allows correct filtering and blending. In addition, premultiplied alpha allows regions of regular alpha blending and regions with additive blending mode to be encoded within the same image. Fires and flames, glows, flares, and other such phenomena can only be represented using premultiplied alpha.

Premultiplication can reduce the available relative precision in the RGB values when using integer or fixed-point representation for the color components, which may cause a noticeable loss of quality if the color information is later brightened or if the alpha channel is removed. In practice, this is not usually noticeable because during typical composition operations, the influence of the low-precision color information in low-alpha areas on the final output image is correspondingly reduced. This loss of precision also makes premultiplied images easier to compress using certain compression schemes, as they do not record the color variations hidden inside transparent regions, and can allocate fewer bits to encode low-alpha areas.

Little CMS can deal with both, unassociated and premultiplied alpha. When you create a color transform across formats holding alpha channels, the color engine by default does just nothing. It skips the alpha channels so you are free to initialize the result with opaque or transparent alpha. This is also to keep compatibility with old versions of Little CMS.

Otherwise, you may want to just copy the alpha channel assuming it will be faster to perform color management and alpha handling at same time. There is flag that you have to specify if wish so: **cmsFLAGS_COPY_ALPHA**.

Please note there are situations when this copy of the alpha channel is not trivial, take for example a color transform from 8 bits TYPE_RGBA_8 to a double floating point TYPE_ABGR_DBL. The color engine will copy the alpha channel but also will convert from the 0..255 range to 0..1.0 of double format. Another example is on images with ***more*** than one alpha channel.

Premultiplied alpha may also be handled by the color engine. For input formats an optimized fixed-point math is applied to recover the components. Color management is then applied on the obtained pixels. For output, it is necessary to use **cmsFLAGS_COPY_ALPHA** flag. Output premultiplied alpha needs input alpha, otherwise no operation is performed and output alpha channel is ignored.

See lcms2.h for predefined formatters handling premultiplied alpha.

The ICC PCS

In order to understand how the relative colorimetric intent works, we should first understand one of the key concepts of ICC color management. That is the profile connection space, known as the PCS.

The PCS is just a convention. An abstraction on how profiles exchange information. A lingua franca used by the profiles to talk with each other. Your images are not likely to be converted to this color space. In fact, in most CMMs, the PCS is not directly accessible. In this way, an average user can effectively ignore the PCS, as it should not make any difference. But for a profile or CMM implementer, understanding how the PCS is defined and works is very important. I've seen many times profiles operating wrongly just because the creator misunderstood some basics of the PCS.

In general, an ICC profile implements conversions between the device it is intended for and the PCS. This may include both directions, from the device to the PCS and from the PCS to the device. The color space of the device may be RGB or CMYK depending on the actual device, while the PCS operates on device-independent colorimetric spaces.

ICC profiles may be used to model many different types of devices, but all share the same connection space. The PCS is based on XYZ or Lab, determined for a specific observer (the CIE standard 1931 colorimetric observer), relative to a specific illuminant (D50) and measured with a specific geometry (0º/45º or 45º/0º) for reflecting media. That is what the ICC spec says. But there is some subtlety in those words.

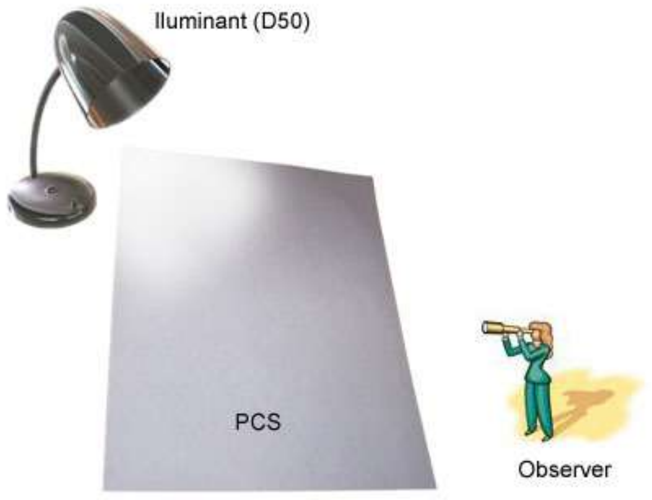
Why 1931 standard observer? This experimentally derived standard observer provides a very good representation of the human visual system color matching capabilities. Unlike device dependent color spaces, if two colors have the same CIE colorimetry they will match if viewed under the conditions for which the CIE colorimetry was defined. Because the imagery is typically produced for a wide variety of viewing environments, it is necessary to go beyond simple application of the CIE system.

The PCS definition has evolved with revisions of the ICC specification. The latest revision, 4.3, contains a very accurate description of how to interpret the PCS. Previous specifications were not so clear, and unfortunately that was a source of misunderstanding. Let's review the ICC v2 PCS, as a starting point.

ICC v2 PCS

The original v2 PCS was modeled as an **IDEAL REFLECTION PRINT**. As such, it had a specific media white and received light from a given illuminant.

Here is a representation of the v2 PCS



We have an observer, an illuminant and a reflection print. The exact meaning of those elements depends of the device being modeled. It is quite intuitive in a printer profile that the reflection print is just the media being printed. But some other cases are not so clear.

For the v2 PCS the ideal reflection print medium was a perfect diffuser, so the task of the profile was among other things, to map from the paper used in real world to that perfect media.

The refection print modeled by the PCS had a theoretical infinite dynamic range and gamut. So, the task of the profile was to map whatever color was represented in the PCS, no matter how hyper-saturated or self-fluorescent, to the real media. That is a rather difficult task if one wants to preserve things like fidelity, gradients, smoothness and detail.

ICC v4 PCS

The latest ICC v4 specification keeps those concepts, although it expands the PCS definition, and makes this virtual refection print more proper to real world media. While the PCS for the perceptual rendering intent is still assumed to be that of a reference reflection print, the dynamic range is limited and clearly defined, and the PCS for the colorimetric rendering intents is no longer assumed to be the colorimetry based on any specific media, but simply the colorimetry of the media as measured.

In theory, the dynamic range of the PCS for colorimetric transforms is infinite. In perceptual, dynamic range is limited both to perceptual black, as v4 perceptual PCS has a nonzero black point. Last revisions of the spec will include a reference medium gamut, which is going to greatly improve the overall quality of profiles. That is because in v4, a perceptual intent should map from source device gamut to the reference medium gamut and then back to the gamut of destination device. Using a realistic, well-specified gamut in the PCS prevents huge movements on re-rendering.

Another important difference between perceptual and relative colorimetric colorimetry in the PCS is, for the perceptual intent, standard PCS viewing conditions are specified. That is because the perceptual intent transform tries to produce colorimetry for a pleasing appearance on the PCS reference print, while relative colorimetric intent transform characterizes the actual device/medium colors in the PCS. The actual viewing conditions used should be provided in the profile viewing conditions tag, but for relative colorimetric, only media white point scaling and chromatic adaptation are performed. In transforms for the colorimetric intents, the range of valid PCS values is unrelated to the reference media white and black points.

It may seem that perceptual and relative colorimetric are using two different PCS, but that’s not really true. The PCS is the same, but with the perceptual intent the PCS colorimetry must be optimally color rendered for the reference medium. In relative colorimetric the PCS colorimetry can be used for any media, so long as the measured data is correctly represented.

Media normalization: demystifying the so called “Wrong von Kries”

The media being modeled may exhibit some chromaticity. That is, for example, in printer profiles, not all papers are equally white.

When reproducing colors, it may be very desirable to “discount” the chromaticity of the media. We want white mapped to white. Else, we would face problems like the “scum dot”, that is, small dispersed dots in the reproduction due to printer halftone.

So, there is a “cooking” to be applied to our data before obtaining the final PCS colorimetry: the media normalization. Its main purpose is to maximize dynamic range and discount media chromaticity. This is called media-relative colorimetry.

Here is the involved math. You got the media white measured in XYZ as (Xw, Yw, Zw), you got your measurements as XYZ, so for obtaining the PCS values:

$$\begin{bmatrix} Xpcs \\ Ypcs \\ Zpcs \end{bmatrix} = \begin{bmatrix} X_{D50}/Xw & 0 & 0 \\ 0 & Y_{D50}/Yw & 0 \\ 0 & 0 & Z_{D50}/Zw \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

In other words, the discounting of media chromaticity is performed using XYZ scaling.

Those trained in traditional colorimetry usually have a negative reaction when they hear of XYZ scaling used to perform adaptation. We should note that the goal of media normalization is just match whites and provide more dynamic range. A secondary goal is to be simple enough to fully recover absolute colorimetry. And this is very easy, indeed. Just use this simple formula in the reverse direction and you get back the original values. So this transform is easily invertible.

Effective media normalization will happen only on printer profiles, as self-luminous devices are going to use their white point as both the assumed adapted white point and the media white point. No normalization is needed in such case as the media white is already D50.

Other tables

Illuminant types	
cmsILLUMINANT_TYPE_UNKNOWN	0x0000000
cmsILLUMINANT_TYPE_D50	0x0000001
cmsILLUMINANT_TYPE_D65	0x0000002
cmsILLUMINANT_TYPE_D93	0x0000003
cmsILLUMINANT_TYPE_F2	0x0000004
cmsILLUMINANT_TYPE_D55	0x0000005
cmsILLUMINANT_TYPE_A	0x0000006
cmsILLUMINANT_TYPE_E	0x0000007
cmsILLUMINANT_TYPE_F8	0x0000008

Table 38

cmsUcrBg	
cmsToneCurve*	Ucr;
cmsToneCurve*	Bg;
cmsMLU*	Desc;

Table 39

Microsoft MHC2 tag

This is a private tag used by Microsoft to describe GPU hardware pipelines for displays. Little CMS supports read/write on this tag since 2.16. Note that creating a profile with that info is a suitable way to access GPU on windows platforms. Windows 10, version 2004 (20H1) and later.

See https://learn.microsoft.com/en-us/windows/win32/wcs/display-calibration-mhc

```
typedef struct {
   cmsUInt32Number   CurveEntries;
   cmsFloat64Number* RedCurve;
   cmsFloat64Number* GreenCurve;
   cmsFloat64Number* BlueCurve;

   cmsFloat64Number  MinLuminance;        // ST.2086 min luminance in nits
   cmsFloat64Number  PeakLuminance;       // ST.2086 peak luminance in nits

   cmsFloat64Number XYZ2XYZmatrix[3][4];

} cmsMHC2Type;
```

Intents

ICC Intents	
INTENT_PERCEPTUAL	0
INTENT_RELATIVE_COLORIMETRIC	1
INTENT_SATURATION	2
INTENT_ABSOLUTE_COLORIMETRIC	3

Table 40

Non-ICC intents	
INTENT_PRESERVE_K_ONLY_PERCEPTUAL	10
INTENT_PRESERVE_K_ONLY_RELATIVE_COLORIMETRIC	11
INTENT_PRESERVE_K_ONLY_SATURATION	12
INTENT_PRESERVE_K_PLANE_PERCEPTUAL	13
INTENT_PRESERVE_K_PLANE_RELATIVE_COLORIMETRIC	14
INTENT_PRESERVE_K_PLANE_SATURATION	15

Table 41

2.0

```
cmsUInt32Number  cmsGetSupportedIntents(cmsUInt32Number nMax,
                                         cmsUInt32Number* Codes,
                                         char** Descriptions);
```

Fills a table with id-numbers and descriptions for all supported intents. *Little CMS* plug-in architecture allows to implement user-defined intents; use this function to get info about such extended functionality. Call with NULL as parameters to get the intent count

Parameters:

nMax: Max array elements to fill.
Codes [] : Pointer to user-allocated array of cmsUInt32Number to hold the intent id-numbers.
Descriptions []: Pointer to a user allocated array of char to hold the intent names.*

Returns:

Supported intents count.

2.6

```
cmsUInt32Number  cmsGetSupportedIntentsTHR(cmsContext ContextID,
                                           cmsUInt32Number nMax,
                                           cmsUInt32Number* Codes,
                                           char** Descriptions);
```

Fills a table with id-numbers and descriptions for all supported intents. *Little CMS* plug-in architecture allows to implement user-defined intents; use this function to get info about such extended functionality. Call with NULL as parameters to get the intent count

Parameters:

ContextID: Handle to user-defined context, or NULL for the global context
nMax: Max array elements to fill.
Codes [] : Pointer to user-allocated array of cmsUInt32Number to hold the intent id-numbers.
Descriptions []: Pointer to a user allocated array of char to hold the intent names.*

Returns:

Supported intent count.

2.0

```
cmsUInt32Number  cmsGetHeaderRenderingIntent(cmsHPROFILE hProfile);
```

Gets the profile header rendering intent. From the ICC spec: "*The rendering intent field shall specify the rendering intent which should be used (or, in the case of a Devicelink profile, was used) when this profile is (was) combined with another profile. In a sequence of more than two profiles, it applies to the combination of this profile and the next profile in the sequence and not to the entire sequence. Typically, the user or application will set the rendering intent dynamically at runtime or embedding time. Therefore, this flag may not have any meaning until the profile is used in some context, e.g. in a Devicelink or an embedded source profile.*"

Parameters:

hProfile: Handle to a profile object

Returns:

A cmsUInt32Number *holding the intent code, as described in* Intents *section.*

2.0

```
void   cmsSetHeaderRenderingIntent(cmsHPROFILE hProfile,
                                   cmsUInt32Number RenderingIntent);
```

Sets the profile header rendering intent. See the discussion above.

Parameters:

hProfile: Handle to a profile object
RenderingIntent: A cmsUInt32Number *holding the intent code, as described in* Intents *section.*

Returns:

None

2.1

```
cmsBool   cmsIsIntentSupported(cmsHPROFILE hProfile,
                               cmsUInt32Number Intent,
                               cmsUInt32Number UsedDirection);
```

Returns TRUE if the requested intent is implemented in the given direction. *Little CMS* has a fallback strategy that allows to specify any rendering intent when creating the transform, but the intent really being used may be another if the requested intent is not implemented.

Parameters:

hProfile: Handle to a profile object
Intent: A cmsUInt32Number *holding the intent code, as described in* Intents *section.*
UsedDirection: any of those constants:

```
#define LCMS_USED_AS_INPUT      0
#define LCMS_USED_AS_OUTPUT     1
#define LCMS_USED_AS_PROOF      2
```

Returns:

TRUE if the intent is implemented, FALSE otherwise.

Flags

To command the whole process. Some or none of this values can be joined via the "or" | operator.

cmsFLAGS_NOCACHE	0x0040 // Inhibit 1-pixel cache
cmsFLAGS_NOOPTIMIZE	0x0100 // Inhibit optimizations
cmsFLAGS_NULLTRANSFORM	0x0200 // Don't transform anyway
Proofing flags	
cmsFLAGS_GAMUTCHECK	0x1000 // Out of Gamut alarm
cmsFLAGS_SOFTPROOFING	0x4000 // Do softproofing
Misc	
cmsFLAGS_BLACKPOINTCOMPENSATION	0x2000
cmsFLAGS_NOWHITEONWHITEFIXUP	0x0004 // Don't fix scum dot
cmsFLAGS_HIGHRESPRECALC	0x0400 // Use more memory to give // better accurancy.
cmsFLAGS_LOWRESPRECALC	0x0800 // Use less memory to // minimize used resouces
For devicelink creation	
cmsFLAGS_8BITS_DEVICELINK	0x0008 // Create 8 bits devicelinks
cmsFLAGS_GUESSDEVICECLASS	0x0020 // Guess device class // (for transform2devicelink)
cmsFLAGS_KEEP_SEQUENCE	0x0080 // Keep profile sequence for // devicelink creation
Specific to a particular optimizations	
cmsFLAGS_FORCE_CLUT	0x0002 // Force CLUT optimization
cmsFLAGS_CLUT_POST_LINEARIZATION	0x0001 // create postlinearization // tables if possible
cmsFLAGS_CLUT_PRE_LINEARIZATION	0x0010 // create prelinearization // tables if possible
Unbounded mode control	
cmsFLAGS_NONEGATIVES	0x8000 // Prevent negative numbers // in floating point transforms
Fine-tune control over number of gridpoints	
cmsFLAGS_GRIDPOINTS(n)	(((n) & 0xFF) << 16)
CRD special	
cmsFLAGS_NODEFAULTRESOURCEDEF	0x01000000
Alpha channel	
cmsFLAGS_COPY_ALPHA	0x04000000 // Alpha channels are // copied to destination

Table 42

Color transforms

2.0

```
cmsHTRANSFORM  cmsCreateTransform(cmsHPROFILE Input,
                                    cmsUInt32Number InputFormat,
                                    cmsHPROFILE Output,
                                    cmsUInt32Number OutputFormat,
                                    cmsUInt32Number Intent,
                                    cmsUInt32Number dwFlags);
```

Creates a color transform for translating bitmaps.

Parameters:

Input: Handle to a profile object capable to work in input direction
InputFormat: A bit-field format specifier as described in Formatters section.
Output: Handle to a profile object capable to work in output direction
OutputFormat: A bit-field format specifier as described in Formatters section.
Intent: A cmsUInt32Number *holding the intent code, as described in* Intents.
dwFlags: A combination of bit-field constants described in Table 42.

Returns:

A handle to a transform object on success, NULL on error.

2.0

```
cmsHTRANSFORM  cmsCreateTransformTHR(cmsContext ContextID,
                                    cmsHPROFILE Input,
                                    cmsUInt32Number InputFormat,
                                    cmsHPROFILE Output,
                                    cmsUInt32Number OutputFormat,
                                    cmsUInt32Number Intent,
                                    cmsUInt32Number dwFlags);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.
Input: Handle to a profile object capable to work in input direction
Output: Handle to a profile object capable to work in output direction
InputFormat: A bit-field format specifier as described in Formatters section.
OutputFormat: A bit-field format specifier as described in Formatters section.
Intent: A cmsUInt32Number *holding the intent code, as described in* Intents *section.*
dwFlags: A combination of bit-field constants described in Table 42.

Returns:

A handle to a transform object on success, NULL on error.

2.0

```
void  cmsDeleteTransform(cmsHTRANSFORM hTransform);
```

Closes a transform handle and frees any associated memory. This function does NOT free the profiles used to create the transform.

Parameters:

*hTransform**:** Handle to a color transform object.*

Returns:

None

2.0

```
void   cmsDoTransform(cmsHTRANSFORM hTransform,
                      const void * InputBuffer,
                      void * OutputBuffer,
                      cmsUInt32Number Size);
```

This function translates bitmaps according of parameters setup when creating the color transform.

Parameters:

*hTransform**:** Handle to a color transform object.*
InputBuffer: A pointer to the input bitmap
OutputBuffer: A pointer to the output bitmap.
Size: the number of PIXELS to be transformed.

Returns:

None

***Notes**:*

This function is re-entrant. It never fails and it never send error logs.

You can re-use same transform handle in different threads calling this function.

2.4 DEPRECATED

```
void   cmsDoTransformStride(cmsHTRANSFORM hTransform,
                            const void * InputBuffer,
                            void * OutputBuffer,
                            cmsUInt32Number Size,  cmsUInt32Number Stride);
```

Deprecated. DO NOT USE. Use cmsDoTransformLineStride instead.

This function translates bitmaps according of parameters setup when creating the color transform. On planar-organized buffers, the parameter stride specifies the separation between planes, which may be different of the number of pixels to transform. The main application of this function is when several threads are transforming pixels from different zones of same planar buffer. Otherwise it is identical to ***cmDoTransform***

Parameters:

*hTransform**:** Handle to a color transform object.*
InputBuffer: A pointer to the input bitmap
OutputBuffer: A pointer to the output bitmap.
Size: the number of PIXELS to be transformed.
Stride: Plane separation on planar formats

Returns:

None

2.8

```
void   cmsDoTransformLineStride(cmsHTRANSFORM  Transform,
                                    const void* InputBuffer,
                                     void* OutputBuffer,
                                     cmsUInt32Number PixelsPerLine,
                                     cmsUInt32Number LineCount,
                                     cmsUInt32Number BytesPerLineIn,
                                     cmsUInt32Number BytesPerLineOut,
                                     cmsUInt32Number BytesPerPlaneIn,
                                     cmsUInt32Number BytesPerPlaneOut);
```

This function translates bitmaps with complex organization. Each bitmap may contain several lines, and every may have padding. The distance from one line to the next one is BytesPerLine{In/Out}. In planar formats, each line may hold several planes, each plane may have padding. Padding of lines and planes should be same across all bitmap. I.e. all lines in same bitmap have to be padded in same way. This function may be more efficient that repeated calls to cmsDoTransform(), especially when customized plug-ins are being used.

Parameters:

*hTransform**:** Handle to a color transform object.*
InputBuffer: A pointer to the input bitmap
OutputBuffer: A pointer to the output bitmap.
PixelsPerLine: The number of pixels for line, which is same on input and in output.
LineCount: The number of lines, which is same on input and output
BytesPerLine{In,Out}: The distance in bytes from one line to the next one.
BytesPerPlaneIn{In,Out}: The distance in bytes from one plane to the next one *inside a line. Only applies in planar formats.*

Returns:

None

Notes:

- **This function is quite efficient, and is used by some plug-ins to give a big speedup. If you can load whole image to memory and then call this function over it, it will be much faster than any other approach.**
- BytesPerPlaneIn{In,Out} is ignored if the formats used are not planar. Please note 1-plane planar is indistinguishable from 1-component chunky, so BytesPerPlane is ignored as well in this case.
- If the image does not have any gaps between the pixels and lines BytesPerLine{} are equal to the pixel's size * PixelsPerLine.
- To specify padding between pixels, use T_EXTRA() and EXTRA_SH() to specify extra channels.
- **This function is re-entrant. It never fails and it never send error logs.**
- **You can re-use same transform handle in different threads calling this function.**

To compute the pixel channel size, you could use T_BYTES macro for integer formats. If you want to include double formats as well, use this small code:

```
cmsUInt32Number PixelChannelSize(cmsUInt32Number Format)
{
   cmsUInt32Number fmt_bytes = T_BYTES(Format);

   // For double, the T_BYTES field is zero
   if (fmt_bytes == 0)
      return sizeof(cmsUInt64Number);

   // Otherwise, it is already correct for all formats
   return fmt_bytes;
}
```

Pixel size is then the number of channels, plus the extra channels times the channel size

```
PixelSize = (T_CHANNELS(Format) + T_EXTRA(Format)) * PixelChannelSize(Format)
```

Proofing transforms

A proofing transform does emulate the colors that would appear as the image were rendered on a specific device. That is, for example, with a proofing transform I can see how will look a photo of my little daughter if rendered on my HP printer. Since most printer profiles does include some sort of gamut-remapping, it is likely colors will not look as the original. Using a proofing transform, it can be done by using the appropriate function. Note that this is an important feature for final users, it is worth of all color-management stuff if the final media is not cheap.

2.0

```
cmsHTRANSFORM  cmsCreateProofingTransform(cmsHPROFILE Input,
                                           cmsUInt32Number InputFormat,
                                           cmsHPROFILE Output,
                                           cmsUInt32Number OutputFormat,
                                           cmsHPROFILE Proofing,
                                           cmsUInt32Number Intent,
                                           cmsUInt32Number ProofingIntent,
                                           cmsUInt32Number dwFlags);
```

Same as cmsCreateTransform(), but including soft-proofing. The obtained transform emulates the device described by the "Proofing" profile. Useful to preview final result without rendering to the physical medium. To enable proofing and gamut check you need to include following flags:

cmsFLAGS_GAMUTCHECK: Color out of gamut are flagged to a fixed color defined by the function *cmsSetAlarmCodes*

cmsFLAGS_SOFTPROOFING: does emulate the Proofing device.

Parameters:

Input: Handle to a profile object capable to work in input direction
Output: Handle to a profile object capable to work in output direction
InputFormat: A bit-field format specifier as described in Formatters section.
OutputFormat: A bit-field format specifier as described in Formatters section.
Intent: A cmsUInt32Number *holding the intent code, as described in* Intents *section.*
ProofingIntent: A cmsUInt32Number *holding the intent code, as described in* Intents *section.*
dwFlags: A combination of bit-field constants described in Table 42.

Returns:

A handle to a transform object on success, NULL on error.

2.0

```
cmsHTRANSFORM  cmsCreateProofingTransformTHR(cmsContext ContextID,
                                    cmsHPROFILE Input,
                                    cmsUInt32Number InputFormat,
                                    cmsHPROFILE Output,
                                    cmsUInt32Number OutputFormat,
                                    cmsHPROFILE Proofing,
                                    cmsUInt32Number Intent,
                                    cmsUInt32Number ProofingIntent,
                                    cmsUInt32Number dwFlags);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.
Input: Handle to a profile object capable to work in input direction
Output: Handle to a profile object capable to work in output direction
InputFormat: A bit-field format specifier as described in Formatters section.
OutputFormat: A bit-field format specifier as described in Formatters section.
Intent: A cmsUInt32Number *holding the intent code, as described in* Intents *section.*
ProofingIntent: A cmsUInt32Number *holding the intent code, as described in* Intents *section.*
dwFlags: A combination of bit-field constants described in Table 42.

Returns:

A handle to a transform object on success, NULL on error.

2.0

```
void    cmsSetAlarmCodes(cmsUInt16Number AlarmCodes[cmsMAXCHANNELS]);
```

Sets the global codes used to mark out-out-gamut on Proofing transforms. Values are meant to be encoded in 16 bits.

Parameters:

AlarmCodes: Array [16] of codes. ***ALL 16 VALUES MUST BE SPECIFIED****, set to zero unused channels.*

Returns:

None

2.0

```
void  cmsGetAlarmCodes(cmsUInt16Number  AlarmCodes[cmsMAXCHANNELS]);
```

Gets the current global codes used to mark out-out-gamut on Proofing transforms. Values are meant to be encoded in 16 bits.

Parameters:

AlarmCodes: Array [16] of codes. ***ALL 16 VALUES WILL BE OVERWRITTEN****.*

Returns:

None

2.6

```
void  cmsSetAlarmCodesTHR(cmsContext ContextID,
                          const cmsUInt16Number AlarmCodes[cmsMAXCHANNELS]);
```

Sets the codes used to mark out-out-gamut on Proofing transforms for a given context. Values are meant to be encoded in 16 bits.

Parameters:

ContextID: Handle to user-defined context, or NULL for the global alarm codes
AlarmCodes: Array [16] of codes. ***ALL 16 VALUES MUST BE SPECIFIED****, set to zero unused channels.*

Returns:

None

2.6

```
void cmsGetAlarmCodesTHR(cmsContext ContextID,
                              cmsUInt16Number AlarmCodes[cmsMAXCHANNELS]);
```

Gets the current codes used to mark out-out-gamut on Proofing transforms for the given context. Values are meant to be encoded in 16 bits.

Parameters:

ContextID: Handle to user-defined context, or NULL for the global context
AlarmCodes: Array [16] of codes. ***ALL 16 VALUES WILL BE OVERWRITTEN****.*

Returns: *None*

2.0

```
cmsFloat64Number   cmsSetAdaptationState(cmsFloat64Number d);
```

Sets adaptation state for absolute colorimetric intent, on all but *cmsCreateExtendedTransform*. *Little CMS* can handle incomplete adaptation states.

Parameters:

d: Degree on adaptation 0=Not adapted, 1=Complete adaptation, in-between=Partial adaptation. Use negative values to return the global state without changing it.

Returns:

Previous global adaptation state.

2.6

```
cmsFloat64Number   cmsSetAdaptationStateTHR(cmsContext ContextID,
                                             cmsFloat64Number d);
```

Sets adaptation state for absolute colorimetric intent in the given context. Adaptation state applies on all but *cmsCreateExtendedTransformTHR()*. *Little CMS* can handle incomplete adaptation states.

Parameters:

ContextID: Handle to user-defined context, or NULL for the global context
d: Degree on adaptation 0=Not adapted, 1=Complete adaptation, in-between=Partial adaptation. Use negative values to return the global state without changing it.

Returns:

Previous global adaptation state.

Multiprofile transforms

User passes in an array of handles to open profiles. The returned color transform do "smelt" all profiles in a single devicelink. Color spaces must be paired with the exception of Lab/XYZ, which can be interchanged.

2.0

```
cmsHTRANSFORM  cmsCreateMultiprofileTransform(cmsHPROFILE hProfiles[],
                                    cmsUInt32Number nProfiles,
                                    cmsUInt32Number InputFormat,
                                    cmsUInt32Number OutputFormat,
                                    cmsUInt32Number Intent,
                                    cmsUInt32Number dwFlags);
```

Parameters:

hProfiles[] : Array of handles to open profile objects.
nProfiles: Number of profiles in the array.
InputFormat: A bit-field format specifier as described in Formatters section.
OutputFormat: A bit-field format specifier as described in Formatters section.
Intent: A cmsUInt32Number *holding the intent code, as described in* Intents *section.*
dwFlags: A combination of bit-field constants described in Table 42.

Returns:

A handle to a transform object on success, NULL on error.

2.0

```
cmsHTRANSFORM  cmsCreateMultiprofileTransformTHR(cmsContext ContextID,
                                    cmsHPROFILE hProfiles[],
                                    cmsUInt32Number nProfiles,
                                    cmsUInt32Number InputFormat,
                                    cmsUInt32Number OutputFormat,
                                    cmsUInt32Number Intent,
                                    cmsUInt32Number dwFlags);
```

Same as anterior, but allowing a ContextID to be passed through.

Parameters:

ContextID: Pointer to a user-defined context cargo.
hProfiles[] : Array of handles to open profile objects.
nProfiles: Number of profiles in the array.
InputFormat: A bit-field format specifier as described in Formatters section.
OutputFormat: A bit-field format specifier as described in Formatters section.
Intent: A cmsUInt32Number *holding the intent code, as described in* Intents *section.*

dwFlags: A combination of bit-field constants described in Table 42.

Returns:

A handle to a transform object on success, NULL on error.

2.0

```
cmsHTRANSFORM  cmsCreateExtendedTransform(cmsContext ContextID,
                                          cmsUInt32Number nProfiles, cmsHPROFILE hProfiles[],
                                          cmsBool  BPC[],
                                          cmsUInt32Number Intents[],
                                          cmsFloat64Number AdaptationStates[],
                                          cmsHPROFILE hGamutProfile,
                                          cmsUInt32Number nGamutPCSposition,
                                          cmsUInt32Number InputFormat,
                                          cmsUInt32Number OutputFormat,
                                          cmsUInt32Number dwFlags);
```

Extended form of multiprofile color transform creation, exposing all parameters for each profile in the chain. All other transform cration functions are wrappers to this call.

Parameters:

ContextID: Pointer to a user-defined context cargo.
hProfiles[] : Array of handles to open profile objects.
nProfiles: Number of profiles in the array.
BPC [] : Array of black point compensation states
hGamutProfile: Handle to a profile holding gamut information for gamut check. Only used if cmsFLAGS_GAMUTCHECK specified. Set to NULL for no gamut check.
nGamutPCSPosition: Position in the chain of Lab/XYZ PCS to check against gamut profile Only used if cmsFLAGS_GAMUTCHECK specified.
InputFormat: A bit-field format specifier as described in Formatters section.
OutputFormat: A bit-field format specifier as described in Formatters section.
Intent: A *cmsUInt32Number holding the intent code, as described in Intents section.*
dwFlags: A combination of bit-field constants described in Table 42.

Returns:

A handle to a transform object on success, NULL on error.

Dynamically changing the input/output formats

Not all transforms can be changed, cmsChangeBuffersFormat only works on transforms created originally with at least 16 bits of precision.

2.2

```
cmsUInt32Number   cmsGetTransformInputFormat(cmsHTRANSFORM hTransform);
```

Returns the input format associated with a given transform.

Parameters:

*hTransform**:** Handle to a color transform object.*

Returns:

The input format associated with the given transform or 0 if NULL parameter

2.2

```
cmsUInt32Number  cmsGetTransformOutputFormat(cmsHTRANSFORM hTransform);
```

Returns the output format associated with a given transform.

Parameters:

*hTransform**:** Handle to a color transform object.*

Returns:

The output format associated with the given transform or 0 if NULL parameter

2.1

```
cmsBool    cmsChangeBuffersFormat(cmsHTRANSFORM hTransform,
                                  cmsUInt32Number InputFormat,
                                  cmsUInt32Number OutputFormat);
```

This function does change the encoding of buffers in a yet-existing transform. Not all transforms can be changed, cmsChangeBuffersFormat only works on transforms created originally with at least 16 bits of precision. This function is provided for backwards compatibility and should be avoided whenever possible, as it prevents transform optimization.

Parameters:

Transform: Handle to a color transform object.
InputFormat: A bit-field format specifier as described in Formatters section.
OutputFormat: A bit-field format specifier as described in Formatters section.

Returns:

TRUE on success FALSE on error.

PostScript generation

When dealing with PostScript, instead of creating a transform, it is sometimes desirable to delegate the color management to PostScript interpreter. *Little CMS* does provide functions to translate input and output profiles into Color Space Arrays (CSA) and Color Rendering Dictionaries (CRD).

- CRD are equivalent to output (printer) profiles. Can be loaded into printer at startup and can be stored as resources.
- CSA are equivalent to input and workspace profiles, and are intended to be included in the document definition.

Since the length of the resultant PostScript code is unknown in advance, you can call the functions with len=0 and Buffer=NULL to get the length. After that, you need to allocate enough memory to contain the whole block.

Devicelink profiles are supported, as long as input color space matches Lab/XYZ for CSA or output color space matches Lab/XYZ for CRD.

WARNING: Precision of PostScript is limited to 8 bits per sample. If you can choose between normal transforms and CSA/CRD, normal transforms will give more accuracy.

2.0

```
cmsUInt32Number  cmsGetPostScriptColorResource(cmsContext ContextID,
                                               cmsPSResourceType Type,
                                               cmsHPROFILE hProfile,
                                               cmsUInt32Number Intent,
                                               cmsUInt32Number dwFlags,
                                               cmsIOHANDLER* io);
```

Little CMS 2 unified method to create postscript color resources. Serialization is performed by the given iohandler object.

Parameters:

ContextID: Pointer to a user-defined context cargo.
*Type: Either **cmsPS_RESOURCE_CSA** or **cmsPS_RESOURCE_CRD***
hProfile: Handle to a profile object
Intent: A cmsUInt32Number *holding the intent code, as described in* Intents *section.*
dwFlags: A combination of bit-field constants described in Table 42*.*
Iohandler: Pointer to a serialization object.

Returns:

The resource size in bytes on success, 0 en error.

2.0

```
cmsUInt32Number  cmsGetPostScriptCSA(cmsContext ContextID,
                                      cmsHPROFILE hProfile,
                                      cmsUInt32Number Intent,
                                      cmsUInt32Number dwFlags,
                                      void* Buffer,  cmsUInt32Number dwBufferLen);
```

A wrapper on *cmsGetPostScriptColorResource* to simplify CSA generation.

Parameters:

ContextID: Pointer to a user-defined context cargo.
hProfile: Handle to a profile object
Intent: A cmsUInt32Number *holding the intent code, as described in* Intents *section.*
dwFlags: A combination of bit-field constants described in Table 42.
Buffer: Pointer to a user-allocated memory block or NULL. If specified, It should be big enough to hold the generated resource.
dwBufferLen: Length of Buffer in bytes.

Returns:

The resource size in bytes on success, 0 en error.

2.0

```
cmsUInt32Number  cmsGetPostScriptCRD(cmsContext ContextID,
                                      cmsHPROFILE hProfile,
                                      cmsUInt32Number Intent,
                                      cmsUInt32Number dwFlags,
                                      void* Buffer,  cmsUInt32Number dwBufferLen);
```

A wrapper on *cmsGetPostScriptColorResource* to simplify CRD generation.

Parameters:

ContextID: Pointer to a user-defined context cargo.
hProfile: Handle to a profile object
Intent: A cmsUInt32Number *holding the intent code, as described in* Intents *section.*
dwFlags: A combination of bit-field constants described in Table 42.
Buffer: Pointer to a user-allocated memory block or NULL. If specified, It should be big enough to hold the generated resource.
dwBufferLen: Length of Buffer in bytes.

Returns:

The resource size in bytes on success, 0 en error.

Δ E metrics

You don't have to spend too long in the color management world before you come across the term Delta-E. As with many things color, it seems simple to understand at first, yet the closer you look, the more elusive it gets. Delta-E (dE) is a single number that represents the 'distance' between two colors. The idea is that a dE of 1.0 is the smallest color difference the human eye can see. So any dE less than 1.0 is imperceptible and it stands to reason that any dE greater than 1.0 is noticeable. Unfortunately it's not that simple. Some color differences greater than 1 are perfectly acceptable, maybe even unnoticeable. Also, the same dE color difference between two yellows and two blues may not look like the same difference to the eye and there are other places where it can fall down. It's perfectly understandable that we would want to have a system to show errors. After all, we've spent the money on the instruments; shouldn't we get numbers from them? Delta-E numbers can be used for:

- how far off is a print or proof from the original
- how much has a device drifted
- how effective is a particular profile for printing or proofing
- removes subjectivity (as much as possible)

These functions does compute the difference between two Lab colors, using several difference spaces.

2.0

```
cmsFloat64Number  cmsDeltaE(const cmsCIELab* Lab1,  const cmsCIELab* Lab2);
```

The L*a*b* color space was devised in 1976 and, at the same time delta-E 1976 (dE76) came into being. If you can imagine attaching a string to a color point in 3D Lab space, dE76 describes the sphere that is described by all the possible directions you could pull the string. If you hear people speak of just plain 'delta-E' they are probably referring to dE76. It is also known as dE-Lab and dE-ab. One problem with dE76 is that Lab itself is not 'perceptually uniform' as its creators had intended. So different amounts of visual color shift in different color areas of Lab might have the same dE76 number. Conversely, the same amount of color shift might result in different dE76 values. Another issue is that the eye is most sensitive to hue differences, then chroma and finally lightness and dE76 does not take this into account.

Parameters:

Lab1: Pointer to first cmsCIELab value as described in Table 13

Lab2: Pointer to second cmsCIELab value as described in Table 13

Returns:

The dE76 metric value.

2.0

```
cmsFloat64Number  cmsCMCdeltaE(const cmsCIELab* Lab1,
                               const cmsCIELab* Lab2,
                               cmsFloat64Number  l,  cmsFloat64Number  c);
```

In 1984 the CMC (Colour Measurement Committee of the Society of Dyes and Colourists of Great Britain) developed and adopted an equation based on LCH numbers. Intended for the textiles industry, CMC l:c allows the setting of lightness (l) and chroma (c) factors. As the eye is more sensitive to chroma, the default ratio for l:c is 2:1 allowing for 2x the difference in lightness than chroma (numbers). There is also a 'commercial factor' (cf) which allows an overall varying of the size of the tolerance region according to accuracy requirements. A cf=1.0 means that a delta-E CMC value <1.0 is acceptable.

CMC l:c is designed to be used with D65 and the CIE Supplementary Observer. Commonly-used values for l:c are 2:1 for acceptability and 1:1 for the threshold of imperceptibility.

Parameters:

Lab1: Pointer to first cmsCIELab value as described in Table 13
Lab2: Pointer to second cmsCIELab value as described in Table 13

Returns:

The dE CMC metric value.

2.0

```
cmsFloat64Number  cmsBFDdeltaE(const cmsCIELab* Lab1, const cmsCIELab* Lab2);
```

BFD delta E metric.

Parameters:

Lab1: Pointer to first cmsCIELab value as described in Table 13
Lab2: Pointer to second cmsCIELab value as described in Table 13

Returns:

The dE BFD metric value.

2.0

```
cmsFloat64Number  cmsCIE94DeltaE(const cmsCIELab* Lab1,
                                 const cmsCIELab* Lab2);
```

A technical committee of the CIE (TC1-29) published an equation in 1995 called CIE94. The equation is similar to CMC but the weighting functions are largely based on RIT/DuPont tolerance data derived from automotive paint experiments where sample surfaces are smooth. It also has ratios, labeled *kL* (lightness) and *Kc* (chroma) and the commercial factor (*cf*) but these tend to be preset in software and are not often exposed for the user (as it is the case in Little CMS).

Parameters:

Lab1: Pointer to first cmsCIELab value as described in Table 13
Lab2: Pointer to second cmsCIELab value as described in Table 13

Returns:

The CIE94 dE metric value.

2.0

```
cmsFloat64Number  cmsCIE2000DeltaE(const cmsCIELab* Lab1,
                                   const cmsCIELab* Lab2,
                                   cmsFloat64Number Kl,
                                   cmsFloat64Number Kc,
                                   cmsFloat64Number Kh);
```

Delta-E 2000 is the first major revision of the dE94 equation. Unlike dE94, which assumes that L* correctly reflects the perceived differences in lightness, dE2000 varies the weighting of L* depending on where in the lightness range the color falls. dE2000 is still under consideration and does not seem to be widely supported in graphics arts applications.

Parameters:

Lab1: Pointer to first cmsCIELab value as described in Table 13
Lab2: Pointer to second cmsCIELab value as described in Table 13

Returns:

The CIE2000 dE metric value.

Temperature <-> Chromaticity (Black body)

Color temperature is a characteristic of visible light that has important applications. The color temperature of a light source is determined by comparing its chromaticity with that of an ideal black-body radiator. The temperature (usually measured in kelvin, K) is that source's color temperature at which the heated black-body radiator matches the color of the light source for a black body source. Higher color temperatures (5,000 K or more) are cool (bluish white) colors, and lower color temperatures (2,700–3,000 K) warm (yellowish white through red) colors.

2.0

```
cmsBool cmsWhitePointFromTemp(cmsCIExyY* WhitePoint,
                              cmsFloat64Number  TempK);
```

Correlates a black body chromaticity from given temperature in ºK. Valid range is 4000K-25000K.

Parameters:

WhitePoint: Pointer to a user-allocated cmsCIExyY variable to receive the resulting chromaticity.
TempK: Temperature in ºK

Returns:

TRUE on success, FALSE on error.

2.0

```
cmsBool cmsTempFromWhitePoint(cmsFloat64Number* TempK,
                              const cmsCIExyY* WhitePoint);
```

Correlates a black body temperature in ºK from given chromaticity.

Parameters:

TempK: Pointer to a user-allocated cmsFloat64Number variable to receive the resulting temperature.
WhitePoint: Target chromaticity in cmsCIExyY

Returns:

TRUE on success, FALSE on error.

CIE CAM02

Viewing conditions. Please note those are CAM model viewing conditions, and not the ICC tag viewing conditions, which I'm naming cmsICCViewingConditions to make differences evident. Unfortunately, the tag cannot deal with surround La, Yb and D value so is basically useless to store CAM02 viewing conditions.

cmsViewingConditions	
cmsCIEXYZ	whitePoint;
cmsFloat64Number	Yb;
cmsFloat64Number	La;
int	surround;
cmsFloat64Number	D_value;

Table 43

surround	
AVG_SURROUND	1
DIM_SURROUND	2
DARK_SURROUND	3
CUTSHEET_SURROUND	4

Table 44

```
D_CALCULATE      (-1)
```

2.0

```
cmsHANDLE cmsCIECAM02Init(cmsContext ContextID,
                          const cmsViewingConditions* pVC);
```

Does create a CAM02 object based on given viewing conditions. Such object may be used as a color appearance model and evaluated in forward and reverse directions. Viewing conditions structure is detailed in Table 43. The surround member has to be one of the values enumerated in Table 44. Degree of chromatic adaptation (*d*), can be specified in 0...1.0 range, or the model can be instructed to calculate it by using D_CALCULATE constant (-1).

Parameters:

ContextID: Pointer to a user-defined context cargo.
pVC: Pointer to a structure holding viewing conditions (Table 44)

Returns:

Handle to CAM02 object or NULL on error.

2.0

```
void cmsCIECAM02Done(cmsHANDLE hModel);
```

Terminates a CAM02 object, freeing all involved resources.

Parameters:

hModel: Handle to a CAM02 object

Returns:

None

2.0

```
void cmsCIECAM02Forward(cmsHANDLE hModel,
                        const cmsCIEXYZ* pIn,
                        cmsJCh* pOut);
```

Evaluates the CAM02 model in the forward direction XYZ → JCh

Parameters:

hModel: Handle to a CAM02 object

pIn: Points to the input XYZ value

pOut: Points to the output JCh value

Returns:

None

2.0

```
void cmsCIECAM02Reverse(cmsHANDLE hModel,
                        const cmsJCh* pIn,
                        cmsCIEXYZ* pOut);
```

Evaluates the CAM02 model in the reverse direction JCh → XYZ

Parameters:

hModel: Handle to a CAM02 object

pIn: Points to the input JCh value

pOut: Points to the output XYZ value

Returns:

None

Gamut boundary description

Gamut boundary description by using Jan Morovic's Segment maxima method. Many thanks to Jan for allowing me to use his algorithm.

2.0

```
cmsHANDLE  cmsGBDAlloc(cmsContext ContextID);
```

Allocates an empty gamut boundary descriptor with no known points.

Parameters:

ContextID: Pointer to a user-defined context cargo.

Returns:

A handle to a gamut boundary descriptor on success, NULL on error.

2.0

```
void  cmsGBDFree(cmsHANDLE hGBD);
```

Frees a gamut boundary descriptor and any associated resources.

Parameters:

hGBD: Handle to a gamut boundary descriptor.

Returns:

None

2.0

```
cmsBool  cmsGDBAddPoint(cmsHANDLE hGBD, const cmsCIELab* Lab);
```

Adds a new sample point for computing the gamut boundary descriptor. This function can be called as many times as known points. No memory or other resources are wasted by adding new points. The gamut boundary descriptor cannot be checked until *cmsGDBCompute*() is called.

Parameters:

hGBD: Handle to a gamut boundary descriptor.

Lab: Pointer to a cmsCIELab value as described in Table 13

Returns:

TRUE on success, FALSE on error.

2.0

```
cmsBool  cmsGDBCompute(cmsHANDLE hGDB,  cmsUInt32Number dwFlags);
```

Computes the gamut boundary descriptor using all know points and interpolating any missing sector(s). Call this function after adding all know points with cmsGDBAddPoint() and before using cmsGDBCheckPoint().

Parameters:

hGBD: Handle to a gamut boundary descriptor.
dwFlags: reserved (unused). Set it to 0

Returns:

TRUE on success, FALSE on error

2.0

```
cmsBool   cmsGDBCheckPoint(cmsHANDLE hGBD,  const cmsCIELab* Lab);
```

Checks whatever a Lab value is inside a given gamut boundary descriptor.

Parameters:

hGBD: Handle to a gamut boundary descriptor.
Lab: Pointer to a cmsCIELab value as described in Table 13

Returns:

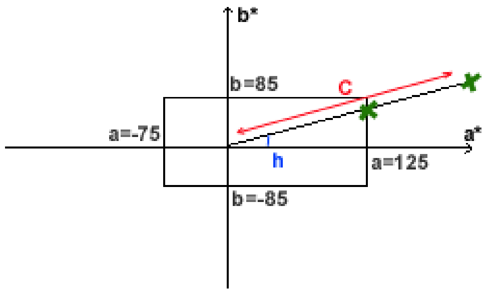
TRUE if point is inside gamut, FALSE otherwise.

Gamut mapping

I'm using LCh (polar form of Lab) to do the clipping.

L = L
C = sqrt(a*a+b*b)
h = atan(b/a)

Where C=colorfulness and h=hue.



L is unchanged and not used. The gamut boundaries are the black rectangle. I take a Lab value, if inside gamut, don't touch anything, if outside, for example, the green point, I convert to LCh, keep h constant, and reduce C (in red) until inside gamut. This gives the second green point, with quite different a, b, but visually similar.

2.0

```
cmsBool   cmsDesaturateLab(cmsCIELab* Lab,
                           double amax, double amin,
                           double bmax, double bmin);
```

Parameters:

Lab: Pointer to a cmsCIELab value as described in Table 13
amin, amax, bmin, bmax: boundaries of gamut rectangle

Returns:

TRUE on success, FALSE on error

MD5 message digest

In cryptography, MD5 (Message-Digest algorithm 5) is a widely used cryptographic hash function with a 128-bit hash value. As an Internet standard (RFC 1321), MD5 has been employed in a wide variety of security applications, and is also commonly used to check the integrity of files. ICC profiles can use MD5 as a checksum, and as a unique identifier for a profile.

Profile ID as computed by MD5 algorithm

cmsProfileID (union)	
cmsUInt8Number	ID8[16];
cmsUInt16Number	ID16[8];
cmsUInt32Number	ID32[4];

Table 45

2.0

```
cmsBool cmsMD5computeID(cmsHPROFILE hProfile);
```

Computes a MD5 checksum and stores it as Profile ID in the profile header.

Parameters:

hProfile: Handle to a profile object

Returns:

TRUE on success, FALSE on error

2.0

```
void cmsGetHeaderProfileID(cmsHPROFILE hProfile, cmsUInt8Number* ProfileID);
```

Retrieves the Profile ID stored in the profile header.

Parameters:

hProfile: Handle to a profile object
ProfileID: Pointer to a Profile ID union as described in Table 45

Returns:

None

2.0

```
void cmsSetHeaderProfileID(cmsHPROFILE hProfile, cmsUInt8Number* ProfileID);
```

Replaces the the Profile ID stored in the profile header.

Parameters:

hProfile: Handle to a profile object
ProfileID: Pointer to a Profile ID union as described in Table 45

Returns:

None

CGATS.17-200x handling

ANSI CGATS.17 is THE standard text file format for exchanging color measurement data. This standard text format (the ASCII version is by far the most common) is the format accepted by most color measurement and profiling applications.

It consists of a Preamble section containing originator information, keyword definitions, etc and then one or more data sections, each consisting of header and data subsections. The header subsection is where the BEGIN_DATA_FORMAT and END_DATA_FORMAT delimiters define the actual data types / units contained in the following tables. The data subsection contains the BEGIN_DATA and END_DATA delimiters which contain the actual color information in tabular form.

CGATS.17 text files can contain device (RGB, CMYK, etc), colorimetric (Lab, XYZ, etc), densitometric, spectral, naming and other information so it is a fairly comprehensive storage and exchange format.

2.0

```
cmsHANDLE  cmsIT8Alloc(cmsContext ContextID);
```

Allocates an empty CGATS.17 object.

Parameters:

ContextID: Pointer to a user-defined context cargo.

Returns:

A handle to a CGATS.17 object on success, NULL on error.

2.0

```
void   cmsIT8Free(cmsHANDLE cmsIT8);
```

This function frees the CGATS.17 object. After a call to this function, all memory pointers associated with the object are freed and therefore no longer valid.

Parameters:

hIT8: A handle to a CGATS.17 object.

Returns:

None

Tables

In the *Little CMS* implementation, a CGATS.17 object may contain any number of tables. Tables are separated by END_DATA keyword. This agrees with the latest CGATS.17 spec.

2.0

```
cmsUInt32Number  cmsIT8TableCount(cmsHANDLE hIT8);
```

This function returns the number of tables found in the current CGATS object.

Parameters:

hIT8: A handle to a CGATS.17 object.

Returns:

The number of tables on success, 0 on error.

2.0

```
cmsInt32Number   cmsIT8SetTable(cmsHANDLE hIT8, cmsUInt32Number nTable);
```

This function positions the IT8 object in a given table, identified by its position. Setting nTable to Table Count + 1 does allocate a new empty table

Parameters:

hIT8: A handle to a CGATS.17 object.
nTable: The table number (0 based)

Returns:

The current table number on success, -1 on error.

Persistence

These are functions to load/save CGATS.17 objects from file and memory stream.

2.0

```
cmsHANDLE   cmsIT8LoadFromFile(cmsContext ContextID,  const char* cFileName);
```

This function allocates a CGATS.17 object and fills it with the contents of cFileName. Used for reading existing CGATS files.

Parameters:

ContextID: Pointer to a user-defined context cargo.
cFileName: The CGATS.17 file name to read/parse

Returns:

A handle to a CGATS.17 on success, NULL on error.

2.0

```
cmsHANDLE      cmsIT8LoadFromMem(cmsContext ContextID,
                                 void *Ptr,
                                 cmsUInt32Number len);
```

Same as anterior, but the IT8/CGATS.13 stream is read from a memory block.

Parameters:

ContextID: Pointer to a user-defined context cargo.
Ptr: Points to a block of contiguous memory containing the CGATS.17 stream.
len: stream size measured in bytes.

Returns:

A handle to a CGATS.17 on success, NULL on error.

2.0

```
cmsBool    cmsIT8SaveToFile(cmsHANDLE hIT8,
                            const char* cFileName);
```

This function saves a CGATS.17 object to a file.

Parameters:

hIT8: A handle to a CGATS.17 object.
cFileName: Destination filename. Existing file will be overwritten if possible.

Returns:

TRUE on success, FALSE on error

2.0

```
cmsBool   cmsIT8SaveToMem(cmsHANDLE hIT8,
                          void *MemPtr,
                          cmsUInt32Number* BytesNeeded);
```

This function saves a CGATS.17 object to a contiguous memory block. Setting *MemPtr* to NULL forces the function to calculate the needed amount of memory.

Parameters:

hIT8: A handle to a CGATS.17 object.
MemPtr: Pointer to a user-allocated memory block or NULL. If specified, It should be big enough to hold the generated resource.
BytesNeeded: Points to a user-allocated cmsUInt32Number *which will receive the needed memory size in bytes.*

Returns:

TRUE on success, FALSE on error

Type and comments

The sheet type is an identifier placed on the very first line of the CGATS.17 object.

2.0

```
const char* cmsIT8GetSheetType(cmsHANDLE hIT8);
```

This function returns the type of the IT8 object. Memory is handled by the CGATS.17 object and should not be freed by the user.

Parameters:

hIT8: A handle to a CGATS.17 object.

Returns:

A pointer to internal block of memory containing the type on success, NULL on error.

2.0

```
cmsBool  cmsIT8SetSheetType(cmsHANDLE hIT8,  const char* Type);
```

This function sets the type of a CGATS.17 object

Parameters:

hIT8: A handle to a CGATS.17 object.
Type: The new type

Returns:

TRUE on success, FALSE on error

2.0

```
cmsBool   cmsIT8SetComment(cmsHANDLE hIT8,    const char* cComment);
```

This function is intended to provide a way automated IT8 creators can embed comments into the file. Comments have no effect, and its only purpose is to document any of the file meaning. On this function the calling order is important; as successive calls to cmsIT8SetComment do embed comments in the same order the function is being called.

Parameters:

hIT8: A handle to a CGATS.17 object.
cComment: The comment to inserted

Returns:

TRUE on success, FALSE on error.

Properties

Properties are pairs <***identifier***> <***value***>. Each table may contain any number of properties. Its primary purpose is store simple settings. Additionally, sub-properties are allowed if <***value***> is a string in the form:

"SUBPROP1,1;SUBPROP2,2;…"

2.0

```
cmsBool  cmsIT8SetPropertyStr(cmsHANDLE hIT8,
                              const char* cProp,
                              const char *Str);
```

Sets a property as a literal string in current table. The string is enclosed in quotes "".

Parameters:

hIT8: A handle to a CGATS.17 object.
cProp: A string holding property name.
Str: The literal string.

Returns:

TRUE on success, FALSE on error.

2.0

```
cmsBool  cmsIT8SetPropertyDbl(cmsHANDLE hIT8,
                              const char* cProp,
                              cmsFloat64Number Val);
```

Sets a property as a cmsFloat64Number in current table.

Parameters:

hIT8: A handle to a CGATS.17 object.
cProp: A string holding property name.
Val: The data for the intended property as cmsFloat64Number.

Returns:

TRUE on success, FALSE on error.

2.0

```
cmsBool  cmsIT8SetPropertyHex(cmsHANDLE hIT8,
                              const char* cProp,
                              cmsUInt32Number Val);
```

Sets a property as an hexadecimal constant (appends 0x) in current table.

Parameters:

hIT8: A handle to a CGATS.17 object.
cProp: A string holding property name.
Val: The value to be set (32 bits max)

Returns:

TRUE on success, FALSE on error

2.0

```
cmsBool   cmsIT8SetPropertyUncooked(cmsHANDLE hIT8,
                                    const char* cProp, const char* Buffer);
```

Sets a property with no interpretation in current table. No quotes “” are added. No checking is performed, and it is up to the programmer to make sure the string is valid.

Special prefixes:

0b : Binary
0x : Hexadecimal

Parameters:

hIT8: A handle to a CGATS.17 object.
cProp: A string holding property name.
Buffer: A string holding the uncooked value to place in the CGATS file.

Returns:

TRUE on success, FALSE on error.

2.0

```
cmsBool cmsIT8SetPropertyMulti(cmsHANDLE hIT8,
                               const char* Key, const char* SubKey,
                               const char *Buffer)
```

Adds a new sub-property to the property ***Key***. Value of ***buffer*** is interpreted literally.

Parameters:

hIT8: A handle to a CGATS.17 object.
cKey: A string holding property name.
SubKey: A string holding the sub-property name.
Buffer: A string holding the uncooked value of sub-property.

Returns:

TRUE on success, FALSE on error.

2.0

```
const char*  cmsIT8GetProperty(cmsHANDLE hIT8,  const char* cProp);
```

Gets a property as a literal string in current table. Memory is handled by the CGATS.17 object and should not be freed by the user.

Parameters:

hIT8: A handle to a CGATS.17 object.
cProp: A string holding property name.

Returns:

A pointer to internal block of memory containing the data for the intended property on success, NULL on error.

2.0

```
cmsFloat64Number  cmsIT8GetPropertyDbl(cmsHANDLE hIT8, const char* cProp);
```

Gets a property as a cmsFloat64Number in current table.

Parameters:

hIT8: A handle to a CGATS.17 object.
cProp: A string holding property name.

Returns:

The data for the intended property interpreted as cmsFloat64Number *on success, 0 on error.*

2.0

```
cmsUInt32Number  cmsIT8EnumProperties(cmsHANDLE cmsIT8,
                                      char ***PropertyNames);
```

Enumerates all properties in current table.

Parameters:

hIT8: A handle to a CGATS.17 object.
*PropertyNames: A pointer to a variable of type char** which will receive the table of property name strings.*

Returns:
The number of properties in current table on success, 0 on error.

2.0

```
cmsUInt32Number cmsIT8EnumPropertyMulti(cmsHANDLE hIT8,
                                        const char* cProp,
                                        const char ***SubpropertyNames)
```

Enumerates all the identifiers found in a multi-value property in current table.

Parameters:
hIT8: A handle to a CGATS.17 object.
cProp: A string holding property name
SubpropertyNames*: A pointer to a variable of type char** which will hold the table.*

Returns:
The number of identifiers found, or 0 on error.

Datasets

- Number of colums (Samples) is given by predefined property **NUMBER_OF_FIELDS**
- Number of rows (Patches) is given by predefined property **NUMBER_OF_SETS**

2.0

```
const char*   cmsIT8GetDataRowCol(cmsHANDLE cmsIT8, int row, int col);
```

Gets a cell [row, col] as a literal string in current table. This function is fast since it has not to search columns or rows by name.

Parameters:

hIT8: A handle to a CGATS.17 object.
row, col: The position of the cell.

Returns:

A pointer to internal block of memory containing the data for the intended cell on success, NULL on error.

2.0

```
cmsFloat64Number   cmsIT8GetDataRowColDbl(cmsHANDLE hIT8,
                                          int row, int col);
```

Gets a cell [row, col] as a cmsFloat64Number in current table. This function is fast since it has not to search columns or rows by name.

Parameters:

hIT8: A handle to a CGATS.17 object.
row, col: The position of the cell.

Returns:

The data for the intended cell interpreted as cmsFloat64Number *on success, 0 on error.*

2.0

```
cmsBool   cmsIT8SetDataRowCol(cmsHANDLE hIT8,
                              int row, int col,
                              const char* Val);
```

Sets a cell [row, col] as a literal string in current table. This function is fast since it has not to search columns or rows by name.

Parameters:

hIT8: A handle to a CGATS.17 object.
row, col: The position of the cell.
Val: The value to be set, as a literal string.

Returns:

TRUE on success, FALSE on error

2.0

```
cmsBool   cmsIT8SetDataRowColDbl(cmsHANDLE hIT8,
                                 int row, int col,
                                 cmsFloat64Number Val);
```

Sets a cell [Patch, Sample] as a cmsFloat64Number in current table. This function is fast since it has not to search columns or rows by name.

Parameters:

hIT8: A handle to a CGATS.17 object.
row, col: The position of the cell.
Val: The value to be set, as a cmsFloat64Number

Returns:

TRUE on success, FALSE on error

2.0

```
const char*  cmsIT8GetData(cmsHANDLE hIT8,
                           const char* cPatch,
                           const char* cSample);
```

Gets a cell [Patch, Sample] as a literal string (uncooked string) in current table. Memory is handled by the CGATS.17 object and should not be freed by the user.

Parameters:

hIT8: A handle to a CGATS.17 object.
cPatch: The intended patch name (row)
cSample: The intended sample name (column)

Returns:

A pointer to internal block of memory containing the data for the intended cell on success, NULL on error.

2.0

```
cmsFloat64Number   cmsIT8GetDataDbl(cmsHANDLE hIT8,
                                    const char* cPatch,
                                    const char* cSample);
```

Gets a cell [Patch, Sample] as a cmsFloat64Number in current table.

Parameters:

hIT8: A handle to a CGATS.17 object.
cPatch: The intended patch name (row)
cSample: The intended sample name (column)

Returns:

The data for the intended cell interpreted as cmsFloat64Number *on success, 0 on error.*

2.0

```
cmsBool   cmsIT8SetData(cmsHANDLE hIT8,
                        const char* cPatch,
                        const char* cSample,
                        const char *Val);
```

Sets a cell [Patch, Sample] as a literal string (uncooked string) in current table.

Parameters:

hIT8: A handle to a CGATS.17 object.
cPatch: The intended patch name (row)
cSample: The intended sample name (column)
Val: The value to be set, as a literal

Returns:

TRUE on success, FALSE on error

2.0

```
cmsBool   cmsIT8SetDataDbl(cmsHANDLE hIT8,
                           const char* cPatch,
                           const char* cSample,
                           cmsFloat64Number Val);
```

Sets a cell [Patch, Sample] as a cmsFloat64Number in current table.

Parameters:

hIT8: A handle to a CGATS.17 object.
cPatch: The intended patch name (row)
cSample: The intended sample name (column)
Val: The value to be set, as a cmsFloat64Number

Returns:

TRUE on success, FALSE on error

2.0

```
int    cmsIT8FindDataFormat (cmsHANDLE hIT8, const char* cSample);
```

Returns the position (column) of a given data sample name in current table. First column is 0 (SAMPLE_ID).

Parameters:

hIT8: A handle to a CGATS.17 object.

Returns:

Column number if found, -1 if not found

2.0

```
cmsBool   cmsIT8SetDataFormat(cmsHANDLE hIT8,  int n,  const char *Sample);
```

Sets column names in current table. First column is 0 (SAMPLE_ID). Special property NUMBER_OF_FIELDS must be set before calling this function.

Parameters:

hIT8: A handle to a CGATS.17 object.
n: Column to set name
Sample: Name of data

Returns:

TRUE on success, FALSE on error

2.0

```
int    cmsIT8EnumDataFormat(cmsHANDLE hIT8, char ***SampleNames);
```

Returns an array with pointers to the column names in current table. SampleNames may be NULL to get only the number of column names. Memory is associated with the CGATS.17 object, and should not be freed by the user.

Parameters:

hIT8: A handle to a CGATS.17 object.
*SampleNames: A pointer to a variable of type char** which will hold the table.*

Returns:

The number of column names in table on success, -1 on error.

2.0

```
const char*  cmsIT8GetPatchName(cmsHANDLE hIT8, int nPatch, char* buffer);
```

Fills buffer with the contents of SAMPLE_ID column for the set given in nPatch. That usually corresponds to patch name. Buffer may be NULL to get the internal memory block used by the CGATS.17 object. If specified, buffer gets a copy of such block. In this case it should have space for at least 1024 characters.

Parameters:

hIT8: A handle to a CGATS.17 object.
nPatch : set number to retrieve name
buffer: A memory buffer to receive patch name, or NULL to allow function to return internal memory block.

Returns:

A pointer to the patch name, either the user-supplied buffer or an internal memory block. NULL if error.

2.0

```
void  cmsIT8DefineDblFormat(cmsHANDLE hIT8,  const char* Formatter);
```

Sets the format string for float numbers. It uses the “C” sprintf convention. The default format string is "%.10g"

Parameters:

hIT8: A handle to a CGATS.17 object.

Returns:

None

Screening structures

cmsPRINTER_DEFAULT_SCREENS	0x0001
cmsFREQUENCE_UNITS_LINES_CM	0x0000
cmsFREQUENCE_UNITS_LINES_INCH	0x0002

Table 46

Spot Shape	
cmsSPOT_UNKNOWN	0
cmsSPOT_PRINTER_DEFAULT	1
cmsSPOT_ROUND	2
cmsSPOT_DIAMOND	3
cmsSPOT_ELLIPSE	4
cmsSPOT_LINE	5
cmsSPOT_SQUARE	6
cmsSPOT_CROSS	7

Table 47

cmsScreeningChannel	
cmsFloat64Number	Frequency;
cmsFloat64Number	ScreenAngle;
cmsUInt32Number	SpotShape;

Table 48

cmsScreening	
cmsUInt32Number	Flag;
cmsUInt32Number	nChannels;
cmsScreeningChannel	Channels[cmsMAXCHANNELS];

Table 49

Named color lists

Specialized dictionaries for dealing with named color profiles.

2.0

```
cmsNAMEDCOLORLIST* cmsAllocNamedColorList(cmsContext ContextID,
                                           cmsUInt32Number n,
                                           cmsUInt32Number ColorantCount,
                                           const char* Prefix,
                                           const char* Suffix);
```

Allocates an empty named color dictionary.

Parameters:

ContextID: Pointer to a user-defined context cargo.

N: Initial number of spot colors in the list

Colorant count: Number of channels of device space (i.e, 3 for RGB, 4 for CMYK, etc,)

Prefix, Suffix: fixed strings for all spot color names, e.g., "coated", "system", …

Returns:

A pointer to a newly created named color list dictionary on success, NULL on error.

2.0

```
void    cmsFreeNamedColorList(cmsNAMEDCOLORLIST* v);
```

Destroys a Named color list object, freeing any associated resources.

Parameters:

v: A pointer to a named color list dictionary object.

Returns:

None

2.0

```
cmsNAMEDCOLORLIST* cmsGetNamedColorList(cmsHTRANSFORM xform);
```

Retrieve a named color list from a given color transform.

Parameters:

*xform**:** Handle to a color transform object.*

Returns:

A pointer to a named color list dictionary on success, NULL on error.

2.0

```
cmsNAMEDCOLORLIST* cmsDupNamedColorList(const cmsNAMEDCOLORLIST* v);
```

Duplicates a named color list object, and all associated resources.

Parameters:

v: A pointer to a named color list dictionary object.

Returns:

A pointer to a newly created named color list dictionary on success, NULL on error.

2.0

```
cmsBool  cmsAppendNamedColor(cmsNAMEDCOLORLIST* v,
                             const char* Name,
                             cmsUInt16Number PCS[3],
                             cmsUInt16Number Colorant[cmsMAXCHANNELS]);
```

Adds a new spot color to the list. If the number of elements in the list exceeds the initial storage, the list is realloc'ed to accommodate things.

Parameters:

v: A pointer to a named color list dictionary object.
Name: The spot color name without any prefix or suffix specified in cmsAllocNamedColorList
PCS [3]: Encoded PCS coordinates.
Colorant[]: Encoded values for device colorant.

Returns:

TRUE on success, FALSE on error

2.0

```
cmsUInt32Number cmsNamedColorCount(const cmsNAMEDCOLORLIST* v);
```

Returns the number of spot colors in a named color list.

Parameters:

v: A pointer to a named color list dictionary object.

Returns:

the number of spot colors on success, 0 on error.

2.0

```
cmsInt32Number  cmsNamedColorIndex(const cmsNAMEDCOLORLIST* v,
                                    const char* Name);
```

Performs a look-up in the dictionary and returns an index on the given color name.

Parameters:

v: A pointer to a named color list dictionary object.

Returns:

Index on name, or -1 if the spot color is not found.

2.0

```
cmsBool  cmsNamedColorInfo(const cmsNAMEDCOLORLIST* NamedColorList,
                           cmsUInt32Number nColor,
                           char* Name,
                           char* Prefix,
                           char* Suffix,
                           cmsUInt16Number* PCS,
                           cmsUInt16Number* Colorant);
```

Gets extended information on a spot color, given its index. Required storage is of fixed size.

Parameters:

NamedColorList: A pointer to a named color list dictionary object.
nColor: Index to the spot color to retrieve
Name: Pointer to a 256-char array to get the name, NULL to ignore.
Prefix: Pointer to a 33-char array to get the prefix, NULL to ignore
Suffix: Pointer to a 33-char array to get the suffix, NULL to ignore.
PCS: Pointer to a 3-cmsUInt16Number to get the encoded PCS, NULL to ignore
PCS: Pointer to a 16-cmsUInt16Number to get the encoded Colorant, NULL to ignore

Returns:

TRUE on success, FALSE on error.

Profile sequences.

Profile sequence descriptors. Some fields come from profile sequence descriptor tag, others come from Profile Sequence Identifier Tag. The user is allowed to access the members of those structures. Profile sequence can be read/written by using cmsReadTag and cmsWriteTag functions.

cmsPSEQDESC	
cmsSignature	deviceMfg;
cmsSignature	deviceModel;
cmsUInt64Number	attributes;
cmsTechnologySignature	technology;
cmsProfileID	ProfileID;
cmsMLU*	Manufacturer;
cmsMLU*	Model;
cmsMLU*	Description;

Table 50

cmsSEQ	
cmsUInt32Number	n;
cmsContext	ContextID;
cmsPSEQDESC*	seq;

Table 51

2.0

```
cmsSEQ* cmsAllocProfileSequenceDescription(cmsContext ContextID,
                                           cmsUInt32Number n);
```

Creates an empty container for profile sequences.

Parameters:

ContextID: Pointer to a user-defined context cargo.

N : Number of profiles in the sequence

Returns:

A pointer to a profile sequence object on success, NULL on error.

2.0

```
cmsSEQ* cmsDupProfileSequenceDescription(const cmsSEQ* pseq);
```

Duplicates a profile sequence object, and all associated resources.

Parameters:

Pseq: A pointer to a profile sequence object.

Returns:

A pointer to a profile sequence object on success, NULL on error.

2.0

```
void    cmsFreeProfileSequenceDescription(cmsSEQ* pseq);
```

Destroys a profile sequence object, freeing all associated memory.

Parameters:

Pseq: A pointer to a profile sequence object.

Returns:

None

Multilocalized unicode management

MLU funtions are the low-level interface to access the localization features of V4 ICC profiles. *Little CMS* does offer a high-level interface for easy operation. You may want, however, handle those objects by yourself.

LanguageCode: first name language code from ISO 639-2.

http://lcweb.loc.gov/standards/iso639-2/iso639jac.html

CountryCode: first name region code from ISO 3166.

http://www.iso.ch/iso/en/prods-services/iso3166ma/index.html

```
#define  cmsNoLanguage "\0\0"
#define  cmsNoCountry    "\0\0"
#define  cmsV2Unicode    "\xff\xff"
```

2.0

```
cmsMLU*  cmsMLUalloc(cmsContext ContextID,  cmsUInt32Number nItems);
```

Allocates an empty multilocalized unicode object.

Parameters:

ContextID: Pointer to a user-defined context cargo.

Returns:

A pointer to a multilocalized unicode object on success, NULL on error.

2.0

```
void    cmsMLUfree(cmsMLU* mlu);
```

Destroys a multilocalized unicode object, freeing any associated resources.

Parameters:

mlu: a pointer to a multilocalized unicode object.

Returns:

None

2.0

```
cmsMLU*  cmsMLUdup(const cmsMLU* mlu);
```

Duplicates a multilocalized unicode object, and all associated resources.

Parameters:

mlu: a pointer to a multilocalized unicode object.

Returns:

A pointer to a multilocalized unicode object on success, NULL on error.

2.0

```
cmsBool  cmsMLUsetASCII(cmsMLU* mlu,
                              const char LanguageCode[3], const char CountryCode[3],
                              const char* ASCIIString);
```

Fills an ASCII (7 bit) entry for the given Language and country.

Parameters:

mlu: a pointer to a multilocalized unicode object.
Language Code []: Array of 3 chars describing the language
CountryCode []: Array of 3 chars describing the country
ASCIIString: String to add.

Returns:

TRUE on success, FALSE on error.

2.0

```
cmsBool  cmsMLUsetWide(cmsMLU* mlu,
                             const char LanguageCode[3], const char CountryCode[3],
                             const wchar_t* WideString);
```

Fills a UNICODE wide char (16 bit) entry for the given Language and country.

Parameters:

mlu: a pointer to a multilocalized unicode object.
Language Code []: Array of 3 chars describing the language
CountryCode []: Array of 3 chars describing the country
WideString: String to add.

Returns:

TRUE on success, FALSE on error.

2.16

```
cmsBool  cmsMLUsetUTF8(cmsMLU* mlu,
                       const char LanguageCode[3], const char CountryCode[3],
                       const char_t* UTF8String);
```

Fills a wide char (16 bit) entry for the given Language and country by converting it from UTF8 encoding.

Parameters:

mlu: a pointer to a multilocalized unicode object.
Language Code []: Array of 3 chars describing the language
CountryCode []: Array of 3 chars describing the country
UTF8String: String to add.

Returns:

TRUE on success, FALSE on error.

2.0

```
cmsUInt32Number cmsMLUgetASCII(const cmsMLU* mlu,
                                                   const char LanguageCode[3],
                                                   const char CountryCode[3],
                                                   char* Buffer, cmsUInt32Number BufferSize);
```

Gets an ASCII (7 bit) entry for the given Language and country. Set Buffer to NULL to get the required size.

Parameters:

mlu: a pointer to a multilocalized unicode object.
Language Code []: Array of 3 chars describing the language
CountryCode []: Array of 3 chars describing the country
Buffer: Pointer to a char buffer
BufferSize: Size of given buffer.

Returns:

Number of bytes read into buffer.

2.0

```
cmsUInt32Number cmsMLUgetWide(const cmsMLU* mlu,
                                                        const char LanguageCode[3],
                                                        const char CountryCode[3],
                                                         wchar_t* Buffer,
                                                         cmsUInt32Number BufferSize);
```

Gets a UNICODE wchar_t (16 bit) entry for the given Language and country. Set Buffer to NULL to get the required size.

Parameters:

mlu: a pointer to a multilocalized unicode object.
Language Code []: Array of 3 chars describing the language
CountryCode []: Array of 3 chars describing the country
Buffer: Pointer to a wchar_t buffer
BufferSize: Size of given buffer.

Returns:

Number of bytes read into buffer.

2.16

```
cmsUInt32Number cmsMLUgetUTF8(const cmsMLU* mlu,
                              const char LanguageCode[3],
                              const char CountryCode[3],
                              char* Buffer,
                              cmsUInt32Number BufferSize);
```

Gets a UTF8 entry for the given Language and country. Set Buffer to NULL to get the required size.

Parameters:

mlu: a pointer to a multilocalized unicode object.
Language Code []: Array of 3 chars describing the language
CountryCode []: Array of 3 chars describing the country
Buffer: Pointer to a wchar_t buffer
BufferSize: Size of given buffer.

Returns:

Number of bytes read into buffer.

2.0

```
cmsBool cmsMLUgetTranslation(const cmsMLU* mlu,
                             const char LanguageCode[3],
                             const char CountryCode[3],
                             char ObtainedLanguage[3],
                             char ObtainedCountry[3]);
```

Obtains the translation rule for given multilocalized unicode object.

Parameters:

mlu: a pointer to a multilocalized unicode object.
Language Code []: Array of 3 chars describing the language
CountryCode []: Array of 3 chars describing the country
ObtainedLanguage []: Array of 3 chars to get the language translation.
ObtainedCode []: Array of 3 chars to get the country translation.

Returns:

TRUE on success, FALSE on error

2.5

```
cmsUInt32Number   cmsMLUtranslationsCount(const cmsMLU* mlu);
```

Obtains the number of true translations stored in a given multilocalized unicode object.

Parameters:

mlu: a pointer to a multilocalized unicode object.

Returns:

Number of translations on success, 0 on error.

2.5

```
cmsBool  cmsMLUtranslationsCodes(const cmsMLU* mlu,
                                             cmsUInt32Number idx,
                                             char LanguageCode[3],
                                             char CountryCode[3]);
```

Obtains the translation codes for a true translation stored in a given multilocalized unicode object.

Parameters:

mlu: a pointer to a multilocalized unicode object.

idx: index to the true translation to retrieve info. 0-based.

Language Code []: Array of 3 chars to store the code describing the language

CountryCode []: Array of 3 chars to store the code describing the country

Returns:

TRUE on success, FALSE on error

Dictionary

This is a simple linked list used to store pairs Name-Value for the dictionary meta-tag, as described in ICC spec 4.4

```
typedef struct _cmsDICTentry_struct {

    struct _cmsDICTentry_struct* Next;

    cmsMLU *DisplayName;
    cmsMLU *DisplayValue;
    wchar_t* Name;
    wchar_t* Value;

} cmsDICTentry;
```

2.2

```
cmsHANDLE   cmsDictAlloc(cmsContext ContextID);
```

Allocates an empty dictionary linked list object.

Parameters:

ContextID: Pointer to a user-defined context cargo.

Returns:

On success, a handle to a newly created dictionary linked list. NULL on error.

2.2

```
void  cmsDictFree(cmsHANDLE hDict);
```

Destroys a dictionary linked list object, freeing any associated resource.

Parameters:

hDict: Handle to a dictionary linked list object.

Returns:

None

2.2

```
cmsHANDLE   cmsDictDup(cmsHANDLE hDict);
```

Duplicates a dictionary linked list object.

Parameters:

hDict: Handle to a dictionary linked list object.

Returns:

On success, a handle to a newly created dictionary linked list object. On error, NULL.

2.2

```
cmsBool  cmsDictAddEntry(cmsHANDLE hDict,
                         const wchar_t* Name, const wchar_t* Value,
                         const cmsMLU *DisplayName,
                         const cmsMLU *DisplayValue);
```

Adds data to a dictionary linked list object. No check for duplicity is made. Dictionary and Name parameters a required, rest is optional an NULL may be used.

Parameters:

hDict: Handle to a dictionary linked list object.
Name, Value: Wide char strings. Value may be NULL
DisplayName, Display Value: Multilocalized Unicode objects. May be NULL.

Returns:

Operation result

2.2

```
const cmsDICTentry* cmsDictGetEntryList(cmsHANDLE hDict)
```

Returns a pointer to first element in linked list.

Parameters:

hDict: Handle to a dictionary linked list object.

Returns:

Pointer to element on success, NULL on error or end of list.

2.2

```
const cmsDICTentry* cmsDictNextEntry (const cmsDICTentry* e)
```

Returns a pointer to the next element in linked list.

Parameters:

e: Pointer to element

Returns:

Pointer to element on success, NULL on error or end of list.

Tone curves

Tone curves are powerful constructs that can contain curves specified in diverse ways. The curve is stored in segments, where each segment can be sampled or specified by parameters. A 16 bit simplification of the *whole* curve is kept for optimization purposes. For float operation, each segment is evaluated separately. Plug-ins may be used to define new parametric schemes.

2.0

```
cmsFloat32Number cmsEvalToneCurveFloat(const cmsToneCurve* Curve,
                                       cmsFloat32Number v);
```

Evaluates the given floating-point number across the given tone curve.

Parameters:

Curve: pointer to a tone curve object.
V: floating point number to evaluate

Returns:

Operation result

2.0

```
cmsUInt16Number   cmsEvalToneCurve16(const cmsToneCurve* Curve,
                                     cmsUInt16Number v);
```

Evaluates the given 16-bit number across the given tone curve. This function is significantly faster than cmsEvalToneCurveFloat, since it uses a pre-computed 16-bit lookup table.

Parameters:

Curve: pointer to a tone curve object.
V: 16 bit Number to evaluate

Returns:

Operation result

Parametric curves

See a table of built-in types below. User can increase the number of available types by using a proper plug-in. Parametric curves allow 10 parameters at most.

Function	Number	Parameters	Comment
$Y = X^{\gamma}$	1	γ	
$Y = (aX + b)^{\gamma} \quad \left(X \geq -\frac{b}{a}\right)$ $Y = 0 \quad \left(X < -\frac{b}{a}\right)$	2	γ a b	CIE 122-1966
$Y = (aX + b)^{\gamma} + c \quad \left(X \geq -\frac{b}{a}\right)$ $Y = c \quad \left(X < -\frac{b}{a}\right)$	3	γ a b c	IEC 61966-3
$Y = (aX + b)^{\gamma} \quad (X \geq d)$ $Y = cX \quad (X < d)$	4	γ a b c d	IEC 61966-2.1 (sRGB)
$Y = (aX + b)^{\gamma} + e \quad (X \geq d)$ $Y = (cX + f) \quad (X < d)$	5	γ a b c d e f	
$Y = (aX + b)^{\gamma} + c$	6	γ a b c	Identical to 5, unbounded.
$Y = a\log(b\,X^{\gamma} + c) + d$	7	γ a b c d	
$Y = ab^{(cX+d)} + e$	8	a b c d e	
$Y = (1 - (1 - X)^{1/\gamma})^{1/\gamma}$	108	γ	S-Shaped sigmoidal (deprecated do not use)
$Yb(k,\gamma) = \frac{1}{1 + e^{-k\gamma}} - \frac{1}{2}$ $C = \frac{1}{2\,Yb(k,1)}$ $Y(k) = C * Yb(k, 2\gamma - 1) + 0.5$	109	γ	Centered Sigmoid.

Table 52

2.0

```
cmsToneCurve*   cmsBuildParametricToneCurve(cmsContext ContextID,
                                            cmsInt32Number Type,
                                            const cmsFloat64Number Params[]);
```

Builds a parametric tone curve according Table 52

Parameters:

ContextID: Pointer to a user-defined context cargo.
Type: Number of parametric tone curve, according to Table 52 *for built-in, or other if tone-curve plug-in is being used.*
Params[10]: Array of tone curve parameters, according to Table 52 *for built-in, or other if tone-curve plug-in is being used.*

Returns:

Pointer to a newly created tone curve object on success, NULL on error.

2.0

```
cmsToneCurve*   cmsBuildGamma(cmsContext ContextID,
                              cmsFloat64Number Gamma);
```

Simplified wrapper to cmsBuildParametricToneCurve. Builds a parametric curve of type 1.

Parameters:

ContextID: Pointer to a user-defined context cargo.
Gamma: Value of gamma exponent

Returns:

Pointer to a newly created tone curve object on success, NULL on error.

2.16

```
const cmsCurveSegment* cmsGetToneCurveSegment(cmsInt32Number segment,
                                              const cmsToneCurve* t);
```

Returns a pointer to the segment structure stored into the tone curve object. Interpretation of the meaning and number of parameters depends of the parametric type, as described in Table 52 Please note more parametric curve types can be added across plug-ins.

If the tone curve has not the asked segment, the function returns NULL.

This function can be used to iterate tone curves to get all segments and types.

```
segment_num=0;
do {
        Seg = cmsGetToneCurveSegment(segment_num, curve);
        If (Seg == NULL) break;
        segment_num++;

        [ do whatever you wish with segment, type 0 means tabulated ]

} while (1);
```

Parameters:

Curve: pointer to a tone curve object.

Returns:

Pointer to the internal segment structure.

Segmented curves

Segmented curves are formed by several segments. This structure describes a curve segment.

cmsCurveSegment	
cmsFloat32Number x0, x1;	Domain; for x0 < x <= x1
cmsInt32Number Type;	Parametric type, Type == 0 means sampled segment. Negative values are reserved
cmsFloat64Number Params[10];	Parameters if Type != 0
cmsUInt32Number nGridPoints;	Number of grid points if Type == 0
cmsUInt32Number* SampledPoints;	Points to an array of floats if Type == 0

Table 53

2.0

```
cmsToneCurve*  cmsBuildSegmentedToneCurve(cmsContext ContextID,
                                          cmsInt32Number nSegments,
                                          const cmsCurveSegment Segments[]);
```

Builds a tone curve from given segment information.

Parameters:

ContextID: Pointer to a user-defined context cargo.
nSegments: Number of segments
Segments[]: Array of structures described in Table 53

Returns:

Pointer to a newly created tone curve object on success, NULL on error.

Tabulated curves

2.0

```
cmsToneCurve*  cmsBuildTabulatedToneCurve16(cmsContext ContextID,
                                             cmsInt32Number nEntries,
                                             const cmsUInt16Number values[]);
```

Builds a tone curve based on a table of 16-bit values. Tone curves built with this function are restricted to 0…1.0 domain.

Parameters:

ContextID: Pointer to a user-defined context cargo.
nEntries: Number of sample points
values []: Array of samples. Domain is 0…65535.

Returns:

Pointer to a newly created tone curve object on success, NULL on error.

2.0

```
cmsToneCurve*    cmsBuildTabulatedToneCurveFloat(cmsContext ContextID,
                                                 cmsUInt32Number nEntries,
                                                 const cmsFloat32Number  values[]);
```

Builds a tone curve based on a table of floating point values. Tone curves built with this function are **not** restricted to 0…1.0 domain.

Parameters:

ContextID: Pointer to a user-defined context cargo.
nEntries: Number of sample points
values []: Array of samples. Domain of samples is 0…1.0

Returns:

Pointer to a newly created tone curve object on success, NULL on error.

Curve handling

2.0

```
void cmsFreeToneCurve(cmsToneCurve* Curve);
```

Destroys a tone curve object, freeing any associated resource.

Parameters:

Curve: pointer to a tone curve object.

Returns:

None

2.0

```
void    cmsFreeToneCurveTriple(cmsToneCurve* Curves[3]);
```

Destroys tree tone curve object placed into an array. This function is equivalent to call three times cmsFreeToneCurve, one per object. It exists because conveniency.

Parameters:

Curves []: array to 3 pointers to tone curve objects.

Returns:

None

2.0

```
cmsToneCurve*   cmsDupToneCurve(const cmsToneCurve* Src);
```

Duplicates a tone curve object, and all associated resources.

Parameters:

Src: pointer to a tone curve object.

Returns:

Pointer to a newly created tone curve object on success, NULL on error.

2.0

```
cmsToneCurve*  cmsReverseToneCurve(const cmsToneCurve* InGamma);
```

Creates a tone curve that is the inverse f^{-1} of given tone curve.

Parameters:

InGamma: pointer to a tone curve object.

Returns:

Pointer to a newly created tone curve object on success, NULL on error.

2.0

```
cmsToneCurve*  cmsReverseToneCurveEx(cmsInt32Number nResultSamples,
                                      const cmsToneCurve* InGamma);
```

Creates a tone curve that is the inverse f^{-1} of given tone curve. In the case it couldn't be analytically reversed, a tablulated curve of nResultSamples is created.

Parameters:

nResultSamples: Number of samples to use in the case origin tone curve couldn't be analytically reversed
InGamma: pointer to a tone curve object.

Returns:

Pointer to a newly created tone curve object on success, NULL on error.

2.0

```
cmsToneCurve*    cmsJoinToneCurve(cmsContext ContextID,
                                  const cmsToneCurve* X,
                                  const cmsToneCurve* Y,
                                  cmsUInt32Number nPoints);
```

Composites two tone curves in the form $Y^{-1}(X(t))$

Parameters:

ContextID: Pointer to a user-defined context cargo.
X, Y : Pointers to tone curve objects.
nPoints: Sample rate for resulting tone curve.

Returns:

Pointer to a newly created tone curve object on success, NULL on error.

2.0

```
cmsBool   cmsSmoothToneCurve(cmsToneCurve* Tab,
                             cmsFloat64Number lambda);
```

Smoothes tone curve according to the lambda parameter. From: Eilers, P.H.C. (1994) Smoothing and interpolation with finite differences. in: Graphic Gems IV, Heckbert, P.S. (ed.), Academic press.

Parameters:

Tab: pointer to a tone curve object.
Lambda: degree of smoothing (

Returns:

TRUE on success, FALSE on error

Information on tone curve functions

Those functions do return information or estimations about given tone curves.

2.0

```
cmsBool  cmsIsToneCurveMultisegment(const cmsToneCurve* InGamma);
```

Returns TRUE if the tone curve contains more than one segment, FALSE if it has only one segment.

Parameters:

InGamma: pointer to a tone curve object.

Returns:

TRUE or FALSE.

2.0

```
cmsBool cmsIsToneCurveLinear(const cmsToneCurve* Curve);
```

Returns an estimation of cube being an identity (1:1) in the [0..1] domain. Does not take unbounded parts into account. This is just a coarse approximation, with no mathematical validity.

Parameters:

Curve: pointer to a tone curve object.

Returns:

TRUE or FALSE.

2.0

```
cmsBool cmsIsToneCurveMonotonic(const cmsToneCurve* t);
```

Returns an estimation of monotonicity of curve in the [0..1] domain. Does not take unbounded parts into account. This is just a coarse approximation, with no mathematical validity.

Parameters:

t: pointer to a tone curve object.

Returns:

TRUE or FALSE.

2.0

```
cmsBool  cmsIsToneCurveDescending(const cmsToneCurve* t);
```

Returns TRUE if $(0) > f(1)$, FALSE otherwise. Does not take unbounded parts into account.

Parameters:

t: pointer to a tone curve object.

Returns:

TRUE or FALSE.

2.0

```
cmsFloat64Number   cmsEstimateGamma(const cmsToneCurve* t,
                                     cmsFloat64Number Precision);
```

Estimates the apparent gamma of the tone curve by using least squares fitting to a pure exponential expression in the $f(x) = x^{\gamma}$. The parameter γ is estimated at the given precision.

Parameters:

t: pointer to a tone curve object.
Precision: The maximum standard deviation allowed on the residuals, 0.01 is a fair value, set it to a big number to fit any curve, mo matter how good is the fit.

Returns:

The estimated gamma at given precision, or -1.0 if the fitting has less precision.

2.4

```
cmsUInt32Number  cmsGetToneCurveEstimatedTableEntries (const cmsToneCurve* t);
```

Tone curves do maintain a shadow low-resolution tabulated representation of the curve. This function returns the number of entries such table has.

Parameters:

t: pointer to a tone curve object.

Returns:

The number of entries for the internal table estimating the curve.

2.4

```
cmsUInt16Number*   cmsGetToneCurveEstimatedTable(const cmsToneCurve* t);
```

Tone curves do maintain a shadow low-resolution tabulated representation of the curve. This function returns a pointer to this table.

Parameters:

t: pointer to a tone curve object.

Returns:

A pointer to the estimation table, which has 16-bit precision.

Pipelines

Pipelines are a convenient way to model complex operations on image data. Each pipeline may contain an arbitrary number of **stages.** Each stage performs a single operation. Pipelines may be optimized to be executed on a certain format (8 bits, for example) and can be saved as LUTs in ICC profiles.

2.0

```
cmsPipeline*  cmsPipelineAlloc(cmsContext ContextID,
                               cmsUInt32Number InputChannels,
                               cmsUInt32Number OutputChannels);
```

Allocates an empty pipeline. Final Input and output channels must be specified at creation time.

Parameters:

ContextID: Pointer to a user-defined context cargo.
InputChannels, OutputChannels: Number of channels on input and output.

Returns:

A pointer to a pipeline on success, NULL on error.

2.0

```
void   cmsPipelineFree(cmsPipeline* lut);
```

Frees a pipeline and all owned stages.

Parameters:

lut: Pointer to a pipeline object.

Returns:

None

2.0

```
cmsPipeline*   cmsPipelineDup(const cmsPipeline* Orig);
```

Duplicates a pipeline object, and all associated resources.

Parameters:

Orig: Pointer to a pipeline object.

Returns:

A pointer to a pipeline on success, NULL on error.

2.0

```
cmsBool    cmsPipelineCat(cmsPipeline* l1, const cmsPipeline* l2);
```

Appends pipeline l2 at the end of pipeline l1. Channel count must match.

Parameters:

l1, l2: Pointer to a pipeline object.

Returns:

TRUE on success, FALSE on error.

2.0

```
void cmsPipelineEvalFloat(const cmsFloat32Number In[],
                                    cmsFloat32Number Out[],
                                    const cmsPipeline* lut);
```

Evaluates a pipeline using floating point numbers.

Parameters:

In[]: Input values.
Out[]: Output values.
lut: Pointer to a pipeline object.

Returns:

None

2.0

```
void    cmsPipelineEval16(const cmsUInt16Number In[],
                                    cmsUInt16Number Out[],
                                    const cmsPipeline* lut);
```

Evaluates a pipeline usin 16-bit numbers, optionally using the optimized path.

Parameters:

In[]: Input values.
Out[]: Output values.
lut: Pointer to a pipeline object.

Returns:

None

2.0

```
cmsBool   cmsPipelineEvalReverseFloat(cmsFloat32Number Target[],
                                      cmsFloat32Number Result[],
                                      cmsFloat32Number Hint[],
                                      const cmsPipeline* lut);
```

Evaluates a pipeline in the reverse direction, using Newton's method.

Parameters:

Target[]: Input values.
Result[]: Output values.
Hint[]: Where begin the search
lut: Pointer to a pipeline object.

Returns:

TRUE on success, FALSE on error.

2.0

```
cmsUInt32Number    cmsPipelineInputChannels(const cmsPipeline* lut);
```

Returns the number of input channels of a given pipeline.

Parameters:

lut: Pointer to a pipeline object.

Returns:

Number of channels on success, 0 on error.

2.0

```
cmsUInt32Number    cmsPipelineOutputChannels(const cmsPipeline* lut);
```

Returns number of output channels of a given pipeline.

Parameters:

lut: Pointer to a pipeline object.

Returns:

Number of channels on success, 0 on error.

2.0

```
cmsUInt32Number   cmsPipelineStageCount(const cmsPipeline* lut);
```

Returns number of stages of a given pipeline.

Parameters:

lut: Pointer to a pipeline object.

Returns:

Number of stages of pipeline.

2.0

```
void   cmsPipelineInsertStage(cmsPipeline* lut, cmsStageLoc loc, cmsStage* mpe);
```

Inserts a stage on either the head or the tail of a given pipeline. Note that no duplication of mpe structures is done, this function only adds a *reference* of mpe in the pipeline linked list. You cannot free the mpe object after using this function.

Parameters:

lut: Pointer to a pipeline object.
Loc: enumerated constant, either **cmsAT_BEGIN** or **cmsAT_END**
Mpe: Pointer to a stage object

Returns:

None

2.0

```
void  cmsPipelineUnlinkStage(cmsPipeline* lut, cmsStageLoc loc, cmsStage** mpe);
```

Removes the stage from the pipeline. Additionally it can grab the stage **without freeing it**. To do so, caller must specify a variable to receive a pointer to the stage being unlinked. If mpe is NULL, the stage is then removed and freed.

Parameters:

lut: Pointer to a pipeline object.
Loc: enumerated constant, either **cmsAT_BEGIN** or **cmsAT_END**
mpe: Pointer to a variable to receive a pointer to the stage object being unlinked. NULL to free the resource automatically.

Returns:

None

2.0

```
cmsStage*  cmsPipelineGetPtrToFirstStage(const cmsPipeline* lut);
```

Get a pointer to the first stage in the pipeline, or NULL if pipeline is empty. Intended for iterators.

Parameters:

lut: Pointer to a pipeline object.

Returns:

A pointer to a pipeline stage on success, NULL on empty pipeline.

2.0

```
cmsStage*  cmsPipelineGetPtrToLastStage(const cmsPipeline* lut);
```

Get a pointer to the last stage in the pipeline, or NULL if pipeline is empty. Intended for iterators.

Parameters:

lut: Pointer to a pipeline object.

Returns:

A pointer to a pipeline stage on success, NULL on empty pipeline.

2.0

```
cmsStage* cmsStageNext(const cmsStage* mpe);
```

Returns next stage in pipeline list, or NULL if end of list. Intended for iterators.

Parameters:

mpe: a pointer to the actual stage object.

Returns:

A pointer to the next stage in pipeline or NULL on end of list.

2.0

```
cmsBool   cmsPipelineCheckAndRetreiveStages(const cmsPipeline* Lut,
                                                cmsUInt32Number n,  ... );
```

This function is quite useful to analyze the structure of a Pipeline and retrieve the Stage elements that conform the Pipeline. It should be called with the Pipeline, the number of expected stages and then a list of expected types followed with a list of double pointers to Stage elements. If the function founds a match with current pipeline, it fills the pointers and returns TRUE if not, returns FALSE without touching anything.

Parameters:

Lut: Pointer to a pipeline object.
N: Number of expected stages
…: list of types followed by a list of pointers to variables to receive pointers to stage elements

Returns:

TRUE on success, FALSE on error.

2.0

```
cmsBool   cmsPipelineSetSaveAs8bitsFlag(cmsPipeline* lut, cmsBool On);
```

Sets an internal flag that marks the pipeline to be saved in 8 bit precision. By default, all pipelines are saved on 16 bits precision on AtoB/BToA tags and in floating point precision on DToB/BToD tags.

Parameters:

lut: Pointer to a pipeline object.
On: State of the flag, TRUE=Save as 8 bits, FALSE=Save as 16 bits

Returns:

TRUE on success, FALSE on error

Stage functions

Stages are single-step operations that can be chained to create pipelines. Actual stage types does include matrices, tone curves, Look-up interpolation and user-defined. There are functions to create new stage types and a plug-in type to allow stages to be saved in multi profile elements tag types. See the plug-in API for further details.

2.0

```
cmsStage*   cmsStageAllocIdentity(cmsContext ContextID,
                                          cmsUInt32Number nChannels);
```

Creates an empty (identity) stage that does no operation. May be needed in order to save the pipeline as AToB/BToA tags in ICC profiles.

Parameters:

ContextID: Pointer to a user-defined context cargo.

nChannels: Number of channels

Returns:

A pointer to a pipeline stage on success, NULL on error.

2.0

```
cmsStage*   cmsStageAllocToneCurves(cmsContext ContextID,
                                            cmsUInt32Number nChannels,
                                            cmsToneCurve* const Curves[]);
```

Creates a stage that contains nChannels tone curves, one per channel. Setting Curves to NULL forces identity (1:1) curves to be used. The stage keeps and owns a private copy of the tone curve objects.

Parameters:

ContextID: Pointer to a user-defined context cargo.

nCurves: Number of Channels of stage

Curves[] : Array of tone curves objects, one per channel.

Returns:

A pointer to a pipeline stage on success, NULL on error.

2.0

```
cmsStage*  cmsStageAllocMatrix(cmsContext ContextID,
                               cmsUInt32Number Rows,  cmsUInt32Number Cols,
                               const cmsFloat64Number* Matrix,
                               const cmsFloat64Number* Offset);
```

Creates a stage that contains a matrix plus an optional offset. out = matrix * in + offset Note that Matrix is specified in double precision, whilst CLUT has only float precision. That is because an ICC profile can encode matrices with far more precision that CLUTS.

Parameters:

ContextID: Pointer to a user-defined context cargo.
Rows, Cols: Dimensions of matrix
Matrix []: Points to a matrix of [Rows, Cols] ***Row major***
Offset[]: Points to a vector of [Rows], NULL if no offset is to be applied.

Returns:

A pointer to a pipeline stage on success, NULL on error.

Note: For plug-in writers, the order of matrix and offset is different from VEC3 and MAT3 since those are unrelated types, used only in plug-ins.

2.0

```
cmsStage*  cmsStageAllocCLut16bit(cmsContext ContextID,
                                  cmsUInt32Number nGridPoints,
                                  cmsUInt32Number inputChan,
                                  cmsUInt32Number outputChan,
                                  const cmsUInt16Number* Table);
```

Creates a stage that contains a 16 bits multidimensional lookup table (CLUT). Each dimension has same resolution. The CLUT can be initialized by specifying values in *Table* parameter. The recommended way is to set Table to NULL and use *cmsStageSampleCLut16bit* with a callback, because this way the implementation is independent of the selected number of grid points.

The CLUT is organized as an i-dimensional array with a given number of grid points in each dimension, where i is the number of input channels in the table. The dimension corresponding to the first input channel varies least rapidly and the dimension corresponding to the last input channel varies most rapidly. Each grid point value is an o-byte array, where o is the number of output channels. The first sequential byte of the entry contains the function value for the first output function, the second sequential byte of the entry contains the function value for the second output function, and so on until all the output functions have been supplied. Each byte in the CLUT is appropriately normalized to the range 0 to 255.

Parameters:

ContextID: Pointer to a user-defined context cargo.

nGridPoints: the number of nodes (same for each component).
inputChan: Number of input channels.
outputChan: Number of output channels.
Table: a pointer to a table of cmsUInt16Number, holding initial values for nodes. If NULL the CLUT is initialized to zero.

Returns:

A pointer to a pipeline stage on success, NULL on error.

2.0

```
cmsStage* cmsStageAllocCLutFloat(cmsContext ContextID,
                                 cmsUInt32Number nGridPoints,
                                 cmsUInt32Number inputChan,
                                 cmsUInt32Number outputChan,
                                 const cmsFloat32Number * Table);
```

Creates a stage that contains a float multidimensional lookup table (CLUT). Each dimension has same resolution. The CLUT can be initialized by specifying values in *Table* parameter. The recommended way is to set Table to NULL and use *cmsStageSampleCLutFloat* with a callback, because this way the implementation is independent of the selected number of grid points.

Parameters:

ContextID: Pointer to a user-defined context cargo.

nGridPoints: the number of nodes (same for each component).
inputChan: Number of input channels.
outputChan: Number of output channels.
Table: a pointer to a table of cmsFloat32Number, holding initial values for nodes. If NULL the CLUT is initialized to zero.

Returns:

A pointer to a pipeline stage on success, NULL on error.

2.0

```
cmsStage*  cmsStageAllocCLut16bitGranular(cmsContext ContextID,
                                          const cmsUInt32Number clutPoints[],
                                                cmsUInt32Number inputChan,
                                                cmsUInt32Number outputChan,
                                          const cmsUInt16Number* Table);
```

Similar to cmsStageAllocCLut16bit, but it allows different granularity on each CLUT dimension.

Parameters:

ContextID: Pointer to a user-defined context cargo.

ContextID: Pointer to a user-defined context cargo.

clutPoints[]: Array [inputChan] holding the number of nodes for each component.
inputChan: Number of input channels.
outputChan: Number of output channels.
Table: a pointer to a table of cmsUInt16Number, holding initial values for nodes. If NULL the CLUT is initialized to zero.

Returns:

A pointer to a pipeline stage on success, NULL on error.

2.0

```
cmsStage* cmsStageAllocCLutFloatGranular(cmsContext ContextID,
                                         const cmsUInt32Number clutPoints[],
                                         cmsUInt32Number inputChan,
                                         cmsUInt32Number outputChan,
                                         const cmsFloat32Number * Table);
```

Similar to *cmsStageAllocCLutFloat*, but it allows different granularity on each CLUT dimension.

Parameters:

ContextID: Pointer to a user-defined context cargo.
clutPoints[]: Array [inputChan] holding the number of nodes for each component.
inputChan: Number of input channels.
outputChan: Number of output channels.
Table: a pointer to a table of cmsFloat32Number, holding initial values for nodes.

Returns:

A pointer to a pipeline stage on success, NULL on error.

2.0

```
cmsStage*  cmsStageDup(cmsStage* mpe);
```

Duplicates a pipeline stage and all associated resources.

Parameters:

Mpe: a pointer to the stage to be duplicated.

Returns:

A pointer to a pipeline stage on success, NULL on error.

2.0

```
void    cmsStageFree(cmsStage* mpe);
```

Destroys a pipeline stage object, freeing any associated resources. The stage should first be unlinked from any pipeline before proceeding to free it.

Parameters:

mpe: a pointer to a stage object.

Returns:

None

2.0

```
cmsUInt32Number    cmsStageInputChannels(const cmsStage* mpe);
```

Returns the number of input channels of a given stage object.

Parameters:

mpe: a pointer to a stage object.

Returns:

Number of input channels of pipeline stage object.

2.0

```
cmsUInt32Number    cmsStageOutputChannels(const cmsStage* mpe);
```

Returns the number of output channels of a given stage object.

Parameters:

mpe: a pointer to a stage object.

Returns:

Number of output channels of pipeline stage object.

2.0

```
cmsStageSignature  cmsStageType(const cmsStage* mpe);
```

Returns the type of a given stage object, enumerated in Table 31

Parameters:

mpe: a pointer to a stage object.

Returns:

The type of a given stage object, enumerated in Table 31

2.13

```
cmsContext  cmsGetStageContextID(const cmsStage* mpe);
```

Returns the context of a given stage object

Parameters:
mpe: a pointer to a stage object.

Returns:
The context of a given stage object

2.16

```
void*  cmsStageData(const cmsStage* mpe);
```

Returns a pointer to the internal data stored in the context structure. This structure varies according of the stage type. See typedefs for all core structures in lcms2_plugin.h

```
_cmsStageToneCurvesData
_cmsStageMatrixData
_cmsStageCLutData
```

Parameters:
mpe: a pointer to a stage object.

Returns:
The context of a given stage object

Notes:
Plug-ins may define new data types

Sampling CLUT

Those functions are provided to populate CLUT stages in a way that is independent of the number of nodes. The programmer has to provide a callback that will be invoked on each CLUT node. LittleCMS does fill the In[] parameter with the coordinates that addresses the node. It also fills the Out[] parameter with CLUT contents on the node, so this can be used also to get CLUT contents after reading it from an ICC profile. In this case, a special flag can be specified to make sure the CLUT is being accessed as read-only and not modified.

```
typedef cmsInt32Number (* cmsSAMPLER16)
                        (CMSREGISTER const cmsUInt16Number In[],
                         CMSREGISTER cmsUInt16Number Out[],
                         CMSREGISTER void * Cargo);
```

```
typedef cmsInt32Number (* cmsSAMPLERFLOAT)
                       (CMSREGISTER const cmsFloat32Number In[],
                        CMSREGISTER cmsFloat32Number Out[],
                        CMSREGISTER void * Cargo);
```

Use this flag to prevent changes being written to destination.

```
#define SAMPLER_INSPECT     0x01000000
```

2.0

```
cmsBool cmsStageSampleCLut16bit(cmsStage* mpe,
                                cmsSAMPLER16 Sampler,
                                void* Cargo,
                                cmsUInt32Number dwFlags);
```

Iterate on all nodes of a given CLUT stage, calling a 16-bit sampler on each node.

Parameters:

mpe: a pointer to a CLUT stage object.
Sampler: 16 bit callback to be executed on each node.
Cargo: Points to a user-supplied data which be transparently passed to the callback.
dwFlags: Bit-field flags for different options. Only SAMPLER_INSPECT is currently supported.

Returns:

TRUE on success, FALSE on error.

2.0

```
cmsBool cmsStageSampleCLutFloat(cmsStage* mpe,
                                 cmsSAMPLERFLOAT Sampler,
                                 void* Cargo,
                                 cmsUInt32Number dwFlags);
```

Parameters:

mpe: a pointer to a CLUT stage object.
Sampler: Floating point callback to be executed on each node.
Cargo: Points to a user-supplied data which be transparently passed to the callback.
dwFlags: Bit-field flags for different options. Only SAMPLER_INSPECT is currently supported.

Returns:

TRUE on success, FALSE on error.

Slicing space functions

Those functions do slice a multidimensional space into equally spaced steps and then executes a callback on each division. Each component may be divided into different slices (granularity). The sampler is identical to the callback used in cmsStageSampleCLut16bit, but ***out*** parameter comes set to NULL since there is no table to populate. The callback type is described in the above paragraph.

2.0

```
cmsBool cmsSliceSpace16(cmsUInt32Number nInputs,
                        const cmsUInt32Number clutPoints[],
                         cmsSAMPLER16 Sampler, void * Cargo);
```

Slices target space executing a 16 bits callback of type cmsSAMPLER16.

Parameters:

nInputs: Number of components in target space.
clutPoints[]: Array [nInputs] holding the division slices for each component.
Sampler: 16 bit callback to execute on each slice.
Cargo: Points to a user-supplied data which be transparently passed to the callback.

Returns:

TRUE on success, FALSE on error.

2.0

```
cmsBool cmsSliceSpaceFloat(cmsUInt32Number nInputs,
                           const cmsUInt32Number clutPoints[],
                           cmsSAMPLERFLOAT Sampler, void * Cargo);
```

Slices target space executing a floating point callback of type cmsSAMPLERFLOAT.

Parameters:

nInputs: Number of components in target space.
clutPoints[]: Array [nInputs] holding the division slices for each component.
Sampler: Floating point callback to execute on each slice.
Cargo: Points to a user-supplied data wich be transparently passed to the callback.

Returns:

TRUE on success, FALSE on error.

Conclusion

The canonical site for *Little CMS* is https://www.littlecms.com, for suggestions and bug reporting, please contact me at: info@littlecms.com

You can get additional information on *Little CMS* in the documents listed below:

- *Little CMS* Tutorial
- *Little CMS* Plug-In API

Thank you for using this Little Color Management System.