

Little CMS2

How to use the engine in your applications
2.20

---

https://www.littlecms.com

Copyright © 2026 Marti Maria Saguer, all rights reserved.

Table of Contents

Introduction 4
Documentation 5
Requeriments 6
Include files 6
Basic Concepts 7
Source code conventions 7
The const keyword 7
The register keyword 7
Basic Types 8
Step-by-step Example 9
Open the profiles 10
Identify the intended format of pixels 10
Create the transform 12
Rendering intents 12
Optimization 14
Apply the transform 14
Scanline padding 14
Scanline overlap 14
Finishing the color transform 15
Special profile types 16
Embedded profiles 16
Device-link profiles 16
Named color profiles 18
Built-in profiles 20
On-the-fly profiles 21
Proofing 25
Black point compensation 27
Black preserving intents 29
Linear spaces 30
Error Logging 31
Getting information from profiles 33

Textual information 33
Reading the unicode variant on V2 profiles 34
Profile header fields 35
Profile Directory 36
Profile capabilities 36
Reading tags 37
Writing tags 38
The linked tag feature 38
Creating new profiles 38
Tone curves 39
Pipelines and Multi Processing elements 41
CLUT element 43
Matrix element 43
Curve set element 43
Additions and the processing element plug-in 43
Helper functions 44
Colorimetric space conversions 44
Converting encoded values 44
Linear Bradford Chromatic Adaptation 45
Color difference functions 46
Delta-E metrics 46
PostScript generation 48
CIECAM02 50
CGATS parser 53
Overview 53
Memory management 53
Additions 53
Strict CGATS 54
Gamut boundary description 55
Conclusion 57

Introduction

Welcome to the Little Color Management System. With this library you can enable your applications to use International Color Consortium (ICC) profiles. Little CMS does accept profiles conformant with ICC 4.3 or below, and supports all features described in the ICC specification. Little CMS can operate with old V2 ICC profiles as well. The CMM does all necessary adjustments and allows you to operate and mix both kind of profiles.





This file has been written to present the Little CMS core library to “would-be” writers of applications. It first describes the concepts on which the engine is based, and then how to use it to obtain transformations, colorspace conversions and other color-related functionality. This document doesn't even try to explain the basic concepts of color management. For a comprehensive explanation, I would recommend the excellent color & gamma FAQs by Charles A. Poynton:

http://www.poynton.com

For more details about profile architecture, you can reach the latest ICC specs on:

http://www.color.org

*****PLEASE NOTE THAN Little CMS IS NOT AN ICC SUPPORTED LIBRARY*****

I will assume the reader does have a working knowledge of the C programming language. This doesn’t mean Little CMS can only be used by C applications, but it seems the easiest way to present the API functionality. Little CMS is meant to be portable to any **C99** compliant compiler.

Documentation

Little CMS documentation is hold in three different papers. This one you are reading is the tutorial. Its goal is to introduce the engine and to guide you in its basic usage. It does not, however, give details on all available functionality. For that purpose, you can use the API reference, which gives information on all the constants, structures and functions in the engine. The third document is the plug-in documentation. It details how to extend the engine to fit your particular purposes. You need some experience in the core API to write plug-ins, therefore, the plug-in API reference is somehow more advanced that the remaining two.

Aside documentation, there are sample programs that you can explore. Those are located in the “utils” folder. Those programs are also handy in isolation. This is the list of utilities, each one is documented elsewere.

- Tificc : Color manage tiff files
- Jpgicc: Color manage jpeg files
- transicc: Color calculator, convert numbers across ICC profiles
- linkicc: link two or more profiles in a single devicelink.
- tiffdiff: utility to get color differences in two similar tiff files
- psicc: Generate CRD and CSA for PostScript

Requeriments

In order to improve portability and minimize code complexity, Little CMS.x **requires** a **C99** compliant compiler. This requeriment has been relexed on Microsoft's Visual Studio because its wide adoption by industry (VC is not *fully* C99 compliant). Borland C 5.5 (available for free) has been tested and found to work Ok. gcc and the Intel compiler does work ok.

Include files

Any application using Little CMS has to include just one header.

```
#include "lcms2.h"
```

The header has been renamed to lcms2.h in order to improve the adoption of version 2. In fact, both Little CMS 1.x and 2.x can coexist installed in same machine. This is very important on platforms like linux, where Little CMS is nested deep in the dependency tree. Little CMS no longer relies on icc34.h or any file coming from ICC. All constants are now prefixed by "cms" and there is just one single license for all the package.

Lcms2.h does expose the API, and only the API. Unlike 1.xx series, all internal functions are no longer accesible for client applications.

A special case are the Little CMS plug-ins. Those constructs can access more functions that the API, just because they are supposed to access Little CMS internals to add new functionality. There is a specialized include file for that:

```
#include "lcms2_plugin.h"
```

This file should only be included when defining plug-ins. It defines some additional functions and is described in the Little CMS2.x Plugin API document.

Basic Concepts

Little CMS defines several kinds of structures, that are used to manage the various abstractions required to access ICC profiles. The main structures are **profiles** and **transforms**. In a care of good encapsulation, these objects are not directly accessible from a client application. Rather, the user receives a *'handle'* for each object it queries and wants to use. This handle is a stand-alone reference; it cannot be used like a pointer to access directly the object's data. This approach is used on other parts of the API as well, across a generic handle.

There are typedef's for such handles:

- **cmsHPROFILE identifies a handle to an open profile.**
- **cmsHTRANSFORM identifies a handle to a transform.**
- **cmsHANDLE identifies a generic object.**

Source code conventions

- All API functions and types have their label prefixed by 'cms' (lower-case). All plug-in building aids are prefixed by '_cms' (lower-case).
- Some functions does accepts flags. In such cases, you can build the flags specifier joining the values with the bitwise-or operator '|'.
- Functions does report error by the return code.
- An important note is that the engine should not leak memory when returning an error, e.g., querying the creation of an object will allocate several internal tables that will be freed if a disk error occurs during a load.

The const keyword

'const' is your friend. Since Little CMS requires now **C99**, I have enforced the use of const whatever possible. My advice is to hint the compiler with const on all chances of constant objects; it is very useful to find bugs. So, if the compiler complains on any Little CMS function because a const parameter, don't blame the API, revise your code and probably you would find a glitch.

The register keyword

Little CMS uses and requires C99, and this standard allows the use of ´register´ keyword. Some compilers complain about this as being deprecated. Well, standards cannot be changed in retro-active way unless you have a time machine! But anyway, if you want to compile the code as C++14 or things like that, you can use the toggle CMS_NO_REGISTER_KEYWORD, this is set automatically if a C++17 compiler is detected. For Visual Studio, you need to add **/Zc:__cplusplus** to make it follow the standard. I don't know why.

Basic Types

In order to guarantee portability, lcms2.h does define several base types. If you don't need your code to be portable, you can still use 'int', 'long' etc. But using Little CMS types you make sure about the representation of the data. Here are the basic types. See the API reference for further details.

Type	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 double*
cmsBool	?	No	*TRUE, FALSE Boolean type, which will be using the native integer*

Step-by-step Example

Here is an example to show, step by step, how a client application can transform a bitmap between two ICC profiles using the lcms API.

```
#include "lcms2.h"

int main(void)
{
    cmsHPROFILE hInProfile, hOutProfile;
    cmsHTRANSFORM hTransform;
    int i;

    hInProfile  = cmsOpenProfileFromFile("HPSJTW.icc", "r");
    hOutProfile = cmsOpenProfileFromFile("sRGBColorSpace.icc", "r");

    hTransform = cmsCreateTransform(hInProfile,
                                    TYPE_BGR_8,
                                    hOutProfile,
                                    TYPE_BGR_8,
                                    INTENT_PERCEPTUAL, 0);

    cmsCloseProfile(hInProfile);
    cmsCloseProfile(hOutProfile);

    for (i=0; i < AllScanlinesTilesOrWatseverBlocksYouUse; i++)
    {
        cmsDoTransform(hTransform, YourInputBuffer,
                                YourOutputBuffer,
                                YourBuffersSizeInPixels);
    }

    cmsDeleteTransform(hTransform);

    return 0;
}
```

This is slightly different from the sample on 1.xx series, as Little CMS allows you to close the profiles after creating the transform. On 1.xx you have to keep profiles open on all transform life, that is no longer required in Little CMS.x

Open the profiles

You will need the profile handles for create the transform. In this example, I will create a transform using a HP Scan Jet profile as input, and sRGB profile as output. This task can be done by following lines:

```
cmsHPROFILE hInProfile, hOutProfile;

    hInProfile  = cmsOpenProfileFromFile("HPSJTW.icc", "r")
    hOutProfile = cmsOpenProfileFromFile("sRGBColorSpace.icc", "r")
```

You surely have noticed a second parameter with a small "r". This parameter is used to set the access mode. It describes the "opening mode" like the C function fopen(). If the function fails, it return NULL. In this example we don't check the return code because simplicity sake, but you should do that if you care about segfaults!

> ***WARNING! Opening with 'w' WILL OVERWRITE YOUR PROFILE!***
> ***Don't do this except if you want to create a NEW profile.***

Opening a profile only will take a small fraction of memory. The BToA or AToB tables, which usually are big, are only loaded at transform-time, and on demand. You can safely open a lot of profiles if you wish to do so.

Little CMS is a standalone color engine, it knows nothing about where the profiles are placed. It does assume nothing about a specific directory (as Windows does, currently expects profiles to be located on SYSTEM32/SPOOL/DRIVERS/COLOR folder in main windows directory), so for get this example working, you need to copy the profiles in the local directory.

Identify the intended format of pixels

Little CMS can handle a lot of formats:

- 8 and 16 bits per sample
- up to 15 channels
- extra (unused) channels like alpha
- swapped-channels like BGR
- endian-swapped 16 bps formats like PNG
- chunky and planar organization
- Reversed (negative) channels
- Floating-point numbers
- Alpha channels, even premultiplied

For describing such formats, lcms does use a 32-bit value, referred below as "formatters ". This is just a 32-bit word holding information about the format in bits. Normally, you need not to worry about how a format pecifier is built. There are several (most usual) encodings predefined as constants, but there are a lot more. See the API reference for a complete list. Let's now say that there are specifiers for many color spaces encoded in 8 bits, in 16 bits and in floating-point. This latter in 32 or 64 bits per channel. Here are some samples:

TYPE_RGB_DBL	TYPE_CMYK_DBL	TYPE_Lab_FLT
TYPE_XYZ_FLT	TYPE_GRAY_8	TYPE_GRAY_16
TYPE_RGB_8	TYPE_RGB_8_PLANAR	TYPE_BGR_16
TYPE_BGR_16_SE	TYPE_RGBA_8	TYPE_CMY_8
TYPE_CMY_16_PLANAR		.. etc...

For example, if you are transforming a windows .bmp to a bitmap for display, you will use TYPE_BGR_8 for both, input and output buffers, windows does store images as B,G,R and not as R,G,B. Another example, you need to convert from a CMYK separation to RGB in order to display; then you would use TYPE_CMYK_8 on input and TYPE_BGR_8 on output. If you need to do the separation from a TIFF, TYPE_RGB_8 on input and TYPE_CMYK_8 on output. Please note TYPE_RGB_8 and TYPE_BGR_8 are *not* same.

Alpha channels can be handled by using formatters. Little CMS supports staight (unassociated) alpha and premultiplied alpha as well. See the API reference for more details on alpha channels.

The format specifiers are useful above color management. This will provide a way to handle a lot of formats, converting them in a single, well-known one. For example, if you need to deal with several pixel layouts coming from a file (TIFF for example), you can use a fixed output format, say TYPE_BGR_8 and then, vary the input format on depending on the file parameters. Little CMS also provides a flag for inhibit color management if you want speed and don't care about profiles. see cmsFLAGS_NULLTRANSFORM for more info.

Create the transform

When creating transform, you are giving to Little CMS all information it needs about how to translate your pixels. The syntax for simple transforms is:

```
cmsHTRANSFORM hTransform;

    hTransform = cmsCreateTransform(hInputProfile,
                                    TYPE_BGR_8,
                                    hOutputProfile,
                                    TYPE_BGR_8,
                                    INTENT_PERCEPTUAL, 0);
```

You give the profile handles, the format of your buffers, the rendering intent and a combination of flags controlling the transform behaviour.

Rendering intents

It's out of scope of this document to define the exact meaning of rendering intents. I will try to make a quick explanation here, but often the meaning of intents depends on the profile manufacturer.

INTENT_PERCEPTUAL:

Hue hopefully maintained (but not required), lightness and saturation sacrificed to maintain the perceived color. White point changed to result in neutral grays. Intended for images.

INTENT_RELATIVE_COLORIMETRIC:

Within and outside gamut; same as Absolute Colorimetric. White point changed to result in neutral grays.

INTENT_SATURATION:

Hue and saturation maintained with lightnesssacrificed to maintain saturation. White point changed to result in neutral grays. Intended for business graphics (make it colorful charts, graphs, overheads, ...)

INTENT_ABSOLUTE_COLORIMETRIC:

Within the destination device gamut; hue, lightness and saturation are maintained. Outside the gamut; hue and lightness are maintained, saturation is sacrificed. White point for source and destination; unchanged. Intended for spot colors (Pantone, TruMatch, logo colors, ...)

With Little CMS there are additional intents. Those does not belong to the ICC spec, and therefore they are labeled as “user” intents. In fact, by using plug-ins you can extend the list of available intents.

Little CMS.1 does add following non-ICC intents by default:

- *INTENT_PRESERVE_K_ONLY_PERCEPTUAL*
- *INTENT_PRESERVE_K_ONLY_RELATIVE_COLORIMETRIC*
- *INTENT_PRESERVE_K_ONLY_SATURATION*
- *INTENT_PRESERVE_K_PLANE_PERCEPTUAL*
- *INTENT_PRESERVE_K_PLANE_RELATIVE_COLORIMETRIC*
- *INTENT_PRESERVE_K_PLANE_SATURATION*

All those new intents deal with black preservation. They are described below, see the black preservation section.

Not all profiles does support all intents, there is a function for inquiring which intents are really supported for a given profile, but if you specify a intent that the profile doesn't handle, Little CMS will select default intent instead.

This is the algorithm for selecting ICC intents:

```
INTENT_PERCEPTUAL:

        If adequate table is present in profile,
        then it is used. Else default intent of profiles is used

INTENT_RELATIVE_COLORIMETRIC:

        If adequate table is present in profile,
        then it is used. Else revert to perceptual
        intent.

INTENT_SATURATION:

        If adequate table is present in profile,
        then it is used. Else revert to perceptual
        intent.

INTENT_ABSOLUTE_COLORIMETRIC:

        relative colorimetric intent is used
        with undoing of chromatic adaptation.
```

Optimization

Little CMS tries to optimize profile chains whatever possible. There are some built-in optimization schemes, and you can add new schemas by using a plug-in. This generally improves the performance of the transform, but may introduce a small delay when creating the transform. In modern machines this is not noticeable at all. If you are going to transform just few colors, you don't need this precalculations. Then, the flag cmsFLAGS_NOOPTIMIZE in cmsCreateTransform() can be used to inhibit the optimization process. See the API reference for a more detailed discussion of the flags.

Apply the transform

Next, you can translate your bitmap, calling repeatedly the processing function:

```
cmsDoTransform(hTransform, YourInputBuffer,
                           YourOutputBuffer,
                           YourBuffersSize);
```

This function is intended to be quite fast. You can use this function for translating a scan line, a tile, a strip, or whole image at time.

Scanline padding

Windows, stores the bitmaps in a particular way... for speed purposes, does align the scan lines to double word boundaries, a bitmap has in windows always a size multiple of 4. This is OK, since no matter if you waste a couple of bytes, but if you call cmsDoTransform() and passes it WHOLE image, Little CMS doesn't know nothing about this extra padding bytes. It assumes that you are passing a block of BGR triplets with no alignment at all. This result in a strange looking "lines" in obtained bitmap. The solution most evident is to convert scan line by scan line instead of whole image. This is as easy as to add a for() loop, and the time penalty is so low that is impossible to detect.

Scanline overlap

It is safe to use same block for input and output, but only if the input and output are coded in same format. For example, you can safely use only one buffer for RGB to RGB but you cannot use same buffer for RGB as input and CMYK as output.

Finishing the color transform

New with Little CMS is the ability to free profiles just after creating the transform. A open profile may take big amounts of memory, so it is a good idea to free the resources as soon as you don't need them. Color transforms does take also resources, so you have to free them to avoid leaks.

This can be done by calling:

```
cmsDeleteTransform(hTransform);
cmsCloseProfile(hInputProfile);
cmsCloseProfile(hOutputProfile);
```

Note that cmsDeleteTransform() does NOT automatically free associated profiles. This works in such way to let programmers to use a open profile in more than one transform.

Special profile types

Aside the normal, file-based profiles, there are a number of situations where you may want something different. Here are listed such special cases.

Embedded profiles

Some image file formats, like TIFF, JPEG or PNG, does include the ability of embed profiles. This means that the input profile for the bitmap is stored inside the image file. Little CMS provides a specialised profile-opening function for deal with such profiles.

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

This function works like cmsOpenProfileFromFile(), but assuming that you are given full profile in a memory block rather than a filename. Here there is no "r", since these profiles are always read-only. A successful call will return a handle to an opened profile that behaves just like any other file-based. NULL if the function fails.

Memory based profiles does not waste more resources than memory, so you can have tons of profiles opened sumultaneously by using this function. Once opened, you can safely FREE the memory block. Little CMS keeps a temporary copy. You can retrieve information of this profile, but generally these are minimal shaper-matrix profiles with little if none handy info present.

Be also warned that you may find WRONG profiles embedded, i.e., profiles marked as using different colorspace that one the profile really manages. Little CMS is NOT likely to understand these profiles since they will be wrong at all.

Device-link profiles

Device-link profiles are "smelted" profiles that represents a whole transform rather than single-device profiles. In theory, device-link profiles may have greater precision that input/output chains and are faster to load. If you plan to use device-link profiles, be warned there are drawbacks about its inter-operability and the gain of speed is almost null. Perhaps their only advantage is when restoration from CMYK with great precision is required, since CMYK to pcs CLUTs can become very, very big.

For creating a device-link transform, you may open the device link profile as usual, using cmsOpenProfileFromFile(). Then, create the transform with cmsCreateMultiprofileTransform.

```
hDeviceLink = cmsOpenProfileFromFile("MYDEVLINK.icc", "r");

hTransform =  cmsCreateMultiprofileTransform(&hDevicelink,
                                    1,
                                    TYPE_RGB_8,
                                    TYPE_BGR_8,
                                    INTENT_PERCEPTUAL,
                                    0);
```

That's all. Little CMS will understand and transparently handle the device-link profile. Note the first parameter is an array of handles, so you can use '&' in this particular case. Another option is to use the device link profile as input and the output profile parameter equal to NULL:

```
hDeviceLink = cmsOpenProfileFromFile("MYDEVLINK.icc", "r");

hTransform  = cmsCreateTransform(hDeviceLink, TYPE_RGB_8,
                            NULL, TYPE_BGR_8,
                            INTENT_PERCEPTUAL,
                            0);
```

There is also a function for dumping a transform into a devicelink profile.

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

This profile can be used in any other transform or saved to disk/memory. Note that you must specify the version number. That is required because v4 profiles may be implemented in a quite different way of v2. Settinng proper version number will assure you compatibility with other software. 4.2 is the latest ICC revision. 3.4 will assure compatibility with old software.

If you want to save information on which profiles has been used in the transform, you must include the special flag **cmsFLAGS_KEEP_SEQUENCE** when creating the transform. This is done in such way because the original profiles may hold multi localized descriptions, and the total memory may be very big. Unless you need to create strictly compliant device links, you need not this flag.

Named color profiles



Named color profiles are a special kind of profiles handling lists of spot colors. The typical example is PANTONE®. Little CMS deals with named color profiles like all other types, except they must be in input stage and the encoding supported is limited to a one single channel of 16-bit that works as an index to the table.

Little CMS has no affiliation with PANTONE Company. PANTONE® is a trademark of Pantone, Inc. PANTONE Color identification is solely for artistic purposes and not intended to be used for specification.

Let's assume we have a Named color profile holding only 4 colors:

- CYAN
- MAGENTA
- YELLOW
- BLACK

We create a transform using:

```
hTransform = cmsCreateTransform(hNamedColorProfile,
                                TYPE_NAMED_COLOR_INDEX,
                                hOutputProfile,
                                TYPE_BGR_8,
                                INTENT_PERCEPTUAL, 0);
```

TYPE_NAMED_COLOR_INDEX is a special encoding for these profiles, it represents a single channel holding the spot color index. In our case value 0 will be "CYAN", value 1 "MAGENTA" and so one. For converting between string and index, you have to retrieve the list by using

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

Then there are several function to deal with such lists. For example, there is an auxiliary function:

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

That will perform a look up on the spot colors database and return the color number or -1 if the color was not found. Other additional functions for named color transforms are:

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

That returns the number of colors present on transform database.

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

That returns extended information about a given color. Named color profiles does hold two coordinates for each color, let's take our PANTONE example. This profile would contain for each color the CMYK colorants plus its PCS coordinates, usually in Lab space. Little CMS can work with named color using both coordinates. Creating a transform with two profiles, if the input one is a named color, then you obtain the translated color using PCS.

Example, named color → sRGB will give the color patches in sRGB

On the other hand, creating a multiprofile transform with only one named color profile returns the device coordinates, that is, CMYK colorants in our PANTONE sample.

Example: Named color will give the CMYK amount for each spot color.

So, you can use a named color profile in two different ways, as output, giving the index and getting the CMYK values or as input and getting the Lab for that color.

- A transform which involves only one named color profile will give the CMYK values for the spot color on the printer the profile is describing. This would be the normal usage.
- A transform Named color -> another printer will give on the output printer the spot a color as if they were printed in the printer named color profile is describing. This is useful for soft proofing.

Built-in profiles

There are several built-in profiles that programmer can use without the need of any disk file. These does include:

- sRGB profile
- L*a*b profiles
- XYZ profile
- Gray profiles
- RGB matrix-shaper.
- Linearization device link
- Ink-Limiting device link
- Adjust device link.
- NULL profile
- .CUBE import device link

Many of there are very useful for tricking & trapping. For example, creating a transform from sRGB to Lab could be done without any disk file.

Something like:

```
hsRGB = cmsCreate_sRGBProfile();
hLab  = cmsCreateLab4Profile()

xform = cmsCreateTransform(hSRGB, TYPE_RGB_DBL, hLab,
                           TYPE_Lab_DBL,
                           INTENT_PERCEPTUAL, 0);
```

Then you can convert directly form double sRGB values (in 0..1.0 range) to Lab by using:

```
double RGB[3];
cmsCIELab Lab;

RGB[0] = 0.1; RGB[1] = 0.2 RGB[2] = 0.3;
cmsDoTransform(xform, RGB, &Lab, 1);

.. get result on "Lab" variable ..
```

The NULL profile returns zero for any input color. This is useful for out-of-gamut warning. If you need to know which pixels are out of gamut, but want only zeros or ones as result, you can use the NULL profile as output and turn on the gamut warning feature.

Some of those built-ins does accept parameters. That means, the built in primitive does not generate a unique profile but families of profiles with same functionality. I will call that “on the fly” profiles. You can create your own RGB or Gray input profiles "on the fly". See next section on how to do that.

On-the-fly profiles

There are several situations where it will be useful to build a minimal profile using adjusts only available at run time. Surely you have seen the classical pattern-gray trick for adjusting gamma: the end user moves a scroll bar and when pattern seems to match background gray, then gamma is adjusted. Another trick is to use a black background with some gray rectangles. The user chooses the most neutral grey, giving the white point or the temperature in °K. All these visual gadgets are not part of Little CMS, you must implement them by yourself if you like. But Little CMS will help you with a function for generating a virtual profile based on the results of these tests.

Another usage would be to build colorimetric descriptors for file images that does not include any embedded profile, but does include fields for identifying original colorspace.

One example is TIFF files. The TIFF 6.0 spec talks about "RGB Image Colorimetry" (See section 20) a "colorimetric" TIFF image has all needed parameters (WhitePointTag=318, PrimaryChromacitiesTag=318, TransferFunction=301, TransferRange=342)

Obtain a emulated profile from such files is easy since the contents of these tags does match the cmsCreateRGBProfile() parameters. Also PNG can come with information for build a virtual profile, See the gAMA and cHRM chunks.

RGB virtual profiles

This is the main function for creating virtual RGB profiles:

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

It takes as arguments the white point, the primaries and 3 tone curves. The profile emulated is always operating in RGB space. Once created, a handle to a profile is returned. This opened profile behaves like any other file or memory based profile. Virtual RGB profiles are implemented as matrix-shaper, so they cannot compete against CLUT based ones, but generally are good enough to provide a reasonable alternative to generic profiles. To simplify the parameters construction, there are additional functions, for example:

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

This function computes the xyY chromacity of white point using the temperature. Screen manufacturers often includes a white point hard switch in monitors, but they refer as "Temperature" instead of chromacity. Most extended temperatures are 5000K, 6500K and 9300K.

It returns TRUE if a valid white point can be computed, or FALSE if the temperature were non valid. You must give a pointer to a cmsCIExyY struct for holding resulting white point. For primaries, currently I don't know any trick or proof for identifying primaries, so here are a few chromacities of most extended. Y is always 1.0

	RED		GREEN		BLUE	
	x	y	x	y	x	y
NTSC	0.67,	0.33,	0.21,	0.71,	0.14,	0.08
EBU(PAL/SECAM)	0.64,	0.33,	0.29,	0.60,	0.15,	0.06
SMPTE	0.630,	0.340,	0.310,	0.595,	0.155,	0.070
HDTV	0.670,	0.330,	0.210,	0.710,	0.150,	0.060
CIE	0.7355,0.2645,0.2658,0.7243,0.1669,0.0085					

These are TRUE primaries, not colorants. Little CMS does include a white-point balancing and a chromatic adaptation using a method called Bradford Transform for D50 adaptation.

The tone curves can be generated by any tone curve creation function. The simplest one is cmsBuildGamma, which creates a pure-exponential function like CRT gamma. See the tone curves section for more information on how to create such curves.

Gray virtual profiles

Another kind of profiles that can be built on runtime are gray scale profiles. This can be accomplished by the function:

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

This one is somehow easier, since it only takes one curve (the transfer function) and the media white point. Of course gray scale does not need primaries, since they are monochrome. The primary here is the white point itself.

Linearization device links

This is a very handy type of virtual profiles. It may be use for several things, like linearizing printers or applying curves to RGB images. They basically apply a transform that is channel-independent. That is, each channel response is independent of the rest of channels. That may be understood as applying curves to each channel, but the response can be tabulated and is not restricted to curves.

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

You need to specify the color space the profile is operating, which must be the same on input and output, and the tone curves to apply to each channel. The number of channels is implicit in the color space.

Ink limiting device links

Intended mainly for CMYK. It uses the hypercube algorithm. Works on CMYK → CMYK, and the parameter specifies the total amount of ink in %. Black ink is never reduced, CMY are reduced proportionally to meet the limits.

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

The Color Space parameter is provided for future extensions. Currently it only supports CMYK.

Bright, Contrast, Hue, Saturation and white point.

Provided for compatibility with previous versions. With this function you can adjust Brightness, Contrast, Hue and Saturation in a color transform. Additionally you can modify the color temperature. It mimics the controls found on most monitors. Operates on $L^*a^*b^*$ → $L^*a^*b^*$, so this profile should be inserted into input and output profiles.

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

Ranges are:

Bright: 0=no op, < 0 decrease, > 0 increase
Contrast: 1=no op, < 1 decrease, > 1 increase
Saturation: 0=no op, < 0 decrease, > 1 increase
Hue: 0=no op, up to 360º, hue displacement

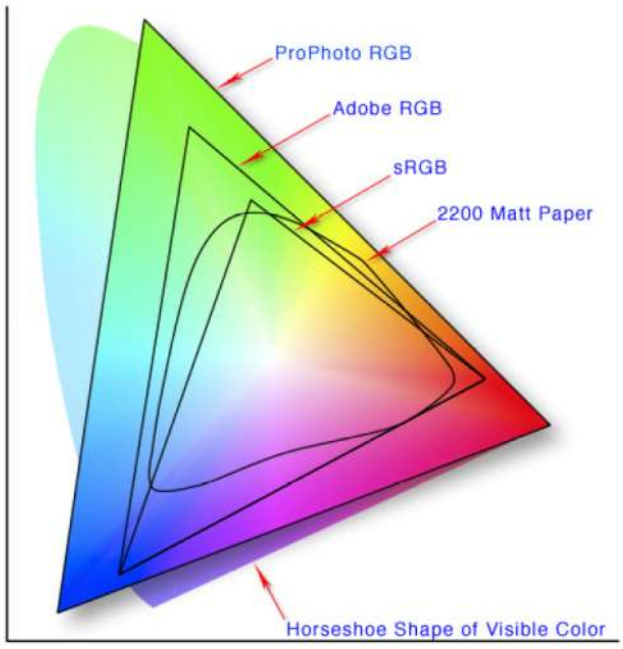
Adjusts are done in the LCh space, by using those formulae:

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

Proofing

An additional ability of Little CMS is to create "proofing" transforms.

A proofing transform can 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.



The creation of a proofing transform involves three profiles, the input and output ones as cmsCreateTransform() plus another, representing the emulated profile.

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

Also, there is another parameter for specifying the intent for the proof. The Intent here represents the intent the user will select when printing, and the proofing intent represent the intent system is using for showing the proofed color. Since some printers can archive colors that displays cannot render (darker ones) some gamut-remapping must be done to accommodate such colors. Normally INTENT_ABSOLUTE_COLORIMETRIC is to be used: it is likely the user wants to see the exact colors on screen, cutting off these un-representable colors.

INTENT_RELATIVE_COLORIMETRIC could serve as well.

Proofing transforms can also be used to show the colors that are out of the printer gamut. You can activate this feature by using the cmsFLAGS_GAMUTCHECK flag in dwFlags field.

Then, the function:

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

Can be used to define the out-of-gamut marker. Range is 0..0xffff

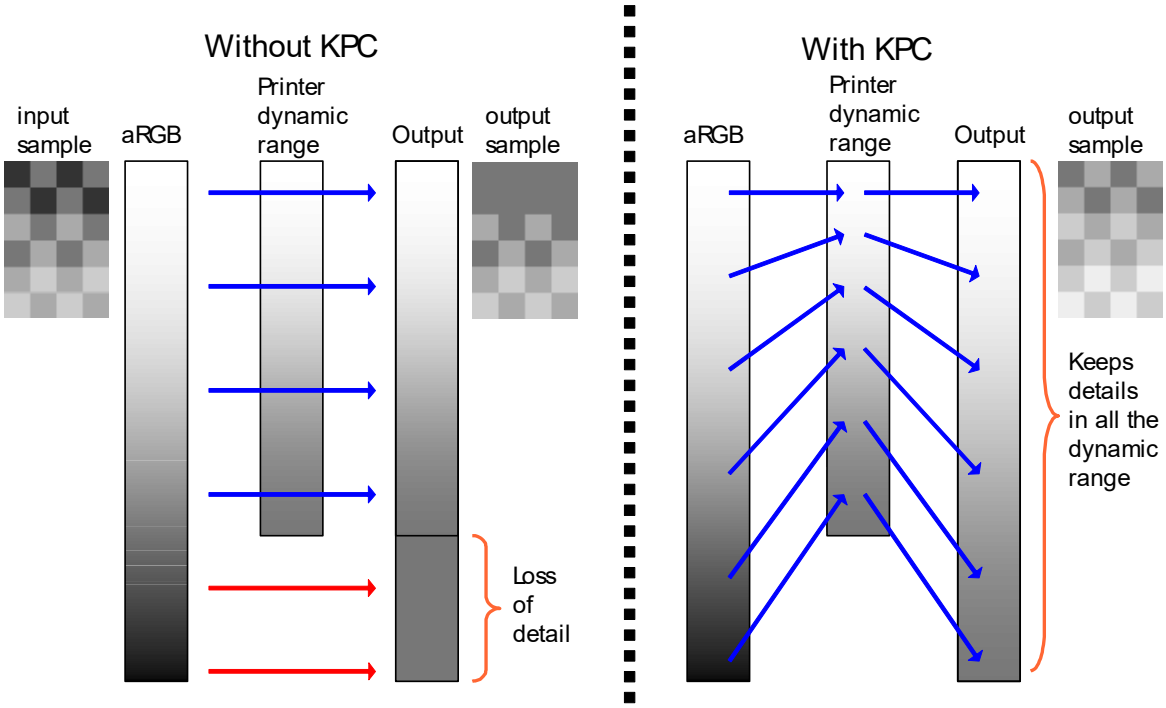
For activating the preview or gamut check features, you **MUST** include the corresponding flags

```
cmsFLAGS_SOFTPROOFING
cmsFLAGS_GAMUTCHECK
```

This is done in such way because the user usually wants to compare with/without softproofing. Then, you can share same code. If any of the flags is present, the transform does the proofing stuff. If not, the transform ignores the proofing profile/intent and behaves like a normal input-output transform. In practical usage, you need only to associate the check boxes of "softproofing" and "gamut check" with these flags.

Black point compensation

Black Point Compensation (BPC) controls whether to adjust for differences in black points when converting colors between color spaces. When Black Point Compensation is enabled, color transforms map white to white and luminance of black to luminance of black. The black point compensation feature does work better in conjunction with relative colorimetric intent. Perceptual intent should make no difference, although it may affect some (wrong) profiles.



The mechanics are simple. BPC does scale full image across gray axis in order to accommodate the darkest tone origin media can render to darkest tone destination media can render. Let's take an example. You have a separation (CMYK image) for, say, SWOP. Then you want to translate this separation to another media on another printer. The destination media/printer can deliver a black darker that original SWOP. Now you have several options.

a) Use perceptual intent, and let profile do the gamut remapping for you. Some users complain about the profiles moving too much the colors. This is the "normal" ICC way.
b) Use relative colorimetric. This will not move any color, but depending on different media you would end with "flat" images, taking only a fraction of available grays or a "collapsed" images, with loss of detail in dark shadows.

c) Use relative colorimetric + black point compensation. This is the discussion theme. Colors are unmoved *except* gray balance that is scaled in order to accommodate to the dynamic range of new media. Is not a smart CMM, but a fist step letting the CMM to do some remapping.

The algorithm used for black point compensation is a XYZ linear scaling in order to match endpoints. You can enable the BPC feature by using this in the dwFlags field, it works on softproofs too.

```
cmsFLAGS_BLACKPOINTCOMPENSATION
```

Black preserving intents

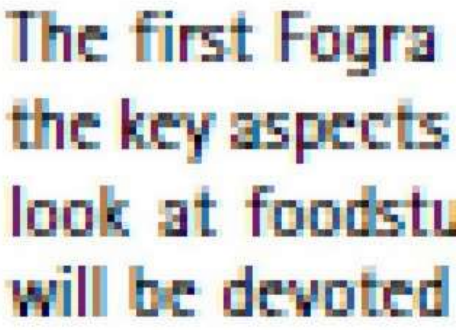
Black preservation deals with CMYK → CMYK transforms, and is intended to preserve, as much as possible, the black (K) channel whilst matching color by using CMY inks. There is a tradeoff between accuracy and black preservation, so you lost some accuracy in order to preserve the original separation. Not to be a big problem in most cases, benefits of keeping K channel are huge!

No, this does not belong to normal ICC workflow. ICC has tried to address such need but still there is nothing in the spec.



Let's see what the issue is. Suppose we work in press. Press are very tied to standards, US press uses SWOP and European folks are more toward FOGRA. Japanese people use other standards like TOYO, for example. Each standard is very well detailed and presses are setup to faithfully emulate any of these standards.

Ok, let's imagine you got an image ad, looking like that. This is a very usual flier; now just imagine instead of getting it in PDF, you get it as a raster file. Say in a CMYK TIFF, ready for a SWOP press. And you want to print in on a FOGRA27!!



If I convert from SWOP to FOGRA27, the ICC profiles totally mess up the K channel, so a portion of the picture that originally is using only black ink, after the conversion, gets Cyan, Magenta, Yellow and well, a little bit of black as well. Now please realize what happens on all the text in the Flier.

Little CMS has added two different modes to deal with that: Black-ink-only preservation and black-plane preservation. The first is simple and effective: do all the colorimetric transforms but keep only K (preserving L*) where the source image is only black. The second mode is fair more complex and tries to preserve the WHOLE K plane.

Linear spaces

Linear spaces are those that are operating in linear XYZ gamma space. You should NEVER user linear gamma spaces to store your 8-bit images. Why? Because in 8 bits you have 256 levels, and in linear gamma the separation between those levels is not perceptually uniform. That means you have very few levels to encode the effective dynamic range of your image and many levels are wasted in highlights. Hold on, you would say, RAW images are encoded in linear gamma and they work quite well, isn't it? You are right… but I said 8 bits, remember? If you move to 16 bits or floating point, you can still use linear encoding, but with some care.

When you use lcms to create a color transform joining two or more profiles, you are creating a devicelink profile. You don't see it as a file; it lives in memory, and is destroyed when you delete the transform. But it is there. Devicelinks can be implemented in different ways, for example they can be implemented as a set of curves, or by a matrix, or by a 3D CLUT table, or by a combination of all elements above. Some of those ways are better than others in terms of xput, others are better in terms of image quality. CMMs have to "guess" which is the best combination of elements for a given set of profiles. There is balance between quality and performance. For some corner cases, optimizing for speed can effectively introduce defects on quality.

An optimization method used by lcms when the transform converts from 16 bits to 16 bits, is a CLUT table. This is just a 3D (or 4D in CMYK) grid with nodes. Pixel values are interpolated across nodes. For example, the distortion you get when going from sRGB to Adobe RGB is stored in a 3D grid of 17 nodes on each R, G, B side. When a pixel arrives, the corresponding nodes that enclose the value are selected and the result is interpolated. In our 17 nodes example, a value of, say, (100, 100, 100) will go on the 100*(17-1)/255 = 6.7 so the nodes 6 and 7 of each side will be taken for interpolation.

Let's now take a linear space. Since as said, many colors are collapsed to a relatively few codes due to the gamma encoding, almost all dynamic range is confined to few nodes. That means, In a 17 nodes grid, most image dynamic range will fall in 5 or 6 nodes. And this is the reason you got posterization in shadows: most of dark tones falls in just 1-2 nodes and linear interpolation cannot deal with the non-linear nature of the transform linear-gamma 2.2.

How to solve this? The most evident way is to not use 3D CLUT optimization. The CMM already does that if you use floating point. Placing a cmsFLAGS_NOOPTIMIZE in all transforms would prevent problems, but at big performance penalty that is hard to explain just to fix this specific case. In 2.13 there is some code to detect optimization used on linear spaces.

My recommendation for programmers would be to allow end user to turn optimization off for general usage, or at least to provide a specialized workflow for RAW handling with optimizations turned off, that is the only place when linear XYZ makes sense. For users, I would recommend to NEVER use linear XYZ spaces. They are good for nothing, nor for storage, nor for image processing. The very few algorithms that need to be done in linear can do and undo the conversion when processing. But anyway, there are people with strong opinions on this field and everybody is free to do whatever they want. This is just a recommendation; please don't take it as a stone-engraved truth.

Error Logging

Error handling has changed drastically with Little CMS2. This is mostly because most operations that can fail, can also report the status. So, in Little CMS2, there are no longer error handlers. When a function fails, it just return the failure status. For example, if a color transform cannot be created, the function returns NULL instead of a valid handle.

So, you have to check the return of the function for error handling. This approach is more robust and can deal with multithreaded environments. Also, the memory management is easier in this way and a failing function should not leak any resources. With Little CMS2, leaked resurces are bugs, which was not so clear on 1.x series.

For debugging purposes, it may be handy to know what is making that function to fail. And here goes the error logging. Some Little CMS functions does have a error logging facility. When they are going to fail, they call a user-defined hook. This hook does recive an ASCII text in english with some clues about the error. This is the error logging callback, and you can set it up by calling following function:

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

To be accepted by this function, your hook should be defined as this example:

```
void MyLogErrorHandler(cmsContext ContextID,
                       cmsUInt32Number ErrorCode, const char *Text)
{
    printf("%s\n", Text);
}
```

If you take a look on the parameters, you will see a context identifier, an error code and the descriptive text. The context id is giving the environment. That is the same pointer you passed to the low level or THR function. In this way you can share same hook for different threads, for example. Please note in some special situations ContextID can be NULL.

The numeric code gives some clue on the nature of the message. Can be used to give some sort of information to end user, but it does not fully describe the nature of error. Think on it as families of errors.

Again, the error logging is a debug feature. It should NOT be used to notify end user about errors. Unless you want to confuse the end user with arcane messages, it is better to just say "the profile cannot be open" instead of a criptic "corrupted MPE in DToA2 tag". On the other side, this message can be of great aid for developers to locate why this particular profile is failing.

Table of numeric Error Codes

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

Notes:

It is important to note a single function call may trigger more than a single error logging entry. In our anterior sample, opening a profile with corrupted data would trigger an event for corrupted profile and another for invalid tag.

The logging function should NOT terminate the program, as this obviously can leave unfreed resources. It is the programmer's responsability to check each function return code to make sure it didn't fail.

Getting information from profiles

Textual information

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.

There are four kinds of textual information you may want to retrieve:

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

And here you have two functions to get the requiered information, either in UNICODE. UTF8 or in plain ASCII.

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

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

```
cmsUInt32Number cmsGetProfileInfoUTF8(cmsHPROFILE hProfile,
                                    cmsInfoType Info,
                                    const char LanguageCode[3],
                                    const char CountryCode[3],
                                    char* Buffer,
                                    cmsUInt32Number BufferSize);
```

Note that ASCII is strictly 7 bits, so you need to use wide chars or UTF8 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. Language and country codes are a lot, you can get the full list here:

Language Code: first name language code from ISO-639/2.
http://lcweb.loc.gov/standards/iso639-2/iso639jac.html

Country Code: first name region code from ISO-3166.
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:

```
cmsNoLanguage
```

For the language and

```
cmsNoCountry
```

For the country

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 Little CMS to choose any other country (or language!) if what you ask for is not available.

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 and 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.

Profile header fields

These are the functions to access all members of ICC profile header. There are counter-parts to set the header members on profile creation. See the API reference for more details.

```
cmsUInt32Number  cmsGetHeaderFlags(cmsHPROFILE hProfile);

void  cmsGetHeaderAttributes(cmsUInt64Number* Flags, cmsHPROFILE hProfile);
void  cmsGetHeaderProfileID(cmsHPROFILE hProfile, cmsUInt8Number* ProfileID);

cmsBool cmsGetHeaderCreationDateTime(struct tm *Dest, cmsHPROFILE hProfile);
cmsUInt32Number  cmsGetHeaderRenderingIntent(cmsHPROFILE hProfile);
cmsUInt32Number  cmsGetHeaderCreator(cmsHPROFILE hProfile);

cmsUInt32Number  cmsGetHeaderManufacturer(cmsHPROFILE hProfile);
cmsUInt32Number  cmsGetHeaderModel(cmsHPROFILE hProfile);

cmsColorSpaceSignature cmsGetPCS(cmsHPROFILE hProfile);
cmsColorSpaceSignature cmsGetColorSpace(cmsHPROFILE hProfile);
cmsProfileClassSignature cmsGetDeviceClass(cmsHPROFILE hProfile);

cmsFloat64Number cmsGetProfileVersion(cmsHPROFILE hProfile);
cmsUInt32Number  cmsGetEncodedICCversion(cmsHPROFILE hProfile);
```

Profile Directory

You can iterate across the profile directory to list all available tags. There is a function to check if a given tag is present on directory. There is a limit of 100 ***different*** tags per profile. Not to be a problem, since actual ICC spec defines less than that number.

```
cmsInt32Number    cmsGetTagCount(cmsHPROFILE hProfile);
cmsTagSignature   cmsGetTagSignature(cmsHPROFILE hProfile, cmsUInt32Number n);
cmsBool           cmsIsTag(cmsHPROFILE hProfile, cmsTagSignature sig);
```

Profile capabilities

Some special functions can check the profile internal implementation:

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

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

cmsBool cmsIsMatrixShaper(cmsHPROFILE hProfile);

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

This one helps on inquiring if a determinate intent is supported by an opened profile. You must give a handle to profile, the intent and a third parameter specifying how the profile would be used. The function does return TRUE if intent is supported or FALSE if not. If the intent is not supported, Little CMS will use default intent (usually perceptual).

Reading tags

The low level interface for reading tags is as simple as a single function. In Little CMS you can read a tag from an open profile by doing:

```
Tag = cmsReadTag(hProfile, TagSignature);
```

Little CMS will return (if found) a pointer to a structure holding the tag. Simple, but not simpler as the structure is not the 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. Writing tags is almost the same, 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.

Anyway, you may decide all that is useless and you want just to write/read bytes to the profile, in this case the raw variants are for you.

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

That's alike, but different in two important aspects. 1st, the memory is not owned by the profile, but by *you*, so *you* have to allocate the neccesary amount of memory to hold entire tag. To know the needed size, just pass NULL as buffer and 0 as buffer size. The function will return the number of needed bytes.

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, then you have to interpret all those bytes!

IMPORTANT NOTE: There is a direct relationship "tag type" → "Little CMS type". On the API reference, there is a table listing all tag types and the Little CMS type used to read/write.

Writing tags

Writing tags is almost the same. You just provide a pointer to the structure and the tag name. Little CMS does all serialization.

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

Function returns TRUE on success, FALSE on error.

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

The RAW version does the same 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.

The linked tag feature

Some profile creators may want to reuse the same values for several different tags. To do that you may want to use the linked tag feature.

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

After this call, a new tag of signature ‘sig’ is created. The entry, however, points to the same location as tag ‘dest’.

Creating new profiles

Use this function to create an empty profile, ready to be populated.

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

In addition, when you open a profile with 'w' as access mode, you got a simpler Lab identity. That is, a profile marked as Lab colorspace that passes input untouched to output. You can use that as a basis of new profiles, by setting colorspace, device class, PCS and then add the required tags.

Tone curves

Tone curves are a powerful tool to model 1D transformations. Little CMS provides several primitives to create, group and apply such tool. Tone Curves in the engine can be bounded and unbounded, specified as math expressions or by means of tables. Tone Curve primitives does support:

- Tabulated curves
- Parametric curves
- Segmented curves

Tabulated curves were introduced as *curveType* in ICC version 2. In Version 4 of the ICC Profile Specification, *parametricCurveType* was introduced as an alternative to *curveType* for the representation of one-dimensional transfer functions. Either type can be used for the TRC tags or for the A-curves, B-curves, and M-curves embedded in *lutAtoBType* or *lutBtoAType* tags. In contrast to the older *curveType*, the new V. 4 type defines curves by closed-form expressions, rather than by 1D Look-Up Tables. Each curve is a scalar function of a scalar variable, but the expressions also involve constants, or *parameters*, which are encoded in the corresponding profile tags.

The V4 Profile specification supports five different *Function Types*, requiring between 1 and 7 parameters. The Specification places no restrictions on the values of these parameters, aside from those imposed by the format. According to Clause 10.15, the parameters are encoded in the s15Fixed16Number format. Thus, the values can range from –32768 to almost +32768 (actually 32768 – 1/65536) in steps of 1/65536, or 0.0000152587890625. These restrictions are quite mild and, in practice, are hardly noticeable.

In Types 1, 2, 3, and 4, the domain is divided into two segments, and the power law is employed only in the higher segment. For instance, the definition of Type 1 is:

$$\begin{aligned} y = f_1(x) &= 0, && 0 \le x < -b/a \\ &= s^{\gamma} && -b/a \le x \le 1 \end{aligned}$$

In normal usage, a will be positive and b will be negative, so that the segment boundary, $-b/a$, occurs at a positive value of x. The function is identically zero in the lower segment

Type 2 curves have a similar structure, with a segment boundary at $x = -b/a$, so the same analysis applies. However, in Types 3 and 4, the segment boundary is defined by an independent parameter, d. In the absence of restrictions, it is quite possible for d to be less

than $-b/a$. There can then be values of x in the higher segment ($x > d$) at which $s = ax + b$ will be negative.

Types 3 and 4 do not enjoy a similar guarantee. Here is the definition of Type 3:

$$y = f_3(x) = cx, \qquad 0 \le x < d$$
$$= s^{\gamma}, \qquad d \le x \le 1$$

Type 4 is defined as follows:

$$y = f_4(x) = cx + f, \qquad 0 \le x < d$$
$$= s^{\gamma} + e, \qquad d \le x \le 1$$

Where e and f are additional parameters. See the API reference for more details on supported parametric curves.

Recently, the multiProcessElementsType introduced new curve types to the ICC specification, including formula curves (defined similarly to the parametricCurveType) and sampled curve segments. The issues of imaginary numbers, continuity, and smoothness discussed above also apply to MPE segmented curves. However, continuity and smoothness may have different considerations since MPE curves are unbounded and segmented curves are most likely be used to perform the clipping that is required to set up inputs to following CLUT elements.

You can add new types of parametric curves by using the tone curve plug-in, but you have to be carefull on that, since the obtained profiles will not be compliant with the ICC spec. See the Plug_in API for more details.

Pipelines and Multi Processing elements

Previously, all internal handling of profiles was done by using several, unrelated structures. We had a LUT, a Matrix Shaper and then customized adjustments hardcoded in certain places. That was not a very good idea in terms of flexibility and maintainability, so in version 2, I joined all that together in a more general and flexible concept: the pipelines.

That is just what the name suggests. A pipeline is a construct that has several steps, the stages, which are evaluated in order. You can feed some values to the pipeline and then get the results after the evaluation. Moreover, pipelines can be optimized for performance. If a given pipeline contains, for example, only matrices, a suitable optimization would be to multiply all matrices and therefore simplify several steps in a single matrix.

All processing in Little CMS is performed by means of pipelines. And this has been a valuable tool in order to implement new things, like multi-processing elements type. What is that? Let's see the full story…

In November 2006 the ICC approved the multiProcessingElements Tag type as part of the Floating Point Encoding Range addendum to the ICC profile Specification. This new tag type's primary purposes were to overcome limited precision in ICC profiles by allowing for the direct encoding of floating point data in an ICC profile, remove bounding restrictions for both device side and PCS encoding ranges, and provide for backwards compatibility to the existing ICC profile specification.

Eight new tags that use this new Tag type are defined to allow for the optional substitution of the tags that define rendering intents in an ICC profile. D2Bx/B2Dx tags can now exist in a profile in addition to A2Bx/B2Ax tags. A CMM can now optionally first look for D2Bx/B2Dx tags and use them instead of the A2Bx/B2Ax tags or Matrix/TRC tags. Explicit definition of the absolute rendering intent is also now possible through the use of D2B3 and B2D3 tags.

The eight new tags are optional which means that existing Color Management systems can ignore them as private tags. This allows for profiles to be built and embedded in images, or otherwise used in workflows that do not support the use of these new tags without breaking those workflows. It is hoped that the use and adoption of these tags will be made easier because their presence shouldn't break existing workflows.

The D2Bx/B2Dx tags all make use of the new multiProcessingElements tag type. Important aspects of this tag type include:

- This tag type provides for an arbitrary sequence of processing elements to perform the device to PCS or PCS to device conversion. Processing elements can be thought of as transformation steps that convert input channel data to output channel data.

- All the processing elements encode data using 32-bit IEEE 754 floating-point encoding.
- The absolute rendering intent can be encoded with D2B3/B2D3 tags.
- The initial repertoire of processing elements includes N-dimensional lookup tables, NxM matrices, sets of one dimensional segmented curves, and two future expansion elements.
- The PCS for D2Bx/B2Dx tags is the floating point equivalent of the PCS in A2Bx/B2Ax tags. When D2Bx tags are connected to B2Dx tags no clipping is performed in the PCS.
- D2Bx tags can be connected to B2Ax or Matrix/TRC based tags with appropriate clipping of the PCS values as needed, and A2Bx or Matrix/TRC based tags can be connected to B2Dx tags.
- The CMM performs NO manipulation of data between processing elements. The CMM simply passes the results from one processing element to the next processing element.
- Generally, up to 65535 channels of floating point data can be passed between processing elements.
- Processing element types are not required to support the upper limit of 65535 input and 65535 output channels.
- The channel usage of the first and last elements in a D2Bx/B2Dx tag must agree with the channel usage requirements of both the containing D2Bx/B2Dx tag and the profile header. (Note: Currently the color fields of the profile header limit the maximum number of channels to 15).

The fall back behavior of using A2Bx/B2Ax tags is prescribed if processing elements are encountered that are unknown to the CMM. This allows for a graceful handling for future expansion of the processing element repertoire.

The device encoding range for D2Bx and B2Dx tags is unbounded, but conversions and clipping may need to be made to be compatible with A2Bx and B2Ax tags. The equivalent device encoding range of A2Bx and B2Ax tags is converted to the range of 0.0 to 1.0 when applying D2Bx and B2Dx tags.

The initial repertoire of processing element includes N-dimensional lookup tables, NxM matrices, and sets of one dimensional segmented curves. These all use 32-bit IEEE 754 floating-point encoding for processing and data storage purposes. An arbitrary number of these elements can be combined in any order to accomplish the purpose of defining a transform. Output channels from preceding elements are direct inputs to succeeding elements.

CLUT element

The CLUT element is used to store N-dimensional lookup tables. They can accept up to 15 input channels (constrained by the allowed 'Number of grid points in each dimension' in the CLUT element encoding) and output up to 65535 channels. The CLUT input range is from 0.0 to 1.0 since using grid points represents the sampled range, and clipping is prescribed for values outside this range. Scaling/conversion may need to be performed by a processing element before a CLUT element to get values into the range from 0.0 to 1.0. The output range of a CLUT element is the entire floating-point encoding range.

Matrix element

The matrix element can be used to store an NxM matrix with a constant offset vector. The input and output dimensions need not be the same, and up to 65535 channels can be used for both input and output. The input and output range is the entire floating-point encoding range.

Curve set element

The curve set element encodes multiple one dimensional curves. Up to 65535 separate curves can be defined. The curves are segmented to allow the entire floating-point encoding range to be used as both input and output. Up to 65535 segments are possible for each curve with positions and definitions of each segment definable. Each segment can be defined as a formula or sampled curve segment. Formula segments define a function type and provide parameters to the function. Sampled segments are equally spaced sample points defining a 1 dimensional look up table.

Additions and the processing element plug-in

Possible additions to the processing element repertoire are also under consideration within the ICC. Two future expansion element types were included for expansion purposes. These elements encode a single signature value and have no prescribed operations. They simply pass the channel data to the next processing element.

Helper functions

Here are some functions that may be useful.

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

Colorimetric space conversions

```
void cmsXYZ2xyY(cmsCIExyY* Dest, const cmsCIEXYZ* Source);
void cmsxyY2XYZ(cmsCIEXYZ* Dest, const cmsCIExyY* Source);
void cmsXYZ2Lab(const cmsCIEXYZ* WhitePoint, cmsCIELab* Lab, const cmsCIEXYZ* xyz);
void cmsLab2XYZ(const cmsCIEXYZ* WhitePoint, cmsCIEXYZ* xyz, const cmsCIELab* Lab);
void cmsLab2LCh(cmsCIELCh*LCh, const cmsCIELab* Lab);
void cmsLCh2Lab(cmsCIELab* Lab, const cmsCIELCh* LCh);
```

Converting encoded values

```
void cmsLabEncoded2Float(cmsCIELab* Lab, const cmsUInt16Number wLab[3]);
void cmsLabEncoded2FloatV2(cmsCIELab* Lab, const cmsUInt16Number wLab[3]);
void cmsFloat2LabEncoded(cmsUInt16Number wLab[3], const cmsCIELab* Lab);
void cmsFloat2LabEncodedV2(cmsUInt16Number wLab[3], const cmsCIELab* Lab);
void cmsXYZEncoded2Float(cmsCIEXYZ* fxyz, const cmsUInt16Number XYZ[3]);
void cmsFloat2XYZEncoded(cmsUInt16Number XYZ[3], const cmsCIEXYZ* fXYZ);
```

Linear Bradford Chromatic Adaptation



In color science, **chromatic adaptation** is the estimation of the representation of an object under a different light source than the one in which it was recorded. A common application is to find a *chromatic adaptation transform* (CAT) that will make the recording of a neutral object appear neutral while keeping other colors also looking realistic. This function implements the so called Bradford chromatic adaptation, but simplified in the blue zone to be linear and reversible. You can get CAT02 as well by using CAM02 appearance model.

```
cmsBool cmsAdaptToIlluminant(cmsCIEXYZ* Result,
                             const cmsCIEXYZ* SourceWhitePt,
                             const cmsCIEXYZ* Illuminant,
                             const cmsCIEXYZ* Value);
```

Color difference functions

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)

Delta-E metrics

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

```
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.

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

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

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.

```
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. That is the case in Little CMS.

```
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.

PostScript generation

The PostScript language does not support ICC technology. That is, it does not have constructs which allows the specification of an ICC profile in a PostScript file. Instead, from Level 2 (1989) on, PostScript contains its own color description system and from Level 3 (to be precise, from Level 2 version 2016) on, it has its own color management system, complete with profiles, rendering intents and CMM. This system, called PostScript Color Management (PCM) operates only during printing, within a PostScript RIP.

When PostScript code is transferred to the rip for printing, in the printing stream the CSA, containing the information for converting colors from the source space to the XYZ space, is also inserted. Alternatively, an EPS with a built-in CSA can be saved.

A CRD that resides in the RIP contains the information to change the XYZ coordinates into the ink percentages for that printer.

When the PostScript code with its CSA arrives at the rip, the PostScript interpreter (which acts as a color engine) converts the source colors into XYZ and from XYZ to the colors of the printer. This process is defined by the rip programmer using what PostScript calls "color rendering procedures".

The selection of the suitable CRD is the task of the PostScript interpreter, which operates as programmed. Many rips allow the selection of different CRDs according to print resolution, type of paper and rendering intent.

If the PostScript printing stream or the EPS has no inserted CSA, the rip may use a default CSA. If the rip has no resident CRD, or does not wish to use it, another may be downloaded, by the operator or by the application itself.

These functions do translate input and output profiles into Color Space Arrays (CSA) and Color Rendering Dictionaries (CRD)

Unified method to access postscript color resources

```
typedef enum { cmsPS_RESOURCE_CSA,
               cmsPS_RESOURCE_CRD } cmsPSResourceType;

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

Compatibility methods:

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

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

- 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.

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

Example:

```
Size = cmsGetPostScriptCSA(hProfile, INTENT_PERCEPTUAL, NULL, 0);
 if (Size == 0) error()

 Block = malloc(Size);
 cmsGetPostScriptCSA(hProfile, INTENT_PERCEPTUAL, Block, Size);
```

Devicelink profiles are supported, as long as input colorspace matches Lab/XYZ for CSA or output colorspace matches Lab/XYZ for CRD. This can be used in conjuntion with cmsCreateMultiprofileTransform(), and cmsTransform2DeviceLink() to embed complex color flow into PostScript.

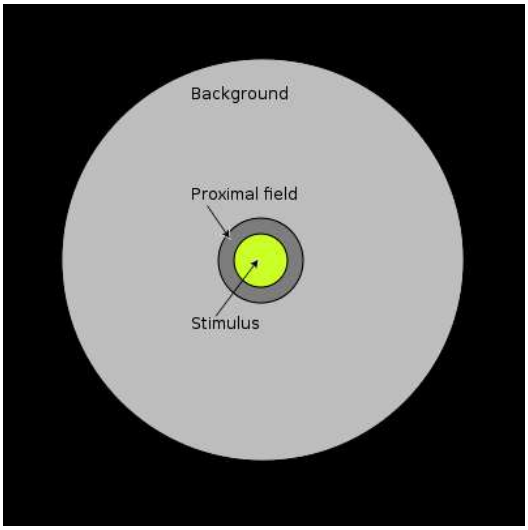
WARNING: Precision of PostScript is limited to 8 bits per sample. If you can choose between normal transforms and CSA/CRD, normal transforms will give more accurancy. However, there are situations where there is no chance.

CIECAM02

Published in 2002 by the CIE Technical Committee 8-01 (Color Appearance Modeling for Color Management Systems), as of 2010 CIECAM02 is the most recent color appearance model ratified by the CIE, and the successor of CIECAM97s.

The model input data are the adapting field luminance in cd/m2 normally taken to be 20% of the luminance of white in the adapting field), La , the relative tristimulus values of the stimulus, XYZ, the relative tristimulus values of white in the same viewing conditions, "white Point", and the relative luminance of the background, Yb . Relative tristimulus values should be expressed on a scale from Y = 0 for a perfect black to Y = 100 for a perfect reflecting diffuser.

All CIE tristimulus values are obtained using the CIE 1931 Standard Colorimetric Observer (2°).



The inner circle is the stimulus, from which the tristimulus values should be measured in CIE XYZ using the 2° standard observer. The intermediate circle is the proximal field, extending out another 2°. The outer circle is the background, reaching out to 10°, from which the relative luminance (Yb) need be measured. If the proximal field is the same color as the background, the background is considered to be adjacent to the stimulus. Beyond the circles which comprise the display field (display area, viewing area) is the surround field (or peripheral area), which can be considered to be the entire room. The totality of the proximal field, background, and surround is called the adapting field (the field of view that supports adaptation—extends to the limit of vision).

```
typedef struct {

        cmsCIEXYZ whitePoint;
        double       Yb;
        double       La;
        int          surround;
        double      D_value;

        } cmsViewingConditions;
```

Surround can be one of these:

```
#define AVG_SURROUND      1
#define DIM_SURROUND      2
#define DARK_SURROUND     3
```

D_value (adaptation degree) is any value between 0 and 1

The functions for dealing with CAM02 appearance model are:

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

void       cmsCIECAM02Done(cmsHANDLE hModel);

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

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

For example, to convert XYZ values from a given viewing condition to another:

- Create descriptions of both viewing conditions by using cmsCIECAM02Init
- Convert XYZ to JCh using cmsCIECAM02Forward for viewing condition 1
- Convert JCh back to XYZ using cmsCIECAM02Reverse for viewing condition 2
- when done, free both descriptions

```
cmsViewingConditions vc1, vc2;
cmsJCh Out;
cmsCIEXYZ In;
cmsHANDLE h1, h2;

    vc.whitePoint.X = 98.88;
    vc.whitePoint.Y = 90.00;
    vc.whitePoint.Z = 32.03;
    vc.Yb = 18;
    vc.La = 200;
    vc.surround = AVG_SURROUND;
    vc.D_value  = 1.0;

    h1 = cmsCIECAM02Init(0, &vc);


    vc2.whitePoint.X = 98.88;
    vc2.whitePoint.Y = 100.00;
    vc2.whitePoint.Z = 32.03;
    vc2.Yb = 20;
    vc2.La = 20;
    vc2.surround = AVG_SURROUND;
    vc2.D_value  = 1.0;

    h2 = cmsCIECAM02Init(0, &vc);

    In.X= 19.31;
    In.Y= 23.93;
    In.Z =10.14;

    cmsCIECAM02Forward(h1, &In, &Out);
    cmsCIECAM02Reverse(h2, &Out, &In);

    cmsCIECAM02Done(h1);
    cmsCIECAM02Done(h2);
```

See the CIECAM02 paper on CIE site for further details.

CGATS parser

Overview

This module intends to be an all-purpose parser for CGATS format. It allows reading, decoding, writing and to some extent modifying CGATS files. It supports latest CGATS.17 like multi table extensions and some additional features not yet described in the spec.

Memory management

In order to simplify the task, a memory manager is integrated within the parser. In this way, users of the module need NOT to free any memory but the parser at whole. That is, user allocates a parser and obtains a handle to it. Then, several functions accessing this handle may return pointers to memory blocks. Those memory chunks are maintained by the parser, and may be dereferenced in the lifetime of the parser. As soon as the user frees the parser by using cmsIT8Free, all memory chunks are automatically freed. This methodology has proven to be both simple and effective to deal with the complexities of associated memory.

Additions

Several additions to CGATS.17 are provided in this parser. Those are not part of the CGATS standard, but at the design time, we thought them would be very useful.

```
.INCLUDE “url_or_filename”
$INCLUDE “url_or_filename”
```

Works like #include in C language. That is, the directive will be replaced on parsing time with the contents of the file or URL referenced. Files may be nested.

```
0x 0b Hexadecimal and binary constants
```

Any numeric constant prefixed by “0x” will be understood as a hexadecimal constant. Also any numeric constant prefixed by “0b” will be understood as a binary number.

Examples:

```
PROPERTY  12                    # That's ok
PROPERTY  A_PROPERTY      # That's ok too
PROPERTY "A PROPERTY"     # That's ok, and contains whitespaces
PROPERTY A PROPERTY     # Wrong!
```

```
COEFFICIENT_A "1.2345"
HEX_VALUE "0x1234"
WITH_EXPONENT  "1.2E4"
```

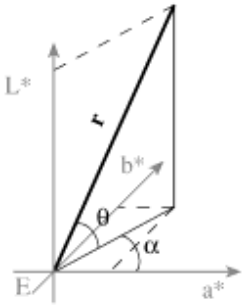
Strict CGATS

CGATS requires KEYWORD if the identifier is not in the predefined list. However, for simplicity sake, I would relax this requeriment. There is a special compilation toggle to turn off this feature.

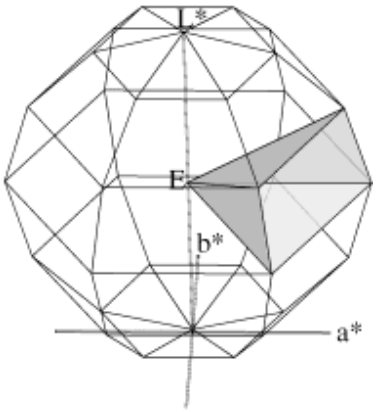
```
CMS_STRICT_CGATS
```

Gamut boundary description

The determination of gamut boundaries is a task that can be carried out for a number of motives. It can be done simply for the purpose of understanding what range of colors is present in a given image. It can be done to see what range of colors is achievable on a given color reproduction medium. It can also be done to see what color gamut is predicted by a characterization model and, hence, to see how well that model is suited for the calculation of gamut boundaries.



(a)



(b)

Little CMS implements Jan Morovic's "Segment Maxima" algorithm.

Using this method, the gamut boundary of a color reproduction medium (or an image from it) is described by a table containing the most extreme colors for each **segment** of color space. This segmentation can be carried out either in terms of *L**, *C**, and h_{ab}, or spherical coordinates whereby spherical coordinates can be calculated from orthogonal CIELAB coordinates.

A gamut boundary descriptor can be created, accessed and freed across following functions.

```
cmsHANDLE cmsGBDAlloc(cmsContext ContextID);
void      cmsGBDFree(cmsHANDLE hGBD);
cmsBool   cmsGDBAddPoint(cmsHANDLE hGBD, const cmsCIELab* Lab);
cmsBool   cmsGDBCompute(cmsHANDLE  hGDB, cmsUInt32Number dwFlags);
int       cmsGDBCheckPoint(cmsHANDLE hGBD, const cmsCIELab* Lab);
```

You first allocate an empty GBD by using the *cmsGBDAlloc* function, then add all known points with *cmsGDBAddPoint* and then interpolate missing sectors by calling the *cmsGDBCompute* function. After that you can check points to be on gamut by using *cmsGDBCheckPoint* function. GBD internal structure is hidden across a generic handle. Flags are not currently used.

For more info on that, you can find a detailed description of the algorithm in

> *“Calculating medium and image gamut boundaries for gamut mapping”*
> *Ján Morovič *, M. Ronnier Luo*
> *Colour & Imaging Institute, Kingsway House, Kingsway, Derby DE22 3HL, United Kingdom*

Conclusion

That's almost all you must know to begin experimenting with profiles, just a couple of words about the possibilities ICC profiles can give to programmers. ColorSpace profiles are valuable tools for converting from/to exotic file formats. I'm using Little CMS to read Lab TIFF using the popular Sam Leffler's TIFFLib. Also, the ability to deal with CMYK separations are much better that the infamous 1-CMY method. Abstract profiles can be used to manipulate color of images, contrast, brightness and true-gray reductions can be done fast and accurately. Grayscale conversions can be done exceptionally well, and even in tweaked colorspaces that does emulate more gray levels that the output device can effectively render.

Little CMS does all calculation on floating point basis, you can take advantage of that precision for HDR images. Some formats (TIFF for example) do support this. Photoshop as well. That can be used to efficiently emulate more than 8 bits per sample. You probably will not notice this effect on screen, but it can be seen on printed or film media.

There is a huge quantity of profiles moving around the net, and there is very good software for generating them, so future compatibility seems to be assured.

I thank you for your time and consideration.

Enjoy!

Sample 1: How to convert RGB to CMYK

This is easy. Just use a transform between RGB profile to CMYK profile.

```
#include "lcms2.h"


int main(void)
{

    cmsHPROFILE hInProfile, hOutProfile;
    cmsHTRANSFORM hTransform;
    int i;


    hInProfile  = cmsOpenProfileFromFile("sRGBColorSpace.ICM", "r");
    hOutProfile = cmsOpenProfileFromFile("MyCmyk.ICM", "r");



    hTransform = cmsCreateTransform(hInProfile,
                                    TYPE_RGB_8,
                                    hOutProfile,
                                    TYPE_CMYK_8,
                                    INTENT_PERCEPTUAL, 0);

    cmsCloseProfile(hInProfile);
    cmsCloseProfile(hOutProfile);

    for (i=0; i < AllScanlinesTilesOrWatseverBlocksYouUse; i++)
    {
        cmsDoTransform(hTransform, YourInputBuffer,
                                YourOutputBuffer,
                                YourBuffersSizeInPixels);
    }



    cmsDeleteTransform(hTransform);

    return 0;
}
```

Sample 2: How to convert from CMYK to RGB.

Just exchange profiles and format descriptors:

```
#include "lcms2.h"

int main(void)
{

    cmsHPROFILE hInProfile, hOutProfile;
    cmsHTRANSFORM hTransform;
    int i;


    hInProfile  = cmsOpenProfileFromFile("MyCmyk.icc", "r");
    hOutProfile = cmsOpenProfileFromFile("sRGBColorSpace.icc", "r");


    hTransform = cmsCreateTransform(hInProfile,
                                    TYPE_CMYK_8,
                                    hOutProfile,
                                    TYPE_RGB_8,
                                    INTENT_PERCEPTUAL, 0);

    cmsCloseProfile(hInProfile);
    cmsCloseProfile(hOutProfile);

    for (i=0; i < AllScanlinesTilesOrWatseverBlocksYouUse; i++)
    {
        cmsDoTransform(hTransform, YourInputBuffer,
                                YourOutputBuffer,
                                YourBuffersSizeInPixels);
    }


    cmsDeleteTransform(hTransform);

    return 0;
}
```

Sample 3: How to deal with Lab/XYZ spaces

This is more elaborated. There is a Lab identity Built-In profile involved.

```
#include "lcms2.h"

// Converts Lab(D50) to sRGB:

int main(void)
{

    cmsHPROFILE hInProfile, hOutProfile;
    cmsHTRANSFORM hTransform;
    int i;
    cmsUInt8Number RGB[3];
    cmsCIELab Lab;


    hInProfile  = cmsCreateLab4Profile(NULL);
    hOutProfile = cmsOpenProfileFromFile("sRGBColorSpace.icc", "r");

    hTransform = cmsCreateTransform(hInProfile,
                                    TYPE_Lab_DBL,
                                    hOutProfile,
                                    TYPE_RGB_8,
                                    INTENT_PERCEPTUAL, 0);

    cmsCloseProfile(hInProfile);
    cmsCloseProfile(hOutProfile);

    for (i=0; i < AllLabValuesToConvert; i++)
    {
        // Fill in the Float Lab

        Lab.L = Your L;
        Lab.a = Your a;
        Lab.b = Your b;

        cmsDoTransform(hTransform, &Lab, RGB, 1);

        .. Do whatsever with the RGB values in RGB[3]
    }



    cmsDeleteTransform(hTransform);

    return 0;
}
```