# *Little CMS* Plug-in API
# http://www.*littlecms*.com

*by Marti Maria*

*Ver 2.2*

# Contents

# Introduction

In computing, a plug-in consists of a computer program that interacts with a host to provide a certain, usually very specific, function "on demand". One of the main improvements in *Little CMS* 2.x is the ability to use such plug-in architecture. By using plug-ins you can use the normal API to access customized functionality. Licensing is another compelling reason; you can move all your intellectual property into plug-ins and be able to upgrade the core *Little CMS* library, keeping it in the open source side.

There are 9 types of plug-ins currently supported:

- Memory management
- Interpolation
- Tone curve types
- Formatters
- Tag types
- Tags
- Rendering intents
- Multi processing elements
- Optimizations

This manual details how to write any of those 9 types of plug-ins. It does not discuss how to use them, as plug-ins are basically extensions of the base system and as such, works in the same way.

The way Plug-in architecture is designed hides the internal implementation to the user. A plug-in user only sees a single object, which is the entry point for all plug-in libraries. This is done in such way for plug-in collections coming for 3rd parties: They publish the entry point, and the users of those collections need only to “plug” this entry point into the core engine to get the intended functionality.

## Using plug-ins from the core engine

Plug-ins are “plugged” to *Little CMS* by using a single function:

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

The "Plugin" parameter may hold one or several plug-ins, as defined by the plug-in developer. To write plug-ins, there is an additional include file lcms2_plugin.h, which declares functions not in the public API but in the plug-in internal API. For example I/O access, matrix manipulation, and all the types needed to populate the plug-in structures. Those functions begin with the prefix "_cms" (note the leading underscore) to denote those are extended functionality and should not be called by the application directly. Plug-in registering may take some memory. If you want to do a total cleanup before exit and therefore free this memory you may want to call:

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

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

## Plug-in packages

Despite it is declared as a void pointer, cmsPlugIn needs some structured data to deal with. Let's take a look on the internals of the structure accepted by **cmsPlugin**()

```
typedef struct _cmsPluginBaseStruct {

    cmsUInt32Number             Magic;
    cmsUInt32Number             ExpectedVersion;
    cmsUInt32Number             Type;
    struct _cmsPluginBaseStruct* Next;

} cmsPluginBase;
```

Your package has to export a pointer to such structure. On depending on the plug-in type, there may be some extra fields after this base.

For example, a tag plug-in definition has this structure:

```
typedef struct {

   cmsPluginBase      base;
   cmsTagSignature    Signature;
   cmsTagDescriptor Descriptor;

} cmsPluginTag;
```

As you can see, the **cmsPluginBase** struct always begin the definition block.

## Example:

Imagine you are a printer vendor and want to include in your profiles a private tag for storing the ink consumption. So, you register a private tag with the ICC, and you get the private signature "inkc". Ok, now you want to store this tag as a **Lut16Type**, so it will be driven by PCS and will return one channel giving the relative ink consumption by color. Writing a plug-in in *Little CMS* 2 will allow **cmsReadTag** and c**msWriteTag** to deal with you new data exactly as any other standard tag.

To do so, you have to fill a **cmsPluginTag** structure to declare the plug-in. This structure includes the base, which is common to all plug-ins:

```
cmsPluginTag plugin;

plugin.base.Magic = cmsPluginMagicNumber;
plugin.base.ExpectedVersion = 2000;
plugin.base.Type = cmsPluginTagSig;
plugin.base.Next = NULL;
```

That latter identifies your plug-in as "tag type". Now you have to set the additional fields that only apply to “tag definition” plug-ins. This can be done with following code:

```
plugin.Signature = inkc_constant;
plugin.Descriptor.ElemCount = 1;
plugin.Descriptor.nSupportedTypes = 1;
plugin.Descriptor.SupportedTypes[0] = cmsSigLut16Type;
```

This adds some additional info about the *type* used by your tag:

- How many elements of that type the tag is going to hold (usually one)
- In how many different types the tag may come (again, usually one)
- And then the needed type(s).

That is all. You can “plug” the new functionality to *Little CMS* by calling:

```
cmsPlugin(&plugin);
```

## Multiple plug-ins with same entry point

As a plug-in writer, you may want to encapsulate several plug-in in the same package. This is easy with the way *Little CMS* deals with plug-ins. In the "Next" field of base plug-in, you can place a pointer to the next plug-in you want to register:

```
cmsPluginTag plugin;

plugin.base.Magic = cmsPluginMagicNumber;
plugin.base.ExpectedVersion = 2000;
plugin.base.Type = cmsPluginTagSig;
plugin.base.Next = (cmsPluginBase*) &secondPlugin;
```

In this way, your package may have unlimited plug-ins, and all will be registered with a single call to **cmsPlugin**. Last plug-in in the chain must have a **NULL** in the "Next" field.

## Placing plug-ins in a separate DLL (Windows® only)

The plug-in API is exported by the main lcms2.dll using PASCAL convention. This is the normal way DLL does work. That means, you can place your plug-in packages in a separate DLL and therefore you can keep the standard, non-customized *Little CMS* DLL safe for future upgrades. Of course you can also put the plug-ins in a single DLL, but keeping both systems isolated can be handy. In this way lcms2.dll can be upgraded to a new revision and your plug-in dll, if properly written, will keep working. All the details are handled by the header files. All what you have to do is to compile your plug-in DLL using the **`CMS_DLL`** toggle, as plug-in package is basically a client of the core engine. When placing plug-ins in a separate DLL, make sure to handle memory with the provided Plug-in memory management API. Failure to do so may yield unexpected results.

## Requirements

*Little CMS* requires C99 to compile, and to write plug-ins your compiler should support following include files (they are very common)

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

You have not to include any of those files, just the file lcms2_plugin.h, which will take care of including all necessary requirements.

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

## Plugin structure

Any plug-in should be declared with at least these common fields. On depending on the type, additional fields would be required.

```
typedef struct _cmsPluginBaseStruct {

    cmsUInt32Number             Magic;
    cmsUInt32Number             ExpectedVersion;
    cmsUInt32Number             Type;
    struct _cmsPluginBaseStruct* Next;

} cmsPluginBase;
```

*Magic:*

Identifies the structure as a *Little CMS* 2 plug-in. It must contain following value:

```
cmsPluginMagicNumber      0x61637070     'acpp'
```

*ExpectedVersion:*

The expected *Little CMS* version; 2000 in current release. *Little CMS* core will accept plug-ins with expected version less or equal that the core version. If a plug-in is marked for a version greater that the core, plug-in will be rejected. That means downgrading core engine may disable certain plug-ins (as it should be).

*Type:*

It defines behaviour of plug-in. There are 9 plug-in types currently defined:

| Type | Hex | ASCII |
|---|---|---|
| cmsPluginMemHandlerSig | 0x6D656D48 | 'memH' |
| cmsPluginInterpolationSig | 0x696E7048 | 'inpH' |
| cmsPluginParametricCurveSig | 0x70617248 | 'parH' |
| cmsPluginFormattersSig | 0x66726D48 | 'frmH |
| cmsPluginTagTypeSig | 0x74797048 | 'typH' |
| cmsPluginTagSig | 0x74616748 | 'tagH' |
| cmsPluginRenderingIntentSig | 0x696E7448 | 'intH' |
| cmsPluginMultiProcessElementSig | 0x6D706548 | 'mpeH' |
| cmsPluginOptimizationSig | 0x6F707448 | 'optH' |

*Next:*

Points to the next plug-in header in multi plug-in packages. Set it to **NULL** to mark end of chain.

## Memory management plugin

By using this plug-in type, a programmer can override memory management done by *Little CMS*. Multiple occurrences of this type of plug-in are allowed, but each time a plug-in of this type is set, it replaces the old one.

*Type:*

```
cmsPluginMemHandlerSig     0x6D656D48    'memH'
```

*Plug-in header structure:*

```
typedef struct  {

     cmsPluginBase base;

     // Required
     void * (* MallocPtr)(cmsContext ContextID, cmsUInt32Number size);
     void    (* FreePtr)(cmsContext ContextID, void *Ptr);
     void * (* ReallocPtr)(cmsContext ContextID, void* Ptr,
                                        cmsUInt32Number NewSize);
     // Optional
     void * (* MallocZeroPtr)(cmsContext ContextID, cmsUInt32Number size);
     void * (* CallocPtr)(cmsContext ContextID, cmsUInt32Number num,
                                        cmsUInt32Number size);
     void * (* DupPtr)(cmsContext ContextID, const void* Org,
                                        cmsUInt32Number size);

} cmsPluginMemHandler;
```

Setting optional function pointers to NULL forces *Little CMS* to use **MallocPtr, FreePtr and ReallocPtr** functions for all operation. If you provide all set of functions, *Little CMS* will use the optional memory operations when possible. This works in such way to allow optimizations when using advanced memory managers. All functions get called with a **ContextID** that identifies the calling environment. It may be zero on certain special cases. This **ContextID** is provided by the user when calling *Little CMS* API functions.

Plug-ins should NOT call those functions directly. They should manage memory by calling the plug-in memory management API, described below. This API does call this plug-in to do its functionality. Changing memory managers with ongoing operations may yield unexpected results.

*Example*:

```
#include “lcms2_plugin.h”

static void* my_malloc(cmsContext ContextID, cmsUInt32Number size)
{
    return malloc(size);
}

static void my_free(cmsContext ContextID, void *Ptr)
{
    free(Ptr);
}

static void* my_realloc(cmsContext ContextID,
                        void *Ptr, cmsUInt32Number new_size)
{
    return realloc(Ptr, new_size);
}

cmsPluginMemHandler MemHandler = {{
                                    cmsPluginMagicNumber,
                                    2000,
                                    cmsPluginMemHandlerSig,
                                    NULL
                                   },
                                   my_malloc,
                                   my_free,
                                   my_realloc,
                                   NULL,
                                   NULL,
                                   NULL };
```

This example changes the internal *Little CMS* memory management to use plain C malloc(), free() and realloc() functions. This is indeed a bad idea, as the internal memory manager does some extra checks to make sure no overflow exploits are being tried, but you may want to use this capability to do other things, like use your own memory manager or to access out-of board memory in embedded systems. In the test bed application, a customized memory manager which adds extra levels of check is being used. You can refer to this program for a more sophisticated example of memory manager replacement.

## Interpolation plug-in

By using this plug-in type, programmer may change or increase the interpolation done by *Little CMS*. To fully understand what means this, it is necessary to clarify some concepts now.

*Little CMS* internal operation is based on **pipelines**. Each pipeline may contain a number of stages. Those stages may be of several kinds, and there are two kinds which need interpolation. One is tone curves, where a 1D curve is applied to each channel. The second kind is multidimensional lookup tables (CLUT) where a number of channels are interpolated across a multidimensional grid. In each one of those cases, the final value is interpolated across a number of nodes. By using the interpolation plug-in you can change the algorithm that applies in such cases. Please note that does NOT apply to the whole pipeline, only to the specific steps that are using interpolation. If you want to accelerate the pipeline evaluation by using some sort of ASIC or GPU, that is certainly possible by using the optimization or the applier plug-ins, but not changing the interpolation.

The structure is based on the idea of interpolator factory. That is, the programmer supplies a call back function. When *Little CMS* needs to do some interpolation, it calls this function specifying the number of input and output channels, the base type (16 bits or floating point) and gives some hints about the use it want to do to the routine, like to use a trilinear-like. The factory then should return a function pointer if the plug-in implements this particular interpolation or NULL to regret. In this case, the default interpolator provided by *Little CMS* will be used instead. Only one factory can be set at time. Further calls to **cmsPlugin** with this type will replace the behavior of previous plug-in. Interpolators have no states and no memory, and therefore cannot hold private data.

*Type:*

```
cmsPluginInterpolationSig       0x696E7048    'inpH'
```

There is a limitation on the maximum input dimensions:

```
#define MAX_INPUT_DIMENSIONS 8
```

That is indeed necessary because tables of more than those dimensions are so huge that grown out of control when node count increases.

*Plug-in header structure:*

```
typedef struct {

   cmsPluginBase              base;
   cmsInterpFnFactory         InterpolatorsFactory;

} cmsPluginInterpolation;
```

*This is the definition of the factory*

```
cmsInterpFunction (* cmsInterpFnFactory)(cmsUInt32Number nInputChannels,
                                         cmsUInt32Number nOutputChannels,
                                         cmsUInt32Number dwFlags);
```

*And here are the possible flags*

```
#define CMS_LERP_FLAGS_16BITS      0x0000
#define CMS_LERP_FLAGS_FLOAT       0x0001
#define CMS_LERP_FLAGS_TRILINEAR   0x0100
```

Since the interpolators may have different parameter types on float and 16 bits, the factory returns a union of pointers, although at the end this behaves just a single pointer.

```
typedef union {
   _cmsInterpFn16             Lerp16;
   _cmsInterpFnFloat          LerpFloat;
} cmsInterpFunction;
```

Interpolators for 16 bits and floating point are very alike. They have, however, some differences due the fact 16 bits are primarily intended for performance (throughput)

## 16 bits interpolation.

The returned function has to perform precision-limited interpolation and is supposed to be quite fast. Reference Implementations are tetrahedral or trilinear, and plug-ins may choose to implement any other interpolation algorithm.

```
void (* _cmsInterpFn16)(register const cmsUInt16Number Input[],
                        register cmsUInt16Number Output[],
                        register const struct _cms_interp_struc* p);
```

## Floating point interpolation.

The returned function has to perform full precision interpolation using floats. This is not a time critical function. Reference Implementations are tetrahedral or trilinear, and plug-ins may choose to implement any other interpolation algorithm.

```
void (* _cmsInterpFnFloat)(cmsFloat32Number const Input[],
                           cmsFloat32Number Output[],
                           const struct _cms_interp_struc* p);
```

## Interpolation parameters

When an interpolator is called, *Little CMS* provides a pointer to several pre-computed parameters to help the interpolation task. Is up to the interpolator to use such parameters or ignore them.

```
typedef struct _cms_interp_struc {

   cmsContext           ContextID;

   cmsUInt32Number  dwFlags;
   cmsUInt32Number  nInputs;
   cmsUInt32Number  nOutputs;
   cmsUInt32Number  nSamples[MAX_INPUT_DIMENSIONS];
   cmsUInt32Number  Domain[MAX_INPUT_DIMENSIONS];
   cmsUInt32Number  opta[MAX_INPUT_DIMENSIONS];
   const void *         Table;

   cmsInterpFunction  Interpolation;

} cmsInterpParams;
```

*dwFlags : A copy of the flags specified when requesting the interpolation*
*nInputs, nOutputs: Channels on input and output.*
*nSamples[]: Number of grid points in each input dimension*
*Domain[]:  Number of grid points minus one in each input dimension.*
*Opta[]:  The result of multiplying Domain[n]*Opta[n-1] (offset in the table in base type).*
*Table: Points to a portion of memory holding the table of gridpoints*
*Interpolation: points to the interpolator itself*

*Example:*

```
void LinLerp1Dfloat(const cmsFloat32Number Value[],
                    cmsFloat32Number Output[],
                    const cmsInterpParams* p)
{
       cmsFloat32Number y1, y0;
       cmsFloat32Number val2, rest;
       int cell0, cell1;
       const cmsFloat32Number* LutTable = p ->Table;

       if (Value[0] == 1.0) {
           Output[0] = LutTable[p -> Domain[0]]; return; }

       val2 = p -> Domain[0] * Value[0];
       cell0 = (int) floor(val2);
       cell1 = (int) ceil(val2);
       rest = val2 - cell0;
       y0 = LutTable[cell0] ;
       y1 = LutTable[cell1] ;

       Output[0] = y0 + (y1 - y0) * rest;
}

cmsInterpFunction
my_Interpolators_Factory(cmsUInt32Number nInputChannels,
                         cmsUInt32Number nOutputChannels,
                         cmsUInt32Number dwFlags)
{
    cmsInterpFunction Interpolation;
    cmsBool  IsFloat = (dwFlags & CMS_LERP_FLAGS_FLOAT);

    memset(&Interpolation, 0, sizeof(Interpolation));

    if (nInputChannels == 1 && nOutputChannels == 1 && IsFloat) {

        Interpolation.LerpFloat = LinLerp1Dfloat;
    }

    return Interpolation;
}

cmsPluginInterpolation Plugin = {
                { cmsPluginMagicNumber,
                  2000,
                  cmsPluginInterpolationSig,
                  NULL },
                  my_Interpolators_Factory };
```

## Parametric curves plug-in

By using this plug-in type, programmer may increase or replace the list of supported parametric tone curves. You can access this new type by using **cmsBuildParametricToneCurve**, as well as "curves" pipeline stage. Each call to **cmsPlugin** with this type will add new curves to the list if the type ID is not used. If the type ID already exists, the tone curve implementation is replaced.

*Type*:

```
cmsPluginParametricCurveSig       0x70617248     'parH'
```

There is a limit on the number of curves a single plug-in can describe:

```
#define MAX_TYPES_IN_LCMS_PLUGIN    20
```

If you need more curves types, you can use two linked plug-ins, as described above.

*Structure:*

```
typedef struct {
  cmsPluginBase          base;

  cmsUInt32Number    nFunctions;
  cmsUInt32Number    FunctionTypes[MAX_TYPES_IN_LCMS_PLUGIN];
  cmsUInt32Number    ParameterCount[MAX_TYPES_IN_LCMS_PLUGIN];

  cmsParametricCurveEvaluator Evaluator;

} cmsPluginParametricCurves;
```

*Evaluator:*

*callback for user-supplied parametric curves.*

May implement more than one type, and have to implement evaluation of the curve in both, forward and reverse directions.

```
cmsFloat64Number (* cmsParametricCurveEvaluator)
                                        ( cmsInt32Number Type,
                                          const cmsFloat64Number Params[10],
                                          cmsFloat64Number R);
```

Note the **Type** parameter is described as signed. A negative type means same function but analytically inverted. Max. number of params is 10. Each parametric curve plug-in may implement an arbitrary number of curve types, up to 20:

Since *Little CMS* can work as unbounded CMM, the domain of R is effectively from minus infinite to infinite. However, the normal, in-range domain is 0…1.0, so you have to normalize your function to get values of R = [0...1.0] and deal with remaining cases if you want your function to be able to work in unbounded mode.

*FunctionTypes*:

Id's for each parametric curve described by the plug-in

*ParameterCount*:

Number of parameters for each parametric curve described by the plug-in

*Example*:

```
#include “lcms2_plugin.h”

#define TYPE_SINH  1000

static
cmsFloat64Number my_fns(cmsInt32Number Type,
                        const cmsFloat64Number Params[],
                        cmsFloat64Number R)
{
    switch (Type) {

    case TYPE_SINH:
        Val = Params[0]* sinh(R);
        break;

    case -TYPE_SINH:
        Val = asinh(R) / Params[0];
        break;
     }

   return Val;
}

cmsPluginParametricCurves NewCurvePlugin = {
                                    {
                                      cmsPluginMagicNumber,
                                      2000,
                                      cmsPluginParametricCurveSig,
                                      NULL
                                     },
                                     1,
                                     {TYPE_SINH},
                                     {1},
                                     my_fns};
```

This example adds a new parametric curve under the ID number of 1000. This is a basic hyperbolic function, the hyperbolic sine "sinh" multiplied by the first parameter. Math expression of function is $f(x) = p_0 \sinh(x)$ the implementation adds an analytical reversing of the curve when parametric curve is requested with a negative id.

## Formatters Plug-in

*Little CMS* can handle a lot of formats of image data. For describing such formats, *Little CMS* does use a 32-bit value, referred below as format specifier. Each bit in those 32 bits has specific meaning:

**O TTTTT U Y F P X S EEE CCCC BBB**

| |
|---|
| O: Reserved, internal use only |
| T: Pixel type |
| F: Flavor 0=MinIsBlack(Chocolate) 1=MinIsWhite(Vanilla) |
| P: Planar? 0=Chunky, 1=Planar |
| X: swap 16 bps endianess? |
| S: Do swap? ie, BGR, KYMC |
| E: Extra samples |
| C: Channels (Samples per pixel) |
| B: bytes per sample |
| Y: Swap first - changes ABGR to BGRA and KCMY to CMYK |

This plug-in adds new format handlers, replacing them if they already exist.

*Type*:

```
cmsPluginFormattersSig          0x66726D48    'frmH'
```

Plug-in may implement an arbitrary number of formatters by implementing a format factory.

*Structure:*

```
typedef struct {
   cmsPluginBase           base;
   cmsFormatterFactory     FormattersFactory;

} cmsPluginFormatters;
```

```
typedef enum { cmsFormatterInput=0,
               cmsFormatterOutput=1 } cmsFormatterDirection;

#define CMS_PACK_FLAGS_16BITS      0x0000
#define CMS_PACK_FLAGS_FLOAT       0x0001
```

*Factory callback:*

```
cmsFormatter (* cmsFormatterFactory)(cmsUInt32Number Type,
                                     cmsFormatterDirection Dir,
                                     cmsUInt32Number dwFlags);
```

The factory have to return a cmsFormatter type. This type holds a pointer to a formatter that can be either 16 bits or 32 bit float.

```
typedef union {

   cmsFormatter16    Fmt16;
   cmsFormatterFloat FmtFloat;

} cmsFormatter;
```

Formatters dealing with floats (bps = 4) or double (bps = 0) types are requested via FormatterFloat callback. Others come across Formatter16 callback.

```
cmsUInt8Number* (* cmsFormatter16)
                        (register struct _cmstransform_struct* CMMcargo,
                         register cmsUInt16Number Values[],
                         register cmsUInt8Number*  Buffer,
                         register cmsUInt32Number Stride);

cmsUInt8Number* (* cmsFormatterFloat)
                       (struct _cmstransform_struct* CMMcargo,
                        cmsFloat32Number Values[],
                        cmsUInt8Number*  Buffer,
                        cmsUInt32Number Stride);
```

*Example:*

```
unsigned char* my_Unroll8(register void* nfo,
                          register cmsUInt16Number wIn[],
                          register cmsUInt8Number* accum,
                          register cmsUInt32Number Stride)
{
    wIn[0] = accum[0]) << 8;
    wIn[1] = (accum[1] + 128) << 8;
    wIn[2] = (accum[2] + 128) << 8;
    return accum + 3;
}

cmsFormatter my_FormatterFactory(cmsUInt32Number Type,
                                 cmsFormatterDirection Dir,
                                 cmsUInt32Number dwFlags)
{
    cmsFormatter Result = { NULL };

    if ((Type == TYPE_My_Lab) &&
        !(dwFlags & CMS_PACK_FLAGS_FLOAT) &&
        (Dir == cmsFormatterInput)) {
            Result.Fmt16 = my_Unroll8;
    }
    return Result;
}

cmsPluginFormatters Plugin = { {cmsPluginMagicNumber,
                               2000,
                               cmsPluginFormattersSig,
                               NULL},
                               my_FormatterFactory };
```

This example implements decoding a new format of Lab values. The format comes as L [0..FF] and a and b as signed chars.

## Tag plug-in

This is the tag plugin, which identifies new tags with existing types. This plug-in has been discussed as an example at the beginning of this document.

```
cmsPluginTagSig              0x74616748    'tagH'
```

Each Plug-in implements a single tag:

```
typedef struct {
  cmsPluginBase     base;

  cmsTagSignature  Signature;
  cmsTagDescriptor Descriptor;

} cmsPluginTag;
```

This function should return the desired type for this tag, given the version of profile and the data being serialized.

```
typedef struct {

  cmsUInt32Number    ElemCount;       // If this tag needs an array
                                      // how many elements should keep

  // For reading.
  cmsUInt32Number    nSupportedTypes;   // In how many types this tag can come

  cmsTagTypeSignature SupportedTypes[MAX_TYPES_IN_LCMS_PLUGIN];

  // For writting
  cmsTagTypeSignature (* DecideType)(double ICCVersion, const void *Data);

} cmsTagDescriptor;
```

*DecideType: Callback to select the type based on the version of the ICC profile. It got called on writing operations. 'data' is a pointer to the tag contents, i.e., the data supplied by the user to cmsWriteTag().*

*Example:*

```
#define inkc_constant 0x696E6B43

cmsPluginTag plugin = {
                       {cmsPluginMagicNumber,
                        2000,
                        cmsPluginTagSig, NULL},
                       {inkc_constant,
                        1, 1, {cmsSigLut16Type}, NULL}
                      };
```

## Tag type handler plug-in

Tag type plug-in complements tag plug-in by adding new types. Types are responsible of the structure returned when *cmsReadTag* is called. Each type is free to return anything it wants, and it is up to the caller to know in advance what is the type contained in the tag.

*Type*:

```
cmsPluginTagTypeSig          0x74797048    'typH'
```

*Each plug-in implements a single type*

```
typedef struct {

    cmsPluginBase        base;
    cmsTagTypeHandler    Handler;

} cmsPluginTagType;
```

To add a new type, programmer has to implement several callbacks for reading, writing, duplicating and setting free the in-memory representation of the type. When designing a new type, the first step should be to create a structure to hold the representation of data. This structure may be the same as used to serialize on disk, but usually that is not the case. Once the programmer has written all callback functions, she has to fill the handler structure with pointers to those routines.

There is a copy of ContextID in the tag type handler structure. This member is there for simplicity sake, and the plug-in developer may read this value, but needs not to initialize it. *Little CMS* will set this member to proper value when invoking the plug-in.

*The type handler structure:*

```
typedef struct _cms_typehandler_struct {

    cmsTagTypeSignature Signature;    // The signature of the type

    // Allocates and reads items
    void *   (* ReadPtr)(struct _cms_typehandler_struct* self,
                         cmsIOHANDLER*     io,
                         cmsUInt32Number*  nItems,
                         cmsUInt32Number   SizeOfTag);

    // Writes n Items
    cmsBool  (* WritePtr)(struct _cms_typehandler_struct* self,
                          cmsIOHANDLER*    io,
                          void*            Ptr,
                          cmsUInt32Number  nItems);

    // Duplicate an item or array of items
    void*   (* DupPtr)(struct _cms_typehandler_struct* self,
                       const void *Ptr,
                       cmsUInt32Number n);

    // Free all resources
    void    (* FreePtr)(struct _cms_typehandler_struct* self,  void *Ptr);

    // The calling thread
    cmsContext    ContextID;

} cmsTagTypeHandler;
```

*Signature: Identifies the type being implemented by the plug-in.*
*ReadPtr: Pointer to read function.*
*WritePtr: Pointer to write callback*
*DupPtr: Pointer to Duplicate callback*
*FreePtr: Pointer to Free callback*
*ContextID: (ReadOnly) contains the context for the last function accessing the plug-in*

*Example:*

```
void *Type_int_Read(struct _cms_typehandler_struct* self,
                    cmsIOHANDLER* io,
                    cmsUInt32Number* nItems,
                    cmsUInt32Number SizeOfTag)
{
    int* Ptr = (int*) _cmsMalloc(self ->ContextID, sizeof(int));
    if (Ptr == NULL) return NULL;
    if (!_cmsReadIntU32Number(io, Ptr)) return NULL;
    *nItems = 1;
    return Ptr;
}

cmsBool Type_int_Write(struct _cms_typehandler_struct* self,
                       cmsIOHANDLER* io,
                       void* Ptr, cmsUInt32Number nItems)
{
    return _cmsWriteUInt32Number(io, *(cmsUInt32Number*) Ptr);
}

void* Type_int_Dup(struct _cms_typehandler_struct* self,
                   const void *Ptr, cmsUInt32Number n)
{
    return _cmsDupMem(self ->ContextID, Ptr, n * sizeof(int));
}

void Type_int_Free(struct _cms_typehandler_struct* self,
                   void* Ptr)
{
    _cmsFree(self ->ContextID, Ptr);
}

cmsPluginTagType Plugin = { {cmsPluginMagicNumber,
                             2000,
                             cmsPluginTagTypeSig,
                             NULL},
                             { SigIntType,
                               Type_int_Read,
                               Type_int_Write,
                               Type_int_Dup,
                               Type_int_Free,
                               NULL
                              }};
```

## Intent plug-in

This plug-in implements new rendering intents. To do so, the callback has to join all profiles specified in the array in a single pipeline doing any necessary adjustments. Any custom intent in the chain redirects to the custom callback. If more than one custom intent is found, the one located first is invoked. Usually users should use only one custom intent, so mixing custom intents in same multiprofile transform is not supported.

```
cmsPluginRenderingIntentSig       0x696E7448    'intH'
```

Each plug-in defines a single intent number.

```
typedef struct {

   cmsPluginBase        base;

   cmsUInt32Number   Intent;
   cmsIntentFn          Link;
   char                    Description[256];

} cmsPluginRenderingIntent;
```

Plug-in has to specify a linker callback that accepts a chain of profiles and return a pipeline implementing the new intent. All significant modificators are passed as parameters.

```
cmsPipeline* (* cmsIntentFn)( cmsContext          ContextID,
                              cmsUInt32Number  nProfiles,
                              cmsUInt32Number  Intents[],
                              cmsHPROFILE        hProfiles[],
                              cmsBool               BPC[],
                              cmsFloat64Number  AdaptationStates[],
                              cmsUInt32Number   dwFlags);
```

## Support functions for intent plug-ins

The default ICC intents (perceptual, saturation, rel.col and abs.col)

2.0

```
cmsPipeline* _cmsDefaultICCintents(cmsContext          ContextID,
                                   cmsUInt32Number     nProfiles,
                                   cmsUInt32Number     Intents[],
                                   cmsHPROFILE         hProfiles[],
                                   cmsBool             BPC[],
                                   cmsFloat64Number    AdaptationStates[],
                                   cmsUInt32Number     dwFlags);
```

This function implements the standard ICC intents perceptual, relative colorimetric, saturation and absolute colorimetric. Can be used as a basis for custom intents.

***Parameters:***

*ContextID: Pointer to a user-defined context cargo.*
*nProfiles: Number of profiles in the chain*
*Intents[]: Intent to apply on each profile to profile joint.*
*hProfiles[]: Handles to open profiles*
*BPC[]: Array of black point compensation states for each profile to profile joint*
*AdaptationStates[]: Array of observer adaptation states for each profile to profile joint.*
*dwFlags: color transform flags (see Little CMS API for further details)*

**Returns:**

*A pointer to a newly created pipeline holding the color transform on success, NULL on error.*

*Example:*

```
cmsPipeline*  MyNewIntent(cmsContext      ContextID,
                          cmsUInt32Number nProfiles,
                          cmsUInt32Number TheIntents[],
                          cmsHPROFILE     hProfiles[],
                          cmsBool         BPC[],
                          cmsFloat64Number AdaptationStates[],
                          cmsUInt32Number dwFlags)
{
    cmsPipeline*    Result;
    cmsUInt32Number ICCIntents[256];
    cmsUInt32Number i;

 for (i=0; i < nProfiles; i++)
       ICCIntents[i] = (TheIntents[i] == 300) ? INTENT_PERCEPTUAL :
                                                TheIntents[i];

 if (cmsGetColorSpace(hProfiles[0]) != cmsSigGrayData ||
     cmsGetColorSpace(hProfiles[nProfiles-1]) != cmsSigGrayData)
           return _cmsDefaultICCintents(ContextID, nProfiles,
                                        ICCintents, hProfiles,
                                        BPC, AdaptationStates,
                                        dwFlags);

    Result = cmsPipelineAlloc(ContextID, 1, 1);
    if (Result == NULL) return NULL;

    cmsPipelineInsertStage(Result, cmsAT_BEGIN,
                           cmsStageAllocIdentity(ContextID, 1));

    return Result;
}

cmsPluginRenderingIntent RIPlugin =
                    {cmsPluginMagicNumber,
                     2000,
                     cmsPluginRenderingIntentSig,
                     NULL},
                     300,
                     MyNewIntent,
                       "bypass gray to gray rendering intent" };
```

This example creates a new rendering intent, at intent number 300, that is identical to perceptual intent for all color spaces but gray to gray transforms, in this case it bypasses the data. Note that it has to clear all occurrences of intent 300 in the intents array to avoid infinite recursion.

# Stages

When dealing with pipelines, there is the possibility for the programmer to create new, customized stages that cannot be modeled by using any of the yet existing steps. Additionally, there is a plug-in type that allows saving such user defined stages as multi profile elements in DToB/BToD tags.

## Creating new stage types

To create a new Stage type, following function must be used:

2.0

```
cmsStage* _cmsStageAllocPlaceholder(cmsContext           ContextID,
                                    cmsStageSignature    Type,
                                    cmsUInt32Number      InputChannels,
                                    cmsUInt32Number      OutputChannels,
                                    _cmsStageEvalFn      EvalPtr,
                                    _cmsStageDupElemFn   DupElemPtr,
                                    _cmsStageFreeElemFn  FreePtr,
                                    void*                Data);
```

***Parameters:***

*ContextID: Pointer to a user-defined context cargo.*
*Type: Identifier for the new stage type.*
*InputChannels, OutputChannels: Number of channels for this stage.*
*EvalPtr: Callback to evaluate the stage.*
*DupElemPtr: If user data is being used, callback to duplicate the data.*
*FreePtr: If user data is being used, callback to set data free.*
*Data: Pointer to user-defined data or NULL if no data is needed.*

**Returns:**

*A pointer to the newly created stage on success, NULL on error.*

The Stage element can accept private data. If so, you need to supply callback functions to duplicate and free private data.

```
typedef void (* _cmsStageEvalFn)    (const cmsFloat32Number In[],
                                     cmsFloat32Number Out[],
                                     const cmsStage* mpe);

typedef void*(* _cmsStageDupElemFn)  (cmsStage* mpe);

typedef void (* _cmsStageFreeElemFn) (cmsStage* mpe);
```

*Example:*

```
void EvaluateNegate(const cmsFloat32Number In[],
                    cmsFloat32Number Out[],
                    const cmsStage *mpe)
{
    Out[0] = 1.0 - In[0];
    Out[1] = 1.0 - In[1];
    Out[2] = 1.0 - In[2];
}

#define SigNegate ((cmsStageSignature)0x6E202020)

cmsStage* StageAllocNegate(cmsContext ContextID)
{
    return _cmsStageAllocPlaceholder(ContextID,
                SigNegateType, 3, 3, EvaluateNegate,
                NULL, NULL, NULL);
}
```

*This example creates a stage that reverses the input (negative). It works on 3 → 3 channels.*

## Stages plug-in.

This plug-in type allows programmer to add new multi-process elements in the MPE tag, and in this way extend the types documented in "*Floating-Point Device Encoding Range*" addendum to ICC spec 4.2.

Using this plug-in allows such new stages to be stored on profiles.

*Type*:

```
cmsPluginMultiProcessElementSig     0x6D706548    'mpeH'
```

Plug-in structure:

```
typedef struct {
    cmsPluginBase          base;
    cmsTagTypeHandler Handler;

} cmsPluginMultiProcessElement;
```

To describe the serialization, the same structure as tag type handler is being used, although there are some differences:

- DupPtr and FreePtr of cmsTagTypeHandler are not used and have to be set to NULL.
- ReadPtr must call cmsStageAllocPlaceholder to create the stage
- WritePtr can access the stage internals by using all cmsStage functions.
    - cmsStageInputChannels
    - cmsStageOutputChannels
    - cmsStageType
    - cmsStageData

*Example (as a continuation of previous sample):*

```
void *Type_negate_Read(struct _cms_typehandler_struct* self,
                       cmsIOHANDLER* io,
                       cmsUInt32Number* nItems,
                       cmsUInt32Number SizeOfTag)
{
    cmsUInt16Number   Chans;
    if (!_cmsReadUInt16Number(io, &Chans)) return NULL;
    if (Chans != 3) return NULL;

    *nItems = 1;
    return StageAllocNegate(self -> ContextID);
}

cmsBool Type_negate_Write(struct _cms_typehandler_struct* self,
                          cmsIOHANDLER* io,
                          void* Ptr, cmsUInt32Number nItems)
{

    if (!_cmsWriteUInt16Number(io, 3)) return FALSE;
    return TRUE;
}

cmsPluginMultiProcessElement Plugin = {
                           {cmsPluginMagicNumber,
                            2000,
                            cmsPluginMultiProcessElementSig,
                            NULL},
                            { SigNegate,
                              Type_negate_Read,
                              Type_negate_Write,
                              NULL,
                              NULL,
                              NULL
                             }};
```

This example creates a new multi processing element that saves our “negate” stage on DToB/BToD tags. To do so, cmsWriteTag() should be called with a pipeline containing “negate” stages.

## Optimization plug-in.

Using this plug-in, additional optimization strategies may be implemented.

To create transforms, *Little CMS* does create chains of operators by using pipelines. Once created, those pipelines are passed to the optimization engine to remove redundancies and perform any optimizations that would increase the performance/throughput of the pipeline. The optimization engine consist on series of algorithms that are applied to the pipeline chain, if suitable. By using optimization plug-in, a programmer can add new optimization algorithms to the existing list. Formats suitable for optimization are 8 and 16 bits. No optimization is possible on floating-point data.

The optimization algorith can decide to implement the evaluation of resulting pipeline in any way it wants. To do so, it has to register a specilized callback that would be responsible of evaluating the optimized version of LUT. This callback has this form:

cmsOPTeval16Fn:

```
void  <OptimizationCallback>(register const cmsUInt16Number In[],
                             register cmsUInt16Number Out[],
                             register const void* Data);
```

It is also posible to allocate and maintain an amount of user-supplied data, used only by the optimization callback. The plug-in writer, then, have to supply two additional callbacks. One for duplicating this data and another to free any resource associated with this data.

```
typedef void     (* _cmsOPTfreeDataFn)(cmsContext ContextID, void* Data);
```

```
typedef void*    (* _cmsOPTdupDataFn)(cmsContext ContextID,  const void* Data);
```

Those last functions are optional, and only required if the optimization callback is using private data.  It is the optimization algorithm which have to setup the optimized callbacack and possible user defined data. For that purpose, there is a specialized function:

2.0

```
void _cmsPipelineSetOptimizationParameters (cmsPipeline* Lut,
                                            _cmsOPTeval16Fn Eval16,
                                            void* PrivateData,
                                            _cmsOPTfreeDataFn FreePrivateDataFn,
                                            _cmsOPTdupDataFn DupPrivateDataFn);
```

Establishes the optimization parameters for a given pipeline. Private data may be NULL, and that means the optimized callback needs no additional data. If not NULL, the Free and Dup callbacks must be specified as well.

***Parameters:***

*Lut: Pipeline to be optimized*
*Eval16: User-supplied callback for fast evaluation of pipeline*
*PrivateData: Initial private data, NULL if not used.*
*FreePrivateDataFn: User-supplied callback to free private data, NULL if not used.*
*PrivateDataFn: User-supplied callback to duplicate private data, NULL if not used.*

**Returns:**

**None**

The optimization plug-in exports the optimizer algorith as a function callback. That function have to return TRUE if any optimization is done on the LUT, this terminates the optimization search. Or FALSE if it is unable to optimize and want to give a chanceto the rest of optimization algorithms.

*Type*:

```
cmsPluginOptimizationSig    0x6F707448    'optH'
```

*Structure:*

```
typedef struct {

    cmsPluginBase    base;
    _cmsOPToptimizeFn  OptimizePtr;    // Optimization algorith entry point

} cmsPluginOptimization;
```

*Optimization algorithm callback:*

```
cmsBool  (* _cmsOPToptimizeFn)(cmsPipeline** Lut,
                                cmsUInt32Number  Intent,
                                cmsUInt32Number* InputFormat,
                                cmsUInt32Number* OutputFormat,
                                cmsUInt32Number* dwFlags);
```

This function may be used to set the optional evaluator and a block of private data. If private data is being used, an optional duplicator and free functions should also be specified in order to duplicate the pipeline construct. Use NULL to inhibit such functionality.

*Example:*

```
cmsBool MyOptimize(cmsPipeline** Lut,
                   cmsUInt32Number  Intent,
                   cmsUInt32Number* InputFormat,
                   cmsUInt32Number* OutputFormat,
                   cmsUInt32Number* dwFlags)
{
    cmsStage* mpe;

    //  Only curves in this LUT?
    for (mpe = cmsPipelineGetPtrToFirstStage(Src);
         mpe != NULL;
         mpe = cmsStageNext(mpe)) {
            if (cmsStageType(mpe) != cmsSigCurveSetElemType)
                                                  return FALSE;
    }

    *dwFlags |= cmsFLAGS_NOCACHE;
    _cmsPipelineSetOptimizationParameters(*Lut,
                FastEvaluateCurves, NULL, NULL, NULL);

    Return TRUE;
}

cmsPluginOptimization Plugin = {
                         {cmsPluginMagicNumber,
                          2000,
                          cmsPluginOptimizationSig,
                          NULL},
                          MyOptimize};
```

This example detects whatever the pipeline contains only curves and in this case provides a hypothetical fast evaluator (not listed). Note that the plug-in also inhibits the 1-pixel cache, because the “FastEvaluateCurves” function is supposed to be faster than caching.

# Plug-in support API

## I/O handlers

IO handlers are abstractions used to deal with files or streams. All reading/writing of ICC profiles, PostScript resources and CGATS are done across IO handlers. IO handlers do support random access. The IO handler API allows you to access the low level functions, as well as to write new handlers for specialized devices.

*IO handler structure.*

```
struct _cms_io_handler {

   void*                stream;
   cmsContext           ContextID;
   cmsUInt32Number      UsedSpace;
   cmsUInt32Number      ReportedSize;

   char                 PhysicalFile[cmsMAX_PATH];

   cmsUInt32Number      (* Read)(struct _cms_io_handler* iohandler, void *Buffer,
                                 cmsUInt32Number size,
                                 cmsUInt32Number count);
   cmsBool              (* Seek)(struct _cms_io_handler* iohandler,
                                 cmsUInt32Number offset);
   cmsBool              (* Close)(struct _cms_io_handler* iohandler);
   cmsUInt32Number      (* Tell)(struct _cms_io_handler* iohandler);
   cmsBool              (* Write)(struct _cms_io_handler* iohandler,
                                 cmsUInt32Number size,  const void* Buffer);
};
```

## Read/Write functions

2.0

```
cmsBool  _cmsReadUInt8Number(cmsIOHANDLER* io,  cmsUInt8Number* n);
cmsBool  _cmsReadUInt16Number(cmsIOHANDLER* io, cmsUInt16Number* n);
cmsBool  _cmsReadUInt32Number(cmsIOHANDLER* io, cmsUInt32Number* n);
cmsBool  _cmsReadFloat32Number(cmsIOHANDLER* io, cmsFloat32Number* n);
cmsBool  _cmsReadUInt64Number(cmsIOHANDLER* io, cmsUInt64Number* n);
cmsBool  _cmsRead15Fixed16Number(cmsIOHANDLER* io, cmsFloat64Number* n);
cmsBool  _cmsReadXYZNumber(cmsIOHANDLER* io, cmsCIEXYZ* XYZ);
cmsBool  _cmsReadUInt16Array(cmsIOHANDLER* io, cmsUInt32Number n,
                                                cmsUInt16Number* Array);
```

Reads several types from the given IOHANDLER.

***Parameters:***

*Io: pointer to the cmsIOHANDLER object.*
*Param: Pointer to an object to receive the result.*

**Returns:**

*TRUE on success, FALSE on error.*

2.0

```
cmsBool  _cmsWriteUInt8Number(cmsIOHANDLER* io, cmsUInt8Number n);
cmsBool  _cmsWriteUInt16Number(cmsIOHANDLER* io, cmsUInt16Number n);
cmsBool  _cmsWriteUInt32Number(cmsIOHANDLER* io, cmsUInt32Number n);
cmsBool  _cmsWriteFloat32Number(cmsIOHANDLER* io, cmsFloat32Number n);
cmsBool  _cmsWriteUInt64Number(cmsIOHANDLER* io, cmsUInt64Number n);
cmsBool  _cmsWrite15Fixed16Number(cmsIOHANDLER* io, cmsFloat64Number n);
cmsBool  _cmsWriteXYZNumber(cmsIOHANDLER* io, const cmsCIEXYZ* XYZ);
cmsBool  _cmsWriteUInt16Array(cmsIOHANDLER* io, cmsUInt32Number n,
                                              const cmsUInt16Number* Array);
```

Writes several types to the given IOHANDLER.

***Parameters:***

*Io: pointer to the cmsIOHANDLER object.*
*Param: Object to write.*

**Returns:**

*TRUE on success, FALSE on error.*

## Type base helper functions.

2.0

```
cmsTagTypeSignature _cmsReadTypeBase(cmsIOHANDLER* io);
```

Reads a cmsTagTypeSignature from the given IOHANDLER.

***Parameters:***

*Io: pointer to an cmsIOHANDLER object.*

**Returns:**

*cmsTagTypeSignature or 0 on error.*

2.0

```
cmsBool  _cmsWriteTypeBase(cmsIOHANDLER* io,  cmsTagTypeSignature sig);
```

Writes s a cmsTagTypeSignature to  the given IOHANDLER.

***Parameters:***

*Io: pointer to an cmsIOHANDLER object.*

*Sig: cmsTagTypeSignature to be written.*

**Returns:**

*TRUE on success, FALSE on error.*

## Alignment & misc. functions.

2.0

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

Skips bytes on the given IOHANDLER until a 32-bit aligned position.

***Parameters:***

*Io: pointer to a cmsIOHANDLER object.*

**Returns:**

*TRUE on success, FALSE on error.*

2.0

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

Writes zeros on the given IOHANDLER until a 32-bit aligned position.

***Parameters:***

*Io: pointer to cmsIOHANDLER object.*

**Returns:**

*TRUE on success, FALSE on error.*

2.0

```
cmsBool  _cmsIOPrintf(cmsIOHANDLER* io, const char* frm, ...);
```

Outputs printf-like strings to the given IOHANDLER. To deal with text streams. 2K at most

***Parameters:***

*Io: pointer to cmsIOHANDLER object.*
*Frm: format string (printf-like)*
*…: optional parameters (printf-like)*

**Returns:**

*TRUE on success, FALSE on error.*

## Fixed point helper functions

2.0

```
cmsFloat64Number    _cms8Fixed8toDouble(cmsUInt16Number fixed8);
```

Converts from 8.8 fixed point to cmsFloat64Number.

***Parameters:***

*fixed8:  8.8 encoded fixed point value.*

**Returns:**

*cmsFloat64Number  holding the value.*

2.0

```
cmsUInt16Number    _cmsDoubleTo8Fixed8(cmsFloat64Number val);
```

Converts from cmsFloat64Number to 8.8 fixed point, rounding properly.

***Parameters:***

*Val: cmsFloat64Number  holding the value.*

**Returns:**

*8.8 encoded fixed point value.*

2.0

```
cmsFloat64Number    _cms15Fixed16toDouble(cmsS15Fixed16Number fix32);
```

Converts from 15.16 (signed) fixed point to cmsFloat64Number.

***Parameters:***

*Fix32: 15.16 (signed) fixed point encoded fixed point value.*

**Returns:**

*cmsFloat64Number  holding the value.*

2.0

```
cmsS15Fixed16Number _cmsDoubleTo15Fixed16(cmsFloat64Number v);
```

Converts from cmsFloat64Number to 15.16 fixed point, rounding properly.

***Parameters:***

*V: cmsFloat64Number holding the value.*

**Returns:**

*15.16 (signed) fixed point encoded fixed point value.*

## Date/time helper functions

2.0

```
void    _cmsEncodeDateTimeNumber(cmsDateTimeNumber *Dest,
                                 const struct tm *Source);
```

Decodes from the standard "C" struct tm to ICC date and time format.

***Parameters:***

*Dest: a pointer to a cmsDateTimeNumber structure.*
*Source: a pointer to a struct tm structure.*

**Returns:**

**None**

2.0

```
void    _cmsDecodeDateTimeNumber(const cmsDateTimeNumber *Source,
                                 struct tm *Dest);
```

Decodes from ICC date and time format to the standard "C" struct tm.

***Parameters:***

*Source: a pointer to a cmsDateTimeNumber structure.*
*Dest: a pointer to a struct tm structure.*

**Returns:**

**None**

## Error Logging API

For debugging purposes, it may be handy to know what is making a function to fail. This function add traces to let developer what is going on.

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

*Table 1*

2.0

```
void  cmsSignalError(cmsContext ContextID,  cmsUInt32Number ErrorCode,
                                            const char *ErrorText, ... );
```

***Parameters:***

*ContextID:  Pointer to a user-defined context cargo.*

*ErrorCode: Error family, as stated in Table 1*

*ErrorText:  Error description, printf-like*

*…:  additional printf-like parameters.*

**Returns:**

**None**

Warning: As this function uses a variable number of parameters, implementing such function on Windows DLL, which uses the PASCAL calling convention, is undefined for some compilers. Then, for example, Borland C++ 5.5 does not support this function. If you are using such compiler and want to place your plug-ins in a separate DLL, you cannot use this function at all. If you need this functionality in your plug-in, consider to use any other compiler instead, or link your plug-ins within the *Little CMS* DLL.

## Memory management API

Those are the memory management primitives as used by the core engine. It uses the memory management plug-in if defined.

2.0

```
void* _cmsMalloc(cmsContext ContextID, cmsUInt32Number size);
```

Allocate size bytes of uninitialized memory.

***Parameters:***

*ContextID: Pointer to a user-defined context cargo.*

*Size: amount of memory to allocate in bytes*

***Returns:***

*Pointer to newly allocated block, or NULL on error.*

2.0

```
void  _cmsFree(cmsContext ContextID, void* Ptr);
```

Cause the space pointed to by Ptr to be deallocated; that is, made available for further allocation. If ptr is a null pointer, no action will occur.

***Parameters:***

*ContextID: Pointer to a user-defined context cargo.*

*Ptr: pointer to memory block.*

***Returns:***

**None**

2.0

```
void* _cmsMallocZero(cmsContext ContextID, cmsUInt32Number size);
```

Allocate size bytes of memory. Initialize it to zero.

***Parameters:***

*ContextID: Pointer to a user-defined context cargo.*

*Size: amount of memory to allocate in bytes*

***Returns:***

*Pointer to newly allocated block, or NULL on error.*

2.0

```
void* _cmsCalloc(cmsContext ContextID, cmsUInt32Number num,
                 cmsUInt32Number size);
```

Allocate space for an array of num elements each of whose size in bytes is size. The space shall be initialized to all bits 0.

***Parameters:***

*ContextID: Pointer to a user-defined context cargo.*

*Num: number of array elements*

*Size: Array element size in bytes*

***Returns:***

*Pointer to newly allocated block, or NULL on error.*

2.0

```
void* _cmsRealloc(cmsContext ContextID, void* Ptr, cmsUInt32Number NewSize);
```

The size of the memory block pointed to by the P*tr* parameter is changed to the New*Size* bytes, expanding or reducing the amount of memory available in the block.

***Parameters:***

*ContextID: Pointer to a user-defined context cargo.*

*Ptr: pointer to memory block.*

*NewSize: number of bytes.*

***Returns:***

*Pointer to newly allocated block, or NULL on error.*

2.0

```
void* _cmsDupMem(cmsContext ContextID, const void* Org, cmsUInt32Number size);
```

Duplicates the contents of memory at “Org” into a new block

***Parameters:***

*ContextID: Pointer to a user-defined context cargo.*
*Org: pointer to source memory block.*
*Size: number of bytes to duplicate.*

***Returns:***

*Pointer to newly allocated copy, or NULL on error.*

## Vector & Matrix API

Those are low-level primitives to opearate with 3-component vectors and 3x3 matrices.

### VEC3 vectors

Vectors are defined as using 64-bit floating point numbers (cmsFloat64Numbers)

```
typedef struct {
    cmsFloat64Number n[3];

    } cmsVEC3;
```

2.0

```
void  _cmsVEC3init(cmsVEC3* r,
                   cmsFloat64Number x,
                   cmsFloat64Number y,
                   cmsFloat64Number z);
```

Populates a vector.

***Parameters:***

r: a pointer to a cmsVEC3 object to receive the result
*x, y, z: components of the vector.*
*b: A pointer to second cmsVEC3 object.*

***Returns:***

**None**

2.0

```
void   _cmsVEC3minus(cmsVEC3* r, const cmsVEC3* a, const cmsVEC3* b);
```

*Vector subtraction.*

***Parameters:***

r: a pointer to a cmsVEC3 object to receive the result
*a: A pointer to first cmsVEC3 object.*
*b: A pointer to second cmsVEC3 object.*

***Returns:***

**None**

2.0

```
void    _cmsVEC3cross(cmsVEC3* r, const cmsVEC3* u, const cmsVEC3* v);
```

Vector cross product.

***Parameters:***

r: *a pointer to a cmsVEC3 object to receive the result.*
*u: A pointer to first cmsVEC3 object.*
*v: A pointer to second cmsVEC3 object.*

***Returns:***
**None**

2.0

```
cmsFloat64Number   _cmsVEC3dot(const cmsVEC3* u, const cmsVEC3* v);
```

Vector dot product

***Parameters:***

*u: A pointer to first cmsVEC3 object.*
*v: A pointer to second cmsVEC3 object.*

***Returns:***
*Dot product* $u \cdot v$

2.0

```
cmsFloat64Number   _cmsVEC3length(const cmsVEC3* a);
```

Euclidean length of 3D vector

***Parameters:***

*a: A pointer to first cmsVEC3 object.*
*b: A pointer to second cmsVEC3 object.*

***Returns:***
*Euclidean length* $\sqrt{x^2 + y^2 + z^2}$

2.0

```
cmsFloat64Number  _cmsVEC3distance(const cmsVEC3* a, const cmsVEC3* b);
```

Returns euclidean distance between two 3D points.

***Parameters:***

*a: A pointer to first cmsVEC3 object.*
*b: A pointer to second cmsVEC3 object.*

***Returns:***

*Euclidean distance* $\sqrt{a^2 + b^2}$

### MAT3 matrices

3x3 Matrices are formed by 3 VEC3 vectors. The Plugin API provides several low-level primitives for 3x3 Matrix math.

```
typedef struct {
    cmsVEC3 v[3];

    } cmsMAT3;
```

2.0

```
void    _cmsMAT3identity(cmsMAT3* a);
```

Fills “a” with identity matrix

**Parameters:**

*a: A pointer to a cmsMAT3 object.*

***Returns:***

**None**

2.0

```
cmsBool  _cmsMAT3isIdentity(const cmsMAT3* a);
```

Return true if “a” is close enough to be interpreted as identity. Else return false

**Parameters:**

*a: A pointer to a cmsMAT3 object.*

***Returns:***

*TRUE on identity, FALSE on non-identity.*

2.0

```
void  _cmsMAT3per(cmsMAT3* r, const cmsMAT3* a, const cmsMAT3* b);
```

Multiply two matrices.

**Parameters:**

r: *a pointer to a cmsMAT3 object to receive the result.*
*a: A pointer to first cmsMAT3 object.*
*b: A pointer to second cmsMAT3 object.*

***Returns:***
**None**

2.0

```
cmsBool  _cmsMAT3inverse(const cmsMAT3* a, cmsMAT3* b);
```

Inverse of a matrix $b = a^{-1}$. Returns false if singular matrix

**Parameters:**

*a: A pointer to the cmsMAT3 to be inverted.*
*b: A pointer to a cmsMAT3 object to store the result.*

***Returns:***
*TRUE on success, FALSE on error.*

2.0

```
cmsBool  _cmsMAT3solve(cmsVEC3* x,  cmsMAT3* a, cmsVEC3* b);
```

Solves a system in the form $Ax = b$. Returns FALSE if singular matrix

**Parameters:**

x: *a pointer to a cmsVEC3 object to receive the result.*
*a: A pointer to first cmsVEC3 object.*
*b: A pointer to second cmsVEC3 object.*

***Returns:***
*TRUE on success, FALSE on error.*

2.0

```
void  _cmsMAT3eval(cmsVEC3* r, const cmsMAT3* a, const cmsVEC3* v);
```

Evaluates a matrix and stores the result in “r”.

**Parameters:**

r: *a pointer to a cmsVEC3 object to receive the result.*
*a: A pointer to the cmsMAT3 object containing the transformation matrix.*
*v: a pointer to a cmsVEC3 object to be evaluated.*

***Returns:***
**None**

## Conclusion

*Little CMS* 2.x plug-in system is a convenient way to enhance the CMM functionality in many aspects, but there are chances you don’t need to use plug-in to add new functionality at all. If you can stay in the standard *Little CMS* API, I would recommend avoiding writing plug-ins. Avoiding plug-ins is convenient because backwards compatibility, clarity and maintainability. The normal API has been designed with easy-to-use goals; on the other hand, on plug-in API functionality is the most desirable attribute.

If you decide to write extensions, please note there are many ways to do that. One example would be to write functions using the plug-in API, but without exporting any plug-in. This is ok if you need to use low-level functions that are not present in the normal API.