http://www.littlecms.com

Copyright © 2017 Marti Maria Saguer, all rights reserved.

U w R

v u R

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 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 library, keeping it in the open source side.

There are 11 types of plug-ins currently supported:

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

This manual details how to write any of those 11 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.

R

R R R wRu wRw wR

Plug-ins are “plugged” to when creating the contexts.

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

Parameters:

Returns:

The user can install additional plug-ins to a yet existing context by using this function.

This one installs plug-ins in the default context. Only useful when using Little CMS as a static library. This function is deprecated, and the recommended way is by using contexts.

This function returns default context 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.

This function is same as anterior, but operating in the giving context.

R

Rsu s wR

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 ()

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:

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

R

Ws wR

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 , so it will be driven by PCS and will return one channel giving the relative ink consumption by color. Writing a plug-in in 2 will allow and to deal with you new data exactly as any other standard tag.

To do so, you have to fill a 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 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 by calling:

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

And that is. Now lcms2 understand about “inkc” tags and can read and write them. R

wR R Rs wRw R R

As a plug-in writer, you may want to encapsulate several plug-in in the same package. This is easy with the way 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 . Last plug-in in the chain must have a in the “Next” field.

R su R R RsRw s s wRV R v R RR

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 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 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.

w w w R

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.

R8(w(8(8(sh)14(o)-5(u)be)4(2()-143(g77(p)13d5(n)3())] TETBT 1.491398.832 5119.4 RG(.)] TETBT140 051398.832 5119.8 Tm( )] TET g3

The expected version; 2040 in current release. 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).

It defines behaviour of plug-in. There are 10 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'
cmsPluginTransformSig	0x7A666D48	'xfmH'

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

w R s s w w R R

By using this plug-in type, a programmer can override memory management done by . 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.

Setting optional function pointers to NULL forces to use functions for all operation. If you provide all set of functions, 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 that identifies the calling environment. It may be zero on certain special cases. This is provided by the user when calling 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.

:

```
#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 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. R

w s R R

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

internal operation is based on . 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 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 will be used instead. Only one factory can be set at time. Further calls to with this type will replace the behavior of previous plug-in. Interpolators have no states and no memory, and therefore cannot hold private data.

There is a limitation on the maximum input dimensions:

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

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.

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)

R R w s R

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.

s R R w s R

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.

R

w s Rs s ww R

When an interpolator is called, 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.

```
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 };
```

R

s s w uRu wR R

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 , as well as "curves" pipeline stage. Each call to 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.

:

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

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

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

Note the 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 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.

:

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

:

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

:

```
#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};
```

R

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. R

s w R R

can handle a lot of formats of image data. For describing such formats, 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.

:

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

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.

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

```
cmsUInt8Number* my_Unroll8(struct _cmstransform_struct* 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.R

R

s R R

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.

Each Plug-in implements a single tag:

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

```
#define inkc_constant 0x696E6B43

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

s R wRs v wR R

Tag type plug-in complements tag plug-in by adding new types. Types are responsible of the structure returned when 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.

:

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. will set this member to proper value when invoking the plug-in.

```
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 (!_cmsReadUInt32Number(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
                                 }};
```

R

w R R

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.

Each plug-in defines a single intent number.

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.

R

R u R R w R R

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

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

Parameters:

```
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.

R

s wRR

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.

U ws R w R s wR wR

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

Parameters:

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

```
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 SigNegateType ((cmsStageSignature)0x6E202020)

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

R

s wR IR

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 “ ” addendum to ICC spec 4.2.

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

:

Plug-in structure:

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

```
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},
                          { SigNegateType
                            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.

s R IR

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

To implement transforms, 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 specialized callback that would be responsible of evaluating the optimized version of LUT. This callback has this form:

cmsOPTeval16Fn:

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.

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:

```
  _
```

Set 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:

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.

:

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.

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

    //  Only curves in this LUT?
    for (mpe = cmsPipelineGetPtrToFirstStage(*Lut);
         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.

R

R s R R R RR

Using this plug-in, a programmer can get replace completely the color transform logic. The plug-in can supply a callback to be used when user calls () or ()

:

```
0x7A666D48 'xfmH'
```

It is also posible to allocate and maintain an amount of data, used only by the transform callback supplied by the plug-in. The plug-in writer, then, have to provide an additional callback to free any resource associated with this data. This callback type is used in other plug-ins as well.

Such data is optional. Use NULL when the plug-in callback does not need private data.

The function is invoked for each new color transform cretead after plug-in registration. The factory can accept to provide transform entry points if the actual transform parameters meets its requeriments. The factory can also change the actual transform parameters after it has accepted the task.

Previous to 2.8, the generic transform function was slightly different. lcms2.8 regognizes te veersion stamp for 2.4 to 2.7 and provides a conversion stage. The old generic transform is now deprecated.

It is equivalent to cmsDoTransformStride(), which is deprecated as well.

Parameters:

Example

```
// The factory
cmsBool Dispatch(_cmsTransformFn* xformPtr, void** UserData,
                 _cmsFreeUserDataFn* FreePrivateDataFn,
                 cmsPipeline** Lut,
                 cmsUInt32Number* InputFormat,
                 cmsUInt32Number* OutputFormat,
                 cmsUInt32Number* dwFlags)
{
     if (*InputFormat == MY_TYPE_RGB_8) {

          xformPtr = my_fn;
          return TRUE;
     }

     Return FALSE;
}

// The plug-in structure
cmsPluginTransform Plugin = {

     { cmsPluginMagicNumber, 2040, cmsPluginTransformSig, NULL},

     Dispatch
     };
```

w R R R

Using this plug-in, a programmer can control how read and write operations on profiles are locked and therefore control concurrency. By default, pthreads library is used on all implementations but Windows. On Windows, Critical sections are used.

:

R

User has to provide following functions:

R

Example (Windows)

```
static
void* MyMtxCreate(cmsContext id)
{
   return (void*) CreateMutex( NULL, FALSE, NULL);
}

static
void MyMtxDestroy(cmsContext id, void* mtx)
{
    CloseHandle((HANDLE) mtx);
}

static
cmsBool MyMtxLock(cmsContext id, void* mtx)
{
    WaitForSingleObject((HANDLE) mtx, INFINITE);
    return TRUE;
}

static
void MyMtxUnlock(cmsContext id, void* mtx)
{
    ReleaseMutex((HANDLE) mtx);
}


static cmsPluginMutex MutexPluginSample = {

     { cmsPluginMagicNumber, 2060, cmsPluginMutexSig, NULL},

     MyMtxCreate,  MyMtxDestroy,  MyMtxLock,  MyMtxUnlock
};
```

R

R u R

Returns a pointer to the user data associated with current color transform

Parameters:

R S R

R s v w R

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.

R

WS V wR u R

Reads several types from the given IOHANDLER.

Parameters:

Writes several types to the given IOHANDLER.

Parameters:

R

wR s wR w wR u R

Reads a cmsTagTypeSignature from the given IOHANDLER.

Parameters:

S w R R uR u R

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

Parameters:

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

Parameters:

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

Parameters:

R

wR R w wR u R

Converts from 8.8 fixed point to cmsFloat64Number.

Parameters:

Converts from cmsFloat64Number to 8.8 fixed point, rounding properly.

Parameters:

Converts from 15.16 (signed) fixed point to cmsFloat64Number.

Parameters:

Converts from cmsFloat64Number to 15.16 fixed point, rounding properly.

Parameters:

R

Vs w wR w wR u R

W R ES R

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

Parameters:

<span style="color:red">Warning</span>: 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 DLL. R

w R s s w w IS R

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

Allocate size bytes of uninitialized memory.

Parameters:

Returns:

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:

Returns:

Allocate size bytes of memory. Initialize it to zero.

Parameters:

Returns:

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:

Returns:

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

Parameters:

Returns:

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

Parameters:

Returns:

R

wu RR s ES R

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

WU R wu R

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

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

    } cmsVEC3;
```

Populates a vector.

Parameters:

r: a pointer to a cmsVEC3 object to receive the result

Returns:

Parameters:

r: a pointer to a cmsVEC3 object to receive the result

Returns:

Vector cross product.

Parameters:

r:

Returns:

Vector dot product

Parameters:

Returns:

$u \cdot v$

Euclidean length of 3D vector

Parameters:

Returns:

$\sqrt{x^2 + y^2 + z^2}$

Returns euclidean distance between two 3D points.

Parameters:

Returns:

$$\sqrt{a^2 + b^2}$$

R

S R s uwR

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;
```

Multiply two matrices.

r:

Returns:

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

Returns:

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

x:

Returns:

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

r:

Returns:

U u R

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 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.