Thorlabs Scientific Color Processing C API

Rev. D 2019-03-19 ITN003542-D01

Contents

0.1 Thorlabs Scientific Color Processing Component Suite

0.1.1 Introduction

The target audience for this document is the experienced software engineer with a background in color image processing.

The color processing suite consists of 3 pritimive components that are designed to be used in tandem to enable an application to convert a Bayer pattern monochrome image into a color image with 3 color channels.

The components are:

- a look-up table (LUT) module
- a demosaic module
- a color processing module

Look Up Table (LUT) Module

The LUT module allows the user to create an instance object which contains a LUT data array of user specified values.

The LUT concept is useful for modeling any generic mathematical function which can then be applied to transform an input data set into the desired output at very high speed (very low computational cost).

The LUT instance internally implements the memory management for the LUT data and exposes functions that allow the access and manipulation of that data.

Furthermore, it provides a function which applies the LUT transform to the input data in a single operation. This function has been optimized using Intel's AVX2 vector instructions to accelerate the computation for machines which support that instruction set. It also includes a legacy scalar processing path for older generation hardware which lacks support for the new instructions.

The user can create any number LUT instances each containing a unique transformation function.

Demosaic Module

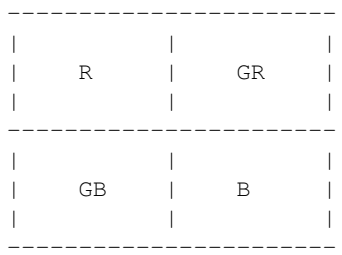
The demosaic module offers a function which accepts an input buffer containing monochrome pixel data (presumably generated by an imaging device capable of producing Bayer pattern image data) and "explodes" that data into 3 color channels of image data.

The main transform function has been optimized using Intel's AVX2 vector instructions to accelerate the computation for machines which support that instruction set. It also includes a legacy scalar processing path for older generation hardware which lacks support for the new instructions.

Currently, the demosaic module supports color sensors which implement a Bayer pattern color pixel array. The implementation can be extended to support other color pixel arrangements.

See http://en.wikipedia.org/wiki/Bayer_filter for more information on Bayer pattern color sensors.

The Bayer pattern primitive that the demosaic module currently supports is:



where:

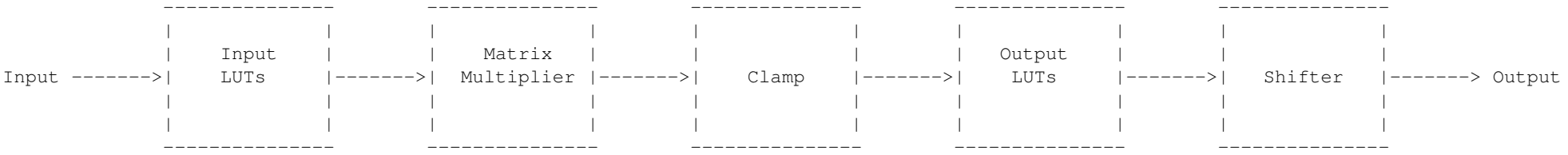
- R = a red pixel
- GR = a green pixel next to a red pixel
- B = a blue pixel
- GB = a green pixel next to a blue pixel

Color Processing Module

The color processing module allows the user to create a color processing object that enables applications to process an unprocessed debayered image (the output of the demosaic module) to yield a color accurate output image.

The instance implements a simple matrix based color processing pipeline with additional pre and post processing units to further enhance its ability to support different use cases.

The following is a diagram of the pipeline structure:



Each block in the diagram above is a functional unit in the processing pipeline. The arrows indicate the data flow direction.

- Input LUT unit
  - Allows the user to apply a custom mathematical function to the input data prior to the matrix multiplier stage.
  - A separate LUT function can be specified for each color channel therefore there are 3 LUTs in this pipe stage.
  - The user can choose to selectively enable any combination of the 3 LUTS or to disable all of them.
  - It is the user's responsibility to ensure that the LUT transform keeps the data within 16 bits of precision. If the LUT output exceeds 16 bits, then the lower 16 bits of the result are used.
- Matrix multipler unit
  - This is the heart of the color processing pipeline which implements the color correction computation.
  - The user can concatenate any number of 3x3 matrices which yields a single resultant matrix that is applied to the data.
- Clamp unit
  - This unit is used to restict the set of output values from the matrix multiplier stage to a predetermined range.
  - This is necessary because the matrix multiplier uses floating point arithmetic which can result in out of range output values.

  - The user can control the minimum and maximum values of the range on a per color channel basis.

- Output LUT unit
  - Allows the user to apply a custom mathematical function to the data after it has been transformed by the matrix multiplier.
  - A separate LUT function can be specified for each color channel therefore there are 3 LUTs in this pipe stage.
  - The user can choose to selectively enable any combination of the 3 LUTS or to disable all of them.
  - It is the user's responsibility to ensure that the LUT transform keeps the data within 16 bits of precision. If the LUT output exceeds 16 bits, then the lower 16 bits of the result are used.

- Shifter unit
  - Allows the user to shift the data prior to writing it to the output buffer.
  - A separate shift distance can be specified per color channel.
    - A positive integral shift distance denotes a left shift.
    - A 0 (zero) shift distance leaves the data unchanged.
    - A negative integral shift distance denotes a right shift.
  - It is the user's responsibility to ensure that the shifter unit does not change the data to an out-of-range value.

The instance contains a vector optimized pipeline implementation using Intel's AVX2 instructions to accelerate the computation for machines that support that instruction set.

It also includes a legacy scalar pipeline implementation for older generation hardware which lack support for the new instructions.

The application can create any number of color processing instances which can be uniquely configured to process the input image data differently.

Notes

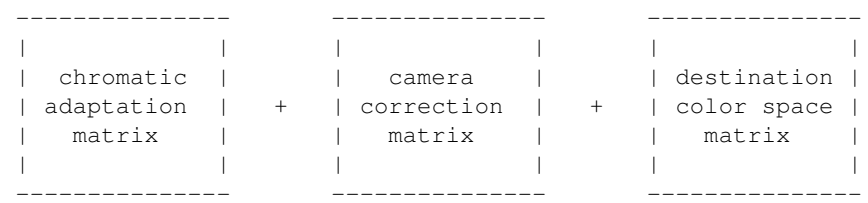
- The data path is fixed at 16 bits and this represents the maximum bit depth of the output data.
- None of the modules in this component suite are thread safe. If the user intends to use these modules in a concurrent execution environment, it is their responsibility to ensure thread safety.
- Each module contains both scalar and vector implementations. The user does not need to specifically choose between the two - that is done automatically by the respective module based on a run-time interrogation of the CPU's capabilities.

Example

The following example is written in C-style C++03 code to demonstrate a common use case for the color processing components.

It uses color matrix data from a real color camera and configures the color processing module to output sRGB color images that can be displayed on an sRGB color monitor.

The sequential order of matrix concatenation is significant - this example uses the following generally accepted matrix concatenation order:



The example code uses a matrix that represents a merged camera correction and destination color space (sRGB) matrix.

The output LUTs are used to apply the non-linear part of the sRGB destination color space transformation.

It is assumed the user is writing code for a Microsoft Windows based machine; however, the example can be easily ported to any other platform (e.g. Linux).

```
#include <Windows.h>
#include <cstdio>
#include <cmath>
#include "tl_color_demosaic.h"
#include "tl_color_processing.h"
#include "tl_color_error.h"
#include "tl_color_enum.h"

double sRGBCompand(double colorPixelIntensity)
{
    const double expFactor = 1 / 2.4;
    return ((colorPixelIntensity <= 0.0031308) ? colorPixelIntensity * 12.92 : ((1.055 * pow(colorPixelIntensity, expFactor)) - 0.055));
}

void sRGB_companding_LUT(int bit_depth, int* lut)
{
    int max_pixel_value = (1 << bit_depth) - 1;
```

```
    int LUT_size = max_pixel_value + 1;
    const double dMaxValue = static_cast <double> (max_pixel_value);
    for (int i = 0; i < LUT_size; ++i)
        lut[i] = static_cast <unsigned short> (sRGBCompand(static_cast <double> (i) / dMaxValue) * dMaxValue);
}

int create_color_frame(unsigned short* input_monochrome_frame, int mono_image_width, int mono_image_height, unsigned short* output_color_frame)
{
    // Load the demosaic module.
    HMODULE demosaic_module_handle = ::LoadLibrary("thorlabs_tsi_demosaic.dll");
    if (!demosaic_module_handle)
    {
        printf("Failed to open the demosaic library!\n");
        return 1;
    }

    // Map handles to the demosaic module exported functions.
    TL_DEMOSAIC_MODULE_INITIALIZE tl_demosaic_module_initialize =
        reinterpret_cast <TL_DEMOSAIC_MODULE_INITIALIZE> (::GetProcAddress(demosaic_module_handle, "tl_demosaic_module_initialize"));
    if (!tl_demosaic_module_initialize)
    {
        printf("Failed to map demosaic initialize function!\n");
        ::FreeLibrary(demosaic_module_handle);
        return 1;
    }

    TL_DEMOSAIC_TRANSFORM_16_TO_48 tl_demosaic_transform_16_to_48 =
        reinterpret_cast <TL_DEMOSAIC_TRANSFORM_16_TO_48> (::GetProcAddress(demosaic_module_handle, "tl_demosaic_transform_16_to_48"));
    if (!tl_demosaic_transform_16_to_48)
    {
        printf("Failed to map demosaic function!\n");
        ::FreeLibrary(demosaic_module_handle);
        return 1;
    }

    TL_DEMOSAIC_MODULE_TERMINATE tl_demosaic_module_terminate = reinterpret_cast <TL_DEMOSAIC_MODULE_TERMINATE> (
        ::GetProcAddress(demosaic_module_handle, "tl_demosaic_module_terminate"));
    if (!tl_demosaic_module_terminate)
    {
        printf("Failed to map destroy function!\n");
        ::FreeLibrary(demosaic_module_handle);
        return 1;
    }

    HMODULE cc_module_handle = ::LoadLibrary("thorlabs_tsi_color_processing.dll");
    if (!cc_module_handle)
    {
```

```
    printf("Failed to load the color processing module!\n");
    ::FreeLibrary(demosaic_module_handle);
    return 1;
}

TL_COLOR_PROCESSING_MODULE_INITIALIZE tl_color_processing_module_initialize =
    reinterpret_cast <TL_COLOR_PROCESSING_MODULE_INITIALIZE> (::GetProcAddress(cc_module_handle, "tl_color_processing_module_initialize"));
if (!tl_color_processing_module_initialize)
{
    printf("Failed to map the color processing module initialize function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_CREATE_COLOR_PROCESSOR tl_color_create_color_processor =
    reinterpret_cast <TL_COLOR_CREATE_COLOR_PROCESSOR> (::GetProcAddress(cc_module_handle, "tl_color_create_color_processor"));
if (!tl_color_create_color_processor)
{
    printf("Failed to map the create color processor function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_GET_BLUE_INPUT_LUT tl_color_get_blue_input_LUT =
    reinterpret_cast <TL_COLOR_GET_BLUE_INPUT_LUT> (::GetProcAddress(cc_module_handle, "tl_color_get_blue_input_LUT"));
if (!tl_color_get_blue_input_LUT)
{
    printf("Failed to map the get blue input LUT function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_GET_GREEN_INPUT_LUT tl_color_get_green_input_LUT =
    reinterpret_cast <TL_COLOR_GET_GREEN_INPUT_LUT> (::GetProcAddress(cc_module_handle, "tl_color_get_green_input_LUT"));
if (!tl_color_get_green_input_LUT)
{
    printf("Failed to map the get green input LUT function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_GET_RED_INPUT_LUT tl_color_get_red_input_LUT =
    reinterpret_cast <TL_COLOR_GET_RED_INPUT_LUT> (::GetProcAddress(cc_module_handle, "tl_color_get_red_input_LUT"));
```

```
if (!tl_color_get_red_input_LUT)
{
    printf("Failed to map the get red input LUT function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_ENABLE_INPUT_LUTS tl_color_enable_input_LUTs =
    reinterpret_cast <TL_COLOR_ENABLE_INPUT_LUTS> (::GetProcAddress(cc_module_handle, "tl_color_enable_input_LUTs"));
if (!tl_color_enable_input_LUTs)
{
    printf("Failed to map the enable input LUTs function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_APPEND_MATRIX tl_color_append_matrix = reinterpret_cast <TL_COLOR_APPEND_MATRIX> (::GetProcAddress(cc_module_handle, "tl_color_append_matrix"));
if (!tl_color_append_matrix)
{
    printf("Failed to map the append matrix function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_CLEAR_MATRIX tl_color_clear_matrix = reinterpret_cast <TL_COLOR_CLEAR_MATRIX> (::GetProcAddress(cc_module_handle, "tl_color_clear_matrix"));
if (!tl_color_clear_matrix)
{
    printf("Failed to map the clear matrix function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_GET_BLUE_OUTPUT_LUT tl_color_get_blue_output_LUT =
    reinterpret_cast <TL_COLOR_GET_BLUE_OUTPUT_LUT> (::GetProcAddress(cc_module_handle, "tl_color_get_blue_output_LUT"));
if (!tl_color_get_blue_output_LUT)
{
    printf("Failed to map the get blue output LUT function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_GET_GREEN_OUTPUT_LUT tl_color_get_green_output_LUT =
```

```
    reinterpret_cast <TL_COLOR_GET_GREEN_OUTPUT_LUT> (::GetProcAddress(cc_module_handle, "tl_color_get_green_output_LUT"));
if (!tl_color_get_green_output_LUT)
{
    printf("Failed to map the get green output LUT function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_GET_RED_OUTPUT_LUT tl_color_get_red_output_LUT =
    reinterpret_cast <TL_COLOR_GET_RED_OUTPUT_LUT> (::GetProcAddress(cc_module_handle, "tl_color_get_red_output_LUT"));
if (!tl_color_get_red_output_LUT)
{
    printf("Failed to map the get red output LUT function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_ENABLE_OUTPUT_LUTS tl_color_enable_output_LUTs =
    reinterpret_cast <TL_COLOR_ENABLE_OUTPUT_LUTS> (::GetProcAddress(cc_module_handle, "tl_color_enable_output_LUTs"));
if (!tl_color_enable_output_LUTs)
{
    printf("Failed to map the enable output LUTs function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_TRANSFORM_48_TO_48 tl_color_transform_48_to_48 =
    reinterpret_cast <TL_COLOR_TRANSFORM_48_TO_48> (::GetProcAddress(cc_module_handle, "tl_color_transform_48_to_48"));
if (!tl_color_transform_48_to_48)
{
    printf("Failed to map the transform 48 to 48 function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_TRANSFORM_48_TO_24 tl_color_transform_48_to_24 =
    reinterpret_cast <TL_COLOR_TRANSFORM_48_TO_24> (::GetProcAddress(cc_module_handle, "tl_color_transform_48_to_24"));
if (!tl_color_transform_48_to_24)
{
    printf("Failed to map the transform 24 function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
```

```
}

TL_COLOR_TRANSFORM_48_TO_32 tl_color_transform_48_to_32 =
    reinterpret_cast <TL_COLOR_TRANSFORM_48_TO_32> (::GetProcAddress(cc_module_handle, "tl_color_transform_48_to_32"));
if (!tl_color_transform_48_to_32)
{
    printf("Failed to map the transform 32 function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_DESTROY_COLOR_PROCESSOR tl_color_destroy_color_processor =
    reinterpret_cast <TL_COLOR_DESTROY_COLOR_PROCESSOR> (::GetProcAddress(cc_module_handle, "tl_color_destroy_color_processor"));
if (!tl_color_destroy_color_processor)
{
    printf("Failed to map the color processor destroy function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

TL_COLOR_PROCESSING_MODULE_TERMINATE tl_color_processing_module_terminate =
    reinterpret_cast <TL_COLOR_PROCESSING_MODULE_TERMINATE> (::GetProcAddress(cc_module_handle, "tl_color_processing_module_terminate"));
if (!tl_color_processing_module_terminate)
{
    printf("Failed to map the color processing module terminate function!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

// Initialize the demosaic module.
if (tl_demosaic_module_initialize() != TL_COLOR_NO_ERROR)
{
    printf("Failed to initialize the demosaic library!\n");
    ::FreeLibrary(demosaic_module_handle);
    ::FreeLibrary(cc_module_handle);
    return 1;
}

// Initialize the color processing module.
if (tl_color_processing_module_initialize() != TL_COLOR_NO_ERROR)
{
    printf("Failed to initialize the color processing module!\n");
    tl_demosaic_module_terminate();
    ::FreeLibrary(cc_module_handle);
```

```
    ::FreeLibrary(demosaic_module_handle);
    return 1;
}

// Allocate a temporary buffer (3x larger than the monochrome buffer) to hold the demosaic (only) data.
unsigned short* demosaic_color_buffer = new unsigned short[mono_image_width * mono_image_height * 3];

// Demosaic the monochrome image data.
if (tl_demosaic_transform_16_to_48(mono_image_width
    , mono_image_height
    , 0
    , 0
    , TL_COLOR_FILTER_ARRAY_PHASE_BAYER_BLUE
    , TL_COLOR_FORMAT_BGR_PLANAR
    , TL_COLOR_FILTER_TYPE_BAYER
    , 14
    , input_monochrome_frame
    , demosaic_color_buffer) != TL_COLOR_NO_ERROR)
{
    printf("Failed to demosaic the monochrome image!\n");
    delete[](demosaic_color_buffer);
    tl_demosaic_module_terminate();
    tl_color_processing_module_terminate();
    ::FreeLibrary(cc_module_handle);
    ::FreeLibrary(demosaic_module_handle);
    return 1;
}

// Create a color processor instance.
void* color_processor_inst = tl_color_create_color_processor(14, 14); // 14-bit image data
if (!color_processor_inst)
{
    printf("Failed to create a color processor instance!\n");
    delete[](demosaic_color_buffer);
    tl_demosaic_module_terminate();
    tl_color_processing_module_terminate();
    ::FreeLibrary(cc_module_handle);
    ::FreeLibrary(demosaic_module_handle);
}

// configure sRGB output color space
// The matrix values used here are hard coded for illustrative purposes.
// The user should obtain the camera specific matrix values directly from the camera itself via API calls in the camera SDK.
// Please consult the camera SDK documentation for details.
float white_balance_matrix [9] = { 1.0f, 0.0f, 0.0f, 0.0f, 1.37f, 0.0f, 0.0f, 0.0f, 2.79f };
float color_correction_matrix [9] = { 1.25477f, -0.15359f, -0.10118f, -0.07011f, 1.13723f, -0.06713f, 0.0f, -0.26641f, 1.26641f };
```

```
tl_color_append_matrix (color_processor_inst, white_balance_matrix);
tl_color_append_matrix (color_processor_inst, color_correction_matrix);

// Use the output LUTs to configure the sRGB nonlinear (companding) function.
sRGB_companding_LUT(14, tl_color_get_blue_output_LUT(color_processor_inst));
sRGB_companding_LUT(14, tl_color_get_green_output_LUT(color_processor_inst));
sRGB_companding_LUT(14, tl_color_get_red_output_LUT(color_processor_inst));

// Enable the sRGB nonlinear LUTs.
tl_color_enable_output_LUTs (color_processor_inst, 1, 1, 1);

// Color process the demosaic color frame.
if (tl_color_transform_48_to_48(color_processor_inst
    , demosaic_color_buffer // input buffer
    , TL_COLOR_FORMAT_BGR_PLANAR          // input buffer format
    , 0                     // blue minimum clamp value
    , (1 << 14) - 1         // blue maximum clamp value (14 bit pixel data)
    , 0                     // green minimum clamp value
    , (1 << 14) - 1         // green maximum clamp value
    , 0                     // red minimum clamp value
    , (1 << 14) - 1         // red maximum clamp value
    , 0                     // blue shift distance
    , 0                     // green shift distance
    , 0                     // red shift distance
    , output_color_frame    // output buffer
    , TL_COLOR_FORMAT_BGR_PIXEL       // output buffer format
    , mono_image_width * mono_image_height) != TL_COLOR_NO_ERROR)
{
    printf("Failed to color process the demosaic color frame!\n");
}

// Destroy the color processor instance.
if (tl_color_destroy_color_processor(color_processor_inst) != TL_COLOR_NO_ERROR)
{
    printf("Failed to destroy the color processor instance!\n");
}

// Clean up.
delete[](demosaic_color_buffer);

// Terminate the color processing module
if (tl_color_processing_module_terminate() != TL_COLOR_NO_ERROR)
{
    printf("Failed to terminate the color processing module!\n");
}

// Terminate the demosaic module.
```

```
    if (tl_demosaic_module_terminate() != TL_COLOR_NO_ERROR)
    {
        printf("Failed to terminate the demosaic library!\n");
    }

    // Unload the color processing module.
    ::FreeLibrary(cc_module_handle);

    // Unload the demosiac module.
    ::FreeLibrary(demosaic_module_handle);

    return 0;
}
```

0.2 File Index

0.2.1 File List

Here is a list of all documented files with brief descriptions:

0.3 File Documentation

0.3.1 tl_color_demosaic.h File Reference

```
#include "tl_color_enum.h"
```

Typedefs

- typedef int(∗ TL_DEMOSAIC_MODULE_INITIALIZE) (void)
- typedef int(∗ TL_DEMOSAIC_TRANSFORM_16_TO_48) (int width, int height, int x_origin, int y_origin, enum TL_COLOR_FILTER_ARRAY_PHASE color_phase, enum TL_COLOR_FORMAT output_color_format, enum TL_COLOR_FILTER_TYPE color_filter_type, int bit_depth, unsigned short ∗input_buffer, unsigned short ∗output_buffer)
- typedef int(∗ TL_DEMOSAIC_MODULE_TERMINATE) (void)

0.3.1.1 Typedef Documentation

0.3.1.1.1 TL_DEMOSAIC_MODULE_INITIALIZE

```
typedef int(* TL_DEMOSAIC_MODULE_INITIALIZE) (void)
```

This function initializes the demosaic module. It must be called prior to calling any other demosaic module API function.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.1.1.2 TL_DEMOSAIC_MODULE_TERMINATE

```
typedef int(* TL_DEMOSAIC_MODULE_TERMINATE) (void)
```

This function gracefully terminates the demosaic module. It must be called prior to unloading the demosaic shared library to ensure proper cleanup of platform resources.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.1.1.3 TL_DEMOSAIC_TRANSFORM_16_TO_48

```
typedef int(* TL_DEMOSAIC_TRANSFORM_16_TO_48) (int width, int height, int x_origin, int y_origin, enum TL_COLOR_FILTER_ARRAY_PHASE color_phase,
enum TL_COLOR_FORMAT output_color_format, enum TL_COLOR_FILTER_TYPE color_filter_type, int bit_depth, unsigned short *input_buffer, unsigned short
*output_buffer)
```

This function takes an input buffer containing monochrome image data and writes a color image into the specified output buffer using a standard demosaic computation.

The transformation can be viewed as "expanding" the single channel monochrome pixel data into three color channels of pixel data.

The implementation uses AVX2 vector instructions to accelerate the computation on machines which support that instruction set.

A legacy scalar implementation is also included to support older generation hardware which do not support the new instructions.

The user does not need to choose between the vector vs. scalar implementation - that is done automatically based on a run time interrogation of the CPU capabilities.

Parameters

in	*width*	The width (x-axis) in pixels of the region of interest (ROI) specified in the input buffer.
in	*height*	The height (y-axis) in pixels of the ROI specified in the input buffer.
in	*x_origin*	The x coordinate of the origin pixel in the ROI relative to the full frame.
in	*y_origin*	The y coordinate of the origin pixel in the ROI relative to the full frame.
in	*color_phase*	The Bayer pattern color (TL_COLOR_FILTER_ARRAY_PHASE) of the origin pixel in the full frame.
in	*output_color_format*	The desired pixel order (TL_COLOR_FORMAT) that should be used when the color data is written to the output buffer.
in	*color_filter_type*	The color filter type (TL_COLOR_FILTER_TYPE) of the device which produced the input data.
in	*bit_depth*	The pixel bit depth of the input buffer data. The maximum number of bits used to specify a pixel intensity value in the input buffer.
in	*input_buffer*	A pointer to the 16-bit input monochrome data.
out	*output_buffer*	A pointer to the output 16-bit color data buffer.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.2 tl_color_enum.h File Reference

Enumerations

- enum TL_COLOR_FILTER_ARRAY_PHASE { TL_COLOR_FILTER_ARRAY_PHASE_BAYER_RED, TL_COLOR_FILTER_ARRAY_PHASE_BAYER_BLUE, TL_COLOR_FILTER_ARRAY_PHASE_BAYER_GREEN_LEFT_OF_BLUE, TL_COLOR_FILTER_ARRAY_PHASE_MAX }
- enum TL_COLOR_FORMAT { TL_COLOR_FORMAT_BGR_PLANAR, TL_COLOR_FORMAT_BGR_PIXEL, TL_COLOR_FORMAT_RGB_PIXEL, TL_COLOR_FORMAT_MAX }
- enum TL_COLOR_FILTER_TYPE { TL_COLOR_FILTER_TYPE_BAYER, TL_COLOR_FILTER_TYPE_MAX }

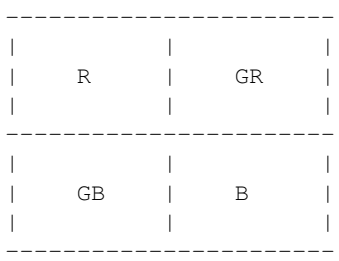
0.3.2.1 Enumeration Type Documentation

0.3.2.1.1 TL_COLOR_FILTER_ARRAY_PHASE

`enum TL_COLOR_FILTER_ARRAY_PHASE`

The TL_COLOR_FILTER_ARRAY_PHASE enumeration lists all the possible values that a pixel in a Bayer pattern color arrangement could assume.

The classic Bayer pattern is



where:

- R = a red pixel
- GR = a green pixel next to a red pixel
- B = a blue pixel
- GB = a green pixel next to a blue pixel

The primitive pattern shown above represents the fundamental color pixel arrangement in a Bayer pattern color sensor. The basic pattern would extend in the X and Y directions in a real color sensor containing millions of pixels.

Notice that the color of the origin (0, 0) pixel logically determines the color of every other pixel.

It is for this reason that the color of this origin pixel is termed the color "phase" because it represents the reference point for the color determination of all other pixels.

Every TSI color camera provides the sensor specific color phase of the full frame origin pixel as a discoverable parameter.

Enumerator

TL_COLOR_FILTER_ARRAY_PHASE_BAYER_RED	A red pixel.
TL_COLOR_FILTER_ARRAY_PHASE_BAYER_BLUE	A blue pixel.
TL_COLOR_FILTER_ARRAY_PHASE_BAYER_GREEN_LEFT_OF_RED	A green pixel next to a red pixel.
TL_COLOR_FILTER_ARRAY_PHASE_BAYER_GREEN_LEFT_OF_BLUE	A green pixel next to a blue pixel.
TL_COLOR_FILTER_ARRAY_PHASE_MAX	A sentinel value (DO NOT USE).

0.3.2.1.2 TL_COLOR_FILTER_TYPE

`enum TL_COLOR_FILTER_TYPE`

The TL_COLOR_FILTER_TYPE enumeration lists all the possible types of color sensor pixel arrangements that can be found in TSI's color camera product line.

Every TSI color camera provides the color filter type of its sensor as a discoverable parameter.

Enumerator

TL_COLOR_FILTER_TYPE_BAYER	A Bayer pattern color sensor.
TL_COLOR_FILTER_TYPE_MAX	A sentinel value (DO NOT USE).

0.3.2.1.3 TL_COLOR_FORMAT

`enum TL_COLOR_FORMAT`

The TL_COLOR_FORMAT enumeration lists all the possible options for specifying the order of color pixels in input and/or output buffers.

This enumeration appears as an argument in certain API functions across the different color modules that a programmer must specify to determine the behavior of that function.

Depending on the context, it can specify

- the desired pixel order that a module must use when writing color pixel data into an output buffer
- the pixel order that a module must use to interpret data in an input buffer.

Enumerator

TL_COLOR_FORMAT_BGR_PLANAR	The color pixels blue, green, and red are grouped in separate planes in the buffer: BBBBBBBB..., GGGGGGGG..., RRRRRRRR....
TL_COLOR_FORMAT_BGR_PIXEL	The color pixels blue, green, and red are clustered and stored consecutively in the following pattern: BGRBGRBGR...
TL_COLOR_FORMAT_RGB_PIXEL	The color pixels blue, green, and red are clustered and stored consecutively in the following pattern: RGBRGBRGB...
TL_COLOR_FORMAT_MAX	A sentinel value (DO NOT USE).

0.3.3 tl_color_error.h File Reference

Enumerations

- enum TL_COLOR_ERROR {
  TL_COLOR_NO_ERROR, TL_COLOR_MODULE_NOT_INITIALIZED, TL_COLOR_NULL_INSTANCE_HANDLE, TL_COLOR_NULL_INPUT_BUFFER_POINTER,
  TL_COLOR_NULL_OUTPUT_BUFFER_POINTER,
  TL_COLOR_IDENTICAL_INPUT_AND_OUTPUT_BUFFERS, TL_COLOR_INVALID_COLOR_FILTER_ARRAY_PHASE, TL_COLOR_INVALID_COLOR_FILTER_TYPE,
  TL_COLOR_INVALID_BIT_DEPTH, TL_COLOR_INVALID_INPUT_COLOR_FORMAT,
  TL_COLOR_INVALID_OUTPUT_COLOR_FORMAT, TL_COLOR_INVALID_BIT_SHIFT_DISTANCE, TL_COLOR_INVALID_CLAMP_VALUE, TL_COLOR_INVALID_INPUT_IMAGE_WIDT
  TL_COLOR_INVALID_INPUT_IMAGE_HEIGHT,
  TL_COLOR_ERROR_MAX }

0.3.3.1 Enumeration Type Documentation

0.3.3.1.1 TL_COLOR_ERROR

`enum TL_COLOR_ERROR`

The TL_COLOR_ERROR enumeration lists all the possible error codes that any color processing API function could return.

Enumerator

TL_COLOR_NO_ERROR	Functions will return this error code to indicate SUCCESS.
TL_COLOR_MODULE_NOT_INITIALIZED	The module has not been initialized therefore it is in an undefined state.
TL_COLOR_NULL_INSTANCE_HANDLE	The specified module instance handle is NULL.
TL_COLOR_NULL_INPUT_BUFFER_POINTER	The specified input buffer pointer is NULL.
TL_COLOR_NULL_OUTPUT_BUFFER_POINTER	The specified output buffer pointer is NULL.
TL_COLOR_IDENTICAL_INPUT_AND_OUTPUT_BUFFERS	The same buffer pointer was specified for both the input and output buffers.
TL_COLOR_INVALID_COLOR_FILTER_ARRAY_PHASE	The specified color filter array phase is invalid.
TL_COLOR_INVALID_COLOR_FILTER_TYPE	The specified color filter type is unknown.
TL_COLOR_INVALID_BIT_DEPTH	The specified pixel bit depth is invalid.
TL_COLOR_INVALID_INPUT_COLOR_FORMAT	The specified input color format is unknown.

Enumerator

TL_COLOR_INVALID_OUTPUT_COLOR_FORMAT	The specified output color format is unknown.
TL_COLOR_INVALID_BIT_SHIFT_DISTANCE	The specified bit shift distance is invalid.
TL_COLOR_INVALID_CLAMP_VALUE	The specified pixel clamp value is invalid.
TL_COLOR_INVALID_INPUT_IMAGE_WIDTH	The specified input image width is invalid.
TL_COLOR_INVALID_INPUT_IMAGE_HEIGHT	The specified input image height is invalid.
TL_COLOR_ERROR_MAX	A sentinel value (DO NOT USE).

0.3.4 tl_color_LUT.h File Reference

Typedefs

- typedef int(∗ INITIALIZE_LUT_MODULE) (void)
- typedef void ∗(∗ CREATE_LUT) (int size)
- typedef int ∗(∗ GET_LUT_DATA) (void ∗handle)
- typedef int(∗ TRANSFORM_LUT_16) (void ∗handle, unsigned short ∗buffer, int number_of_elements)
- typedef int(∗ DESTROY_LUT) (void ∗handle)
- typedef int(∗ TERMINATE_LUT_MODULE) (void)

0.3.4.1 Typedef Documentation

0.3.4.1.1 CREATE_LUT

```
typedef void*(* CREATE_LUT) (int size)
```

This function creates a LUT instance and returns a pointer to that instance.

The LUT instance is a handle to the internal buffer containing the LUT data.

Parameters

in	*size*	The size in bytes of the LUT data array.

Returns

A handle to the LUT instance. A zero (0) handle indicates failure to create the instance.

0.3.4.1.2 DESTROY_LUT

```
typedef int(* DESTROY_LUT) (void *handle)
```

This function destroys the specified LUT instance.

After this function has been called for the specified instance handle, it is an error to subsequently use that instance in any way.

Any attempt to do so could result in undefined and unpredictable behavior.

Parameters

in	*handle*	The LUT instance handle (obtained by calling create_LUT()).

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.4.1.3 GET_LUT_DATA

```
typedef int*(* GET_LUT_DATA) (void *handle)
```

This function returns a pointer to the internal LUT data array.

This would allow a user to directly manipulate individual LUT data elements.

The individual LUT data elements are stored as 32-bit integers.

Parameters

in	*handle*	The LUT instance handle (obtained by calling create_LUT()).

Returns

A pointer to the LUT data buffer. A zero (0) pointer indicates failure.

0.3.4.1.4 INITIALIZE_LUT_MODULE

```
typedef int(* INITIALIZE_LUT_MODULE) (void)
```

This function initializes the LUT module.

It must be called prior to calling any other LUT module API function.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.4.1.5 TERMINATE_LUT_MODULE

```
typedef int(* TERMINATE_LUT_MODULE) (void)
```

This function gracefully terminates the LUT module.

It must be called prior to unloading the LUT shared library to ensure proper cleanup of platform resources.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.4.1.6 TRANSFORM_LUT_16

```
typedef int(* TRANSFORM_LUT_16) (void *handle, unsigned short *buffer, int number_of_elements)
```

This function applies the LUT transform directly to the supplied buffer. It overwrites the data in the supplied buffer so in that sense, it is destructive.

The implementation uses AVX2 vector instructions to accelerate the computation on machines which support that instruction set.

A legacy scalar implementation is also included to support older generation hardware which do not support the new instructions.

The user does not need to choose between the vector vs. scalar implementation - that is done automatically based on a run time interrogation of the CPU capabilities.

Parameters

in	*handle*	The LUT instance handle (obtained by calling create_LUT()).
in,out	*buffer*	The data buffer containing the data to be transformed using the LUT data.
in	*number_of_elements*	The number of (16-bit) elements in the supplied buffer.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.5 tl_color_processing.h File Reference

```
#include "tl_color_enum.h"
```

Typedefs

- typedef int(∗ TL_COLOR_PROCESSING_MODULE_INITIALIZE) (void)
- typedef void ∗(∗ TL_COLOR_CREATE_COLOR_PROCESSOR) (int input_LUT_size_bits, int output_LUT_size_bits)
- typedef int ∗(∗ TL_COLOR_GET_BLUE_INPUT_LUT) (void ∗handle)
- typedef int ∗(∗ TL_COLOR_GET_GREEN_INPUT_LUT) (void ∗handle)
- typedef int ∗(∗ TL_COLOR_GET_RED_INPUT_LUT) (void ∗handle)
- typedef int(∗ TL_COLOR_ENABLE_INPUT_LUTS) (void ∗handle, int blue_LUT_enable, int green_LUT_enable, int red_LUT_enable)
- typedef int(∗ TL_COLOR_APPEND_MATRIX) (void ∗handle, float ∗matrix)
- typedef int(∗ TL_COLOR_CLEAR_MATRIX) (void ∗handle)
- typedef int ∗(∗ TL_COLOR_GET_BLUE_OUTPUT_LUT) (void ∗handle)
- typedef int ∗(∗ TL_COLOR_GET_GREEN_OUTPUT_LUT) (void ∗handle)
- typedef int ∗(∗ TL_COLOR_GET_RED_OUTPUT_LUT) (void ∗handle)
- typedef int(∗ TL_COLOR_ENABLE_OUTPUT_LUTS) (void ∗handle, int blue_LUT_enable, int green_LUT_enable, int red_LUT_enable)
- typedef int(∗ TL_COLOR_TRANSFORM_48_TO_48) (void ∗handle, unsigned short ∗input_buffer, enum TL_COLOR_FORMAT input_buffer_format, unsigned short blue↩ _output_min_value, unsigned short blue_output_max_value, unsigned short green_output_min_value, unsigned short green_output_max_value, unsigned short red_↩ output_min_value, unsigned short red_output_max_value, int output_blue_shift_distance, int output_green_shift_distance, int output_red_shift_distance, unsigned short ∗output_buffer, enum TL_COLOR_FORMAT output_buffer_format, int number_of_bgr_tuples)
- typedef int(∗ TL_COLOR_TRANSFORM_48_TO_64) (void ∗handle, unsigned short ∗input_buffer, enum TL_COLOR_FORMAT input_buffer_format, unsigned short blue↩ _output_min_value, unsigned short blue_output_max_value, unsigned short green_output_min_value, unsigned short green_output_max_value, unsigned short red_↩ output_min_value, unsigned short red_output_max_value, int output_blue_shift_distance, int output_green_shift_distance, int output_red_shift_distance, unsigned short ∗output_buffer, enum TL_COLOR_FORMAT output_buffer_format, int number_of_bgr_tuples)
- typedef int(∗ TL_COLOR_TRANSFORM_48_TO_24) (void ∗handle, unsigned short ∗input_buffer, enum TL_COLOR_FORMAT input_buffer_format, unsigned short blue↩ _output_min_value, unsigned short blue_output_max_value, unsigned short green_output_min_value, unsigned short green_output_max_value, unsigned short red_↩ output_min_value, unsigned short red_output_max_value, int output_blue_shift_distance, int output_green_shift_distance, int output_red_shift_distance, unsigned char ∗output_buffer, enum TL_COLOR_FORMAT output_buffer_format, int number_of_bgr_tuples)

- typedef int(* TL_COLOR_TRANSFORM_48_TO_32) (void *handle, unsigned short *input_buffer, enum TL_COLOR_FORMAT input_buffer_format, unsigned short blue_output_min_value, unsigned short blue_output_max_value, unsigned short green_output_min_value, unsigned short green_output_max_value, unsigned short red_output_min_value, unsigned short red_output_max_value, int output_blue_shift_distance, int output_green_shift_distance, int output_red_shift_distance, unsigned char *output_buffer, enum TL_COLOR_FORMAT output_buffer_format, int number_of_bgr_tuples)
- typedef int(* TL_COLOR_DESTROY_COLOR_PROCESSOR) (void *handle)
- typedef int(* TL_COLOR_PROCESSING_MODULE_TERMINATE) (void)

0.3.5.1 Typedef Documentation

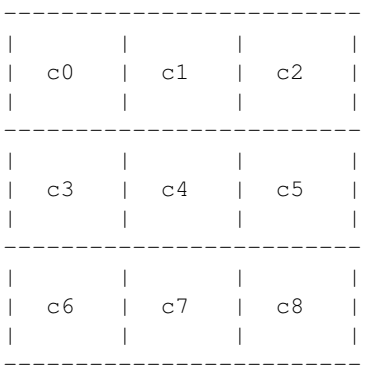
0.3.5.1.1 TL_COLOR_APPEND_MATRIX

```
typedef int(* TL_COLOR_APPEND_MATRIX) (void *handle, float *matrix)
```

This function concatenates a 3x3 matrix to the internal instance resultant matrix.

The ordering of the floating point values in the matrix array relative to their positions in the 3x3 matrix is:

Assuming the specified array coefficients are [c0, c1, c2, c3, c4, c5, c6, c7, c8], the layout of the specified coefficients in the matrix is illustrated below.



Parameters

in	*handle*	The color processing instance handle.
in	*matrix*	A pointer to a 9 element array of 32-bit floating point values that represent the 3x3 matrix coefficients.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.5.1.2 TL_COLOR_CLEAR_MATRIX

```
typedef int(* TL_COLOR_CLEAR_MATRIX) (void *handle)
```

This function resets the internal instance resultant matrix to the identity matrix.

The identity matrix contains the value 1.0 in the diagonal coefficients and 0.0 values for all other coefficients.

```
-------------------------
|       |       |       |
|  1.0  |  0.0  |  0.0  |
|       |       |       |
-------------------------
|       |       |       |
|  0.0  |  1.0  |  0.0  |
|       |       |       |
-------------------------
|       |       |       |
|  0.0  |  0.0  |  1.0  |
|       |       |       |
-------------------------
```

This corresponds to the floating point array: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]

Parameters

in	*handle*	The color processing instance handle.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.5.1.3 TL_COLOR_CREATE_COLOR_PROCESSOR

```
typedef void*(* TL_COLOR_CREATE_COLOR_PROCESSOR) (int input_LUT_size_bits, int output_LUT_size_bits)
```

This function creates a color processing instance and returns a pointer to that instance.

The color processing instance is a handle to the internal color processing state which consists of:

- resultant 3x3 matrix coefficients (initial state is the identity matrix)
- blue color channel input LUT (initial state is all zeroes)
- green color channel input LUT (initial state is all zeroes)
- red color channel input LUT (initial state is all zeroes)
- blue color channel output LUT (initial state is all zeroes)
- green color channel output LUT (initial state is all zeroes)
- red color channel output LUT (initial state is all zeroes)

The user can specify the input and output LUT size separately for added flexibility. The common case would be to set both of these to the bit depth of the color pixel data.

If the user does not need to use the input or output LUTs, then they can set either or both of these sizes to 0 to decrease the memory footprint of the color processing instance.

Parameters

in	*input_LUT_size_bits*	The input LUT size specified in bits. The LUT buffer size (for each color channel) is interpreted to be 2^input_LUT_size_bits elements.
in	*output_LUT_size_bits*	The output LUT size specified in bits. The LUT buffer size (for each color channel) is interpreted to be 2^output_LUT_size_bits elements.

Returns

A handle to the LUT instance. A zero (0) handle indicates failure to create the instance.

0.3.5.1.4 TL_COLOR_DESTROY_COLOR_PROCESSOR

```
typedef int(* TL_COLOR_DESTROY_COLOR_PROCESSOR) (void *handle)
```

This function destroys the specified color processing instance. After this function has been called for the specified instance handle, it is an error to subsequently use that instance in any way. Any attempt to do so could result in undefined and unpredictable behavior.

Parameters

in	*handle*	The color processing instance handle (obtained by calling create_LUT()).

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.5.1.5 TL_COLOR_ENABLE_INPUT_LUTS

```
typedef int(* TL_COLOR_ENABLE_INPUT_LUTS) (void *handle, int blue_LUT_enable, int green_LUT_enable, int red_LUT_enable)
```

This function individually enables the 3 (blue, green, and red) input LUTs.

If a LUT is enabled, its transform function is applied to the data. If a LUT is disabled, its transform is not applied to the data.

This allows the user to (pre)load the LUTs with data and selectively apply them to input data.

Parameters

in	*handle*	The color processing instance handle.
in	*blue_LUT_enable*	Enables/disables the blue input LUT. Set to 1 to enable the LUT or 0 (zero) to disable it.
in	*green_LUT_enable*	Enables/disables the green input LUT. Set to 1 to enable the LUT or 0 (zero) to disable it.
in	*red_LUT_enable*	Enables/disables the red input LUT. Set to 1 to enable the LUT or 0 (zero) to disable it.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.5.1.6 TL_COLOR_ENABLE_OUTPUT_LUTS

```
typedef int(* TL_COLOR_ENABLE_OUTPUT_LUTS) (void *handle, int blue_LUT_enable, int green_LUT_enable, int red_LUT_enable)
```

This function individually enables the 3 (blue, green, and red) output LUTs.

If a LUT is enabled, its transform function is applied to the data. If a LUT is disabled, its transform is not applied to the data.

This allows the user to (pre)load the LUTs with data and selectively apply them to input data.

Parameters

in	*handle*	The color processing instance handle.
in	*blue_LUT_enable*	Enables/disables the blue output LUT. Set to 1 to enable the LUT or 0 (zero) to disable it.
in	*green_LUT_enable*	Enables/disables the green output LUT. Set to 1 to enable the LUT or 0 (zero) to disable it.
in	*red_LUT_enable*	Enables/disables the red output LUT. Set to 1 to enable the LUT or 0 (zero) to disable it.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.5.1.7 TL_COLOR_GET_BLUE_INPUT_LUT

```
typedef int*(* TL_COLOR_GET_BLUE_INPUT_LUT) (void *handle)
```

This function returns a pointer to the blue input LUT data array.

This would allow a user to directly manipulate individual LUT data elements.

The individual LUT data elements are stored as 32-bit integers.

Parameters

in	*handle*	The color processing instance handle.

Returns

A pointer to the LUT data buffer. A zero (0) pointer indicates failure.

0.3.5.1.8 TL_COLOR_GET_BLUE_OUTPUT_LUT

```
typedef int*(* TL_COLOR_GET_BLUE_OUTPUT_LUT) (void *handle)
```

This function returns a pointer to the blue output LUT data array.

This would allow a user to directly manipulate individual LUT data elements.

The individual LUT data elements are stored as 32-bit integers.

Parameters

in	*handle*	The color processing instance handle.

Returns

A pointer to the LUT data buffer. A zero (0) pointer indicates failure.

0.3.5.1.9 TL_COLOR_GET_GREEN_INPUT_LUT

```
typedef int*(* TL_COLOR_GET_GREEN_INPUT_LUT) (void *handle)
```

This function returns a pointer to the green input LUT data array.

This would allow a user to directly manipulate individual LUT data elements.

The individual LUT data elements are stored as 32-bit integers.

Parameters

in	*handle*	The color processing instance handle.

Returns

A pointer to the LUT data buffer. A zero (0) pointer indicates failure.

0.3.5.1.10 TL_COLOR_GET_GREEN_OUTPUT_LUT

```
typedef int*(* TL_COLOR_GET_GREEN_OUTPUT_LUT) (void *handle)
```

This function returns a pointer to the green output LUT data array.

This would allow a user to directly manipulate individual LUT data elements.

The individual LUT data elements are stored as 32-bit integers.

Parameters

in	*handle*	The color processing instance handle.

Returns

A pointer to the LUT data buffer. A zero (0) pointer indicates failure.

0.3.5.1.11 TL_COLOR_GET_RED_INPUT_LUT

```
typedef int*(* TL_COLOR_GET_RED_INPUT_LUT) (void *handle)
```

This function returns a pointer to the red input LUT data array.

This would allow a user to directly manipulate individual LUT data elements.

The individual LUT data elements are stored as 32-bit integers.

Parameters

in	*handle*	The color processing instance handle.

Returns

A pointer to the LUT data buffer. A zero (0) pointer indicates failure.

0.3.5.1.12 TL_COLOR_GET_RED_OUTPUT_LUT

```
typedef int*(* TL_COLOR_GET_RED_OUTPUT_LUT) (void *handle)
```

This function returns a pointer to the red output LUT data array.

This would allow a user to directly manipulate individual LUT data elements.

The individual LUT data elements are stored as 32-bit integers.

Parameters

in	*handle*	The color processing instance handle.

Returns

A pointer to the LUT data buffer. A zero (0) pointer indicates failure.

0.3.5.1.13 TL_COLOR_PROCESSING_MODULE_INITIALIZE

```
typedef int(* TL_COLOR_PROCESSING_MODULE_INITIALIZE) (void)
```

This function initializes the color processing module. It must be called prior to calling any other color processing module API function.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.5.1.14 TL_COLOR_PROCESSING_MODULE_TERMINATE

```
typedef int(* TL_COLOR_PROCESSING_MODULE_TERMINATE) (void)
```

This function gracefully terminates the color processing module. It must be called prior to unloading the color processing shared library to ensure proper cleanup of platform resources.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.5.1.15 TL_COLOR_TRANSFORM_48_TO_24

```
typedef int(* TL_COLOR_TRANSFORM_48_TO_24) (void *handle, unsigned short *input_buffer, enum TL_COLOR_FORMAT input_buffer_format, unsigned short
blue_output_min_value, unsigned short blue_output_max_value, unsigned short green_output_min_value, unsigned short green_output_max_value, unsigned
short red_output_min_value, unsigned short red_output_max_value, int output_blue_shift_distance, int output_green_shift_distance, int output_red↩
_shift_distance, unsigned char *output_buffer, enum TL_COLOR_FORMAT output_buffer_format, int number_of_bgr_tuples)
```

This function performs the color transform computation on the input data and writes the resulting data to the specified output buffer.

This function expects 16-bit input data and produces 8-bit output data.

The specified input and output buffers must be separate.

Parameters

in	*handle*	The color processing instance handle.
in	*input_buffer*	A pointer to the input buffer.
in	*input_buffer_format*	The format (TL_COLOR_FORMAT) that must be assumed to correctly interpret the input data.
in	*blue_output_min_value*	The minimum blue pixel intensity value to allow. Used by the clamp unit to ensure that all blue pixel values are greater than or equal to to this value.
in	*blue_output_max_value*	The maximum blue pixel intensity value to allow. Used by the clamp unit to ensure that all blue pixel values are less than or equal to this value.

Parameters

in	*green_output_min_value*	The minimum green pixel intensity value to allow. Used by the clamp unit to ensure that all green pixel values are greater than or equal to to this value.
in	*green_output_max_value*	The maximum green pixel intensity value to allow. Used by the clamp unit to ensure that all green pixel values are less than or equal to this value.
in	*red_output_min_value*	The minimum red pixel intensity value to allow. Used by the clamp unit to ensure that all red pixel values are greater than or equal to to this value.
in	*red_output_max_value*	The maximum red pixel intensity value to allow. Used by the clamp unit to ensure that all red pixel values are less than or equal to this value.
in	*output_blue_shift_distance*	The distance to shift all blue pixel values by. A positive (+) value specifes a left shift, a 0 specifies no shift, and a negative (-) value specifies a right shift.
in	*output_green_shift_distance*	The distance to shift all blue pixel values by. A positive (+) value specifes a left shift, a 0 specifies no shift, and a negative (-) value specifies a right shift.
in	*output_red_shift_distance*	The distance to shift all blue pixel values by. A positive (+) value specifes a left shift, a 0 specifies no shift, and a negative (-) value specifies a right shift.
out	*output_buffer*	A pointer to the output buffer.
in	*output_buffer_format*	The format (TL_COLOR_FORMAT) that must be used when writing the output data to the output buffer.
in	*number_of_bgr_tuples*	The number BGR 3-tuple quantities in the input and output buffers.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.5.1.16 TL_COLOR_TRANSFORM_48_TO_32

```
typedef int(* TL_COLOR_TRANSFORM_48_TO_32) (void *handle, unsigned short *input_buffer, enum TL_COLOR_FORMAT input_buffer_format, unsigned short
blue_output_min_value, unsigned short blue_output_max_value, unsigned short green_output_min_value, unsigned short green_output_max_value, unsigned
short red_output_min_value, unsigned short red_output_max_value, int output_blue_shift_distance, int output_green_shift_distance, int output_red↩
_shift_distance, unsigned char *output_buffer, enum TL_COLOR_FORMAT output_buffer_format, int number_of_bgr_tuples)
```

This function performs the color transform computation on the input data and writes the resulting data to the specified output buffer.

This function expects 16-bit input data and produces 8-bit output data with a 0 (zero) padding byte for each color 3-tuple.

The specified input and output buffers must be separate.

Parameters

in	*handle*	The color processing instance handle.
in	*input_buffer*	A pointer to the input buffer.
in	*input_buffer_format*	The format (TL_COLOR_FORMAT) that must be assumed to correctly interpret the input data.
in	*blue_output_min_value*	The minimum blue pixel intensity value to allow. Used by the clamp unit to ensure that all blue pixel values are greater than or equal to to this value.
in	*blue_output_max_value*	The maximum blue pixel intensity value to allow. Used by the clamp unit to ensure that all blue pixel values are less than or equal to this value.
in	*green_output_min_value*	The minimum green pixel intensity value to allow. Used by the clamp unit to ensure that all green pixel values are greater than or equal to to this value.
in	*green_output_max_value*	The maximum green pixel intensity value to allow. Used by the clamp unit to ensure that all green pixel values are less than or equal to this value.
in	*red_output_min_value*	The minimum red pixel intensity value to allow. Used by the clamp unit to ensure that all red pixel values are greater than or equal to to this value.
in	*red_output_max_value*	The maximum red pixel intensity value to allow. Used by the clamp unit to ensure that all red pixel values are less than or equal to this value.
in	*output_blue_shift_distance*	The distance to shift all blue pixel values by. A positive (+) value specifes a left shift, a 0 specifies no shift, and a negative (-) value specifies a right shift.
in	*output_green_shift_distance*	The distance to shift all blue pixel values by. A positive (+) value specifes a left shift, a 0 specifies no shift, and a negative (-) value specifies a right shift.
in	*output_red_shift_distance*	The distance to shift all blue pixel values by. A positive (+) value specifes a left shift, a 0 specifies no shift, and a negative (-) value specifies a right shift.
out	*output_buffer*	A pointer to the output buffer.
in	*output_buffer_format*	The format (TL_COLOR_FORMAT) that must be used when writing the output data to the output buffer.
in	*number_of_bgr_tuples*	The number BGR 3-tuple quantities in the input and output buffers.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.5.1.17 TL_COLOR_TRANSFORM_48_TO_48

```
typedef int(* TL_COLOR_TRANSFORM_48_TO_48) (void *handle, unsigned short *input_buffer, enum TL_COLOR_FORMAT input_buffer_format, unsigned short
blue_output_min_value, unsigned short blue_output_max_value, unsigned short green_output_min_value, unsigned short green_output_max_value, unsigned
short red_output_min_value, unsigned short red_output_max_value, int output_blue_shift_distance, int output_green_shift_distance, int output_red↩
_shift_distance, unsigned short *output_buffer, enum TL_COLOR_FORMAT output_buffer_format, int number_of_bgr_tuples)
```

This function performs the color transform computation on the input data and writes the resulting data to the specified output buffer.

This function expects 16-bit input data and produces 16-bit output data.

The specified input and output buffers must be separate and equal in size.

Parameters

in	*handle*	The color processing instance handle.
in	*input_buffer*	A pointer to the input buffer.
in	*input_buffer_format*	The format (TL_COLOR_FORMAT) that must be assumed to correctly interpret the input data.
in	*blue_output_min_value*	The minimum blue pixel intensity value to allow. Used by the clamp unit to ensure that all blue pixel values are greater than or equal to to this value.
in	*blue_output_max_value*	The maximum blue pixel intensity value to allow. Used by the clamp unit to ensure that all blue pixel values are less than or equal to this value.
in	*green_output_min_value*	The minimum green pixel intensity value to allow. Used by the clamp unit to ensure that all green pixel values are greater than or equal to to this value.
in	*green_output_max_value*	The maximum green pixel intensity value to allow. Used by the clamp unit to ensure that all green pixel values are less than or equal to this value.
in	*red_output_min_value*	The minimum red pixel intensity value to allow. Used by the clamp unit to ensure that all red pixel values are greater than or equal to to this value.
in	*red_output_max_value*	The maximum red pixel intensity value to allow. Used by the clamp unit to ensure that all red pixel values are less than or equal to this value.
in	*output_blue_shift_distance*	The distance to shift all blue pixel values by. A positive (+) value specifes a left shift, a 0 specifies no shift, and a negative (-) value specifies a right shift.
in	*output_green_shift_distance*	The distance to shift all blue pixel values by. A positive (+) value specifes a left shift, a 0 specifies no shift, and a negative (-) value specifies a right shift.
in	*output_red_shift_distance*	The distance to shift all blue pixel values by. A positive (+) value specifes a left shift, a 0 specifies no shift, and a negative (-) value specifies a right shift.

Parameters

out	*output_buffer*	A pointer to the output buffer.
in	*output_buffer_format*	The format (TL_COLOR_FORMAT) that must be used when writing the output data to the output buffer.
in	*number_of_bgr_tuples*	The number BGR 3-tuple quantities in the input and output buffers.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

0.3.5.1.18 TL_COLOR_TRANSFORM_48_TO_64

```
typedef int(* TL_COLOR_TRANSFORM_48_TO_64) (void *handle, unsigned short *input_buffer, enum TL_COLOR_FORMAT input_buffer_format, unsigned short
blue_output_min_value, unsigned short blue_output_max_value, unsigned short green_output_min_value, unsigned short green_output_max_value, unsigned
short red_output_min_value, unsigned short red_output_max_value, int output_blue_shift_distance, int output_green_shift_distance, int output_red↩
_shift_distance, unsigned short *output_buffer, enum TL_COLOR_FORMAT output_buffer_format, int number_of_bgr_tuples)
```

This function performs the color transform computation on the input data and writes the resulting data to the specified output buffer.

This function expects 16-bit input data and produces 16-bit output data with a 0 (zero) padding word for each color 3-tuple.

The specified input and output buffers must be separate and equal in size.

Parameters

in	*handle*	The color processing instance handle.
in	*input_buffer*	A pointer to the input buffer.
in	*input_buffer_format*	The format (TL_COLOR_FORMAT) that must be assumed to correctly interpret the input data.
in	*blue_output_min_value*	The minimum blue pixel intensity value to allow. Used by the clamp unit to ensure that all blue pixel values are greater than or equal to to this value.
in	*blue_output_max_value*	The maximum blue pixel intensity value to allow. Used by the clamp unit to ensure that all blue pixel values are less than or equal to this value.

Parameters

in	*green_output_min_value*	The minimum green pixel intensity value to allow. Used by the clamp unit to ensure that all green pixel values are greater than or equal to to this value.
in	*green_output_max_value*	The maximum green pixel intensity value to allow. Used by the clamp unit to ensure that all green pixel values are less than or equal to this value.
in	*red_output_min_value*	The minimum red pixel intensity value to allow. Used by the clamp unit to ensure that all red pixel values are greater than or equal to to this value.
in	*red_output_max_value*	The maximum red pixel intensity value to allow. Used by the clamp unit to ensure that all red pixel values are less than or equal to this value.
in	*output_blue_shift_distance*	The distance to shift all blue pixel values by. A positive (+) value specifes a left shift, a 0 specifies no shift, and a negative (-) value specifies a right shift.
in	*output_green_shift_distance*	The distance to shift all blue pixel values by. A positive (+) value specifes a left shift, a 0 specifies no shift, and a negative (-) value specifies a right shift.
in	*output_red_shift_distance*	The distance to shift all blue pixel values by. A positive (+) value specifes a left shift, a 0 specifies no shift, and a negative (-) value specifies a right shift.
out	*output_buffer*	A pointer to the output buffer.
in	*output_buffer_format*	The format (TL_COLOR_FORMAT) that must be used when writing the output data to the output buffer.
in	*number_of_bgr_tuples*	The number BGR 3-tuple quantities in the input and output buffers.

Returns

A TL_COLOR_ERROR value to indicate success or failure (TL_COLOR_NO_ERROR indicates success).

Index