Enable Color Management on Linux

By

Jesse Barnes, Sirisha Muppavarapu,
Shashank Sharma and Susanta Bhattacharjee

Revision History

Revision	Date	Modified by	Details
0.5	1st May 2015	Jesse Barnes, Sirisha Muppavarapu	Initial version
0.6	7th May 2015	Shashank Sharma, Susanta Bhattacharjee	Modified structures for querying capabilities dynamically. Added notes on usage of the structures.
0.7	8th May 2015	Jim Bish	Add user space documentation for generating the CSC matrix values from source and destination chromaticity values.
0.71	11th May 2015	Susanta Bhattacharjee	Added CSC format for VLV
0.72	14th May 2015	Shashank Sharma	Added KMD design and details
0.73	18th May 2015	Susanta Bhattacharjee	Added 16 bit gamma precision for VLV. Renamed gamma_type to gamma_precision
0.74	21 May 2015	Shashank Sharma	Added task to restore color correction across suspend/resume cycles in UMD responsibility section
0.75	1 June 2015	Shashank Sharma	Added list / credit of reviewers
0.76	12 June 2015	Susanta Bhattacharjee	Added generic one dimensional lookup table manipulation algorithms

0.8	14 June 2015	Shashank Sharma	Added new data structures agreed over the design discussions. They are generic across the drivers. Also added description about how to get color capabilities of a platform, how to get a property and how to set a property, in the driver side design section. Also added Damien in the reviewers list
0.81	16 June 2015	Susanta Bhattacharjee	Explained platform HW capability more explicitly. Changed CSC to CTM for better understanding.

Scope

This document provides a design overview of Color Management support on core Linux. In this document we take ChromeOS as our target OS (as a case study) and suggest the modifications to the Linux kernel module to support calibrated display profile.

Primary Authors

Jesse Barnes
Sirisha Muppavarapu
Shashank Sharma
Susanta Bhattacharjee

Technical Discussions contributors:

Kalyan Kondapally
Ander Conselvan De Oliveira

Reviewers:

Matthew D Roper
Damien Lespiau
Sonika Jindal

Background

A key element in the success of any user experience is the quality and performance of their display. The only way that a display will deliver good color and grayscale accuracy is if it is calibrated to an industry standard specification, which for computers, digital cameras, and HDTVs is sRGB or Rec.709. Chromaticity and Gamma calibration are two key elements which can improve the display quality.

We want the display to work best on Intel Architecture for better user experience. The primary task in this project is to expose the hardware capabilities in the kernel through the KMS (Kernel Mode Settings)

Hardware Capabilities

The following are the hardware capabilities exposed on IA. Specified palette tables are applicable separately for three color channels.

- Baytrail (BYT)
    - 256 x 8 bit palette table
    - 129 x 16 bit palette table - HW interpolated
    - 3x3 Color Transform Matrix (CTM) - also called Color Space Conversion (CSC) matrix
    - Per sprite 8 x 10 bit palette table - HW interpolated
    - Per sprite contrast, hue, brightness and saturation control

- Haswell(HSW), Broadwell(BDW), Skylake(SKL)
    - 256 x 8 bit palette table
    - 1025 x 10 bit palette table
    - 513 x 12 bit palette table - HW interpolated
    - 512 x 10 bit pre-CTM palette table + 512 x 10 bit post-CTM palette table. This is called split palette mode
    - Per plane 18 x 10 bit palette table
    - 3x3 Color Transform Matrix (CTM) - also called Color Space Conversion (CSC) matrix
    - Gamut Enhancement (mapping for expansion or compression) using 17 saturation bins

- Cherryview(CHV)

    - 65 x 14 bit pre-CTM palette table - also referred as Degamma
    - 257 x 10 post-CTM palette table - also referred as gamma
    - 256 x 8 bit palette table - same as BYT
    - 129 x 16 bit palette table - HW interpolated - same as BYT
    - 3x3 Color Transform Matrix (CTM) - also called Color Space Conversion (CSC) matrix

○ Gamut Enhancement (expansion or compression)

Table of color correction capabilities across various Intel platforms

Platform	property	level	Types	Use case
BYT	Palette	pipe	256 x 8 bit x 3	Useful mostly for linear curves
			129 x 16 bit x 3	Gamma calibration, Brightness/Contrast adjustment, White point adjustment
		plane	8 x 10 bit x 3	
	CTM	pipe	3x3 matrix	Hue/Saturation/White point adjustment
	Contrast	plane	single value	Adjustments only on sprite planes. May be useful for video planes.
	Brightness	plane	single value	
	Hue	plane	single value	
	Saturation	plane	single value	
CHT	Palette	pipe	256 x 8 bit x 3	Useful mostly for linear curves
			129 x 16 bit x 3	Gamma calibration, Brightness/Contrast adjustment, White point adjustment
		plane	8 x 10 bit x 3	
		pipe	65 x 14 bit x 3 Pre CTM and 257 x 10 bit x 3 post CTM	ICC Profile along with 3x3 CTM, Hue/Saturation/White point adjustment
	CTM	pipe	3x3 matrix	
	Contrast	plane	single value	Adjustments only on sprite planes. May be useful for video planes.
	Brightness	plane	single value	
	Hue	plane	single value	

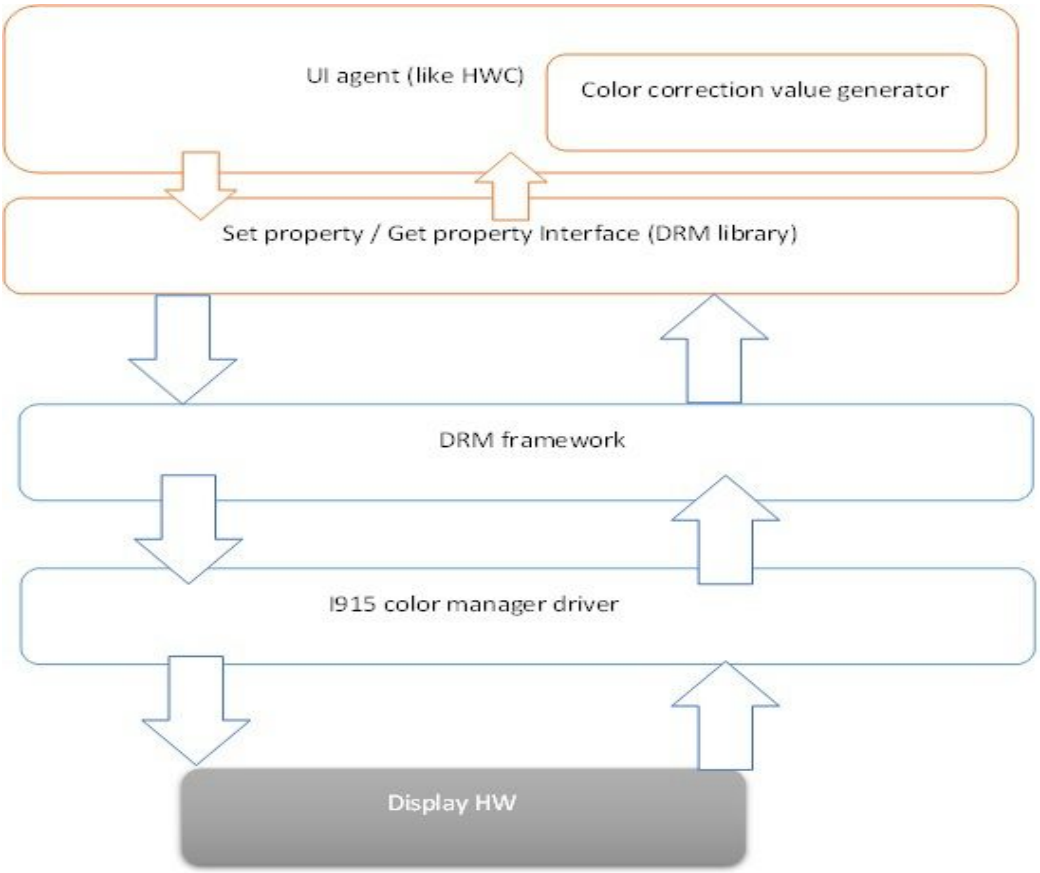
	Saturation	**plane**	single value	
HSW / BDW / SKL / BXT	**Palette**	**plane**	18 x 10 bit x 3	Gamma calibration, Brightness/Contrast adjustment, White point adjustment
		pipe	1025 x 10 bit x 3	
			513 x 12 bit x 3	
			256 x 8 bit x 3	Useful mostly for linear curves
			512 x 10 bit x 3 pre and post CTM	ICC Profile along with 3x3 CTM, Hue/Saturation/White point adjustment
	CTM	**pipe**	3x3 matrix	
	CGE	**pipe**	expansion	Can transform pixels with specified saturation band. Used in windows CUI.
			compression	

Architecture - The BIG Picture

- Driver can program and fine tune HW, to provide an enhanced and more attractive/accurate display.
- These color correction and color management coefficients can be calculated by any user space UI client in form of:
    - Raw color correction values
    - Color profiles
    - Color settings from an UI app

Color correction process needs following software modules:

- A user space UI agent (like HWC for Android), which decides the color profile to be applied, or which color modes to be programmed. This decision making can be based on direct user input, or based on the usage mode.
- A userspace UI app / agent / library to parse the color profiles / color modes and prepares the raw color correction values for the driver.
- An interface, to pass the correction values to KMD
- KMD to formats the values as required by the target HW platform and programs display HW.



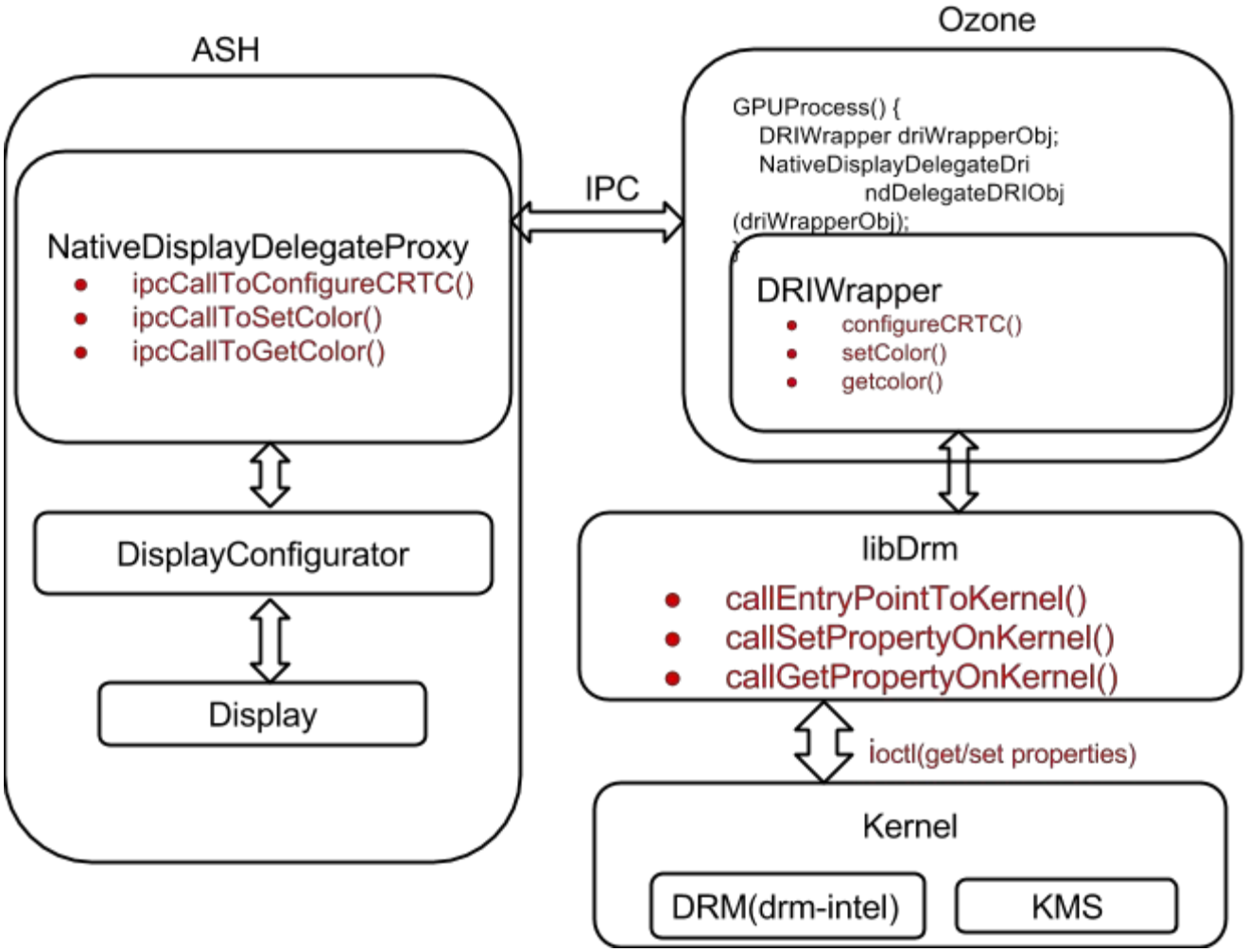
User Space

When ChromeOS boots, there are 3 processes that are started. 1) Browser Process(BP), 2) Render Process(RP) and 3) GPU Process(GPUP). The access to the HW/GPU is only through the GPU Process. An app runs in the Render process. Whenever RP requests memory, BP requests the GPUP on its behalf and GPUP is the one which does the communication with the HW/GPU through mesa, libDRM,Ozone(through IPC)etc.

ASH is the Window Management module which lives in the Browser process.
Ozone: Ozone is a set of abstract interfaces and there are OS specific ozone implementations that actually implement these interfaces. ex: Ozone-wayland is for Tizen and ozone-GBM/DRI for ChromeOS. So when BP and GPUP are communicating, it is actually ASH communicating with Ozone through IPC.

Going a bit deeper to the next level, In the BP, The *Display* class talks to *DisplayConfigurator* class which inturn talks to the *NativeDisplayDelegateProxy* class. This proxy class talks to Ozone in GPUP through *GPUProcessHost* class using IPC. The *GPUProcessHost* inturn talks to *NativeDisplayDelegateDRI* using IPC. The *NativeDisplayDelegateDRI* class talks to *DriWrapper* class which is a wrapper to the libDrm methods. The methods in libDrm are the

wrappers for the actual ioctls in kernel DRM module that do the get/set on the properties using the entrypoints to the kernel.



Color Management Requirements

1) The APIs need to be using Properties with Atomic Mode Setting. **libDrm interfaces and Kernel Modules API/implementation will be Intel contribution.**
2) As per “(Google) Add support for default profile per panel, should be SoC agnostic. Chrome will read the panel's EDID id string and load the correct profile.” **The support in Ozone will be provided by Google**
3) As per “(Samsung) OEM/Panel Vendors will provide profile.” **Samsung is responsible for generating the calibrated profile files.**

KMD/Driver's responsibilities:

- Publish platform capability so that user mode code can be made platform agnostic
- Parse raw color correction data coming from user space, check validity of the data and format as expected by target HW
- Arrange in the desired register format, and program the correction registers
- Enable / Disable the correction
- Cache and restore the color correction values across suspend/resume cycles to maintain the display quality

UMD / Userspace / UI Manager's responsibilities:

- Parse the color profile / color correction values, and pack the DRM data structures for color corrections
- Persist the values across reboots / suspend / resume cycles.

Kernel side Atomic Mode Setting

Lot of things are unclear from the kernel side.

1) What is the entrypoint to Kernel for Atomic Mode Setting?
2) Probably use *drm_mode_obj_set_property_ioctl*() for ioctls?
3) There seems to be some existing support for gamma tables. how do we reuse the infrastructure with Atomic properties? And what happens to the legacy support?
4) How to handle the info in the bug filed by Ville (see the link for more details) https://jira01.devtools.intel.com/browse/VIZ-4361 suggests: May be not a issue for POC?
    a) convert set_gamma_ioctl to blob property
    b) "The gamma table registers are not double buffered on vblank, so in order to make the update appear atomic we'll need to do the register writes during the vblank. There are a lot of registers involved, so trying to do the update from the vblank interrupt itself might not be a good idea and instead we'll need some kind of vblank work for it."
    c) "The hardware supports modes where the gamma table itself is larger or a ramp, and also each entry has higher bpc. I'm not entirely sure how we'll handle this."
    d) "On more recent hardware we have the option of using 10 bit gamma tables. This is 1024 entry table with 10bpc. There's also an extra 1025th entry to deal with color values >1.0."
    e) "There's also another kind of gamma mode. In this mode we just provide the hardware a set of gamma points (128 for old hw, 512 for new hw), and the hardware will linearly interpolate between them to mimic a larger gamma table."

f) “And there's another type of gamma mode on more recent platforms. This basically the 10bit mode split into two half size gamma tables. So two gamma tables of 512 entries each with 10bpc.”
g) “All the >8bpc modes also have an extra gamma entry for color values 1.0-3.0. The hardware will linearly interpolate between the 1.0 entry and the extra entry to produce the pixel data. The problem here is that the extra entry actually uses >16bit values, so if we want to expose the full hardware capabilities, we'll need to overcome the 16bpc limit somehow.”

As per a presentation from Rob Clark on KMS:Atomic Mode Setting: http://www.x.org/wiki/Events/XDC2012/XDC2012AbstractRobClark-KMS/xdc2012-atomic-pagefilp.pdf the following needs to be done for Atomic Modeset:

1) Properti-fication
2) Split Mode object mutable state
3) Use Atomic functions finally

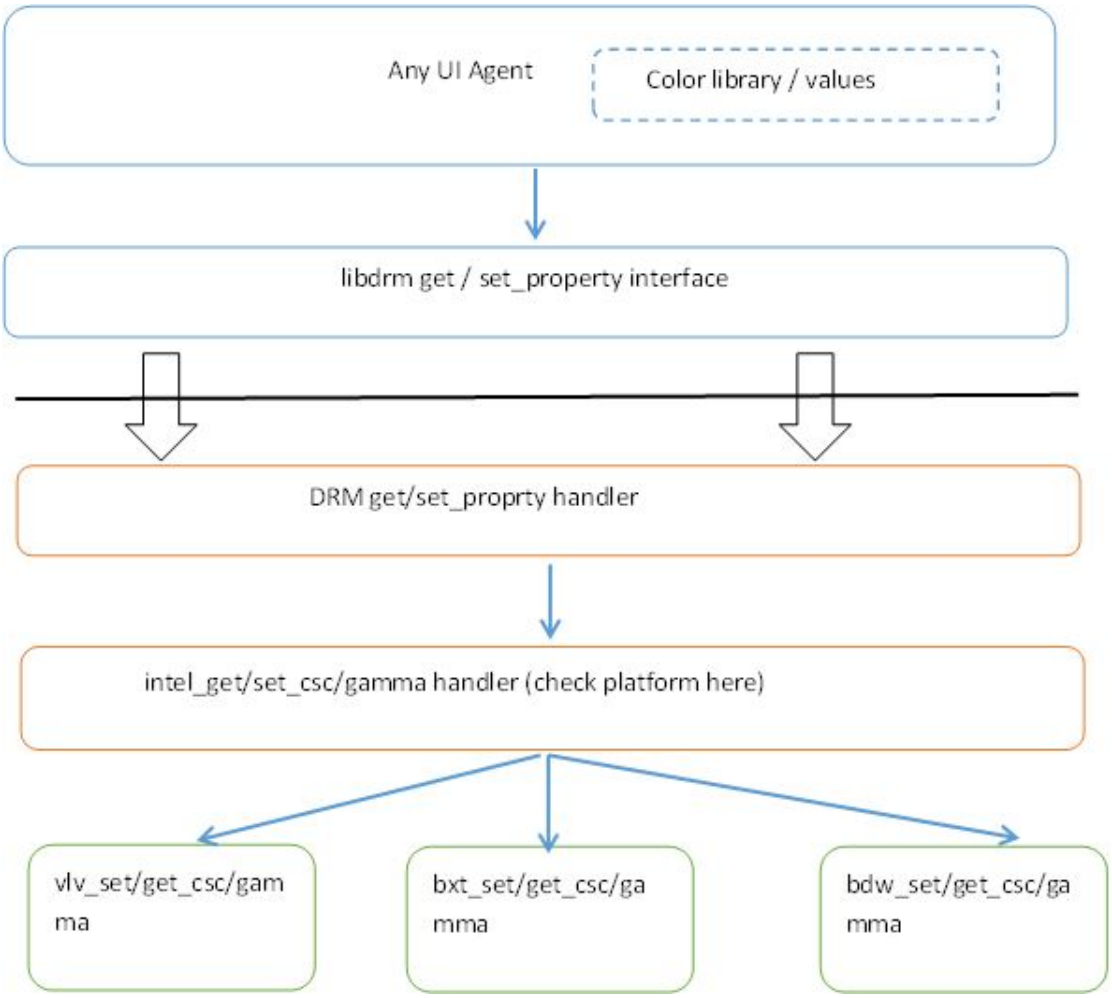
Kernel API Requirements

Gamma, CTM and CGE needs to be implemented as properties for Atomic Mode Setting.

- Gamma - needs to support 8, 10, 12 bit
  - up to 4k table needs to be loaded
- CTM - just a 3x3 matrix
  - small array should be fine
- CGE - 17 weights
  - 5 dwords of registers

Driver side - High level design

Set / Get property / Query color capabilities:



How to query platform color capabilities

- Driver provides a read-only property to showcase the color capabilities of the complete platform, and attach to each CRTC/Plane accordingly.
- Userspace can query the platform capabilities by reading this property.
- **Data structure to be used for query:**

```
/* Get platform capabilities of palette correction property (like Gamma
 * correction) */
struct drm_palette_caps {
        /* Structure version. Should be 1 currently */
        in __u32 version;

        /* Number of entries in palette_sampling_types array below*/
        out __u32 num_supported_types;

        /* Every entry in this array will indicate details of
         * one mode of palette correction property. Any platform can publish up to
         * 4 sampling types in order of descending preference
```

```
         * for example, gamma correction on Intel HW is supported in 3 different
         * modes 8 bit, 10 bit and 12 bit mode. Every mode has different no of
         * corrections coefficients expected. Details of the structure below. */
        out palette_sampling_types[4];
};


/* Palette color correction can be supported in different modes in a HW.
 * This array gives details of such modes of operations. For example,
 * gamma correction on Intel HW can be supported in 8 bit mode with
 * 256 coefficients or 10 bit mode with 128 coefficients. */
struct drm_palette_sampling_details {
        /* Maximum possible precision for the fractional part of curve samples
         * This tells driver how many bits to consider for the fractional part */
        __u32 sample_fract_precisions;

        /*  Palette sample value > 1.0 is supported only for the last sample on most
        Intel HW platforms. This tells the maximum possible integer value for the last
        sample in the palette table supported by the HW. */
        __u32 last_sample_int_max;

        /* Max value of the integer part of the samples in the palette table upto last
        but one. This is 0 on all present Intel HW platforms. Future Intel HW or HW
        from other IHVs may have this > 0. */
        __u32 remaining_sample_int_max;
        /* Number of samples required */
        __u32 num_samples;
};

/* CTM = color transformation matrix. This structure showcases the
 * platform's capabilities of color transformation using a matrix */
struct drm_ctm_caps {
        /* Structure version. Should be 1 currently */
        in __u32 version;

        /* Maximum possible precision for the fractional part of matrix
         * coefficients in bits */
        out __u32 ctm_coeff_fract_precisions;

        /* Maximum possible value for the integer part of matrix coefficients */
        out __u32 ctm_coeff_int_max;

        /* Minimum possible (negative) value for the integer part of matrix
         * coefficients */
        out __i32 ctm_coeff_int_min;
};

/* CGE = Color gamut enhancement. This structure showcases the
 * platform's capabilities of color gamut enhancement */
struct drm_cge_caps {
        /* Structure version. Should be 1 currently */
        in __u32 version;

        /* Maximum value of CGE weights */
        out __u32 cge_max_weight;
};
```

```
/* Overall structure to showcase all the color capabilities of the platform */
struct drm_color_caps {
        /* Structure version. Should be 1 currently */
        in __u32 version;
        /* For padding and future use */
        __u32 reserved;

        /* Palette capabilities to be applied after CTM */
        out struct drm_palette_caps palette_after_ctm_caps;

        /* Palette capabilities to be applied before CTM */
        out struct drm_palette_caps palette_before_ctm_caps;

        /*  capabilities CTM */
        out struct drm_ctm_caps ctm_caps;
        /*  capabilities CGM */
        out struct drm_cge_caps cgm_caps;
};
```

Data Structures to get/set a color correction property.

```
/* Raw R G B pixel / correction values */
typedef struct {
        __u32 r32;     // Data is in 16.16 fixed point format
        __u32 g32;     // Data is in 16.16 fixed point format
        __u32 b32;     // Data is in 16.16 fixed point format
}__drm_r32g32b32;

/* Structure to set / get a CTM property */
struct drm_ctm {
        /* Structure version. Should be 1 currently */
        in __u32 version;

        /* Each value is in S15.16 format. This is 3x3 matrix in row major format
         * Integer part will be clipped to nearest max/min boundary as indicated by
         *ctm_coeff_int_max */
        in out __u32 ctm_coeff [9];
};

/* Structure to set/get a palette property (like gamma) */
struct drm_palette {
        /* Structure version. Should be 1 currently */
        in __u32 version;
        /* Get call => This will reflect the last set value */
        /* Set call => This has to be a supported value as published during capability query.
        Feature will be be disabled if num_samples is 0 */
        in out __u32 palette_num_samples;
        /* Starting of palette LUT in R32G32B32 format. Each of RGB value is in 16.16
         * unsigned fixed point format. Actual number of samples will depend upon
         * palette_num_samples. This is populated with last applied value while get call. */
        in out __drm_r32g32b32 palette_lut[0];
};

/* Structure to set/get for a gamut enhancement property */
struct drm_pipe_cge {
```

```
        /* Structure version. Should be 1 currently */
        in __u32 version;
        #define I915_CGE_ENABLED (1 << 1)
        __u32 flags;
        /* For padding and future use */
        __u32 reserved;
        /* Values should be in range [0, maxVal]. maxVal is reported in capability */
        __u32 cge_weights[17];
};
```

How to get/set a color correction property.

- DRM framework defines new structures at framework and UAPI level (struct drm_color_caps, drm_palette / drm_ctm / drm_cge etc).
- Driver to create master capabilities property, and fill with a blob of struct drm_color_caps. This property will be used to answer queries of platform capabilities.
- Driver create the palette property, cte property and cgm property using struct drm_property **drm_property_create*(struct drm_device *dev, int flags, const char *name, int num_values). All properties will be blob type.
- Attach the property to CRTC using *void drm_object_attach_property(struct drm_mode_object *obj,struct drm_property *property,uint64_t init_val)*.
- Get capabilities:
    - USP to call get_property on master color correction property
    - Kernel will respond with all the platform color properties, in form of blob of drm_color_caps type .

- Set:
    - USP to create a structure drm_palette / drm_ctm / drm_cge and fill correction values in corresponding section.
    - USP to create a blob out of this structure, using drm layer's set_blob_ioctl
    - USP to pass the blob_id of the blob as .value of set_porperty_ioctl for the corresponding property.
    - Kernel will find the corresponding blob using the blob_id, and apply the correction values.
- Get:
    - USP to call a get_property() call to get the blob_id.
    - DRM/driver will load the blob_id for the current blob as a response.
    - USP to call a get_blob() with that blob_id
    - DRM framework / driver to respond with the appropriate blob.
    - USP to decode the blob and get the correction values.

2. Split Mode Object Mutable State

On the lines of what is given in the above presentation for planes, if we extend that to crtc, to split mode object mutable state, we need 3 methods to do this:

- *int drm_crtc_check_state(struct drm_crtc *crtc, struct drm_crtc_state *state)*
- *void drm_crtc_commit_state(struct drm_crtc *crtc,struct drm_crtc_state *state);*
- *int drm_crtc_set_property(struct drm_crtc *crtc,struct drm_crtc_state *state,*
  *struct drm_property *property,uint64_t value, void *blob_data)*

These methods are not existing in the upstream kernel. But the patches by Rob Clark are provided in http://lists.freedesktop.org/archives/dri-devel/2014-July/064582.html

Data Structures used in the above methods:

- *struct drm_crtc* - defined in *include/drm/drm.h*
- *struct drm_crtc_state* - mutable srtc state is defined in Rob Clark's patch above is a newer version than thats already defined in include/drm/drm.h. The state struct is needed to check the validity of the mode object
- *struct drm_property* - is defined in include/drm/drm.h

Drivers should wrap state structs w/ their own to add driver specifics:

- What does this mean?
- Where does the code go?
- What needs to be done here?
- We need to put HW specific infrastructure into the intel driver

3. Atomic Functions

The presentation above suggests 4 atomic methods:
• *atomic_begin*(dev) - allocate state token
• *atomic_check*(dev, state) – check proposed state
– Use drm_*_check_state() for common stuff
• *atomic_commit*(dev, state) – commit proposed state
– Do driver specific stuff, then drm_*_commit_state()
• *atomic_end*(dev, state) – cleanup/deallocate

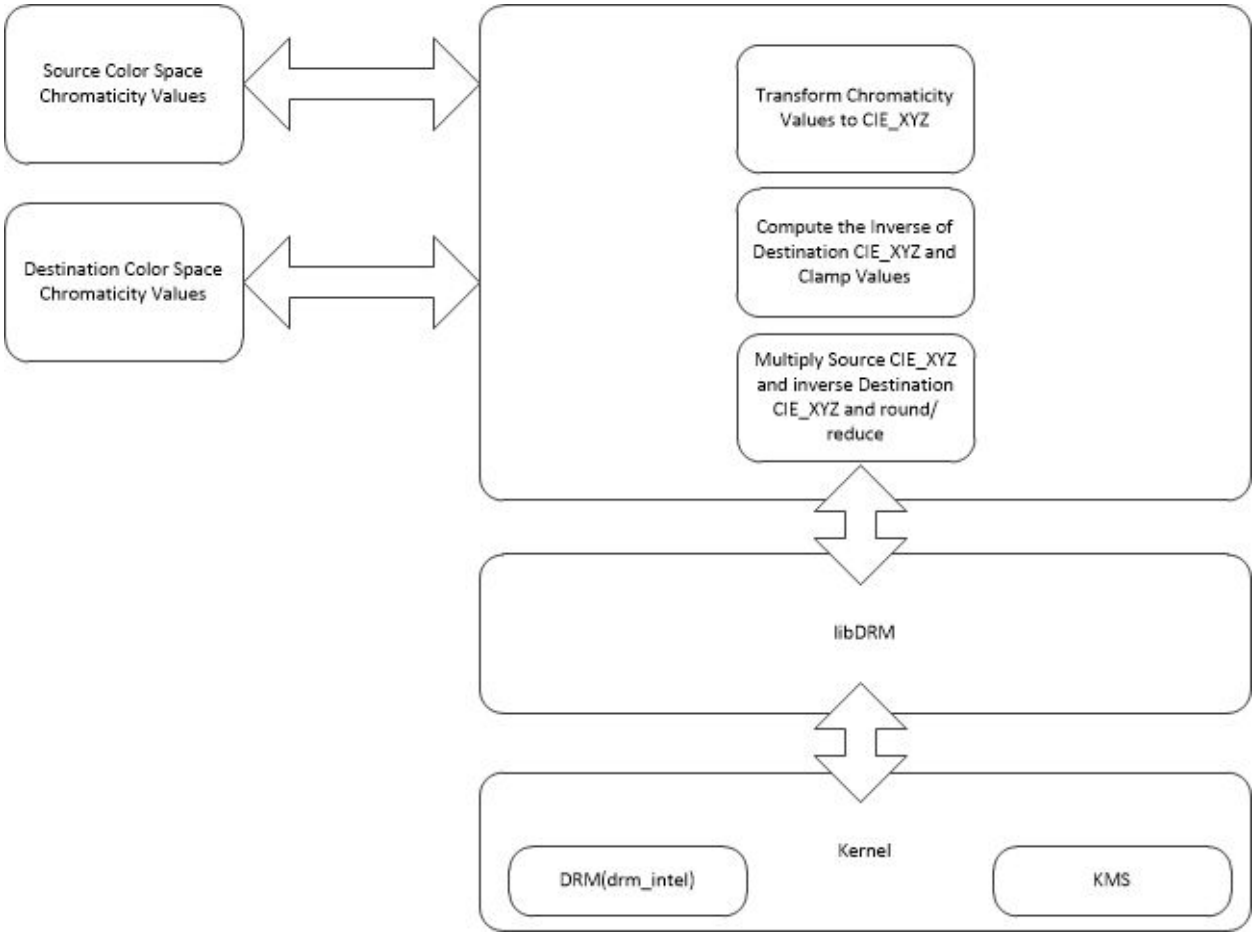
These proposed methods may reside in the file drivers/gpu/drm/drm_atomic.c as per the patch http://lists.freedesktop.org/archives/dri-devel/2014-July/064582.html But the upstream code doesnt have any matching code in drm_atomic.c

The corresponding methods are:

1) *struct drm_atomic_state *drm_atomic_begin(...)*
2) *int drm_atomic_check(struct drm_device *dev, struct drm_atomic_state *state);*
    a) calls *drm_atomic_check_crtc_state*() on all the registered crtcs
3) *static int atomic_commit(...)*
    a) calls *drm_atomic_commit_crtc_state*() on all the registered crtcs
4) *atomic_end*(dev,state)

Computing CTM coefficients

The diagram below highlights the flow for computing the coefficients for the CTM. The inputs for the calculation are the source and destination chromaticity values for the color space.



The following table lists the chromaticity values for common color spaces:

Color Space	White Point		Primary Colors					
	xW	yW	Xr	Yr	Xg	Yg	Xb	Yb
ITU-R BT.2020	0.3127	0.3290	0.708	0.292	0.170	0.797	0.131	0.046
ITU-R BT.709	0.3127	0.3290	0.64	0.33	0.30	0.60	0.15	0.06
ITU-R BT.601 (625 Line)	0.3127	0.3290	0.640	0.330	0.290	0.600	0.150	0.060
ITU-R BT.601 (525 Line)	0.3127	0.3290	0.630	0.340	0.310	0.595	0.155	0.070
ProPhoto RGB	0.3457	0.3585	0.7347	0.2653	0.1596	0.8404	0.0366	0.0001
Wide Gamut RGB	0.3457	0.3585	0.7347	0.2653	0.1152	0.8264	0.1566	0.0177
Adobe RGB	0.3127	0.3290	0.6400	0.3300	0.2100	0.7100	0.1500	0.0600
sRGB	0.3127	0.3290	0.6400	0.3300	0.3000	0.6000	0.1500	0.0600
CIE-RGB (1931 white D65)	0.31271	0.32902	0.7347	0.2653	0.2738	0.7174	0.1666	0.0089
PAL	0.31271	0.32902	0.64	0.33	0.29	0.60	0.15	0.06
NTSC (1987)	0.31271	0.32902	0.63	0.34	0.31	0.595	0.155	0.07
NTSC (1953)	0.310	0.316	0.67	0.33	0.21	0.71	0.14	0.08
NTSC (Japan)	0.2848	0.2932	0.63	0.34	0.31	0.595	0.155	0.07

Note: for panel adjustment a colorimeter can be used to fine tune the chromaticity values.

The user space would define a function with a signature similar to below:

```
struct chromaticity_entry {
  int xR;  // primaries
  int yR;
  int xG;
  int yG;
  int xB;
  int yB;
```

```
  int xW;  // white point
  int yW;
};
```

```
int convert_to_ctm(struct chromaticity_entry *src, struct chromaticity_entry *dst,
        int64_t *ctm)
```

The flow for this function is as follows:

- Convert source and destination chromaticity values to integer. Multiply by 1000
- Convert source and destination chromaticity values to CIE_XYZ format
- Calculate the inverse matrix from the destination matrix in CIE_XYZ format
- Clamp if necessary for < 0 values
- Multiple the source CIE_XYZ matrix by the inverse destination CIE_XYZ matrix. The resulting matrix is stored in the *ctm parameter.
- Round slightly up/down depending on sign of the result and divide by 10

Note: All math should be performed using unsigned int64 types. The *ctm result from above is ready to be passed via the struct **drm_intel_ctm** defined above to drm.

Below is a hacky unoptimized implementation (taken from mdfld_ctm.c written by Jim Liu at Intel for the MCG kernel) of a command line test tool that will convert source sRGB chromaticity to BT2020 chromaticity. It produces two output matrices. One is the unsigned int64 values and the second is formatted to match the bspec definition of CSC_COEFFICIENT_FORMAT for Skylake.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <math.h>
#include <inttypes.h>

#define MULTIPLIER_CHROM          10000
#define MULTIPLIER_MATRIX         10000
#define MULTIPLIER_MATRIX1        100000
#define MULTIPLIER_MATRIX2        100
#define BIT10                     0x0400
#define CC_1_POS                  16

#define ctm_sign(x) ((x < 0) ? false : true)
```

```
// simple arrays instead of cleaner structures for test app.
int BT_2020[8] =        {7080, 2920, 1700, 7970, 1310, 0460, 3127, 3290};
int sRGB[8] =           {6400, 3300, 3000, 6000, 1500, 0600, 3127, 3290};

#define SRGB            1
#define bt_2020         2

int align_to(int arg, int align)
{
   return ((arg + (align - 1)) & (~(align - 1)));
}

/**
*  ctm_matrix_mult_func
*
*
*/
static int64_t ctm_matrix_mult_func(int64_t * M1, int64_t * M2)
{
   int64_t result = 0;
   int i = 0;

   for (i = 0; i < 3; i++)
      result += M1[i] * M2[i];

   return result;
}

/**
*  ctm_matrix_mult
*
*  3x3 matrix multiply 3x3 matrix
*
*/
static void ctm_matrix_mult(int64_t * M1, int64_t * M2, int64_t * M3)
{
   int64_t temp1[3], temp2[3];
   int i = 0;

   for (i = 0; i < 3; i++) {
      temp1[i] = M1[i];
      temp2[i] = M2[i * 3];
   }
```

```
M3[0] = ctm_matrix_mult_func(temp1, temp2);

for (i = 0; i < 3; i++) {
    temp1[i] = M1[i];
    temp2[i] = M2[(i * 3) + 1];
}

M3[1] = ctm_matrix_mult_func(temp1, temp2);

for (i = 0; i < 3; i++) {
    temp1[i] = M1[i];
    temp2[i] = M2[(i * 3) + 2];
}

M3[2] = ctm_matrix_mult_func(temp1, temp2);

for (i = 0; i < 3; i++) {
    temp1[i] = M1[i + 3];
    temp2[i] = M2[i * 3];
}

M3[3] = ctm_matrix_mult_func(temp1, temp2);

for (i = 0; i < 3; i++) {
    temp1[i] = M1[i + 3];
    temp2[i] = M2[(i * 3) + 1];
}

M3[4] = ctm_matrix_mult_func(temp1, temp2);

for (i = 0; i < 3; i++) {
    temp1[i] = M1[i + 3];
    temp2[i] = M2[(i * 3) + 2];
}

M3[5] = ctm_matrix_mult_func(temp1, temp2);

for (i = 0; i < 3; i++) {
    temp1[i] = M1[i + 6];
    temp2[i] = M2[i * 3];
}
M3[6] = ctm_matrix_mult_func(temp1, temp2);
```

```
    for (i = 0; i < 3; i++) {
        temp1[i] = M1[i + 6];
        temp2[i] = M2[(i * 3) + 1];
    }

    M3[7] = ctm_matrix_mult_func(temp1, temp2);

    for (i = 0; i < 3; i++) {
        temp1[i] = M1[i + 6];
        temp2[i] = M2[(i * 3) + 2];
    }

    M3[8] = ctm_matrix_mult_func(temp1, temp2);
}

/**
*  ctm_div64
*
*  division with 64bit dividend and 64bit divisor.
*
*/
static int64_t ctm_div64(int64_t dividend, int64_t divisor)
{
    bool sign = !(ctm_sign(dividend) ^ ctm_sign(divisor));
    uint64_t temp_N = (uint64_t) - 1;
    uint64_t temp_divid = 0, temp_divis = 0, temp_quot = 0;

    if (dividend < 0) {
        temp_divid = temp_N - ((uint64_t) dividend) + 1;
    } else {
        temp_divid = (uint64_t) dividend;
    }

    if (divisor < 0) {
        temp_divis = temp_N - ((uint64_t) divisor) + 1;
    } else {
        temp_divis = (uint64_t) divisor;
    }

    temp_quot = temp_divid / temp_divis;

    if (!sign)
```

```
        temp_quot = temp_N - ((uint64_t) temp_quot) + 1;

    return (int64_t) temp_quot;
}

/**
*  ctm_det_2x2_matrix
*
*  get the determinant of 2x2 matrix
*
*  note: the 2x2 matrix will be represented in a vector with 4 elements.
*  M00 = V0, M01 = V1, M10 = V2, M11 = V3.
*  det M = V0 * V3 - V1 * V2
*/
int64_t ctm_det_2x2_matric(int64_t * M2)
{
    return ((M2[0] * M2[3]) - (M2[1] * M2[2]));
}

/**
*  ctm_det_3x3_matrix
*
*  get the determinant of 3x3 matrix
*
*  note: the 3x3 matrix will be represented in a vector with 9 elements.
*  M00 = V0, M01 = V1, M02 = V2, M10 = V3, M11 = V4, M12 = V5, M20 = V6, M21
*  = V7, M22 = V8.
*  det M = V0*(V8*V4 - V7*V5) - V3*(V8*V1 - V7*V2) + V6*(V5*V1 - V4*V2)
*/
int64_t ctm_det_3x3_matric(int64_t * M3)
{
    int64_t M2_0[4], M2_1[4], M2_2[4];
    int64_t det0 = 0;
    int64_t det1 = 0;
    int64_t det2 = 0;

    M2_0[0] = M3[4];
    M2_0[1] = M3[5];
    M2_0[2] = M3[7];
    M2_0[3] = M3[8];
    det0 = ctm_det_2x2_matric(M2_0);

    M2_1[0] = M3[1];
```

```
    M2_1[1] = M3[2];
    M2_1[2] = M3[7];
    M2_1[3] = M3[8];
    det1 = ctm_det_2x2_matric(M2_1);

    M2_2[0] = M3[1];
    M2_2[1] = M3[2];
    M2_2[2] = M3[4];
    M2_2[3] = M3[5];
    det2 = ctm_det_2x2_matric(M2_2);

    return ((M3[0] * det0) - (M3[3] * det1) + (M3[6] * det2));
}

/**
*  ctm_inverse_3x3_matrix
*
*  get the inverse of 3x3 matrix
*
*  note: the 3x3 matrix will be represented in a vector with 9 elements.
*  M00 = V0, M01 = V1, M02 = V2, M10 = V3, M11 = V4, M12 = V5, M20 = V6, M21
*  = V7, M22 = V8.
*/
int64_t ctm_inverse_3x3_matrix(int64_t * M3, int64_t * M3_out)
{
    int64_t M2[4];
    int64_t det_M3 = 0;
    int64_t det[9];
    int i = 0;

    M2[0] = M3[4];
    M2[1] = M3[5];
    M2[2] = M3[7];
    M2[3] = M3[8];
    det[0] = ctm_det_2x2_matric(M2);

    M2[0] = M3[2];
    M2[1] = M3[1];
    M2[2] = M3[8];
    M2[3] = M3[7];
    det[1] = ctm_det_2x2_matric(M2);

    M2[0] = M3[1];
```

```
M2[1] = M3[2];
M2[2] = M3[4];
M2[3] = M3[5];
det[2] = ctm_det_2x2_matric(M2);

M2[0] = M3[5];
M2[1] = M3[3];
M2[2] = M3[8];
M2[3] = M3[6];
det[3] = ctm_det_2x2_matric(M2);

M2[0] = M3[0];
M2[1] = M3[2];
M2[2] = M3[6];
M2[3] = M3[8];
det[4] = ctm_det_2x2_matric(M2);

M2[0] = M3[2];
M2[1] = M3[0];
M2[2] = M3[5];
M2[3] = M3[3];
det[5] = ctm_det_2x2_matric(M2);

M2[0] = M3[3];
M2[1] = M3[4];
M2[2] = M3[6];
M2[3] = M3[7];
det[6] = ctm_det_2x2_matric(M2);

M2[0] = M3[1];
M2[1] = M3[0];
M2[2] = M3[7];
M2[3] = M3[6];
det[7] = ctm_det_2x2_matric(M2);

M2[0] = M3[0];
M2[1] = M3[1];
M2[2] = M3[3];
M2[3] = M3[4];
det[8] = ctm_det_2x2_matric(M2);

for (i = 0; i < 9; i++)
    M3_out[i] = det[i];
```

```
  det_M3 = ctm_det_3x3_matric(M3);

  return det_M3;
}

/**
*  ctm_func1
*
*  ctm interim function #1
*
*/
static int64_t ctm_func1(int64_t chrom1, int64_t chrom2)
{
  return ctm_div64((chrom1 * MULTIPLIER_MATRIX), chrom2);
}

/**
*  ctm_func2
*
*  ctm interim function #2
*
*/
static int64_t ctm_func2(int64_t chrom1, int64_t chrom2)
{
  return ctm_div64((MULTIPLIER_CHROM - chrom1 - chrom2) *
         MULTIPLIER_MATRIX, chrom2);
}

/**
*  ctm_func3
*
*  ctm interim function #3
*
*/
static int64_t ctm_func3(int64_t M3_1, int64_t M3_2, int64_t M3_3,
         int64_t chrom1, int64_t chrom2)
{
  return ctm_div64(M3_1 * chrom1,
         chrom2) + M3_2 + ctm_div64(M3_3 * (MULTIPLIER_CHROM -
                    chrom1 - chrom2),
                  chrom2);
}
```

```
/**
*  ctm_func4
*
*  ctm interim function #4
*
*/
static int64_t ctm_func4(int64_t Y1, int64_t chrom1, int64_t chrom2,
          int64_t det1)
{
  return ctm_div64(MULTIPLIER_MATRIX1 * Y1 * chrom1, chrom2 * det1);
}

/**
*  ctm_func5
*
*  ctm interim function #5
*
*/
static int64_t ctm_func5(int64_t Y1, int64_t det1)
{
  return ctm_div64(MULTIPLIER_MATRIX1 * Y1, det1);
}

/**
*  ctm_func6
*
*  ctm interim function #6
*
*/
static int64_t ctm_func6(int64_t Y1, int64_t chrom1, int64_t chrom2,
          int64_t det1)
{
  return ctm_div64(MULTIPLIER_MATRIX1 * Y1 * (MULTIPLIER_CHROM - chrom1 -
                chrom2), chrom2 * det1);
}

/**
*  ctm_XYZ
*
*  Get the transformation matrix from the input color space to CIE XYZ color
*  space.
*
```

```
*  note: the input parameters are the chromaticity values. They are 10000
*  times the actual values.
*  xr = chrom[0], yr = chrom[1], xg = chrom[2], yg = chrom[3], xb =
*  chrom[4], yb = chrom[5], xw = chrom[6], yw = chrom[7].
*
*  See display SAS for the detailed algorithm.
*
*/
static void ctm_XYZ(int *chrom, int64_t * M_ctm)
{
  int64_t M3_in[9];
  int64_t M3_out[9];
  int64_t det_M3 = 0;
  int64_t Y[3];
  int i = 0;

  /*
   * Get the matrix to convert from RGB space to XYZ space.
   *
   */

  for (i = 0; i < 3; i++) {
    M3_in[i] = ctm_func1(chrom[2 * i], chrom[(2 * i) + 1]);
    M3_in[i + 3] = MULTIPLIER_MATRIX;
    M3_in[i + 6] = ctm_func2(chrom[2 * i], chrom[(2 * i) + 1]);
  }

  det_M3 = ctm_inverse_3x3_matrix(M3_in, M3_out);
  det_M3 = ctm_div64(det_M3, MULTIPLIER_MATRIX);

  for (i = 0; i < 3; i++)
    Y[i] = ctm_func3(M3_out[i * 3], M3_out[(i * 3) + 1],
          M3_out[(i * 3) + 2], chrom[6], chrom[7]);

  for (i = 0; i < 3; i++) {
    M_ctm[i] =
      ctm_func4(Y[i], chrom[i * 2], chrom[(i * 2) + 1], det_M3);
    M_ctm[i + 3] = ctm_func5(Y[i], det_M3);
    M_ctm[i + 6] =
      ctm_func6(Y[i], chrom[i * 2], chrom[(i * 2) + 1], det_M3);
  }

  for (i = 0; i < 9; i++) {
```

```
        if (M_ctm[i] > 0)
            M_ctm[i] = ctm_div64(M_ctm[i] + 5, 10);
        else
            M_ctm[i] = ctm_div64(M_ctm[i] - 5, 10);
    }
}

/**
*  ctm_to_16bit_sign_exp_mantissa_register_value
*
*  Convert the 64 bit integer to a 16 bit sign-exponent-mantissa format
*  format {15, 14:12, 11:3, 2:0} - lower bits are reserved
*
* very unoptimized but left to show example
*
*/
static void ctm_to_16bit_sign_exp_mantissa_register_value(int64_t ctm, uint16_t * reg_val)
{
    uint64_t temp_N = (uint64_t) - 1;
    uint64_t temp64 = 0;
    uint32_t temp_32;
    bool sign = false;
    uint16_t remain = 0;
    uint8_t integer = 0;
    uint32_t temp;

    uint16_t mantissa_bits = 0;
    uint16_t exp_bits = 0;
    uint16_t truncated;


    /*
     * Convert the signed number to absolute value.
     *
     */
    if (ctm < 0) {
        sign = true;
        temp64 = temp_N - ((uint64_t) ctm) + 1;
        temp_32 = (uint32_t) temp64;
    } else {
        temp64 = (uint64_t) ctm;
        temp_32 = (uint32_t) temp64;
    }
```

```
  /*
   * Convert the absolute value to register value.
   *
   */
  integer = temp_32 / 1024;
  remain = temp_32 % 1024;

  *reg_val = 0;
  remain = (remain * 1024) / 1024;

  // per bspec + BDW+ excluding CHT uses 16 bit values - 15 (sign) 14:12 (exp) 11:3 (mantissa)
2:0 (reserved)
  temp = remain;
  while (temp)  {
     temp >>= 1;
     mantissa_bits++;
   }

  if (mantissa_bits ) {
     exp_bits = 12; // our format has the exp starting at but 12.
     while (mantissa_bits > 8) {
        truncated |= remain & 1;
        mantissa_bits--;
        exp_bits ++;
      }

     while (mantissa_bits < 8) {
       remain <<= 1;
       mantissa_bits++;
       exp_bits--;
     }
  }

  if (exp_bits + truncated > 15) {
     exp_bits = 15;
     remain = 0;
   } else {
      if (mantissa_bits) {
        // bias the exponent
        exp_bits += 14;
        // only 3 bits for exponent
        exp_bits = (exp_bits & ~(0xFFF8));
```

```
        }
    }

  if (sign)
    *reg_val = ((1 << 15) | exp_bits << 12 | (remain & ((1u << 12) - 1)));
  else
    *reg_val = (exp_bits << 12 | (remain & ((1u << 12) - 1)));

  // lower 3 bits are reserved
  *reg_val = (*reg_val & ~(0x07));

  if (integer > 1) {
    fprintf(stderr, "Invalid parameters\n");
    exit(0);
    }
}


/**
*  ctm_print
*
*  print color matrix coefficients register
*
*  note: the 3x3 matrix will be represented in a vector with 9 elements.
*  M00 = V0, M01 = V1, M02 = V2, M10 = V3, M11 = V4, M12 = V5, M20 = V6, M21
*  = V7, M22 = V8.
*
*/
void ctm_print(int64_t * ctm)
{
  uint16_t reg_val1 = 0, reg_val2 = 0, reg_val3 = 0;
  int32_t  i = 0;

  fprintf (stderr, "Matrix Values - Raw\n");
  for (i = 0; i < 9; i +=3) {
     fprintf (stderr, "[%4.4" PRId64 "|%4.4" PRId64 "|%4.4" PRId64 "]\n",
        ctm[i], ctm[i+1], ctm[i+2]);
  }

                    fprintf (stderr, "Matrix Values - SKL bspec format
{15(sign):14-12(exponent):11-3(mantissa):2-0(reserved)}\n");
  for (i = 0; i < 9; i += 3) {
     ctm_to_16bit_sign_exp_mantissa_register_value(ctm[i], &reg_val1);
```

```
        ctm_to_16bit_sign_exp_mantissa_register_value(ctm[i + 1], &reg_val2);
        ctm_to_16bit_sign_exp_mantissa_register_value(ctm[i + 2], &reg_val3);

        fprintf (stderr, "[%4.4x|%4.4x|%4.4x]\n",
                reg_val1, reg_val2, reg_val3);
    }
}

/**
*  ctm_operation
*
*  Program DC register to perform ctm.
*
*  note: the input parameters are the chromaticity values. They are 10000
*  times the actual values.
*  xr = chrom[0], yr = chrom[1], xg = chrom[2], yg = chrom[3], xb =
*  chrom[4], yb = chrom[5], xw = chrom[6], yw = chrom[7].
*
*  chrom1 represents the input color space; chrom2 represents the output
*  color space.
*
*  See display SAS for the detailed algorithm.
*
*/
void ctm(int *chrom1, int *chrom2)
{
    int64_t M3_out[9];
    int64_t det_M3 = 0;
    int64_t ctm1[9];
    int64_t ctm2[9];
    int64_t ctm2_inv[9];
    int64_t ctm[9];
    int32_t i = 0;

    /*
     * Get the matrix to convert from RGB space to XYZ space.
     *
     */

    ctm_XYZ(chrom1, ctm1);
    ctm_XYZ(chrom2, ctm2);

    det_M3 = ctm_inverse_3x3_matrix(ctm2, M3_out);
```

```
    det_M3 = ctm_div64(det_M3, MULTIPLIER_MATRIX2);

    for (i = 0; i < 9; i++) {
        ctm2_inv[i] =
            ctm_div64(M3_out[i] * MULTIPLIER_MATRIX1 * 1000, det_M3);

        if (ctm2_inv[i] > 0)
            ctm2_inv[i] = ctm_div64(ctm2_inv[i] + 50, 100);
        else
            ctm2_inv[i] = ctm_div64(ctm2_inv[i] - 50, 100);
    }

    ctm_matrix_mult(ctm1, ctm2_inv, ctm);

    for (i = 0; i < 9; i++) {
        if (ctm[i] > 0)
            ctm[i] = ctm_div64(ctm[i] + 50000, 100000);
        else
            ctm[i] = ctm_div64(ctm[i] - 50000, 100000);
    }

    ctm_print(ctm);
}

void usage()
{
    fprintf(stderr, "usage: source_chromaticity dest_chromaticity\n");
    exit(0);
}

void parse_cmdline(char *argv[], int **src_chrom, int **dst_chrom)
{
    // src param
    switch (atoi(argv[1])){
    case SRGB:
        *src_chrom = &sRGB[0];
        break;
    default:
        fprintf (stderr, "ERROR: undefined Source Chromaticity\n");
        exit(0);
    }

    switch (atoi(argv[2])) {
```

```
    case bt_2020:
        *dst_chrom = &BT_2020[0];
        break;
    default:
        fprintf (stderr, "ERROR: undefined Destination Chromaticity\n");
        exit(0);
    }

    if (!src_chrom || !dst_chrom) {
        fprintf (stderr, "ERROR: Source or Destination Chromaticity Undefined!\n");
        exit(0);
    }
}

int main(int argc, char *argv[])
{
    int *src_chrom = NULL;
    int *dst_chrom = NULL;

    if( argc < 3 )
        usage();

    parse_cmdline(argv, &src_chrom, &dst_chrom);
    ctm(src_chrom, dst_chrom);
    return 0;
}
```

One dimensional Lookup table manipulation

The following code can create, convert, combine one dimensional LUTs. These generic algorithms use double precision floating point operations to maximize accuracy. User mode code can use these routines for various LUT manipulation needs. For example an ICC profile may specify 256 point degamma LUT and we need to convert to 65 points while applying on CHT platform.

```
#include <stdint.h>

#define APPROXIMATE_FACTOR              0.5

typedef enum {
    DRM_COLOR_COMPONENT_RED = 0,
    DRM_COLOR_COMPONENT_GREEN,
```

```
  DRM_COLOR_COMPONENT_BLUE
}drm_color_component;

typedef struct {
    uint32_t r32;          // Data is in 16.16 fixed point format
    uint32_t g32;          // Data is in 16.16 fixed point format
    uint32_t b32;          // Data is in 16.16 fixed point format
}drm_r32g32b32;


typedef struct {
    uint32_t num_samples;
    uint32_t max_val;
    drm_r32g32b32 *lut_data;
}drm_one_d_lut;


void generate_srgb_gamma_lut(drm_one_d_lut *lut)
{
    double normalized_input, output;
    double gamma = 2.4;
    double k0 = 0.04045;
    double phi = 12.92;
    double threshold = k0 / phi;

    if (NULL != lut && NULL != lut->lut_data) {
        for (uint32_t i = 0; i < lut->num_samples; i++) {
            normalized_input = (double)i / (double)(lut->num_samples - 1);

            if (normalized_input < threshold)
                output = lut->max_val * phi * normalized_input +
                    APPROXIMATE_FACTOR;
            else
                output = lut->max_val * (1.055 *
                    pow(normalized_input, 1.0 / gamma) - 0.055)
                        + APPROXIMATE_FACTOR;

            if (output > lut->max_val)
                output = lut->max_val;

            lut->lut_data[i].r32 = lut->lut_data[i].g32 = lut->lut_data[i].b32 = output;
        }
    }
```

```
}


void generate_srgb_degamma_lut(drm_one_d_lut *lut)
{
    uint32_t out_val;
    double gamma = 2.4;
    double k0 = 0.04045;
    double phi = 12.92;

    if (NULL != lut && NULL != lut->lut_data) {
        for (uint32_t i = 0; i < lut->num_samples; i++) {
            double normalized_input = (double)i / (double)(lut->num_samples - 1);

            if (normalized_input <= k0)
                out_val = ((double)lut->max_val * normalized_input / phi) +
                    APPROXIMATE_FACTOR;
            else
                out_val = (double)lut->max_val * pow((normalized_input + 0.055) /
                    1.055, gamma) + APPROXIMATE_FACTOR;

            if (out_val > lut->max_val)
                out_val = lut->max_val;

            lut->lut_data[i].r32 = lut->lut_data[i].g32 = lut->lut_data[i].b32 = out_val;
        }
    }
}


void generate_gamma_lut(drm_one_d_lut *lut, double gamma)
{
    uint32_t out_val;

    if (NULL != lut && NULL != lut->lut_data) {
        for (uint32_t i = 0; i < lut->num_samples; i++) {
            double normalized_input = (double)i / (double)(lut->num_samples - 1);

            out_val = lut->max_val * pow(normalized_input, gamma) +
                APPROXIMATE_FACTOR;

            if (out_val > lut->max_val)
                out_val = lut->max_val;
```

```
                lut->lut_data[i].r32 = lut->lut_data[i].g32 = lut->lut_data[i].b32 = out_val;
        }
    }
}

double apply_lut_on_component(drm_one_d_lut *lut, double input, drm_color_component component)
{
    double out_val;
    double d_index = (double)input * (double)(lut->num_samples - 1);
    uint32_t i_index = d_index;
    double diff_index = d_index - (double)i_index;

    if (i_index < (lut->num_samples - 1)) {
        switch (component) {
            case DRM_COLOR_COMPONENT_RED:
            out_val = (double)lut->lut_data[i_index].r32 *
                (1.0 - diff_index) + (double)lut->lut_data[i_index + 1].r32 * diff_index;
            break;

            case DRM_COLOR_COMPONENT_GREEN:
            out_val = (double)lut->lut_data[i_index].g32 * (1.0 - diff_index) +
                (double)lut->lut_data[i_index + 1].g32 * diff_index;
            break;

           case DRM_COLOR_COMPONENT_BLUE:
           out_val = (double)lut->lut_data[i_index].b32 * (1.0 - diff_index) +
                (double)lut->lut_data[i_index + 1].b32 * diff_index;
            break;
        }
    } else {
        switch (component) {
            case DRM_COLOR_COMPONENT_RED:
            out_val = (double)lut->lut_data[i_index].r32;
            break;
            case DRM_COLOR_COMPONENT_GREEN:
            out_val = (double)lut->lut_data[i_index].g32;
            break;
            case DRM_COLOR_COMPONENT_BLUE:
            out_val = (double)lut->lut_data[i_index].b32;
            break;
        }
```

```
    }

    out_val /= (double)lut->max_val;
    return out_val;
}


void combine_luts(drm_one_d_lut *lut1, drm_one_d_lut *lut2, drm_one_d_lut *lut_combined)
{
    double inp_val, out_val, tmp_val;

    for (uint32_t i = 0; i < lut_combined->num_samples; i++) {
        inp_val = (double)i / (double)(lut_combined->num_samples - 1);

        tmp_val = apply_lut_on_component(lut1, inp_val,
                DRM_COLOR_COMPONENT_RED);
        out_val = apply_lut_on_component(lut2, tmp_val,
                DRM_COLOR_COMPONENT_RED);
         out_val = out_val * (double)lut_combined->max_val + APPROXIMATE_FACTOR;

         if (out_val > lut_combined->max_val) out_val = lut_combined->max_val;
         lut_combined->lut_data[i].r32 = out_val;

        tmp_val = apply_lut_on_component(lut1, inp_val,
                DRM_COLOR_COMPONENT_GREEN);
        out_val = apply_lut_on_component(lut2, tmp_val,
                DRM_COLOR_COMPONENT_GREEN);

        out_val = out_val * (double)lut_combined->max_val + APPROXIMATE_FACTOR;
        if (out_val > lut_combined->max_val) out_val = lut_combined->max_val;
        lut_combined->lut_data[i].g32 = out_val;

        tmp_val = apply_lut_on_component(lut1, inp_val,
                DRM_COLOR_COMPONENT_BLUE);
        out_val = apply_lut_on_component(lut2, tmp_val,
                DRM_COLOR_COMPONENT_BLUE);
         out_val = out_val * (double)lut_combined->max_val + APPROXIMATE_FACTOR;

        if (out_val > lut_combined->max_val) out_val = lut_combined->max_val;
         lut_combined->lut_data[i].b32 = out_val;
    }
}
```

```
double compare_luts(drm_one_d_lut *lut1, drm_one_d_lut *lut2)
{
    double err = 0.0;
    double val1, val2;

    if (lut1->num_samples != lut2->num_samples)
            return 100.0;

    for (uint32_t i = 0; i < i < lut1->num_samples; i++) {
            val1 = (double)lut1->lut_data[i].r32 / (double)lut1->max_val;
            val2 = (double)lut2->lut_data[i].r32 / (double)lut2->max_val;
            err += abs(val1 - val2);

            val1 = (double)lut1->lut_data[i].g32 / (double)lut1->max_val;
            val2 = (double)lut2->lut_data[i].g32 / (double)lut2->max_val;
            err += abs(val1 - val2);

            val1 = (double)lut1->lut_data[i].b32 / (double)lut1->max_val;
            val2 = (double)lut2->lut_data[i].b32 / (double)lut2->max_val;
            err += abs(val1 - val2);
    }

    err /= (double)(lut1->num_samples * 3);
    err *= 100.0;
    return err;
}

void convert_lut(drm_one_d_lut *lut_src, drm_one_d_lut *lut_dst)
{
    double inp_val, out_val;

    for (uint32_t i = 0; i < lut_dst->num_samples; i++) {
            inp_val = (double)i / (double)(lut_dst->num_samples - 1);

           out_val = apply_lut_on_component(lut_src, inp_val,
                    DRM_COLOR_COMPONENT_RED);
            out_val = out_val * (double)lut_dst->max_val + APPROXIMATE_FACTOR;
            if (out_val > lut_dst->max_val) out_val = lut_dst->max_val;
            lut_dst->lut_data[i].r32 = out_val;

           out_val = apply_lut_on_component(lut_src, inp_val,
                    DRM_COLOR_COMPONENT_GREEN);
            out_val = out_val * (double)lut_dst->max_val + APPROXIMATE_FACTOR;
```

```
        if (out_val > lut_dst->max_val) out_val = lut_dst->max_val;
                lut_dst->lut_data[i].g32 = out_val;

        out_val = apply_lut_on_component(lut_src, inp_val,
                DRM_COLOR_COMPONENT_BLUE);
        out_val = out_val * (double)lut_dst->max_val + APPROXIMATE_FACTOR;
        if (out_val > lut_dst->max_val) out_val = lut_dst->max_val;
                lut_dst->lut_data[i].b32 = out_val;
    }
}
```

```
/*
 * One dimensional Lookup table manipulation examples
 * The following examples show a few typical LUT manipulation use cases.
 */

void main(void)
{
   // Create sRGB Gamma table with 512 samples. Each sample is 10 bit precise
   drm_one_d_lut srgb_gamma;
   srgb_gamma.num_samples = 257;
   srgb_gamma.max_val = 1023;
   srgb_gamma.lut_data = (drm_r32g32b32 *)
                  malloc(srgb_gamma.num_samples * sizeof(drm_r32g32b32));

   generate_srgb_gamma_lut(&srgb_gamma);

   // Create sRGB DeGamma table with 512 samples. Each sample is 10 bit precise
   drm_one_d_lut srgb_degamma;
   srgb_degamma.num_samples = 65;
   srgb_degamma.max_val = 16383;
   srgb_degamma.lut_data = (drm_r32g32b32 *)
         malloc(srgb_degamma.num_samples * sizeof(drm_r32g32b32));

   generate_srgb_degamma_lut(&srgb_degamma);

   // Create a combined LUT of different number of samples and precision
   drm_one_d_lut combined_lut;
   combined_lut.num_samples = 256;
   combined_lut.max_val = 255;
   combined_lut.lut_data = (drm_r32g32b32 *)
         malloc(combined_lut.num_samples * sizeof(drm_r32g32b32));

   combine_luts(&srgb_gamma, &srgb_degamma, &combined_lut);

   //Convert LUT number of samples and precisions
   drm_one_d_lut converted_lut;
   converted_lut.num_samples = 1025;
   converted_lut.max_val = 65535;
   converted_lut.lut_data = (drm_r32g32b32 *)
         malloc(converted_lut.num_samples * sizeof(drm_r32g32b32));

   convert_lut(&srgb_gamma, &converted_lut);
```

```
    // Create a non-SRGB gamma LUT
    drm_one_d_lut non_srgb_lut;
    non_srgb_lut.num_samples = 1024;
    non_srgb_lut.max_val = 1023;
    non_srgb_lut.lut_data = (drm_r32g32b32 *)
            malloc(non_srgb_lut.num_samples * sizeof(drm_r32g32b32));

    generate_gamma_lut(&non_srgb_lut, 1.0);

    free(srgb_gamma.lut_data);
    free(srgb_degamma.lut_data);
    free(combined_lut.lut_data);
    free(converted_lut.lut_data);
    free(non_srgb_lut.lut_data);
}
```