# Linux DRM Basic Concepts and Features

**A wedge**

In the last article, we mentioned that DRM is a graphics rendering architecture under linux, which is used to manage display output and buffer allocation.

The application can directly manipulate the ioctl of drm or use the interface provided by framebuffer to perform display related operations.

Later, everyone thought this was too low, so let's just package it into a library. So libdrm was born, it is a library, which provides a series of friendly control packages, so that we can control the display more conveniently.

To understand how DRM outputs the user's drawing to the display screen, we cannot avoid understanding these concepts:

Framebuffer

CRTC

Encoder

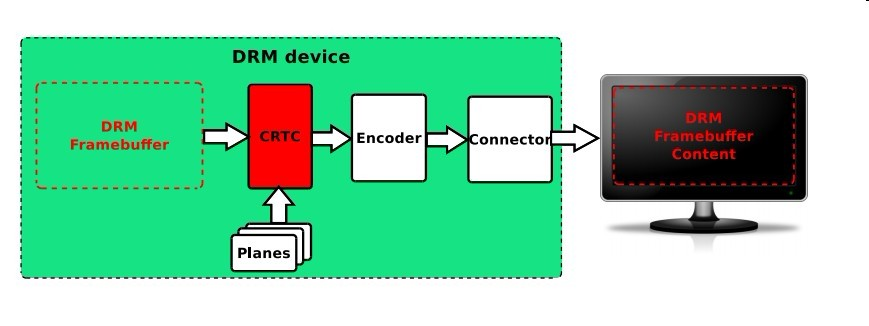
Connector

Display Device(LCD)

Then we will introduce some features of DRM.

Finally, I will introduce the Component Framework on which the DRM Driver of the RK platform depends in less space.

**2. Basic concepts involved in DRM**



Let's focus on the five blocks in the green box together.

**2.1 DRM Framebuffer**

It is a memory area, I understand it as a canvas, which can be accessed by both the driver and the application layer. Before painting, it needs to be formatted. We need to set whether you want to paint oil painting or Chinese painting (**color mode, such as RGB24, YUV, etc**.), and how large the canvas needs to be **(resolution).**

What is a canvas in graphics?

In computer science and visualization, a canvas is a container that holds various drawing elements (lines, shapes, text, frames containing others elements, etc.). It takes its name from the canvas used in visual arts.

**2.2 CRTC**

Literally translated as cathode camera context. I would say what it does is read the pixel data of the current scan buffer and generate video mode timing signals from it with the help of a PLL circuit. You might just start muttering and complaining about what the hell. Simple, the way of speaking is simple, in simple terms he is the context of the display output, I understand it as a scanner. It connects the Framebuffer address internally and the Encoder externally.

**It will scan the content on your canvas (Framebuffer), overlay the content of Planes, and pass it to the Encoder.**

**2.3 Planes**

Literally translated as plane**. It is the same memory address as the framebuffer.** What does it do? Imagine a scene where the author is writing an article while watching an action blockbuster very inattentively. Every frame of an action blockbuster has a lot of changes and needs to be updated in full. It is impossible to squeeze a word out of writing an article for a long time, and basically no update is required. The author's naughty pulls the use of graphics cards to two extremes. **One is the Video Mode with full-frame high-speed update, and the other is the Graphics Mode with a small range of text interaction. Now it's the turn of Planes, which provides a high-speed channel for Video refresh, making Video a separate layer, which can be superimposed on or below Graphic, and has functions such as zooming.**

Looking at the picture at the beginning of this section**, there can be multiple Planes, so the last picture (image) scanned by the scanner (CRTC) is actually a combination of Framebuffer and Planes (Blending).**

**2.4 Encoder**

Literally translated as encoder**. Its function is to encode (convert) the pixels of the memory into the signals required by the display.**

You can think of its role as if you want to display your painting on different Display Devices all over the world, **you naturally need to convert it into different electrical signals, such as DVID, VGA, YPbPr, CVBS, Mipi, eDP, etc.**

**So we need such an Encoder for signal conversion**. The interaction between it and CRTC is what we call ModeSetting, which includes the color mode provided earlier, as well as timing (Timing).

**2.5 Connector**

Literally translated as connector. Connector usually corresponds to a physical connector (VGA, DVI, FPD-Link, HDMI, DisplayPort, S-Video …) It will connect a physical display output device (monitor, laptop panel, …).

Information related to the currently physically connected output device (such as connection status, EDID data, DPMS status or supported video modes) is also stored within the Connector.

**3. Features in DRM**

3.1 GEM

Graphics Execution Manager

Graphics APIs such as OpenGL continue to become more complex as the video memory size increases. Reinitializing the graphics card state on context switches can be quite expensive. Also, now the Linux desktop needs an optimal way to share the offscreen cache with the composition manager.

These requirements led to the development of a new method to manage the graphics buffer area in the kernel, the GEM.

3.1.1 GEM memory management

GEM provides a set of APIs with explicit memory management primitives. With GEM, user-space programs can create, process, and destroy memory objects in GPU video memory. These objects are collectively referred to as "GEM objects", and from a user space perspective, they always exist, so they don't need to be reloaded when the program regains the GPU. **When a user-space program needs a lot of video memory space (to store framebuffer, texture data, or other data required by the GPU), it will use the GEM API to allocate memory for DRM Driver requests. This API also provides functions for filling and releasing buffers. And can release the associated memory when the user space process (due to accidental or normal termination) closes the DRM device descriptor.**

GEM also provides methods for two or more user-space processes to access the same DRM device (ie, share the same GEM object). GEM provides a method, flink (in fact, this method is very unsafe), to get the GEM name from the GEM handle. A process passes the GEM name (32bit integer) to another process via IPC. The process that receives the GEM name can then obtain a local GEM handle to point to the original GEM object.

So if a malicious third-party application knows the GEM name, it can access and modify the contents of the GEM object. So later by introducing DMA-BUF mechanism to overcome this defect.

3.1.2 GEM memory synchronization

**Besides managing the video storage space, another important task of the video memory manager is to deal with the problem of memory synchronization between GPU and CPU.**

Because the current memory architecture is very complex, it usually involves multi-level caches of system memory and video memory. Therefore, the video memory manager is also responsible for handling cache coherency in order to ensure the coherence of the shared data between the GPU and the CPU. This means that the video memory management system is highly dependent on the GPU hardware and memory architecture and therefore driver specific.

GEM defines "memory domains" for memory synchronization, and these memory domains are not related to the GPU, they are designed with the UMA memory architecture, making it unsuitable for other memory architectures, such as those with separate VRAMs. Therefore, externally, the DRM driver will expose a unified GEM API to the user space, and internally, the driver will implement different memory managers that are more suitable for specific hardware and memory architectures.

3.2 KMS

Kernel Mode Setting

**In order to work properly, the video card and graphics adapter must be responsible for setting some modes, including screen resolution, color depth, refresh rate, etc., within the parameters supported by themselves or the extended screen. This action is mode-setting, and it usually requires access to graphics hardware -- the ability to manipulate the video card's registers.**

**Mode-setting is performed before starting to use the framebuffer, or when the mode is changed by the application or the user.**

3.2.1 UMS

Initially, userspaces that want to use the graphics framebuffer also need to be responsible for mode-setting, so they need access to the video hardware. **A good example is the Display Server on Unix-like systems such as the X Server, whose mode-settings are placed in the respective DDX driver for each specific graphics card.**

This approach, called UMS (User space Mode-Setting), has several serious problems. It not only breaks the isolation between hardware and programs established by the operating system, but also allows the hardware to produce inconsistent states when multiple processes perform mode-setting at the same time.

So in order to avoid these conflicts, X Server becomes the only user space program for real-time mode-setting, and other user space programs rely on X Server for mode-setting.

**Initially mode-setting is placed in the X Server startup process, but later it can also be mode-setting during its running process.**

But UMS has many problems.

For example, during the boot process of the Linux system, the Linux kernel must set the "minnimal text" mode for the virtual console.

**In addition, the framebuffer driver also contains the code to configure the framebuffer device mode.**

**To resolve these conflicts, some Display Servers (such as the X Server) save the mode-setting state when switching from a graphical environment to a text virtual console, and restore it when switching back.**

**This leads to new problems, such as flickering when switching, display failure or even corruption of output devices.**

3.2.2 mode-setting

**In order to solve these problems, mode-setting is put into the Kernel alone, to be precise in the DRM module.**

**Then, each process, including the X Server, can instruct the kernel to implement mode-setting operations, and the kernel will ensure consistency of operations. The mode-setting operations performed by these new kernel APIs and code added to the DRM module are called KMS .**

It has so many benefits,

1. **the first being that it gets rid of the repetitive mode-setting code from the Kernel (console, fbdev) and Userspace (X Server DDX Driver).**
2. **The second is that the graphics operating system no longer needs to care about the writing of the mode-setting part of the code.**
3. **The third is that because this single centralized mode management is provided, it becomes easier to switch between different instances of the console and X Server.**
4. **The fourth is that after the mode-setting is put into the kernel, it can be used from the boot process (this used to cause flickering problems).**
5. **In addition, because KMS is part of the kernel, it can also use many resources of the kernel space (such as interrupts). Mode recovery after suspend/resume is simpler because it is left to the kernel to manage. The kernel also makes it easier to hot-plug new display devices.**

**Since mode-setting is closely related to memory management -- framebuffer is basically a memory buffer -- it is tightly integrated with image memory management.** This is also the reason why it is placed in the DRM module instead of being a separate subsystem.

3.2.3 KMS Driver

In order not to break the backward compatibility of the DRM API, KMS provides a special feature for the DRM Driver. Any DRM Driver needs to choose whether to use the DRIVER\_MODESET flag when registering the DRM Core to indicate whether to support the KMS API.

Drivers that support KMS API are often called KMS Drivers to distinguish them from traditional DRM Drivers.

3.2.4 KMS Device Mode

The KMS is responsible for shaping and managing output devices, abstracting them into a series of hardware modules (often on the display controller's display output pipeline).

**We have introduced these modules before, CRTC, Planes, Encoder, Connector. But the introduction at that time was relatively popular. The following is the Google translated version on Wikipedia:**

**CRTCs: Each CRTC (from the CRT controller) represents the scan engine of the display controller and points to the framebuffer. The goal of the CRTC is to read the pixel data of the current scan buffer and generate video mode timing signals therefrom with the aid of a PLL circuit.**

**The number of CRTCs determines how many independent output devices the hardware can handle simultaneously, so in order to use a multi-head configuration, you need at least one CRTC per display device.**

**Two or more CRTCs can also work in clone mode, sending the same image to multiple output devices if they scan the same framebuffer.**

**Connectors:** Connectors represent where the display controller sends the video signal from the scanout operation to be displayed.

Usually, Connectors in KMS correspond to physical connectors

(VGA, DVI, FPD-Link, HDMI, DisplayPort, S-Video …) and it will connect a physical display output device (monitor, laptop panel, …).

**Information related to the currently physically connected output device (such as connection status, EDID data, DPMS status, or supported video modes) is also stored within the connector.**

**Encoders:** The display controller must encode the video mode timing signal from the CRTC using a format suitable for the intended connector. An encoder represents a block of hardware capable of performing one of these encodings. **Connectors can only receive signals from one encoder at a time, and each type of connector supports only a few encodings. There may also be other physical limitations, not every CRTC is connected to every available encoder, limiting the possible combinations of CRTC encoder connectors.**

Planes: planes are not hardware blocks, **but memory objects that contain buffers supplied to the scan engine (CRTC). The plane that holds the framebuffer is called the main plane, and each CRTC must have an associated plane because it is the source of the parameters that the CRTC determines. Parameters include, video mode - display resolution (width and height), pixel size, pixel format, refresh rate, etc.**

Four, component framework

The DRM of the RK platform also relies on the component framework .

Here is the discussion on the RK Socs DRM Driver Patch on the kernel mailing list: https://lkml.org/lkml/2014/12/2/161

At the beginning of the email we can see that the birth of the RK platform DRM Driver depends on the major changes in 15 versions.

One of the most important points mentioned is that it uses the component framework.

Since there are many devices mounted under DRM, the boot sequence may cause problems:

The driver generates a probe deferral because it waits for the preparation of another resource, resulting in an indeterminate order.

If the sub-device is not loaded properly, the main device is loaded, causing the device to fail to work.

The sub-devices may have a timing relationship with each other, and the indeterminate loading order may cause the device to work sometimes and sometimes not.

Now the kernel is compiled with multi-threading, and the order of compilation will also affect the loading order of the driver.

For the RK platform, because multiple VOPs are used, it is necessary to start the probe of the DRM Driver after all the VOPs are started, that is, to postpone the probe.

At this time, a unified management mechanism is needed to integrate all the devices and load them in a unified order. After all the components are loaded, bind them to the master.

For a more detailed article introducing components, refer to the process of component authors discussing the component framework with others:

https://patchwork.kernel.org/patch/3431851/

Our exposure to components will stop at the component part of the drm master probe. We will analyze the main logic of the component in the rockchip drm master probe with a separate section in the analysis of the drm driver code in the next chapter.

Reference article

wikipedia drm: https://en.wikipedia.org/wiki/Direct\_Rendering\_Manager

landley drm: http://www.landley.net/kdocs/htmldocs/drm.html

ubuntu drm kms: http://manpages.ubuntu.com/ manpages/zesty/en/man7/drm-kms.7.html

https://lkml.org/lkml/2014/12/2/161

https://patchwork.kernel.org/patch/3431851/

# Linux DRM architecture

Reprinted from:

http://manpages.ubuntu.com/manpages/utopic/man7/drm-kms.7.html

To paraphrase according to your own understanding:

Summary:

DRM is a graphics rendering architecture (Direct Render Manager) under [linux](https://so.csdn.net/so/search?q=linux&spm=1001.2101.3001.7020" \t "_blank) , specifically a graphics card driver architecture (how does the driver play? The function is encapsulated into standard interfaces such as open/close/ioctl, and the application calls these interfaces to drive the device. ).

As a graphics card, the most basic function is to output the user's drawing to the display screen. How does DRM implement it? Let's first take a look at the basic elements that DRM summarizes "this matter" for you:

Canvas (FrameBuffer), Drawing Field (CRTC), Output Converter (Encoder), Connector (Connector), and then to the display

**1 Canvas (FrameBuffer)**

For computers, FrameBuffer is a piece of memory that can be accessed by both the driver and the application layer. Of course, there must be a certain format before drawing. For example, I can specify what color mode (RGB24, I420, YUUV, etc.), resolution How big is it, and what parameters are there, then you have to go to the drawing site to see :p

**2 Drawing Field (CRTC)**

The abbreviation translates to the context of the negative camera tube. In DRM, CRTC means the context of the display output. First, CRTC refers to a FrameBuffer address, and an Encoder is connected to it. How do they communicate with each other? This is what the display mode (ModeSet) has to do. ModeSet includes the color mode mentioned above, and the timing of the display (timings, ModeLines, etc. all represent this meaning), etc. Usually the timing can be expressed as follows

PCLK  HFP  HBP  HSW  X\_RES  VFP  VBP  VSW  Y\_RES

Pixel Clock  Horizontal Front Retrace  Horizontal Retrace  Horizontal Sync Head  Horizontal Effective Length  Vertical Front Retrace  Vertical Back Retrace  Vertical Sync Head  Vertical Effective Length

A CRTC can be connected to multiple Encoders, what are they used for, and realize the function of copying the screen.

**3 Output converter (Encoder)**

Thinking about the CRT is complicated enough. Our graphics cards are very capable and can be connected to various devices. Obviously, different signal converters are required for the output to convert the pixels of the memory into the signals required by the display (DVID, VGA). , YPbPr , CVBS etc...)

**4 Connector (Connector)**

Not referring to the physical wire, back to DRM which is an abstract data structure representing the connected display device, from here we can get the EDID of the device, DPMS connection status, etc.

**5 Display plane (Planner)**

Hey, how come there is one more. I am also very bored, the above stuff is not enough to work? In fact, many innovations often stem from people's dissatisfaction with the real world. You need to learn words, watch movies and play games, and you can watch movies while chatting. Two concepts are opposed here, the Graphics mode for small-scale updates such as text interaction, and the Video mode for fast full-frame updates. These two modes bring the use of graphics cards to two extremes.

So the concept of Planner has played a very good role, it provides a green channel for video refresh, and sometimes it is not mixed with graphics, sometimes it is a new layer (or overlay), which can be superimposed on Graphic or Below, you can also zoom...

The document says that the Planner is also on the FrameBuffer. It doesn't matter. Here we see that the stuff to be displayed in the CRTC should be a blending.

After understanding the concept, the next article will analyze the specific data structure and interface.

Reference documentation:

http://manpages.ubuntu.com/manpages/utopic/man7/drm-kms.7.html

http://events.linuxfoundation.org/sites/events/files/lcjpcojp13\_pinchart.pdf

http://landley.net/kdocs/htmldocs/drm.html

http://events.linuxfoundation.org/sites/events/files/slides/brezillon-drm-kms.pdf

http://elinux.org/images/7/71/Elce11\_dae.pdf

# The principle and application of linux drm

Reference link

dma:https://blog.csdn.net/abc3240660/article/details/81942190#t6

Linux DRM KMS 驱动简介:https://blog.csdn.net/zhuyong006/article/details/80942777

DRM（Direct Rendering Manager:https://blog.csdn.net/hexiaolong2009/article/details/83720940

**Application scenarios**

The linux drm (Direct Rendering Manager) was originally designed as a set of display data transmission process, which is used to throw the video data collected by the camera to the display for display.

An intermediate layer libdrm is encapsulated between the drm driver and the application. The application can operate the drm driver by calling the interface provided by libdrm , which is relatively simple.

**main module**

The drm system is mainly divided into three modules: libdrm, GEM, and KMS.

**libdrm**

libdrm runs in the user space and is a bridge between the application and the kernel. Its function is to fill in the structure required by the kernel and pass it into the kernel through ioctl calls. After the kernel is filled, it returns to the application space.

**GEM**

GEM (Graphic Execution Manager) is mainly responsible for buffer operations.

**KMS**

KMS (Kernel Mode Setting) is mainly responsible for the setting of relevant parameters (including resolution, refresh rate, power state (sleep wake-up), etc.) and switching of the display screen (switching of the display buffer, multi-layer synthesis method, and each image display position of the layer).

**fundamental element**

**KMS**

**1. CRTC: A hardware module that scans the display buffer and generates timing signals, usually referred to as Display Controller;**

a) DPMS (Display Power Manage System) power state management (crtc\_funcs->dpms)

b) Converting Framebuffer to standard LCDC Timing is actually a process of refreshing a frame of image (crtc\_funs->mode\_set)

c) Frame switching, that is, in During VBlank fading, switch Framebuffer (crtc\_funcs->page\_flip)

**2. Encoder: Responsible for converting the timing timing output from CRTC to what is needed by external devices**

a) DPMS (Display Power Manage System) power state management (encoder\_funcs->dpms)

b) Package the lcdc Timing output by the VOP into the corresponding interface timing HDMI TMDS / … (encoder\_funcs->mode\_set)

The encoder is the connector between crtc and connector

**3. CONNECTOR: The connector that connects the physical display device, such as HDMI, DisplayPort, DSI bus, usually bound with the Encoder driver, and is most closely related to the display;**

a) Get and report the hotplug status of the display

b) Read and parse the EDID information of the panel

**4. PLANE: Hardware layer, some Display hardware supports multi-layer composite display, but all Display Controllers must have at least one plane;**

a) plane is the connector between ctrc and framebuffer

b) each crtc must have at least one plane

c) Plane in DRM and plane in the YUV/YCbCr graphics format we often say are completely two different concepts. The plane in the YUV graphics format refers to the arrangement of image data in memory. Generally, the Y channel occupies a continuous memory block, and the UV channel occupies another continuous memory block. We call it YUV-2plane (also called YUV 2 plane). plane), which belongs to the software level. Plane in DRM refers to a single hardware layer module used for multi-layer synthesis in Display Controller, which belongs to the hardware level.

d) With the continuous updating of software technology, the performance requirements for hardware are getting higher and higher. Under the premise of use, the requirements for power consumption are becoming more and more stringent. Originally, the GPU can handle all graphics tasks, but because its power consumption is too high, the designers decided to hand over some simple tasks to the Display Controller (such as compositing), and let the GPU focus on drawing (ie rendering) This main task reduces the burden on the GPU, thereby achieving the purpose of reducing power consumption and improving performance. Thus, Plane (hardware layer unit) was born

**5. FB: Framebuffer is the display content of a single layer, the only basic element that has nothing to do with hardware**

a) The buffer is a piece of allocated memory, which has nothing to do with hardware. Generally, the buffer is associated with the data buf of the camera through the DMA\_BUF mechanism to achieve the purpose of fast data transfer

**6. property: Any parameter you want to set can be made into property, which is the most flexible and convenient Mode setting mechanism in DRM drivers**

a) The introduction of the Atomic mechanism can reduce the maintenance workload of the upper-layer application interface. When developers have new functions to add, they do not need to add new function names and IOCTLs. They only need to add a property in the underlying driver, and then get/operate the value of the property in their own applications.

b) Property The simple summary of the structure is mainly composed of three parts: name, id and value. The id is the globally unique identifier of the property in the DRM framework.

c) The flexibility of parameter setting is enhanced. One IOCTL can set multiple properties at the same time, reducing the number of switching between userspace and kernel space, and at the same time meeting the requirements of different hardware for parameter settings to the greatest extent, improving software efficiency

**GEM**

DUMB: A buffer type that only supports continuous physical memory. It is implemented based on the general CMA API in the kernel and is mostly used in simple scenarios with small resolutions.

PRIME: Both continuous and non-continuous physical memory are supported. Based on the DMA-BUF mechanism, buffer sharing can be realized, which is mostly used in complex scenarios with large memory.

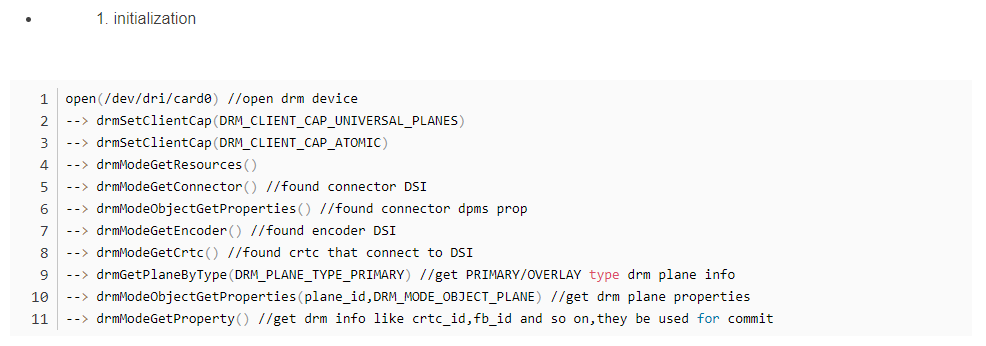
fence: buffer synchronization mechanism, implemented based on the kernel dma\_fence mechanism, used to prevent asynchronous problems in the display content

**module behavior**

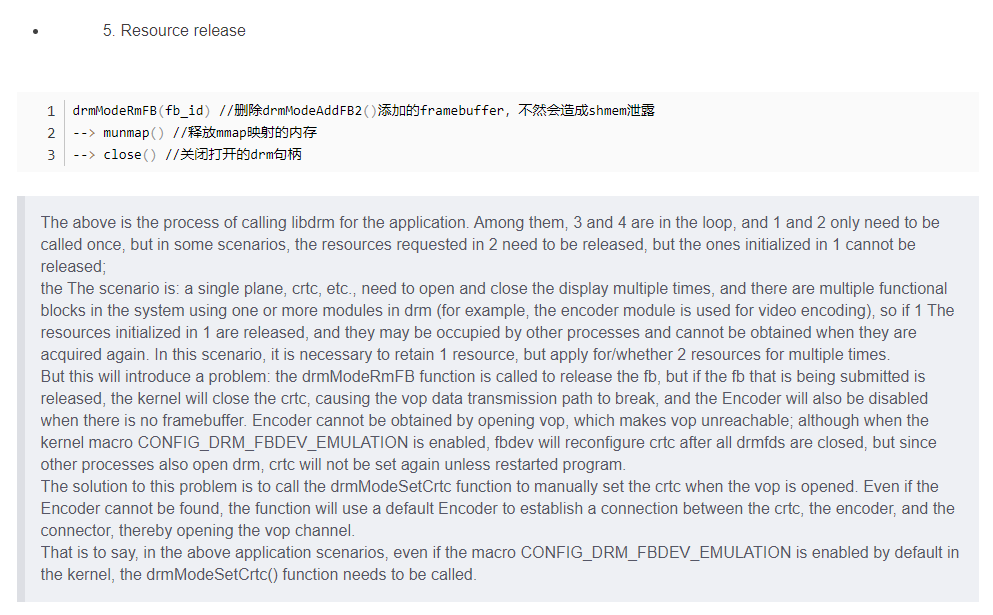
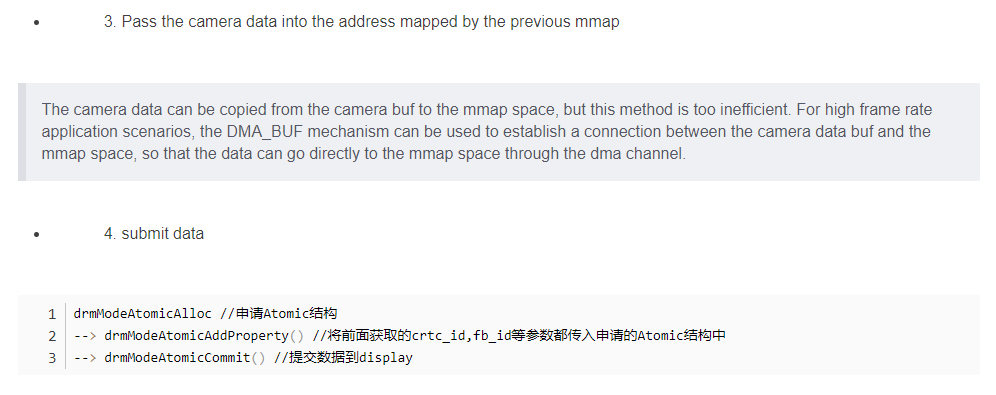
Taking the process displayed by HDMI Monitor as an example, the behavior of CRTC / Encoder / Connector under the example analysis

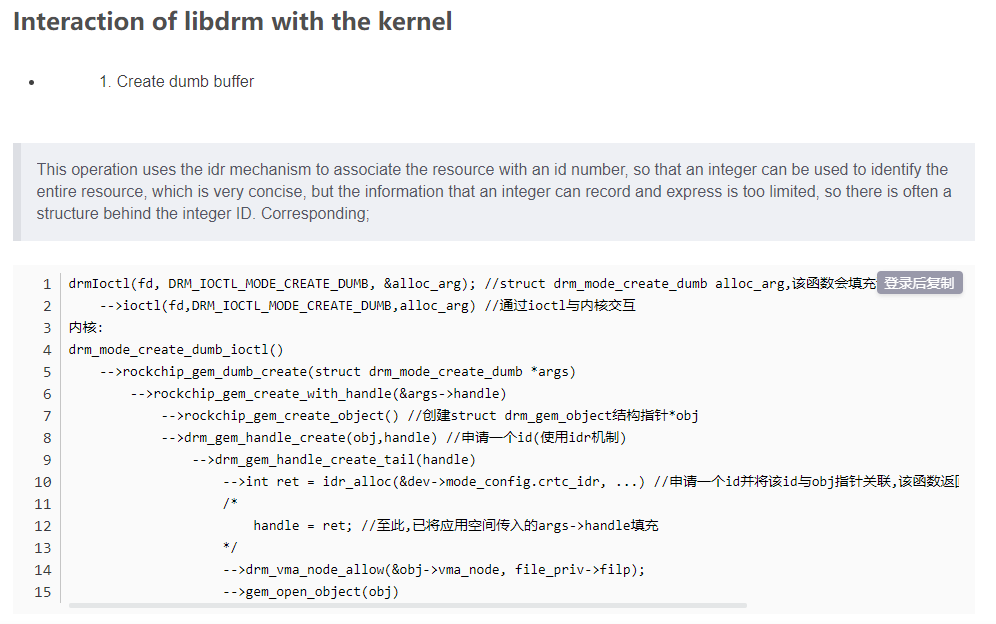
1. First, the HDMI driver detects the Plugin signal of the TV, reads the EDID signal of the TV, and obtains the resolution information of the TV (DRM Connector)
2. Userspace fills the framebuffer with the data to be displayed, and then informs the VOP device to start displaying through the libdrm interface
3. Then the VOP driver converts the data in the framebuffer into standard LCDC Timing timing (DRM CRTC)
4. At the same time, the HDMI driver aligns the LCDC timing configuration of the HDMI hardware module with the VOP output timing, and prepares to convert the input LCDC Timing into the HDMI TMDS signal (DRM Encoder) recognized by the TV.

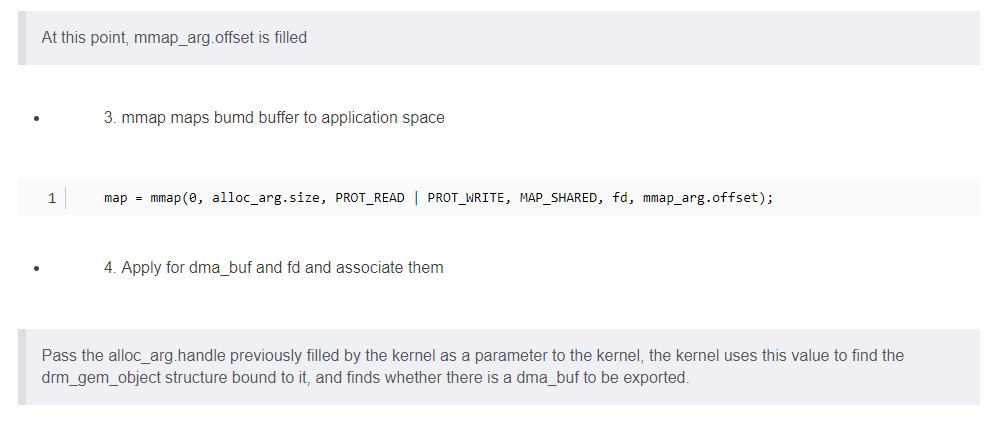
# DRM application call process

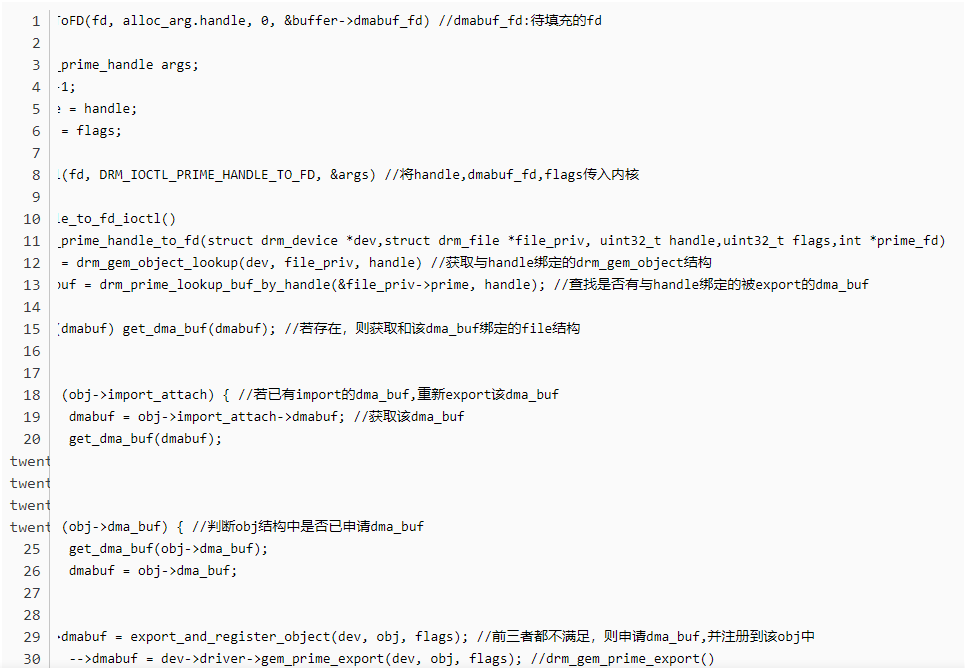


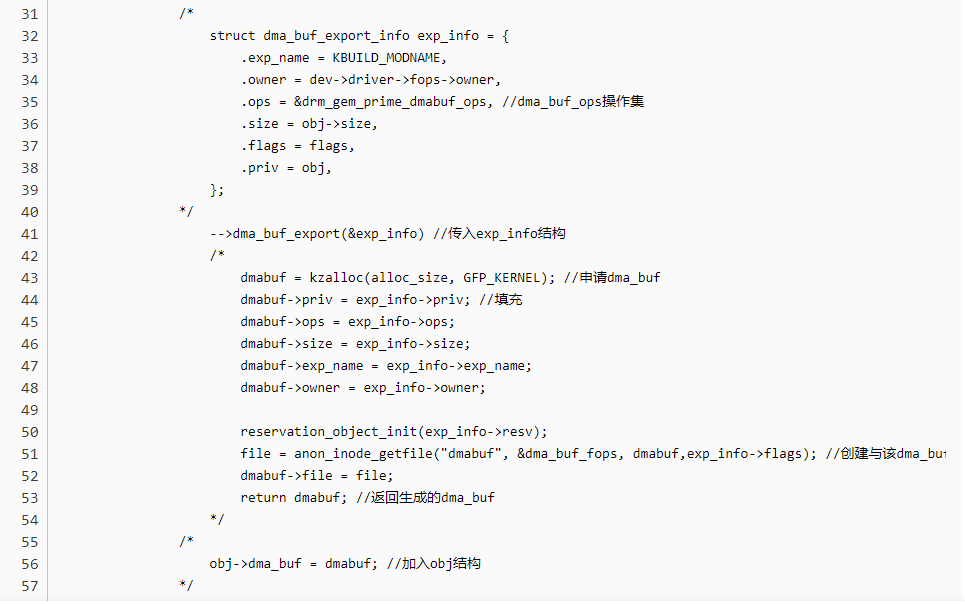
The order of obtaining crtc, encoder, connector, plane is connector, encoder, crtc, plane. Each structure of the former has the id number of the latter. After finding the connector, you can use the encoder\_id under its structure to find the one connected to yourself. The encoder module is the same until the connection path from the plane to the connector is found.

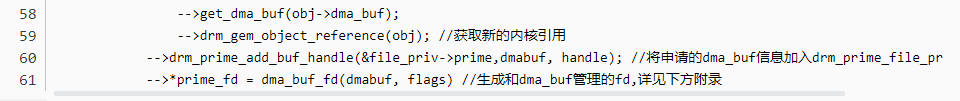


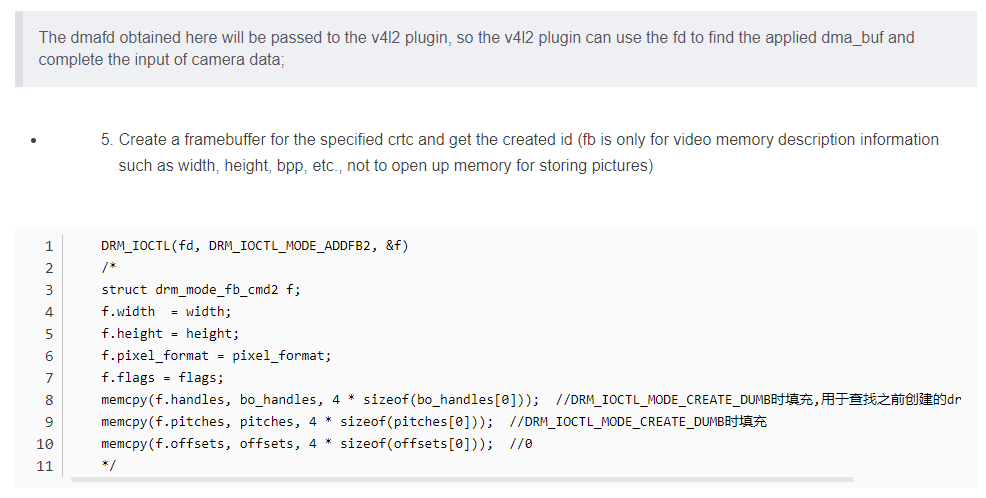


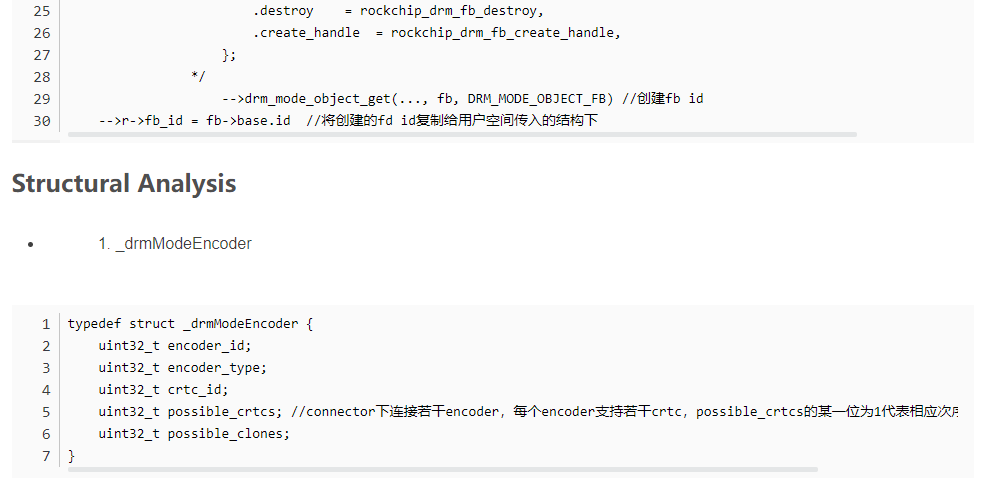




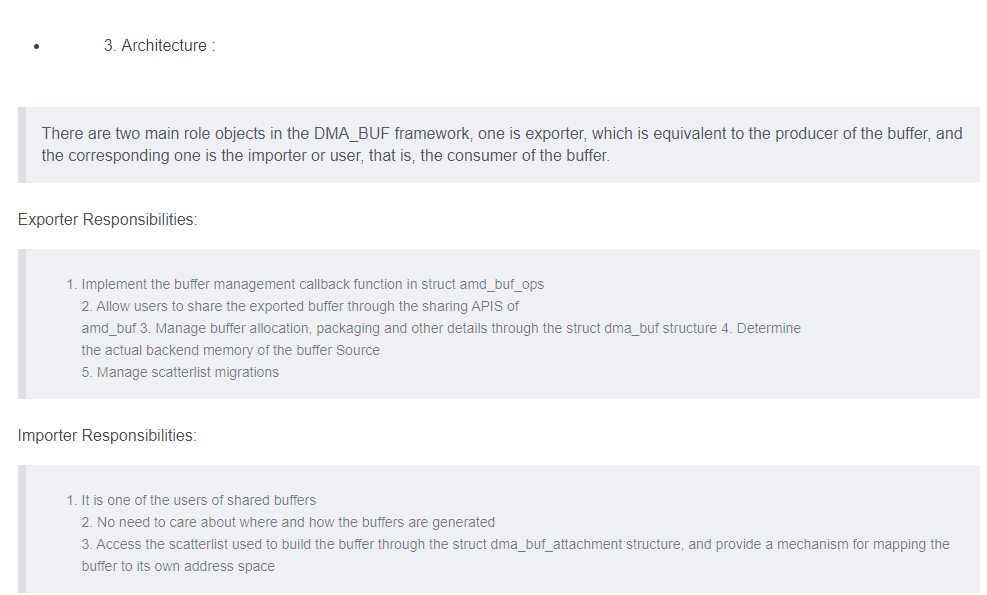
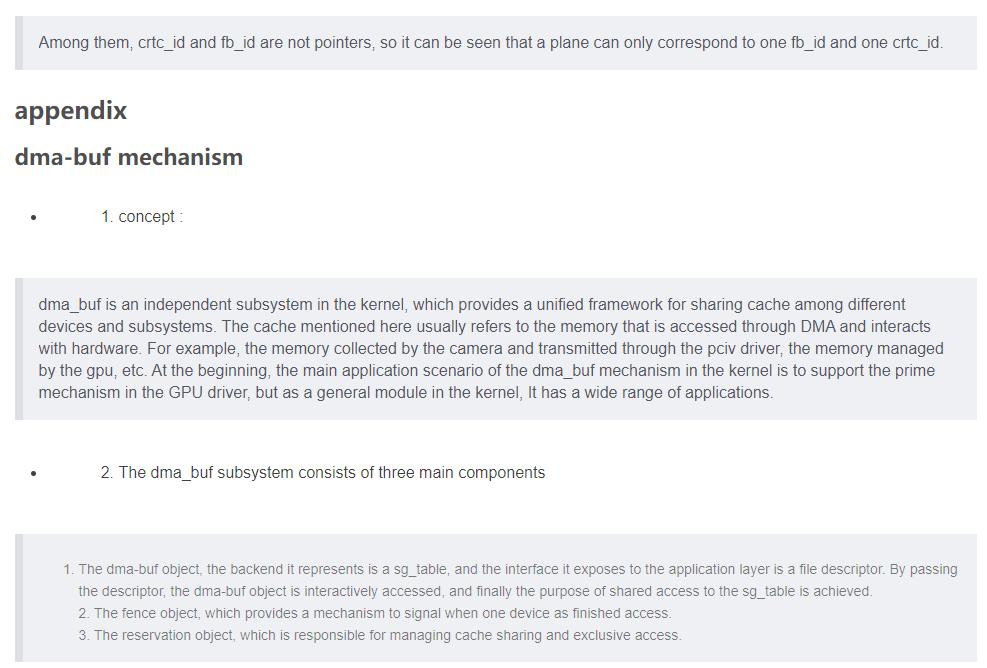
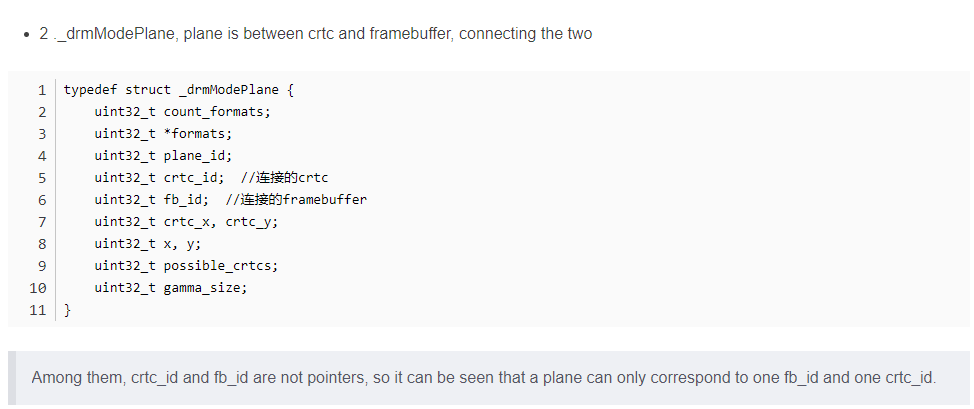


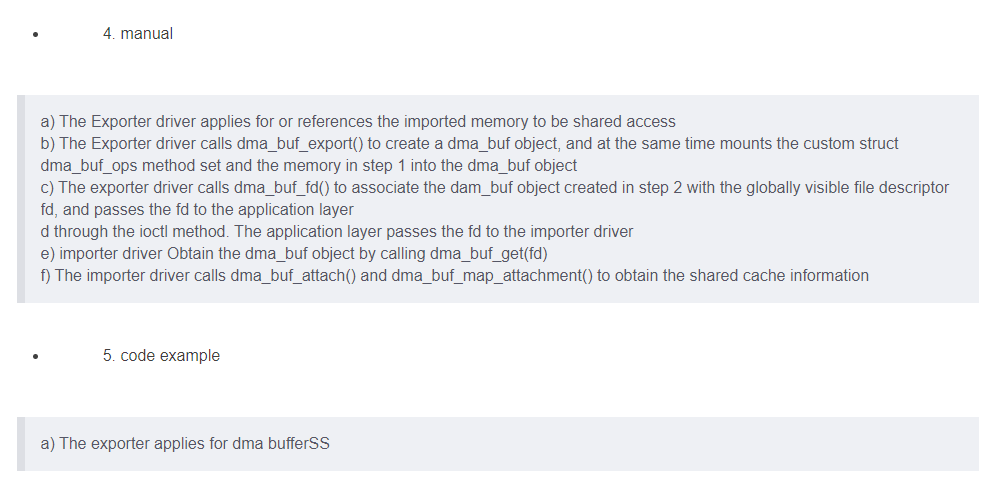
















# Basic concepts and usage examples of Linux DRM (C language)

**I. Introduction**

In this week's work, in order to solve the customer's problem, I checked the source code in awtk-linux-fb, and I was very interested in the content about DRM in it. After consulting my colleagues, I checked the information on the Internet. This article summarizes the learning of DRM. , and practice the use of DRM in C language. I would like to thank my colleagues for answering my questions.

**2. What is DRM**

**DRM (Direct Rendering Manager) is the direct rendering manager .** It is produced to solve the problem of cooperative use of Video Card resources by multiple programs. It provides a set of APIs to user space for accessing and manipulating the GPU.

To put it simply, DRM is a graphics rendering architecture under Linux that manages display output and allocates buffers. Applications can directly manipulate DRM's ioctl or use the interface provided by framebuffer to perform display-related operations. Later, it was encapsulated into libdrm library, so that users can control the display more conveniently.

**3. Basic Concepts Included in DRM**

To understand how DRM outputs the user's drawing to the display screen, the following concepts cannot be avoided. The specific relationship is shown in the following figure:

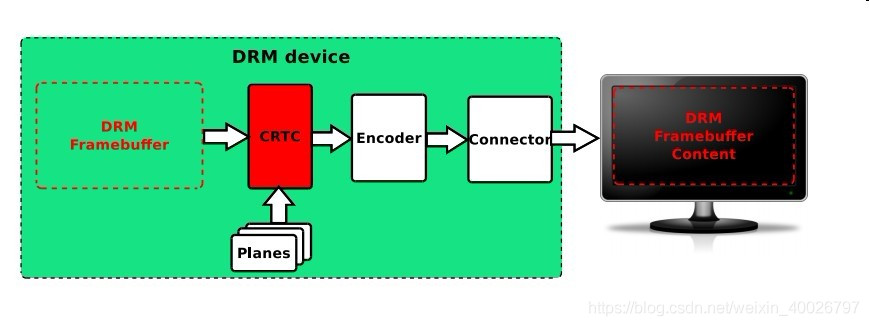
**Framebuffer**

**CRTC**

**Encoder**

**Connector**

**Display Device(LCD)**



**3.1 DRM Framebuffer**

It is a memory area, which can be understood as a canvas, which can be accessed by both the driver and the application layer. Before drawing, you need to format it, set the color mode (such as RGB24, YUV, etc.) and the size (resolution) of the canvas.

**3.2 CRTC**

Cathode tube context. This name is very difficult to understand, but in simple terms, it is the context of the display output, which can be understood as a scanner. The CRTC connects the Framebuffer address internally and the Encoder externally. It scans the contents of the Framebuffer, superimposes the contents of the Planes, and finally transmits it to the Encoder.

**3.3 Planes**

flat. It is the same memory address as Framebuffer. What does it do? For example, on a computer, while typing and chatting on WeChat while watching a movie, two concepts are opposed here. Typing is a text interaction, which is a Graphics mode for small-scale updates; watching a movie is a full-frame high-speed update Video mode. These two modes The use of graphics cards has been pulled to two extremes.

At this time, Planes has played a very good role. It provides a high-speed channel for Video refresh, making Video a separate layer, which can be superimposed on or below Graphic, and has functions such as scaling.

There can be multiple Planes, which is equivalent to layer stacking, so the image scanned by the scanner (CRTC) is actually a combination (Blending) of Framebuffer and Planes.

**3.4 Encoder**

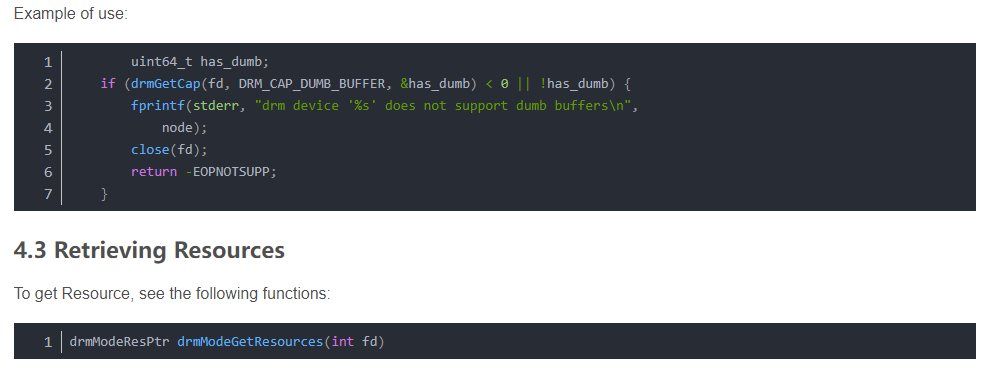
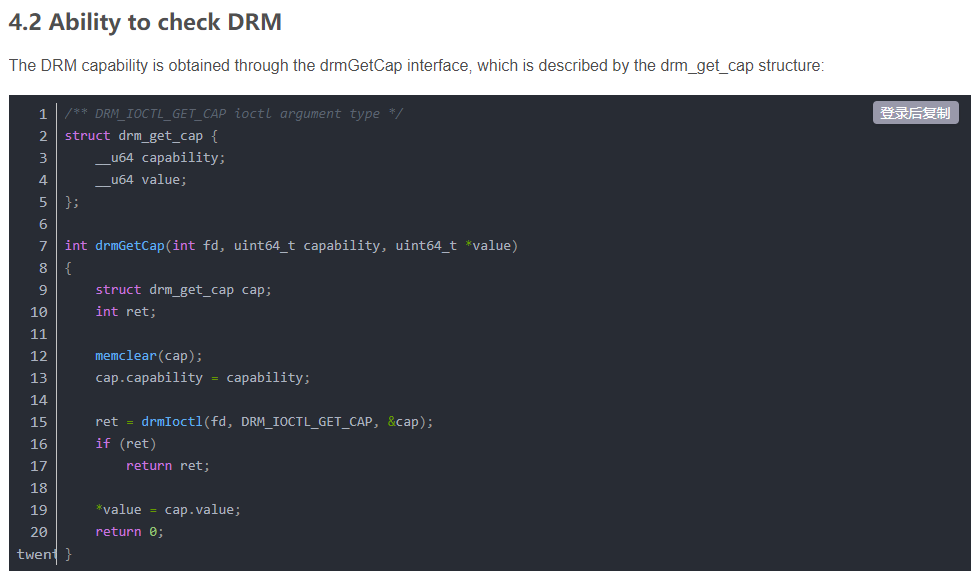
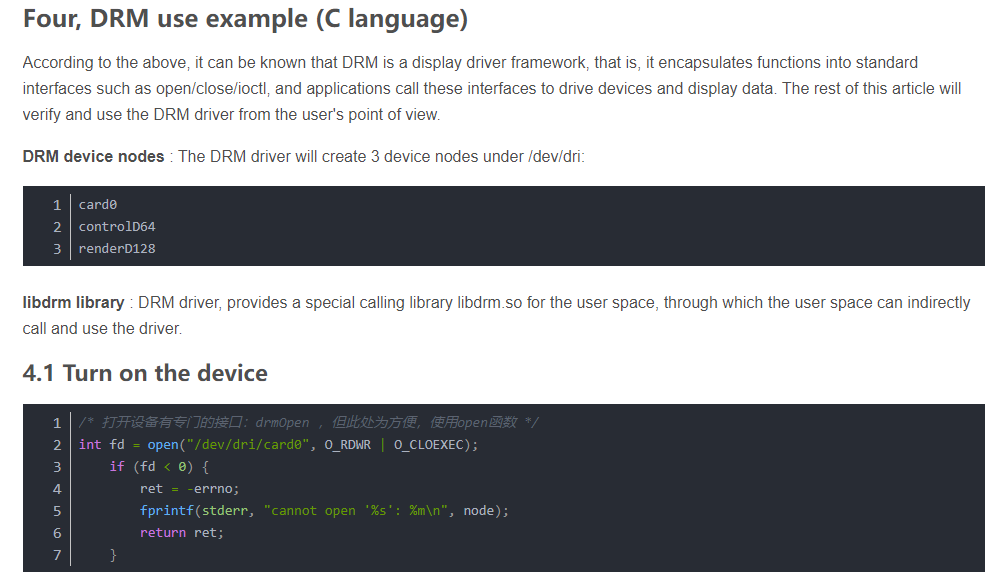
Encoder. Its function is to encode (convert) the pixels of the memory into the signals required by the display. The simple understanding is that if you need to display the picture on different devices (Display Device), you need to convert the picture into different electrical signals, such as DVID, VGA, YPbPr, CVBS, Mipi, eDP, etc.

The interaction between Encoder and CRTC is what we call ModeSetting, which includes the aforementioned color mode, timing, and so on.

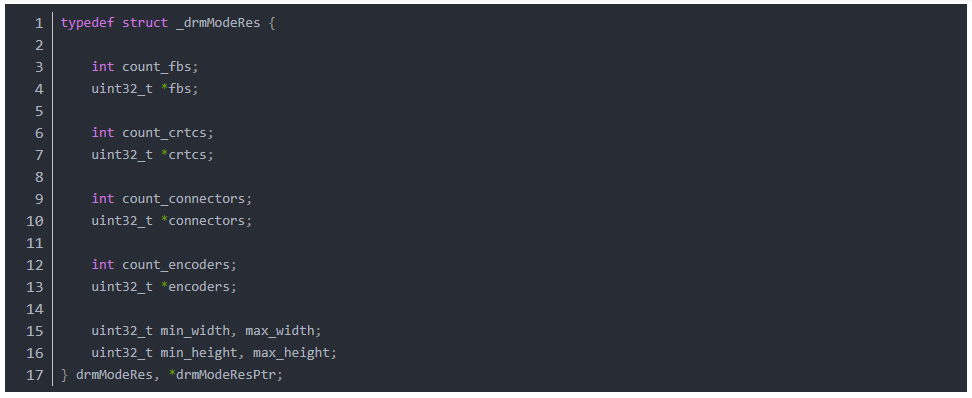
**3.5 Connector**

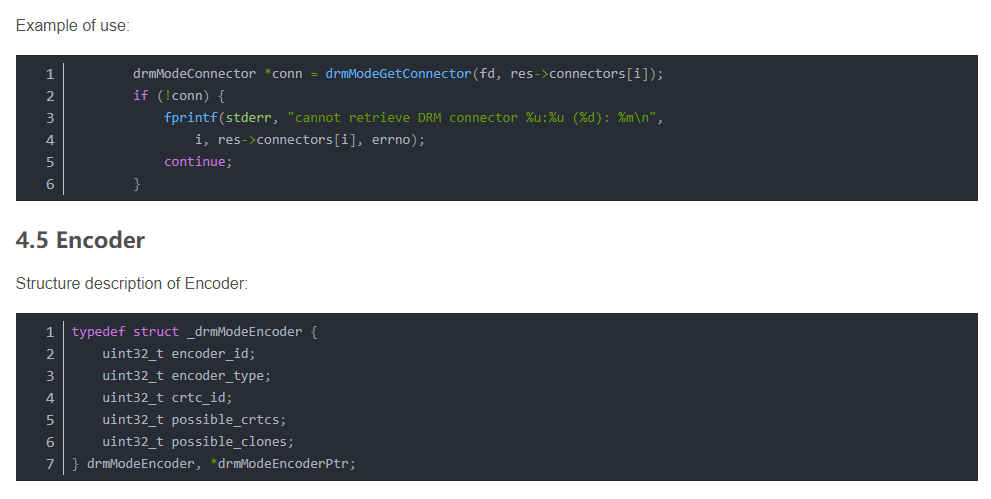
Connector. It often corresponds to a physical connector (such as VGA, DVI, FPD-Link, HDMI, DisplayPort, S-Video, etc.), it does not refer to a physical cable, in DRM, Connector is an abstract data structure that represents a connected display device , the information related to the currently physically connected output device can be obtained from the Connector, such as connection status, EDID data, DPMS status, supported video modes, etc.

**Four, DRM use example (C language)**

According to the above, it can be known that DRM is a display driver framework, that is, it encapsulates functions into standard interfaces such as open/close/ioctl, and applications call these interfaces to drive devices and display data. The rest of this article will verify and use the DRM driver from the user's point of view.

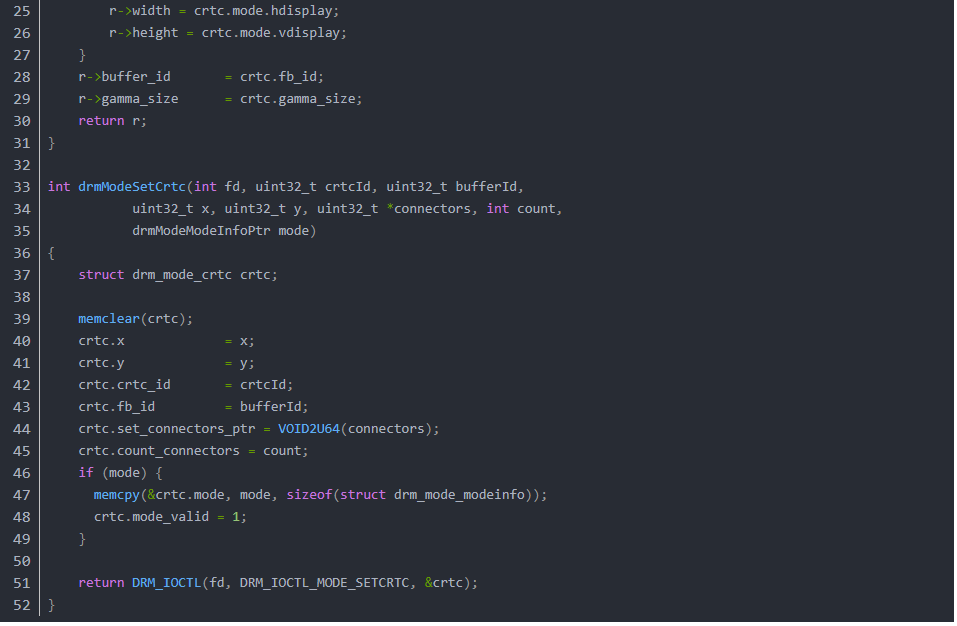


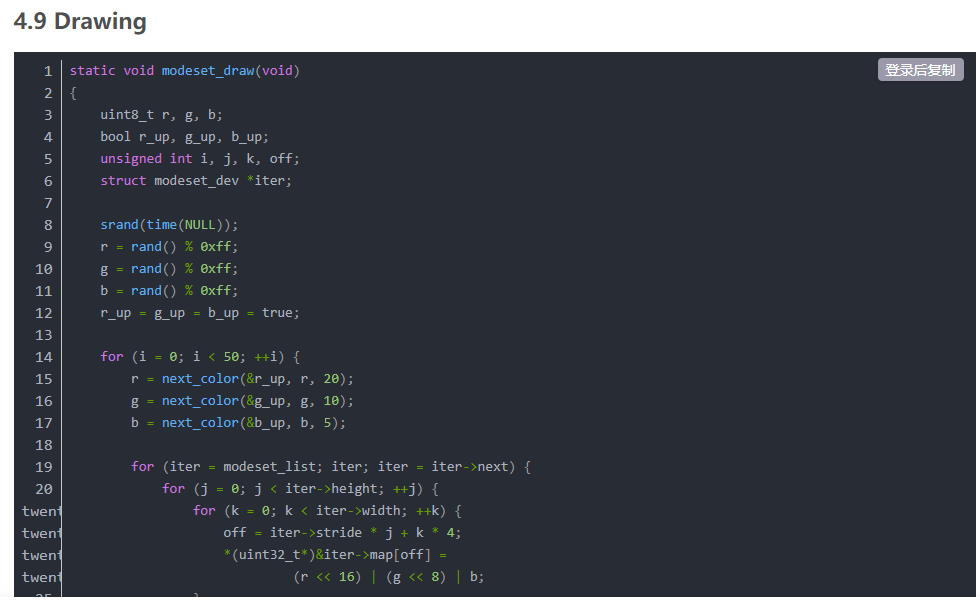


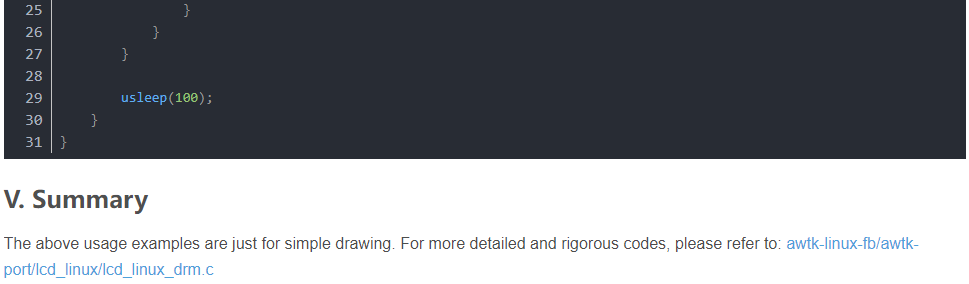












# [RK3399][Android7.1] DRM module introduction in Display

OS: Android 7.1

Board: Firefly-RK3399

Kernel: v4.4.55

**DRM introduction :**

The full name of DRM is Direct Rendering Manager, which is a device-independent kernel-level driver. The kernel provides direct access to hardware. It was originally designed for PCs to support complex graphics devices, and later used in embedded systems.

PCs generally have graphics cards and have their own video memory, while embedded systems do not.

**DRM composition :**

KMS(Kernel Mode Setting): change resolution and bit depth

DRI (Direct Rendering Infrastructure): Direct access to the hardware interface

GEM (Graphics Execution Manager): memory management

DRM Driver in kernel side: access hardware

**Without DRM, how can embedded systems achieve display and memory management?**

Display: Based on Linux Framebuffer

Multimedia: Based on V4L2

Buffer Manager: Based on ION, PMEM, etc.

**Why choose DRM?**

The community has been maintaining

Provides fine-grained control in the display section

user-space graphic is widely used

Offers a full set of advanced features

**Why not choose FBDEV or V4L2?**

FBDEV: There are fewer community maintainers; features such as overlay hw cursor cannot be provided; developers are encouraged to migrate to DRM/KMS in the future.

V4L2: Mainly used for video module, display module is a bit overkill.

**What are the advantages of DRM?**

Control all hardware devices through a single device node

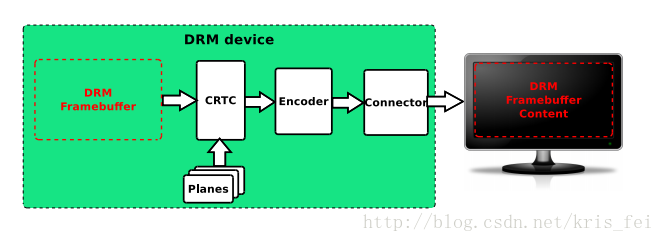
Generic Access Hardware Interface

General memory management mechanism

**Composition of KMS Framework:**

**Framebuffer, CRTC, Encoder and Connector, and Plane and drm device.**

The location of each module in the framework is as follows:

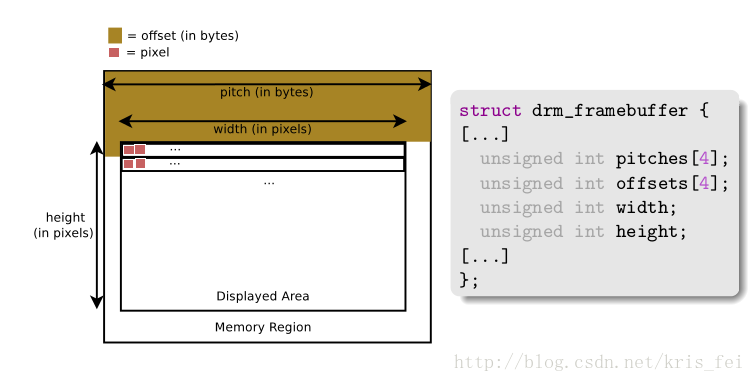


**Framebuffer:**

Memory information such as width, height, bpp, etc.

The code is represented by struct drm\_framebuffer,

which is created in rockchip\_drm\_fbdev\_create()@rockchip\_drm\_fbdev.c.



**CRTC:**

It used to represent the CRT controller, which is currently mainly used for display control, such as

the configuration of display timings, resolution, sending the framebuffer content to the display, updating the framebuffer, etc.

It is represented by two structures struct drm\_crtc\_funcs and struct drm\_crtc\_helper\_funcs in the code.

Created in vop\_create\_crtc()@rockchip\_drm\_vop.c.

**Encoder:**

Convert the data into a suitable format and send it to the connector. For example, HDMI requires TMDS information, and the encoder converts the data into the TMDS format required by HDMI.

It is represented by two structures struct drm\_encoder\_funcs and struct drm\_encoder\_helper\_funcs in the code.

Because it has a very close relationship with the connnector, its registration is placed in the driver file of each connector, such as rockchip\_dp\_drm\_create\_encoder()@analogix\_dp-rockchip.c

**Connector:**

Represents a specific external interface, such as edp, hdmi, mipi, etc. It is used to transmit signals to external hardware display devices and detect the access of external display devices.

Represented in the code by struct drm\_connector\_funcs and struct drm\_connector\_helper\_funcs.

There are several driver files, rockchip\_lvds.c, analogix\_dp-rockchip.c, cdn-dp-core.c, dw-mipi-dsi.c, dw\_hdmi-rockchip.c.

The registration of analogix\_dp-rockchip.c is placed in analogix\_dp\_core.c and will be called indirectly through rockchip\_dp\_bind().

Some of the contents of the two modules, Encoder and Connector, overlap, so it is difficult to separate them clearly.

**Planes:**

A Plane represents an image layer, and the final image consists of one or more Planes.

Different types of Planes:

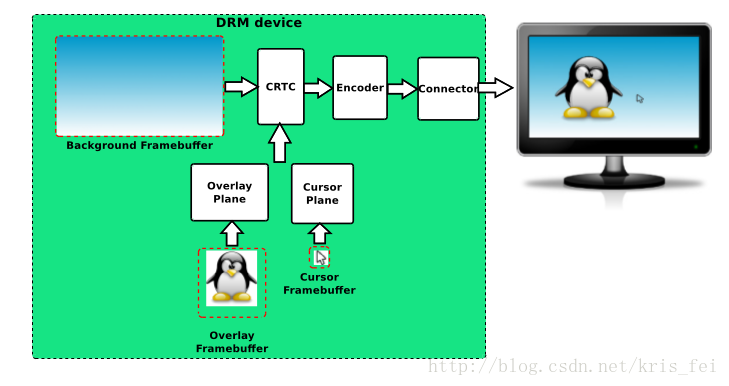
DRM\_PLANE\_TYPE\_PRIMARY: must have, due to display background or image content

DRM\_PLANE\_TYPE\_OVERLAY: Used to display Overlay

DRM\_PLANE\_TYPE\_CURSOR: used to display the mouse

It is represented by struct drm\_plane\_funcs in the code.

Create registration in vop\_plane\_init()@Rockchip\_drm\_vop.c.



**drm device:**

handles user space requests.

The execution is represented by struct drm\_driver in the code.

Created and registered in rockchip\_drm\_bind()@Rockchip\_drm\_drv.c.

Reference:

brezillon-drm-kms.pdf

DRM Driver Development For Embedded Systems.pdf

# Linux DRM (three) RK platform DRM code analysis- Rockchip

## 1. Overview

I don’t know if you still remember, I have quoted the introduction of DRM in Wiki before, let’s review it again: DRM consists of two parts: one is the subsystem of Kernel, this subsystem implements a layer of framework for hardware [GPU](https://so.csdn.net/so/search?q=GPU&spm=1001.2101.3001.7020) operation package. The second is to provide a libdrm library, which encapsulates a series of APIs for image display. On the whole, it is similar to the Direct Frame Buffer used on Android. Android Kernel takes the framework of FB, and abstracts a FBDEV in HAL for unified management of FB IOCTL. DRM is equivalent to direct centralized processing of graphics devices, and an additional libdrm library is added.

Its overall context is as follows:

# 

# 

# 

Component framework

Before talking about the startup process, let's take a brief look at the Component framework.

Because there are many devices attached to drm, the boot sequence often causes various problems:

* It is entirely possible for a driver to probe deferral while waiting for the preparation of another resource, resulting in an indeterminate order
* The sub device is not loaded, the main device is loaded, causing the device to fail to work
* The sub-devices may have a timing relationship with each other, and the loading order may be uncertain. Sometimes the device can work, and sometimes it cannot work.
* Now the kernel is compiled with multi-threading, and the order before and after compilation will also affect the loading order of the driver.

At this time, a unified management mechanism is needed to integrate all devices and load them in a unified order. Display-subsystem is officially used to solve this problem and depends on the Component driver. Through this driver, all devices can be Add them together in the form of components, and after all components are loaded, bind/unbind them together.

Code path: drivers/base/component.c

The following is the main logic of the Component in the rockchip drm master probe stage. In order to reduce the space, the irrelevant code has been removed:

# 

# 

# 

Based on Component mining, in the probe stage:

* Parse the information of each device in dts
* Add to Component match list
* After the device is loaded, the master device binds

### **RK DRM Device Driver**

**device tree**

display\_subsystem: display-subsystem {undefined  
        compatible = "rockchip,display-subsystem";  
        ports = <&vopl\_out>, <&vopb\_out>;  
        status = "disabled";  
};

- compatible: Should be "rockchip, display-subsystem"  
- ports: Should contain a list of phandles pointing to display interface port of vop devices. vop definitions as defined in  
kernel/Documentation/devicetree/bindings/display/rockchip/rockchip-vop .TXT

**drm driver**

code path:

drm\_driver rockchip\_drm\_driver = struct {static undefined  
.driver\_features = DRIVER\_MODESET | DRIVER\_GEM |  
DRIVER\_PRIME | DRIVER\_ATOMIC |  
DRIVER\_RENDER,  
.preclose = rockchip\_drm\_preclose,  
.lastclose = rockchip\_drm\_lastclose,  
.get\_vblank\_counter = drm\_vblank\_no\_hw\_counter,  
.Open = rockchip\_drm\_open,  
.postclose = rockchip\_drm\_postclose,  
.enable\_vblank = rockchip\_drm\_crtc\_enable\_vblank,  
.disable\_vblank = rockchip\_drm\_crtc\_disable\_vblank,  
.gem\_vm\_ops = &rockchip\_drm\_vm\_ops,  
.gem\_free\_object = rockchip\_gem\_free\_object,  
.dumb\_create = rockchip\_gem\_dumb\_create,  
.dumb\_map\_offset = rockchip\_gem\_dumb\_map\_offset,  
.dumb\_destroy = drm\_gem\_  
= drm\_gem\_prime\_handle\_to\_fd .prime\_handle\_to\_fd,  
.prime\_fd\_to\_handle = drm\_gem\_prime\_fd\_to\_handle,  
.gem\_prime\_import = drm\_gem\_prime\_import,  
.gem\_prime\_export = drm\_gem\_prime\_export,  
.gem\_prime\_get\_sg\_table = rockchip\_gem\_prime\_get\_sg\_table,  
.gem\_prime\_import\_sg\_table = rockchip\_gem\_prime\_import\_sg\_table,  
.gem\_prime\_vmap = rockchip\_gem\_prime\_vmap,  
.gem\_prime\_vunmap = rockchip\_gem\_prime\_vunmap,  
.gem\_prime\_mmap = rockchip\_gem\_mmap\_buf,  
#ifdef CONFIG\_DEBUG\_FS  
.debugfs\_init = rockchip\_drm\_debugfs\_init,  
.debugfs\_cleanup = rockchip\_drm\_debugfs\_cleanup,  
#endif  
.ioctls = rockchip\_ioctls,  
.num\_ioctls = ARRAY\_SIZE(rockchip\_ioctls),  
.fops = &rockchip\_drm\_driver\_fops,  
.name = DRIVER\_NAME,  
.desc = DRIVER\_DESC,  
.date = DRIVER\_DATE,  
.major = DRIVER\_MAJOR,  
.minor = DRIVER\_MINOR,  
};

**vop driver**

code path:

drivers/gpu/drm/rockchip/rockchip\_drm\_vop.c

drivers/gpu/drm/rockchip/rockchip\_vop\_reg.c

Structure:

struct vop;  
// vop drives the root structure, a vop corresponds to a struct vop structure

struct vop\_win;  
// Describe layer information, a hardware layer corresponds to a struct vop\_win structure

Register read and write: In order to be compatible with various versions of vop, the vop driver uses a register-level abstraction, and a structure is used to save the abstract relationship, so that the theme logic only needs to operate the abstract function definition, which is defined by the abstract read-write interface. Write to the real vop hardware according to the abstract relationship.

Example:

static const struct vop\_win\_phy rk3288\_win23\_data = {undefined  
        .enable = VOP\_REG(RK3288\_WIN2\_CTRL0, 0x1, 4),  
}

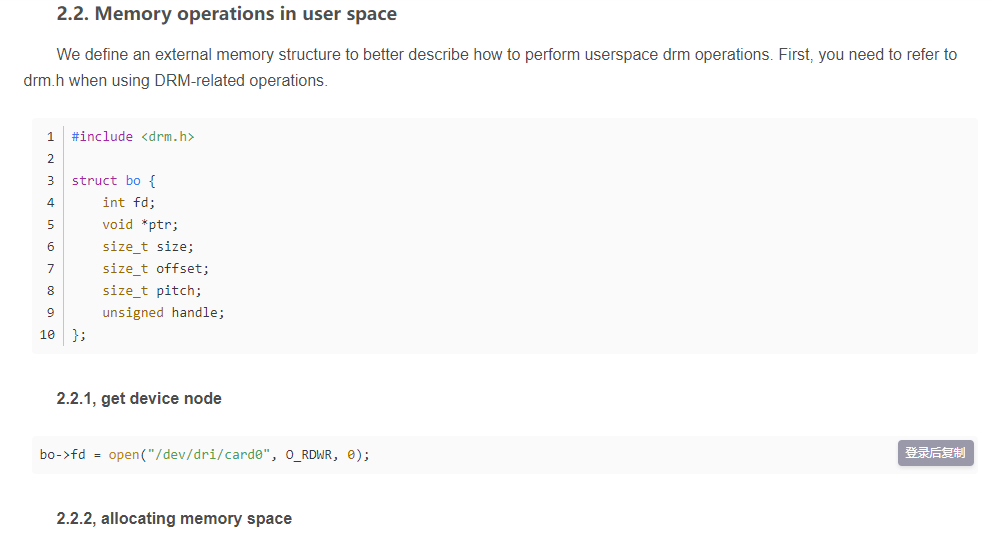
static const struct vop\_win\_phy rk3368\_win23\_data = {undefined  
        .enable = VOP\_REG(RK3368\_WIN2\_CTRL0, 0x1, 4),  
}

The address distribution of the rk3368 and rk3288 layers is different, but when the structure is defined, different hardware layer bits can be mapped to the same enable function, so that when the vop driver calls VOP\_WIN\_SET(vop, win, enable, 1), Can operate to the real vop register.

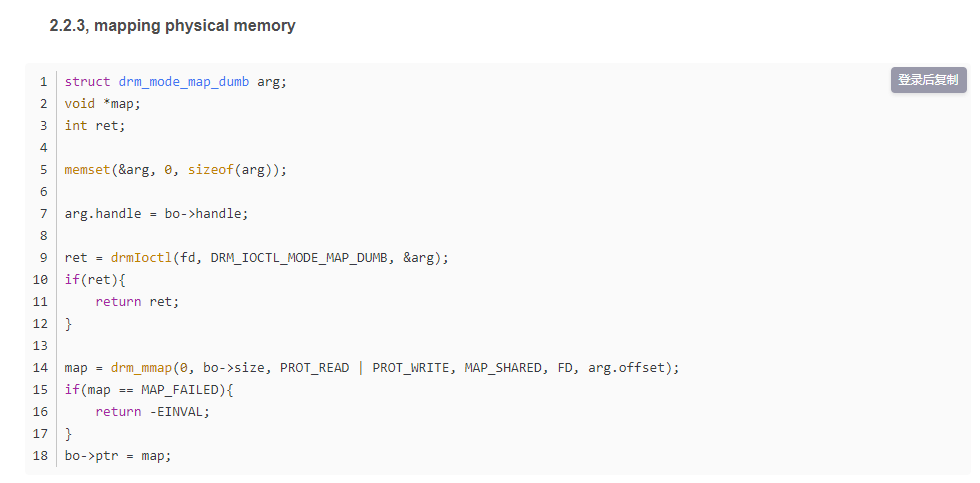
### **2.1. Device file cardX**

DRM is in kernel space, which means that user space needs to apply for its services through system calls. However DRM does not define its own system calls. Instead, it follows the "Everything is file" principle and exposes GPU access via the filesystem in the /dev/dri/ directory. DRM will detect each GPU, generate the corresponding DRM device, and create the device file /dev/dri/cardX to connect with the GPU. X is a value from 0-15, the default is Card0.

User-space programs that wish to access the GPU must open this file and communicate with the DRM using ioctls. Different ioctls correspond to different functions of the DRM [API .](https://so.csdn.net/so/search?q=API&spm=1001.2101.3001.7020)











### **2.2, DRM libdrm**

libdrm was created to facilitate the connection between user space and the DRM subsystem. It just provides wrappers (C) for some functions written for each ioctl, exposure, structure of the DRM API. Using the libdrm library not only avoids exposing the kernel interface directly to the user space, but also has common advantages such as code reuse.

### **2.3, DRM code structure**

Divided into two parts: general DRM Core and DRM Driver adapted to different types of hardware.

1. **DRM Core** provides the basic framework that different DRM drivers can register, and provides user space with a minimal set of ioctls with generic, hardware-independent capabilities.
2. **The DRM Driver** implements the hardware-dependent part of the API. It provides implementations of the remaining ioctls not covered by DRM Core, and it can also extend the API to provide additional ioctls. For example, a specific DRM Driver provides some enhanced APIs, and libdrm in user space also needs additional libdrm-driver extensions to use these additional ioctls.

### **2.4. DRM API**

**DRM Core** exports multiple interfaces to user-space applications, so that the corresponding libdrm is packaged into functions for later use.

The interface of the specific device exported by the DRM Driver can be used by the user space through ioctls and sysfs.

### **2.5. DRM-Master and DRM-Auth**

There are several ioctls in the DRM API that are limited to a single process in user space due to concurrency issues. To achieve this restriction, DRM devices are divided into Master and Auth. The above ioctls can only be called by the DRM-Master process. The file handle of the process that opened /dev/dri/cardX will be marked as master, especially the first process that calls the SET\_MASTER ioctl. Processes that are not DRM-Master will return errors when using these restricted ioctls. A process can also give up the Master role through the DROP\_MASTER ioctl to make other processes become Master.

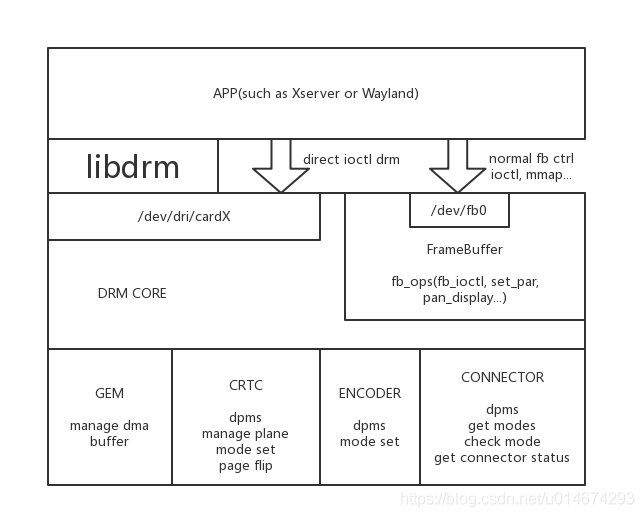
X Servers or other Display Servers are usually the DRM-Master processes for the DRM devices they manage. When the DRM device is started, these Display Servers open the device node and obtain DRM-Master permissions until the device is closed.

For other userspace processes, there is another way to obtain these restricted permissions for DRM devices, which is DRM-Auth. It is an authentication method for DRM devices to prove that the process has obtained permission from the DRM-Master for them to access restricted ioctls.

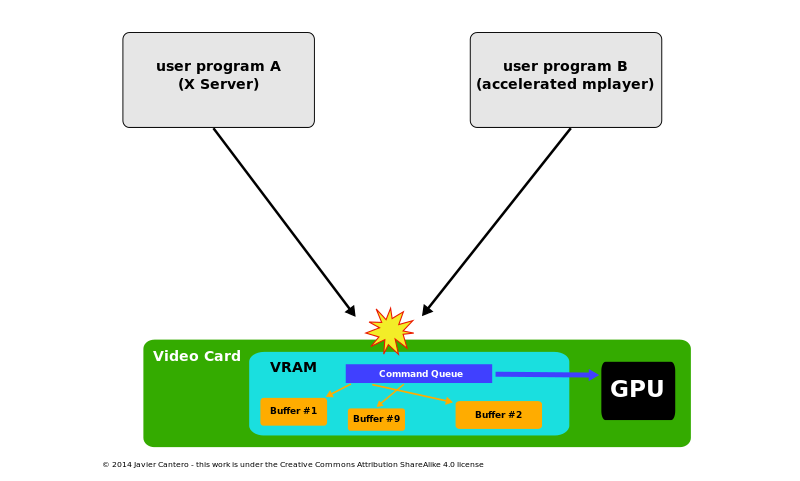
step:

* The **DRM Client** uses the GET\_MAGIC ioctl to obtain a 32-bit integer token from the DRM device. and passed to the DRM-Master by any means (usually IPC).
* The **DRM-Master** process uses the AUTH-MAGIC ioctl to return the token to the DRM device.
* The device compares the token given by the DRM-Master with the Auth one. If passed, the process file handle is given special permissions.

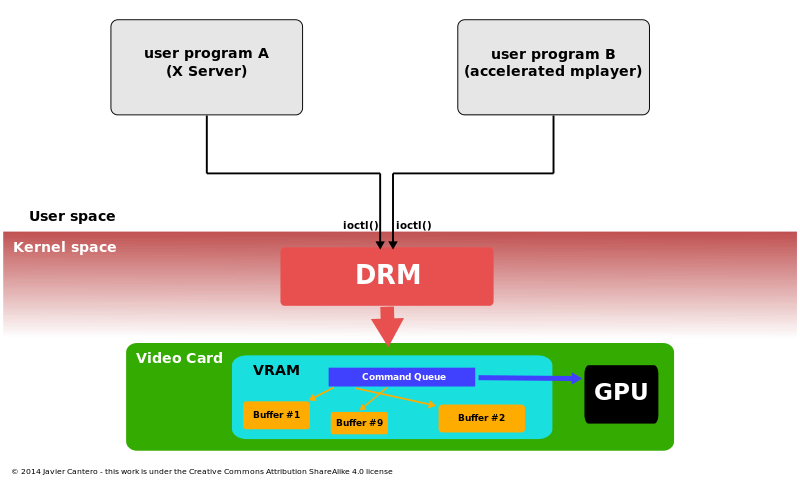
**The architecture of DRM in the source code (pictured from Mark.Yao):**



**Use DRM to access Video Card (pictured from Wikipedia):**



**When there is no DRM, the user space process accesses the GPU, as shown in the figure above.**



**With DRM, the way the user space accesses the GPU is shown in the figure above.**

# Code analysis of linux drm architecture

**The previous article introduced the architecture of the linux display driver drm, here we review it in a certain order:**

**1 I connect the monitor to the DVI output of the graphics card, this connection is abstracted into Connector**

**2 Drive the Encoder that will assign the DVI signal on the DVI Connector. If it is not assigned, all available encoders will be found on the connector resource**

**3 The encoder is used for image scanning on-site crtc service. The driver may assign crtc to the encoder, or you can find the available crtc from the encoder's possible\_crtc**

**4 crtc scan site to configure the physical memory area fb to display the image**

**5 After the relationship of fb -> crtc -> encoder -> connector is bound, the drawing work has already started, you can write any drawing on fb, and then it will be displayed immediately!**

**6 However, in order to avoid image tearing, you can create multiple fb (buffer) to refresh the drawing by pageFlip operation.**

**7 Of course, there is also a plane specially used for video refresh, and the plane must be bound to crtc to work.**

**Two summary + the use of drm api:**

**api use refer to David Herrmann <dh.herrmann@googlemail.com>'s drm-howto and weston and the modetest program that comes with drm**

**The core configuration of drm api is to bind a crtc relationship fb -> crtc -> encoder -> connector**

**Let's look at the review of "One", why is it in reverse order? Haha, in fact, this is only logical, how to use the api, look down:**

**1 First open the drm driver module, and then get all the resources**

**fd = open("/dev/dri/card0", O\_RDWR | O\_CLOEXEC);**

**drmModeRes　res = drmModeGetResources(fd);**

**What's in res? It tells how many connectors, how many encoders, how many crtcs, and their ids. It's not a complete set at all!**

**2 Start with the connector and follow the trail**

**Get the specific resources of connecotr first**

**drmModeConnector \* conn = drmModeGetConnector(fd, res->connectors[i]);**

**There are two important parts in the conn resource, one is to read out the "modes" of the display through the cable, such as 1920x1080@60, etc. Of course, you have to choose a favorite**

**The other part is that the encoder starts to find the encoder in the order of 2 in a chapter (see the definition of drmModeConnector) :p**

**3 Find a suitable crtc for the encoder**

**Find crtc in the order of 3 in a chapter**

**4 Create fb for crtc**

**The example modetest of drm is very detailed, ARGB can be directly drmModeAddFB, and multi-plane fb can be used drmModeAddFB2**

**5 binding**

**core quad < fb , crtc , conn , mode >**

**int drmModeSetCrtc(int fd, uint32\_t crtcId, uint32\_t bufferId,**

**uint32\_t x, uint32\_t y, uint32\_t \*connectors, int count,**

**drmModeModeInfoPtr mode);**

**6 pageFlip**

**extern int drmModePageFlip(int fd, uint32\_t crtc\_id, uint32\_t fb\_id,**

**uint32\_t flags, void \*user\_data);**

**Remember how to play this.**

**First, poll (drm\_fd) can receive the POLLIN message of drm. There are only two kinds of messages in the message, one is VBLANK, and the other is pageFlip complete**

**Secondly, after receiving the message, you must call drmHandleEvent(drm\_fd , &evctx); to process the message, remember that you must fill in your pageFlip processing function into evctx,**

**What are you doing in the pageFlip　processing function? One frame of pageFlip is over, and of course the next frame of pageFlip is required!**

**7 plane**

**I didn't understand the gameplay of plane. I tried it, using multi-buffer to adjust drmModeSetPlane to change the page to achieve video playback, but I encountered two problems:**

**One is the timing of SetPlane, how to wait for the field synchronization of a display? After reading the code of vblank, I feel that vblank is for the synchronization of one graphics card, but what about I have multiple monitors?**

**The second is the time-consuming operation of SetPlane. My i3 cpu took 22 ~ 23 ms to execute once, which is inexplicable.**

**I thought that the yuv channel of the plane can save resources, but there are two major problems.**