1. **X Windows and X Server**

Take a simple application scenario as an example. Clicking the button triggers the button update action.

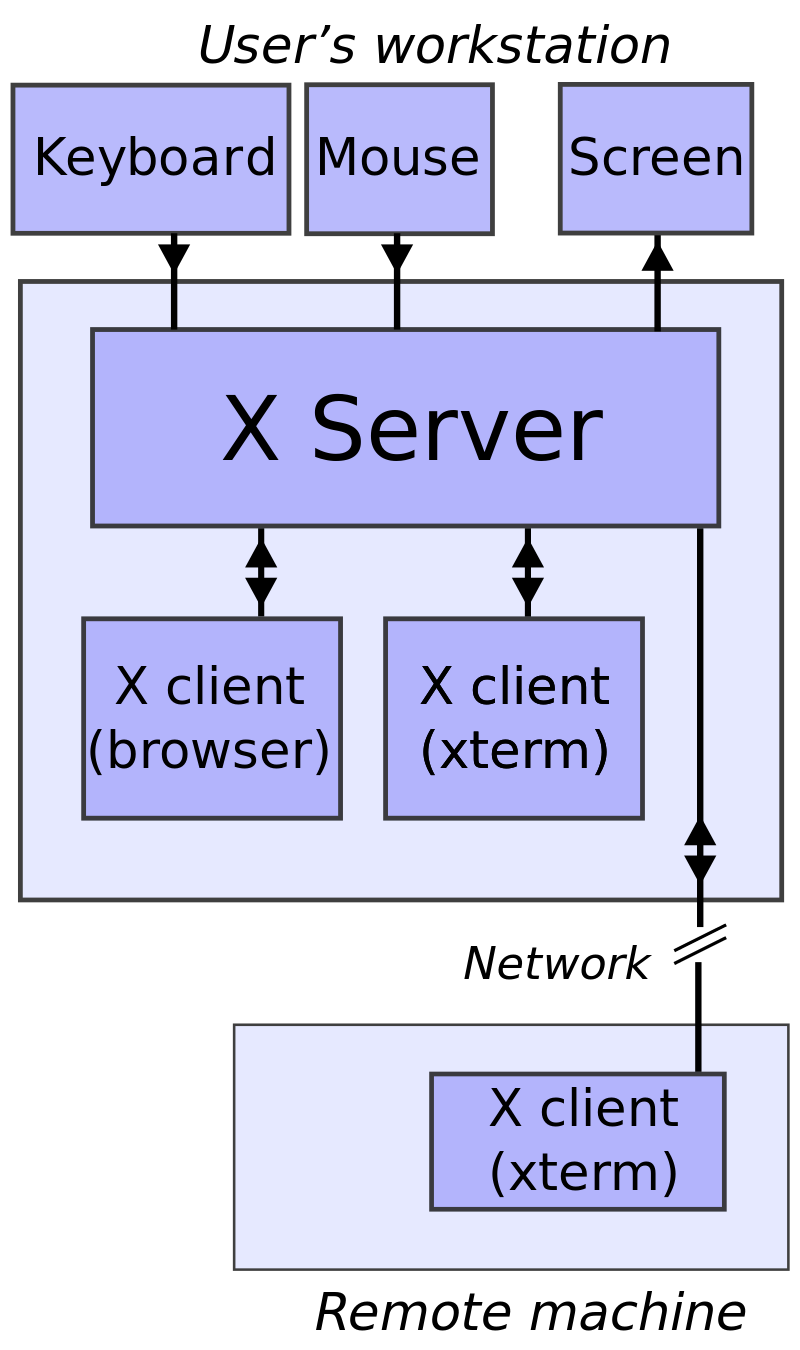
1. The kernel captures the mouse click event and sends it to the X server.

2. The X Server will calculate which window to send this event to (in fact, the window position is controlled by the Compositor, and the X Server cannot correctly calculate the correct position of the button after the Compositor has made a special effect change).

3. The application handles this event (the button update action will be triggered). However, before that it has to send a draw request to the X Server.

4. The X Server receives the drawing request and sends it to the video driver for rendering. X also calculates the update area, and this "spam" is sent to the Compositor.

5. At this point, the Compositor knows it has to recompose an area of ​​the screen. Of course, this still has to send a drawing request to the X Server.

6. Start drawing. But X Server will also do some unnecessary work (window overlap calculation, window clipping calculation, etc.).

Simple example: the X server receives input from a local keyboard and mouse and displays to a screen. A web browser and a terminal emulator run on the user's workstation and a terminal emulator runs on a remote computer but is controlled and monitored from the user's machine

We understand the above architecture diagram through an example.

It presents a disadvantage: the interaction between

X Client - X Server - Compositor is time-consuming and not very efficient.

In addition to the interaction issues, in fact, X Server will also do some repetitive and meaningless work, which will not be repeated here.

If there are pain points, there are solutions, and a new generation of Display Server is about to come out.

Then Wayland was born.

Wayland

Wayland is a Simple Display Server.

Its original mission was to improve upon and replace X Server.

But now it seems that what it has done is not only to replace X Server under X Window, but also to replace X Widnow. But to subvert the concept of X Server/X Client on the Linux desktop.

Replace it with the structure of Compositor/Client .

At the same time, it reuses all Linux kernel graphics and I/O technologies: KMS, GEM, DRM, evdev.

Its architecture diagram is as follows (the diagram comes from imtx.me author TualatriX):

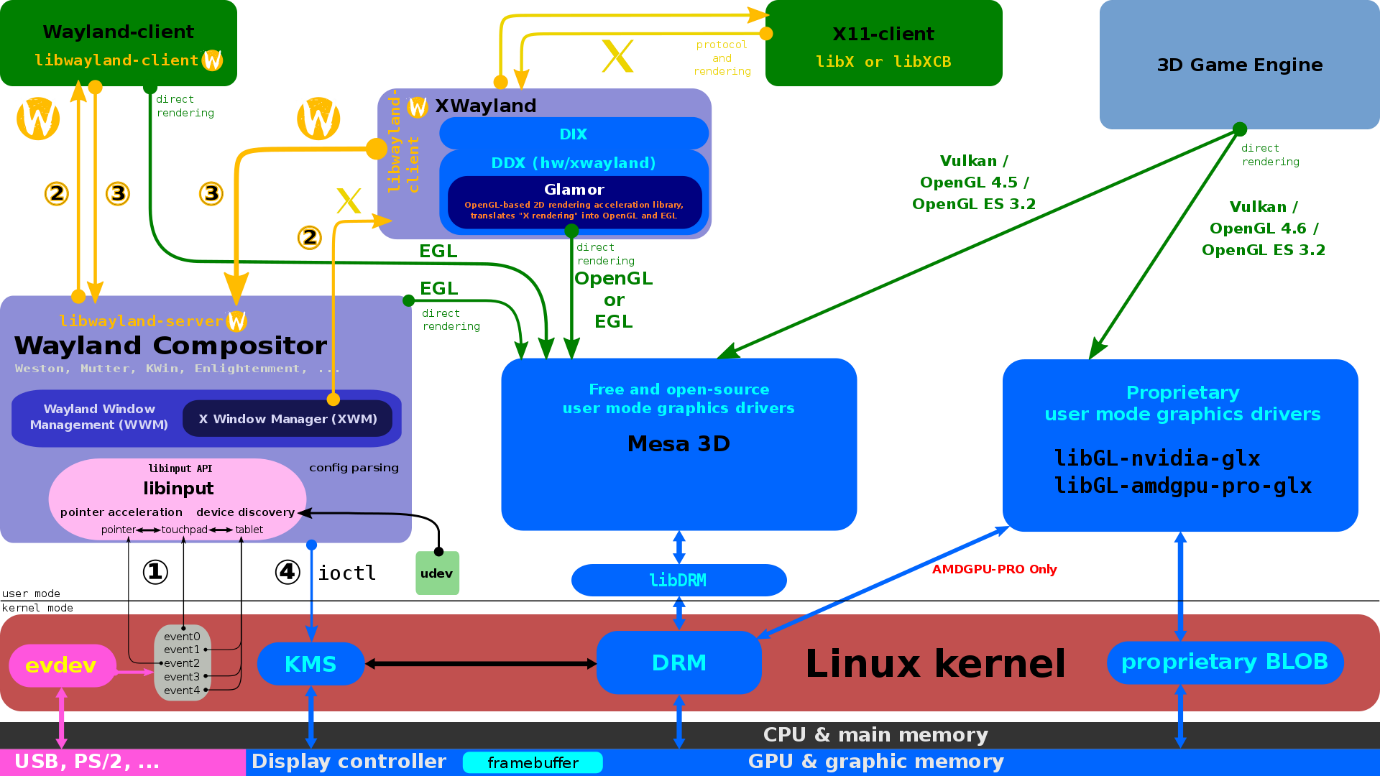
It is also the above example, and its process is as follows:

1. The kernel captures the mouse click event and sends it to Wayland Compositor.

2. Since it is sent directly to Wayland Compositor, Wayland Compositor will correctly calculate the position of the button. At the same time it will send this event to the application where the button is located for processing.

3. The application renders directly without requesting to Wayland Compositor. Just send a message to Wayland Compositor after drawing that the area has been updated.

4. After Wayland Compositor receives this message, it immediately recomposes the entire desktop.



[Wayland compositor](https://en.wikipedia.org/wiki/Wayland_compositor) and its clients use [EGL](https://en.wikipedia.org/wiki/EGL_(API)) to draw directly into the [framebuffer](https://en.wikipedia.org/wiki/Framebuffer); [X.Org Server](https://en.wikipedia.org/wiki/X.Org_Server) with [XWayland](https://en.wikipedia.org/wiki/XWayland" \o "XWayland) and [Glamor](https://en.wikipedia.org/wiki/Glamor_(software)).

So the whole task process based on Wayland is very simple:

1. Based on the Wayland protocol, process evdev information;

2. Notify the Client (that is, the application) to react to related events (as for how the application wants to react, the Compositor does not need to ask);

3. Receive status updates from the Client, recompose graphics or manage new graphics layouts.

Simply put, Waylannd is a display mechanism that removes unnecessary designs in X Window and makes full use of modern Linux kernel graphics technology (KMS, GEM, DRM in the DRM subsystem). Its appearance is natural, and its mission Not to eliminate X Window, but to bring Linux's graphics technology to a higher level. Traditional X Window (ie classic X applications, Gtk 1.x/2.x and other old applications) will also continue to be supported for a long time, running on Wayland Compositor in the form of Wayland Client until the final upgrade, replaced or eliminated.

With the above background, we can better understand the DRM Subsystem.

1. **DRM Subsystem**

In computing, the Direct Rendering Manager (DRM), a subsystem of the Linux kernel, interfaces with the GPUs of modern video cards. DRM exposes an API that user-space programs can use to send commands and data to the GPU, and to perform operations such as configuring the mode setting of the display. DRM was first developed as the kernel space component of the X Server's Direct Rendering Infrastructure,[1] but since then it has been used by other graphic stack alternatives such as Wayland.

User-space programs can use the DRM API to command the GPU to do hardware-accelerated 3D rendering and video decoding as well as GPGPU computing.

From Wikipedia

DRM, the full English name of Direct Rendering Manager, is the Direct Rendering Manager.

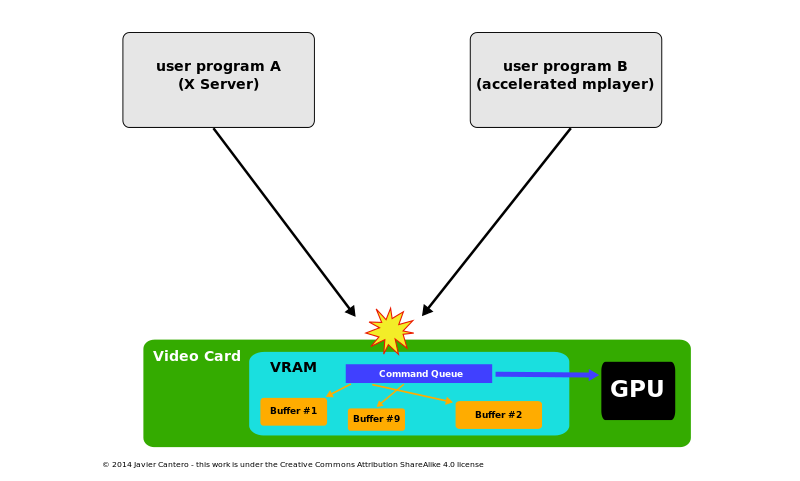
It is produced to solve the problem of cooperative use of Video Card resources by multiple programs. It provides a set of APIs to user space to access and manipulate the GPU.

**fbdev**

**Linux has had an API called fbdev for a long time to manage the framebuffer of the graphics card, but it cannot be used to handle the 3D acceleration requirements of the GPU based on the video card.**

**These Video Cards often need to set up or manage some command queues in the card memory (Video RAM). To assign commands to the GPU, it is also necessary to manage the Buffer and Free Space of the Video RAM itself.**

Programs in the original user space (such as X Server) can manage these resources directly, but these programs usually behave as if they are the only ones accessing these resources. It crashes when multiple programs try to control Video Card resources in their own way at the same time.



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

**DRM**

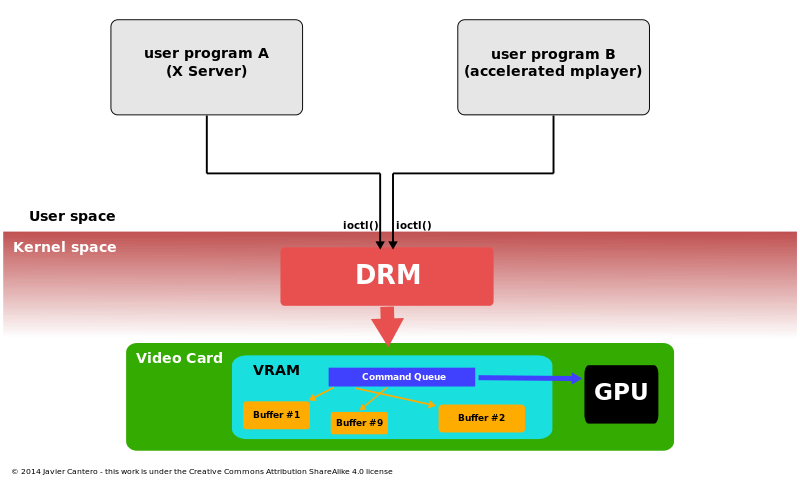
The birth of DRM is to deal with the collaborative use of Video Card resources by multiple programs .

The DRM gets exclusive access to the Video Card, which is responsible for initializing and maintaining the command queue, Video RAM, and other related hardware resources.

Programs that want to use the GPU send requests to the DRM, which acts as an arbitrator to avoid conflicts.

**DRM now covers many functions previously handled by user-space programs, such as framebuffer management and mode setting, memory shared objects, and memory synchronization. Some of these extensions have specific names, such as Graphics Execution Manager GEM or Kernel Mode Setup KMS, which belong to the DRM subsystem.**

DRM is also responsible for handling the switching of GPUs.



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

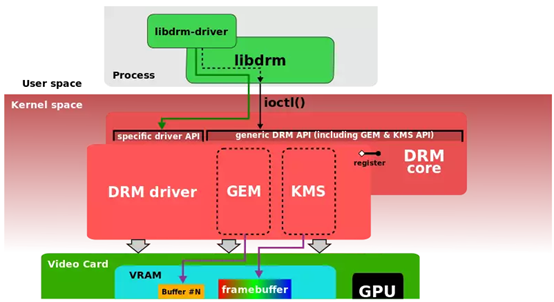
Introduction to DRM

In the Linux kernel, the graphics adapter's framebuffer can be managed using fbdev's [API , but it cannot handle the 3D acceleration capabilities of modern GPU-based graphics cards.](https://so.csdn.net/so/search?q=API&spm=1001.2101.3001.7020)DRM is a subsystem of the Linux kernel, originally developed as a kernel-space component of the X Server DRI, and subsequently used in other graphics stacks (such as Wayland). DRM is responsible for interacting with the GPU interface of modern graphics cards. User programs can use the API provided by DRM to send commands and data to the GPU to implement operations such as display mode settings, GPU hardware-accelerated 3D rendering, video decoding, and GPGPU (general-purpose GPU) computing. .

DRM framework

**The Linux DM framework consists of two parts: DRM core and DRM driver. DRM core implements the basic framework of DRM, can register DRM driver, and provides a set of ioctls for user space. DRM driver mainly implements some hardware drivers such as CPU/**[**GPU**](https://so.csdn.net/so/search?q=GPU&spm=1001.2101.3001.7020)**that it supports , and provides ioctl implementation not covered by DRM core or extended (only available on such hardware). User space programs can use libdrm to access various interfaces of the DRM framework.**

Since only libdrm and Linux DRM have been debugged in the actual project, these two parts will be introduced first. The following figure is excerpted from the network and introduces the DRM framework.



**Linux DRM (2) Basic Concepts and Features**

DRM is an acronym for Direct Render Manager:

Compared with the original framebuffer structure, DRM abstracts and manages more hardware modules, so it is more flexible and convenient;

Contains the following modules:

1. **GEM (Graphics Execution Manager)** is used to manage the application and release of display buffers

**DUMB: Only supports contiguous physical memory, based on the underlying CMA memory management implementation of Linux, mostly used in small resolution scenarios**

**PRIME: can support continuous & non-contiguous memory, based on DMA-BUF mechanism, can realize buffer sharing, mostly used in large memory complex scenarios**

**FENCE: buffer synchronization mechanism, implemented based on dma\_fence, to solve the phenomenon of tearing, frame shaking, frame return and other phenomena that occur when buffer control is not synchronized**

1. **KMS (Kernel Mode Setting)** is used to set display related configuration, display screen

Hardware module division

**CRTC:** Configure parameters such as timing and resolution

**ENCODER:** A module that converts data data to output data format

**Connector**: Connector, display data transmission module, such as LVDS TX / MIPI DSI / HDMI

**Panels**: The physical screen also includes the Overlay layer in the hardware module;

Other related concepts

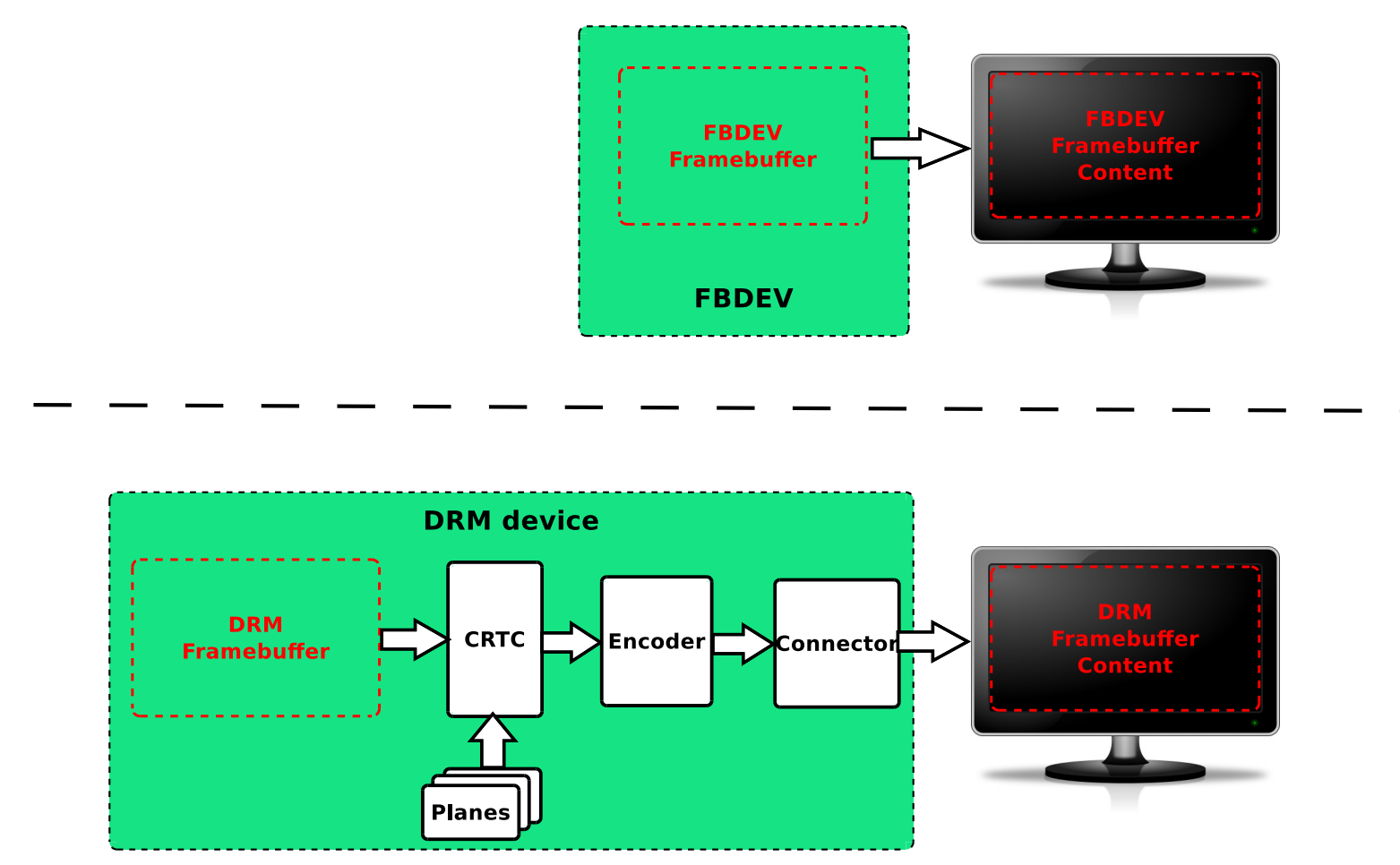
**framebuffer:** Displays the data in the buffer, generally the data rendered by the GPU or the input data of the camera;

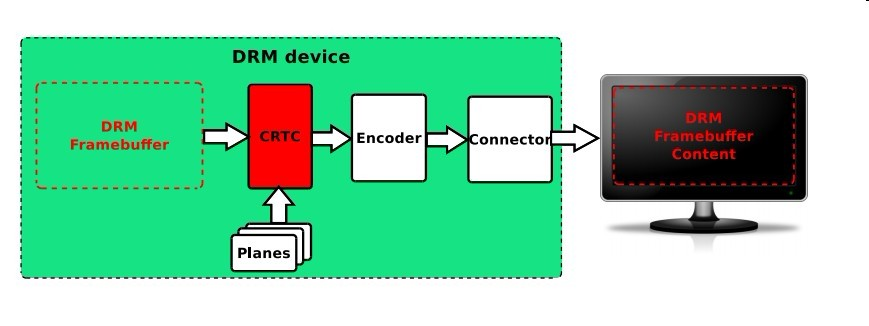
**VBLANK:** The synchronization signal is displayed in the DRM frame, which is triggered by the vsync signal in the DPU;

**page flip:** The data filling in the current framebuffer has been completed, and the display signal can be sent;

property: a general configuration item in the drm framework, used to supplement the content information that cannot be included above'

As shown in the figure, compared with the original FB structure, the DRM device structure abstracts the related hardware modules out of objects for management, and can be configured more flexibly





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, …).

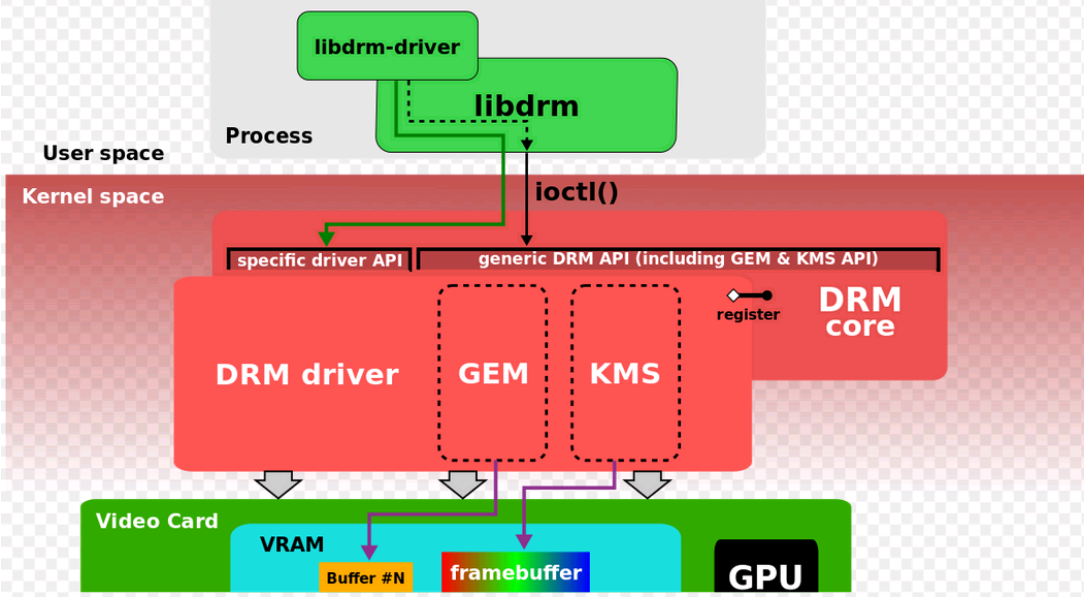
Information related to the currently physically connected output device (such as connection status, EDID data, DPMS status, or supported video modes) is also stored within the connector.

**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.

1. **Code analysis**

First look at the DRM Architecture diagram on Wikipedia:



Implement the libdrm library to provide a standard API at the user layer to operate the DRM device, and obtain the various components (CRTC, Connector, Encoder)

Implement core management in the kernel layer:

* Docking the upper libdrm structure
* Provide a registration interface, and register crtc, encoder, connector, etc. through such interfaces when drv is actually implemented
* The ioctl corresponding to the physical device operation is also linked with the actual drv

GEM management in the kernel

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

1. **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.

1. **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...)

1. **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.

1. **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.

**Main module**

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

1. **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.

1. **GEM**

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

1. **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

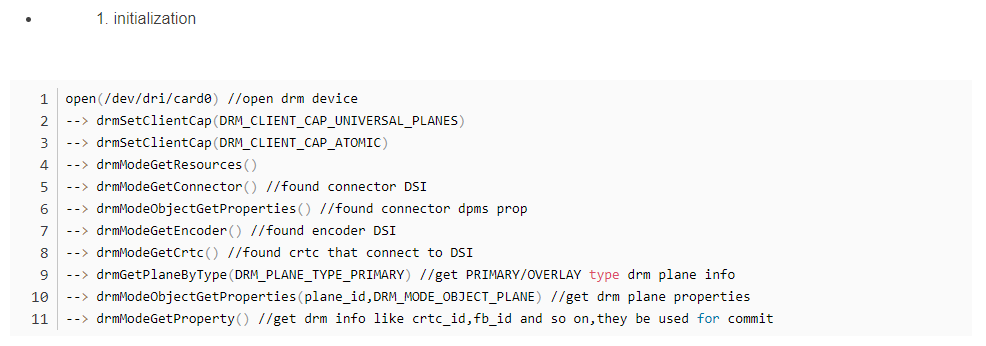
First, the HDMI driver detects the Plugin signal of the TV, reads the EDID signal of the TV, and obtains the resolution information of the TV (DRM Connector)

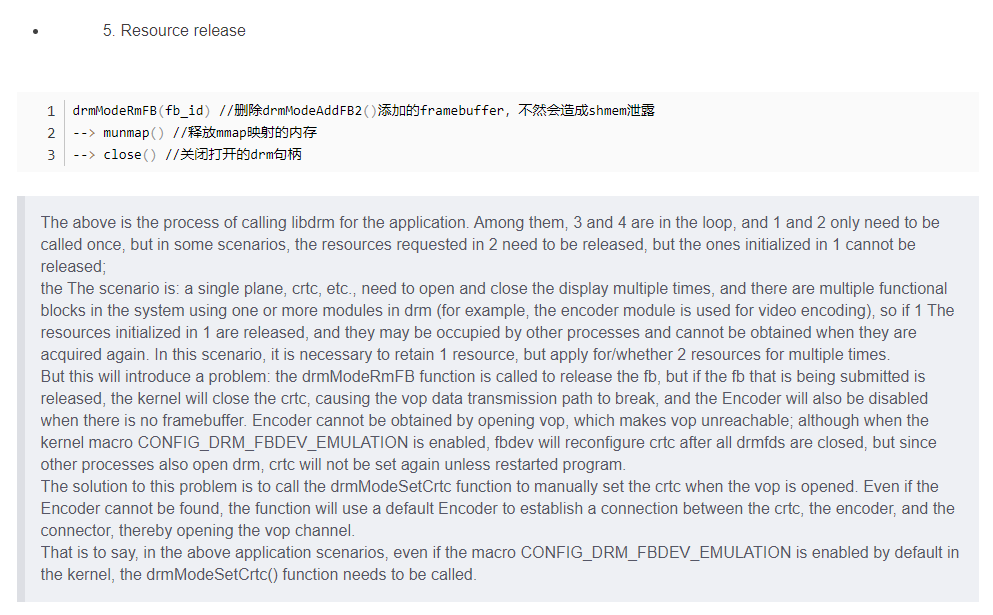
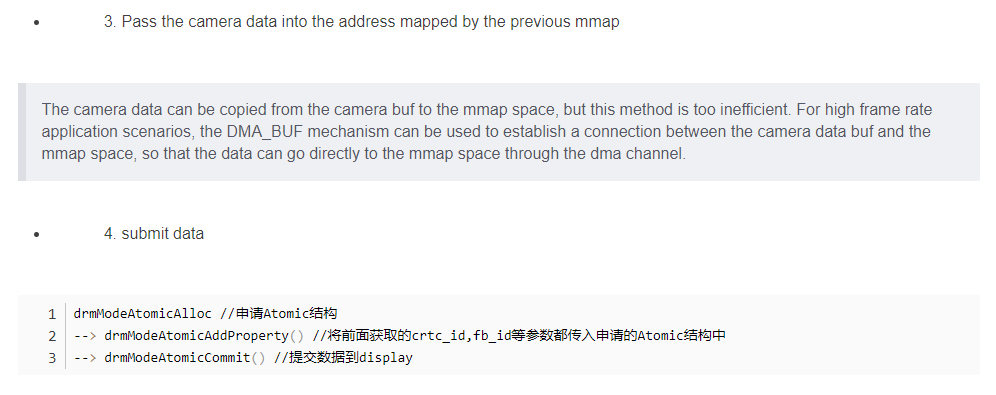
Userspace fills the framebuffer with the data to be displayed, and then informs the VOP device to start displaying through the libdrm interface

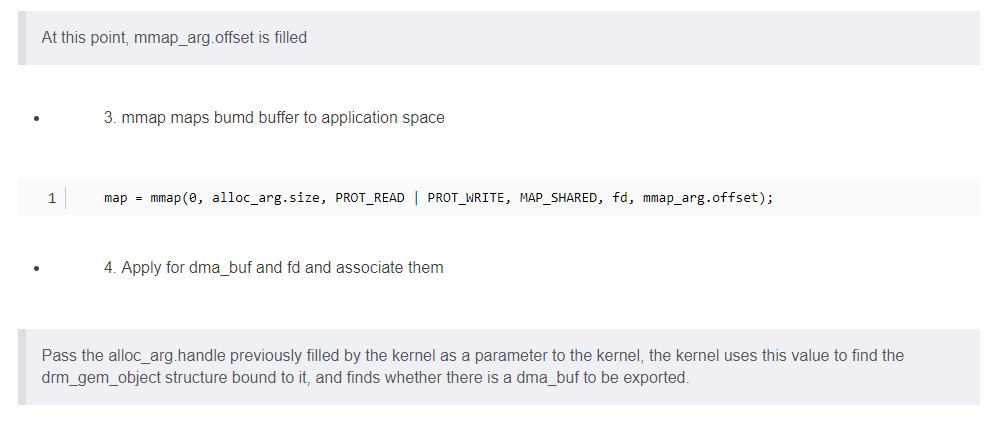
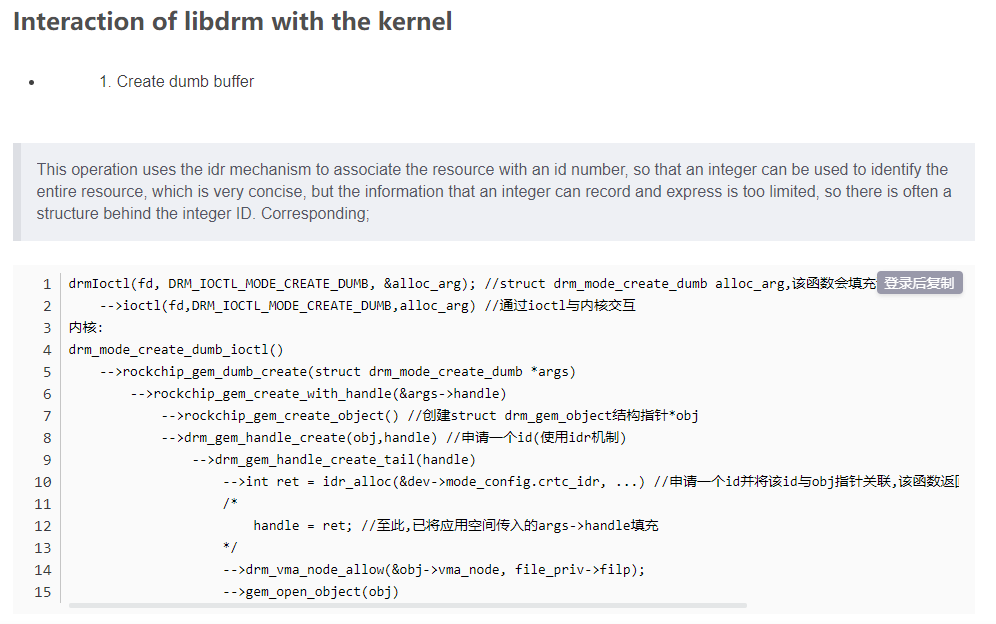
Then the VOP driver converts the data in the framebuffer into standard LCDC Timing timing (DRM CRTC)

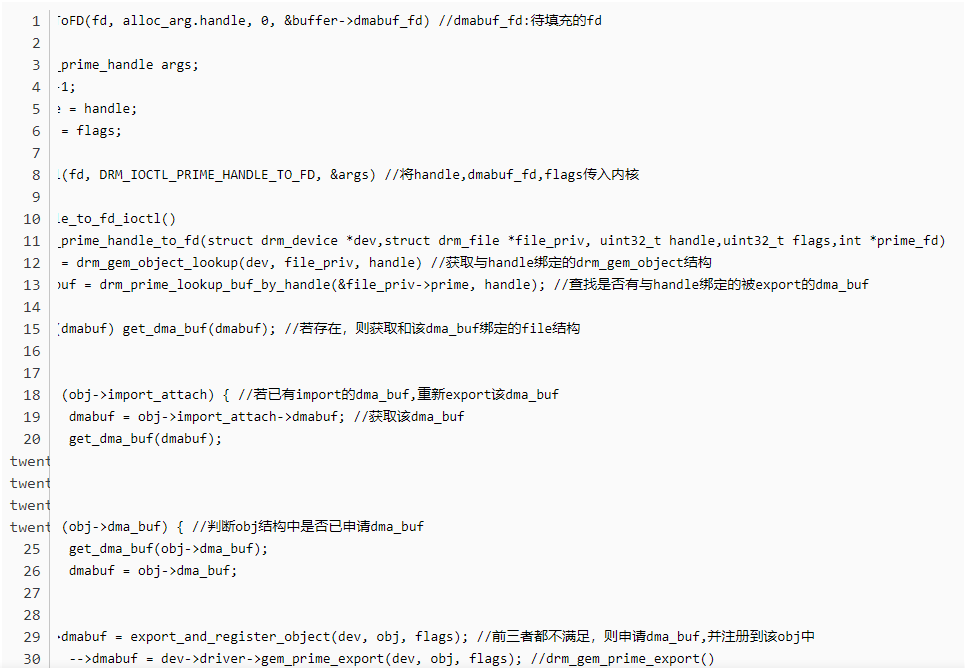
At the same time, the HDMI driver aligns the LCDC timing configuration of the HDMI hardware module with the VOP output timing, and prepares to convert the input LCDC Timing into the HDMI TMDS signal (DRM Encoder) recognized by the TV.

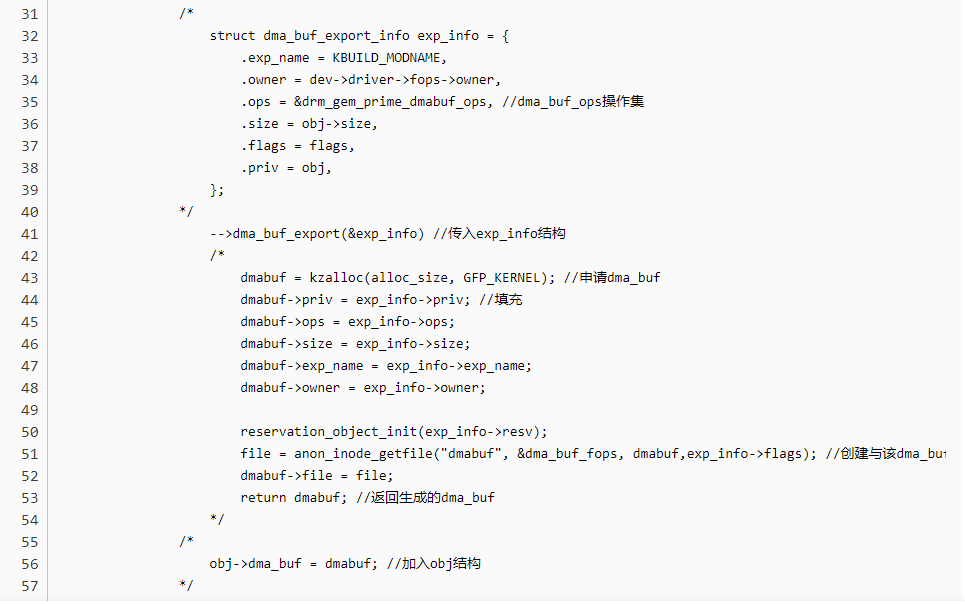
# DRM Application call process

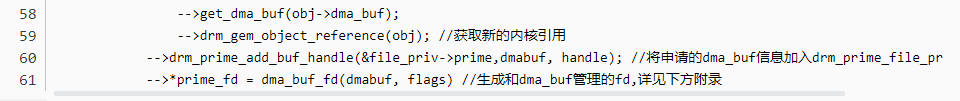


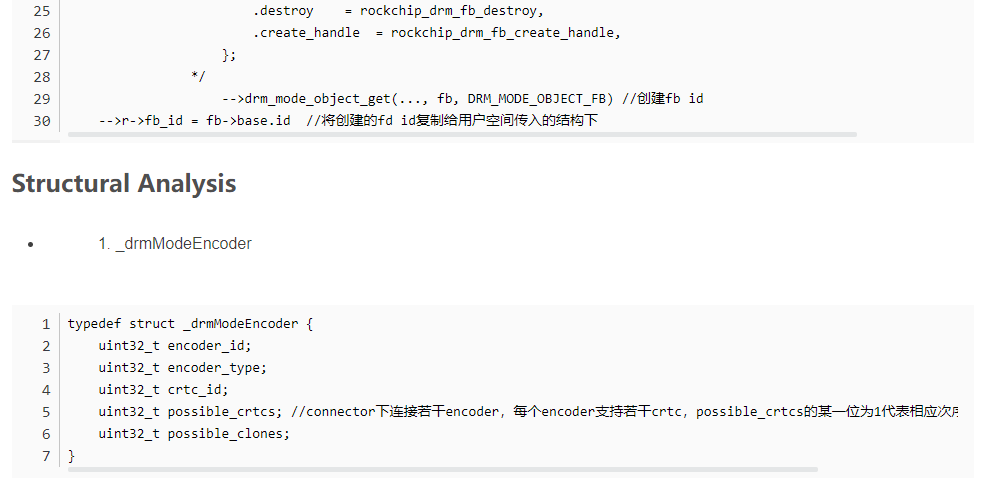
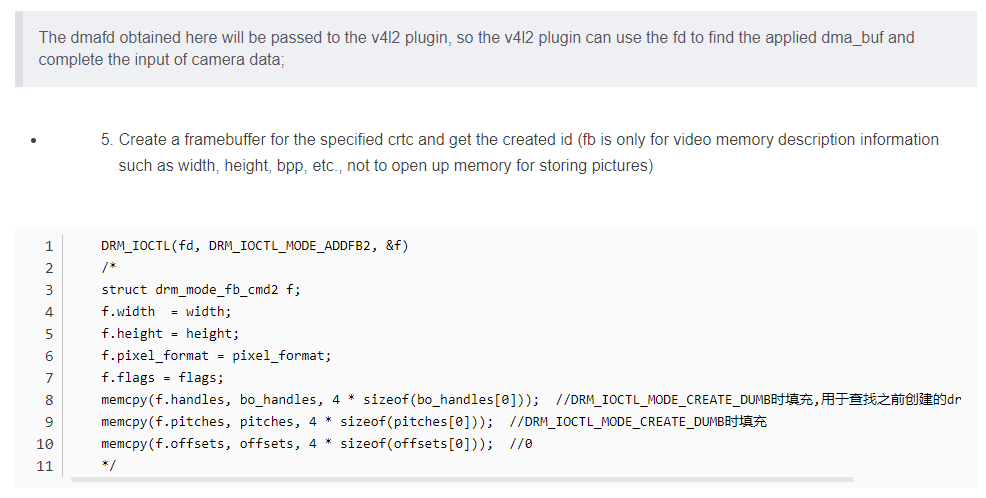
The order of obtaining crtc, encoder, connector, plane is connector, encoder, crtc, plane. Each structure of the former has the id number of the latter. After finding the connector, you can use the encoder\_id under its structure to find the one connected to yourself. The encoder module is the same until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

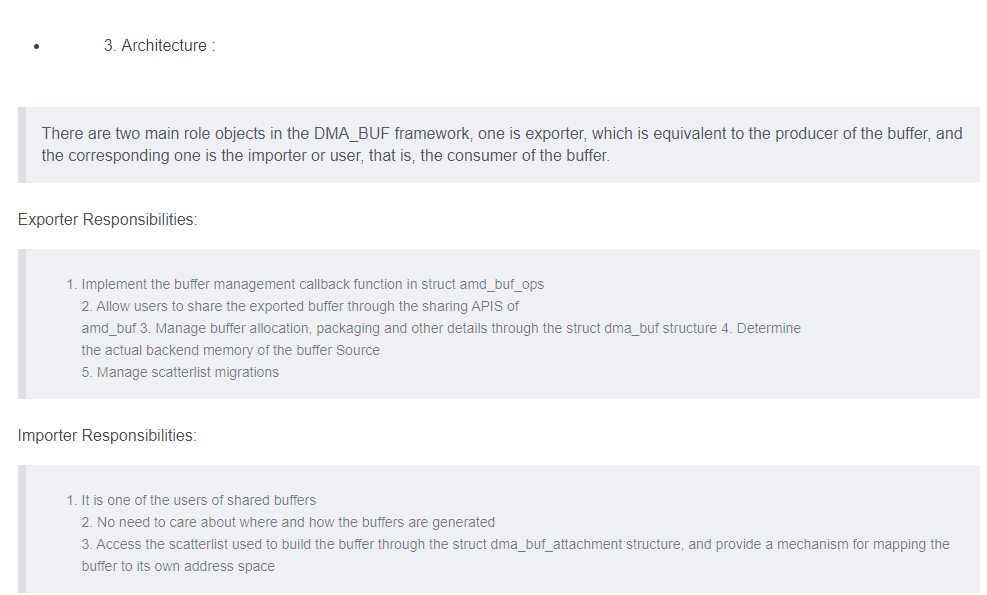
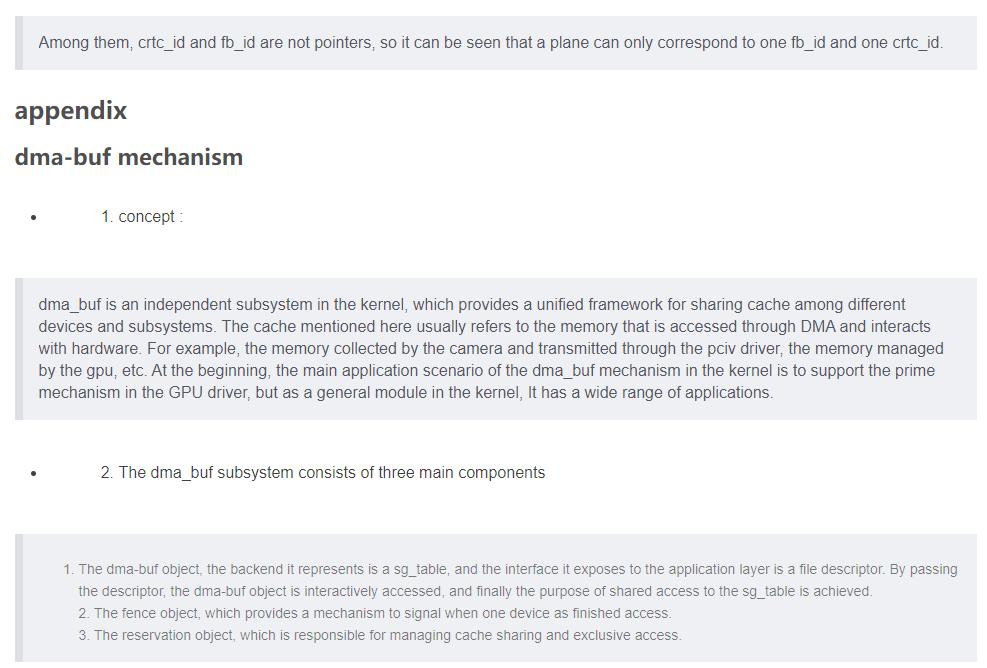
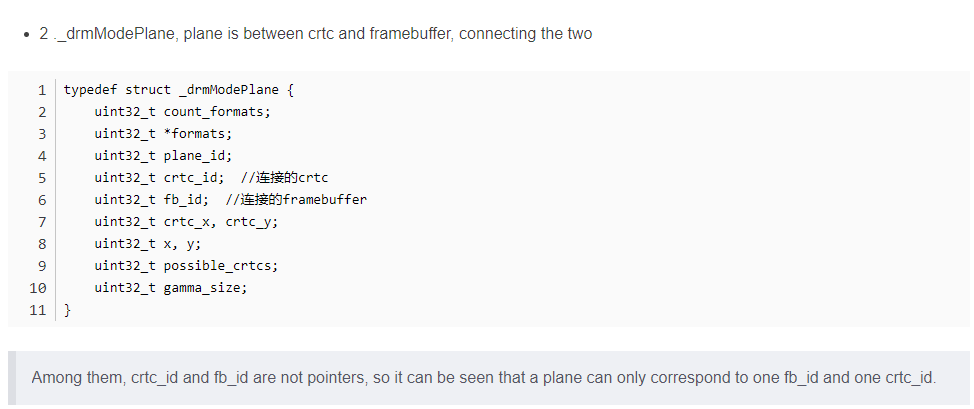


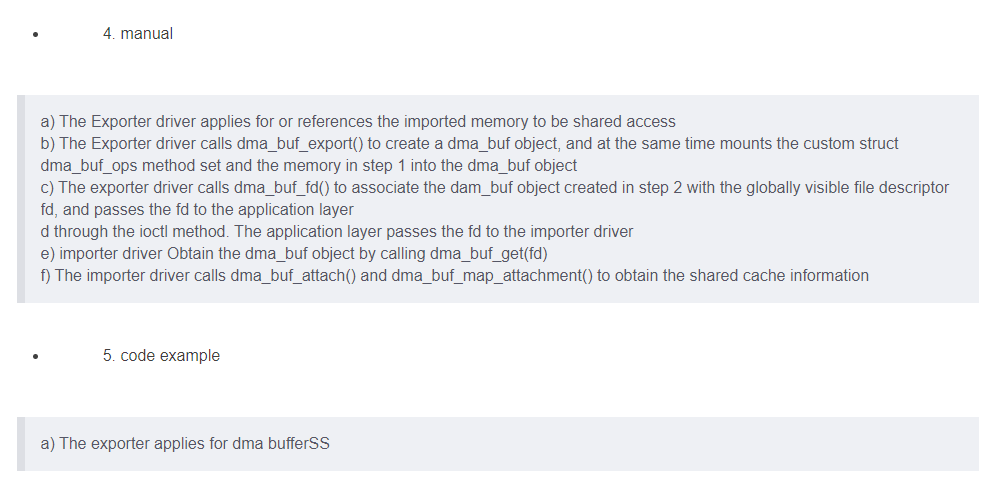








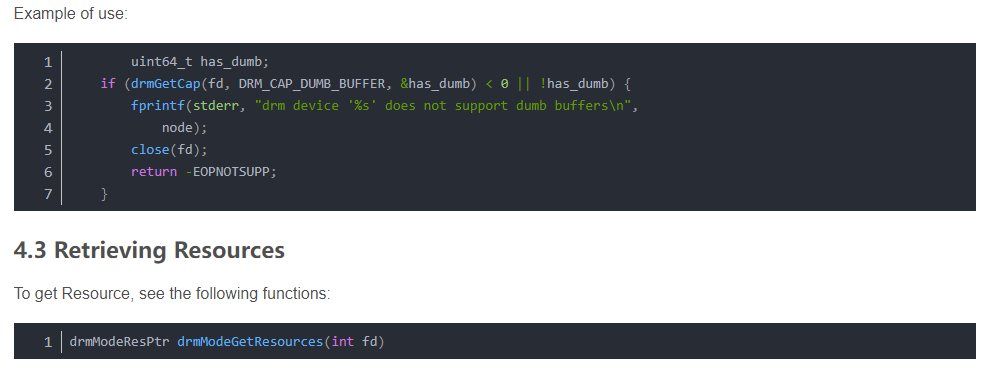
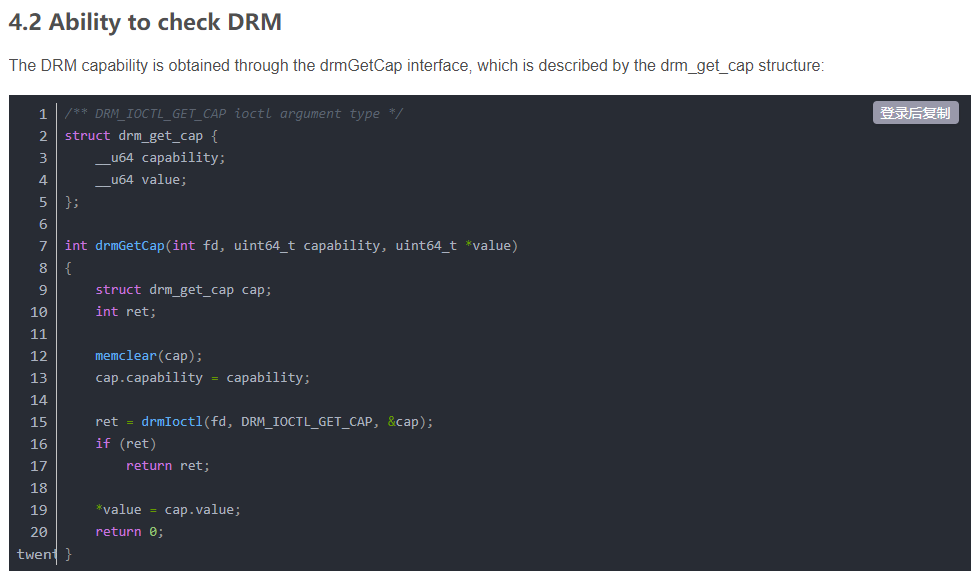
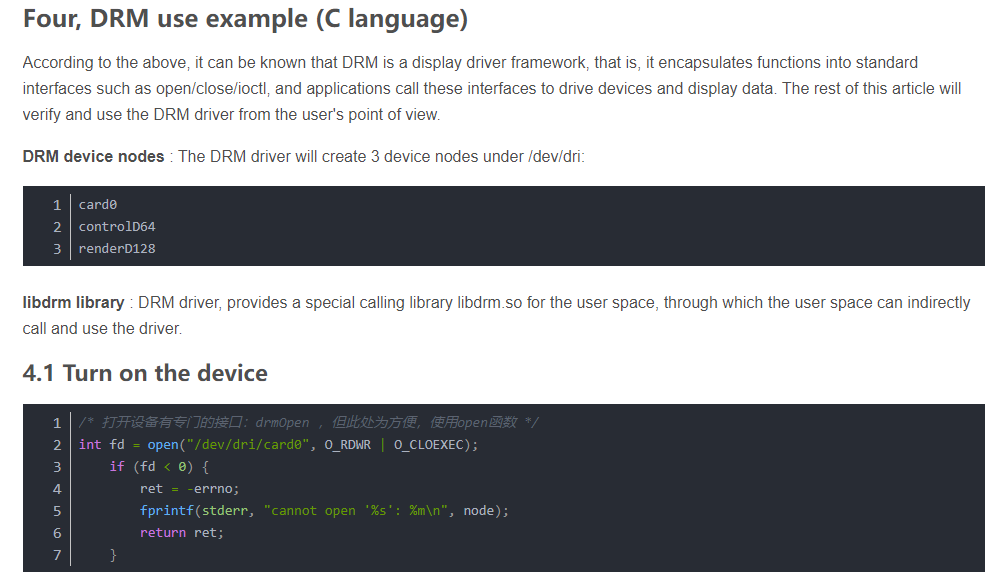




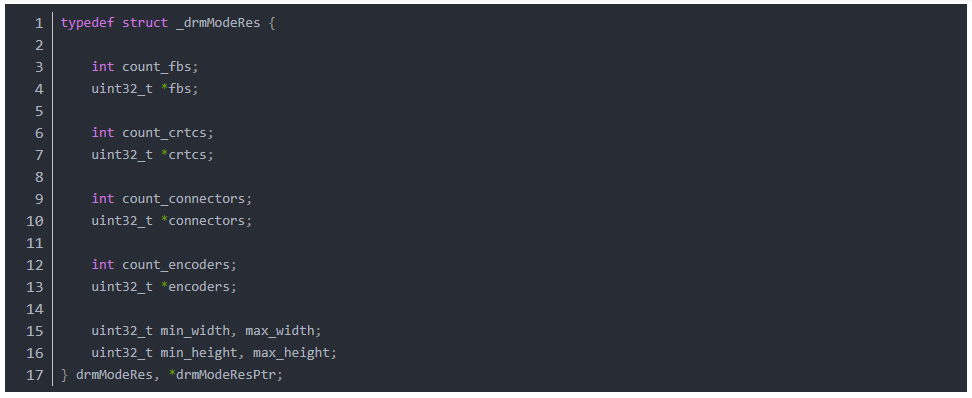


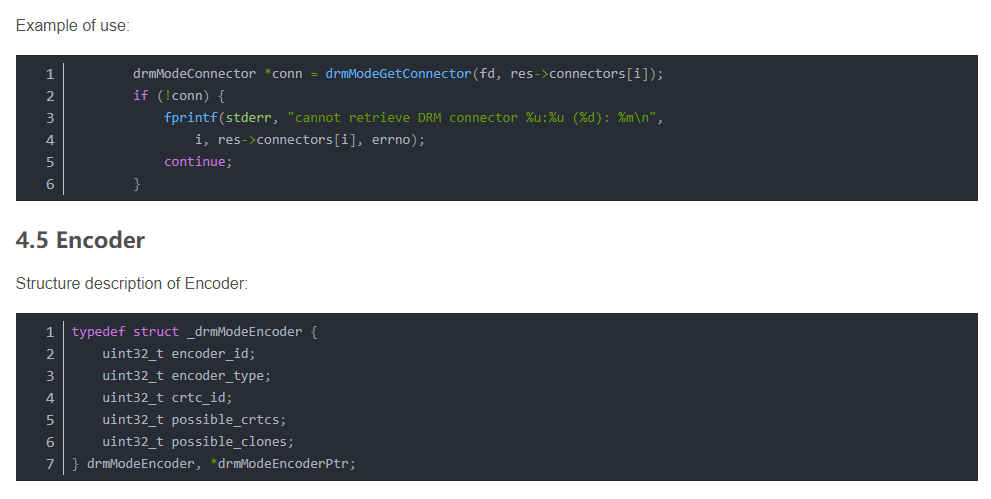


**Four, DRM use example (C language)**

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.

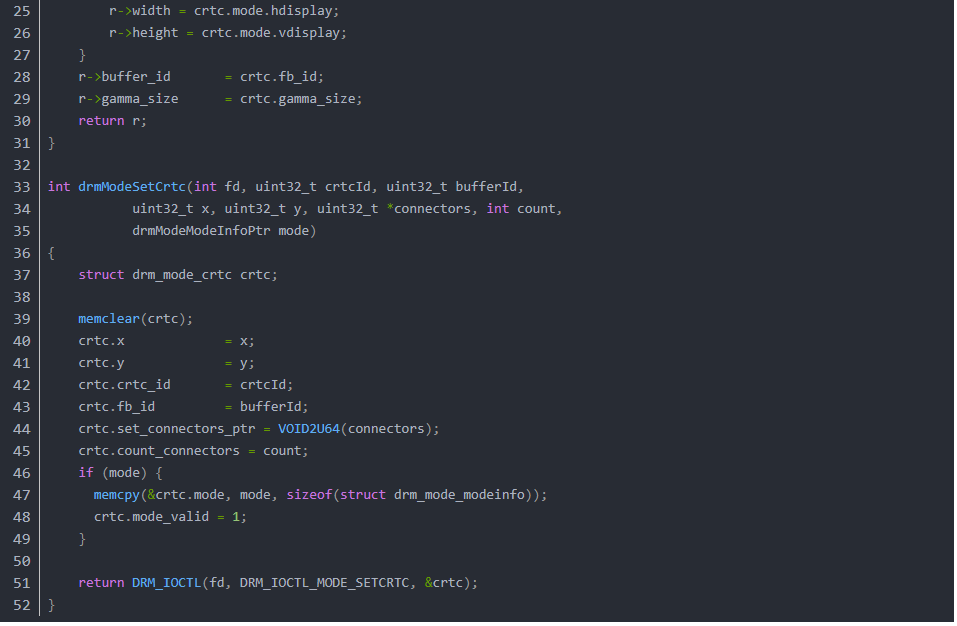


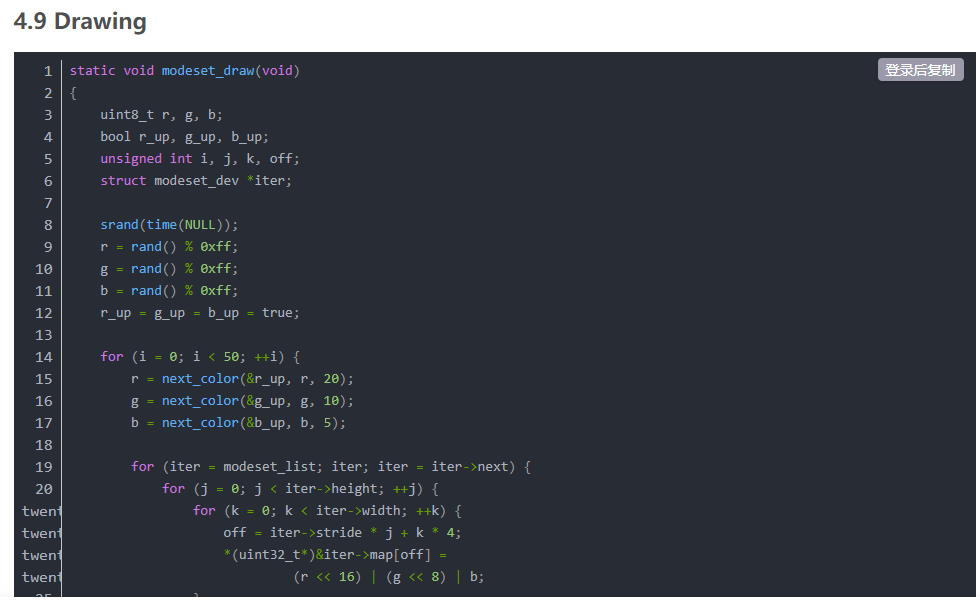


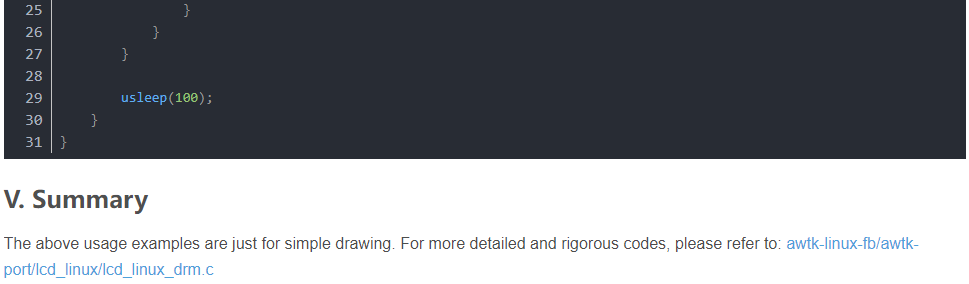












# **Code analysis of linux drm architecture**

# 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!

1. **Start with the connector and follow the trail**

Get the specific resources of connector 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 favourite

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

1. **Find a suitable crtc for the encoder**

Find crtc in the order of 3 in a chapter

1. **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

1. **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);

1. **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!

1. **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.

# **DRM Learning Introduction**

I have been learning DRM for more than a year. Due to the complex architecture , large amount of code, and few domestic references, it is difficult for beginners to learn. Therefore, I decided to share my learning experience with everyone, hoping to be helpful to the students who are learning DRM, and to exchange experience at the same time.

Since I am only responsible for the Display driver in my work, the DRM learning experience I share is limited to the Display part. I can't do anything about the GPU part. If you have relevant experience to share, please let me know in the message. I will visit often. Your blog, we all learn from each other.

**DRM**

DRM is the current mainstream graphics display framework of Linux. Compared with the FB architecture, DRM is more adaptable to the current increasingly updated display hardware. **For example, FB natively does not support multi-layer synthesis, does not support VSYNC, does not support DMA-BUF, does not support asynchronous update, does not support fence mechanism, etc.,** and DRM natively supports these functions. At the same time, DRM can manage GPU and Display drivers in a unified manner, making the software architecture more unified and facilitate management and maintenance.

DRM is divided into modules and can be simply divided into 3 parts: , ,libdrm KMS GEM

# insert image description here

1. **libdrm**

It encapsulates the underlying interface and provides a common API interface to the upper layer, mainly encapsulating various IOCTL interfaces.

1. **KMS**

Kernel Mode Setting, the so-called Mode setting, is actually two things - Update screen and set display parameters.

Update the screen : display the switching of buffers, the composition method of multiple layers, and the display position of each layer.

Set display parameters : including resolution, refresh rate, power state (sleep wakeup), etc.

1. **GEM**

The Graphic Execution Manager is mainly responsible for the allocation and release of the display buffer, and it is also the only place where the GPU uses DRM.

**Fundamental element**

There are many elements involved in the DRM framework, which are roughly as follows:

KMS: , , , , , , GEM: , ,**CRTC, ENCODER, CONNECTOR, PLANE FB VBLANK property DUM BPRIME fence**

# insert image description here

# 

Catalog (continuously updated)

This blog will be used as a directory summary of my DRM learning tutorial. In the future, I will share the learning process of the above knowledge points with you in the form of sample code, and constantly update the directory link, so stay tuned!

Simplest DRM application (single-buffer)

Simplest DRM application (double-buffer)

Simplest DRM application (page-flip)

Simplest DRM application (plane-test)

Advanced DRM application (Property)

DRM Application Advanced (atomic-crtc)

Advanced DRM application (atomic-plane)

Development History of DRM (Direct Rendering Manager)

DRM Driver Development (Beginning)

DRM Driver Development (VKMS)

About the origin of the names of DUMB and PRIME in DRM

DRM GEM driver development (dumb)

Detailed explanation of DRM driver mmap: (1) Preliminary knowledge

Detailed explanation of DRM driver mmap: (2) CMA Helper

LWN Translation: Introduction to Atomic Mode Setting Design (Part 1)

LWN translation: Introduction to Atomic Mode Setting Design (Part 2)

Translation: Mainline Explicit Fencing

dma-buf series:

dma-buf from shallow to deep (1) - the simplest dma-buf driver

dma-buf from shallow to deep (two) - kmap / vmap

dma-buf from shallow to deep (three) - map Attachment

dma-buf from shallow to deep (4) —— mmap

dma-buf from shallow to deep (5) —— File

dma-buf from shallow to deep (6) —— begin / end cpu\_access

dma-buf from shallow to deep (7) — — alloc page version

dma-buf from shallow to deep (8) — ION simplified version

LWN translation: DMA-BUF cache handling: Off the DMA API map (part 1)

LWN translation: DMA-BUF cache handling: Off the DMA API map (part 2)

References

Wiki: Direct Rendering Manager

wowotech: Linux graphic subsystem(2)\_DRI introduction

Boris Brezillon: The DRM/KMS subsystem from a newbie's point of view

Line·Piaoling Blog Park: Graphics System and AMD R600 Graphics Programming in Linux Environment (1)

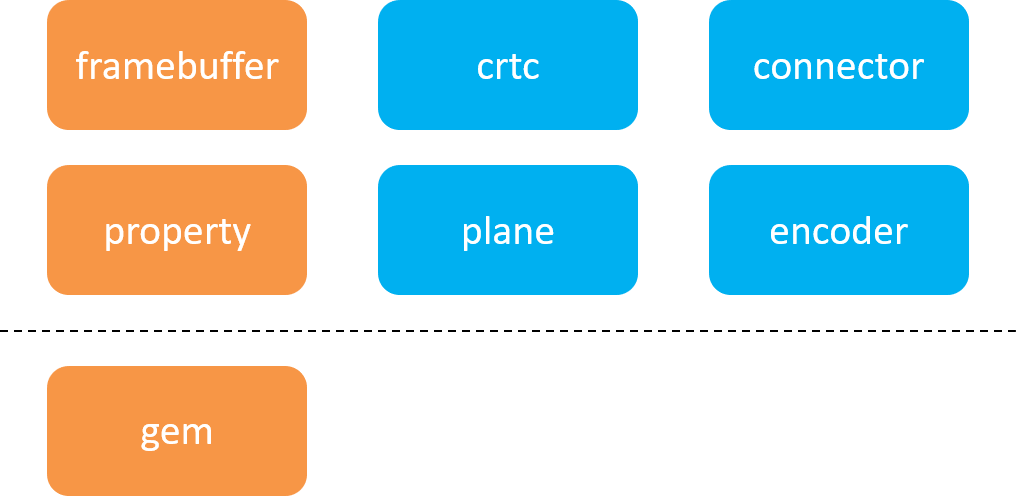
Younix Dirty Sheep CSDN Blog: Linux DRM (2) Basic Concepts and Features

# **DRM DRIVER DEVELOPMENT (OPENING)**

In the previous "Simplest DRM Applications" series of articles, we learned how to write DRM applications using the libdrm interface. Starting with this article, we'll enter a whole new world and learn how to write DRM drivers in kernel space.

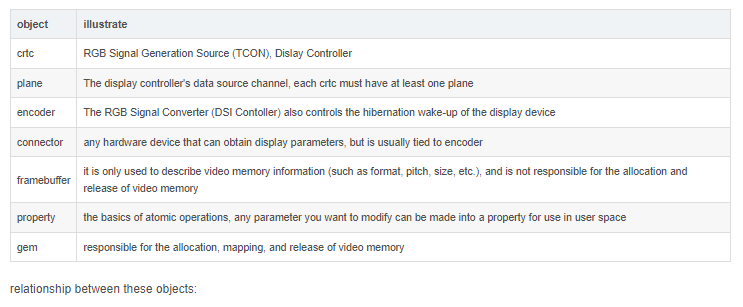
**Objects**

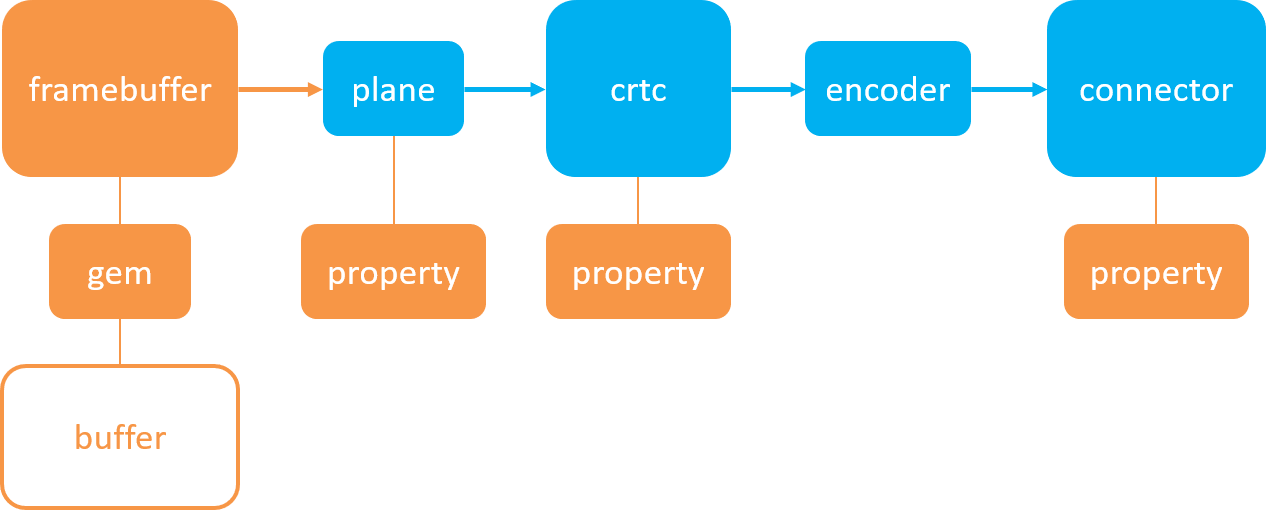
Before I start writing DRM drivers, it's important to give an introduction to objects inside DRM. Because these Objects are at the heart of the DRM framework, they are indispensable.



The blue part of the above figure is the abstraction of physical hardware, and the yellow part is the abstraction of software. Above the dotted line is , below the dotted line is . drm\_mode\_object drm\_gem\_object

I have briefly introduced these objects in the "Introduction to DRM (Direct Rendering Manager) Learning", and as the top priority of this article, I feel it is necessary to emphasize the concept of these objects again:

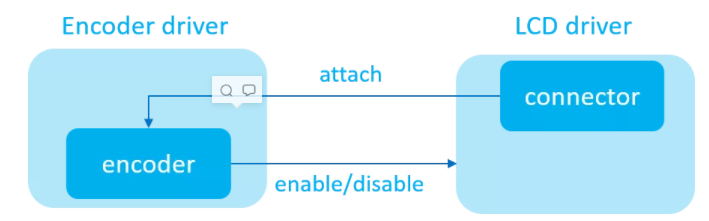




as you can see from the diagram above, **plane** is the link between framebuffer and crtc, and encoder is the link between crtc and connector. the gem deals directly with the physical buffer, not the framebuffer.

It should be noted that even if there is no actual hardware corresponding to the blue part in the above figure, these objects need to be implemented in the software driver, otherwise the DRM subsystem will not work properly.

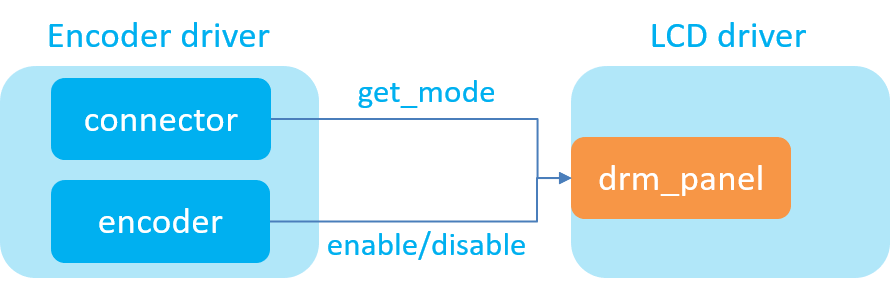
**drm\_panel** does not belong to the category of objects, it is just a collection of callback functions. But its existence reduces the coupling between the LCD driver and the encoder driver.

**Coupling generation:**

(1) **The main function of the connector is to obtain the display parameters, so the connector object will be constructed in the LCD driver.** However, the connector initializes with an encoder object attached, and this encoder object is often generated in another hardware driver, which is bound to produce some coupled code in order to access the encoder object.

(2) **In addition to playing the role of signal conversion, encoder also plays the role of notifying the display device to sleep and wake up.** Therefore, when encoder notifies the LCD driver to perform the corresponding enable/disable operation, it must call the global function exported by the LCD driver, which will inevitably produce a part of the coupling code.

To solve the coupling problem, the DRM subsystem provides developers with a structure that encapsulates the common interface for LCD access by connector & encoder. drm\_panel



**As a result, the coupling between the original Encoder driver and the LCD driver has become the "coupling" between the Encoder driver and the drm\_panel,** the drm\_panel and the LCD driver in the above figure, thus realizing the decoupling between the Encoder driver and the LCD driver.

tips:

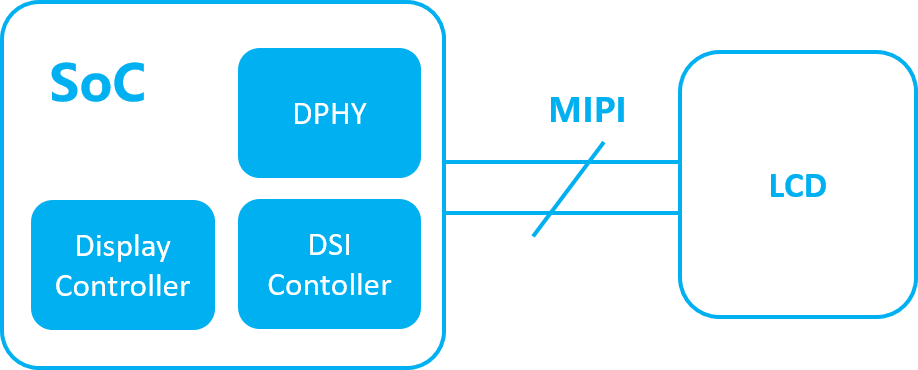
in order to facilitate driver design, encoder and connector are usually initialized in the same driver, that is, where encoder is, the connector is there.

**How to abstract hardware**

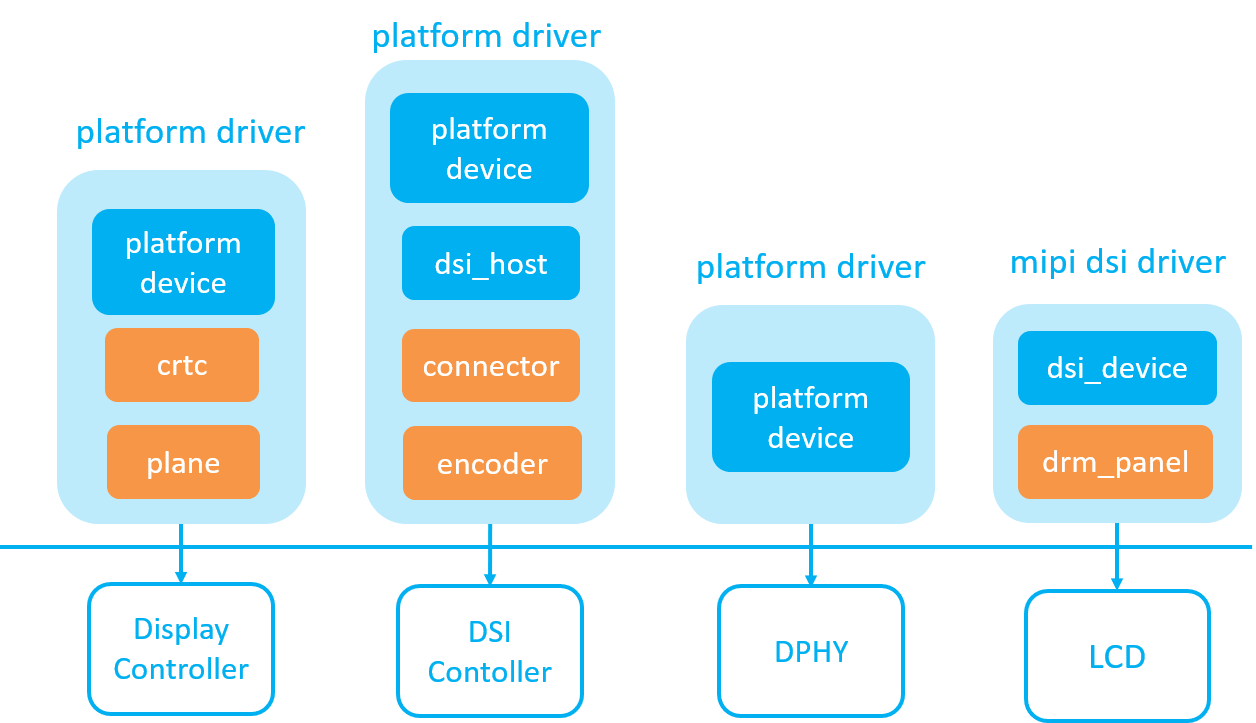
For beginners, what often confuses them is not the concept of objects in DRM, but how to establish the correspondence between these objects and the actual hardware. Because not all Display hardware can correspond well to plane/crtc/encoder/connector objects. Let's learn together how to abstract the display hardware into concrete DRM objects.

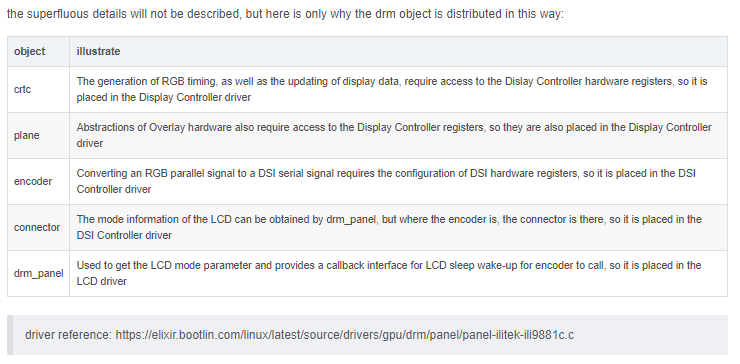
**MIPI DSI interface**

THE FOLLOWING FIGURE SHOWS A HARDWARE CONNECTION BLOCK DIAGRAM OF A TYPICAL MIPI DSI INTERFACE SCREEN:



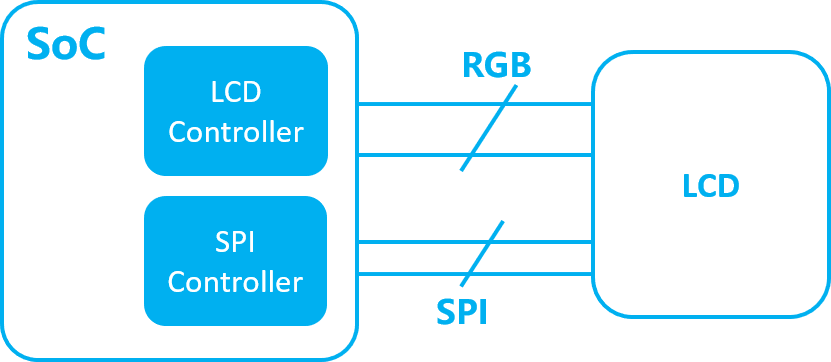
Its correspondence with DRM objects in software architecture is as follows:



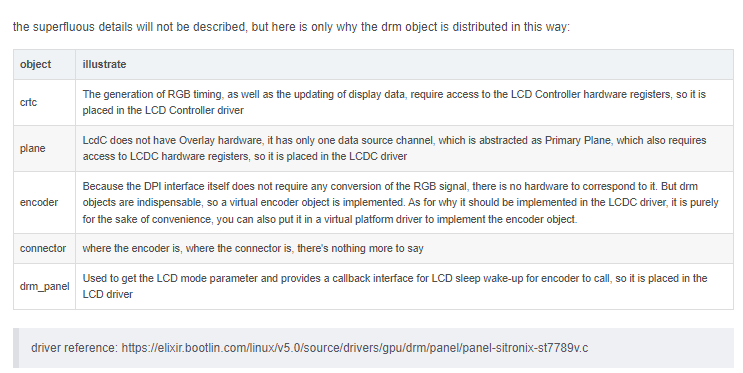


**MIPI DPI interface**

The DPI interface is also what we often call the RGB parallel interface, Video data is transmitted through the RGB parallel bus, and control commands (such as initialization, hibernation, wake-up, etc.) are transmitted through the SPI/I2C bus, such as the early S3C2440 SoC platform. The following figure shows a hardware connection block diagram of a typical MIPI DPI interface screen:



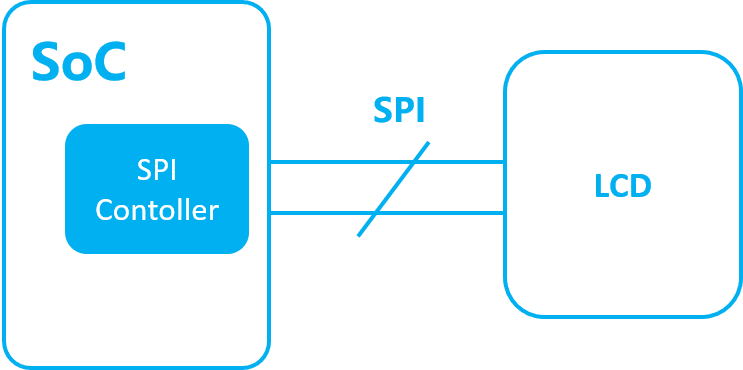
The software architecture of the hardware connection corresponds to the DRM object as follows:



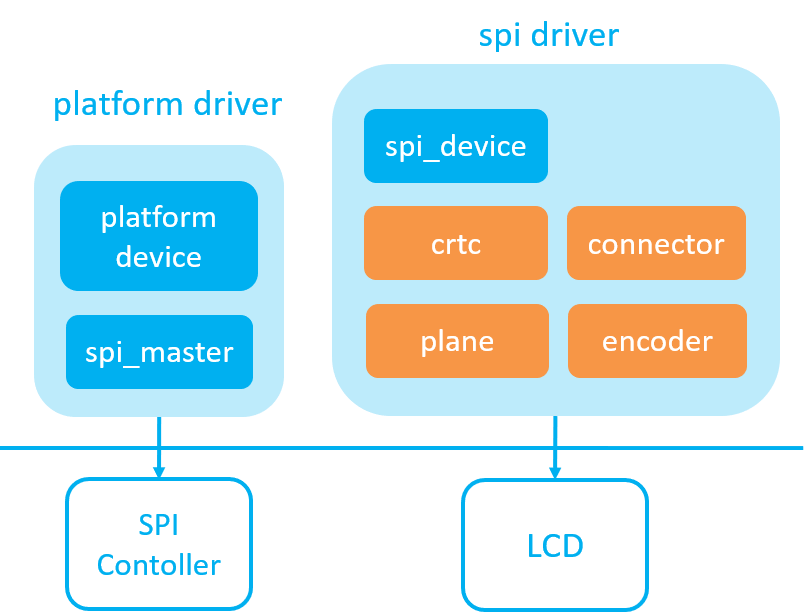
**MIPI DBI interface**

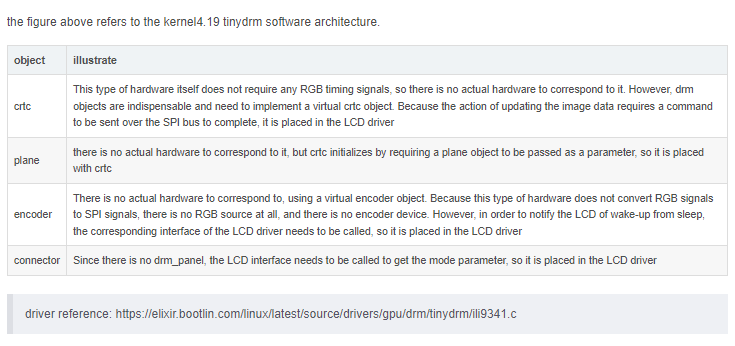
DBI INTERFACE IS WHAT WE USUALLY CALL MCU OR SPI INTERFACE SCREEN, THE VIDEO DATA AND CONTROL COMMANDS OF THIS SCREEN ARE TRANSMITTED THROUGH THE SAME BUS INTERFACE (I80, SPI INTERFACE), AND SUCH SCREENS MUST HAVE BUILT-IN GRAM VIDEO MEMORY, OTHERWISE THE SCREEN CANNOT MAINTAIN NORMAL DISPLAY.

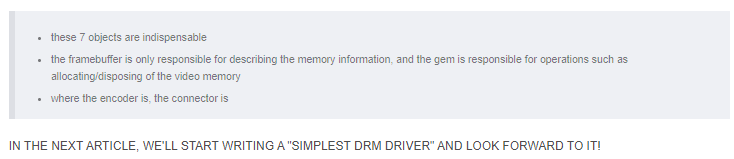
THE FOLLOWING FIGURE SHOWS A HARDWARE CONNECTION BLOCK DIAGRAM OF A TYPICAL DBI INTERFACE SCREEN:



The hardware connection corresponds to the DRM object in software architecture as follows:



**Summarize**



**SIMPLEST DRM DRIVER**

# **How the DRM driver creates the fb device**

how the drm driver creates the fb device

* + what is drm
  + how to use the drm interface
    - libdrm
    - fb device
  + this article discusses the issue of how the drm driver is virtually drm crts for fb devices
    - the device driver creates fbdev
    - complete the creation of the fb device
  + flow chart

**What is drm?**

1. drm is a Linux kernel display system driver framework. DRM is a display framework that manages GPUs
2. PROVIDES MEMORY MANAGEMENT, INTERRUPT HANDLING, AND DMA CONTROL AT THE KERNEL LEVEL
3. provides a unified operating interface for applications

**How to use the drm interface**

1. **libdrm**
2. **fb device**
3. **libdrm**

The kernel provides too many IOCTRL, libdrm is used to simplify programming to manage the current display, and modify the current mode to become KMS ( drm-kms - Kernel Mode-Setting)

With libdrm's powerful API interface, if the kernel supports PRIME API, you can also use the PRIME interface for more flexible memory operations.

Each DRM device provides access to manage which monitors and displays are currently used and what frames to be displayed. **This task is called Kernel Mode-Setting (KMS).**

Historically, this was done in user-space and called User-space Mode-Setting (UMS). Almost all open-source drivers now provide the KMS kernel API to do this in the kernel, however, many non-open-source binary drivers from different vendors still do not support this. You can use **drmModeSettingSupported**(3) to check whether your driver supports this. To understand how KMS works, we need to introduce 5 objects: **CRTCs, Planes, Encoders, Connectors and Framebuffers.**

**CRTCs**

A CRTC short for CRT Controller is an abstraction representing a part of the chip

that contains a pointer to a scanout buffer. Therefore, the number of CRTCs

available determines how many independent scanout buffers can be active at any

given time. The CRTC structure contains several fields to support this: a pointer

to some video memory (abstracted as a frame-buffer object), a list of driven

connectors, a display mode and an (x, y) offset into the video memory to support

panning or configurations where one piece of video memory spans multiple CRTCs. A

CRTC is the central point where configuration of displays happens. You select which

objects to use, which modes and which parameters and then configure each CRTC via

**drmModeCrtcSet(3)** to drive the display devices.

**Planes**

A plane respresents an image source that can be blended with or overlayed on top of

a CRTC during the scanout process. Planes are associated with a frame-buffer to

crop a portion of the image memory (source) and optionally scale it to a

destination size. The result is then blended with or overlayed on top of a CRTC.

Planes are not provided by all hardware and the number of available planes is

limited. If planes are not available or if not enough planes are available, the

user should fall back to normal software blending (via GPU or CPU).

**Encoders**

An encoder takes pixel data from a CRTC and converts it to a format suitable for

any attached connectors. On some devices, it may be possible to have a CRTC send

data to more than one encoder. In that case, both encoders would receive data from the same scanout buffer, resulting in a cloned display configuration across the connectors attached to each encoder.

**Connectors**

A connector is the final destination of pixel-data on a device, and usually

connects directly to an external display device like a monitor or laptop panel. A

connector can only be attached to one encoder at a time. The connector is also the

structure where information about the attached display is kept, so it contains

fields for display data, **EDID data, DPMS and connection status, and information**

**about modes supported on the attached displays**.

**Framebuffers**

Framebuffers are abstract memory objects that provide a source of pixel data to

scanout to a CRTC. Applications explicitly request the creation of framebuffers and

can control their behavior. Framebuffers rely on the underneath memory manager for low-level memory operations. When creating a framebuffer, applications pass a memory handle through the API which is used as backing storage. The framebuffer itself is only an abstract object with no data. It just refers to memory buffers that must be created with the drm-memory(7) API.

**Mode-Setting**

Before mode-setting can be performed, an application needs to call drmSetMaster(3) to

become DRM-Master. It then has exclusive access to the KMS API. A call to

**drmModeGetResources**(3) returns a list of **CRTCs, Connectors, Encoders and Planes.**

Normal procedure now includes: First, you select which connectors you want to use. Users

are mostly interested in which monitor or display-panel is active so you need to make sure

to arrange them in the correct logical order and select the correct ones to use. For each

connector, you need to find a CRTC to drive this connector. If you want to clone output to

two or more connectors, you may use a single CRTC for all cloned connectors (if the

hardware supports this). To find a suitable CRTC, you need to iterate over the list of

encoders that are available for each connector. Each encoder contains a list of CRTCs that

it can work with and you simply select one of these CRTCs. If you later program the CRTC

to control a connector, it automatically selects the best encoder. However, this

procedure is needed so your CRTC has at least one working encoder for the selected

connector. See the Examples section below for more information.

All valid modes for a connector can be retrieved with a call to drmModeGetConnector3 You

need to select the mode you want to use and save it. The first mode in the list is the

default mode with the highest resolution possible and often a suitable choice.

After you have a working connector+CRTC+mode combination, you need to create a framebuffer that is used for scanout. Memory buffer allocation is driver-depedent and described in **drm-memory**(7). You need to create a buffer big enough for your selected mode. Now you can create a framebuffer object that uses your memory-buffer as scanout buffer. You can do this with **drmModeAddFB**(3) and **drmModeAddFB2**(3).

As a last step, you want to program your CRTC to drive your selected connector. You can

do this with a call to **drmModeSetCrtc**(3).

**Page-Flipping**

A call to **drmModeSetCrtc**(3) is executed immediately and forces the CRTC to use the new

scanout buffer. If you want smooth-transitions without tearing, you probably use

double-buffering. You need to create one framebuffer object for each buffer you use. You

can then call **drmModeSetCrtc**(3) on the next buffer to flip. If you want to synchronize

your flips with vertical-blanks, you can use **drmModePageFlip**(3) which schedules your

page-flip for the next vblank.

**Planes**

Planes are controlled independently from CRTCs. That is, a call to **drmModeSetCrtc**(3) does

not affect planes. Instead, you need to call **drmModeSetPlane**(3) to configure a plane. This

requires the plane ID, a CRTC, a framebuffer and offsets into the plane-framebuffer and

the CRTC-framebuffer. The CRTC then blends the content from the plane over the CRTC

framebuffer buffer during scanout. As this does not involve any software-blending, it is

way faster than traditional blending. However, plane resources are limited. See

**drmModeGetPlaneResources**(3) for more information.

**Cursors**

Similar to planes, many hardware also supports cursors. A cursor is a very small buffer

with an image that is blended over the CRTC framebuffer. You can set a different cursor

for each CRTC with **drmModeSetCursor**(3) and move it on the screen with

**drmModeMoveCursor**(3). This allows to move the cursor on the screen without rerendering. If no hardware cursors are supported, you need to rerender for each

frame the cursor is moved.

**EXAMPLES**

Some examples of how basic mode-setting can be done. See the man-page of each DRM function for more information.

**CRTC/Encoder Selection**

If you retrieved all display configuration information via **drmModeGetResources**(3) as

**drmModeRes \*res,** selected a connector from the list in res->connectors and retrieved the

connector-information as **drmModeConnector \*conn** via **drmModeGetConnector**(3) then this example shows, how you can find a suitable CRTC id to drive this connector. This function takes a file-descriptor to the DRM device (see **drmOpen**(3)) as **fd**, a pointer to the retrieved resources as res and a pointer to the selected connector as conn. It returns an

integer smaller than 0 on failure, otherwise, a valid CRTC id is returned.

static int modeset\_find\_crtc(int fd, drmModeRes \*res, drmModeConnector \*conn)

{

drmModeEncoder \*enc;

unsigned int i, j;

/\* iterate all encoders of this connector \*/

for (i = 0; i < conn->count\_encoders; ++i) {

enc = drmModeGetEncoder(fd, conn->encoders[i]);

if (!enc) {

/\* cannot retrieve encoder, ignoring... \*/

continue;

}

/\* iterate all global CRTCs \*/

for (j = 0; j < res->count\_crtcs; ++j) {

/\* check whether this CRTC works with the encoder \*/

if (!(enc->possible\_crtcs & (1 << j)))

continue;

/\* Here you need to check that no other connector

\* currently uses the CRTC with id "crtc". If you intend

\* to drive one connector only, then you can skip this

\* step. Otherwise, simply scan your list of configured

\* connectors and CRTCs whether this CRTC is already

\* used. If it is, then simply continue the search here. \*/

if (res->crtcs[j] "is unused") {

drmModeFreeEncoder(enc);

return res->crtcs[j];

}

}

drmModeFreeEncoder(enc);

}

/\* cannot find a suitable CRTC \*/

return -ENOENT;

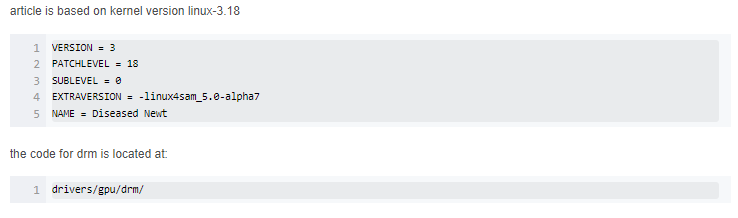
}

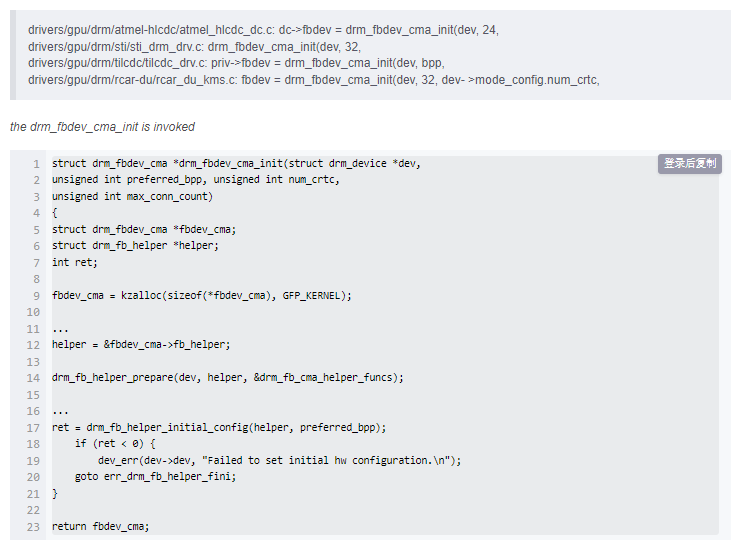
1. **fb device**

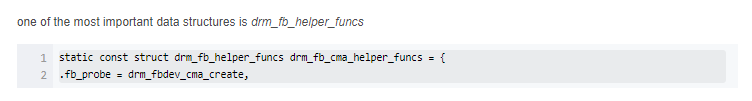
The drm driver can emulate an fb device, default CRTC by default, and for more information about fb device, you can refer to the kernel framebuffer documentation, which is the basis for most Linux systems.

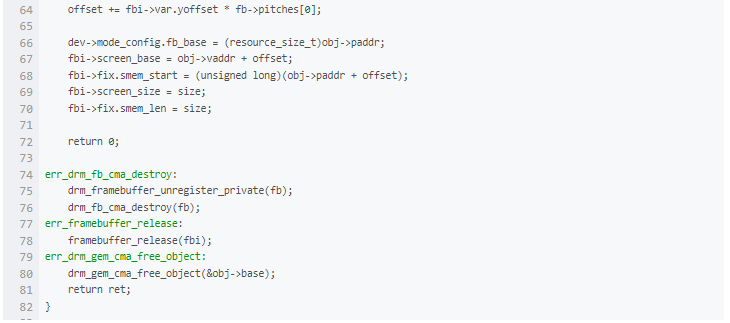
* The X Server, a display service for Linux desktop systems
* Android gralloc, Android display HAL

**The problem discussed in this article is how the drm driver virtualizes drm crts as fb device**

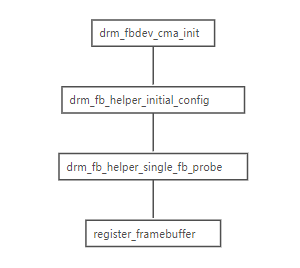


**1. Device driver creates fbdev**



**2. Complete the creation of the fb device:**

**flow chart**



# **[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

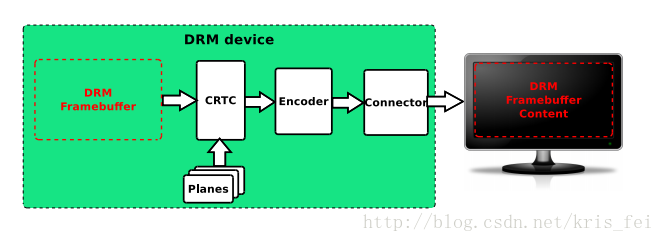
Generic Access Hardware Interface

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:

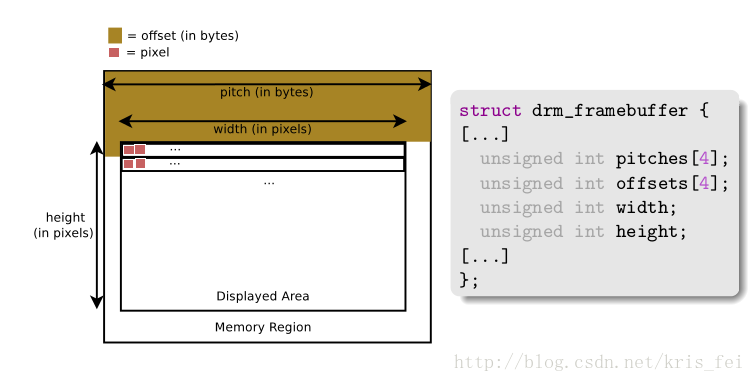


**Framebuffer:**

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:

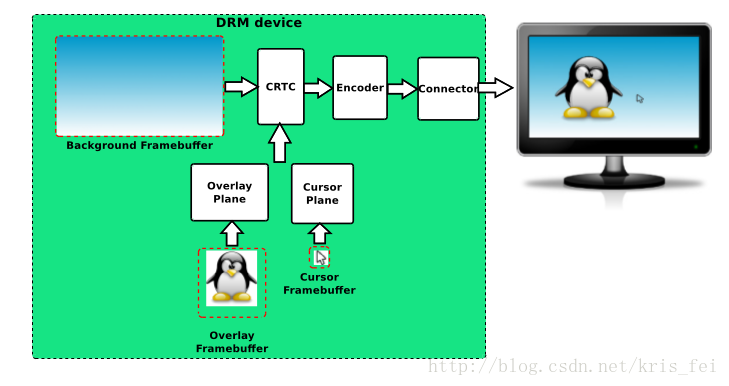
DRM\_PLANE\_TYPE\_PRIMARY: must have, due to display background or image content

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:

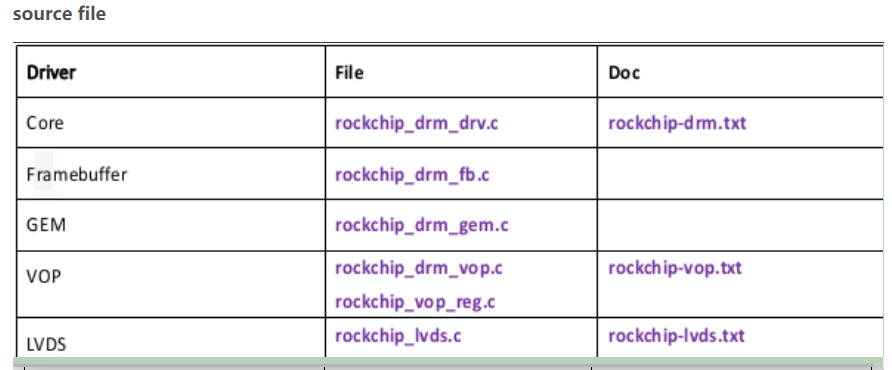
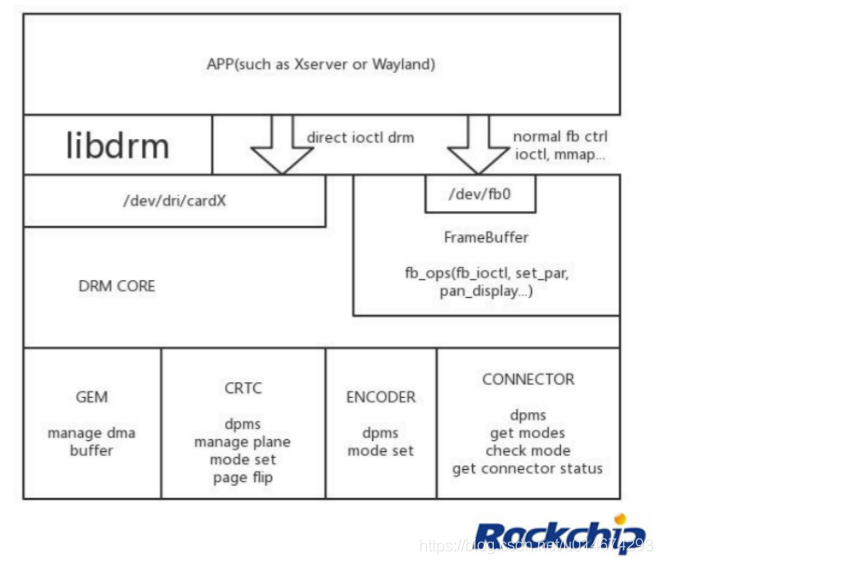
brezillon-drm-kms.pdf

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.

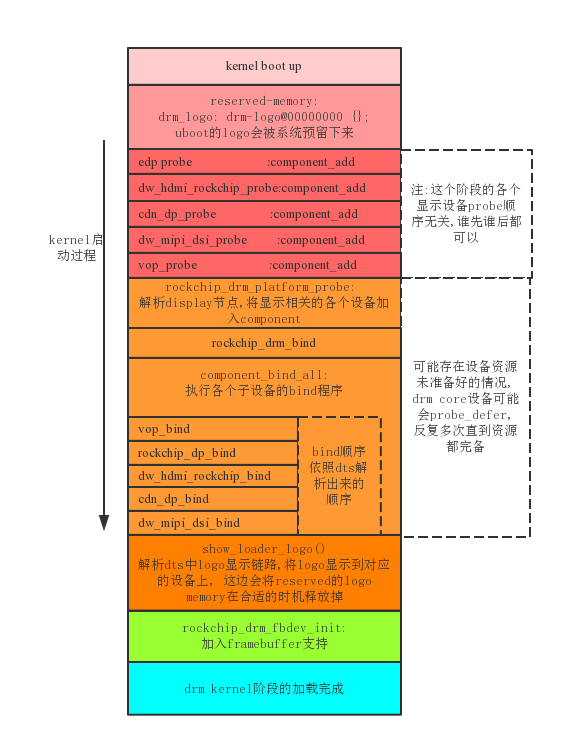
Because there are many devices attached to drm, the boot sequence often causes various problems:

* 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

1. **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

1. **drm driver**

code path:

static struct drm\_driver rockchip\_drm\_driver = {  
.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\_dumb\_destroy,  
.prime\_handle\_to\_fd = drm\_gem\_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,undefined  
};

1. **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),  
}

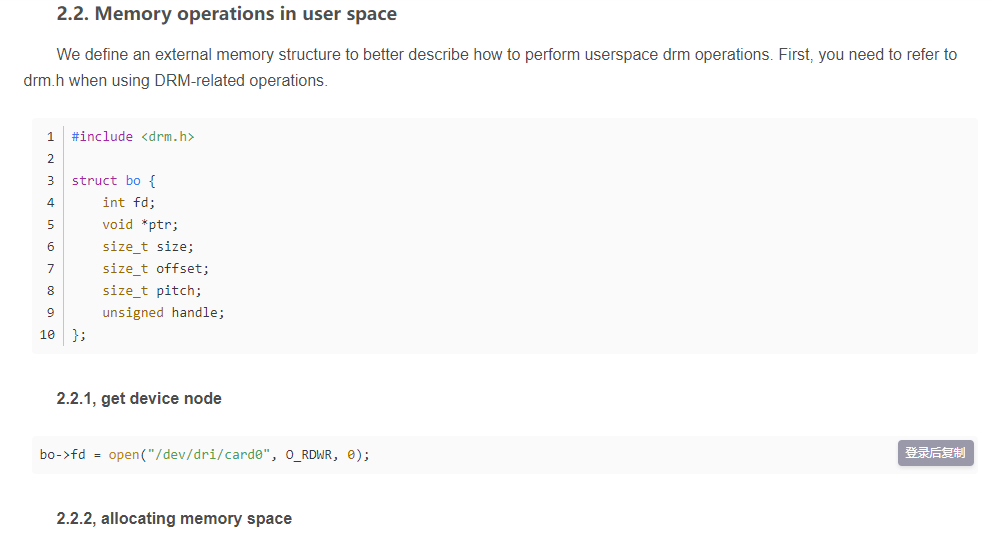
static const struct vop\_win\_phy rk3368\_win23\_data = {undefined  
        .enable = VOP\_REG(RK3368\_WIN2\_CTRL0, 0x1, 4),  
}

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.

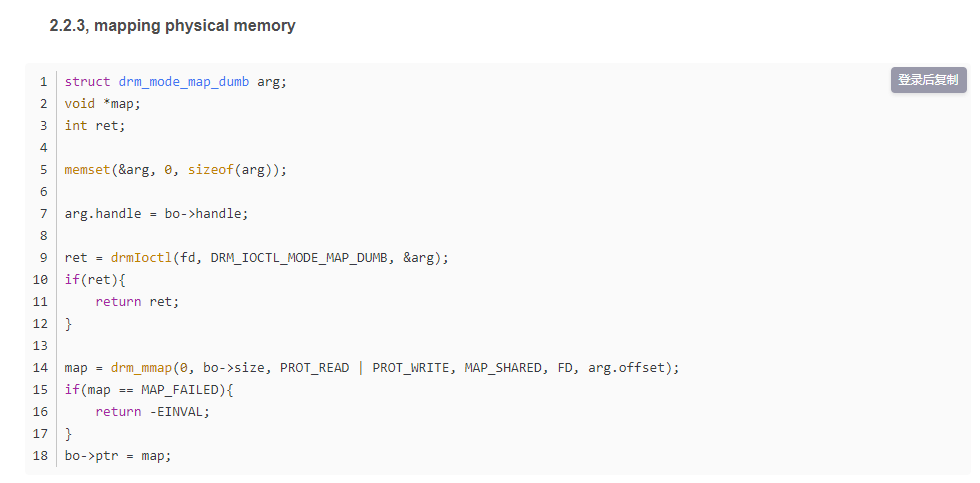
### 2.1. Device file cardX

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.

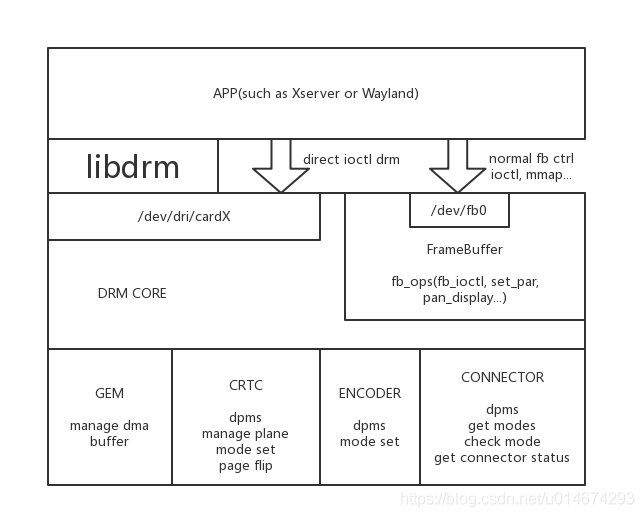
X Servers or other Display Servers are usually the DRM-Master processes for the DRM devices they manage. When the DRM device is started, these Display Servers open the device node and obtain DRM-Master permissions until the device is closed.

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.

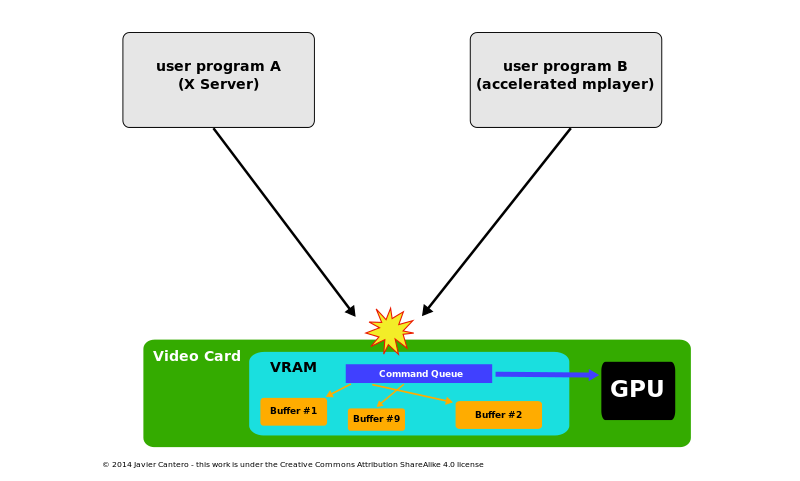
step:

* **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.

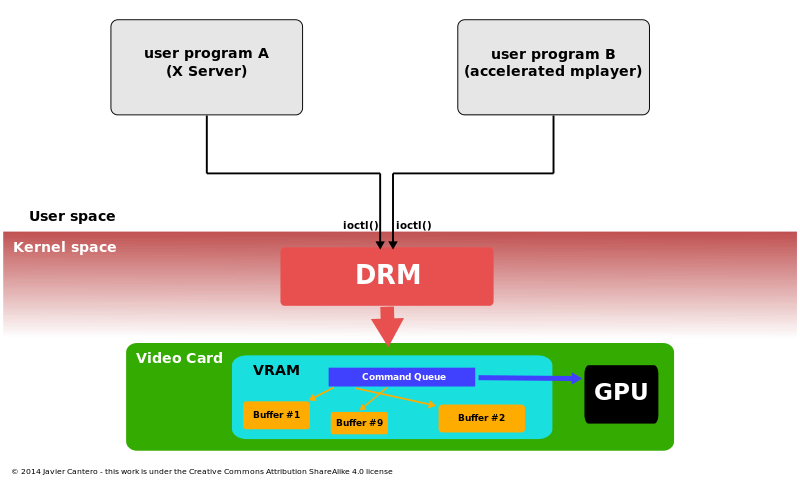
**The architecture of DRM in the source code (pictured from Mark.Yao):**



**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.**

# **[RK3399] DRM Framebuffer in [Android 7.1] Display**

**OS: Android 7.1**

**Board: Firefly-RK3399**

**Kernel: v4.4.55**

**the display cache used by the previous product was fb, that is, framebuffer;**

**after the introduction of drm, the display cache used drm fb.**

**The default use of DRM in the display HAL layer can be drawn from the Android.mk.**

**hardware/rockchip/hwcomposer/Android.mk:**

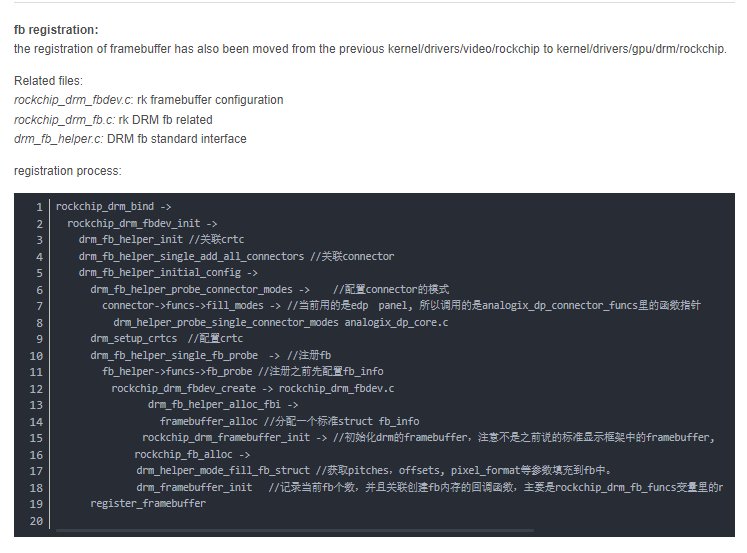
****

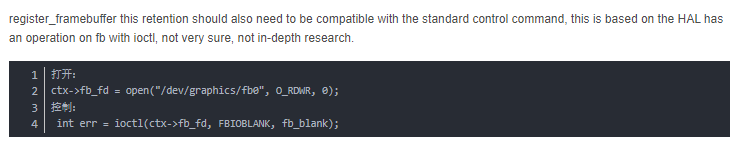
**Addition of drm devices:**

****

**After the addition is complete, you will see the following files in /sys/class/drm: card0 card0-DSI-1 controlD64 renderD128 version**

****

****

****

# **[RK3399][Android7.1] DRM and Framebuffer registration process in Display**

**The kernel driver provides two methods for user space to complete the display function.**

**One is through DRM, which can be done through ioctl, or through the DRM library libdrm encapsulated in user space.**

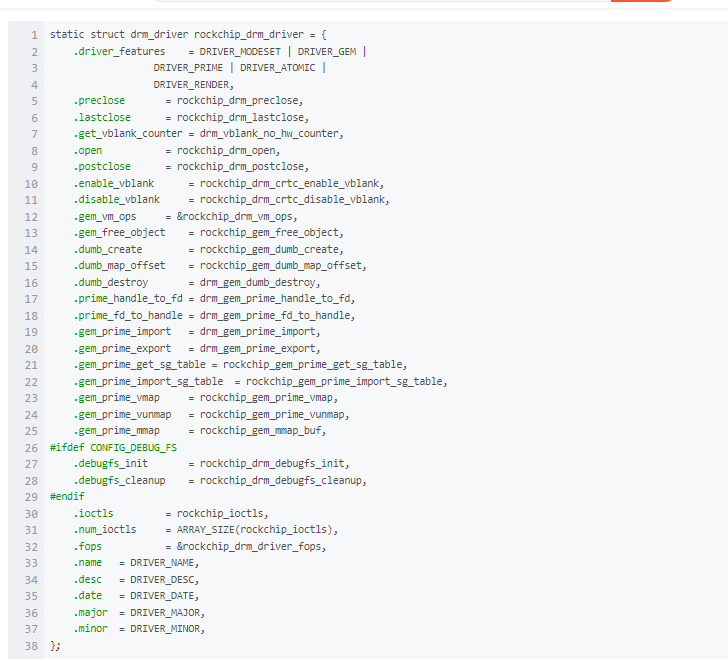
**Another way is to do it through the standard framebuffer in the past, which is also done through ioctl.**

**Via DRM:**

**Driver file: rockchip\_drm\_drv.c**

**Registration process:**

****

****

**The file\_operations exposed to userspace are**

****

**Some callbacks in rockchip\_drm\_driver are called through rockchip\_drm\_driver\_fops, such as**

**drm\_open -> drm\_open\_helper -> dev->driver->open -> rockchip\_drm\_open**

**via framebuffer:**

**The registration of framebuffer has also been moved from the previous kernel/drivers/video/rockchip to kernel/drivers/gpu/drm/rockchip.**

**Related documents:**

**rockchip\_drm\_fbdev.c: rk framebuffer configuration**

**rockchip\_drm\_fb.c: rk DRM fb related**

**drm\_fb\_helper.c: DRM fb standard interface**

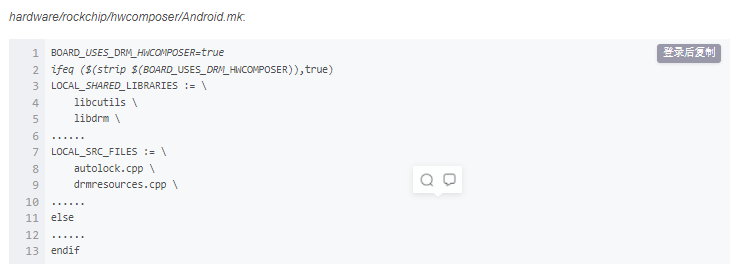
**Registration process:**

****

**The file\_operations exposed to userspace are**

****

**DRM is used by default in the display HAL layer, which can be concluded from Android.mk.**

****

**The libdrm library is also used, located under external/libdrm.**

# **Code analysis of linux drm architecture and linux drm architecture**

summary:

DRM is a direct render manager under linux, specifically a graphics card driver architecture (how to play with drivers?). Encapsulates functions into standard interfaces such as open/close/ioctl, which applications call to drive devices.

AS A GRAPHICS CARD, THE MOST BASIC FUNCTION IS TO OUTPUT THE USER'S DRAWING TO THE DISPLAY, HOW TO ACHIEVE DRM, FIRST LOOK AT DRM TO GIVE YOU A FEW BASIC ELEMENTS OF "THIS THING":

**FrameBuffer, CRTC, Encoder, Connector, and then to the display**

 1 FrameBuffer

For the computer, 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 formatting before drawing, for example, I can specify what kind of color mode (RGB24, I420, YUUV, etc.), how big the resolution is, and what parameters, then you have to go to the drawing site to see :p

2 DRAWING SITE (CRTC)

Shorthand translated is the negative camera tube context, in the DRM CRTC represents the context of the display output, first CRTC refers to a FrameBuffer address, connected to an Encoder. How do the two of them communicate? This is what the display mode (ModeSet) to do, ModeSet includes like the previously mentioned color mode, as well as the display timing (timings, ModeLines, etc. all represent this Italian) and so on, usually the timing can be expressed as follows

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

pixel clock horizontal front sweep horizontal back sweep horizontal back sweep horizontal sync head horizontal effective length vertical front sweep vertical back sweep vertical back scan vertical active length

A CRTC can connect multiple Encoders, what to do, to achieve the function of copying the screen.

3 Output Converter (Encoder)

Thinking about CRT is complicated enough, our graphics cards are very good at connecting a variety of different devices, obviously the output requires different signal converters, converting the pixels of memory into the signals needed by the display (DVID, VGA, YPbPr, CVBS, etc.... ）

4 Connector

DOES NOT REFER TO THE PHYSICAL LINE, BACK TO DRM WHICH IS AN ABSTRACT DATA STRUCTURE THAT REPRESENTS THE CONNECTED DISPLAY DEVICE, FROM WHICH WE CAN GET THE EDID OF THE DEVICE, THE DPMS CONNECTION STATUS, ETC.

5 Planner

Hey, how about one more. I'm also very bored, the above East east is not enough to work? In fact, many innovations often stem from people's dissatisfaction with the real world. You have to watch words to learn, you have to watch movies and games, and you have to watch movies while chatting. Here are two concepts in opposition, such as the graphics mode of small-scale updates such as text interaction, and the video mode with the ultra-fast full-frame update speed, which pulls the use of graphics cards to two extremes.

So the concept of Planner works well, it provides a green channel for video refresh, even not with the graphics, even a new layer (or overlay), which can be superimposed on or under the Graphic, and even zoomed...

The documentation says that Planner is also on FrameBuffer, which is okay, here we see that the CRTC to show the East and West should be a combination (blending).

 after understanding the concepts, the next article analyzes the specific data structures and interfaces.

 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 previous article introduced the architecture of linux's display driver drm, which is reviewed in a certain order here:

1 I connect the monitor to the DVI output of the graphics card, which abstracts into a Connector

2 On the Connector of the DVI, the driver will assign encoders for the DVI signal, and if not, all available encoders will be found on the connector resource

3 encoder is for image scanning field crtc services, the driver may assign crtc to encoder, or find available from encoder's possible\_crtc

4 crtc scans the field to configure fb to display the physical memory area of the image

5 fb -> crtc -> encoder - > connector after this relationship is bound, the drawing work has begun, you can write on fb and get displayed immediately!

6 However, to avoid image tearing, you can build multiple fb (buffers) to refresh the drawing through the pageFlip operation.

7 of course, there is also a plan for video refresh, and plane must also be bound to crtc to work.

2. summarize the use of the + drm api:

THE API uses drm-howto that references David Herrmann <dh.herrmann@googlemail.com> as well as the modetest program that comes with weston and 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 the logical, how to use the api, look down:

1. **first open the drm driver module and get all the resources**

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

drmModeRes res = drmModeGetResources(fd);

**what's in the res, res tells a couple of connectors, a few encoders, a few crtc, etc.** and their ids, completely out of set!

1. **start with the connector and follow the vine to touch the melon**

get the specific resources for connector first

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

there are two important parts of the conn resource, **one is to read out the "modes" of the monitor such as 1920x1080@60 through the cable**, of course, you have to choose a favorite

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

1. **find the right crtc for encoder**

find crtc in the order of 3 in a chapter

1. **create fb for crtc**

The example of drm modetest is written in detail, ARGB can directly **drmModeAddFB**, multiplaneal fb can use **drmModeAddFB2**

1. **binding**

core quads < 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);

1. **pageFlip**

  extern int drmModePageFlip(int fd, uint32\_t crtc\_id, uint32\_t fb\_id,  
                 uint32\_t flags, void \*user\_data);

this play has to be well written down,

1. **First poll** (drm\_fd) can receive a **pollIN** message from drm, which is nothing more than **two kinds of VBLANK and pageFlip complete**
2. Secondly, **after receiving the message, you must call drHandleEvent(drm\_fd, &evctx); To process the message, remember that you must fill in your pageFlip handler into evctx**.
3. **What's going on in the pageFlip handler, pageFlip one frame is over, of course, pageFlip next frame!**
4. **plane**

The gameplay of the plane is not understood, tried the next, using the multi-buffer tuning drmModeSetPlane to rotate the page can achieve video playback, but encountered two problems:

One is the timing of SetPlane, how to wait for a display's field to sync? Looking at the code of vblank feels like vblank is syncing for one graphics card, but I have multiple monitors?

The second is the running time of SetPlane, my i3 cpu execution used 22 ~ 23 ms at a time, inexplicably.

I thought that the plan's yuv channel can save resources, there are two major problems in the can not be done, pageFlip fb can only be the same color space as the monitor, and is usually ARGB !!

**linux ModeSetting learning**

**1. data and data structure:**

**Connector:** Represents a socket on the graphics card, and several Connectors indicate that there will be several outputs.

**CRTC:** Crt controller, which is responsible for transferring data from the frame cache to the Connector, and the data passes through Encoder before being transferred to the Connector. **Frame Caching --> CRTC --> Encoder --> Connector --> display.**

**FrameBuffer:** Here frame caching does not refer to a certain area of video memory, but a concept abstracted by Linux DRM, expressed in fb\_id.

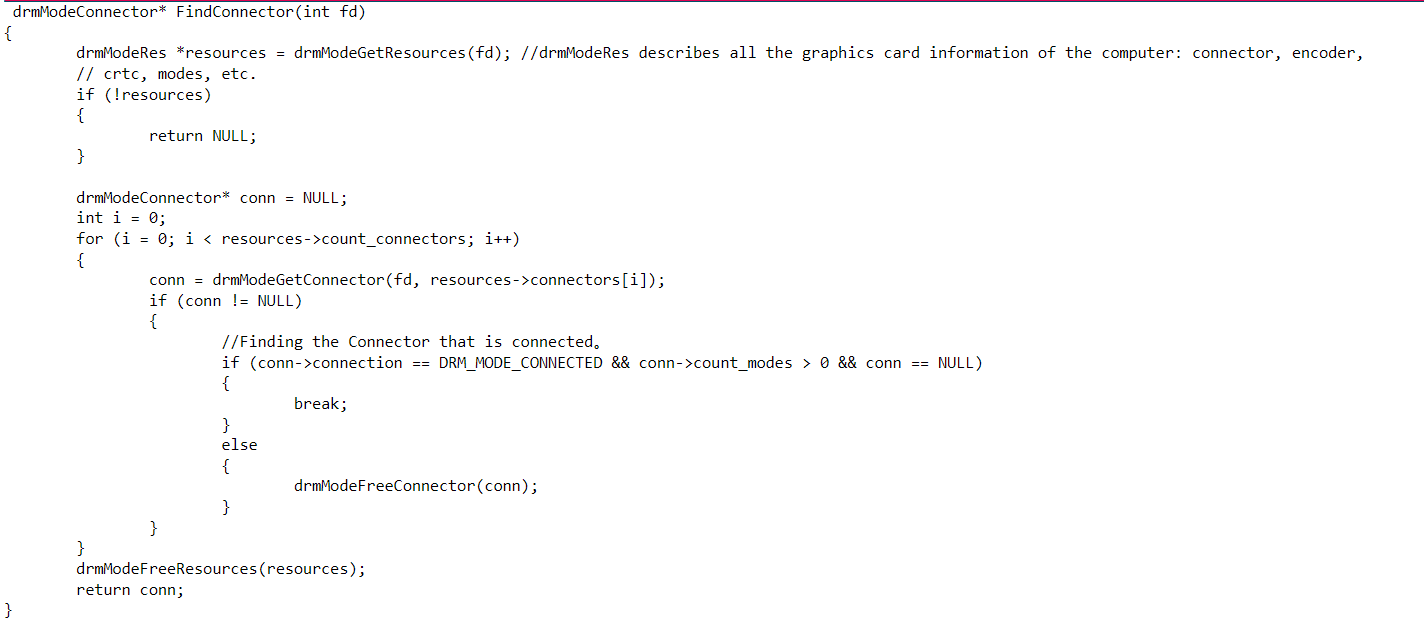
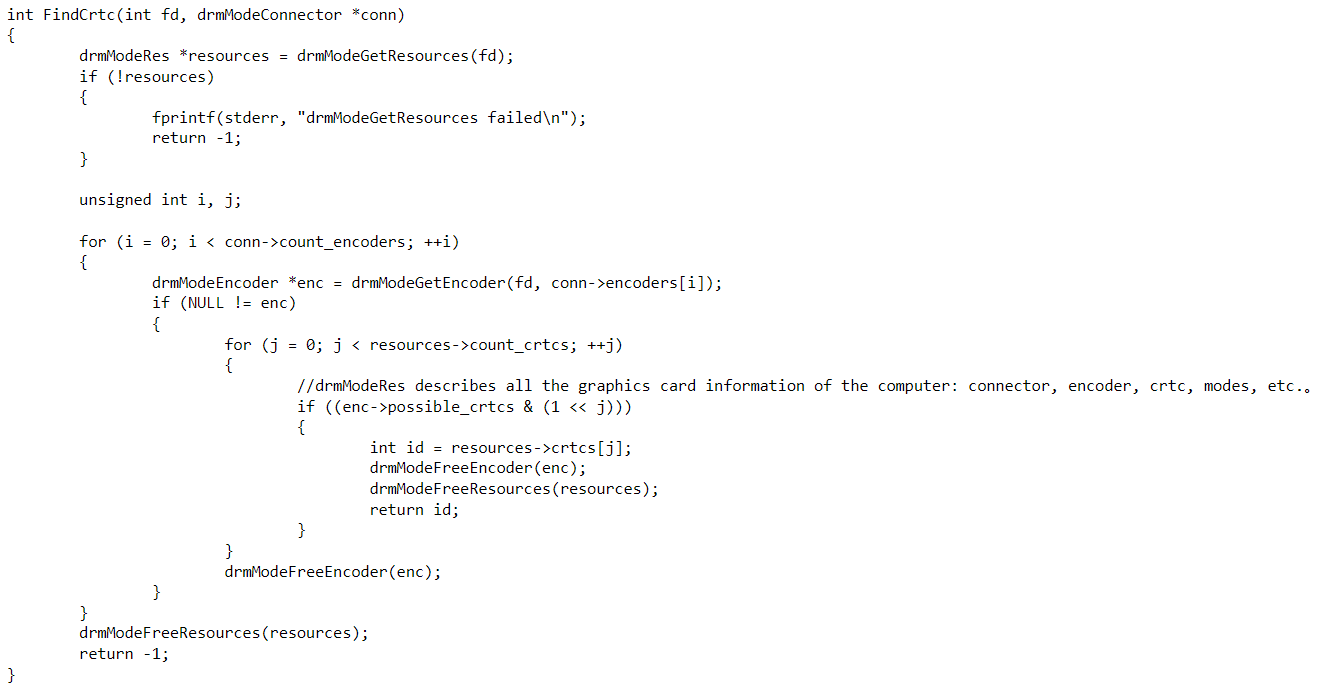
**drm\_mode\_create\_dumb:** the parameter of drmIoctl, which means that the request kernel creates a cache. This is the real memory block, which can be mapped into the virtual memory of the program using mmap. The creation of a Framebuffer must specify a cached id.

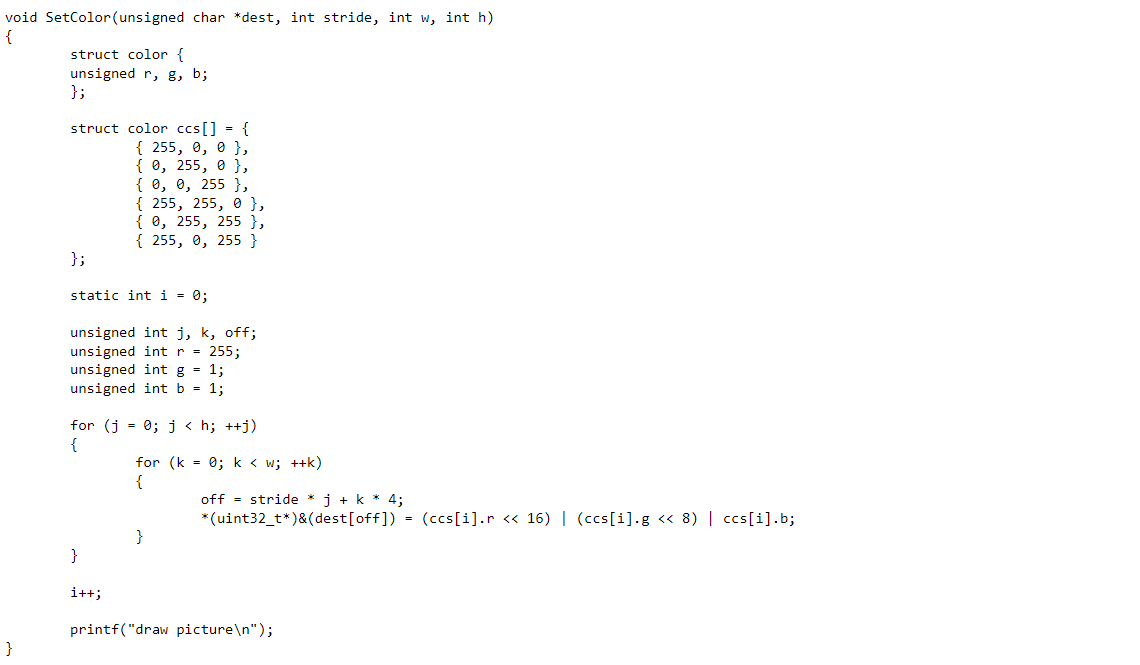
**2. function:**

**drmModeSetCrtc:** One of the most core functions that is responsible for establishing an association from the frame cache to the Connector. Only by calling it can the monitor display the data in the cache.

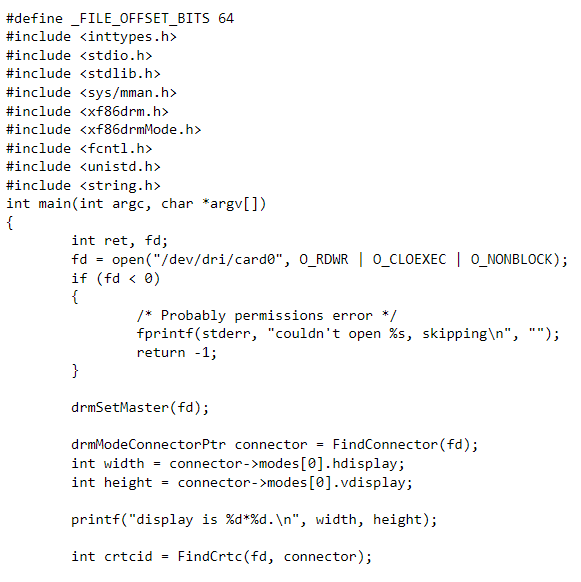
**drmModePageFlip:** The literal translation is to turn the page, the author's understanding is that the drm\_mode\_create\_dumb type of cache is actually a double cache, only after calling this function, crtc can transmit the data previously written to the monitor.

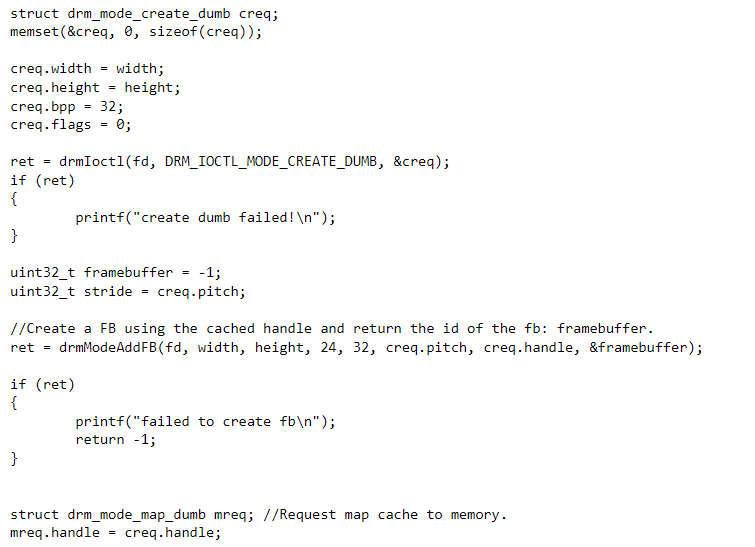
**3. code:**

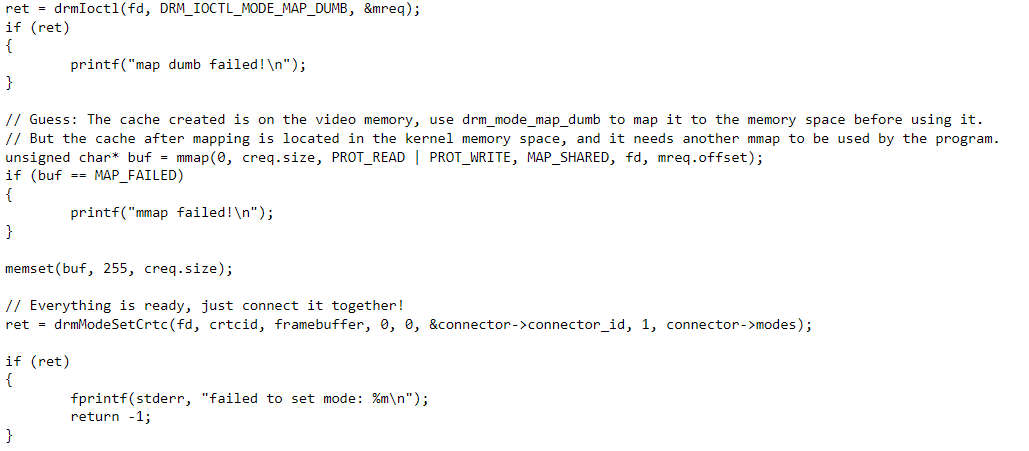
**Locate the Connector that is in the connected state** **1. Look for The Crtc that matches the Connector:**

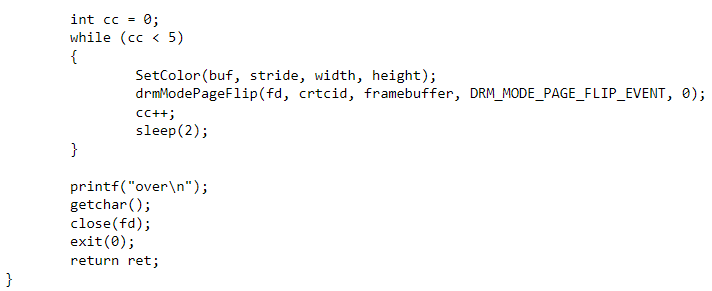
**2. draw a full-color diagram:** 

**3. main function:**









## **dma-buf subsystem(一)\_Jun's blog**

* Linux kernel 5.0 Document

# **1 Overview**

* Two definitions of dma-buf subsystem as the following comments described.
* (1) The **dma-buf subsystem** provides the framework for sharing buffers for hardware (DMA) access across multiple device drivers and subsystems, and for **synchronizing asynchronous hardware access**. This is used, for example, by **drm “prime”** multi-GPU support.
* (2) The **dma-buf framework** provides a generic method for sharing buffers between multiple devices. Device drivers that support dma-buf can export a DMA buffer to userspace as a file descriptor (known as the **exporter** role), import a DMA buffer from userspace using a file descriptor previously exported for a different or the same device (known as the **importer** role), or both.
* The **three main components** of dma-buf subsystem are:
  + (1) **dma-buf**, representing a **sg\_table** and exposed to userspace as a **file descriptor** to allow passing between devices.
  + (2) **fence**, which provides a mechanism to signal when one device as finished access.
  + (3) **reservation**, which manages the shared or exclusive fence(s) associated with the buffer.

# **2 Use the dma-buf**

## **2.1 exporter and importer**

* Any device driver which wishes to be a part of DMA buffer sharing, can do so as either the ‘exporter’ of buffers, or the ‘importer’ of buffers. Say a driver A wants to use buffers created by driver B, then we call B as the exporter, and A as buffer-user/importer.
* **The exporter**
  + implements and manages operations in struct dma\_buf\_ops for the buffer,
  + allows other users to share the buffer by using dma\_buf sharing APIs,
  + manages the details of buffer allocation, wrapped in a struct dma\_buf,
  + decides about the actual backing storage where this allocation happens, and takes care of any migration of scatterlist - for all (shared) users of this buffer.
* **The buffer-user/importer**
  + is one of (many) sharing users of the buffer.
  + doesn’t need to worry about how the buffer is allocated, or where.
  + and needs a mechanism to get access to the scatterlist that makes up this buffer in memory, mapped into its own address space, so it can access the same area of memory. This interface is provided by struct dma\_buf\_attachment.
* Any exporters or users of the dma-buf buffer sharing framework must have a ‘select DMA\_SHARED\_BUFFER’ in their respective Kconfigs.

## **2.2 DMA buffer file descriptor**

* Mostly a DMA buffer file descriptor is simply an opaque object for userspace, and hence the generic interface exposed is very minimal.
* userspace must have a way to request O\_CLOEXEC flag be set when the dma-buf fd is created.

## **2.3 Basic Operation and Device DMA Access**

* The exporter defines his exporter instance using DEFINE\_DMA\_BUF\_EXPORT\_INFO()and calls dma\_buf\_export() to wrap a private buffer object into a dma\_buf. It then exports that dma\_buf to userspace as a file descriptor by calling dma\_buf\_fd().`
* Userspace passes this file-descriptors to all drivers it wants this buffer to share with: First the filedescriptor is converted to a dma\_buf using dma\_buf\_get(). Then the buffer is attached to the device using \*\*dma\_buf\_attach()\*\*.
* Once the buffer is attached to all devices userspace , then it can initiate DMA access to the shared buffer. In the kernel this is done by calling dma\_buf\_map\_attachment() and dma\_buf\_unmap\_attachment().

## **2.4 CPU Access to DMA Buffer Objects**

### 2.4.1 fallback opeartions

* Fallback operations in the kernel, for example when a device is connected over USB and the kernel needs to shuffle the data around first before sending it away. Cache coherency is handled by braketing any transactions with calls to dma\_buf\_begin\_cpu\_access() and dma\_buf\_end\_cpu\_access() access.

"Strain operations in the kernel, such as devices connected via USB, the kernel needs to de-randomize the data before sending, then cache coherence is required for wrapping transfers, it (cache coherence) is through dma\_buf\_begin\_cpu\_access() and dma\_buf\_end\_cpu\_access() realized.

dma\_buf\_begin\_cpu\_access()

braketing any transactions

dma\_buf\_end\_cpu\_access()

2.4.2 vmap

* Since for most kernel internal dma-buf accesses need the entire buffer, a vmap interface is introduced.

// dma\_buf\_vmap - Create virtual mapping for the buffer object into kernel address space.

// This call may fail due to lack of virtual mapping address space

void \*dma\_buf\_vmap(struct dma\_buf \*dmabuf)

void dma\_buf\_vunmap(struct dma\_buf \*dmabuf, void \*vaddr)

# LWN Translation: DMA-BUF cache handling: Off the DMA API map (part 1)

Disclaimer: This article is not original, just a translation!  
Original text: <https://lwn.net/Articles/822521/>  
Author: John Stultz (Linaro member, kernel timekeeping maintainer)  
Note: This article requires background knowledge of [DMA](https://so.csdn.net/so/search?q=DMA&spm=1001.2101.3001.7020) -BUF. If you don’t know DMA-BUF yet, it is recommended to read it first The translator's own ["dma-buf from shallow to deep"](https://blog.csdn.net/hexiaolong2009/category_10838100.html)[series](https://blog.csdn.net/hexiaolong2009/article/details/102596772) Chapter 3 "map attachment" [and](https://blog.csdn.net/hexiaolong2009/article/details/102596825) Chapter 6 "begin / end cpu\_access" .

In the [previous](https://blog.csdn.net/hexiaolong2009/article/details/106745686) article, I introduced some background knowledge about ION, DMA-BUF Heap, DMA [API](https://so.csdn.net/so/search?q=API&spm=1001.2101.3001.7020) , and the basic concept of CPU Cache "ownership", and finally described how DMA-BUF is from the perspective of traditional DMA API Deals with cache synchronization issues. The article concludes by discussing why traditional DMA APIs perform so poorly on modern mobile platforms. This article will discuss with you how to make the DMA-BUF exporter avoid unnecessary cache operations, and give some general suggestions on how to improve these methods.

From the point of view of the DMA API: by calling *dma\_buf\_map\_attachment()* , the ownership of the DMA-BUF is transferred to the DMA device, and by calling *dma\_buf\_unmap\_attachment()* , the ownership is returned to the CPU, and each time these two functions are called, the cache correlation is executed operation. Although such a sequential operation can ensure the correctness of CPU Cache processing, for buffer pipeline operations involving multiple DMA devices, the CPU does not actually participate in accessing these buffers at all, and each cache map and unmap operations increase It can cause significant performance issues.

**Who owns the buffer?**

To avoid these redundant cache operations, the DMA-BUF interface allows some rules of the DMA API to be reversed. It should be noted that the DMA API assumes that the CPU is the natural owner of all memory, and this reverse rule only needs to be considered during a DMA transfer (the ownership of the buffer has been explicitly transferred to the DMA device). [*The DMA-BUF interface requires the CPU to call dma\_buf\_begin\_cpu\_access()*](https://elixir.bootlin.com/linux/v5.7.1/source/drivers/dma-buf/dma-buf.c#L1064) before accessing the DMA-BUF, and call *[dma\_buf\_end\_cpu\_access](https://elixir.bootlin.com/linux/v5.7.1/source/drivers/dma-buf/dma-buf.c" \l "L1100)* () after the access ends . If the CPU wants to access the buffer from user space, it can use the DMA\_BUF\_IOCTL\_SYNCioctl() command to initiate a call to begin/end cpu\_access.

Special interface:

* *dma\_buf\_begin\_cpu\_access()*  
  Through this interface, the exporter driver can ensure that the current buffer is only allowed to be accessed by the CPU. In this process, allocate or swap-in and pin (fixed) backend storage may be required. In addition, the exporter driver also needs to ensure that the direction of the CPU access is consistent with the direction it requests.
* *dma\_buf\_end\_cpu\_access()*  
  This interface is called when the importer completes the CPU Access. The exporter can implement the cache flush operation in this interface and unpin the memory resources pinned in *dma\_buf\_begin\_cpu\_access() .*

When the above interfaces are used, we can think that the DMA-BUF memory belongs to the DMA device by default, not the CPU. Therefore, it is necessary to complete the synchronization operation of the CPU Cache in these interfaces to ensure that the data obtained by the CPU is consistent with that in the DMA-BUF. At the same time, this method can also avoid the expensive cache synchronization operations caused by only passing, mapping and accessing DMA-BUFs between multiple devices.

However, this inconsistent calling rule with the DMA API may cause some confusion, and not all DMA-BUF exporter drivers use the same implementation strategy. Some exporter drivers intend to still follow the DMA API calling rules, flush and invalidate the CPU cache every time the map and unmap operations are performed; other exporter drivers may only be performed in their begin and end callback interfaces. Cache synchronization operation, and some exporter drivers may implement both solutions.

Although DMA BUF is designed to share memory between user space and multiple DMA devices, the exporter that first exports DMA-BUF is often a special driver, which is customized by the manufacturer and strongly related to the driver. The buffer allocation code. For example, a GPU driver that allocates a buffer, then performs rendering operations on it, and returns a handle to user space. The user-space application can then send this buffer, along with other buffers, back to the GPU to composite the web browser window with other windows on the desktop. DMA-BUF provides a more general handle type, so even if the buffer is not used for multi-device sharing, its handle can still be used.

However, it should be known that the cache synchronization operation needs to be considered only when the buffer is shared between the CPU and the DMA device, so the DMA-BUF exporter can do some cache optimization for the case where the buffer is only shared between multiple devices. For example, some DMA-BUF exporter drivers first save the scatter-gather table when performing the DMA mapping operation for the first time, and continue to use these tables as long as subsequent *dma\_buf\_map\_attachment()* calls are executed in the same DMA direction . In this way, we can avoid the expensive cache operation every time we call *dma\_buf\_map\_attachment()* and *dma\_buf\_unmap\_attachment() , and finally release the previous DMA mapping resources in dma\_buf\_detach()* . These optimizations work because the exporter is bound to the DMA device, so the buffer is not actually shared, or the DMA devices that share the buffer are all cache consistent, so there is no need to maintain the cache. operation.

Translator's Note: The above paragraph is really difficult to translate, especially the boldface part below, it is really difficult to understand, so I will post the original text, hoping that someone can correct the mistake, so as not to mislead the children.  
  
  
***But, knowing that the buffer was shared between just the CPU and the device, the DMA-BUF exporter could optimize some of the cache operations.*** For instance, some DMA-BUF exporters cache the scatter/gather table resulting from the first DMA mapping operation and, as long as the dma\_buf\_map\_attachment() calls are done in the same direction, reuse that table. In this way, they can avoid expensive cache operations on each dma\_buf\_map\_attachment() and dma\_buf\_unmap\_attachment() call, finally releasing the mapping in dma\_buf\_detach() . ***These optimizations work because the exporters are tied to the device, so the buffers aren't really being shared,***or the devices the buffers are shared with are cache coherent, so the cache maintenance is unnecessary.

1. The first sentence clearly states that the buffer is shared between the CPU and the device. In this case, the cache operation must be performed every time. How to optimize it?
2. Why does the second sentence say that the exporter is bound to the device? Why is the buffer not really shared? Sharing buffers between multiple DMA devices is not called sharing?

Although this method is effective, it results in that in the upstream version, more than a dozen DMA-BUF exporter drivers have their own different cache processing methods and calling rules. Therefore, when we started to study how to implement a general DMA-BUF exporter framework to support multi-device pipeline from a certain performance point of view, we could not find a clear implementation solution.

**Handling buffer ownership issues with multiple mappings**

While the DMA API provides good documentation on how to use the map and unmap calls (to specify buffer ownership), achieving good performance on mobile platforms often requires multiple DMA devices and CPUs to establish valid buffers at the same time. mapping, which makes the concept of buffer ownership more subtle. For example, in a graphics system, the GPU and Display are usually mapped to the same buffer at the same time. For this reason, the system must establish a framebuffer sharing mapping relationship between multiple devices before the frame is drawn. In this way, the GPU can directly write data to the buffer, and then send a signal to the display driver after the writing is completed, and then the display driver can display the buffer immediately.

For this specific application scenario, DMA-BUF adds [dma-fence based on the](https://www.kernel.org/doc/html/v5.6/driver-api/dma-buf.html" \l "dma-fences)[explicit fence](https://lwn.net/Articles/702339/) architecture , which provides a mechanism for the driver (or user space) to wait for buffer fence. Eventually, another driver will signal the fence, thereby starting the switch of buffer ownership. However, to support this parallel mapping relationship requires careful handling of cache synchronization issues, which are usually implemented by the driver calling the DMA API synchronization interface. When a developer develops with a vendor-specific kernel on an integrated device, he may know which driver a buffer is coming from and to whom, so he can add the most appropriate and correct cache processing code. But once it's beyond his control, things get pretty complicated.

So we see here that there are two different ways of handling *ownership* tracking. **Implicit** handling means that the ownership of the DMA-BUF is switched when dma map or unmap, and **explicit** handling means that the buffer has been mapped to two or more devices, it Ownership is effectively switched through DMA-BUF fence.

The DMA-BUF exporter usually handles cache related operations while passing buffer ownership. They can do this in the implicit context of calls to *dma\_buf\_map\_attachment()* and *dma\_buf\_unmap\_attachment() , or they can do so in calls to dma\_buf\_begin\_cpu\_access()* and *dma\_buf\_end\_cpu\_access()* . However, in the case of explicit handling, the DMA-BUF exporter does not have a callback interface for DMA-BUF fence signals, so the exporter cannot perform any cache management operations for ownership switching, which creates a dilemma. In this case, the responsibility of buffer cache management is allocated to the DMA-BUF exporter and the driver using the buffer. To do this correctly, each driver must understand its position in the buffer pipeline and thus the cache coherency of its downstream devices.

Even more troublesome, even if the DMA-BUF exporter does have a callback interface for the dma-fence signal, it has no way of knowing which ownership tracking method is currently in use. Assuming the explicit processing mode defaults to CPU ownership, do we perform cache operations in the map and unmap functions? Or the implicit handling mode defaults to device ownership and we do cache operations in *dma\_buf\_begin\_cpu\_access()* and *dma\_buf\_end\_cpu\_access() ?*Or do we avoid the extra cache overhead when the driver switches ownership by executing an explicit fence signal? These choices may leave us with an implementation that is either too slow to use, or may be incompatible with some drivers, which completely defeats the original purpose of DMA BUF as a general purpose swap mechanism.

So to a developer trying to write a DMA-BUF exporter driver, this all starts to feel like a [level](https://ozlabs.org/~rusty/ols-2003-keynote/img56.html) 10 ("read the documentation, you'll get it wrong") or level 11 ("follow the routine, you'll get it wrong"), especially if you care about performance. This presents a huge obstacle to the goal of sharing a common DMA-BUF Heap among vendors.

**possible solution**

I think we can improve this situation, and I have some ideas that I can share with you. Since the DMA-BUF interface has deviated from the DMA API, I think we should establish some clear specifications for the use of DMA-BUF, and form a good development document, so that DMA-BUF exporter authors and DMA-BUF users can understand The model has a unified understanding. We should focus on the following directions:

* Creates a formal ownership of the DMA-BUF object outside the implicit map/unmap functions of the DMA API.
* Provide a set of calling mechanisms to track ownership, these interfaces can be added to the *dma buf\_ops* structure, so that the exporter driver can know the state changes of these ownerships.
* Implicit handling mode is deprecated and drivers should be asked to use the new mechanism above to mark ownership switches in explicit handling mode.
* Add some state tracking interfaces to DMA-BUFs, so that we can know their cache state, and only perform corresponding cache operations when ownership switches, so those state tracking interfaces become particularly important.

Most of the above can be achieved by documenting and enhancing the current DMA-BUF exporter invocation mechanism. *The dma\_buf\_begin\_cpu\_access()* and *dma\_buf\_end\_cpu\_access()* calls are sufficient to handle device-to-CPU and CPU-to-device transitions. But we need to clearly define the correct usage specification of these functions, and should always be implemented by the DMA-BUF exporter driver, thus normalizing the notion that buffers are device-owned by default. This way you can safely implement pre-flushed buffers and skip unnecessary cache operations.

However, this method has a disadvantage. For the case that the CPU needs to access the buffer multiple times (the device is not involved in the middle), each call will have an unnecessary cache flush operation. In addition, there is a problem that for a hybrid system with both CPU-coherent and non-coherent devices participating, we may need to do CPU-cache synchronization when switching ownership between these devices. In both cases, it might be helpful to use the device-usage function call and state tracking interface, so that you can decide whether to switch ownership (rather than just use it).

This concept of *ownership* also needs to take into account future partial cache flush operations to allow the CPU and DMA devices to access the same buffer at the same time. In this way, buffer ownership (and related cache operations) will be managed at the granularity of individual cache lines, rather than at the level of the entire buffer, which looks more like advisory range locks on file operations .

It is undeniable that DMA-BUF Heap (and ION before it) in some cases, the user space will know more about the purpose of the buffer than the kernel space. Therefore, it is most appropriate to let user space choose the buffer allocation type for a pipeline. The DMA-BUF design philosophy provides us with very practical flexibility, which allows to leave buffer rules and policies to the exporter driver, so I don't want to eliminate that flexibility. But I do think that as vendor vendors start their ION migration work, it is more important to have a clear and established specification, so that everyone will not fall into the pit, so as to avoid a batch of unnecessary , incompatible heaps and consumers. I hope this article can be a source of inspiration and arouse further discussion.

**thanks**

Many thanks to Rob Clark, Robert Foss, Sumit Semwal, Azam Sadiq Pasha kapatral Syed, Daniel Vetter and Linus Walleij for their early reviews and feedback on these two articles!

Previous: ["LWN translation: DMA-BUF cache handling: Off the DMA API map (part 1)"](https://blog.csdn.net/hexiaolong2009/article/details/106745686)

**DMA-BUF Article Summary:**["My DMA-BUF Column"](https://blog.csdn.net/hexiaolong2009/category_10838100.html)

# LWN translation: DMA-BUF cache handling: Off the DMA API map (part 2)

Disclaimer: This article is not original, just a translation!  
Original text: <https://lwn.net/Articles/822521/>  
Author: John Stultz (Linaro member, kernel timekeeping maintainer)  
Note: This article requires background knowledge of [DMA](https://so.csdn.net/so/search?q=DMA&spm=1001.2101.3001.7020) -BUF. If you don’t know DMA-BUF yet, it is recommended to read it first The translator's own ["dma-buf from shallow to deep"](https://blog.csdn.net/hexiaolong2009/category_10838100.html)[series](https://blog.csdn.net/hexiaolong2009/article/details/102596772) Chapter 3 "map attachment" [and](https://blog.csdn.net/hexiaolong2009/article/details/102596825) Chapter 6 "begin / end cpu\_access" .

In the [previous](https://blog.csdn.net/hexiaolong2009/article/details/106745686) article, I introduced some background knowledge about ION, DMA-BUF Heap, DMA [API](https://so.csdn.net/so/search?q=API&spm=1001.2101.3001.7020) , and the basic concept of CPU Cache "ownership", and finally described how DMA-BUF is from the perspective of traditional DMA API Deals with cache synchronization issues. The article concludes by discussing why traditional DMA APIs perform so poorly on modern mobile platforms. This article will discuss with you how to make the DMA-BUF exporter avoid unnecessary cache operations, and give some general suggestions on how to improve these methods.

From the point of view of the DMA API: by calling *dma\_buf\_map\_attachment()* , the ownership of the DMA-BUF is transferred to the DMA device, and by calling *dma\_buf\_unmap\_attachment()* , the ownership is returned to the CPU, and each time these two functions are called, the cache correlation is executed operation. Although such a sequential operation can ensure the correctness of CPU Cache processing, for buffer pipeline operations involving multiple DMA devices, the CPU does not actually participate in accessing these buffers at all, and each cache map and unmap operations increase It can cause significant performance issues.

**Who owns the buffer?**

To avoid these redundant cache operations, the DMA-BUF interface allows some rules of the DMA API to be reversed. It should be noted that the DMA API assumes that the CPU is the natural owner of all memory, and this reverse rule only needs to be considered during a DMA transfer (the ownership of the buffer has been explicitly transferred to the DMA device). [*The DMA-BUF interface requires the CPU to call dma\_buf\_begin\_cpu\_access()*](https://elixir.bootlin.com/linux/v5.7.1/source/drivers/dma-buf/dma-buf.c#L1064) before accessing the DMA-BUF, and call *[dma\_buf\_end\_cpu\_access](https://elixir.bootlin.com/linux/v5.7.1/source/drivers/dma-buf/dma-buf.c" \l "L1100)* () after the access ends . If the CPU wants to access the buffer from user space, it can use the DMA\_BUF\_IOCTL\_SYNCioctl() command to initiate a call to begin/end cpu\_access.

Special interface:

* *dma\_buf\_begin\_cpu\_access()*  
  Through this interface, the exporter driver can ensure that the current buffer is only allowed to be accessed by the CPU. In this process, allocate or swap-in and pin (fixed) backend storage may be required. In addition, the exporter driver also needs to ensure that the direction of the CPU access is consistent with the direction it requests.
* *dma\_buf\_end\_cpu\_access()*  
  This interface is called when the importer completes the CPU Access. The exporter can implement the cache flush operation in this interface and unpin the memory resources pinned in *dma\_buf\_begin\_cpu\_access() .*

When the above interfaces are used, we can think that the DMA-BUF memory belongs to the DMA device by default, not the CPU. Therefore, it is necessary to complete the synchronization operation of the CPU Cache in these interfaces to ensure that the data obtained by the CPU is consistent with that in the DMA-BUF. At the same time, this method can also avoid the expensive cache synchronization operations caused by only passing, mapping and accessing DMA-BUFs between multiple devices.

However, this inconsistent calling rule with the DMA API may cause some confusion, and not all DMA-BUF exporter drivers use the same implementation strategy. Some exporter drivers intend to still follow the DMA API calling rules, flush and invalidate the CPU cache every time the map and unmap operations are performed; other exporter drivers may only be performed in their begin and end callback interfaces. Cache synchronization operation, and some exporter drivers may implement both solutions.

Although DMA BUF is designed to share memory between user space and multiple DMA devices, the exporter that first exports DMA-BUF is often a special driver, which is customized by the manufacturer and strongly related to the driver. The buffer allocation code. For example, a GPU driver that allocates a buffer, then performs rendering operations on it, and returns a handle to user space. The user-space application can then send this buffer, along with other buffers, back to the GPU to composite the web browser window with other windows on the desktop. DMA-BUF provides a more general handle type, so even if the buffer is not used for multi-device sharing, its handle can still be used.

However, it should be known that the cache synchronization operation needs to be considered only when the buffer is shared between the CPU and the DMA device, so the DMA-BUF exporter can do some cache optimization for the case where the buffer is only shared between multiple devices. For example, some DMA-BUF exporter drivers first save the scatter-gather table when performing the DMA mapping operation for the first time, and continue to use these tables as long as subsequent *dma\_buf\_map\_attachment()* calls are executed in the same DMA direction . In this way, we can avoid the expensive cache operation every time we call *dma\_buf\_map\_attachment()* and *dma\_buf\_unmap\_attachment() , and finally release the previous DMA mapping resources in dma\_buf\_detach()* . These optimizations work because the exporter is bound to the DMA device, so the buffer is not actually shared, or the DMA devices that share the buffer are all cache consistent, so there is no need to maintain the cache. operation.

Translator's Note: The above paragraph is really difficult to translate, especially the boldface part below, it is really difficult to understand, so I will post the original text, hoping that someone can correct the mistake, so as not to mislead the children.  
  
  
***But, knowing that the buffer was shared between just the CPU and the device, the DMA-BUF exporter could optimize some of the cache operations.*** For instance, some DMA-BUF exporters cache the scatter/gather table resulting from the first DMA mapping operation and, as long as the dma\_buf\_map\_attachment() calls are done in the same direction, reuse that table. In this way, they can avoid expensive cache operations on each dma\_buf\_map\_attachment() and dma\_buf\_unmap\_attachment() call, finally releasing the mapping in dma\_buf\_detach() . ***These optimizations work because the exporters are tied to the device, so the buffers aren't really being shared,***or the devices the buffers are shared with are cache coherent, so the cache maintenance is unnecessary.

1. The first sentence clearly states that the buffer is shared between the CPU and the device. In this case, the cache operation must be performed every time. How to optimize it?
2. Why does the second sentence say that the exporter is bound to the device? Why is the buffer not really shared? Sharing buffers between multiple DMA devices is not called sharing?

Although this method is effective, it results in that in the upstream version, more than a dozen DMA-BUF exporter drivers have their own different cache processing methods and calling rules. Therefore, when we started to study how to implement a general DMA-BUF exporter framework to support multi-device pipeline from a certain performance point of view, we could not find a clear implementation solution.

**Handling buffer ownership issues with multiple mappings**

While the DMA API provides good documentation on how to use the map and unmap calls (to specify buffer ownership), achieving good performance on mobile platforms often requires multiple DMA devices and CPUs to establish valid buffers at the same time. mapping, which makes the concept of buffer ownership more subtle. For example, in a graphics system, the GPU and Display are usually mapped to the same buffer at the same time. For this reason, the system must establish a framebuffer sharing mapping relationship between multiple devices before the frame is drawn. In this way, the GPU can directly write data to the buffer, and then send a signal to the display driver after the writing is completed, and then the display driver can display the buffer immediately.

For this specific application scenario, DMA-BUF adds [dma-fence based on the](https://www.kernel.org/doc/html/v5.6/driver-api/dma-buf.html" \l "dma-fences)[explicit fence](https://lwn.net/Articles/702339/) architecture , which provides a mechanism for the driver (or user space) to wait for buffer fence. Eventually, another driver will signal the fence, thereby starting the switch of buffer ownership. However, to support this parallel mapping relationship requires careful handling of cache synchronization issues, which are usually implemented by the driver calling the DMA API synchronization interface. When a developer develops with a vendor-specific kernel on an integrated device, he may know which driver a buffer is coming from and to whom, so he can add the most appropriate and correct cache processing code. But once it's beyond his control, things get pretty complicated.

So we see here that there are two different ways of handling *ownership* tracking. **Implicit** handling means that the ownership of the DMA-BUF is switched when dma map or unmap, and **explicit** handling means that the buffer has been mapped to two or more devices, it Ownership is effectively switched through DMA-BUF fence.

The DMA-BUF exporter usually handles cache related operations while passing buffer ownership. They can do this in the implicit context of calls to *dma\_buf\_map\_attachment()* and *dma\_buf\_unmap\_attachment() , or they can do so in calls to dma\_buf\_begin\_cpu\_access()* and *dma\_buf\_end\_cpu\_access()* . However, in the case of explicit handling, the DMA-BUF exporter does not have a callback interface for DMA-BUF fence signals, so the exporter cannot perform any cache management operations for ownership switching, which creates a dilemma. In this case, the responsibility of buffer cache management is allocated to the DMA-BUF exporter and the driver using the buffer. To do this correctly, each driver must understand its position in the buffer pipeline and thus the cache coherency of its downstream devices.

Even more troublesome, even if the DMA-BUF exporter does have a callback interface for the dma-fence signal, it has no way of knowing which ownership tracking method is currently in use. Assuming the explicit processing mode defaults to CPU ownership, do we perform cache operations in the map and unmap functions? Or the implicit handling mode defaults to device ownership and we do cache operations in *dma\_buf\_begin\_cpu\_access()* and *dma\_buf\_end\_cpu\_access() ?*Or do we avoid the extra cache overhead when the driver switches ownership by executing an explicit fence signal? These choices may leave us with an implementation that is either too slow to use, or may be incompatible with some drivers, which completely defeats the original purpose of DMA BUF as a general purpose swap mechanism.

So to a developer trying to write a DMA-BUF exporter driver, this all starts to feel like a [level](https://ozlabs.org/~rusty/ols-2003-keynote/img56.html) 10 ("read the documentation, you'll get it wrong") or level 11 ("follow the routine, you'll get it wrong"), especially if you care about performance. This presents a huge obstacle to the goal of sharing a common DMA-BUF Heap among vendors.

**possible solution**

I think we can improve this situation, and I have some ideas that I can share with you. Since the DMA-BUF interface has deviated from the DMA API, I think we should establish some clear specifications for the use of DMA-BUF, and form a good development document, so that DMA-BUF exporter authors and DMA-BUF users can understand The model has a unified understanding. We should focus on the following directions:

* Creates a formal ownership of the DMA-BUF object outside the implicit map/unmap functions of the DMA API.
* Provide a set of calling mechanisms to track ownership, these interfaces can be added to the *dma buf\_ops* structure, so that the exporter driver can know the state changes of these ownerships.
* Implicit handling mode is deprecated and drivers should be asked to use the new mechanism above to mark ownership switches in explicit handling mode.
* Add some state tracking interfaces to DMA-BUFs, so that we can know their cache state, and only perform corresponding cache operations when ownership switches, so those state tracking interfaces become particularly important.

Most of the above can be achieved by documenting and enhancing the current DMA-BUF exporter invocation mechanism. *The dma\_buf\_begin\_cpu\_access()* and *dma\_buf\_end\_cpu\_access()* calls are sufficient to handle device-to-CPU and CPU-to-device transitions. But we need to clearly define the correct usage specification of these functions, and should always be implemented by the DMA-BUF exporter driver, thus normalizing the notion that buffers are device-owned by default. This way you can safely implement pre-flushed buffers and skip unnecessary cache operations.

However, this method has a disadvantage. For the case that the CPU needs to access the buffer multiple times (the device is not involved in the middle), each call will have an unnecessary cache flush operation. In addition, there is a problem that for a hybrid system with both CPU-coherent and non-coherent devices participating, we may need to do CPU-cache synchronization when switching ownership between these devices. In both cases, it might be helpful to use the device-usage function call and state tracking interface, so that you can decide whether to switch ownership (rather than just use it).

This concept of *ownership* also needs to take into account future partial cache flush operations to allow the CPU and DMA devices to access the same buffer at the same time. In this way, buffer ownership (and related cache operations) will be managed at the granularity of individual cache lines, rather than at the level of the entire buffer, which looks more like advisory range locks on file operations .

It is undeniable that DMA-BUF Heap (and ION before it) in some cases, the user space will know more about the purpose of the buffer than the kernel space. Therefore, it is most appropriate to let user space choose the buffer allocation type for a pipeline. The DMA-BUF design philosophy provides us with very practical flexibility, which allows to leave buffer rules and policies to the exporter driver, so I don't want to eliminate that flexibility. But I do think that as vendor vendors start their ION migration work, it is more important to have a clear and established specification, so that everyone will not fall into the pit, so as to avoid a batch of unnecessary , incompatible heaps and consumers. I hope this article can be a source of inspiration and arouse further discussion.

**thanks**

Many thanks to Rob Clark, Robert Foss, Sumit Semwal, Azam Sadiq Pasha kapatral Syed, Daniel Vetter and Linus Walleij for their early reviews and feedback on these two articles!

Previous: ["LWN translation: DMA-BUF cache handling: Off the DMA API map (part 1)"](https://blog.csdn.net/hexiaolong2009/article/details/106745686)

**DMA-BUF Article Summary:**["My DMA-BUF Column"](https://blog.csdn.net/hexiaolong2009/category_10838100.html)