# Linux Drm显示驱动

1. **DRM介绍**

DRM（Direct Rendering Manager）是linux的一个子系统，为操作显卡提供一系列的接口函数，通过这些接口，用户空间可以发命令和数据到GPU，例如，设置显示的显示模式等操作。下面是带有DRM和没有DRM的两张对比图，如下：

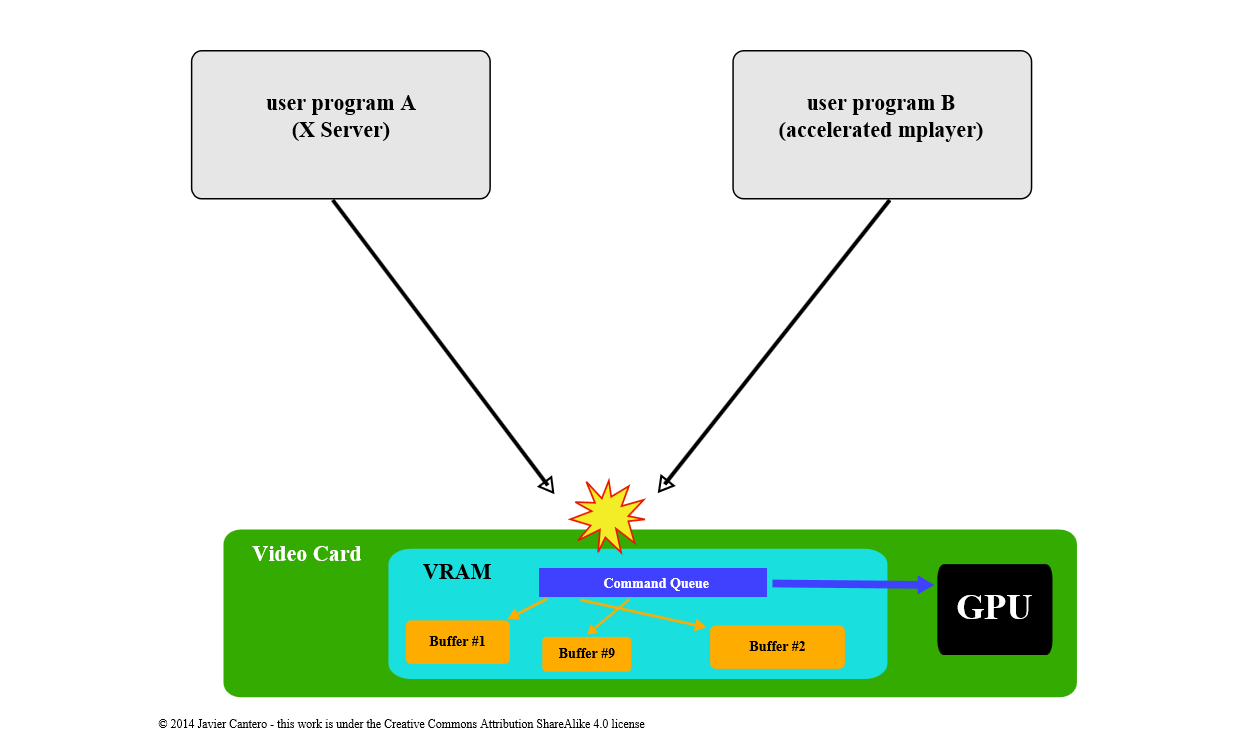


图1 没有DRM

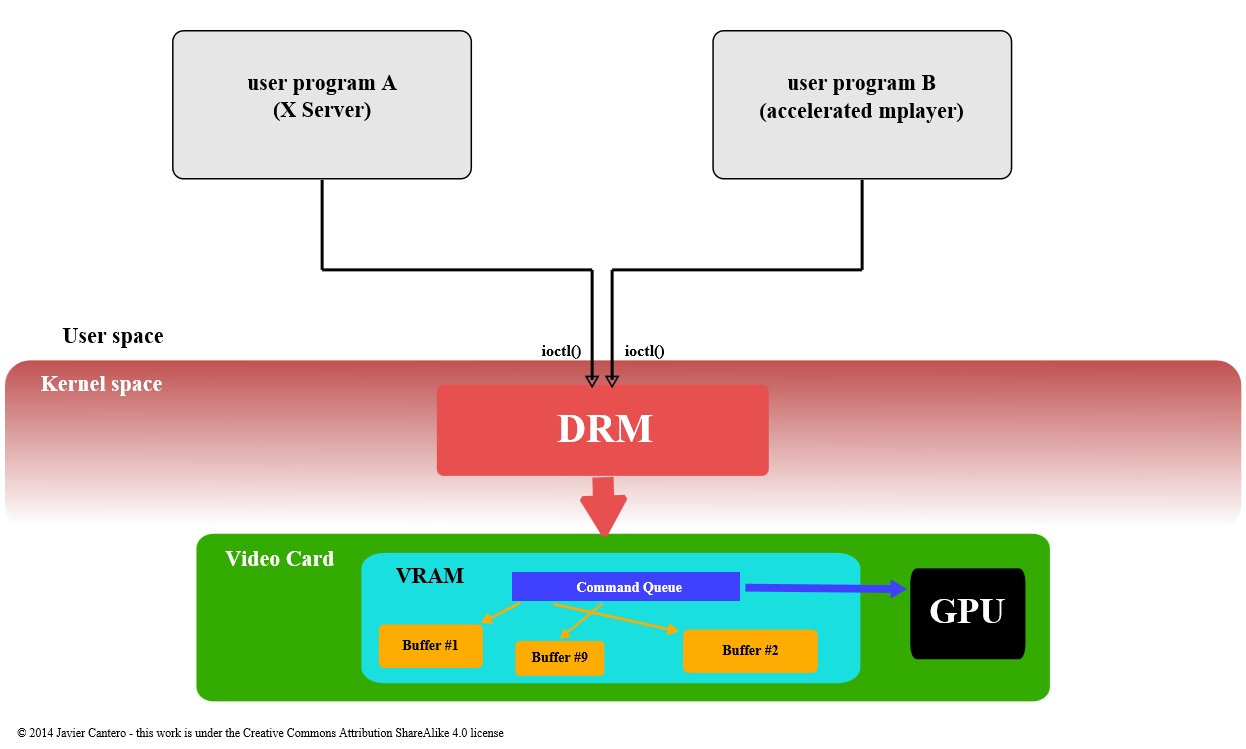


图2 有DRM

由上面两张图可以看到，图一，没有drm的，当多个用户应用操作显卡的时候，就会出现冲突；图2，带有drm，可以管理用户程序的访问，避免同时操作显卡等。Drm提供了专门访问显卡的路径，避免访问冲突，负责初始化和维护命令队列、显存和其他的显卡硬件资源等。

前面多次提及到显卡，但我们对显卡并没有什么概念，这里有必要了解下显卡的概念，参考wiki对显卡的介绍，我们平常使用的桌面电脑一般都集成了一个显卡，这些显卡集成了处理模块（Processing logic）、显示控制器（display contrller）和硬件加速（hardware video acceleration）。其中，processing logic指的是GPU模块，display controller指的是LCDC控制器，hardware video acceleration指的是显示接口HDMI/edp等。

了解完显卡后，我们继续了解drm。Drm框架向用户空间提供了很多的功能，比如framebuffer的管理、模式设置、共享内存object和内存同步等。宏观上看，drm主要分为两个功能部分——GEM和KMS。GEM（Graphics Execution Manager）主要功能是管理Framebuffer，如显存的申请释放（Framebuffer managing）、显存共享机制（Memory sharing objects）和显存同步机制（Memory synchronization）等；而KMS主要完成显卡的配置（Display mode setting）。

本文主要介绍的是drm kms模块。Kms对显示驱动抽象出了三个概念，分别为CRTC、Encoder和Connector。这三个概念非常重要，它们贯彻整个显示驱动，但理解它们也非常简单。CRTC指的是Display Controller，Encoder指的是显示接口HDMI/eDP等，connector指的是连接的monitor/panel等。各个部分的作用如下：

CRTC：

1. 显示电源状态管理DPMS（Display Power Manage System）——crtc\_funcs->dpms；
2. 将framebuffer转换成标准的lcdc timing，即通过crtc\_funcs->mode\_set；

Encoder：

1. 将 display controller 输出的 lcdc Timing 打包转化为对应接口时序 HDMI TMDS / … (encoder\_funcs->mode\_set)；
2. DPMS (Display Power Manage System) 电源状态管理 (encoder\_funcs->dpms)。

Connector：

1. 获取hotplug的状态；
2. 上报获取到的EDID信息。

下面以HDMI monitor显示过程，了解上面各部分的作用。

1. HDMI驱动检测到hdmi monitor的插入信息后，获取monitor的edid信息，如monitor的分辨率信息等——（Connector）；
2. 用户空间根据获取到的monitor信息，设置framebuffer的属性，然后，通知显示驱动可以显示了；
3. CRTC display controller根据framebuffer的属性，转换成lcdc timing——（CRTC）；
4. Encoder把来自display controller的lcdc timing转换成encoder接口的timing——（Encoder）；

前面已经了解了drm架构开发的目的，但本文的目的不是探讨GPU，主要目的是探讨利用drm架构进行显示接口的开发，实际上并没有涉及到GPU。

1. **源码目录**

drm显示驱动源码分布在比较多的目录，下面对目录做一些简单的说明。

gpu目录下的是drm框架核心部分，主要是框架的逻辑部分。

drivers/gpu/drm/ rockchip/

\_\_ rockchip\_drm\_drv.c /\* drm\_driver的注册 \*/

\_\_ rockchip\_drm\_primary.c /\* 主显示屏幕的rochchip\_drm\_subdrv注册 \*/

\_\_ rockchip\_drm\_core.c /\* drm\_device注册接口，encoder和connector的创建 \*/

\_\_ rockchip\_drm\_fbdev.c /\* fbdev的初始化、创建等 \*/

\_\_ rockchip\_drm\_connector.c /\* connector注册、初始化和管理接口 \*/

\_\_ rockchip\_drm\_encoder.c /\* encoder的注册、初始化和管理接口 \*/

video目录为drm框架的底层部分，该部分直接与硬件交流。

drivers/video/rockchip/

\_\_ rk\_drm\_fb.c /\* 底层部分的核心，drm框架使用这里提供的接口操作硬件，提供接口给底层的各部分注册rk\_lcdc\_driver等 \*/

\_\_ rkfb\_sysfs.c display-sys.c /\* 提供sys文件系统接口给用户空间 \*/

\_\_ lcdc/

\_\_ rk312x\_lcdc.c /\* 对应与drm的crtc，crtc硬件的实现 \*/

\_\_ screen/

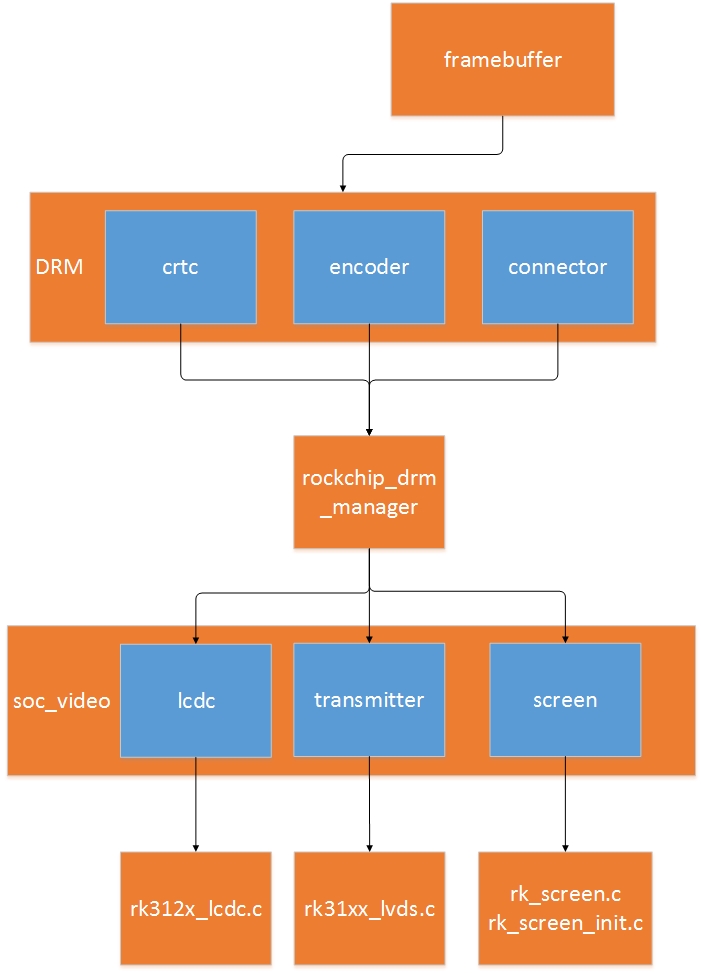
\_\_ rk\_screen.c /\* 对应一个connector，注册一个rk\_screen \*/

\_\_ rk\_screen\_init.c /\* 提供接口初始化rk\_screen，被rk\_screen.c调用 \*/

\_\_ transmitter/

\_\_ rk31xx\_lvds.c /\* 对应于drm的encoder，encoder硬件的实现 \*/

1. **drm驱动框架框图**



如上图所示，通过rockchip提供的rockchip\_drm\_manager，drm框架层操作soc\_video的硬件。

1. **驱动代码分析**

本文将以rockchip px3se lvds显示接口为例分析drm显示驱动，其中，把drm驱动分为两大块分析，第一部分为drm框架部分，第二部分为底层实现部分。

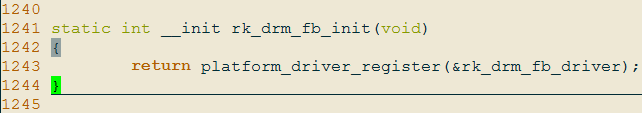
* 1. 底层实现

源码目录一节概括了各个文件的主要作用，把底层分为了底层管理核心、screen、transmitter和lcdc四部分，下面将以这几部分为模块分析底层的实现。

* + 1. 底层管理核心

rk\_drm\_fb.c文件是底层管理核心的实现。与其他驱动一样，rk\_drm\_fb.c通过平台设备驱动的方式，在平台设备驱动的probe函数完成工作，下面我们将从代码的角度来分析。

rk\_drm\_fb\_init函数如下：



rk\_drm\_fb\_init做的工作很简单，只是注册了一个平台设备驱动rk\_drm\_fb\_driver到设备驱动框架中，当platform\_device与rk\_drm\_fb\_driver匹配后，运行rk\_drm\_fb\_drvier的probe函数，该函数如下：

static int rk\_drm\_fb\_probe(struct platform\_device \*pdev)

{

struct rk\_drm\_private \*rk\_drm\_priv = NULL;

--cut --

rk\_drm\_priv = devm\_kzalloc(&pdev->dev,

sizeof(struct rk\_drm\_private), GFP\_KERNEL);

if (!rk\_drm\_priv) {

dev\_err(&pdev->dev, "kmalloc for rk\_drm\_priv fail!");

return -ENOMEM;

}

platform\_set\_drvdata(pdev, rk\_drm\_priv);

if (!of\_property\_read\_u32(np, "rockchip,disp-mode", &mode)) {

rk\_drm\_priv->disp\_mode = mode;

} else {

dev\_err(&pdev->dev, "no disp-mode node found!");

return -ENODEV;

}

dev\_set\_name(&pdev->dev, "rockchip-drmfb");

drm\_fb\_pdev = pdev;

--cut--

}

函数非常简单，第一，为rk\_drm\_priv申请内存空间；第二，设置rk\_drm\_priv为平台设备pdev的driver data，获取设备树disp-mode信息，初始化rk\_drm\_priv的dis\_mode字段；最后，全局变量drm\_fb\_pdev指向平台设备pdev。

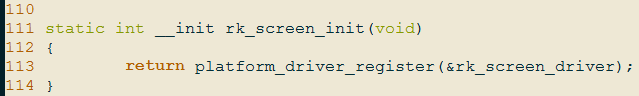
在这里使用了drm\_fb\_pdev全局变量，接下来，我们也会经常看到全局变量的使用，个人观点，应该减少全局变量的使用，因为我看rockchip drm显示驱动的时候，因为全局变量的使用，很难把各个模块联系在一起，造成理解上的困难。

言归正传，虽然probe函数所做的工作非常少，但并不意味着底层核心部分就那么简单，不是的，probe函数主要的作用是为rk\_drm\_priv申请了空间，底层的其他模块调用管理核心提供的很多接口函数会使用到rk\_drm\_priv。结构体rk\_drm\_private包含了很多信息，主要是包含了糅合底层各个模块的信息及接口等。下面分析底层其他模块调用管理核心的时候，再分析rk\_drm\_priv。

* + 1. screen

这一节将分析底层的screen，前面的章节已经讲到，底层的screen对应drm框架中的connector，通俗的来说，screen代表的是我们的显示屏。由前面的章节，我们已经知道，screen是由screen目录下的rk\_screen.c和rk\_screen\_init.c两文件实现，其中，rk\_screen.c是主要部分，rk\_screen\_init.c只是提供了一些接口初始化rk\_screen而已，下面我们进入具体的代码。

rk\_screen\_init函数如下：



与上一节一样，rk\_screen\_init只是注册了一个平台设备驱动，当与设备匹配的时候，将执行rk\_screen\_driver的probe回调函数，probe函数如下所示：

static int rk\_screen\_probe(struct platform\_device \*pdev)

{

--cut--

rk\_screen = devm\_kzalloc(&pdev->dev,

sizeof(struct rk\_screen), GFP\_KERNEL);

if (!rk\_screen) {

dev\_err(&pdev->dev, "kmalloc for rk screen fail!");

return -ENOMEM;

}

ret = rk\_fb\_prase\_timing\_dt(np, rk\_screen);

rk\_fb\_get\_screen\_init(np, rk\_screen);

--cut--

}

在这里，rk\_screen是一个全局变量。首先，probe函数为rk\_screen申请内存空间，然后调用rk\_fb\_prase\_timing\_dt函数，获取设备树提供的pixcel clock、front porch、back proch等信息，利用这些信息，初始化rk\_screen。Lcdc模块注册的时候，会调用screen模块的接口，进而会使用到全局变量rk\_screen。

* + 1. transmitter

接下来，我们将分析transmitter模块，我们已经知道，transmitter模块对应的是drm框架的encoder，负责把Lcdc timing转换为对应接口的timing。与前面一样，transmitter也是通过平台设备驱动，接下来，我们直接进入probe函数，该函数如下所示：

文件位置：drivers/video/rockchip/transmitter/ rk31xx\_lvds.c。

static int rk31xx\_lvds\_probe(struct platform\_device \*pdev)

{

--cut--

lvds = devm\_kzalloc(&pdev->dev, sizeof(struct rk\_lvds\_device), GFP\_KERNEL);

if (!lvds) {

dev\_err(&pdev->dev, "kzalloc rk31xx lvds failed\n");

return -ENOMEM;

}

--cut--

rk\_fb\_get\_prmry\_screen(&lvds->screen);

--cut--

platform\_set\_drvdata(pdev, lvds);

dev\_set\_name(lvds->dev, "rk31xx-lvds");

--cut--

rk31xx\_lvds = lvds;

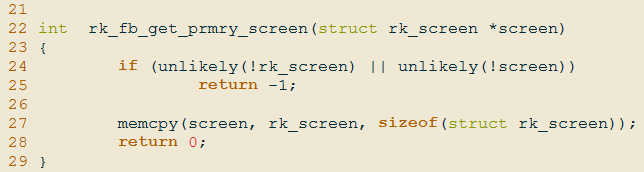
rk\_fb\_trsm\_ops\_register(&trsm\_lvds\_ops, SCREEN\_LVDS);

dev\_info(&pdev->dev, "rk31xx lvds driver probe success\n");

--cut--

}

1. 上面的probe函数代码省略掉了很多我们这里不关心的代码，首先，probe函数为lvds申请内存空间（lvds为rk\_lvds\_device结构体），抽象了一个lvds显示接口；



1. rk\_fb\_get\_prmry\_screen函数如上图所示，由前面screen一节，我们知道，rk\_screen为全局变量，screen模块初始化的时候，已经初始化rk\_screen，在这里，建立transmitter与screen之间的联系，这样，transmitter可以获取到screen的信息，并根据这些信息做不同的处理；
2. rk\_fb\_trsm\_ops\_register函数根据第二个参数把transmitter的回调函数赋值给trsm\_lvds\_ops、trsm\_edp\_ops和trsm\_mipi\_ops中的一个，把操作transmitter的回调函数注册到核心管理层，这样，核心管理层通过这些接口操作transmitter。
   * 1. lcdc

lcdc是显示控制器，对应drm显示驱动框架中的crtc，与前面的screen和transmitter模块一样，lcdc也是通过平台设备驱动注册，probe函数为rk312x\_lcdc\_probe，该函数如下所示：

static int rk312x\_lcdc\_probe(struct platform\_device \*pdev)

{

struct lcdc\_device \*lcdc\_dev = NULL;

struct rk\_lcdc\_driver \*dev\_drv;

struct device \*dev = &pdev->dev;

　　　　--cut--

lcdc\_dev->hwc\_lut\_addr\_base = (lcdc\_dev->regs + HWC\_LUT\_ADDR);

lcdc\_dev->dsp\_lut\_addr\_base = (lcdc\_dev->regs + DSP\_LUT\_ADDR);

lcdc\_dev->prop = PRMRY;

dev\_set\_name(lcdc\_dev->dev, "lcdc%d", lcdc\_dev->id);

dev\_drv = &lcdc\_dev->driver;

dev\_drv->dev = dev;

dev\_drv->prop = lcdc\_dev->prop;

dev\_drv->id = lcdc\_dev->id;

dev\_drv->ops = &lcdc\_drv\_ops;

dev\_drv->lcdc\_win\_num = ARRAY\_SIZE(lcdc\_win);

spin\_lock\_init(&lcdc\_dev->reg\_lock);

--cut--

ret = rk\_fb\_register(dev\_drv, lcdc\_win, lcdc\_dev->id);

if (ret < 0) {

dev\_err(dev, "register fb for lcdc%d failed!\n", lcdc\_dev->id);

goto err\_register\_fb;

}

lcdc\_dev->screen = dev\_drv->screen0;

--cut—

}

1. 函数中的lcdc\_device结构体为lcdc的抽象数据结构，lcdc的信息包含在lcdc\_device结构体里面；函数刚开始初始化lcdc\_dev的一些成员，接着初始化dev\_drv（lcdc\_dev对应的driver），其中，dev\_drv的ops为操作lcdc的回调函数，管理核心部分通过该回调函数操作lcdc；
2. 上一步初始化完dev\_drv后，调用rk\_fb\_register注册dev\_drv到管理核心层，下面我们进入rk\_fb\_register函数，看该函数做了些什么。

rk\_fb\_register函数在rk\_drm\_fb.c文件（底层管理核心），函数如下所示：

int rk\_fb\_register(struct rk\_lcdc\_driver \*dev\_drv,

struct rk\_lcdc\_win \*win, int id)

{

struct rk\_drm\_private \*rk\_drm\_priv = platform\_get\_drvdata(drm\_fb\_pdev);

struct rk\_drm\_display \*drm\_display = NULL;

struct rk\_drm\_screen\_private \*drm\_screen\_priv = NULL;

int i = 0;

--cut--

rk\_drm\_priv->num\_screen++;

drm\_screen\_priv = &rk\_drm\_priv->screen\_priv[i];

drm\_screen\_priv->lcdc\_dev\_drv = dev\_drv;

drm\_screen\_priv->lcdc\_dev\_drv->id = id;

init\_lcdc\_device\_driver(drm\_screen\_priv, win, i);

dev\_drv->irq\_call\_back = rk\_drm\_irq\_handle;

drm\_display = &drm\_screen\_priv->drm\_disp;

drm\_display->num\_win = dev\_drv->lcdc\_win\_num;

atomic\_set(&drm\_screen\_priv->wait\_vsync\_done, 1);

init\_waitqueue\_head(&drm\_screen\_priv->wait\_vsync\_queue);

drm\_display->vop\_dev = dev\_drv->dev;

drm\_display->iommu\_en = dev\_drv->iommu\_enabled;

if (dev\_drv->prop == PRMRY) {

struct fb\_modelist \*modelist\_new;

struct fb\_modelist \*modelist;

struct fb\_videomode \*mode;

prm\_dev\_drv = dev\_drv;

drm\_display->modelist = kmalloc(sizeof(struct list\_head),

GFP\_KERNEL);

INIT\_LIST\_HEAD(drm\_display->modelist);

modelist\_new = kmalloc(sizeof(struct fb\_modelist),

GFP\_KERNEL);

drm\_display->screen\_type = RK\_DRM\_PRIMARY\_SCREEN;

drm\_display->num\_videomode = 1;

drm\_display->best\_mode = 0;

drm\_display->is\_connected = 1;

memcpy(&modelist\_new->mode, &dev\_drv->cur\_screen->mode,

sizeof(struct fb\_videomode));

list\_add\_tail(&modelist\_new->list, drm\_display->modelist);

modelist = list\_first\_entry(drm\_display->modelist,

struct fb\_modelist, list);

mode = &modelist->mode;

if (dev\_drv->ops->open)

dev\_drv->ops->open(dev\_drv, 0, 1);

if (dev\_drv->ops->mmu\_en)

dev\_drv->ops->mmu\_en(dev\_drv);

} else if (dev\_drv->prop == EXTEND) {

--cut—

}

--cut--

}

1. drm\_fb\_pdev为全局变量，通过该全局变量获取到pdev的driver data rk\_drm\_priv；
2. rk\_drm\_screen\_private结构体包含了rk\_screen（代表screen）、rk\_fb\_trsm\_ops（操作transmitter的回调函数）和rk\_lcdc\_driver（代表lcdc driver）等，起到了把screen、transmitter和lcdc联系在一块的作用。在这里，貌似rk\_drm\_screen\_private并没有使用rk\_screen和rk\_fb\_trsm\_ops两个成员，然而，rk\_lcdc\_driver也包含了这两个成员，通过rk\_lcdc\_driver把screen、lcdc和transmitter联系在一起，rk\_drm\_screen\_private只负责找到rk\_lcdc\_driver即可。rk\_drm\_priv是rk\_drm\_screen\_private结构体指针，首先获取rk\_drm\_priv第一个为空的screen\_priv，然后rk\_drm\_priv的lcdc\_dev\_drv指向dev\_drv；
3. 调用init\_lcdc\_device\_driver函数，该函数为dev\_drv的screen0申请内存空间，并获取前面screen已经已经初始化的rk\_screen，初始化screen0，该函数如下所示，已经省掉一部分代码：

static int init\_lcdc\_device\_driver(struct rk\_drm\_screen\_private \*screen\_priv,

struct rk\_lcdc\_win \*def\_win, int index)

{

struct rk\_drm\_private \*rk\_drm\_priv = platform\_get\_drvdata(drm\_fb\_pdev);

struct rk\_lcdc\_driver \*dev\_drv = screen\_priv->lcdc\_dev\_drv;

struct rk\_screen \*screen1 = NULL;

struct rk\_screen \*screen = devm\_kzalloc(dev\_drv->dev,

sizeof(struct rk\_screen), GFP\_KERNEL);

--cut--

dev\_drv->screen0 = screen;

dev\_drv->cur\_screen = screen;

/\* devie use one lcdc + rk61x scaler for dual display\*/

if (rk\_drm\_priv->disp\_mode == ONE\_DUAL) {

screen1 = devm\_kzalloc(dev\_drv->dev,

sizeof(struct rk\_screen), GFP\_KERNEL);

if (!screen1) {

dev\_err(dev\_drv->dev, "malloc screen1 for lcdc%d fail!",

dev\_drv->id);

goto fail\_screen1;

}

screen1->screen\_id = 1;

screen1->lcdc\_id = 1;

dev\_drv->screen1 = screen1;

}

--cut--

if (dev\_drv->prop == PRMRY) {

rk\_fb\_set\_prmry\_screen(screen);

rk\_fb\_get\_prmry\_screen(screen);

dev\_drv->trsm\_ops = rk\_fb\_trsm\_ops\_get(screen->type);

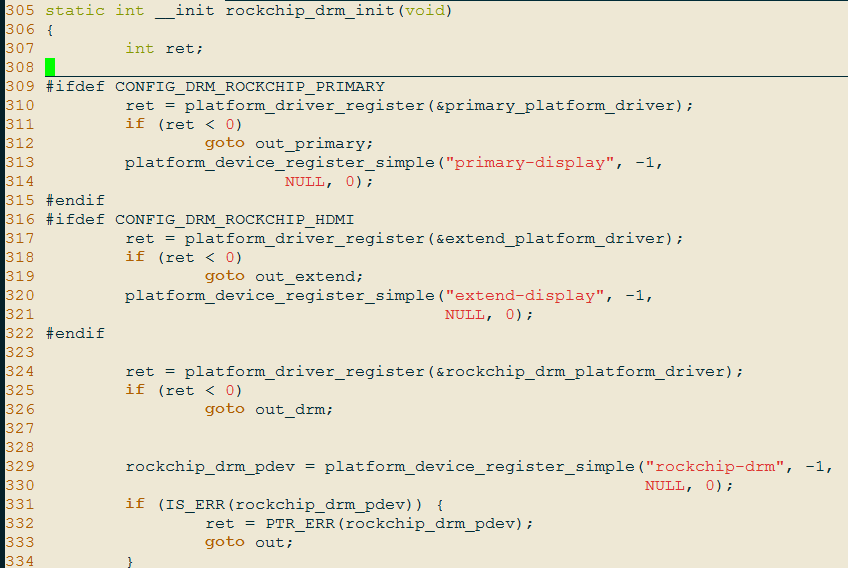
}

--cut--

}

1. 为screen申请内存空间，初始化screen的一部分成员，dev\_drv的screen0指向刚初始化的screen；
2. 如果disp\_mode为ONE\_DUAL模式，与screen0一样，初始化screen1；
3. 接下来，也是要非常留意的地方，就是在这里，建立了底层核心管理与screen之间的联系，rk\_fb\_get\_prmry\_screen获取rk\_screen（即前面screen一节初始化的screen）给screen；
4. 最后，这里也需要非常留意，rk\_fb\_trsm\_ops\_get获取transmitter注册的回调函数，至此，已经完成底层各模块之间的联系。
5. 完成第三步后，根据获取到的screen，建立fb\_videomode。
6. **Drm框架部分**

文件：drivers/gpu/drm/rockchip/rockchip\_drm\_drv.c;



这里只对primary display做讨论，不对extend display做讨论。rockchip\_drm\_init分别注册了primary display和rockchip-drm的platform\_device和platform\_driver。

primary display的probe函数流程图如下所示：

primary\_probe

------> rk\_drm\_get\_display(RK\_DRM\_PRIMARY\_SCREEN)获取rk\_fb\_register初始化的drm\_disp；

------> 获取drm\_disp的modelist，初始化ctx->panel的timing，这样其它部分可以通过这里获取panel信息；

------> 初始化rockchip\_drm\_subdrv，包括manager、probe和remove，drm的crtc、encoder和connector都是通过manager提供的回调函数操作底层硬件模块的；

------> rockchip\_drm\_subdrv\_register 注册subdrv，subdrv将会与drm\_dev匹配，匹配后执行subdrv的probe；

rockchip-drm的platform\_driver的probe函数为rockchip\_drm\_platform\_probe，下面为probe的流程：

rockchip\_drm\_platform\_probe

drm\_platform\_init

drm\_platform\_init为drm框架的接口，参数为drm\_driver（rockchip\_drm\_driver）和pdev

drm\_get\_platform\_dev

调用drm\_fill\_in\_dev初始化drm\_device，drm\_device的driver字段指向drm\_driver,

然后调用drm\_driver->bus->agp\_init；

drm\_device->driver->load

这里的driver为drm\_driver，load回调函数为rockchip\_drm\_load

------> rockchip\_drm\_crtc\_create **/\* crtc的创建 \*/**

创建crtc，这里的crtc->funcs为rockchip\_crtc\_funcs，

private为rockchip\_crtc\_helper\_funcs。

------> rockchip\_drm\_device\_register **/\* encoder和connector的创建 \*/**

1. 与前面创建的subdrv匹配，调用subdrv的probe函数；
2. 调用rockchip\_drm\_create\_enc\_conn，创建encoder和connector;
3. 创建encoder的函数为rockchip\_drm\_encoder\_create，encoder的manager字段指向subdrv的manage，这样，encoder通过manager就可以操作底层。encoder的funcs为rockchip\_encoder\_funcs，private为rockchip\_encoder\_helper\_funcs；
4. 同样，connector的manager指向subdrv的manage，connector的funcs为rockchip\_connector\_funcs，private为rockchip\_connector\_helper\_funcs；
5. 最后，subdrv的encoder和connector分别指向上面创建的encoder和connector；

------> rockchip\_drm\_fbdev\_init **/\* fbinfo的创建与注册 \*/**

1. 初始化drm\_fb\_helper——helper，helper的funcs为rockchip\_drm\_fb\_helper\_funcs，funcs只有fb\_probe回调，fb\_probe主要是创建和初始化fbinfo，初始化fbinfo的fbops为rockchip\_drm\_fb\_ops，下面会调用helper的funcs回调，以创建fbinfo；
2. drm\_fb\_helper\_initial\_config

--> drm\_fb\_helper\_probe\_connector\_modes /\* fbvideomode的获取 \*/

调用connector->funcs->fill\_modes，fill\_modes为rockchip\_drm\_connector\_fill\_modes，fill\_modes继续调用drm\_helper\_probe\_single\_connector\_modes，drm…modes继续调用connector->helper\_private->get\_modes获取drm\_display的videomode；

--> drm\_setup\_crtcs

1. 把获取到的video\_mode放到crtc的desired\_mode，最后放到mode\_set。

--> drm\_fb\_helper\_single\_fb\_probe

1. 根据前面的video\_mode，初始化sizes变量，以fb\_helper和sizes为参数，调用fb\_helper->funcs->fb\_probe回调函数，创建fbinfo；
2. 调用register\_framebuffer，注册framebuffer；

fbinfo的fb\_set\_par回调函数为drm\_fb\_helper\_set\_par，下面以mode\_change来分析fb\_set\_par流程，流程如下所示：

drm\_fb\_helper\_set\_par

drm\_mode\_set\_config\_internal

crtc->funcs->set\_config

set\_config为drm\_crtc\_helper\_set\_config

drm\_crtc\_helper\_set\_config

drm\_crtc\_helper\_set\_mode

------> crtc\_funcs->mode\_fixup crtc\_funcs为crtc->helper\_private

1. crtc\_funcs->mode\_fixup为rockchip\_drm\_crtc\_mode\_fixup，该函数只是直接返回true而已。

------> encoder->helper\_privatte->mode\_fixup

------> encoder->helper\_privatte->prepare

------> crtc\_funcs->prepare

------> crtc\_funcs->mode\_set

1. mode\_set回调函数为rockchip\_drm\_crtc\_mode\_set，该函数会通过manage，调用subdrv的manger->overlay\_ops->mode\_set；

------> encoder->helper\_private->mode\_set

------> crtc\_funcs->commit

------> encoder->helper\_private->commit

1. 该commit为rockchip\_drm\_encoder\_commit

------> manager\_ops->commit

1. 这里的manager\_ops为subdrv的manager的ops字段，subdrv的commit为primary\_commit；
2. primary\_commit调用rk\_drm\_disp\_handle，这个函数在底层的管理核心层，在此，终于把drm框架层与底层联系起来了，rk\_drm\_disp\_handle为主要的沟通函数。