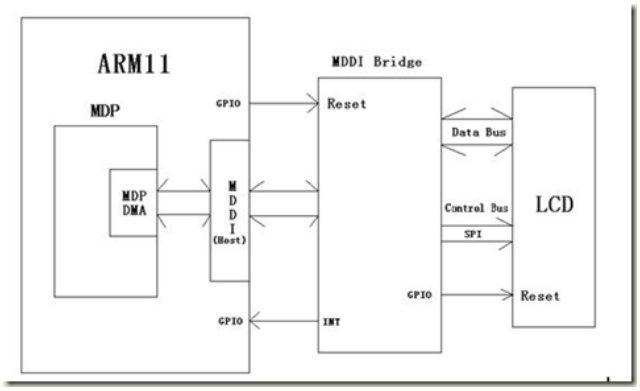
Android Display 架构解析

http://blog.csdn.net/linuxengineer/article/details/8966145
非常好的介绍android display driver的文章，不服不行。

http://blog.csdn.net/bonderwu/archive/2010/08/12/5805961.aspx

Android display 架构分析（一）
http://hi.baidu.com/leowenj/blog/item/429c2dd6ac1480c851da4b95.html
高通 **7** 系列硬件架构分析



如上图，高通7系列 Display的硬件部分主要由下面几个部分组成：

A 、MDP

高通MSM7200A内部模块，主要负责显示数据的转换和部分图像处理功能理，如YUV转RGB，放大缩小、旋转等。MDP内部的MDP DMA负责数据从DDR到MDDI Host的传输（可以完成RGB之间的转换，如RGB565转成RGB666，这个转换工能载目前的code 中没有使用）。

B 、MDDI

一种采用差分信号的高速的串行数据传输总线，只负责数据传输，无其它功能；其中的MDDI Hosat提供并行数据和串行数据之间的转换和缓冲功能。由于外面是VGA的屏幕，数据量较大，为了减少对EBI2总线的影响，传输总线使用MDDI，而非之前的EBI2。

C 、MDDI Bridge

由于现在采用的外接LCD并不支持MDDI接口，故需要外加MDDI转换器，即MDDI bridge，来把MDDI数据转换成RGB接口数据。这里采用的EPSON MDDIBridge还有LCD Controller功能，可以完成其它一些数据处理的功能，如数据格式转换、支持TV-OUT、PIP等；并且还可以提供一定数量的GPIO。目前我们 主要用它把HOST端MDDI传递过来的显示数据和控制数据（初始化配置等）转换成并行的数据传递给LCD。

D 、LCD module

主要是LCD Driver IC 和TFT Panel，负责把MDDI Bridge传来的显存中的图像示在自己的 Panel上。

Android display 架构分析（二）
http://hi.baidu.com/leowenj/blog/item/3fe59f740a6fee17b051b991.html

LINUX_PRO (6)
OS_Install_Tips (12)
C++ (60)
HR (9)
health (1)
tv&&music&&fun (9)
compile (9)
并发处理相关 (8)
const&mutable (2)
callback_func (1)
ipc (1)
camera_framework (9)
sp&wp (3)
selinux (1)
initrc (3)
cmd (2)
virtual_func (2)
stl (1)
log (2)
mediaplayer (1)
parcel (1)
property (2)
debug (1)
busybox (1)
bionic (1)
zygote (2)
surface (2)
jni (1)
笔试题目 (3)
平台设备总线模型 (1)
iic (0)
字符驱动 (1)

文章存档

2016年04月 (32)
2016年03月 (6)
2016年02月 (13)
2015年12月 (43)
2015年11月 (9)

展开

阅读排行

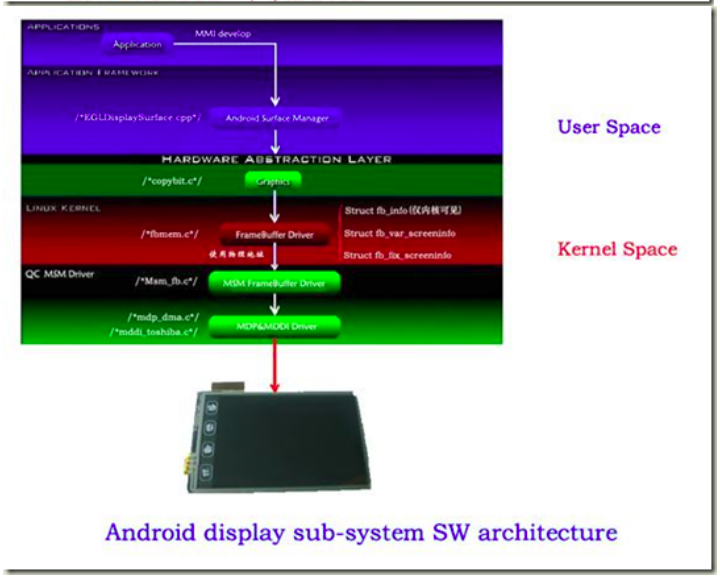
请教：过量喝咖啡对心脏 (4385)
C++和JAVA的区别 -- 给初 (3934)
svn如何取消认证缓存设置 (3792)
android TP (2739)
Android LCD (2562)
高通平台android开发总结 (2280)
Android图形合成和显示系 (2258)
一个离职员工对中兴的回 (2237)
高通QPST Download使用 (2208)
android camera (2088)

推荐文章

* CSDN日报20170828——《4个方法快速打造你的阅读清单》
* Android检查更新下载安装
* 动手打造史上最简单的Recycleview 侧滑菜单
* TCP网络通讯如何解决分包粘包问题
* 程序员的八重境界

Android display SW 架构分析

Note:粉色加重的模块是与 Display 相关的部分。



Android display sub-system SW architecture

下面简单介绍一下上图中的各个Layer：

***** 蓝色部分 - 用户空间应用程序

应用程序层，其中包括Android应用程序以及框架和系统运行库，和底层相关的是系统运行库，而其中和显示相关的就是Android的Surface Manager, 它负责对显示子系统的管理，并且为多个应用程序提 供了2D和3D图层的无缝融合。

***** 黑色部分 - **HAL**层，在**2.2.1**部分会有介绍

***** 红色部分 - **Linux kernel**层

Linux kernel，其中和显示部分相关的就是Linux的FrameBuffer，它是Linux系统中的显示部分驱动程序接口。Linux工作在保护模式 下，User空间的应用程序无法直接调用显卡的驱动程序来直接画屏，FrameBuffer机制模仿显卡的功能，将显卡硬件结构抽象掉，可以通过 Framebuffer的读写直接对显存进行操作。用户可以将Framebuffer看成是显示内存的一个映像，将其映射到进程地址空间之后，就可以直接 进行读写操作，而写操作可以立即反应在屏幕上。这种操作是抽象的，统一的。用户不必关心物理显存的位置、换页机制等等具体细节。这些都是由 Framebuffer设备驱动来完成的。

***** 绿色部分 - **HW** 驱动层

该部分可以看作高通显卡的驱动程序，和高通显示部分硬件相关以及外围LCD相关的驱动都被定义在这边，比如上述的显卡的一些特性都是在这边被初始化的，同样MDP和MDDI相关的驱动也都定义在这里

User Space Display 功能介绍

这里的User Space就是与应用程序相关的上层部分（参考上图中的蓝色部分），其中与Kernel空间交互的部分称之为**HAL** - **HW Abstraction Layer**。

HAL其实就是用户空间的驱动程序。如果想要将 Android 在某硬件平台上执行，基本上完成这些驱动程序就行了。其内定义了 Android 对各硬件装置例如显示芯片、声音、数字相机、GPS、GSM 等等的需求。

HAL存在的几个原因：

1、 并不是所有的硬件设备都有标准的linux kernel的接口。

2、 Kernel driver涉及到GPL的版权。某些设备制造商并不原因公开硬件驱动，所以才去HAL方式绕过GPL。

关闭

* 四大线程池详解

3、 针对某些硬件，Android有一些特殊的需求。
在display部分，HAL的实现code在copybit.c中，应用程序直接操作这些接口即可，具体的接口如下：

```
struct
 copybit_context_t *ctx = malloc(sizeof
(struct
 copybit_context_t));

memset(ctx, 0, sizeof
(*ctx));

ctx->device.common.tag = HARDWARE_DEVICE_TAG;

ctx->device.common.version = 0;

ctx->device.common.module = module;

ctx->device.common.close = close_copybit;

ctx->device.set_parameter = set_parameter_copybit;//设置参数


ctx->device.get
 = get
;

ctx->device.blit = blit_copybit;//传送显示数据


ctx->device.stretch = stretch_copybit;

ctx->mAlpha = MDP_ALPHA_NOP;

ctx->mFlags = 0;

ctx->mFD = open("/dev/graphics/fb0
", O_RDWR, 0);//打开设备
```

Kernel Space Display 功能介绍
这里的Kernel空间（与Display相关）是Linux平台下的FB设备（参考上图中的红色部分）。下面介绍一下FB设备。
Fb即FrameBuffer的简称。framebuffer 是一种能够提取图形的硬件设备，是用户进入图形界面很好的接口。有了framebuffer，用户的应用程序不需要对底层驱动有深入了解就能够做出很好的 图形。对于用户而言，它和/dev下面的其他设备没有什么区别，用户可以把
framebuffer 看成一块内存，既可以向这块内存中写入数据，也可以从这块内存中读取数据。它允许上层应用程序在图形模式下直接对显示缓冲区进行读写操作。这种操作是抽象 的，统一的。用户不必关心物理显存的位置、换页机制等等具体细节。这些都是由Framebuffer设备驱动来完成的。
从用户的角度看，帧缓冲设备和其他位于/dev下面的设备类似，它是一个字符设备，通常主设备号是29，次设备号定义帧缓冲的个数。
在LINUX系统中，设备被当作文件来处理，所有的文件包括设备文件，Linux都提供了统一的操作函数接口。上面的结构体就是Linux为FB设备提供的操作函数接口。
1）、读写（ **read/write** ）接口 ，即读写屏幕缓冲区（应用程序不一定会调用该接口）
2）、映射（ **map** ）操作 （用户空间不能直接访问显存物理空间，需map成虚拟地址后才可以）
由于Linux工作在保护模式，每个应用程序都有自己的虚拟地址空间，在应用程序中是不能直接访问物理缓冲区地址的。为此，Linux在文件操作 file_operations结构中提供了mmap函数，可将文件的内容映射到用户空间。对于帧缓冲设备，则可通过映射操作，可将屏幕缓冲区的物理地址 映射到用户空间的一段虚拟地址中，之后用户就可

关闭

以通过读写这段虚拟地址访问屏幕缓冲区，在屏幕上绘图了。实际上，使用帧缓冲设备的应用程序都是通过映射操 作来显示图形的。由于映射操作都是由内核来完成，下面我们将看到，帧缓冲驱动留给开发人员的工作并不多

3）、**I/O** 控制： 对于帧缓冲设备，对设备文件的ioctl操作可读取/设置显示设备及屏幕的参数，如分辨率，显示颜色数，屏幕大小等等。ioctl的操作是由底层的驱动程序来完成

Note：上述部分请参考文件fbmem.c。

Android display 架构分析（三）

http://hi.baidu.com/leowenj/blog/item/76411bf6237dc429bc31099f.html

Kernel Space Display 架构介绍

如上图所示，除了上层的图形应用程序外，和 **Kernel** 空间有关的包括 **Linux FB** 设备层以及和具体 **HW** 相关的驱动层，对应的源文件分别是**fb_mem.c** 、 **msm_fb.c** 、 **mddi_toshiba.c** 。下面会一一介绍。

函数和数据结构介绍

这个文件包含了 **Linux Fb** 设备的所有接口，主要函数接口和数据结构如下：

A 、 **Fb** 设备的文件操作接口

帧缓冲驱动的编写

帧缓冲设备属于字符设备，与声音设备一样，也采用“文件层-驱动层”的接口方式。

在文件层次上，Linux 为其定义了

```
static struct file_operations fb_fops = {
    owner:  THIS_MODULE,
    read:   fb_read,        /*  读操作      */
    write:  fb_write,       /*  写操作      */
    ioctl:  fb_ioctl,       /*  控制操作    */
    mmap:   fb_mmap,        /*  映射操作    */
    open:   fb_open,        /*  打开操作    */
    release:fb_release,     /*  关闭操作    */
};
```

其中的成员函数都在文件 linux/driver/video/fbmem.c 中定义。

B 、 **3** 个重要的数据结构

FrameBuffer 中有 **3** 个重要的结构体， **fb.h** 中定义，如下：

1 ）、**frame_var_screeninfo**

该结构体定义了显卡的一些可变的特性，这些特性在程序运行期间可以由应用程序动态改变，比较典型的如 **xrex** 和 **yres** 表示在显示屏上显示的真实分辨率、显示的 **bit** 数等，该结构体 **user space** 可以访问。

2 ）、**frame_fix_screeninfo**

该结构体定义了显卡的一些固定的特性，这些特性在硬件初始化时就被定义了以后不可以更改。其中最重要的成员就是 **smem_len** 和 **smem_start**，前者指示显存的大小（目前程序中定义的显存大小为整屏数据 **RGB565** 大小的 **2** 倍）**,** 后者给出了显存的物理地址。该结构体 **user space** 可以访问。

Note ： **smem_start** 是显存的物理地址，应用程序是不可以直接访问的，必须通过 **fb_ops** 中的 **mmp** 函数映射成虚拟地址后，应用程序方可访问。

3 ）、**fb_info**

FrameBuffer 中最重要的结构体，它只能在内核空间内访问。内部定义

关闭

列 **FrameBuffer** 的操作函数，**Open/read/write** 、地址映射等）**.**

C 、其他

1 ）、一个重要的全局变量

struct fb_info *registered_fb[FB_MAX];

这变量记录了所有 **fb_info** 结构的实例， **fb_info** 结构描述显卡的当前状态，所有设备对应的 **fb_info** 结构都保存在这个数组中，当一个**FrameBuffer** 设备驱动向系统注册自己时，其对应的 **fb_info** 结构就会添加到这个结构中，同时 **num_registered_fb** 为自动加 **1** 。

2 ）、注册 **framebuffer** 函数

register_framebuffer(struct fb_info *fb_info);

unregister_framebuffer(struct fb_info *fb_info);

这两个是提供给下层 **FrameBuffer** 设备驱动的接口，设备驱动通过这两函数向系统注册或注销自己。几乎底层设备驱动所要做的所有事情就是填充**fb_info** 结构然后向系统注册或注销它

Android display 架构分析（四）

http://hi.baidu.com/leowenj/blog/item/37e1a8521e35522842a75b99.html

函数和数据结构介绍

该文件为高通显卡的驱动文件，比较重要的函数接口和数据结构如下：

A 、高通 **msm fb** 设备的文件操作函数接口

```
static struct fb_ops msm_fb_ops = {
.owner = THIS_MODULE,
.fb_open = msm_fb_open,
.fb_release = msm_fb_release,
.fb_read = NULL,
.fb_write = NULL,
.fb_cursor = NULL,
.fb_check_var = msm_fb_check_var,     /* 参数检查 */
.fb_set_par = msm_fb_set_par,       /* 设置显示相关参数 */
.fb_setcolreg = NULL, /* set color register */
.fb_blank = NULL,       /* blank display */
.fb_pan_display = msm_fb_pan_display,       /* 显示 */
.fb_fillrect = msm_fb_fillrect,     /* Draws a rectangle */
.fb_copyarea = msm_fb_copyarea, /* Copy data from area to another */
.fb_imageblit = msm_fb_imageblit,   /* Draws a image to the display */
.fb_cursor = NULL,
.fb_rotate = NULL,
.fb_sync = NULL, /* wait for blit idle, optional */
.fb_ioctl = msm_fb_ioctl,    /* perform fb specific ioctl (optional) */
.fb_mmap = NULL,
};
```

B 、高通 **msm fb** 的 **driver** 接口

```
static struct platform_driver msm_fb_driver = {
.probe = msm_fb_probe,// 驱动探测函数
.remove = msm_fb_remove,
#ifndef CONFIG_ANDROID_POWER
.suspend = msm_fb_suspend,
.suspend_late = NULL,
.resume_early = NULL,
.resume = msm_fb_resume,
#endif
.shutdown = NULL,
.driver = {
/* Driver name must match the device name added in platform.c. */
.name = "msm_fb",
},
};
```

C 、 **msm_fb_init**（）

向系统注册 **msm fb** 的 **driver** ，初始化时会调用

D 、 **msm_fb_add_device**

向系统中添加新的 **lcd** 设备，在 **mddi_toshiba.c** 中会被调用

函数和数据结构介绍

关闭

该文件包含了所有和具体 **LCD**（ **Toshiba** ）相关的信息和驱动，重点的数据结构和函数结构如下：

A 、 **LCD** 设备相关信息

```
static struct platform_device this_device_0 = { p>
.name   = "mddi_toshiba_vga",
.id   = TOSHIBA_VGA_PRIM,
.dev      = {
.platform_data = &toshiba_panel_data0,
}
};
```

其中 **toshiba_panel_data0** 包含了硬件 **LCD** 的控制函数，如开关、初始化等等

B 、 **LCD driver** 接口

```
static struct platform_driver this_driver = {
.probe = mddi_toshiba_lcd_probe,
.driver = {
.name   = "mddi_toshiba_vga",
},
};
```

其中 **mddi_toshiba_lcd_probe** 中会调用 **msm_fb_add_device** 接口把具体 **LCD** 添加到系统中去。

C 、 **mddi_toshiba_lcd_init**

注册 **LCD** 设备及 **driver** 到系统中去，同时也把 **LCD** 的固有信息（大小、格式、位率等）一并注册到系统中去。

D 、 **LCD** 相关控制函数

toshiba_common_initial_setup（ ）：初始化 **MDDI bridge**

toshiba_prim_start（ ）：初始化 **LCD**

数据流分析

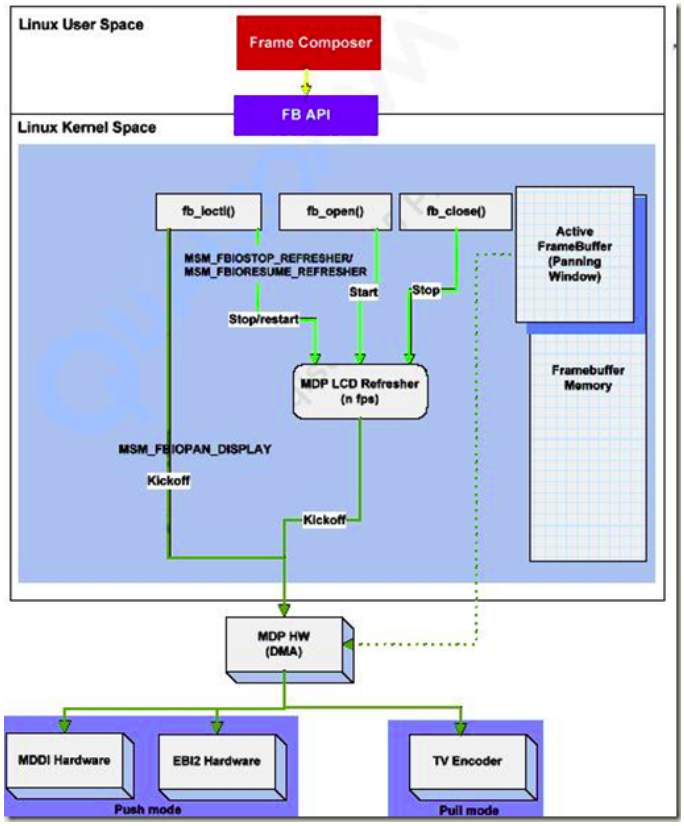
本部分来看一下应用层以下，显示数据的流程是怎样的。

先来分析一下传统的 **Linux** 平台下 **FB** 设备是如果调用的，如下图所示：

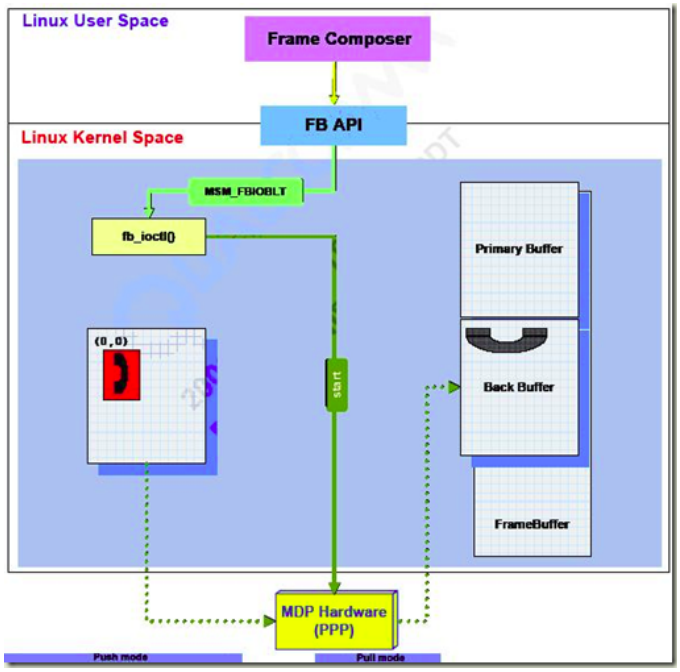
上层调用 **FB API**（主要是 **fb_ioctl()** ）， **fb_ioctl()** 会调用具体显卡的驱动，这里是高通的显卡驱动，其实就是 **MDP DMA** 的驱动，通过 **MDP DMA** 把显示数据经 **MDDI** 接口送到外围 **LCD** 组件。

Note ：这里的 **MDP DMA** 并不对数据进行任何处理（可以完成简单的格式转换，如 **RGB565->RGB666** ）。

关闭



接下来再分析一下 **Android** 平台下显示数据是如何处理的，如下图所示：



同样上层也是调用 **FB API** ，不过这里其实把 **FB bypass** 了，相当于直接调用的是高通 **MDP PPP** 的驱动，然后

关闭

数据经 **PPP** 处理后再经 **MDDI** 接口送出到外围 **LCD** 组件。

Note ：这里的 **MDP PPP** 可以完成很多显示数据处理功能，

如 **YUV->RGB** 、 **Scale** 、 **Rotate** 、 **Blending** 等。

初始化过程分析

Kernel 部分 **display** 的初始化包含下面几个步骤：

1 ）、在 **linux fb** 设备初始化时会向系统中注册 **msm_fb_driver** 。 **Name** 为 **msm_fb** 。

msm_fb_init - > msm_fb_register_driver-> platform_driver_register(&msm_fb_driver)

其中的 **probe** 函数会对 **msm fb** 进行初始化，分配显存等（见 **msm_fb_probe** 函数）。

2 ）、在 LCD 模块初始化时会先向系统中注册驱动（在 mddi_toshiba_lcd_init 函数中）
platform_driver_register(&this_driver); 名字为 mddi_toshiba_vga ；
this_driver 的 probe 函数为 mddi_toshiba_lcd_probe ，其内部会调用 msm_fb_add_device 向系统中添加 MSM fb 设备。
3 ）、调用 platform_device_register(&this_device_0) 向系统中注册设备，名字为 mddi_toshiba_vga ，其中 this_device_0 包含了一些操作LCD 的接口，如 on/off 。
Note: 设备和 driver 的 name 需要一致才可以绑定；另外，如果某些设备不需要让 platform 的总线来管理，那么只需要注册驱动即可，而无须向系统中注册 device ，如 msm_touch 。

Android display 架构分析（五）

http://hi.baidu.com/leowenj/blog/item/7a12ecb77067737f8ad4b266.html

Display 接口介绍

、User Space display接口

在 Android 平台下，应用程序面对的显示部分的接口就是 HAL ，参考 copybit.c ，具体接口如下介绍：
open_copybit
初始化相关变量，并调用 open("/dev/graphics/fb0", O_RDWR, 0); 打开 fb 设备。
set_parameter_copybit
设置各种操作参数，如 rotate 、 alpha 、 dither 等。
stretch_copybit
Copy 一块数据（ Rectangle ）到显存，然后并命令 msm_fb 进行显示。
close_copybit
调用 close(ctx->mFD); 关闭 fb 设备。
Note ：另外，应用程序在使用上面接口之前，需要调用 mapFrameBuffer 接口（ EGLDisplaySurface.cpp ），其功能如下：
1 、 初始化显示相关参数，并设置到底层。
2 、 映射出显存的虚拟地址。

、Kernel display接口

Kernel 部分显示的接口全部都在 fbmem.c 中，这里详细介绍一下：
fb_open
打开 Linux 下 fb 设备。
fb_read/fb_write
读写显存中的数据
fb_ioctl
对显示设备的命令操作。如 get 或 set 一些显示参数、通知底层进行刷屏等。
在典型应用中，画屏的一般步骤如下：
1 ． 打开 /dev/fb 设备文件。
2 ． 用 ioctrl 操作取得当前显示屏幕的参数，如屏幕分辨率，每个像素点的比特数。根据屏幕参数可计算屏幕缓冲区的大小。
3 ． 将屏幕缓冲区映射到用户空间。
4 ． 映射后就可以直接读写屏幕缓冲区，进行绘图和图片显示了。
典型程序段如下：

```
#include
int main()
{
int fbfd = 0;
struct fb_var_screeninfo vinfo;
struct fb_fix_screeninfo finfo;
long int screensize = 0;
/* 打开设备文件 */
fbfd = open("/dev/fb0", O_RDWR);
/* 取得屏幕相关参数 */
ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo); ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo);
/* 计算屏幕缓冲区大小 */
```

关闭

```
screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;
/* 映射屏幕缓冲区到用户地址空间 */
fbp=(char*)mmap(0,screensize,PROT_READ|PROT_WRITE,MAP_SHARED, fbfd, 0);
/* 下面可通过 fbp 指针读写缓冲区 */
...
}
```

典型应用flow分析

在不同应用程序中，上层的调用会有所不同，比如 **Andriod** 下会选择应用程序跳过 **Linux fb** 操作层，直接操作显卡驱动层，称之为 **BLT accelerator** 。

下面看一下 **Android** 平台下画屏的操作流程。

1 、 通过**mapFrameBuffer** 直接把用户空间的数据映射到显存中。

2 、 调用 **HAL** 中的 **stretch** 函数直接命令 **MSM** 设备提取显存数据然后送入 **MDP PPP** 进行处理后输出送到外围 **LCD** 组件。

具体的函数调用流程如下：

copybit_open（ ）； **//** 打开 **BlitEngine** ，同时也打开 **fb** 设备

mapFrameBuffer();// 设置显示参数，同时得到显存虚拟地址

copybit->stretch(copybit, &dst, &src, &sdrect, &sdrect, &it);// 通知底层去刷屏

接下的流程是：

stretch_copybit-> msm_copybit-> fb_ioctl()->msm_fb_ioctl(MSMFB_BLIT)-> msmfb_blit-> mdp_blit-> mdp_ppp_blit->mdp_start_ppp->MDP&MDDI HW operation

Android display 架构分析（六）

http://hi.baidu.com/leowenj/blog/item/78c068dc443c961f48540361.html

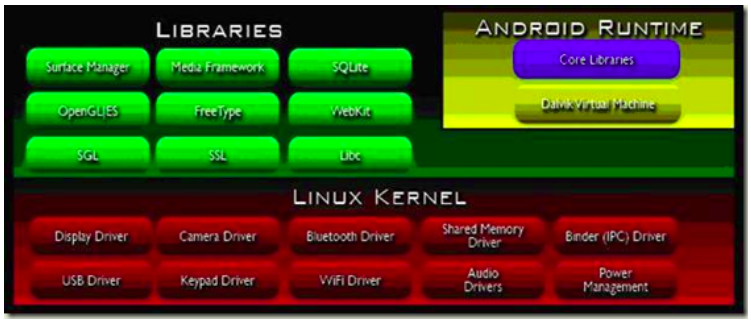
介绍

Note：

本部分介绍的完全是用户空间显示部分的架构，与kernel并没有直接的联系，主要是JNI以下到HAL以上的部分。

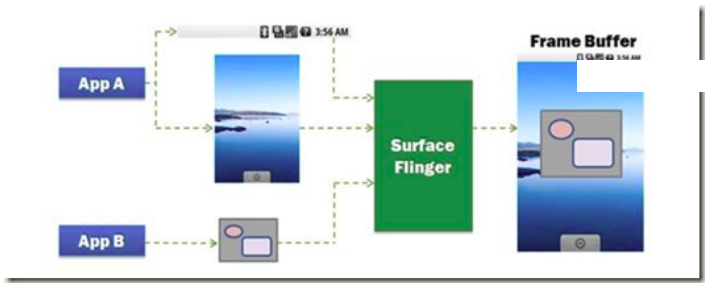
、**Surface manager**（**surface flinger**）简介

Surface manager是用户空间中framework下libraries中负责显示相关的一个模块。如下：

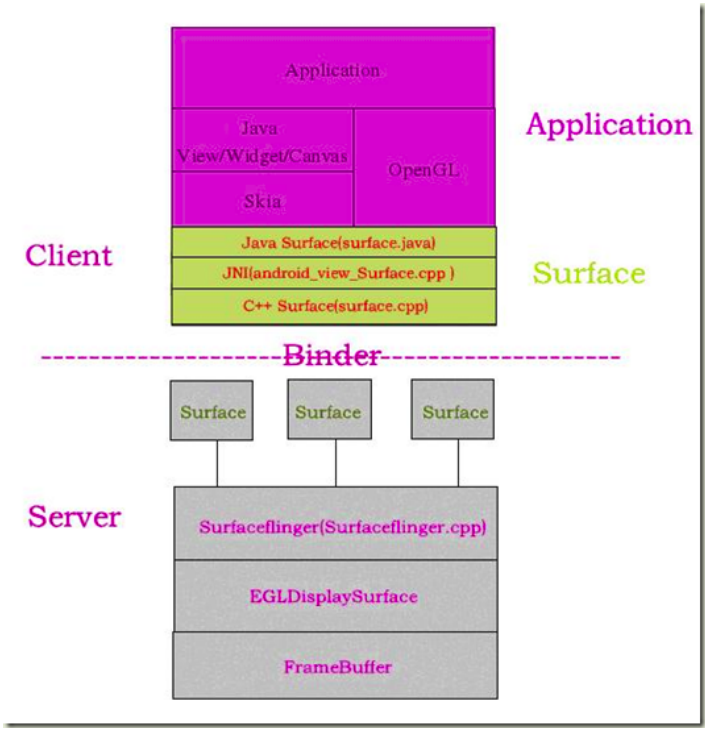


当系统同时执行多个应用程序时，Surface Manager会负责管理显示与存取操作间的互动，另外也负责将2D绘图与3D绘图进行显示上的合成。

surface manager 可以准备一块 surface（可以看作一个layer），把 surface 的 fd (一块内存) 传给一个 app，让 app 可以在上面作画。典型应用如下：



关闭



2 、 架构分析

Android中的图形系统采用Client/Server架构，如下：

Client 端 ：应用程序相关部分。代码分为两部分，一部分是由Java提供的供应用使用的api,另一部分则是由c++写成的底层实现。

Server 端 ：即SurfaceFlinger，负责合成并送入buffer显示。其主要由c++代码编写而成。

Client和Server之前通过**Binder** 的IPC方式进行通信，总体结构图如下：

如上图所示，Surface的client部分其实是提供给各应用程序进行画图操作的一个桥梁，该桥梁通过binder通向server端的 Surfaceflinger，Surfaceflinger负责合成各个surface，然后把buffer传送到framebuffer端进行底层显示。其中每个surface对应2个buffer，一个front buffer, 一个back buffer，更新时，数据更新在back buffer上，需要显示时，则将back buffer和front buffer互换。

下一部分我们重点研究一下Surfaceflinger。

Android display 架构分析（七 **-1** ）

http://hi.baidu.com/leowenj/blog/item/7abbe33a309367ff3b87ce6f.html

流程分析

根据前面的介绍，**surfaceflinger**作为一个**server process**，上层的应用程序（作为**client**）通过**Binder**方式与其进行通信。

Surfaceflinger作为一个**thread**，这里把它分为**3**个部分，如下：

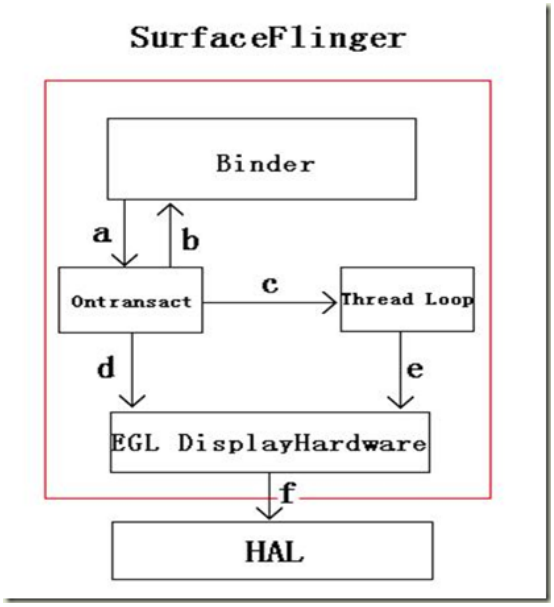
1、 Thread本身处理部分，包括初始化以及thread loop。

2、 Binder部分，负责接收上层应用的各个设置和命令，并反馈状态标志给上层。

3、 与底层的交互，负责调用底层接口（HAL）。

结构图如下：

关闭



注释：

a、 Binder接收到应用程序的命令（如创建surface、设置参数等），传递给flinger。

b、 Flinger完成对应命令后将相关结果状态反馈给上层。

c、 在处理上层命令过程中，根据需要设置event（主要和显示有关），通知Thread Loop进行处理。

d、 Flinger根据上层命令通知底层进行处理（主要是设置一些参数，Layer、position等）

e、 Thread Loop中进行surface的合成并通知底层进行显示（Post buffer）。

f、 DisplayHardware层根据flinger命令调用HAL进行HW的操作。

下面来具体分析一些SurfaceFlinger中重要的处理函数 以及**surface** 、 **Layer** 的属性

1）、**readToRun**

***SurfaceFlinger thread** 的初始化函数，主要任务是分配内存和设置底层接口 **(EGL&HAL)** 。*

```
status_t SurfaceFlinger::readyToRun()
 {
…
…
mServerHeap = new MemoryDealer(4096, MemoryDealer::READ_ONLY);//为IPC分配共享内存
…
mSurfaceHeapManager = new SurfaceHeapManager(this, 8 << 20);//为flinger分配heap，大小为8M，存放具体的显示数据
{
// initialize the main display
GraphicPlane& plane(graphicPlane(dpy));
DisplayHardware* const hw = new DisplayHardware(this, dpy);
plane.setDisplayHardware(hw);//保存显示接口
}
//获取显示相关参数
const GraphicPlane& plane(graphicPlane(dpy));
const DisplayHardware& hw = plane.displayHardware();
const uint32_t w = hw.getWidth();
const uint32_t h = hw.getHeight();
const uint32_t f = hw.getFormat();
…
…
// Initialize OpenGL|ES
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, 0);
glTexParameterx(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameterx(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

关闭

```
glTexParameterx(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
…
…
}
```

2）、ThreadLoop

***Surfaceflinger** 的 **loop** 函数，主要是等待其他接口发送的 **event** ，进行显示数据的合成以及显示。*

```
bool SurfaceFlinger::threadLoop()
{
waitForEvent();//等待其他接口的signal event
…
…
// post surfaces (if needed)
handlePageFlip();//处理翻页机制
const DisplayHardware& hw(graphicPlane(0).displayHardware());
if (LIKELY(hw.canDraw()))
{
// repaint the framebuffer (if needed)
handleRepaint();//合并所有layer并填充到buffer中去
…
…
postFramebuffer();//互换front buffer和back buffer，调用EGL接口进行显示
}
…
…
}
```

3）、createSurface

*提供给应用程序的主要接口，该接口可以创建一个 **surface** ，底层会根据参数创建 **layer** 以及分配内存， **surface** 相关参数会反馈给上层*

```
sp SurfaceFlinger::createSurface(ClientID clientId, int pid,
ISurfaceFlingerClient::surface_data_t* params,
DisplayID d, uint32_t w, uint32_t h, PixelFormat format,
uint32_t flags)
{
…
…
int32_t id = c->generateId(pid);
if (uint32_t(id) >= NUM_LAYERS_MAX) //NUM_LAYERS_MAX=31
{
LOGE("createSurface() failed, generateId = %d", id);
return
}
…
layer = createNormalSurfaceLocked(c, d, id, w, h, format, flags);//创建layer，根据参数（宽高格式）分配内存（共2个buffer：front/back buffer）
if (layer)
{
setTransactionFlags(eTransactionNeeded);
surfaceHandle = layer->getSurface();//创建surface
if (surfaceHandle != 0)
surfaceHandle->getSurfaceData(params);//创建的surface参数反馈给应用层
}
}
```

待续。。。

Android display 架构分析（七 - **2** ）

http://hi.baidu.com/leowenj/blog/item/ba4c5d6378a5da48eaf8f86a.html

关闭

4）、 **setClientState**

*处理上层的各个命令，并根据 **flag** 设置 **event** 通知 **Threadloop** 进行处理*

```
status_t SurfaceFlinger::setClientState(
ClientID cid,
int32_t count,
const layer_state_t* states)
{
Mutex::Autolock _l(mStateLock);
uint32_t flags = 0;
cid <<= 16;
for (int i=0 ; i
{
const layer_state_t& s = states[i];
LayerBaseClient* layer = getLayerUser_l(s.surface | cid);
if (layer)
{
const uint32_t what = s.what;
     // 检测应用层是否设置各个标志，如果有则通知底层完成对应操作，并通知ThreadLoop做对应的处理
   if (what & eDestroyed) //删除该层Layer
    {
if (removeLayer_l(layer) == NO_ERROR)
  {
flags |= eTransactionNeeded;
continue;
}
}
if (what & ePositionChanged) //显示位置变化
    {
if (layer->setPosition(s.x, s.y))
flags |= eTraversalNeeded;
}
if (what & eLayerChanged) //Layer改变
    {
if (layer->setLayer(s.z))
   {
mCurrentState.layersSortedByZ.reorder(
layer, &Layer::compareCurrentStateZ);
flags |= eTransactionNeeded|eTraversalNeeded;
}
}
if (what & eSizeChanged)
     {
if (layer->setSize(s.w, s.h))//设置宽高变化
flags |= eTraversalNeeded;
}
if (what & eAlphaChanged) {//设置Alpha效果
if (layer->setAlpha(uint8_t(255.0f*s.alpha+0.5f)))
                flags |= eTraversalNeeded;
}
if (what & eMatrixChanged) {//矩阵参数变化
if (layer->setMatrix(s.matrix))
flags |= eTraversalNeeded;
}
if (what & eTransparentRegionChanged) {//显示区域变化
if (layer->setTransparentRegionHint(s.transparentRegion))
```

关闭

flags |= eTraversalNeeded;
}
if (what & eVisibilityChanged) {//是否显示
if (layer->setFlags(s.flags, s.mask))
flags |= eTraversalNeeded;
}
}
}
if (flags)
{
setTransactionFlags(flags);//通过signal通知ThreadLoop
}
return NO_ERROR;
}

5）、 **composeSurfaces**

该接口在Threadloop中被调用，负责将所有存在的surface进行合并，OpenGl模块负责这个部分。

6）、 **postFramebuffer**

该接口在Threadloop中被调用，负责将合成好的数据（存于back buffer中）推入在front buffer中，然后调用HAL接口命令底层显示。

7）、 从3中可知，上层每创建一个surface的时候，底层都会同时创建一个layer，下面看一下surface及layer的相关属性。

Note ： **code** 中相关结构体太大，就不全部罗列出来了

A、 **Surface** 相关属性（详细参考文件 **surface.h** ）

a1 ： **SurfaceID** ： 根据此ID把相关surface和layer对应起来

a2 ： **SurfaceInfo**

包括宽高格式等信息

a3 ： 2个buffer指针、buffer索引等信息

B、 **Layer** 相关属性（详细参考文件 **layer.h/layerbase.h/layerbitmap.h** ）

包括Layer的ID、宽高、位置、layer、alpha指、前后buffer地址及索引、layer的状态信息（如eFlipRequested、eBusy、eLocked等）

Android display 架构分析（八）

http://hi.baidu.com/leowenj/blog/item/03aae36137acb8d1e6113a75.html

开发的经验分享

1 Display Driver的工作内容

参考上面linux下fb设备的软件架构，可以知道，要加入一个新的MDDI 接口的LCM，Driver的工作就是要提供自己的mddi_xxxx.c（在这次porting的过程中，为了节省时间，我们直接修改了 mddi_toshiba.c），并且完成和这个lcd相关的HWr的初始化。主要的工作包括：

A、初始化和LCD / LCD背光相关的IO以及电源；

B、编写初始化函数。主要是初始化LCD控制器，这个一般LCD厂商会提供；然后分配显存，这个高通release过来的code已经包含这个动作了，最后是初始化一个fb_info的结构体，在这里主要是把LCD的一些信息登记进来。

C、把LCD的设备以及驱动注册到系统中去。（这里因为是替换现有的驱动，所以相关修改的部分不多。）

上述B、C部分代码请参考kernel/drivers/video/msm/mddi_toshiba.c。

开发过程

关闭

1.2.1 配置**Power**和**IO**

更改一些GPIO的配置以及一些电源的电平配置；然后通过实际测量，确保一下信号正常：

A、供给LCD以及MDDI Bridge的电源；

B、MDDI Bridge以及LCD reset信号；

C、控制背光IC的GPIO工作正常（背光不打开，无法调试LCD）。

1.2.2P orting LCD初始化序列

LCD init的code以及外围MDDI Bridge的初始化code，都可以之前Boston Windows Mobile系统的code base中获得；把这部分code移植到mddi_Toshiba.c中，并更改相应的图像格式、分辨率等配置，编译通过。LCD初始化部分就算基本完 成。

1.2.3 LCD初始化过程的调试

由于硬件在之前Boston load是可以工作的，可以认为硬件连接等没有问题，所以只需关注软件部分就行。
Display部分软件调试过程如下：
A、 开机后，量一下GPIO是否为code中配置预期的状态（可确保code中的
GPIO接口工作正常）；
B、 量一下各个电源是否都处于Code中定义的电平值。这些都OK后，背光
是会亮的（背光的控制比较简单，一个GPIO即可）；
C、 这个时候如果LCD以及MDDI Bridge有被正常初始化的话，屏幕上是会
看出来的。反之，如果屏幕没有显示，需要用JTAG跟一下mddi_Toshiba.c中的初始化函数是否在;
调用过。
目前版本中，是根据外围MDDI Bridge中读到的的厂商号来决定加载哪个驱动模块的。在本次调试中，bootloader中可以正确读到厂商号，所以bootloader中对于 LCD的初始化是有做的，所以屏幕看到的状态就是l
的样子（花屏）。 但Kernel起来后，并没有其他显示，用JTAG跟了后发现，Kernel中MODULE INIT中读不到正确的厂商号，所以说后面的driver没有被加载。接着发现如果在bootloader中如果不做MDDI Bridge的初始化，的话后面的MODULE INIT就可正常运行，该问题目前还没有澄清（现在暂时先把bootloader中的init disable掉）。

1.2.4 LCD的调整

初始化正常后，屏幕会显示UI的相关画面，但明显颜色、位置都不对。
这个可能是数据类型配置不对导致的，即MDP输出的类型、MDDI配置的类型以、LCD接收的类型不匹配导致，也有可能是RGB的顺序不对导致（可 配置成BGR）。经过调试后，把MDP端输出的格式配置成RGB565,同时外围MDDI Bridge以及LCD的input格式也配置成RGB565，这时显示色彩正常了。
如果位置或者方向不对，比如说上下或是左右颠倒，可以更改LCD的配置中的扫描方向即可。

1.2.5 其他

后续发现一个问题，播放video的时候颜色都是黑白的。
这个问题很容易让人误解，按照正常的理解，video decode出来的数据为YCbCr，Y为亮度信号，CbCr为色差信号，如果只有Y信号的话颜色应该就是黑白的。所以有2个怀疑点，一个是decode 出来的数据有误，另一个是MDDI Bridge误把输入的YcbCr信号当作RGB信号进行出来，这个也是有可能的。但很快第二个怀疑点被排除了（因为单更改MDDI input格式后还是不能解决问题）。
后来又详细的看了显示部分的代码，并用JTAG追踪video播放的时候用的显示接口，发现目前所有的显示接口输出的格式都是RGB格式，也就是说 在通过MDP之前YcbCr已经被转化过；而MDP里的转换功能并没有使
用，MDP只是被当作一个DMA完成数据的直接传输，文档中叫做Bypasse。
YcbCr到RGB的转换是由Android的lib来完成。发了个SR给高通，高通的回复也确认了，在6.3.50中，Android上层缺少这个lib（copybit.default.so），6.3.60之后的版本经解决了这个问题。

高通**Android**平台下关于**display**部分的几个关键问题

http://hi.baidu.com/leowenj/blog/item/06f8c0000763b37a3812bb03.html
显示部分的几个问题这几天通过实际测试澄清了一下，主要是下图中各个模块的使用状况以及HAL层几个模块的调用流程。以问题的方式描述如下：
1、 **Ap** 是怎么进行显示的？
Surfaceflinger负责所有上层的显示处理，对于AP（2D或是3D的应用程[illegible]surfaceflinger[illegible]surface，设置好参数，接下来都是统一交给surfaceflinger进行处理
2、 **Surface** 是怎么管理多个**surface**的？
不管有多少个surface，最终送到显示部分的只能是屏幕大小数据，surfaceflinger中利用MDP或是GPU进行多个surface 的合成处理，普通的合成MDP就可完成，但如果是复杂的比如3D的应用等就必须使用GPU，最终合成的好数据会被送到framebuffer中。
3、 **Framebuffer** 是什么？
Framebuffer是Linux中为显示数据分配的一块显存（fb设备中），通常大小是一整个屏幕数据的两倍，对于上层AP而言，只需要将要显 示的数据丢到framebuffer中就OK了，但此时显示数据并未真正的被送到LCD上，而是暂存在framebuffer中而已。

关闭

4、 上层是通过什么方式将显示内容送到**framebuffer**的？

有2个方式（二选一，不会同时在运行）：

A、 普通的显示，使用copybit（MDP）（未使用GPU）

Surfaceflinger通过copybit将要显示的数据送到framebuffer。

Note：copybit可以看做是MDP PPP的接口，它提供了MDP的功能，如多个layer合成，scale、rotate等。

其接口在：android/hardware/msm7k/libcopybit/copybit.cpp

B、 使用GPU（即使用图中的Graphics driver）

当进行复杂的显示处理时，比如3D的应用，GPU把处理好的数据直接丢到framebuffer中，和MDP没有任何关系

5、 Framebuffer 中的数据是如何被送到**LCD**显示的？

图中的Gralloc完成的。

Gralloc有2个功能：

一个是和copybit相同的，里面有MDP PPP的接口（目前没有使用）

另一个则是刷屏（整屏刷）的接口，即将framebuffer中的数据送到lcd上，调用的是MDP DMA的接

这部分的code在android/hardware/msm7k/libgralloc-qsd8k目录下，之前没有留意，以为没有使用

出开机初始化后就创建了disp_loop thread，里面的操作就是调用系统接口

ioctl(m->framebuffer->fd, FBIOPUT_VSCREENINFO, &m->info)

将数据送到lcd

Note ：送数据的时候是2个buffer切换的

另外，上层surfaceflinger也是通过Gralloc中的接口获知屏幕的大小，调用接口为

ioctl(fd, FBIOGET_VSCREENINFO, &info)，info中的屏幕宽高对应的就是底层driver设置的宽高值

6、 OpenGL 是什么？

它是一个图像处理引擎，当需要一些复杂的显示（2D/3D）操作时会用到它。它分为SW方案和HW方案，软件方案就是图中的libagl.so,对 应到目前项目中是libGLES_android.so，它可以完成简单的2D（文字，icon等）处理，通过trace看目前大部分显示操作都是它来完 成的。

Note ：它是软件方案，处理好的数据是通过copybit送到framebuffer的，而不是GPU。

其接口部分参考：android/frameworks/base/opengl/libagl

HW方案就是图中的Graphics driver，它通过使用GPU硬件来完成图像处理，处理后的数据直接送到framebuffer中。其接口部分参考：android/frameworks/base/opengl/libs（有几个版本）

7、 OpenGL 在项目中是如何配置的？

在android/vendor/qcom/msm7627_ffa目录下有一个egl.cfg文件，里面指定了当前版本中的OpenGL信息，目前如下：

0 0 android

0 1 adreno200

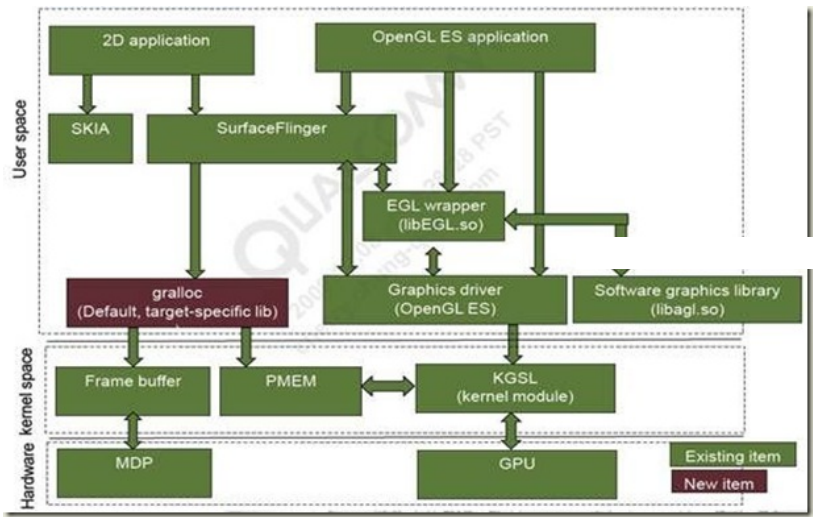
第一行代表该codebase支持SW 方案的OpenGL，是android default的

第二行代表该codebase也支持HW方案的OpenGL，是高通的adreno引擎

如果该cfg文件为空，则只支持default的SW方案。

如果2个方案都在，上层将根据实际应用自行选择使用其一。

该部分请参考：android/frameworks/base/opengl/libs/EGL/loader.cpp



关闭

顶 踩
1 0

上一篇 LCD framebuffer驱动设计文档
下一篇 Android Kernel - Boot Loader

相关文章推荐

- Android display架构分析
- Android-2.2display系统介绍-SW架构
- 轻松拿下Linux进程、线程和调度
- 一招学会Android自定义控件
- android架构解析
- 全志 H5 android linux display 驱动
- 30天掌握机器学习升级版
- 从零练就iOS高手
- Miracast_Introduce_v2_(Android_4.2_Wifi_displa…
- android binder架构
- Python网络爬虫快速入门实战
- Android DevCamp幻灯片分享：千万级并发在线…
- Android从程序员到架构师之路 完整版
- (转)Android display架构分析（一）
- 最适合自学的C++基础知识
- Android 安全架构深究

查看评论

暂无评论

该文章已被禁止评论！

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

关闭