



防爆摄像头



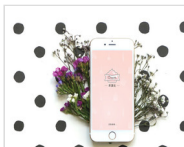
python培训



布袋风管



架构师培训



app开发报价



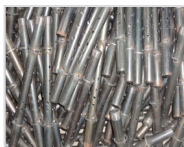
管道除铁器



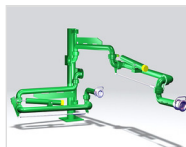
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

等级： **BLOG > 6**

排名： 第4872名

原创： 61篇 转载： 76篇

译文： 0篇 评论： 58条

文章搜索



目录视图

摘要视图

RSS 订阅

CSDN日报20170707——《稀缺：百分之二的选择》 征文 | 你会为 AI 转型么？ 每周荐书 | Android、Keras、ES6（评论送书）

## Android display架构分析-SW架构分析(1-8)

标签： [msm7k](#) [display](#)

2013-04-16 17:26

4999人阅读

[评论\(0\)](#)分类： [android display \( 24 \)](#)

目录(?)

[+]

参考：

[Android display架构分析二-SW架构分析](#)[Android display架构分析三-Kernel Space Display架构介](#)[Android display架构分析四-msm\\_fb.c 函数和数据结构介](#)[高通Android平台下关于display部分的几个关键问题](#)[高通Qc FB驱动 以及 LCD调试过程](#)[Android中的FrameBuffer\(转自 <http://disanji.net/2011/03/03/android->](#)

关闭





防爆摄像头



python培训



布袋风管



架构师培训



app开发报价



管道除铁器



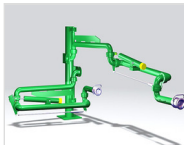
玻璃钢夹砂管



无管道新风系统



注浆管



鹤管

2014年07月 (1)

2014年06月 (1)

2014年01月 (4)

2013年12月 (1)

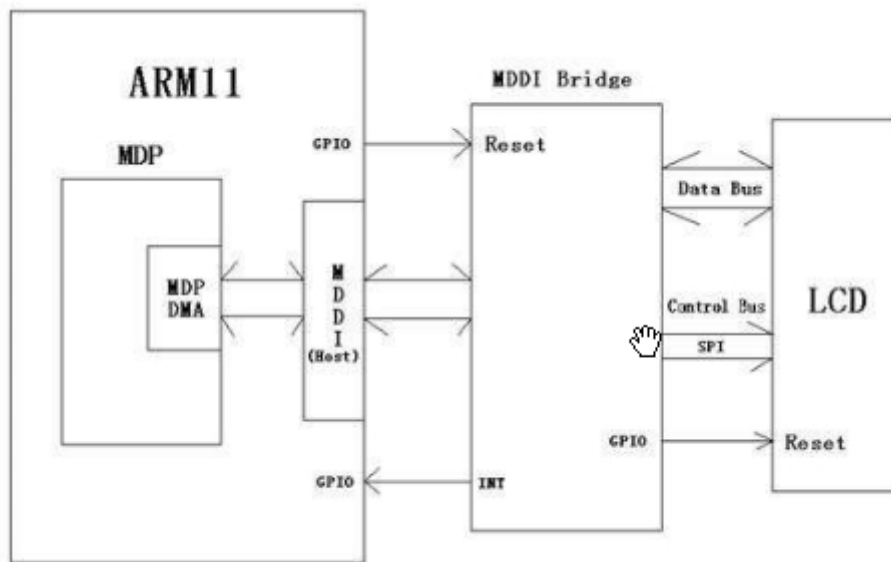
2013年11月 (3)

展开

阅读排行

## 高通7系列硬件架构分析

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放大缩小、旋转等。MDP内部的MDP DMA负责数据从DDR到MDDI的转换，如RGB565转成RGB666，这个转换功能载目前的code 中没

### 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。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 Bridge。MDDI Bridge将MDDI数据转换成RGB接口数据。这里采用的EPSON MDDIBridge还

关闭





- 内存分配器dlmalloc 2.8.1 (4)
- Android图形合成和显示 (3)
- Android编译系统分析 (3)
- GUI显示系统之SurfaceF (3)
- Android显示系统中VSYN (3)

推荐文章

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-SW架构

关闭







防爆摄像头



python培训



布袋风管



架构师培训



app开发报价




管道除铁器



玻璃钢夹砂管



无管道新风系



注浆管



鹤管

[Android上HDMI介绍 \(基于高通\)](#)  
wangguangbo1123: 请问有什么  
安卓开发板或者设备支持HDMI-  
IN吗

[Qualcomm CABL\(content adapti](#)  
岳蓬星: 宇龙的同学啊

[SIGBUS:BUS\\_ADRERR for stac](#)  
百無一用是書生: Hi, you've said  
that R0 = R6\_cur + 0x188 -  
0x104 = ...

[What SurfaceFlinger is doing wh](#)  
百無一用是書生: 敢问楼主，  
kernel的Stack是怎么打出来的。

一、 Overview



上图的原型取自高通的文档，由于原图无法描述现有的架构，我在原图的基础部分，另外其他部分根据现有的软件也做了些许改动。下面先对上图做个大概的分析。

关闭





防爆摄像头



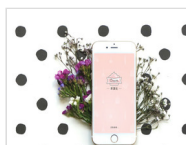
python培训



布袋风管



架构师培训



app开发报价



管道除铁器



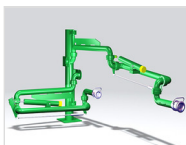
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

最上面一层为应用程序，根据数据类型以及应用的不同可以分为几种。

第一种是最普通的应用，如UI界面的显示，这部分通常数据类型为RGB格式，数据无须再经过特殊的处理。该应用可以说遍布各个应用程序，几乎是实时存在的。

第二种是针对大块YUV数据的应用，如camera的preview、视频的播放等。该应用只针对特定的应用程序，开启时通过overlay直接把大块的YUV数据送到kernel显示。

第三种其实和第一种类似，只不过由于应用的需求在显示之前需要对数据进行2D、3D的处理（使用OpenGL、OpenVG、SVG、SKIA），处理之后的流程和普通的显示就没什么差别了。一般在Game、地图、Flash等应用会用到。

应用之下是framework，其中最核心的就是surfaceflinger了，它为所有的应用程序的显示提供服务。由于接口挂在surfaceflinger里面（虽然2者在功能上不相干），所有使用overlay的AP需要通过surfaceflinger的overlay；另外，由于surfaceflinger需要使用OpenGL来compose surface，这也就是为什么surfaceflinger需要EGL wrapper了，EGL wrapper是对Graphics HAL的封装，除了surfaceflinger会调用它来compose surface层的2D、3D应用也会调用它来进行图形处理。

再下一层就是HAL了。

首先一个是overlay模块，对上提供control channel和data channel；对下则通

再一个是Gralloc模块，注意它是和overlay并列的，它包含2个部分，一部分是对framebuffer进行刷新，这里的framebuffer其实就是UI的数据。由此可见，framebuffer是传统的方式，overlay是Android（éclair以后）后增加的。

红色及右边部分是OpenGL的HAL，其中红色部分代表HW solution，高通提供software graphics library是SW solution，android自身的。HW和SW solution会讲解。

关闭





防爆摄像头



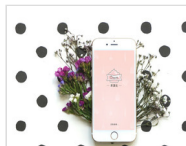
python培训



布袋风管



架构师培训



app开发报价



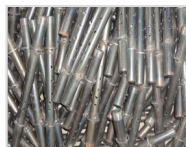
管道除铁器



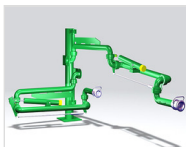
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

再往下就是kernel中的driver了，最主要的就是fb设备驱动以及MDP4 overlay的驱动，从硬件上看2者是并列的，framebuffer最终也是通过overlay方式送入MDP的。PMEM和KGSL分别对应kernel中pmem的driver（/dev/pmem）和Adreno220的driver。

## 二、Surfaceflinger详解

### 1.overview

Surfaceflinger可以说是Android显示系统中的核心，在android当中它是一个service，提供系统范围内的surface composer 功能，它能够将各种应用程序的2D、3D surface 进行组合，合并最终得到的一个main surface 送入显存。简单的说，surfaceflinger就像是画布，它不关心画上去的内容，只是一味的执行合成功能，根据画的位置、大小以及效果等参数。这很像Photoshop中的各个Layer，你可以在不同的layer画任意的内容，每个layer可以设置位置、大小、效果参数等，最终通过merge合成一个layer。

从应用的角度看，每个应用程序可能对应一个或多个图形界面，每个界面可以看作是一个surface。首先，每个surface 有它的位置、大小、内容等元素，这些元素是可以随便变化的；另外不同的surface的位置会有重叠，会涉及到透明度等效果处理问题，这些都是通过surfaceflinger来完成的。当然了，surfaceflinger担任的是一个管理的职责，它不负责效果处理及合成它是通过OpenGL来做的，但前提是surfaceflinger需要把相关参数计算好，如重叠的位置、透明度等。

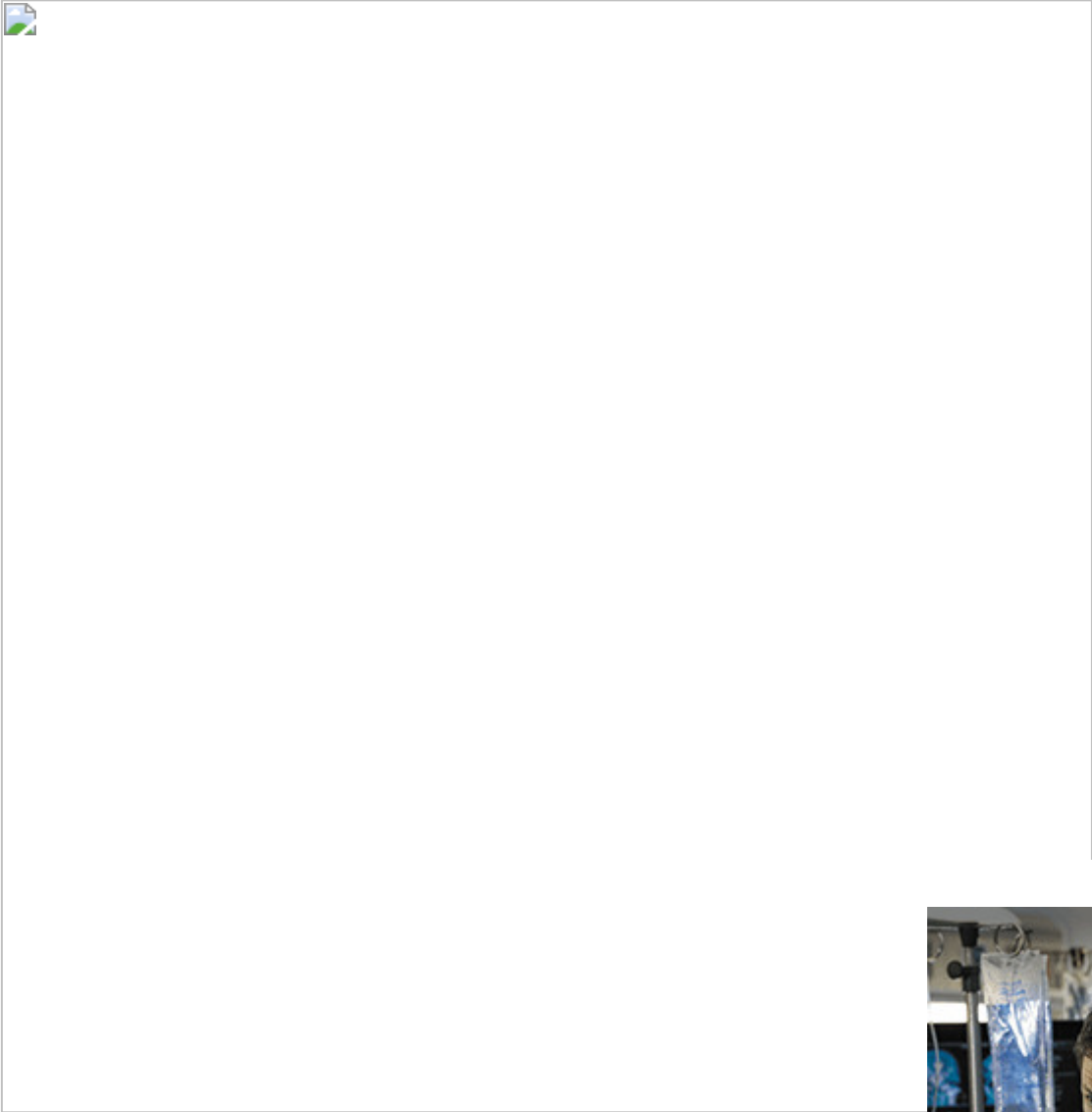
### 2.Surfaceflinger在系统中的位置

Android中的图形系统采用Client/Server架构。服务端负责Surface的合成等处，客户端负责绘制自己的Surface，并向服务端发送消息完成实际处理工作。服务端（即Surfaceflinger）端代码分为两部分，一部分是由Java提供的供应用使用的api,另一部分则是C++实现的。

关闭







除去最上层的应用不算，surface最上层的接口就是java surface了，文件路径 frameworks/base/core/java/android/view/Surface.java，该文件中的接口会被我们从JNI开始看，surface的JNI文件路径如下：



关闭



防爆摄像头



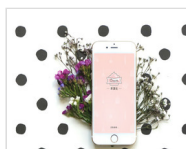
python培训



布袋风管



架构师培训



app开发报价



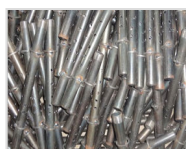
管道除铁器



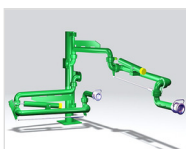
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

frameworks/base/core/jni/android\_view\_Surface.cpp，里面的接口大概分为2类，一类是负责管理ibinder通信的；另一类才是和显示控制相关的，第二类接口会直接调用C实现函数。

C实现的文件路径如下：

frameworks/base/libs/ui/Surface.cpp

我们来看看JNI中一些重要的接口：

SurfaceSession\_init：本接口只会被调用一次，负责创建surfacecomposerclient，主要为进程间通信做准备。对应的销毁函数有SurfaceSession\_destroy和SurfaceSession\_kill。

Surface\_init：负责创建surface，最终会调用到surfaceflinger中的createSurface，对应的销毁函数有Surface\_destroy和Surface\_release。

Surface\_lockCanvas：当对一个surface进行绘图之前要调用的，将该surface锁定，并且得到surface的back buffer，应用可以绘图。

Surface\_unlockCanvasAndPost：当上层绘图完毕后，通过该函数通知底层back buffer已绘制完毕，可以

Surface\_setLayer\

Surface\_setPosition\

Surface\_setSize\

Surface\_hide\

Surface\_show\

Surface\_setOrientation\

Surface\_freeze\

Surface\_unfreeze

Surface\_setFlags\

Surface\_setAlpha\

Surface\_setMatrix:设置surface的一些属性，如大小、位置、方位、截取范围

关闭







surface的结构体属性，如下：

uint32\_t what; //哪一项属性改变

int32\_t x; //显示位置

int32\_t y; //显示位置

uint32\_t z; //layer顺序

uint32\_t w; //宽度

uint32\_t h; //高度

float alpha; //透明度

uint32\_t tint; //色彩，未使用

uint8\_t flags; //标志

uint8\_t mask; //屏蔽命令

uint8\_t reserved;

matrix22\_t matrix; //截取范围

Region transparentRegion; //透明度设置

### 3.JNI与Surfaceflinger的连接通讯

由于JNI及C函数实现与surfaceflinger不在同一个进程（一个在应用端 - 客户端

IPC（Binder）方式实现进程间通信，下图来源于网上，不过我修改了里面的

surfaceflinger建立连接以及创建surface的流程。

关闭





防爆摄像头



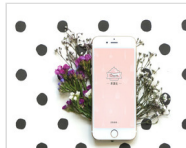
python培训



布袋风管



架构师培训



app开发报价



管道除铁器



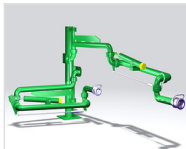
玻璃钢夹砂管



无管道新风系



注浆管



鹤管



JNI和C函数实现我们看作是一个部分

这里看到一个比较重要的部分——SurfaceComposerClient，它是surfaceflinger的客户端，通过它上层才  
surfaceflinger使用Binder联系到一起，IsurfaceComposer和IsurfaceFlingerClient都是用来实现Binder通信  
流程讲解 如下：

应用程序通过JNI接口SurfaceSession\_init创建SurfaceComposerClient。通过SurfaceComposerClient  
调用getComposerService获得IsurfaceComposer的IBinder对象，然后通过  
得IsurfaceFlingerClient的IBinder，通过这个IBinder，JNI就可以调用Surface  
createSurface。由于采用Binder方式，代码部分稍微复杂一些，需要多看几

关闭

#### 4.Surfaceflinger与libui、OpenGL、显示设备的连接

这里不得不提到android对媒体框架中一个很重要的部分，那就是libui，它是一  
如会调用Gralloc、Overlay等HAL层接口。其他的库类继承的方式来调用libui  
接的（包括写显存和对pmem的使用）

Surfaceflinger使用OpenGL来合成surface，所以surfaceflinger会直接调用





防爆摄像头



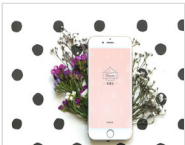
python培训



布袋风管



架构师培训



app开发报价



管道除铁器



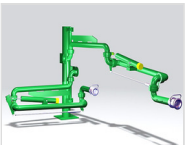
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

它们的架构如下：



这部分的流程比较复杂，主要是各个类的继承绕的比较多，我也是看了很多遍代码以及参考了些资料才理\_\_\_\_，，  
面来详细解释下这个图：

Surfaceflinger在设计时考虑到支持多个屏幕，但目前的版本只支持一个，在s  
一个图中的DisplayHardware，surfaceflinger在初始化时会新建Displayha  
的readyToRun函数），它完成的主要任务一个是建立FramebufferNativeW  
考FramebufferNativeWindow.cpp），再一个就是初始化OpenGL，并创建  
所有的layer最终都将被画到这个main surface上（请参考displayhardware.  
surface、OpenGL和libui中的FramebufferNativeWindow接口就绑定在一起

由于libEGL负责所有layer的最终合成，所以最后数据送往HAL一定要libEGL来

关闭







防爆摄像头



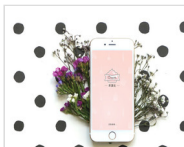
python培训



布袋风管



架构师培训



app开发报价



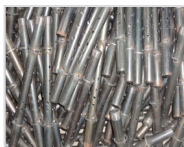
管道除铁器



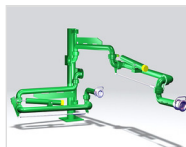
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

postFrameBuffer(surfaceflinger) -> Flip(displayhardware) -> eglSwapBuffers(OpenGL)->  
queueBuffer(libui)->fb\_post(gralloc)

另外图中的GraphicBuffer是libui中提供的对pmem的操作接口，它会直接调用gralloc模块。关于OpenGL和Grall

## Android display架构分析二-SW架构分析

[关闭](#)

**可穿戴技术逆袭帕金森**

英特尔与迈克尔 J. 福克斯帕金森氏症基金会 (MJFF) 携手开发数据分析方案，识别疾病模式并进行归纳，加快实现治疗方案的突破

英特尔、Intel 是英特尔公司在美国和其他国家的商标。\*其他的名称和品牌可能是其他所有者的财产。



防爆摄像头



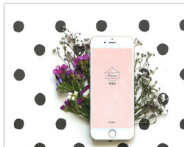
python培训



布袋风管



架构师培训



app开发报价



管道除铁器



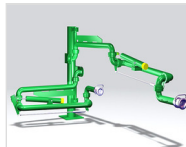
玻璃钢夹砂管



无管道新风系



注浆管



鹤管



关闭



**可穿戴技术逆袭帕金森**

英特尔与迈克尔 J. 福克斯帕金森氏症基金会 (MJFF) 携手开发数据分析方案, 识别疾病模式并进行归纳, 加快实现治疗方案的突破

英特尔、Intel 是英特尔公司在美国和其他国家的商标。\*其他的名称和品牌可能是其他所有者的商标。



防爆摄像头



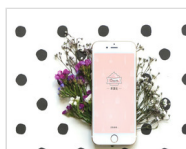
python培训



布袋风管



架构师培训



app开发报价



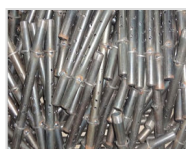
管道除铁器



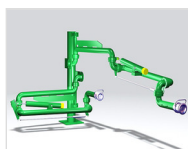
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

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

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

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

User Space Display功能介绍

关闭







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G
- 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

[html]

```
01. struct copybit_context_t *ctx = malloc(sizeof(struct copybit_context_t));
02. memset(ctx, 0, sizeof(*ctx));
03. ctx->device.common.tag = HARDWARE_DEVICE_TAG;
04. ctx->device.common.version = 0;
05. ctx->device.common.module = module;
06. ctx->device.common.close = close_copybit;
07. ctx->device.set_parameter = set_parameter_copybit; //设
08. ctx->device.get = get;
09. ctx->device.blit = blit_copybit; //传送显示数据
10. ctx->device.stretch = stretch_copybit;
11. ctx->mAlpha = MDP_ALPHA_NOP;
12. ctx->mFlags = 0;
13. ctx->mFD = open("/dev/graphics/fb0", O_RDWR, 0); //打开
```

## Android display架构分析三-Kernel Space Display

### 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）

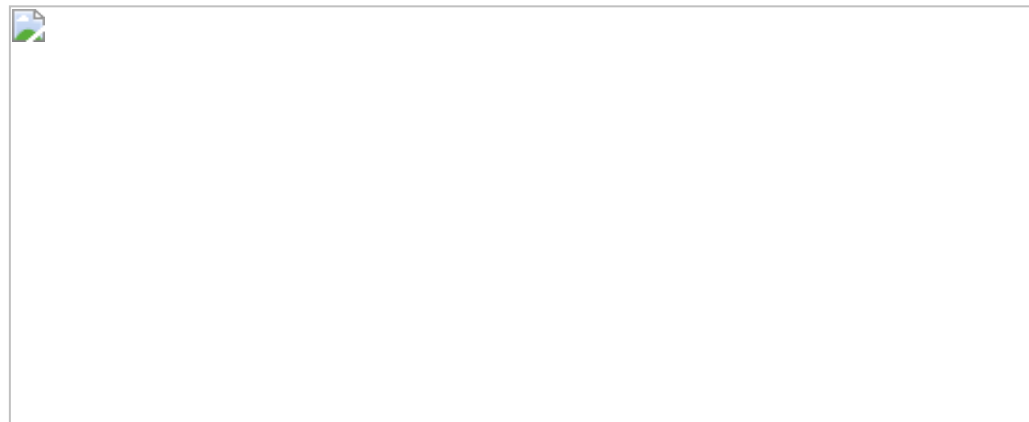
由于Linux工作在保护模式，每个应用程序都有自己的虚拟地址空间，在应用和硬件交互的时候，Linux在文件操作 file\_operations结构中提供了mmap函数，可将文件映射到用户空间。对于帧缓冲设备，则可通过映射操作，可将屏幕缓冲区的物理地址映射到用户空间的一段虚拟地址，读写这段虚拟地址访问屏幕缓冲区，在屏幕上绘图了。实际上，使用帧缓冲设备驱动来驱动显示图形的。由于映射操作都是由内核来完成，下面我们将看到，帧缓冲驱动留

3)、**I/O控制**：对于帧缓冲设备，对设备文件的ioctl操作可读取/设置显示设备参数，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。

关闭





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：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。

自己的一点理解：安卓的display架构是在linux的framebuff下增加了不同的平台显卡驱动（显卡驱动MSM FB），然后基于此开发具体的LCD驱动，目前有三种接口方式，分别是Ic...、mipi、midi，可以查看driver/video/msm目录下的驱动文件，比如Toshiba LCD驱动采用midi，mddi\_toshiba.c文件就是其提供的具体驱动代码

**fb\_mem.c 函数和数据结构介绍**这个文件包含了Linux Fb设备的所有下：

A、Fb设备的文件操作接口 **fb\_fops**

关闭



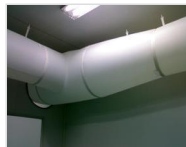




防爆摄像头



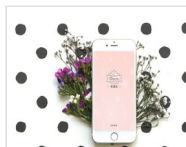
python培训



布袋风管



架构师培训



app开发报价



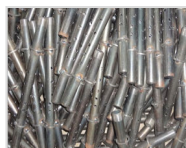
管道除铁器



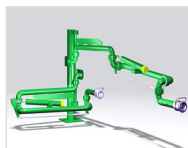
玻璃钢夹砂管



无管道新风系



注浆管



鹤管



## 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xres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，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），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，必须通过内核地址后，应用程序方可访问。

### 3)、fb\_info

关闭





Framebuffer中最重要的结构体，它只能在内核空间内访问。内部定义了fb\_ops结构体（包含一系列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，num\_registered\_fb 为自动加1。

2)、注册framebuffer函数

```
[html]
01. register_framebuffer(struct fb_info *fb_info);
02. 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。几乎所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。

关闭

Android display架构分析四-msm\_fb.c 函数和数据结构介绍

msm\_fb.c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

A、高通msm fb设备的文件操作函数接口 msm\_fb\_ops

```
[html]
01. static struct fb_ops msm_fb_ops = {
```





防爆摄像头



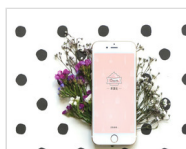
python培训



布袋风管



架构师培训



app开发报价



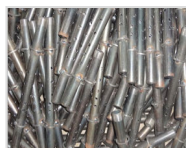
管道除铁器



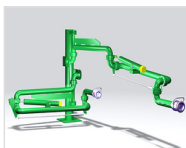
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

```

02. .owner = THIS_MODULE,
03. .fb_open = msm_fb_open,
04. .fb_release = msm_fb_release,
05. .fb_read = NULL,
06. .fb_write = NULL,
07. .fb_cursor = NULL,
08. .fb_check_var = msm_fb_check_var,      /* 参数检查 */
09. .fb_set_par = msm_fb_set_par,          /* 设置显示相关参数 */
10. .fb_setcolreg = NULL, /* set color register */
11. .fb_blank = NULL, /* blank display */
12. .fb_pan_display = msm_fb_pan_display,  /* 显示 */
13. .fb_fillrect = msm_fb_fillrect, /* Draws a rectangle */
14. .fb_copyarea = msm_fb_copyarea, /* Copy data from area to another */
15. .fb_imageblit = msm_fb_imageblit, /* Draws a image to the display */
16. .fb_cursor = NULL,
17. .fb_rotate = NULL,
18. .fb_sync = NULL, /* wait for blit idle, optional */
19. .fb_ioctl = msm_fb_ioctl, /* perform fb specific ioctl (optional) */
20. .fb_mmap = NULL,
21. };

```

## B、高通msm fb的driver接口（驱动接口）msm\_fb\_driver

[html]

```

01. static struct platform_driver msm_fb_driver = {
02.
03. .probe = msm_fb_probe, //驱动探测函数
04. .remove = msm_fb_remove,
05. #ifndef CONFIG_ANDROID_POWER
06. .suspend = msm_fb_suspend,
07. .suspend_late = NULL,
08. .resume_early = NULL,
09. .resume = msm_fb_resume,
10. #endif
11. .shutdown = NULL,
12. .driver = {
13.

```

/\* Driver name must match the device name added

关闭







防爆摄像头



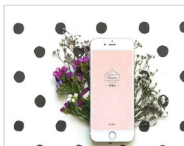
python培训



布袋风管



架构师培训



app开发报价



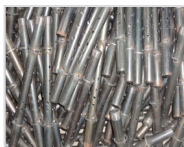
管道除铁器



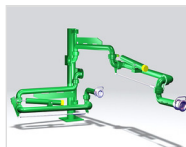
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

```

14.         .name = "msm_fb",
15.     },
16. };

```

### C、msm\_fb\_init ( )

向系统注册msm fb的driver，Msm\_fb.c文件中的初始化时会调用：module\_init(msm\_fb\_init);

### D、msm\_fb\_add\_device ---- probe函数中会被调用：

```

[html]
01. static int __devinit mddi_toshiba_lcd_probe(struct platform_device *pdev)
02. {
03.     .....
03.     msm_fb_add_device(pdev);
04.     return 0;
05. }

```

向系统中添加新的lcd设备，在mddi\_toshiba.c中的probe函数中会被调用：

### mddi\_toshiba.c文件中 函数和数据结构介绍

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

### A、LCD设备相关信息-----platform\_device结构

```

[html]
01. static struct platform_device this_device = {

```

关闭





```
02.     .name    = "mddi_toshiba_vga",
03.     .id      = TOSHIBA_VGA_PRIM,
04.     .dev      = {
05.         .platform_data = &toshiba_panel_data,
06.     }
07. };
```

其中toshiba\_panel\_data包含了硬件LCD的控制函数，如开关、初始化等等，定义如下;

```
[html]
01. static struct msm_fb_panel_data toshiba_panel_data = {
02.
03.     .on          = mddi_toshiba_lcd_on,
04.
05.     .off         = mddi_toshiba_lcd_off,
06.
07. };
```

## B、LCD driver接口(驱动接口)-----platform\_driver结

```
[html]
01. static struct platform_driver this_driver = {
02.     .probe = mddi_toshiba_lcd_probe,
03.     .driver = {
04.         .name    = "mddi_toshiba_vga",
05.     },
06. };
```

其中mddi\_toshiba\_lcd\_probe中会调用msm\_fb\_add\_device接口把具体LCD添

关闭

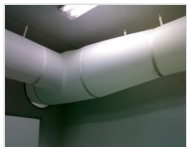




防爆摄像头



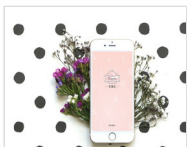
python培训



布袋风管



架构师培训



app开发报价



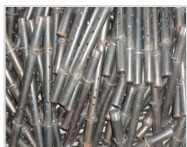
管道除铁器



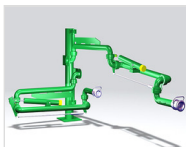
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

## 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。

```
[html]
01. static int __init mddi_toshiba_lcd_init(void)
02. {
03.
04.
05.     return platform_driver_register(&this_driver);
06.
07. }
08.
09.
10.
11. module_init(mddi_toshiba_lcd_init);
```

## D、LCD相关控制函数

toshiba\_common\_initial\_setup ( ) : 初始化MDDI bridge

toshiba\_prim\_start ( ) : 初始化LCD

## Display Kernel数据流分析：

[关闭](#)



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设备是如何调用的**，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。



关闭





防爆摄像头



python培训



布袋风管



架构师培训




app开发报价



管道除铁器



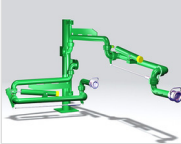
玻璃钢夹砂管



无管道新风系

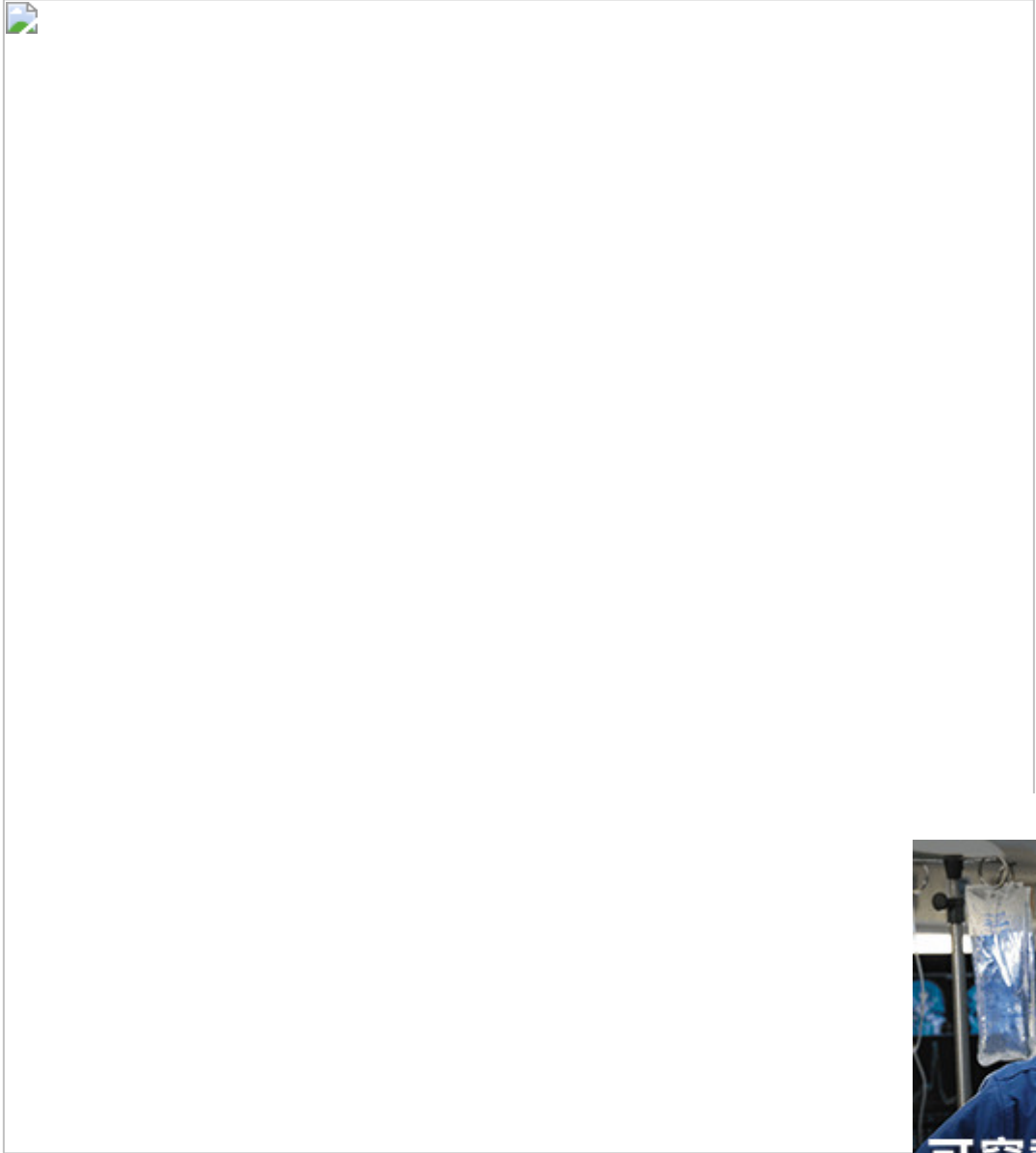


注浆管



鹤管

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）。



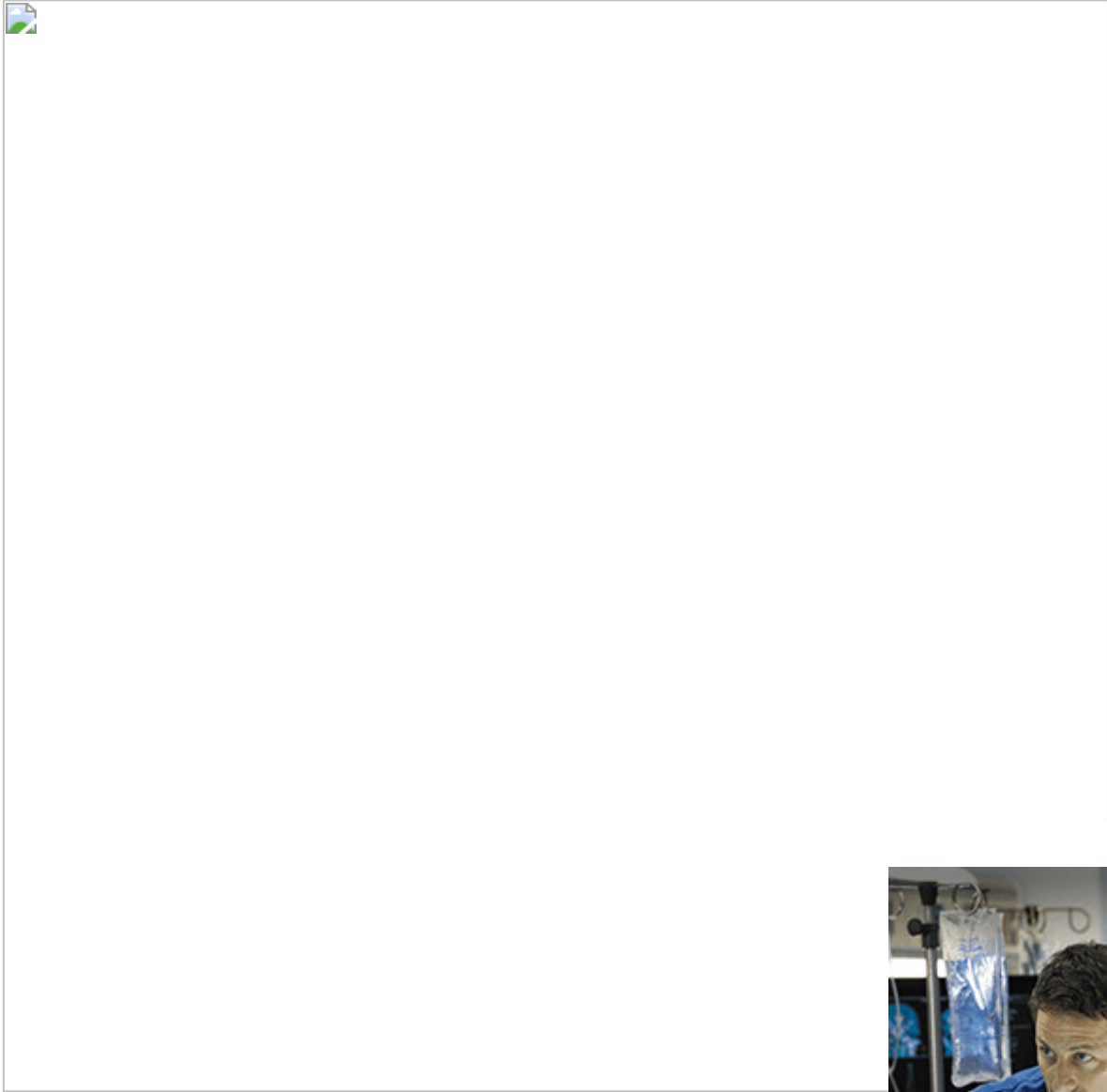
接下来再分析一下Android平台下显示数据是如何处理的，如下图所示



可穿戴技术逆袭帕金森

英特尔与迈克尔 J. 福克斯帕金森氏症基金会 (MJFF) 携手开发数据分析方案，识别疾病模式并进行归纳，加快实现治疗方案的突破

英特尔、Intel 是英特尔公司在美国和其他国家的商标。\*其他的名称和品牌可能是其他所有者的财产。



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、Sc



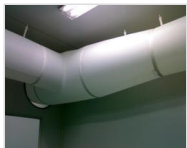




防爆摄像头



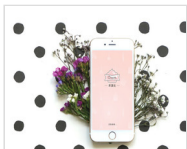
python培训



布袋风管



架构师培训



app开发报价



管道除铁器



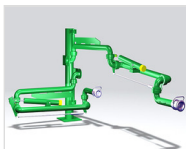
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

## Display Kernel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）。

[cpp]

```
01. static struct platform_driver msm_fb_driver = {
02.     .probe = msm_fb_probe,
03.     .remove = msm_fb_remove,
04.     #ifndef CONFIG_HAS_EARLYSUSPEND
05.     .suspend = msm_fb_suspend,
06.     .resume = msm_fb_resume,
07.     #endif
08.     .shutdown = NULL,
09.     .driver = {
10.         /* Driver name must match the device name added in p
11.         .name = "msm_fb",
12.         .pm = &msm_fb_dev_pm_ops,
13.     },
14. };
```

2)、在LCD模块初始化时会先向系统中注册驱动（在mddi\_toshiba\_lcd\_init

platform\_driver\_register(&this\_driver);名字为mddi\_toshiba\_vga；

关闭





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)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-Display接口介绍

### 1、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.cpp (qcom\diaplay\libcopybit)口如下介绍：

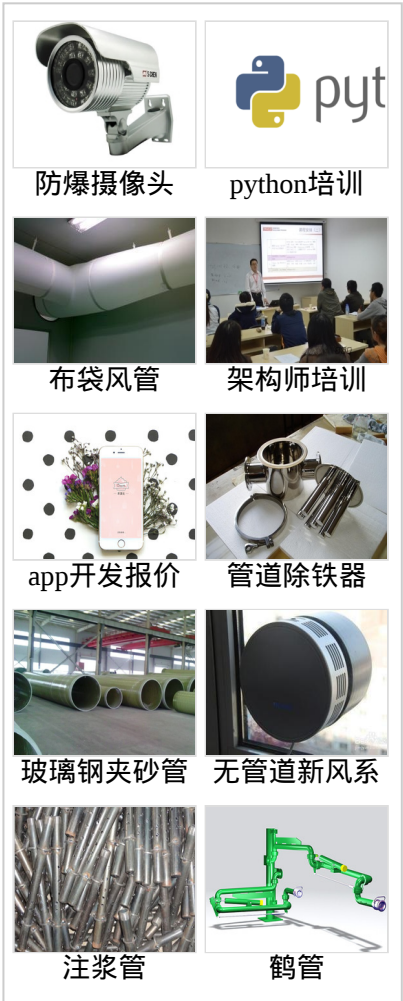
open_copybit	初始化相关变量，并调用open("/dev/graphics/fb0", O_RDWR, 0);打开fb
set_parameter_copybit	设置各种操作参数，如rotate、alpha、dither等。
stretch_copybit	Copy一块数据（Rectangle）到显存，然后并会全mem fb;进行显示
close_copybit	调用close(ctx->mFD);关闭fb设备。

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。

关闭





## 2、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ioctl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：

```
[html]
01. #include <linux/fb.h>
02.
03. int main()
04. {
05.     int fbfd = 0;
06.
07.     struct fb_var_screeninfo vinfo;
08.
09.     struct fb_fix_screeninfo finfo;
10.
```

关闭







防爆摄像头



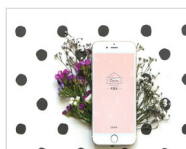
python培训



布袋风管



架构师培训



app开发报价



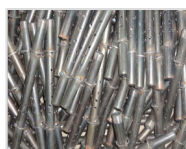
管道除铁器



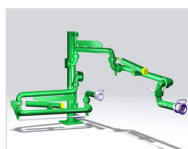
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

```

11. long int screensize = 0;
12.
13. /*打开设备文件*/
14.
15. fbfd = open("/dev/fb0", O_RDWR);
16.
17. /*取得屏幕相关参数*/
18.
19. ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo);
20. ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo);           //该函数最后会将参数传递给finfo、vinfo
21.
22. /*计算屏幕缓冲区大小*/
23.
24. screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;
25.
26. /*映射屏幕缓冲区到用户地址空间*/
27.
28. fbp=(char*)mmap(0, screensize, PROT_READ|PROT_WRITE, MAP_SHARED, fbfd, 0);
29.
30. /*下面可通过fbp指针读写缓冲区*/
31. ...
32. }

```

### 3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围LCD组件。

具体的函数调用流程如下：

关闭





[html]

```
01. copybit_open ( ) ; //打开BlitEngine , 同时也打开fb设备
02.
03. mapFrameBuffer(); //设置显示参数, 同时得到显存虚拟地址
04.
05. copybit->stretch(copybit, &dst, &src, &sdirect, &sdirect, &it); //通知底层去刷屏
```

接下的流程是 :

[html]

```
01. stretch_copybit
02. 调用: status = msm_copybit(ctx, &list);
03.     ->msm_copybit
04.     调
    用int err = ioctl(dev->mFD, MSMFB_BLIT, (struct mdp_blit_req_list const*)list);
05. //此处应该没有调用到: fb_ioctl(), 直接跳到msm_fb_ioctl
06.     ->msm_fb_ioctl(MSMFB_BLIT) //此处将调用msm_fb_ioctl
    msm_fb_ioctl()函数中的switch结构, 跳入MSMFB_BLIT分支调用:
07.     ->case MSMFB_BLIT:
08.
    ret =
09.
    > int ret = mdp_blit(info, &(req_list[i]));
10.
    > mdp_ppp_blit
11.
    -> mdp_start_ppp
12.
    ->MDP&MDDI HW operation
```

关闭

### Android display架构分析六-Surface manager介绍

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, 1、Surface manager ( surface flinger ) 简介



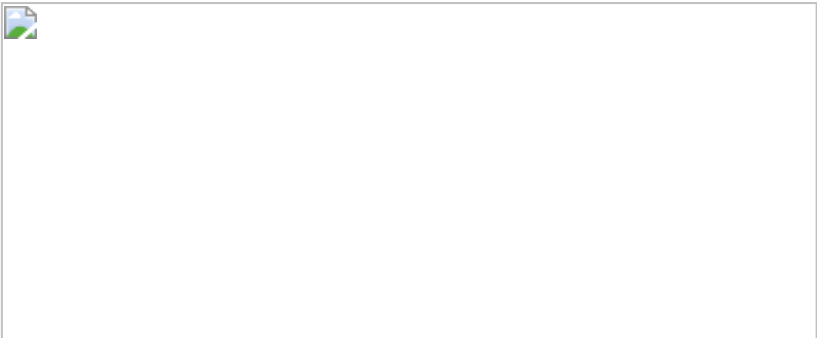


Surface manager是用户空间中framework下libraries中负责显示相关的一个模块。如下：



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：



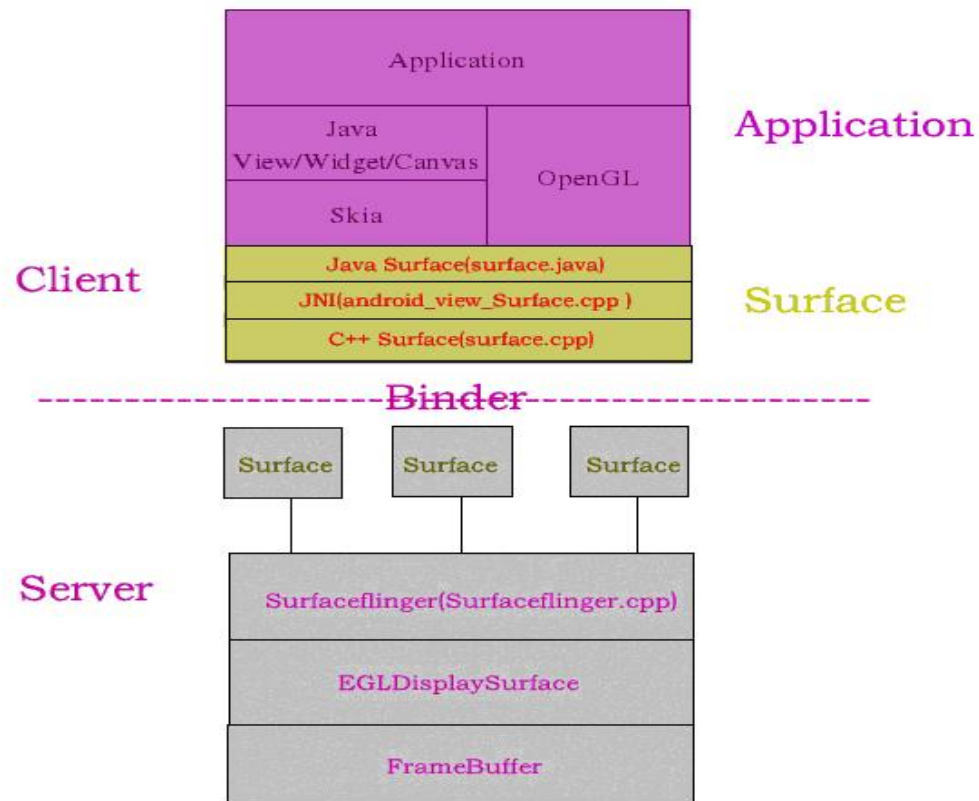
## 2、Surface manager架构分析

Android中的图形系统采用Client/Server架构，如下：

关闭







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

Client和Server之间通过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端的Surfaceflinger，Surfaceflinger负责合成各个surface，然后把buffer传送到个surface对应2个buffer，一个front buffer，一个back buffer，更新时，数据更back buffer和front buffer互换。

关闭



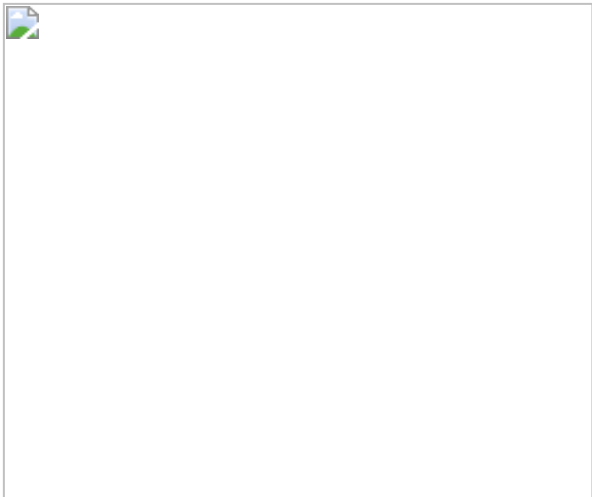
下一部分我们重点研究一下Surfaceflinger。

## Android display架构分析七-Surfaceflinger process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：



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
- 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



防爆摄像头



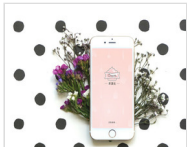
python培训



布袋风管



架构师培训



app开发报价



管道除铁器



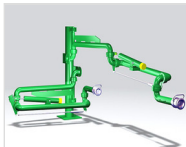
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

关闭





防爆摄像头



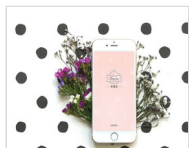
python培训



布袋风管



架构师培训



app开发报价



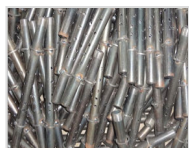
管道除铁器



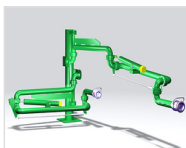
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

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(OpenGL ES 2.0)。

[\[html\] view plaincopyprint?](#)

```
01. status_t SurfaceFlinger::readyToRun()
02.
03. {
04. ...
05. mServerHeap = new MemoryDealer(4096, MemoryDealer::READ_ON
    内存
06.
07. ...
08.
09. mSurfaceHeapManager = new SurfaceHeapManager(this, 8 << 20
    大小为8M，存放具体的显示数据
10.
11. {
12.     // initialize the main display
13.
14.     GraphicPlane& plane(graphicPlane(dpy));
15.
16.     DisplayHardware* const hw = new DisplayHardware(t
17.
```

关闭







```
18.         plane.setDisplayHardware(hw);
19.         //保存显示接口
20.     }
21.
22.     //获取显示相关参数
23.
24.     const GraphicPlane& plane(graphicPlane(dpy));
25.
26.     const DisplayHardware& hw = plane.displayHardware();
27.
28.     const uint32_t w = hw.getWidth();
29.
30.     const uint32_t h = hw.getHeight();
31.
32.     const uint32_t f = hw.getFormat();
33.     ...
34.
35.     // Initialize OpenGL|ES
36.
37.     glActiveTexture(GL_TEXTURE0);
38.
39.     glBindTexture(GL_TEXTURE_2D, 0);
40.
41.     glTexParameterx(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
42.
43.     glTexParameterx(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
44.
45.     glTexParameterx(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
46.
47.     ...
48. }
49.
50. 此部分可以参考文章：SurfaceFlinger启动过程分析 Android系统SurfaceFlinger启动过程分析
```

2)、ThreadLoop

关闭





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

```
[html]
01. bool SurfaceFlinger::threadLoop()
02. {
03.
04.
05.     waitForEvent();           //等待其他接口的signal event
06.     ...
07.
08.     handlePageFlip();         //处理翻页机制
09.
10.     const DisplayHardware& hw(graphicPlane(0).displayHardware());
11.
12.     if (LIKELY(hw.canDraw()))
13.     {
14.
15.
16.         // repaint the framebuffer (if needed)
17.
18.         handleRepaint();       //合并所有layer并填充到buffer中去
19.
20.     ...
21.     postFramebuffer();        //互换front buf
22.     显示
23.     }
24.
25.     ...
26. }
```

在最新的4.1的代码中，threadLoop只调用了waitForEvent，其他部分的实现均是在SurfaceFlinger中实现【SurfaceFlinger.cpp】

谷歌在Android native层实现的一个异步消息机制，在这个机制中几乎不存在任何数据封装到一个消息AMessage结构体中，然后放到队列中去，后台专门有一个





防爆摄像头



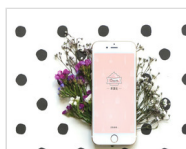
python培训



布袋风管



架构师培训



app开发报价



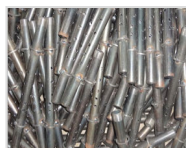
管道除铁器



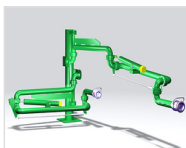
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

行, 执行函数就是onMessageReceived, 这个函数中会有很多分支, 用于处理不同的消息; 在很多类中都会有各种消息post出来, 而后台的异步消息处理线程又是怎么知道发送给哪个类的onMessageReceived函数处理呢, 要搞懂这个问题, 就需要把谷歌实现的这个异步消息处理框架搞明白, 参考文

章: [http://blog.sina.com.cn/s/blog\\_645b74b90101cx69.html](http://blog.sina.com.cn/s/blog_645b74b90101cx69.html)

### 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

[html]

```
01. sp<ISurface> SurfaceFlinger::createSurface(ClientID clientId, int pid,
02.
03.     ISurfaceFlingerClient::surface_data_t* params,
04.
05.     DisplayID d, uint32_t w, uint32_t h, PixelFormat format,
06.
07.     uint32_t flags)
08.
09. {
10. ...
11.     int32_t id = c->generateId(pid);
12.
13.     if (uint32_t(id) >= NUM_LAYERS_MAX) //NUM_LAYERS_MAX=3
14.
15.     {
16.         LOGE("createSurface() failed, generateId = %d", id);
17.         return
18.     }
19. ...
20.     layer = createNormalSurfaceLocked(c, d, id, w, h, format,
    据参数(宽高格式)分配内存(共2个buffer: front/back buffer)
21.
22.     if (layer)
23.     {
24.         setTransactionFlags(eTransactionNeeded);
```

关闭







```
25.
26.     surfaceHandle = layer->getSurface(); //创建surface
27.
28.     if (surfaceHandle != 0)
29.
30.         surfaceHandle->getSurfaceData(params); //创建的surface参数反馈给应用
31. 层
32.     }
33.
34. }
```

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

```
[html]
01. status_t SurfaceFlinger::setClientState(
02.
03.     ClientID cid,
04.
05.     int32_t count,
06.
07.     const layer_state_t* states)
08. {
09.     Mutex::Autolock _l(mStateLock);
10.
11.     uint32_t flags = 0;
12.
13.     cid <= 16;
14.
15.     for (int i=0 ; i<count ; i++)
16.     {
17.         const layer_state_t& s = states[i];
18.
19.         LayerBaseClient* layer = getLayerUser_1(s.surface
20.
```

关闭





```

21.         if (layer)
22.
23.             {
24.
25.                 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
26.
27.                 if (what & eDestroyed) // 删除该层Layer
28.                     {
29.                         if (removeLayer_l(layer) == NO_ERROR)
30.                         {
31.                             flags |= eTransactionNeeded;
32.                             continue;
33.                         }
34.                     }
35.
36.                 if (what & ePositionChanged) // 显示位置变化
37.
38.                     {
39.
40.                         if (layer->setPosition(s.x, s.y))
41.
42.                             flags |= eTraversalNeeded;
43.
44.                     }
45.
46.                 if (what & eLayerChanged)
47.                     {
48.                         if (layer->setLayer(s.z))
49.                         {
50.                             mCurrentState.layersSortedByZ.reorder
51.
52.                                 layer, &Layer::compareCurrents
53.
54.                             flags |= eTransactionNeeded|eTraversal
55.                         }
56.                     }
57.
58.                 if (what & eSizeChanged)

```

关闭





```
59.         {
60.             if (layer-
>setSize(s.w, s.h))
化                                     //设置宽高变
61.                 flags |= eTraversalNeeded;
62.             }
63.
64.             if (what & eAlphaChanged) {
//设置Alpha效果
65.
66.                 if (layer->setAlpha(uint8_t(255.0f*s.alpha+0.5f)))
67.
68.                     flags |= eTraversalNeeded;
69.
70.             }
71.
72.             if (what & eMatrixChanged) { //矩阵参数变化
73.
74.                 if (layer->setMatrix(s.matrix))
75.
76.                     flags |= eTraversalNeeded;
77.
78.             }
79.             if (what & eTransparentRegionChanged) {
显示区域变化
80.
81.                 if (layer->setTransparentRegionHint(s.tran
82.
83.                     flags |= eTraversalNeeded;
84.                 }
85.                 if (what & eVisibilityChanged) { //是否显示
86.
87.                     if (layer->setFlags(s.flags, s.mask))
88.
89.                         flags |= eTraversalNeeded;
90.
91.                 }
92.             }
93.         }
```

关闭







防爆摄像头



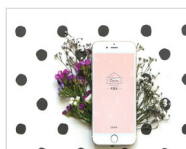
python培训



布袋风管



架构师培训



app开发报价



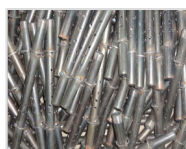
管道除铁器



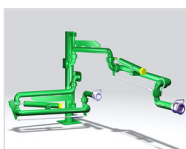
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

```

94.     if (flags)
95.     {
96.         setTransactionFlags(flags);
97.         通过signal通知ThreadLoop
98.     }
99.     return NO_ERROR;
100. }

```

5)、composeSurfaces，在 handleRepaint()中调用到

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，然后让口命令底层显示。

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）

关闭





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-Display 开发的经验分享

### 1添加新的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re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的一些信息登记进去。
- 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。

### 2Display Driver开发过程

#### 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

- 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）。

关闭





### 1.2.2Porting LCD初始化序列

LCD init的代码以及外围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。

### 1.2.3LCD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中的用过。

目前版本中，是根据外围MDDI Bridge中读到的厂商号来决定加载哪个驱动。可以正确读到厂商号，所以bootloader中对于LCD的初始化是有做的，所以屏幕子（花屏）。但Kernel起来后，并没有其他显示，用JTAG跟了后发现，Kernel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MODULE INIT就可正常运行，该问题目前还没有澄清（现在暂时先把bootloa

### 1.2.4LCD的调整

关闭







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为色度信号，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（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Bypass。

YcbCr到RGB的转换是由Android的lib来完成。发了个SR给高通，高通的回复是缺少这个lib（copybit.default.so），6.3.60之后的版本经解决了这个问题。

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描述如下：

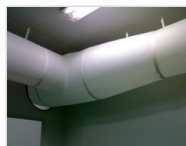




防爆摄像头



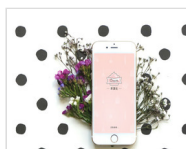
python培训



布袋风管



架构师培训



app开发报价



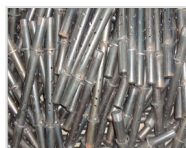
管道除铁器



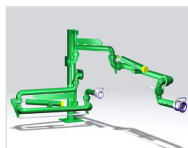
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

## 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的应用程序）而言，只要到surfaceflinger中创建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，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。

## 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

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framebu

关闭





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的接口

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。现在可以看出工厂就创建了disp\_loop thread，里面的操作就是调用系统接口ioctl(m->framebuffer->fd, FBIOPUT\_VSCREENINFO, &m->info)来设置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)来获知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和HW两种方案。SW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。其路径是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，处理后的数据通过DMA送到framebuffer。其路径是android/frameworks/base/opengl/libs（有几个版本）

关闭







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      第一行代表该codebase支持SW 方案的OpenGL，是android default的
- 0 1 adreno200    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

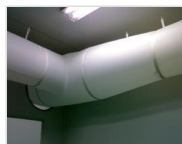




防爆摄像头



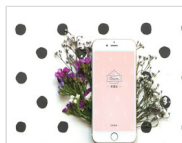
python培训



布袋风管



架构师培训



app开发报价



管道除铁器



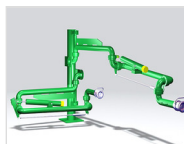
玻璃钢夹砂管



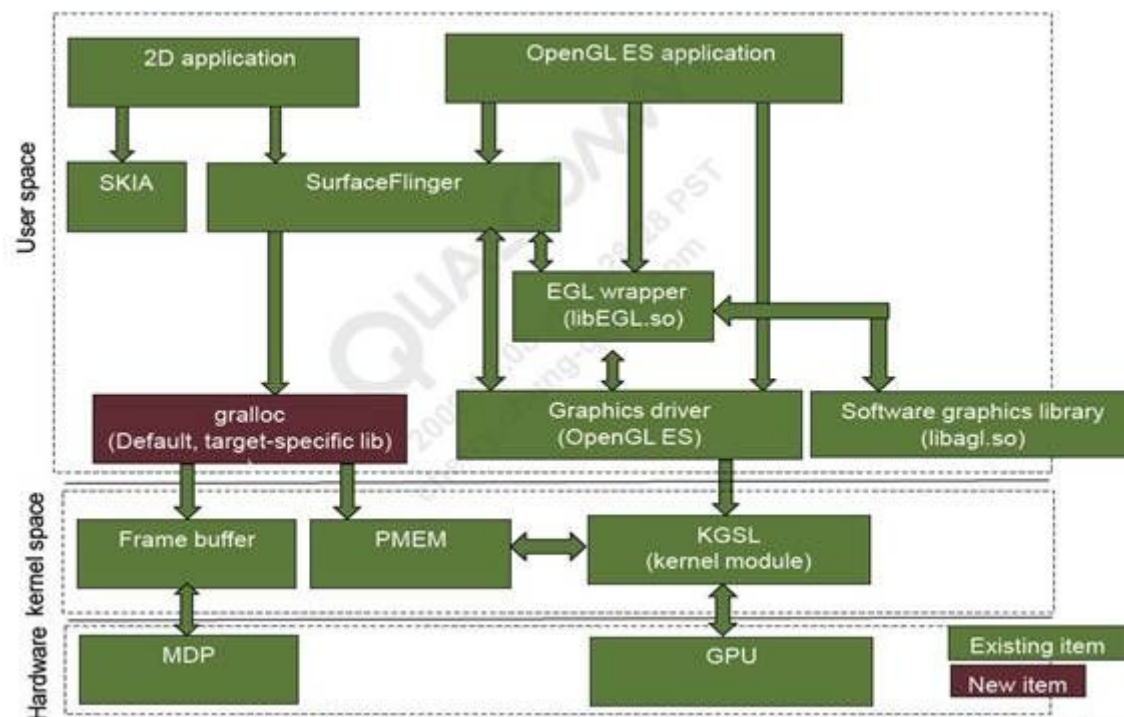
无管道新风系



注浆管



鹤管



关闭

顶 踩

1

0

上一篇 多重继承及虚继承中对象内存的分布

下一篇 802.11 Client Active, Passive Scanning, BGScan, On-roam sc





防爆摄像头



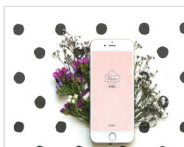
python培训



布袋风管



架构师培训



app开发报价



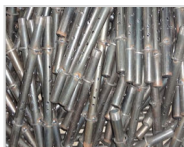
管道除铁器



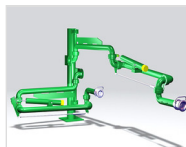
玻璃钢夹砂管



无管道新风系



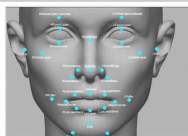
注浆管



鹤管

## 相关文章推荐

- Android display架构分析-SW架构分析(1-4)
- Android Display架构分析--侧重高通平台
- Android display架构分析（七-1）
- Android系统Surface机制的SurfaceFlinger服务渲染..
- Android display架构分析
- Android图形合成和显示系统---基于高通MSM8k M...
- Android游戏开发(一)
- Android display架构分析-SW架构分析(1-4) .
- [转]Android 游戏框架（一个游戏角色在屏幕行走的...
- Android display架构分析六-Surface manager介



人脸识别



北大青鸟



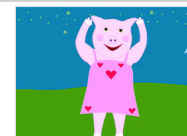
人工智能机器



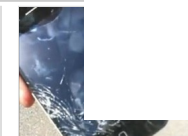
便宜的好手机



摄像机教程



在线英语学习



苹果拆

## 猜你在找

机器学习之概率与统计推断

机器学习之凸优化

响应式布局全新探索

深度学习基础与TensorFlow实践

前端开发在线峰会

机器学习之数学基础

机器学习之矩阵

探究Linux的总线、设备

深度学习之神经网络原理

TensorFlow实战进阶：手

## 查看评论

暂无评论

## 发表评论

用户名： haijuncz

关闭







防爆摄像头



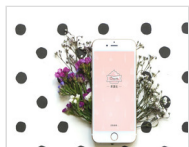
python培训



布袋风管



架构师培训



app开发报价



管道除铁器



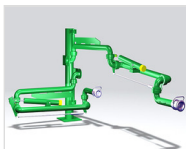
玻璃钢夹砂管



无管道新风系



注浆管



鹤管

评论内容：



Empty text area for comment content.

提交

\* 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

[联系方式](#) | [版权声明](#) | [法律顾问](#) | [问题报告](#) | [合作伙伴](#) | [论坛反馈](#)[webmaster@csdn.net](mailto:webmaster@csdn.net)

400-660-0108

| 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 | 江苏乐知网络技术有限公司

17, CSDN.NET, All Rights Reserved



关闭

**可穿戴技术逆袭帕金森**

英特尔与迈克尔·J. 福克斯帕金森氏症基金会 (MJFF) 携手开发数据分析方案，识别疾病模式并进行归纳，加快实现治疗方案的突破

英特尔、Intel 是英特尔公司在美国和其他国家的商标。\* 其他的名称和品牌可能是其他所有者的商标。