内容目录

[一 显示系统框架 1](#__RefHeading___Toc248_2054555503)

[a. 显示驱动framebuffer(fb)的原理及改进 1](#__RefHeading___Toc250_2054555503)

[b. 多任务系统的显示: 必定有一个显示管理者 1](#__RefHeading___Toc252_2054555503)

[二 修改源码禁用hwc和GPU 2](#__RefHeading___Toc254_2054555503)

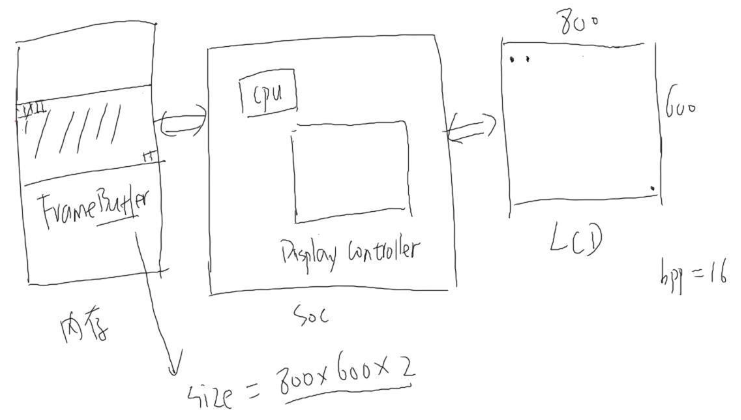
[2.1 tiny4412 2](#__RefHeading___Toc256_2054555503)

[2.2 qcom 4](#__RefHeading___Toc258_2054555503)

# 一 显示系统框架

显示系统001\_框架.jpg

## a. 显示驱动framebuffer(fb)的原理及改进

只有1个fb的缺点

(1) 如果APP写fb速度慢，LCD图像变化慢

(2) 如果APP写fb速度不快不慢，闪烁

(3) APP写fb速度快--OK

改进：使用多个FB，循环 while(1) {

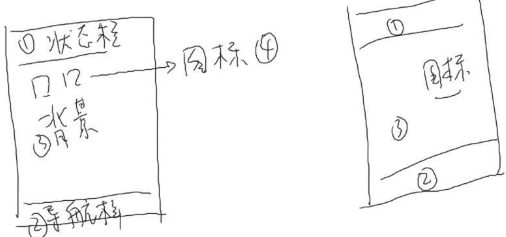
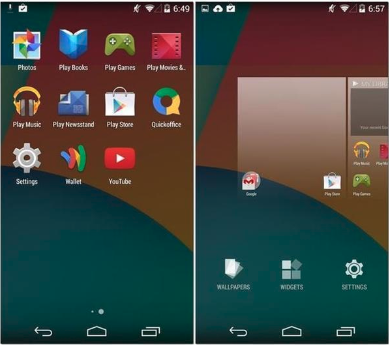
(1) Display Controller使用FB0

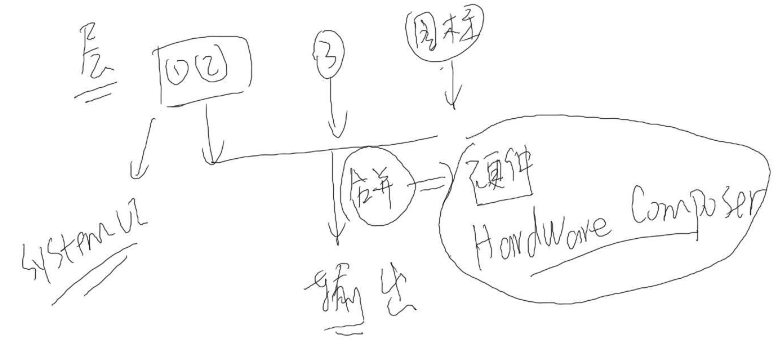
(2) APP写fb1

(3) Display Controller使用FB1

(4) APP写fb0

}

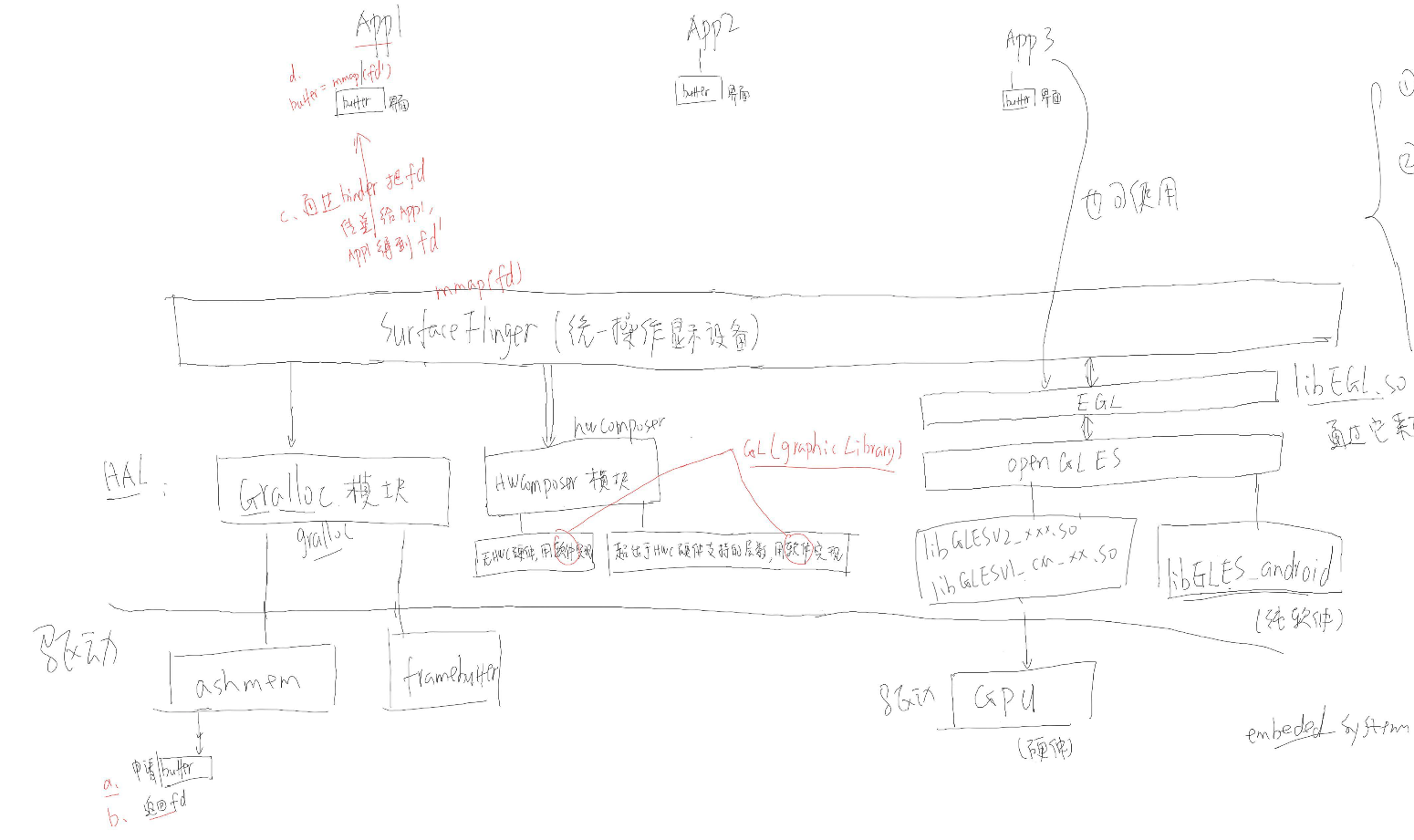


 重复工作1、2、3，其中1和2是systemUI进程，3是进程2，图标是进程3，如果芯片支持合成HardwareComposer那么可以提升性能。

驱动支持HWC:

每一层对应一个驱动/dev/fbx，APP操作某层，直接写对应的framebuffer，硬件自动合成他们。

## b. 多任务系统的显示: 必定有一个显示管理者

 跟高通文档的架构很相近，只不过高通底层是SDM，这边是通用的framebuffer

sufraceflinger:

(1)给APP提供buffer

a.通过gralloc模块向ashmem申请内存

b.得到一个fd

c.通过binder把fd传给某个APP，APP得到fd’

d.APP再mmap(fd’)

(2)APP1、2、3把各自界面发给它，它根据层次、大小进行合成、显示

a.根据各个界面的Z值决定前后顺序，由WindowManagerService确定

b.把这些排序后的buffer传给HWComposer

(3)当HWC不能处理(无HWC硬件/超出HWC层数)buffer时，使用GL(GraphicLibrary)来处理

libEGL硬件GL库、软件GL库

# 二 修改源码禁用hwc和GPU

厂家一般不会提供硬件合成器和GPU源代码，无法分析

## 2.1 tiny4412

git clone https://github.com/weidongshan/SYS\_0003\_Patch\_Disable\_HWC\_GPU\_tiny4412.git

git checkout v1

android-5.0.2\_no\_hwc\_no\_gpu.patch

这个补丁做了3件事:

**a. 去掉厂家提供的gralloc, hwcompser HAL模块**

hardware/libhardware/hardware.c

#if defined(\_\_LP64\_\_)

#define HAL\_LIBRARY\_PATH1 "/system/lib64/hw"

#define HAL\_LIBRARY\_PATH2 "/vendor/lib64/hw"

#define HAL\_LIBRARY\_PATH3 "/odm/lib64/hw"

#else

#define HAL\_LIBRARY\_PATH1 "/system/lib/hw"

#define HAL\_LIBRARY\_PATH2 "/vendor/lib/hw"

#define HAL\_LIBRARY\_PATH3 "/odm/lib/hw"

#endif

static const char \*variant\_keys[] = {

"ro.hardware", /\* This goes first so that it can pick up a different

file on the emulator. \*/

"ro.product.board",

"ro.board.platform",

"ro.arch"

};

hw\_get\_module\_by\_class //查找，load so库

文件名gralloc.属性值.so: gralloc.tiny4412.so, gralloc.exynos4.so，最后 gralloc.default.so

**a.1 删除单板上/system/lib/hw**

gralloc.tiny4412.so

hwcomposer.exynos4.so

adb reboot之后，黑屏一片

查看错误信息

logcat \*:E

hwcomposer module not found

invalid buffer handle given //这个错误视频中查找不到源代码，是厂家另外一个库文件，把相关的都可以干掉/system/lib/egl/，不干掉也没事，因为后续要修改属性

**a.2 修改源码使得编译结果中不含上述文件 (修改vendor/friendly-arm/tiny4412/device-tiny4412.mk, 参考补丁)**

-

**b. 添加属性让android系统认为自己运行于"没有GPU的模拟器"**

修改libagl/Android.mk，给系统添加libGLES\_android.so (软件实现的GL)

**b.1 添加软件GPU库**

开发板: su, mount -o remount /system

在服务器编译软件GPU库: mmm frameworks/native/opengl/libagl

把得到的libGLES\_android.so复制到单板/system/lib/eg/，并添加读属性

adb reboot之后

couldn’t find an OpenGL ES implementation

搜索之后，错误文件

frameworks/native/opengl/libs/EGL/Loader.cpp

找到加载libGLES\_android.so的代码，发现需要修改属性值

**b.2 修改frameworks/native/opengl/libagl/Android.mk (参考补丁)**

-

**b.3 修改属性文件 , 单板 adb shell 进去修改 /system/build.prop，添加:**

ro.kernel.qemu=1

ro.kernel.qemu.gles=0

或修改源码 device/friendly-arm/tiny4412/system.prop 同样添加上述属性, 然后重新编译系统

adb reboot之后

no suitable EGLConfig found, giving up

**c. 修改系统自带的gralloc模块的BUG**

**c.1 查找错误文件**

frameworks/native/services/surfaceflinger/RenderEngine/RenderEngine.cpp

EGLConfig RenderEngine::chooseEglConfig(EGLDisplay display, int format, bool logConfig) {

尝试获得ES2的配置，如果失败尝试获得ES1的配置(硬线相关的)，否则使用简单查询方式来获得配置

}

经过debug，怀疑是配置问题

hardware/libhardware/modules/gralloc/Framebuffer.cpp

HAL\_PIXEL\_FORMAT\_BGRA\_8888

改为：

HAL\_PIXEL\_FORMAT\_RGBA\_8888

mmm hardware/libhardware/modules/gralloc/

把gralloc.default.so复制到单板/system/lib/hw

adb reboot之后，出错:

E/BufferQueueProducer( 2320): [FramebufferSurface] dequeueBuffer: createGraphicBuffer failed

W/GraphicBufferAllocator( 1918): alloc(800, 480, 1, 00001a33, ...) failed -12 (Out of memory)

**c.2 继续修改 hardware\libhardware\modules\gralloc\Framebuffer.cpp**

frameworks/native/libs/ui/GraphicBufferAllocator.cpp

const std::unique\_ptr<const Gralloc2::Allocator> mAllocator;

status\_t GraphicBufferAllocator::allocate(uint32\_t width, uint32\_t height,

PixelFormat format, uint32\_t layerCount, uint64\_t usage,

buffer\_handle\_t\* handle, uint32\_t\* stride,

uint64\_t /\*graphicBufferId\*/, std::string requestorName)

{

。。。

Gralloc2::Error error = mAllocator->allocate(info, stride, handle);//打开Gralloc模块涉及的结构体

if (error == Gralloc2::Error::NONE) {

Mutex::Autolock \_l(sLock);

KeyedVector<buffer\_handle\_t, alloc\_rec\_t>& list(sAllocList);

uint32\_t bpp = bytesPerPixel(format);

alloc\_rec\_t rec;

rec.width = width;

rec.height = height;

rec.stride = \*stride;

rec.format = format;

rec.layerCount = layerCount;

rec.usage = usage;

rec.size = static\_cast<size\_t>(height \* (\*stride) \* bpp);

rec.requestorName = std::move(requestorName);

list.add(\*handle, rec);

return NO\_ERROR;

} else {

ALOGE("Failed to allocate (%u x %u) layerCount %u format %d "

"usage %" PRIx64 ": %d",

width, height, layerCount, format, usage,

error);

return NO\_MEMORY;

}

}

经过code flow追查

hardware/libhardware/modules/gralloc/gralloc.cpp

dev->device.alloc = gralloc\_alloc;

static int gralloc\_alloc(alloc\_device\_t\* dev,

int width, int height, int format, int usage,

buffer\_handle\_t\* pHandle, int\* pStride)

{

。。。

int err;

if (usage & GRALLOC\_USAGE\_HW\_FB) {

err = gralloc\_alloc\_framebuffer(dev, size, usage, pHandle);//从framebuffer里面分配内存，错误信息usage=1a33

if (bufferMask >= ((1LU<<numBuffers)-1)) {

// We ran out of buffers.

return -ENOMEM;//-12

}

} else {

err = gralloc\_alloc\_buffer(dev, size, usage, pHandle);//从ashmem里面分配内存

}

if (err < 0) {

return err;

}

\*pStride = stride;

return 0;

}

hardware/libhardware/modules/gralloc/framebuffer.cpp

+#if 0//不需要再申请framebuffer

/\*

\* Request NUM\_BUFFERS screens (at lest 2 for page flipping)

\*/

info.yres\_virtual = info.yres \* NUM\_BUFFERS;

- uint32\_t flags = PAGE\_FLIP;

#if USE\_PAN\_DISPLAY

if (ioctl(fd, FBIOPAN\_DISPLAY, &info) == -1) {

ALOGW("FBIOPAN\_DISPLAY failed, page flipping not supported");

@@ -195,6 +206,16 @@

info.yres\_virtual = info.yres;

flags &= ~PAGE\_FLIP;

}

+#endif//直接获取即可

+

+ if (ioctl(fd, FBIOGET\_VSCREENINFO, &info) == -1)

+ return -errno;

+ info.yres\_virtual = info.yres\_virtual;//虚拟分辨率，比如800×600，虚拟y可以分配3个800，而yres为800

+ if (info.yres\_virtual > info.yres)

+ flags |= PAGE\_FLIP;

+ else

+ flags &= ~PAGE\_FLIP;

+

hardware/libhardware/modules/gralloc/framebuffer.cpp

+#if 0

if (ioctl(m->framebuffer->fd, FBIOPUT\_VSCREENINFO, &m->info) == -1) {

ALOGE("FBIOPUT\_VSCREENINFO failed");

m->base.unlock(&m->base, buffer);

return -errno;

}

+#else

+ if (ioctl(m->framebuffer->fd, FBIOPAN\_DISPLAY, &m->info) == -1) {//通过FBIOPAN\_DISPLAY 来确定使用那个buffer

+ ALOGE("FBIOPAN\_DISPLAY failed");

+ m->base.unlock(&m->base, buffer);

+ return -errno;

+ }

+#endif

+

**重启之后成功!**

## **2.2 qcom**

**gralloc**

源码位置

hardware/qcom/display/gralloc

LOCAL\_MODULE := gralloc.$(TARGET\_BOARD\_PLATFORM)

库位置

sm6150\_au:/vendor/lib64/hw # ls gralloc.\*.so

gralloc.default.so gralloc.sm6150.so

sm6150\_au:/ # getprop ro.hardware

qcom

sm6150\_au:/ # getprop ro.product.board

sm6150

sm6150\_au:/ # getprop ro.board.platform

sm6150

sm6150\_au:/ # getprop ro.arch

属性值

ro.kernel.qemu

qemu.gles

# 二 最简单的Surface测试程序

git clone https://github.com/weidongshan/APP\_0010\_SurfaceTest.git

Display/APP\_0010\_SurfaceTest

参考demo

frameworks/native/services/surfaceflinger/tests/resize

错误fix

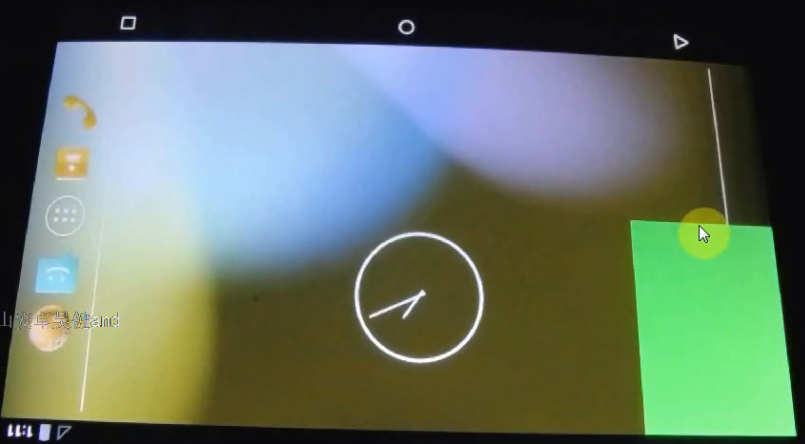
<http://www.aichengxu.com/android/8815305.htm>

取出指定版本:

git checkout v1 // v1, correct the bugs from frameworks/native/services/surfaceflinger/tests/resize

git checkout v2 // v2, display B,G,R color

git checkout v3 // v3, print the buffer address



v1版本编译后多了一个绿色的长方块

int main(int argc, char\*\* argv)

{

// set up the thread-pool

sp<ProcessState> proc(ProcessState::self());

ProcessState::self()->startThreadPool();

// create a client to surfaceflinger

sp<SurfaceComposerClient> client = new SurfaceComposerClient();//获得surface服务

sp<SurfaceControl> surfaceControl = client->createSurface(String8("resize"),//创建surface

160, 240, PIXEL\_FORMAT\_RGB\_565, 0);

sp<Surface> surface = surfaceControl->getSurface();//获得surface

SurfaceComposerClient::openGlobalTransaction();

surfaceControl->setLayer(100000);//设置Z轴，为了覆盖用，dumpsys SurfaceFlinger可以查看到z轴的大小

SurfaceComposerClient::closeGlobalTransaction();

ANativeWindow\_Buffer outBuffer;

surface->lock(&outBuffer, NULL);//获得surface的一个buffer

ssize\_t bpr = outBuffer.stride \* bytesPerPixel(outBuffer.format);

android\_memset16((uint16\_t\*)outBuffer.bits, 0xF800, bpr\*outBuffer.height);//填充buffer，0xF800一种颜色

surface->unlockAndPost();//把buffer提交给surfaceflinger让它显示出来

+ sleep(3);//加入休眠，为了看出变化(v2版本主要改动，也加入了其他颜色)

surface->lock(&outBuffer, NULL);//获得surface的另一个buffer

android\_memset16((uint16\_t\*)outBuffer.bits, 0x07E0, bpr\*outBuffer.height);//填充buffer，另外一种颜色

surface->unlockAndPost();//再次提交

+ sleep(3);

SurfaceComposerClient::openGlobalTransaction();

surfaceControl->setSize(320, 240);

SurfaceComposerClient::closeGlobalTransaction();

+ for (int i = 0; i < 100; i++) {//v3

+ surface->lock(&outBuffer, NULL);

+ printf("%03d buff addr = 0x%x\n", i, (unsigned int)outBuffer.bits);

+ surface->unlockAndPost();

+ }

输出结果，应用程序对于一个surface，分配了三个buffer

000 buff addr = 0x4003e000

001 buff addr = 0x40083000

002 buff addr = 0x403be000

003 buff addr = 0x4003e000

004 buff addr = 0x40083000

005 buff addr = 0x403be000

...

IPCThreadState::self()->joinThreadPool();

return 0;

}