交接文档

## 一、Wifi

### 1.1 wifi死机

wifi硬件使用一段时间经常会出现”AR6000: hifEnableFunc(), Unable to enable AR6K”主要原因在于硬件出现问题，具体问题可能是硬件老化。出现这种现象基本会导致设备无法重启，进而导致设备无法开机。hifEnableFunc这个函数主要是使用wifi芯片工作，与wifi通信是通过sdio总线，并且sdio总线配置为高速（48M），因此可能wifi硬件还没初始化完，我们即刻发送使能命令，就有可能wifi硬件无法响应。针对此问题我们采用如下做法：

while(count > 0){

count --;

ret = sdio\_enable\_func(func);

if (ret) {

AR\_DEBUG\_PRINTF(ATH\_DEBUG\_ERROR, ("AR6000: %s(), Unable to enable AR6K: 0x%X\n",\_\_FUNCTION\_\_, ret));

printk(KERN\_ERR"===========dealing sdio\_enable\_func fail============\n");

A\_MDELAY(1000);

}

else{

printk(KERN\_ERR"===========success============\n");

break;

}

}

通过上面的代码就可以看出，我们采用尝试3次并且失败一次就休眠1s，wifi硬件就有足够的时间初始化，如果还出现问题，那几本判定硬件需要更换。此处有点需要注意的地方就是不能进来就休眠，这样会影响系统启动的时间，原因在于大部分时间以及新设备一般都能及时初始化完。

做到此步还不够，我们应该还要想一个万全之策，那就是出现异常情况要及时通知应用以及wifi使用成功也需要告诉应用。为什么成功也需要告诉应用？主要应用经常搞一些其他的操作，比如禁止wifi连接之类的操作，就会导致应用没有up eth0（应用检测wifi是否正常启动，仅仅通过eth0是否存在来判定），还有一个就是应用可以采取重启设备，因此我们非常有必要把wifi运行起来的状态通知应用。我们可以通过netlink与应用通信，代码如下：

// 2016-01-28@zhouwei add netlink funtion

struct {

\_\_u32 pid;

}user\_process;

static char current\_status[MSG\_LEN];

static struct sock \*netlinkfd = NULL;

static int send\_to\_user(char \*info) //send message to user space

{

int size;

struct sk\_buff \*skb;

unsigned char \*old\_tail;

struct nlmsghdr \*nlh; //message head

int retval;

size = NLMSG\_SPACE(strlen(info)); //message size

skb = alloc\_skb(size, GFP\_ATOMIC); // fifo buff

//init for netlink 's message head

nlh = nlmsg\_put(skb, 0, 0, 0, NLMSG\_SPACE(strlen(info))-sizeof(struct nlmsghdr), 0);

old\_tail = skb->tail;

memcpy(NLMSG\_DATA(nlh), info, strlen(info));

nlh->nlmsg\_len = skb->tail - old\_tail; //set message len

//set control field

NETLINK\_CB(skb).pid = 0;

NETLINK\_CB(skb).dst\_group = 0;

printk(KERN\_ERR "[kernel space] skb->data:%s\n", (char \*)NLMSG\_DATA((struct nlmsghdr \*)skb->data));

//send message

printk(KERN\_ERR "[kernel space] netlinkfd = %p user\_process.pid = %d\n", netlinkfd,user\_process.pid);

retval = netlink\_unicast(netlinkfd, skb, user\_process.pid, MSG\_DONTWAIT);

printk(KERN\_ERR "[kernel space] netlink\_unicast return: %d,message = %s,info = %s\n", retval,(char\*)NLMSG\_DATA(nlh),info);

return 0;

}

static void kernel\_receive(struct sk\_buff \*\_\_skb) //kernel receive message from user space

{

struct sk\_buff \*skb;

struct nlmsghdr \*nlh = NULL;

// char \*data = "This is test message from kernel";

printk(KERN\_ERR "[kernel space] begin kernel\_receive\n");

skb = skb\_get(\_\_skb);

if(skb->len >= sizeof(struct nlmsghdr)){

nlh = (struct nlmsghdr \*)skb->data;

if((nlh->nlmsg\_len >= sizeof(struct nlmsghdr))

&& (\_\_skb->len >= nlh->nlmsg\_len)){

user\_process.pid = nlh->nlmsg\_pid;

printk(KERN\_ERR "[kernel space] data receive from user are:%s\n", (char \*)NLMSG\_DATA(nlh));

printk(KERN\_ERR "[kernel space] user\_pid:%d\n", user\_process.pid);

send\_to\_user(current\_status);

}

}else{

printk(KERN\_ERR "[kernel space] data receive from user are:%s\n",(char \*)NLMSG\_DATA(nlmsg\_hdr(\_\_skb)));

send\_to\_user(current\_status);

}

kfree\_skb(skb);

}

//---!>

ret = hifEnableFunc(device, func);

// 2016-01-28@zhouwei add netlink funtion

if(ret == A\_OK ){

message = WIFI\_CONNECT;//WMI\_CONNECT\_EVENTID;

}

else if(ret == A\_PENDING){

message = WIFI\_READY;//WMI\_READY\_EVENTID;

}

else{

message = WIFI\_ERROR;//WMI\_ERROR\_REPORT\_EVENTID;

}

printk(KERN\_ERR">>>>>>>>>>%s,%d,message = %d<<<<<<<<<<\n",\_\_func\_\_,\_\_LINE\_\_,message);

memset(info,0x00,sizeof(info));

sprintf(info,"%d", message);

printk(KERN\_ERR"wifi error,errror\_num = %d\n", message);

memset(current\_status,0x00,sizeof(current\_status));

if(strlen(info) > MSG\_LEN )

{

lens = MSG\_LEN;

}

else

{

lens = strlen(info);

}

memcpy(current\_status, info, lens);

send\_to\_user( info);

注：目前只发现了使用wifi硬件启动会出问题，后需要发现wifi崩溃的信息时我们通过使用send\_to\_user接口来通知应用wifi的情况。

### wifi time out

如果发现sdio与wifi通信时出现cmd53 发送超时基本上是我们设置的时钟设置太大或者太小，我们可以通过board-9615-storage.c文件来修改，代码如下：

#ifdef CONFIG\_MMC\_MSM\_SDC2\_SUPPORT

static unsigned int sdc2\_sup\_clk\_rates[] = {

400000, 24000000, 48000000 /**/此处就是修改sdio的时钟，开始启动使用400K，后面慢慢切换到48M**

};

static struct mmc\_platform\_data sdc2\_data = {

.ocr\_mask = MMC\_VDD\_27\_28 | MMC\_VDD\_28\_29,

.mmc\_bus\_width = MMC\_CAP\_4\_BIT\_DATA,

.sup\_clk\_table = sdc2\_sup\_clk\_rates,

.sup\_clk\_cnt = ARRAY\_SIZE(sdc2\_sup\_clk\_rates),

.pclk\_src\_dfab = 1,

.pin\_data = &mmc\_slot\_pin\_data[SDCC2],

.sdiowakeup\_irq = MSM\_GPIO\_TO\_INT(GPIO\_SDC2\_DAT1\_WAKEUP),

.msm\_bus\_voting\_data = &sps\_to\_ddr\_bus\_voting\_data,

};

static struct mmc\_platform\_data \*msm9615\_sdc2\_pdata = &sdc2\_data;

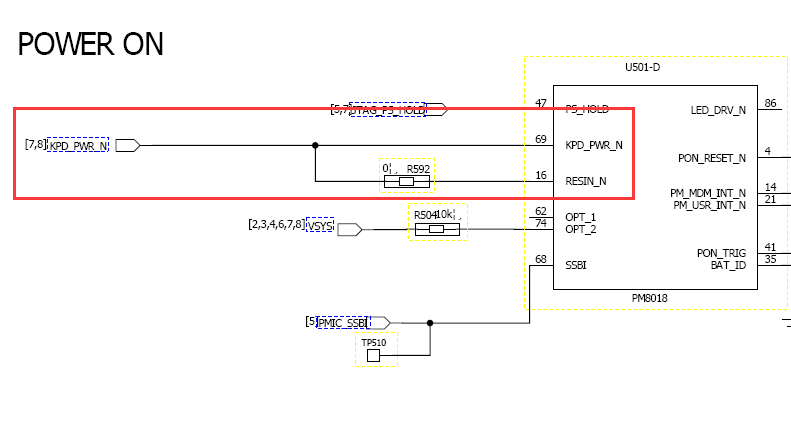
#else

static struct mmc\_platform\_data \*msm9615\_sdc2\_pdata;

#endif

## power reset按键

power 和 reset按键硬件上面是连接在一起，原理图如下：



Pmic中reset按键中有一个定时器，如果按下power按键时，reset按键就开始计时power一直处于按下状态并且超过13s，就会触发硬件reset，否则都是触发power按键事件。具体pmic的reset原理图可以查看硬件相关的文档。

### Power 按键异常

首先按键中断处理函数分为按下和松开两个中断，中断处理函数如下：

static irqreturn\_t pwrkey\_press\_irq(int irq, void \*\_pwrkey)

{

struct pmic8xxx\_pwrkey \*pwrkey = \_pwrkey;

input\_report\_key(pwrkey->pwr, KEY\_POWER, 1);

input\_sync(pwrkey->pwr);

mdm9615\_report\_event(EV\_KEY, KEY\_POWER, 1);

return IRQ\_HANDLED;

}

static irqreturn\_t pwrkey\_release\_irq(int irq, void \*\_pwrkey)

{

struct pmic8xxx\_pwrkey \*pwrkey = \_pwrkey;

input\_report\_key(pwrkey->pwr, KEY\_POWER, 0);

input\_sync(pwrkey->pwr);

mdm9615\_report\_event(EV\_KEY, KEY\_POWER, 0);

return IRQ\_HANDLED;

}

连续按下power按键时应用正常收到的按键时序为：1 0 1 0或者1 0 1。但是应用经常会接收到异常的按键时序，比如1 0 1 0 0 1 1或者1 0 1 1 0 0，为什么会出现此种情况呢？可能因为power与reset接在一起，reset按键影响导致。采取的措施：当应用收到1 0 1 0 0 1 1（应用认为是长按power按键）时序我们需要告诉应用当前是否是长按power按键，如果读取到power按键当前状态是按下，则关机，否则认为这段时序为异常，代码如下：

static ssize\_t power\_key\_show\_rotate(struct device \*dev, struct device\_attribute \*attr, char \*buf)

{

return sprintf(buf, "%d\n", pm8018\_read\_irq\_rt\_stat(PM8018\_IRQ\_BASE + PM8018\_PWRKEY\_PRESS\_IRQ));

}

static ssize\_t power\_key\_store\_rotate(struct device \*dev, struct device\_attribute \*attr, const char \*buf, size\_t count)

{

return 0;

}

static DEVICE\_ATTR(power\_key\_crtl, S\_IRUGO | S\_IWUSR,

power\_key\_show\_rotate,

power\_key\_store\_rotate);

static struct attribute \*power\_key\_attrs[] = {

&dev\_attr\_power\_key\_crtl.attr,

NULL,

};

static struct attribute\_group power\_key\_attr\_group = {

.attrs = power\_key\_attrs,

};

static struct kobject \*power\_key\_kobj;

static int \_\_init pmic8xxx\_pwrkey\_init(void)

{

int ret = 0;

**power\_key\_kobj = kobject\_create\_and\_add("power\_key\_status", kernel\_kobj);**

**if(power\_key\_kobj != NULL){**

**ret = sysfs\_create\_group(power\_key\_kobj, &power\_key\_attr\_group);**

**if (ret) {**

**kobject\_put(power\_key\_kobj);**

**pr\_err("power\_key\_status create attr error: %d\n",ret);**

**}**

**}**

return platform\_driver\_register(&pmic8xxx\_pwrkey\_driver);

}

应用就可以通过cat /sys/kernel/power\_key\_status 来获取power按键的真实状态

## fastboot reboot关机

这个问题主要是给生产带来比较大的麻烦。Fastboot reboot主要牵涉到lk（bootloader）以及开机流程。在fastboot模式下面时我们输入fastboot reboot，就会调用如下函数：

#define REBOOT\_MODE 0x77665503 //add reboot function by zhouwei

void cmd\_reboot(const char \*arg, void \*data, unsigned sz)

{

dprintf(INFO, "rebooting the device\n");

fastboot\_okay("");

reboot\_device(REBOOT\_MODE);//add reboot function by zhouwei

}

reboot\_device(REBOOT\_MODE);这个函数主要是为了往内存地址0x2B000000+0x65C（这个是真实的物理地址）里面写入 0x77665503 。因此输入fastboot reboot以后就会重启设备，此处可能会问fastboot reboot不是能重启机器了吗？确实是fastboot reboot会重启机器，但是到了kernel也会去读0xFA00A000 +0x65C（这个地址是虚拟地址，经过映射后的地址，与真实0x2B000000+0x65C地址对应），如果读取到值不是0x77665503，kernel就会认为是短按power按键，因此就会关闭设备，进而导致fastboot reboot关机，因此fastboot reboot以后就必须传入0x77665503值才能使设备启动，而不是关机。Kernel的代码如下：

msm\_restart\_reason = readl(MSM\_IMEM\_BASE + 0x65C);

writel(0x00, MSM\_IMEM\_BASE + 0x65C);

pr\_err("msm\_restart\_reason=%d\n", msm\_restart\_reason);

//--!>

/\* BugID:1227 Start zouwubin 2015/07/22 combine reset with power key \*/

msm\_power\_reason = readl(MSM\_IMEM\_BASE + 0x66C);

writel(0x00, MSM\_IMEM\_BASE + 0x66C);

printk("%s, %d, 0x%x\n", \_\_func\_\_, \_\_LINE\_\_, msm\_power\_reason);

/\* BugID:1227 End zouwubin 2015/07/22 combine reset with power key \*/

msm9615\_common\_init();

/\* BugID:1227 Start zouwubin 2015/07/07 config pm8018 hardware reset function \*/

is\_hw\_reset = pm8xxx\_get\_reset\_reason();

pr\_err("is\_hw\_reset=%d\n", is\_hw\_reset);

/\* BugID:1227 End zouwubin 2015/07/07 config pm8018 hardware reset function \*/

//<!--Add @2013-02-26 mingjun.liu

if ((0x77665503 == msm\_restart\_reason)||(0x77665502 == msm\_restart\_reason)) {//sys\_reboot,reboot ,sys\_reboot recovery

/\* BugID:1227 Start zouwubin 2015/07/22 combine reset with power key \*/

}

**else if (msm\_power\_reason == 0x77665503) {**

board\_boot\_mode = 1;

printk(KERN\_ERR"=============CIT status = %d==================\n",gpio\_get\_value(CIT\_GPIO));

if(0 == gpio\_get\_value(CIT\_GPIO)){

board\_boot\_mode = 2;

}

printk(KERN\_ERR"boot\_mode = %d,%d,%s\n", board\_boot\_mode, \_\_LINE\_\_, \_\_func\_\_);

/\* BugID:1227 End zouwubin 2015/07/22 combine reset with power key \*/

} else {//key/usb insert

//<!--Add @2013-02-19 mingjun.liu

rt\_stat = pm8018\_read\_irq\_rt\_stat(PM8018\_IRQ\_BASE + PM8018\_PWRKEY\_PRESS\_IRQ);

if (rt\_stat < 0) {

printk(KERN\_INFO"Get pm8018 power key state error!");

}

else {

if( rt\_stat == 0) {

printk(KERN\_INFO"Must keep power key on!");

/\* BugID:1227 Start zouwubin 2015/07/07 config pm8018 hardware reset function \*/

if (RESTART\_BY\_HW\_KEY == is\_hw\_reset) {

printk(KERN\_INFO"RESTART\_BY\_HW\_KEY == is\_hw\_reset!");

}

else if (RESTART\_BY\_KPD == is\_hw\_reset)

{

printk(KERN\_INFO"(RESTART\_BY\_KPD == is\_hw\_reset)!");

//pm\_power\_off();

}

/\* BugID:1227 End zouwubin 2015/07/07 config pm8018 hardware reset function \*/

}

}

board\_boot\_mode = 3;

printk("boot\_mode = %d,%d,%s\n", board\_boot\_mode, \_\_LINE\_\_, \_\_func\_\_);

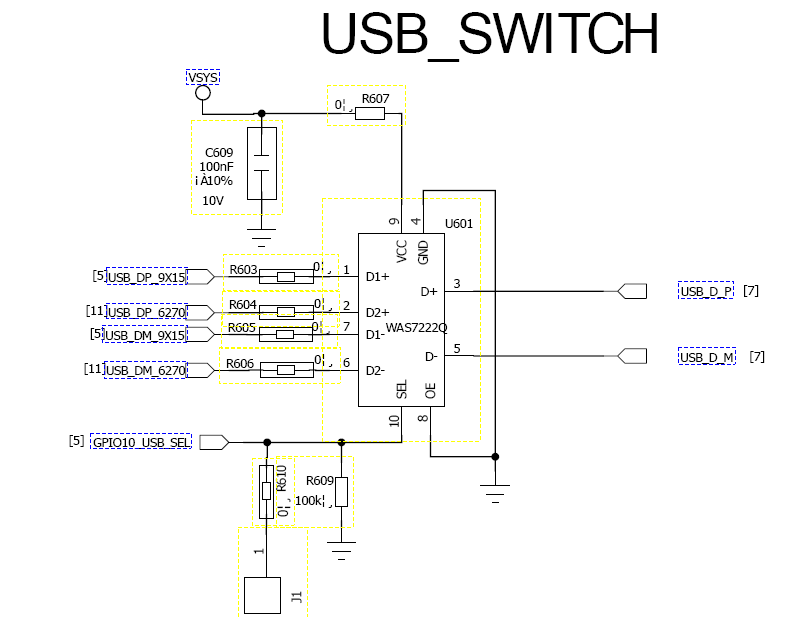
}

## 开机时间

正常来说对于9x15开机是非常快的，快到什么程度呢？只要触碰到power按键，设备立马开机，但是这种对于用户来说肯定不能接受，因此在kernel里面我们需要检测power按键是否按下，如果还是处于按下则认为可以开机，这样做就可以解决触碰到power按键会开机的问题，但是这个问题解决了又会引发其他的问题。我们先了解一下9x15的开机流程，9x15开机先启动不是lk（bootloader），而是modem，因此从按下power按键到启动kernel大概需要6-8s的时间，这样就会造成用户必须持续按下power按键长达6-8s的时间，才能通过kernel的认证，少1s都不行开启设备。因此就会带来了用户感觉开机非常的慢的问题，因此我们需要与modem（sbl1）取得联系，因为modem（sbl1）先启动，只要用户触碰到power按键时，大概3s运行到sbl1立刻运行，所以我们就在sbl1里面检测到按键按下，我们就可以向0x2B000000+0x65C（这个是真实的物理地址）写入0x77665503，kernel检测到这块内存值为0x77665503时就可以开启设备，最终就可以完美解决两大问题：1、触碰到power就开机的问题 2、开机时间长达8s的问题

## 控制6720

这块主要是通过任务是硬件来实现，我们主要通过gpio来控制use switch，硬件原理图如下：

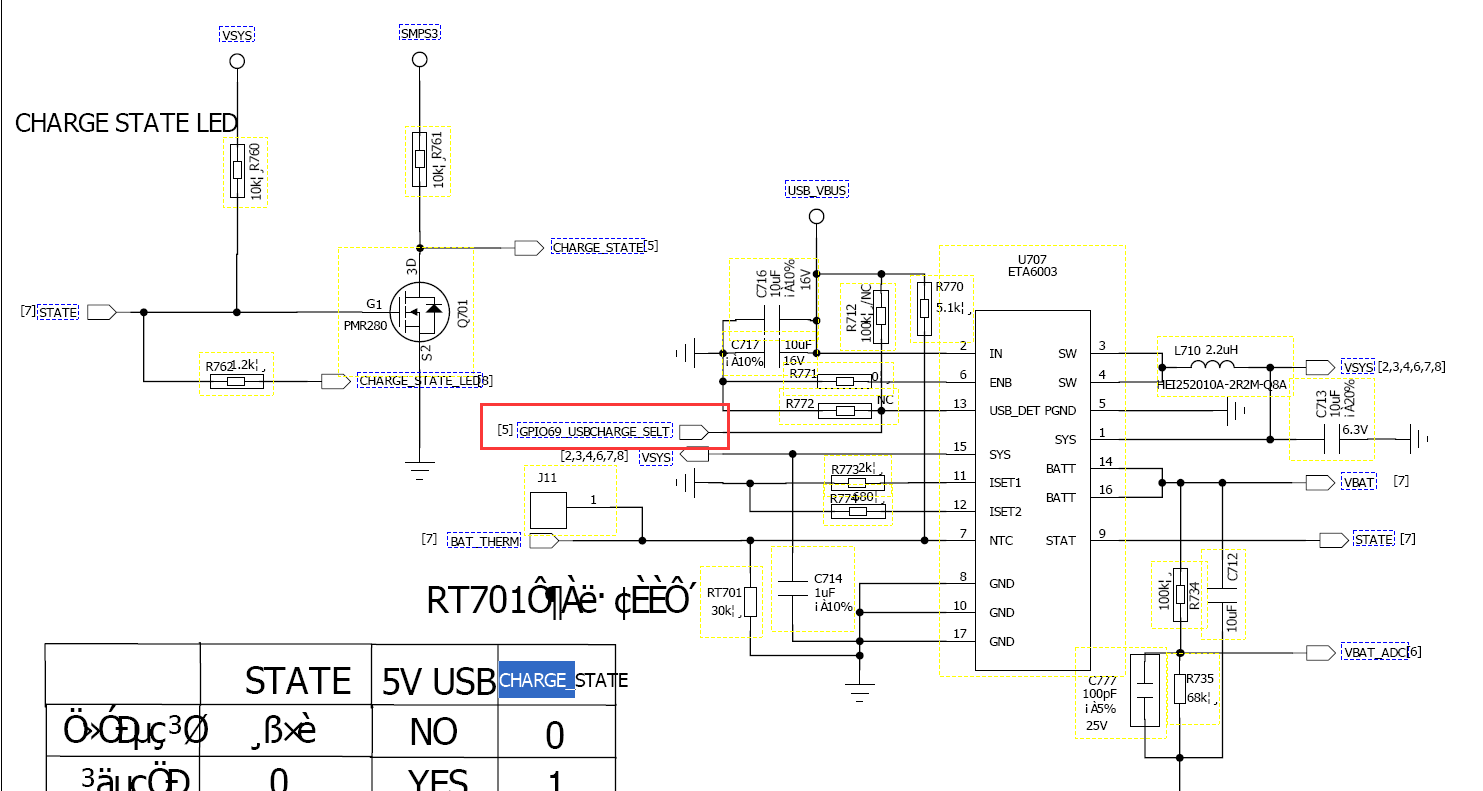


## Uart

Uart主要是配置gpio的引脚属性。代码路径：

## Charge

Charge这块全部交给硬件管理，驱动只负责把状态传递给上层，硬件原理图如下：



从原理图我们就能知道charge的状态通过GPIO69来传递，因此kernel这边需要配置gpio69为中断，GPIO69一旦发生变化，我们立刻通知应用，进而通知用户电池的状态，但是我们还有一点是需要usb的状态，目的是为了告诉用户电池充放电的状态。关于判断电池状态的代码如下：

static int eta6003\_get\_chg\_status(struct eta6003\_chg \*dev)

{

if ((dev->is\_5V\_usb == 1) && (dev->is\_charge\_irq == 1))

{

dev->charge\_stat = POWER\_SUPPLY\_STATUS\_CHARGING;

}

else if ((dev->is\_5V\_usb == 1) && (dev->is\_charge\_irq == 0))

{

dev->charge\_stat = POWER\_SUPPLY\_STATUS\_FULL;

}

else

{

dev->charge\_stat = POWER\_SUPPLY\_STATUS\_NOT\_CHARGING;

}

return dev->charge\_stat;

}

## LED

LED灯主要是控制GPIO，我们就可以达到控制LED灯的亮灭，这部分全部交给应用来处理。

## 休眠唤醒

休眠流程主要是应用发起的，进而kernel收到通知并且执行相应的suspend来使相应的设备进入休眠状态，但是进入休眠时一旦有中断产生，系统立刻就会被唤醒，因此经常遇到无法休眠的问题，就是因为某个中断在进入休眠时没进行关闭中断，在进入休眠时时常产生中断。遇到此问题时要找到具体的中断未关闭谈何容易，因此我们需要研究9x15中断流程，我们发现绝大部中断都会通过gic，此时我们就会想到来监测gic的变化不就ok了吗？对了，我们就是通过gic来找到中断源，进而关闭该中断，gic监测代码如下：

void msm\_gic\_get\_wakeup\_interrupt(void)

{

unsigned int i;

u32 enabled;

unsigned long pending[32];

struct gic\_chip\_data \*gic = &gic\_data[0];

void \_\_iomem \*base = gic\_data\_dist\_base(gic);

for (i = 0; i \* 32 < gic->max\_irq; i++) {

enabled = readl\_relaxed(base + GIC\_DIST\_ENABLE\_CLEAR + i \* 4);

pending[i] = readl\_relaxed(base + GIC\_DIST\_PENDING\_SET + i \* 4);

pending[i] &= enabled;

}

msm\_gic\_wakeup\_interrupt = find\_first\_bit(pending, gic->max\_irq);

}

通过此函数我们就可以清晰看到那个中断中断号，但是实践中却发现并不是我们想象中一下子就能确定中断源，实践中发现只要GPIO中断中就是48，只要是pmic中断就是53，GPIO中断和pmic中断那可就多了，还是无法确定具体中断源。此刻我们也没有过多的办法，只能大致确认那部分中断源，无法确定具体的中断源，因此我们采用比较笨重的办法就是检测所有中断，这个我们可以通过应用来观察，如果唤醒了立刻查看那个中断发现了变化，应用层使用的命令：cat /proc/interrupt，我们就可以有意识地查看某部分中断次数的变化，进而确定具体的中断源。

## ADC

Adc主要是配置，配置代码如下：

static struct pm8xxx\_adc\_amux pm8018\_adc\_channels\_data[] = {

{"vcoin", CHANNEL\_VCOIN, CHAN\_PATH\_SCALING2, AMUX\_RSV1,

ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_DEFAULT},

{"vbat", CHANNEL\_VBAT, CHAN\_PATH\_SCALING2, AMUX\_RSV1,

ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_DEFAULT},

{"vph\_pwr", CHANNEL\_VPH\_PWR, CHAN\_PATH\_SCALING2, AMUX\_RSV1,

ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_DEFAULT},

/\* AMUX8 is used to read either Batt\_id/Batt\_therm.

\* Current configuration is to support Batt\_id. If clients

\* want to read the Batt\_therm, the scaling function needs to be

\* updated to use ADC\_SCALE\_BATT\_THERM instead of ADC\_SCALE\_DEFAULT.

\* E.g.

\* {"batt\_therm", CHANNEL\_BATT\_ID\_THERM, CHAN\_PATH\_SCALING1,

\* AMUX\_RSV2, ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_BATT\_THERM},

\*/

{"batt\_id", CHANNEL\_BATT\_ID\_THERM, CHAN\_PATH\_SCALING1,

AMUX\_RSV2, ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_DEFAULT},

{"pmic\_therm", CHANNEL\_DIE\_TEMP, CHAN\_PATH\_SCALING1, AMUX\_RSV1,

ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_PMIC\_THERM},

{"625mv", CHANNEL\_625MV, CHAN\_PATH\_SCALING1, AMUX\_RSV1,

ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_DEFAULT},

{"125v", CHANNEL\_125V, CHAN\_PATH\_SCALING1, AMUX\_RSV1,

ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_DEFAULT},

{"pa\_therm0", ADC\_MPP\_1\_AMUX3, CHAN\_PATH\_SCALING1, AMUX\_RSV1,

ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_PA\_THERM},

{"xo\_therm", CHANNEL\_MUXOFF, CHAN\_PATH\_SCALING1, AMUX\_RSV0,

ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_XOTHERM},

//<!--Add @2013-03-25 mingjun.liu

{"mpp6", ADC\_MPP\_1\_AMUX5, CHAN\_PATH\_SCALING1, AMUX\_RSV1,

ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_DEFAULT},

{"mpp1", ADC\_MPP\_1\_ATEST\_8, CHAN\_PATH\_SCALING1, AMUX\_RSV1,

ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_DEFAULT},

{"mpp2", ADC\_MPP\_1\_USB\_SNS\_DIV20, CHAN\_PATH\_SCALING1, AMUX\_RSV1,

ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_DEFAULT},

{"mpp3", ADC\_MPP\_1\_DCIN\_SNS\_DIV20, CHAN\_PATH\_SCALING1, AMUX\_RSV1,

ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_DEFAULT},

#if defined(CONFIG\_SLT673) || defined(CONFIG\_SLT675)

{"mpp5", ADC\_MPP\_1\_AMUX4, CHAN\_PATH\_SCALING1, AMUX\_RSV1,

ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_MAINBOARD\_A02},

#else

{"mpp5", ADC\_MPP\_1\_AMUX4, CHAN\_PATH\_SCALING1, AMUX\_RSV1,

ADC\_DECIMATION\_TYPE2, ADC\_SCALE\_BATT\_THERM\_ECTH\_3435K},

#endif

//--!>

};

## OLED驱动

Oled驱动相对比较复杂，使用到的文件/drivers/video/msm/msm\_fb.c、/drivers/video/msm/ebi2\_host.c、/drivers/video/msm/ebi2\_lcd.c、/drivers/video/msm/ebi2\_oled.c(可能为ebi2\_oled1284.c或者ebi2\_oled1281.c等)、board-9615-display.c。

**msm\_fb.c：**这个文件主要关注两个点，1、msm\_fb\_add\_device 函数中mfd->fb\_imgType = MDP\_FB\_GRAY，目的是为了第2步中msm\_fb\_register，2、msm\_fb\_register函数中结合自己配置的mfd->fb\_imgType参数，加入自己配置oled的属性，例如：

case MDP\_FB\_GRAY: /\*register MDP\_GRAY para\*/

fix->type = FB\_TYPE\_PACKED\_PIXELS;

fix->xpanstep = 0;

fix->ypanstep = 0;

var->vmode = FB\_VMODE\_NONINTERLACED;

var->transp.offset = 0;

var->transp.length = 0;

**bpp = 1;主要是这个参数，其他的参数不确定都使用默认即可，bpp指一点需要几个点来描述，bpp=1即表示使用一位即可表示一点的状态，黑白**

break;

**ebi2\_host.c：**这个函数主要关注是mdp\_dma\_pan\_update函数，这个函数类似DMA，需要应用层发出display指令，才会被执行。刷数据到屏幕上时需要设置好怎么刷数据到屏幕上，比如：

n\_width = fbi->fix.line\_length;

n\_height = iBuf->dma\_h/8;

for (i = 0; i < n\_height; i++) {

//如下黑体，设置屏幕刷新的区域问题

**pdata->output(1,0xB0+i);**

**pdata->output(1,0x00);**

**pdata->output(1,0x10);**

for (j = 0; j < n\_width; j++) {

data = \*src;

//<!--Modify @2012-12-13 mingjun.liu

//outp8(mfd->data\_port, data);

pdata->output(0,(u16)data);

//--!>

src++;

}

}

注：如果屏幕没有特别要求只需要执行pdata->refresh(src);即可

**ebi2\_lcd.c:** 这个文件主要是lcd控制器，一般不会去动

**ebi2\_oled.c：** 主要关注一点就是ebi2\_oled\_cmd\_init函数还有一个就是开关机图片数组（g\_image\_pwr\_off和g\_image\_pwr\_on，数组的大小由屏的像素决定），ebi2\_oled\_cmd\_init函数主要是初始化屏幕的参数。

**board-9615-display.c** ： 主要关注两个点：1、MSM\_FB\_PRIM\_BUF\_SIZE 即显存的大小，2、屏幕的电源问题，通过ebi2\_panel\_power函数配置，核心就是regulator\_get（&msm\_ebi2\_lcdc\_device.dev, **"VDDI2"**);）通过名字获取电源，像676的电源是LD05，这个时候我就得填写LD05