Wacklock问题以及suspend流程浅析

拟制人：\_\_\_\_\_\_\_\_\_徐勇 \_\_\_\_\_\_\_\_

审核： \_\_\_\_\_\_\_\_ 赵少鹏\_\_\_\_\_\_\_\_

批准： \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

发布日期：\_\_\_\_\_\_2014.11.20\_\_\_\_\_\_\_

目录

[一、背景 3](#_Toc404383693)

[二、wakelock 3](#_Toc404383694)

[三．supend 7](#_Toc404383695)

[四．问题分析与解决 12](#_Toc404383696)

[五．总结 13](#_Toc404383697)

wakelock问题以及suspend流程浅析

# 一、背景

纽曼S455项目中遇到待机底电流偏大，按power键灭屏后，主板底电流一直跳变，机器无法进入休眠状态。抓取log分析，判断为EINT wakelock被申请，导致系统无法睡眠。下面将通过对suspend代码跟踪，简单介绍它与wakelock之间的联系，并描述wakelock休眠被申请问题的分析方法。

# 二、wakelock

Wakelock是一种锁机制，当它被激活时，能阻止系统进入挂起状态以及其他低功耗模式，使系统无法进行正常的suspend流程。

上层对wake\_lock和wake\_unlock操作实际是对如下连个节点的操作

"/sys/power/wake\_lock"

"/sys/power/wake\_unlock"

在alps\kernel\kernel\power\main.c中

static int \_\_init pm\_init**(**void**)**

**{**

int error **=** pm\_start\_workqueue**();**

**if** **(**error**)**

**return** error**;**

hibernate\_image\_size\_init**();**

hibernate\_reserved\_size\_init**();**

power\_kobj **=** kobject\_create\_and\_add**(**"power"**,** **NULL);**

**if** **(!**power\_kobj**)**

**return** **-**ENOMEM**;**

error **=** sysfs\_create\_group**(**power\_kobj**,** **&**attr\_group**);**

**if** **(**error**)**

**return** error**;**

**return** pm\_autosleep\_init**();**

**}**

函数sysfs\_create\_group(power\_kobj, &attr\_group)会在sys/下面创建节点，具体**&**attr\_group创建如下

static struct attribute **\*** g**[]** **=** **{**

**&**state\_attr**.**attr**,**

#ifdef CONFIG\_PM\_TRACE

**&**pm\_trace\_attr**.**attr**,**

**&**pm\_trace\_dev\_match\_attr**.**attr**,**

#endif

#ifdef CONFIG\_PM\_SLEEP

**&**pm\_async\_attr**.**attr**,**

**&**wakeup\_count\_attr**.**attr**,**

#ifdef CONFIG\_PM\_AUTOSLEEP

**&**autosleep\_attr**.**attr**,**

#endif

#ifdef CONFIG\_PM\_WAKELOCKS

**&**wake\_lock\_attr**.**attr**,**

**&**wake\_unlock\_attr**.**attr**,**

#endif

#ifdef CONFIG\_PM\_DEBUG

**&**pm\_test\_attr**.**attr**,**

#endif

#endif

**NULL,**

**};**

static struct attribute\_group attr\_group **=** **{**

**.**attrs **=** g**,**

**};**

在上面的结构体中定义了一系列所需节点，&wake\_lock\_attr和&wake\_unlock\_attr就在其中。

参考alps\kernel\kernel\power\power.h中的定义

#define power\_attr(\_name) \

static struct kobj\_attribute \_name##\_attr = { \

.attr = { \

.name = \_\_stringify(\_name), \

.mode = 0644, \

}, \

.show = \_name##\_show, \

.store = \_name##\_store, \

}

对wake\_lock和wake\_unlock节点的操作就是调用wake\_lock\_store函数和wake\_unlock\_store函数。在wake\_lock\_store中执行代码，最后到wakeup\_source\_report\_event，对wakeup事件进行上报计数。

alps\kernel\drivers\base\power\wakeup.c

static void wakeup\_source\_report\_event**(**struct wakeup\_source **\***ws**)**

**{**

ws**->**event\_count**++;**

/\* This is racy, but the counter is approximate anyway. \*/

**if** **(**events\_check\_enabled**)**

ws**->**wakeup\_count**++;**

**if** **(!**ws**->**active**)**

wakeup\_source\_activate**(**ws**);**

**}**

static void wakeup\_source\_activate**(**struct wakeup\_source **\***ws**)**

**{**

unsigned int cec**;**

ws**->**active **=** true**;**

ws**->**active\_count**++;**

ws**->**last\_time **=** ktime\_get**();**

**if** **(**ws**->**autosleep\_enabled**)** **{**

//pr\_info("[%s]:ws activate->\t%s\n", \_TAG, ws->name);

ws**->**start\_prevent\_time **=** ws**->**last\_time**;**

**}**

/\* Increment the counter of events in progress. \*/

cec **=** atomic\_inc\_return**(&**combined\_event\_count**);**

trace\_wakeup\_source\_activate**(**ws**->**name**,** cec**);**

**}**

在wake\_unlock\_store函数执行过程中，将传到节点的字符串写到pm\_wake\_unlock(buf)中，最后调用到\_\_pm\_relax(&wl->ws)这个函数，下面我们来看看这个函数。

alps\kernel\drivers\base\power\wakeup.c

void \_\_pm\_relax**(**struct wakeup\_source **\***ws**)**

**{**

unsigned long flags**;**

**if** **(!**ws**)**

**return;**

wl\_info**(**"[wake\_unlock] %s\n"**,** ws**->**name**);**

spin\_lock\_irqsave**(&**ws**->**lock**,** flags**);**

**if** **(**ws**->**active**)**

wakeup\_source\_deactivate**(**ws**);**

spin\_unlock\_irqrestore**(&**ws**->**lock**,** flags**);**

print\_active\_wakeup\_sources**();**

**}**

EXPORT\_SYMBOL\_GPL**(**\_\_pm\_relax**);**

其中关键地方都集中在wakeup\_source\_deactivate**(**ws**)**这个函数，它对之前wakelock申请中得到的combined\_event\_count进行操作，减少对wakelock事件的计数。最后通过判断是否存在wakelock事件，以及wakeup\_count\_wait\_queue等待队列是否激活，来唤醒该等待队列。

static void wakeup\_source\_deactivate**(**struct wakeup\_source **\***ws**)**

**{**

unsigned int cnt**,** inpr**,** cec**;**

ktime\_t duration**;**

ktime\_t now**;**

ws**->**relax\_count**++;**

/\*

\* \_\_pm\_relax() may be called directly or from a timer function.

\* If it is called directly right after the timer function has been

\* started, but before the timer function calls \_\_pm\_relax(), it is

\* possible that \_\_pm\_stay\_awake() will be called in the meantime and

\* will set ws->active. Then, ws->active may be cleared immediately

\* by the \_\_pm\_relax() called from the timer function, but in such a

\* case ws->relax\_count will be different from ws->active\_count.

\*/

**if** **(**ws**->**relax\_count **!=** ws**->**active\_count**)** **{**

ws**->**relax\_count**--;**

**return;**

**}**

ws**->**active **=** false**;**

now **=** ktime\_get**();**

duration **=** ktime\_sub**(**now**,** ws**->**last\_time**);**

ws**->**total\_time **=** ktime\_add**(**ws**->**total\_time**,** duration**);**

**if** **(**ktime\_to\_ns**(**duration**)** **>** ktime\_to\_ns**(**ws**->**max\_time**))**

ws**->**max\_time **=** duration**;**

ws**->**last\_time **=** now**;**

del\_timer**(&**ws**->**timer**);**

ws**->**timer\_expires **=** 0**;**

**if** **(**ws**->**autosleep\_enabled**)** **{**

update\_prevent\_sleep\_time**(**ws**,** now**);**

//printk("[%s]:ws deactivate->\t%s\n", \_TAG, ws->name);

**}**

/\*

\* Increment the counter of registered wakeup events and decrement the

\* couter of wakeup events in progress simultaneously.

\*/

// FIXME: CHECK BUG here ??? if combined\_event\_count = 0x????0000, then atomic\_add\_return(...) --> 0x????ffff

// , which is not the expected result !!!

cec **=** atomic\_add\_return**(**MAX\_IN\_PROGRESS**,** **&**combined\_event\_count**);**

trace\_wakeup\_source\_deactivate**(**ws**->**name**,** cec**);**

split\_counters**(&**cnt**,** **&**inpr**);**

//if (ws->autosleep\_enabled && inpr > 0) dump\_active\_ws();

**if** **(!**inpr **&&** waitqueue\_active**(&**wakeup\_count\_wait\_queue**))**

wake\_up**(&**wakeup\_count\_wait\_queue**);**

**}**

# 三．supend

在上一段讲解wakelock的过程中，我们会发现main.c中创建节点时候会创建state\_attr节点，该节点会对机器进行earlysuspend操作。下面我们先来看看state\_store函数。

static ssize\_t state\_store**(**struct kobject **\***kobj**,** struct kobj\_attribute **\***attr**,** const char **\***buf**,** size\_t n**)**

{

…..

#ifdef CONFIG\_EARLYSUSPEND

if (state == PM\_SUSPEND\_ON || valid\_state(state)) {

error = 0;

request\_suspend\_state(state);

} else

error = -EINVAL;

#else

error = enter\_state(state);

#endif

……

}

如果支持earlysuspend就会执行request\_suspend\_state函数，否则就会直接进入linux的suspend流程。在request\_suspend\_state中会根据当前状态判断是否走earlysuspend或者lateresume流程，然后调用对应的工作队列。

void request\_suspend\_state**(**suspend\_state\_t new\_state**)**

**{**

unsigned long irqflags**;**

int old\_sleep**;**

int wait\_flag **=** 0**;**

spin\_lock\_irqsave**(&**state\_lock**,** irqflags**);**

old\_sleep **=** state **&** SUSPEND\_REQUESTED**;**

**if** **(**earlysuspend\_debug\_mask **&** DEBUG\_USER\_STATE**)** **{**

struct timespec ts**;**

struct rtc\_time tm**;**

getnstimeofday**(&**ts**);**

rtc\_time\_to\_tm**(**ts**.**tv\_sec**,** **&**tm**);**

pm\_warn**(**"%s (%d->%d) at %lld "

"(%d-%02d-%02d %02d:%02d:%02d.%09lu UTC)\n"**,**

new\_state **!=** PM\_SUSPEND\_ON **?** "sleep" **:** "wakeup"**,**

requested\_suspend\_state**,** new\_state**,**

ktime\_to\_ns**(**ktime\_get**()),**

tm**.**tm\_year **+** 1900**,** tm**.**tm\_mon **+** 1**,** tm**.**tm\_mday**,**

tm**.**tm\_hour**,** tm**.**tm\_min**,** tm**.**tm\_sec**,** ts**.**tv\_nsec**);**

**}**

**if** **(!**old\_sleep **&&** new\_state **!=** PM\_SUSPEND\_ON**)** **{**

state **|=** SUSPEND\_REQUESTED**;**

pm\_warn**(**"sys\_sync\_work\_queue early\_sys\_sync\_work\n"**);**

queue\_work**(**sys\_sync\_work\_queue**,** **&**early\_sys\_sync\_work**);**

pm\_warn**(**"suspend\_work\_queue early\_suspend\_work\n"**);**

queue\_work**(**suspend\_work\_queue**,** **&**early\_suspend\_work**);**

**}** **else** **if** **(**old\_sleep **&&** new\_state **==** PM\_SUSPEND\_ON**)** **{**

state **&=** **~**SUSPEND\_REQUESTED**;**

//wake\_lock(&main\_wake\_lock);

///cun

**if** **(**queue\_work**(**suspend\_work\_queue**,** **&**late\_resume\_work**))** **{**

/\*

\* In order to synchronize the backlight turn on timing,

\* block the thread and wait for fb driver late\_resume()

\* callback function is completed

\*/

wait\_flag **=** 1**;**

**}**

**}**

requested\_suspend\_state **=** new\_state**;**

spin\_unlock\_irqrestore**(&**state\_lock**,** irqflags**);**

**if** **(**wait\_flag **==** 1**)** **{**

wait\_for\_completion**(&**fb\_drv\_ready**);**

pr\_warn**(**"wait done\n"**);**

**}**

**}**

在上述函数中调用**&**early\_suspend\_work和**&**late\_resume\_work队列实际就是对early\_suspend和late\_resume函数进行操作。在休眠过程中，early\_suspend函数会对链表上的所有元素进行遍历，调用之前平台注册时的earlysuspend函数，逐个实现设备的休眠。

static void early\_suspend**(**struct work\_struct **\***work**)**

**{**

………

**if** **(**earlysuspend\_debug\_mask **&** DEBUG\_SUSPEND**)**

pm\_warn**(**"call handlers\n"**);**

list\_for\_each\_entry**(**pos**,** **&**early\_suspend\_handlers**,** link**)** **{**

**if** **(**pos**->**suspend **!=** **NULL)** **{**

**if** **(!(**forbid\_id **&** **(**0x1 **<<** count**)))** **{**

//if (earlysuspend\_debug\_mask & DEBUG\_VERBOSE)

pr\_warn**(**"ES handlers %d: [%pf], level: %d\n"**,** count**,** pos**->**suspend**,** pos**->**level**);**

pos**->**suspend**(**pos**);**

**}**

count**++;**

**}**

**}**

mutex\_unlock**(&**early\_suspend\_lock**);**

/\* Remove sys\_sync from early\_suspend, and use work queue to complete sys\_sync \*/

abort**:**

**if** **(**state **==** SUSPEND\_REQUESTED\_AND\_SUSPENDED**)** **{**

//wake\_unlock(&main\_wake\_lock);

#ifdef CONFIG\_MTK\_HIBERNATION

suspend\_state\_t susp\_state **=** get\_suspend\_state**();**

pm\_warn**(**"calling pm\_autosleep\_set\_state() with parameter: %d\n"**,** susp\_state**);**

pm\_autosleep\_set\_state**(**susp\_state**);**

#else

pm\_autosleep\_set\_state**(**PM\_SUSPEND\_MEM**);**

#endif

**}**

**}**

函数最后会执行pm\_autosleep\_set\_state,会更新suspend\_work队列，对try\_to\_suspend函数进行调用。

static void try\_to\_suspend**(**struct work\_struct **\***work**)**

**{**

unsigned int initial\_count**,** final\_count**;**

pm\_log**(**"pm\_get\_wakeup\_count\n"**);**

**if** **(!**pm\_get\_wakeup\_count**(&**initial\_count**,** true**))**

**goto** out**;**

mutex\_lock**(&**autosleep\_lock**);**

pm\_log**(**"pm\_save\_wakeup\_count\n"**);**

**if** **(!**pm\_save\_wakeup\_count**(**initial\_count**))** **{**

mutex\_unlock**(&**autosleep\_lock**);**

**goto** out**;**

**}**

**if** **(**autosleep\_state **==** PM\_SUSPEND\_ON**)** **{**

pm\_warn**(**"leaving state(%d)\n"**,** autosleep\_state**);**

#ifdef CONFIG\_MTK\_HIBERNATION

system\_is\_hibernating **=** false**;**

#endif

mutex\_unlock**(&**autosleep\_lock**);**

**return;**

**}**

#ifdef CONFIG\_MTK\_HIBERNATION

**if** **(**autosleep\_state **>=** PM\_SUSPEND\_MAX**)** **{**

mtk\_hibernate\_via\_autosleep**(&**autosleep\_state**);**

**}**

**else** **{**

pm\_log**(**"pm\_suspend: state(%d)\n"**,** autosleep\_state**);**

**if** **(!**system\_is\_hibernating**)** **{**

pm\_warn**(**"calling pm\_suspend() state(%d)\n"**,** autosleep\_state**);**

pm\_suspend**(**autosleep\_state**);**

**}**

**else** **{**

pm\_warn**(**"system is hibernating: so changing state(%d->%d)\n"**,** autosleep\_state**,** PM\_SUSPEND\_MAX**);**

autosleep\_state **=** PM\_SUSPEND\_MAX**;**

**}**

**}**

#else // !CONFIG\_MTK\_HIBERNATION

**if** **(**autosleep\_state **>=** PM\_SUSPEND\_MAX**)**

hibernate**();**

**else** **{**

pm\_log**(**"pm\_suspend\n"**);**

pm\_suspend**(**autosleep\_state**);**

**}**

#endif // CONFIG\_MTK\_HIBERNATION

mutex\_unlock**(&**autosleep\_lock**);**

**if** **(!**pm\_get\_wakeup\_count**(&**final\_count**,** false**))**

**goto** out**;**

/\*

\* If the wakeup occured for an unknown reason, wait to prevent the

\* system from trying to suspend and waking up in a tight loop.

\*/

**if** **(**final\_count **==** initial\_count**)**

schedule\_timeout\_uninterruptible**(**HZ **/** 2**);**

out**:**

pm\_log**(**"queue\_up\_suspend\_work again\n"**);**

queue\_up\_suspend\_work**();**

**}**

我们重点关住pm\_get\_wakeup\_count函数，这个函数中会有一个循环等待&wakeup\_count\_wait\_queue队列，该队列之前在wake\_unlock中遇到过。

bool pm\_get\_wakeup\_count**(**unsigned int **\***count**,** bool block**)**

**{**

unsigned int cnt**,** inpr**;**

**if** **(**block**)** **{**

DEFINE\_WAIT**(**wait**);**

**for** **(;;)** **{**

prepare\_to\_wait**(&**wakeup\_count\_wait\_queue**,** **&**wait**,**

TASK\_INTERRUPTIBLE**);**

split\_counters**(&**cnt**,** **&**inpr**);**

**if** **(**inpr **==** 0 **||** signal\_pending**(**current**))**

**break;**

print\_active\_wakeup\_sources**();**

schedule**();**

**}**

finish\_wait**(&**wakeup\_count\_wait\_queue**,** **&**wait**);**

**}**

split\_counters**(&**cnt**,** **&**inpr**);**

**\***count **=** cnt**;**

**return** **!**inpr**;**

**}**

这里会判断是否有wakelock锁，如果有直接返回继续等待suspend\_work队列更新，如果没有wakelock锁就会直接进入suspend标准流程。

# 四．问题分析与解决

1、初步抓取mobilelog，分析kernellog中，可以找到大量的关键log

<6>[ 471.067370] 0)active wakeup source: EINT wakelock

<3>[ 471.067396] 0)PM: Some devices failed to suspend

判断应该是休眠后申请了唤醒锁，mtk要求加如下打印。

kernel/kenrel/power/wakelock.c 文件，修改以下全局变量：

int userwakelock\_debug\_mask = 1;

kernel/drivers/base/power/wakeup.c 文件，修改以下全局变量定义：

int wakelock\_debug\_mask = 1;

Kernel/drivers/base/power/main.c文件，修改以下宏定义

#define HIB\_DPM\_DEBUG 1

修改 \_\_pm\_relax 函数以下代码：

void \_\_pm\_relax(struct wakeup\_source \*ws)

{

unsigned long flags;

if (!ws)

return;

wl\_info("[wake\_unlock] %s\n", ws->name);

spin\_lock\_irqsave(&ws->lock, flags);

if (ws->active)

wakeup\_source\_deactivate(ws);

spin\_unlock\_irqrestore(&ws->lock, flags);

// 增加这一行

print\_active\_wakeup\_sources();

}

在kernel/drivers/base/power/wakeup.c文件中的pm\_wakeup\_pending()函数里的

print\_active\_wakeup\_source();函数后面添加上mt\_eint\_print\_status();

2、加上述打印后再次抓取mobilelog，可以看到如下信息

282.769015] 0)[Power/Kernel][wake\_lock\_timeout] EINT wakelock,500ms

<6>[ 282.770048]-1)[Power/clkmgr] [clkmux\_disable\_op]: mux->name=MUX\_UART

<5>[ 282.770067] 1)[UART0] Suspend(0)!

<4>[ 282.770112] 1)[Power/Kernel][pm\_wakeup\_pending]:cnt=6660,saved\_count=6660,inpr=1

<6>[ 282.770141] 1)active wakeup source: EINT wakelock

<4>[ 282.770157] 1)[HIB/DPM][\_\_device\_suspend]async\_error(-16) not zero due pm\_wakeup\_pending return non zero!!

<4>[ 282.770180] 1)[HIB/DPM][dpm\_suspend]async\_error(-16)

<4>[ 282.770201] 1)[HIB/DPM][dpm\_suspend]return error(-16)

<3>[ 282.770211] 1)PM: Some devices failed to suspend

有一个大约500ms的中断申请了wakelock锁，导致系统不能正常休眠。

3、将kernel/mediatek/platform/mt6582/kernel/core/enit.c里面的#define EINT\_DEBUG 0修改为1，再次抓取log，可以查看到如下信息

81]-(0)[0:swapper/0]Got eint:2

<4>[ 224.872200]-(0)[0:swapper/0][EINT] mask addr:f000b0c0 = 4

<4>[ 224.872229] (2)[726:InputReader][Power/Kernel][wake\_unlock] KeyEvents

<4>[ 224.872251]-(0)[0:swapper/0][EINT] mt\_eint\_ack :f000b040, bit: 4

<4>[ 224.872271]-(0)[0:swapper/0]EINT Module - EINT\_STA after ack = 0x0

<4>[ 224.872293]-(0)[0:swapper/0][EINT] unmask addr:f000b100 = 4

<4>[ 224.872322]-(0)[0:swapper/0]<6>active wakeup source: PowerManagerService.Display

<6>[ 224.872340] (2)[726:InputReader]active wakeup source: PowerManagerService.Display

<6>[ 224.872368] (2)[726:InputReader]active wakeup source: EINT wakelock

<4>[ 224.872397] (0)[0:swapper/0][Power/Kernel][wake\_lock\_timeout] EINT wakelock,500ms

<4>[ 224.881406]-(1)[79:mtk-tpd][Power/Kernel][wake\_lock] event3-682

<4>[ 224.881493] (1)[79:mtk-tpd][EINT] unmask addr:f000b100 = 4

<4>[ 224.881795] (1)[726:InputReader][user\_wake\_lock] KeyEvents

<4>[ 224.881837] (1)[726:InputReader][Power/Kernel][wake\_lock] KeyEvents

可知中断2导致了wakelock被申请，查看dct配置，得知该中断为hal中断，机器没有使用该中断，去除该配置，编译烧录后发现待机电流恢复正常。

# 五．总结

通过这次问题，学习到了一些新的调试方法，也熟悉了一下suspend这块的代码，感觉模块的熟悉程度以及调试方法对解决问题有着很大的帮助，以后会多加强这方面的练习。