量产系统稳定性问题

排查指南

版本号: 2.0

发布日期: 2024.10.26

版本历史

版本号	日期	制/修订人	内容描述
0.1	2018.12.20	AW1226	增加《系统死机类》章节
0.2	2018.12.22	AW1151	增加《系统内存类》章节
0.3	2018.12.22	AW1237	增加《休眠唤醒类》章节
0.4	2018.12.22	AWA0863	增加《重启类》章节
0.5	2019.2.25	AW1556	增加《休眠唤醒类》章节节点说明
1.0	2019.2.25	AWA0863	初稿发布
1.1	2019.9.23	AWA0863	增加《休眠唤醒类》章节通用排查方法
1.3	2020.11.6	AW1691	转换成 markdown 格式
1.4	2021.1.13	AW1691	1. 增加 pmu 电池相关的内容 2. 将文档排版得符合外发规范
1.5	2021.11.16	AWA0863	增加系统死机类检查清单
1.6	2023.08.15	AWA1743	增加系统死锁类检查清单
1.7	2023.05.15	AWA0863	增加休眠时间过长问题排查
1.8	2023.10.16	AWA1743	增加系统总线卡死检查清单
1.9	2024.03.11	AWA2159	修改已经失效文档指引，新增部分新平台休眠异常排查描述
2.0	2024.10.26	AWA2256	修改文档指引

目　录

插　图

表　格

1 前言

1.1 文档简介

- 本文档用于快速排查系统稳定性问题，包括系统死机类、系统内存类、休眠唤醒类以及重启类等问题；
- 本文档提供的快速排查方案模型如下：

常规问题场景 + 基础硬件排查 + 基础软件配置排查 + 软件快速调试排查

- 本文档仅用于帮助开发人员快速排查和解决初级问题，如果按照文档中的指导仍无法解决问题，请联系全志原厂工程师进一步协助解决，并提供说明是否按照文档中的排查方案进行过必要的排查，提供排查过程的必要记录和数据
- 由于每个问题涉及的可能模块比较多，因而根据问题来进行分类。每个问题下会列出相关可能模块。

1.2 目标读者

本文档主要适用于:

- 负责嵌入式系统、驱动、稳定性开发人员

1.3 适用范围

表 1-1: 适用产品列表

产品名称	内核版本
所有产品	Linux-4.9，Linux-5.4，Linux-5.10，Linux-5.15

2 系统死机类

系统死机主要分为软件系统 crash、软件系统 block、硬件异常、供电异常、未知异常等。

2.1 检查清单

系统正常稳定运行，是产品稳定性中最关键的一项质量要求，全志对外发布 SDK 是经过了严格的全面老化测试。

当客户端拿到 SDK 经过开发定制，运行在自己的产品工程样机上，发现系统正常运行时无故死机，很大可能是工程开发过程中，软件和硬件变化等因素引起。常见问题有 dram 物料更换、电源方案修改、sys_config 修改、dts 修改、menuconfig 修改等。建议按照如下流程检查，若均未发现异常，再进行后面的系统死机类问题分类排查。

2.1.1 环境配置检查

- 步骤 1：

检查工程环境的问题。确认使用的工程环境，对比 SDK 包没有非预期的或不确定的改动，保证环境是干净的。

- 步骤 2：

检查板级配置。对比 SDK 包默认的板级方案参考配置，sys_config 和 dts 配置是否正确无误，cpu v-f 表是否有更新和是否有改动、dram 驱动版本是否有更新或改动、供电部分是否有改动、各路供电的开电延时和调压延时是否有改动、dram 参数部分是否有改动，改动是否合理、是否经过方案硬件设计人员审核。

说明

cpu v-f 表相关配置详见一号通系统《Linux_CPUFREQ_ 开发指南》文档或咨询方案开发人员。供电相关配置详见一号通系统《Linux_PMIC_ 开发指南》文档或咨询方案开发人员。dram 驱动版本和相关配置咨询方案开发人员或 dram 专业人士。其他未见事项参考一号通《产线量产稳定性检查 checklist》文档。

2.1.2 硬件设计检查

- 步骤 1：

检查板子硬件设计。对比 SDK 包的硬件参考设计原理图和 PCB，检查系统供电设计是否存在不同、上下电时序是否不同、关键供电的电容组合数量位置和 PCB 走线是否不同、dram 模板是否不同是否需要特殊处理，这些不同是否合理、是否经过方案硬件设计人员审核。

说明

硬件设计检查详见一号通《产线量产稳定性检查 checklist》文档。

- 步骤 2：

检查物料。dram、emmc、nand、nor 等物料是否在支持列表里面，是否为拆机料，是否有混料，是否经过方案硬件设计人员确认。

说明

物料检查详见一号通《产线量产稳定性检查 checklist》文档。

2.1.3 样机硬件检查

- 步骤 1：

检查电源。确认外部供电如适配器、电池供电能力是否满足要求，实测样机的上电时序、供电电压、电源纹波，特别是 VDD-CPU、VDD-SYS、AVCC 等供电，是否与供电配置一致、是否正常。

说明

电源检查详见一号通《产线量产稳定性检查 checklist》。

- 步骤 2：

检查 cpu、dram 频率。实测样机的 dram 频率、cpu v-f 表各个频点的电压和频率，是否与配置一致、是否正常。

说明

cpu v-f 表各个频点的电压和频率详见一号通《Linux_CPUFREQ_ 开发指南》文档或咨询方案开发人员。dram 频率可以从启动打印获知 “DRAM CLK = xxx MHz”。其他未见事项参考一号通《产线量产稳定性检查 checklist》文档。

- 步骤 3：

检查 dram 稳定性。使用 DragonHD、全盘扫描、“memtester + 捕鱼达人” 进行 dram 老化测试。

说明

dram 相关测试咨询方案开发人员或 dram 专业人士。

2.2 系统 crash 类

2.2.1 问题现象

主要分为随机性 crash 和固定位置 crash。前者一般由硬件引起，后者则一般为软件问题

1. 内核打印报错 oops/pani, 包含如下关键字：

```
Internal error: Oops: 817 [#1] PREEMPT SMP ARM
end Kernel panic - not syncing: Fatal exception
```

2.android 系统卡死，内核控制台无法响应，内核打印无任何异常信息

2.2.2 问题分析

当 linux kernel 遇到无法处理的异常的时候, 就会报错 oops、panic。内核 oops、panic 可能是下面几种原因：

- 硬件问题
  - ddr 异常
  - cpu 异常
  - 供电异常
- 软件问题
  - 内核代码逻辑 bug
  - 内存异常篡改

排查顺序：

1. **crash 种类排查：**固定位置 crash→ 随机性 crash 如果 3 次以上 crash 情况的都是在同一个位置，那么就判断为固定位置 crash
2. **问题位置排查：**硬件问题 → 软件问题
3. **硬件排查顺序：** ddr→cpu→ 供电

2.2.3 随机性 crash 排查

如果是量产方案，出现系统随机性的 crash，这种情况更多的会是硬件问题，可以从 ddr、cpu、供电三个方向按照顺序进行排查。

2.2.3.1 排查 ddr

- 步骤 1

memtester 测试实验。可以通过 memtester 测试来验证是否 ddr 存在异常。需要在常温、低温、高温下进行测试，为简化实验，可以使用制冷剂、热风枪等替代温箱进行实验。

```
####memtester测试大小随方案dram大小而定，如1G方案可以使用3个128M的测试###
#memtester 128M &
```

如果测试存在报错，则需找 dram 专业人士调整 dram 参数或者排查 dram 硬件问题。

- 步骤 2

减低 dram 频率实验。减低 dram 频率到 360M，可以保证 dram 的稳定性，可能 dram 在板子上运行不稳定或者参数不适配等。

说明

通过修改 sys_config 中的 dram_para 参数来进行 dram 频率的修改

```
[dram_para]
dram_clk=360
```

如果此实验正常下工作正常，则需找 dram 专业人士调整 dram 参数或者排查 dram 硬件问题。

- 步骤 3

dram 自刷新实验。可以通过休眠唤醒 dram 校验功能来验证 dram 进出自刷新后数据是否丢失。需要在常温、低温、高温下进行测试。

说明

打开休眠唤醒 dram 校验功能，详见本文 “设置 dram 校验功能” 章节

如果测试存在报错，则需找 dram 专业人士调整 dram 参数或者排查 dram 硬件问题。

- 步骤 4

dram 换料实验。更换品质更高 dram 物料或者对调 dram 物料进行实验。

说明

通过修改 sys_config 中的 dram_para 参数来进行 dram 参数的调整

如果此实验正常下工作正常，则需找 dram 专业人士调整 dram 参数或者排查 dram 硬件问题。

2.2.3.2 排查 cpu

cpu 异常的可能原因：

- cpu 供电异常
- cpu 多核异常
- cpu 调频异常
- cpu 调压异常
- ic 焊接不良
- cpuvf 不匹配或者不良

排查步骤：

- 步骤 1

先排查 ddr，在保证 ddr 正常工作的情况下来排查 cpu。

- 步骤 2

然后排查多核异常，即屏蔽被怀疑的异常核再测试（直接在 dtsi 把该核去掉）。

- 步骤 3

通过做**定频定核提压**实验可以快速确认问题是否与 cpu 相关。如果测试通过，则说明极可能与 cpu 相关，可做进一步的排查分析。

说明

测试通过是指通过进行该测试后，异常消失。反之，测试不通过是指进行了该测试后，异常依旧存在。

- 步骤 4

根据具体的情况进行 cpu 问题排查。

1. 内核随机 crash，出现异常 log
2. 内核无异常 log

- 步骤 5-1 内核随机 crash 排查

1. 进行 **cpu 提压**实验和**提高 sys 电压**的实验，确认是否为 vf 不匹配。
   如提压实验通过，则说明 cpu 或 sys 的 vf 不匹配。如果实验失败，进行下一步实验。

2. **ic 更换实验**。将正常板子上的 ic 替换到问题板上，将问题板上的 ic 替换到正常板子上。如果两块板子都测试通过，说明为焊接不良。如果问题板 + 正常板 ic 测试通过，正常板 + 异常板 ic 测试失败，则说明为 ic 不良。如果问题板 + 正常板 ic 测试失败，正常板 + 异常板 ic 测试正常，则说明为板级不良。

- 步骤 5-2 内核无异常 log 排查

1. 进行 **cpu 定频定压**实验。

   如果测试通过，则可参考后文实验中的说明做电压相关排查。如果电压排查未发现问题，则进行下一个实验。
2. 进行 **cpu 定压提压**实验。如果测试不通过，则为 cpu 调频存在问题。如果测试通过，则为 cpu vf 不匹配问题。

实验 1：定频定核提压实验

第一步-定频提压

配置 sys_config 中的所有 dvfs table 中所有频点为 1G 或者修改 dts 中的 opp 表所有频点为 1G，电压使用最高频点电压。

```
 L_LV_count = 8

-L_LV1_freq = 1104000000
+L_LV1_freq = 1008000000
 L_LV1_volt = 1300

 L_LV2_freq = 1008000000
-L_LV2_volt = 1200
+L_LV2_volt = 1300

-L_LV3_freq = 912000000
-L_LV3_volt = 1120
+L_LV3_freq = 1008000000
+L_LV3_volt = 1300
 ...
```

如果 dvfs 配置在 dts 中的 opp_table，那么同理的，按上面的方法修改 dts 中的 opp 表即可。如下示例演示如何提高 40mv sys 电压：

查看硬件原理图 sys 是用的哪路电, 再修改 sys_config 中的供电配置。如 sys 是使用 dcdc3 并默认值为 900mv, 可做如下修改：

```
[power_sply]
dcdc3_vol=100940
```

第二步-关闭温控系统

在 menuconfig 下将 thermal 关闭

```
make ARCH=arm menuconfig或者make ARCH=arm64 menuconfig
Device Driver -->关闭 Generic Thermal sysfs drivers
```

第三步-定核

打开所有核，并锁定。

说明

以下命令是以 ic 为四核 cpu 为例去锁定四核。具体锁核的数量根据具体 ic 来定。

```
echo 4 > /sys/kernel/autohotplug/lock
```

如果通过该实验，cpu 恢复正常，则可认为是 cpu 类的问题，需进一步实验排查。如果该实验测试后依旧存在异常，则可能是：

- 焊接不良先排查是否为焊接不良，重新补焊实验验证。
- IC 为不良品再排查是否为不良 ic。可通过替换 ic 的操作验证是否为 ic 是否为不良品。把系统运行正常的板子上良品 ic 拆下来，替换到运行异常的板子上，如果良品 ic 能够在原本异常的板子上正常运行，则说明异常板上原本的 ic 是不良品。如果是不良 ic，交由硬件同事确认，并需增加筛选条件将此类不良品筛除.

实验 2：cpu 提压实验

定压的方法：

配置 sys_config 中的**所有** dvfs table 中**所有**频点的电压都提高 60mv。

```
; sys_config dvfs_table
 L_LV_count = 8

L_LV1_freq = 1104000000
 L_LV1_volt = 1360

 L_LV2_freq = 1008000000
L_LV2_volt = 1300

L_LV3_freq = 912000000
L_LV3_volt = 1220
 ...
```

说明

如果是 opp_table，那么同理的，按上面的方法修改 dts 中的 opp 表即可

如果提压实验通过，则可能是：

- cpu 性能不足，需要更高电压.
- cpu 电压异常。

实验 3：提高 sys 电压

提高 40mv sys 电压。：查看硬件原理图 sys 是用的哪路电, 再修改 sys_config 中的供电配置。如 sys 是使用 dcdc3 并默认值为 900mv, 可做如下修改：

```
[power_sply]
dcdc3_vol=100940
```

如果修改 sys 电压后，测试通过，则可能是以下几种情况：

- sys 电压供电不稳定，包括供电不足、纹波过大的情况。
- IC 性能不足

实验 4：cpu 定频定压实验

cpu 定频定压实验排查。cpu **定频**可以验证 cpu **调频**是否稳定。cpu **定压**可以验证 cpu **调压**是否稳定。电压不稳定的情况分为以下几种：

- 实际电压低于设置电压，例如设置 0.9v，实际只有 0.89v。
- 电压 vdrop 过大。
- 调压过程需要一定的时间，必须保证调压过程中等待电压稳定时间的充裕。

步骤 1

定频定压：配置 sys_config 中的**所有** dvfs table 中**所有**频点为 1G 或者修改 dts 中的 opp 表所有频点为 1G，并把频点对应的电压使用原来的电压。

linux-3.10 支持 dvfs 表方式：

```
[vf_table0]
L_max_freq =1008000000
L_min_freq = 1008000000
```

linux-4.4 及以上：

```
    cpu_opp_l_table0: opp_l_table0 {
      /* compatible = "operating-points-v2"; */
      compatible = "allwinner,opp_l_table0";
      opp_count = <1>;
      opp-shared;

      opp00 {
        ;只留一个频点
        opp-hz = /bits/ 64 <1008000000>;
        opp-microvolt = <880000>;
        axi-bus-divide-ratio = <3>;
        clock-latency-ns = <2000000>;
      };

/*;屏蔽其他频点
      opp01 {
        opp-hz = /bits/ 64 <720000000>;
        opp-microvolt = <880000>;
        axi-bus-divide-ratio = <3>;
        clock-latency-ns = <2000000>;
      };
      ...
```

```
*/
```

如果定频定压实验下，测试通过，则问题原因可能是下列几种：

- cpu 调频存在异常
- 实际电压低于设置电压，例如设置 0.9v，实际只有 0.89v。
- 电压 vdrop 过大。
- 调压过程需要一定的时间，必须保证调压过程中等待电压稳定时间的充裕。

后面三种与电压相关的情况，都可以通过抓取硬件信息去确认。如果电压的这三种情况都确认没有问题，那么就进行后面的步骤验证 cpu 调频是否存在异常。

实验 5：cpu 定压提压

不调节 cpu 电压，只调节 cpu 频率。

定压并提压的方法：配置 sys_config 中的**所有** dvfs table 中**所有**频点的电压为最大频点的电压 +60mv。

```
; sys_config dvfs_table
 L_LV_count = 8

L_LV1_freq = 1104000000
 L_LV1_volt = 1360

 L_LV2_freq = 1008000000
L_LV2_volt = 1360

L_LV3_freq = 912000000
L_LV3_volt = 1360
 ...
```

说明

如果是 opp_table，那么同理的，按上面的方法修改 dts 中的 opp 表即可

如果该实验不通过，则说明为调节 cpu 频率存在异常。

如果该实验通过，则可能是：

- cpu 性能不足。
- cpu 电压异常。

2.2.3.3 排查供电是否正常

步骤 1

```
1.确认外部供电如适配器、电池供电能力是否满足要求。
2.测量cpu供电电压纹波看是否符合规格。
```

步骤 2

1.方案硬件同事确认外部供电满足要求，或使用更可靠的外部供电，排除疑点。
2.用示波器抓取cpu电压，看cpu供电电压是否存在抖动。

2.2.3.4 排查软件问题

随机性 crash 问题一般都是与硬件相关，但如果上述排查实验过后依然找不出问题，那可能是软件问题了。软件方向可以从代码逻辑 bug 和内存篡改方向排查。

2.2.4 固定位置 crash 排查

固定位置 crash 的判断标准：如果 3 次以上 crash 情况的都是在同一个位置。此类固定位置 crash 的问题一般为软件问题，可采用 addr2line、objdump、crashdump 等工具进行正面分析，查明原因。

- addr2line、objdump 这两个工具可以参考网上例子，此类资料网上较多不在这里展开
- crashdump 工具的使用详见一号通《Crashdump_ 使用指南》

2.3 系统 block 类

2.3.1 内核死锁

说明

关键词：softlockup、hardlockup、hang、blocked、detected stalls

2.3.1.1 系统死锁问题

有如下现象之一就可以怀疑是系统死锁问题：

- 使用串口，按键，屏幕等外设都没有反应
- 某应用进程阻塞
- watchdog 未喂狗
- 内核 lockup 类报错

```
kernel: BUG: softlockup - CPU#11 stuck for 4278190091s! [qmgr:5492]

Kernel panic - not syncing: Watchdog detected hard LOCKUP on cpu 1
```

```
INFO: task Appxxx:13189 blocked for more than 120 seconds

INFO: rcu_preempt detected stalls on CPUs/tasks:
 1-...: (25112 GPs behind) idle=9da/0/0 softirq=84384/84385 fqs=0
 (detected by 0, t=25211 jiffies, g=75984, c=75983, q=188)
```

2.3.1.2 系统死锁类问题分析

内核中的锁是用来同步不同线程之间对共享数据的访问的。

内核有着完善的锁机制，各种函数调用嵌套，锁没有管理好的话很容易出现死锁现象。当然，内核也有着完善的死锁检查机制可以帮助我们 debug 死锁类问题。

遇到死锁问题，除了正面走查代码分析，还可以打开内核相关的死锁检测 debug 选项来帮助我们排查。

2.3.1.3 系统死锁类问题解决

- 步骤 1 如果有 lockup 报错，可从代码正面分析问题所在。当然排查死锁类问题需要我们熟悉锁的同步机制，知道如何正确的使用锁。
- 步骤 2 确认已经打开以下内核配置

```
CONFIG_MAGIC_SYSRQ=y
```

利用 sysrq-trigger dump 出内核所有线程的堆栈信息，帮助我们分析系统中的用锁情况。

```
echo t&gt > proc/sysrq-trigger
```

- 步骤 3 打开锁的 debug 调试选项，然后复现问题。在不清楚是哪类死锁问题的时候可以把 **kernel hacking 中所有的锁 debug 选项都打开**，然后再复现问题。如果是死锁，内核会把死锁类型、死锁处的堆栈等信息 dump 出来，帮助开发人员快速定位问题。内核锁相关配置如下：

表 2-1: 内核锁相关配置

含义	配置
死锁检查器	CONFIG_LOCKUP_DETECTOR=y CONFIG_DEBUG_LOCK_ALLOC=y CONFIG_PROVE_LOCKING=y CONFIG_LOCKDEP=y CONFIG_DEBUG_LOCKDEP=y
hung task 检查机制	CONFIG_DETECT_HUNG_TASK=y

含义	配置
	CONFIG_DEFAULT_HUNG_TIMEOUT=120
mutex 锁调试	CONFIG_DEBUG_MUTEXES=y
	CONFIG_DEBUG_RT_MUTEXES=y
spin lock 锁调试	CONFIG_DBEUG_SPINLOCK=y
锁使用信息	CONFIG_LOCK_STAT
debug 原子环境是否会睡眠	CONFIG_DEBUG_ATOMIC_SLEEP
锁自检	CONFIG_DBEUG_LOCKING_API_SELFTESTS

注：CONFIG_PROVE_RAW_LOCK_NESTING 配置项只在 rt-linux 上支持，即未打 linux rt patch 时该配置项是无效的，在未打 rt patch 时打开该配置项会报如下警告，原因是 spin_lock/spin_lock_irq/spin_lock_irqsave 接口在 rt-linux 环境只是普通的互斥量，该类接口会睡眠，故不能在中断服务程序中调用，即在非 rt-linux 环境下打开死锁调试配置项时不需要打开该配置项：

```
[13.34.02.975] [    6.731364] ============================
[13.34.02.975] [    6.731367] [ BUG: Invalid wait context ]
[13.34.02.975] [    6.731371] 5.15.78 #1 Not tainted
[13.34.02.976] [    6.731375] ----------------------------
[13.34.02.976] [    6.731377] swapper/0/1 is trying to lock:
[13.34.02.976] [    6.731381] ffffffc0092f2c50 (task_time_in_state_lock){....}-{3:3}, at: cpufreq_acct_update_power+0x74/0
        xfc
[13.34.02.976] [    6.731410] other info that might help us debug this:
[13.34.02.976] [    6.731412] context-{2:2}
[13.34.02.976] [    6.731415] 6 locks held by swapper/0/1:
[13.34.02.976] [    6.731420]  #0: ffffff80039d91a0 (&dev->mutex){....}-{4:4}, at: __driver_attach+0x7c/0x194
[13.34.02.976] [    6.731441]  #1: ffffffc0091dee08 (cpu_hotplug_lock){++++}-{0:0}, at: cpus_read_lock+0x10/0x20
[13.34.02.977] [    6.731463]  #2: ffffff80037f8528 (subsys mutex#7){+.+.}-{4:4}, at: subsys_interface_register+0x50/0x114
[13.34.02.977] [    6.731487]  #3: ffffff8003b03bb8 (&policy->rwsem){+.+.}-{4:4}, at: cpufreq_online+0x458/0x910
[13.34.02.977] [    6.731507]  #4: ffffffc0091eb9a0 (em_pd_mutex){+.+.}-{4:4}, at: em_dev_register_perf_domain+0x64/0
        x5dc
[13.34.02.978] [    6.731530]  #5: ffffffc0091ebb60 (console_lock){+.+.}-{0:0}, at: vprintk_emit+0x108/0x330
[13.34.02.978] [    6.731548] stack backtrace:
[13.34.02.978] [    6.731551] CPU: 7 PID: 1 Comm: swapper/0 Not tainted 5.15.78 #1
[13.34.02.978] [    6.731558] Hardware name: sun55iw3 (DT)
[13.34.02.978] [    6.731562] Call trace:
[13.34.02.978] [    6.731564]  dump_backtrace+0x0/0x1a0
[13.34.02.978] [    6.731573]  show_stack+0x18/0x24
[13.34.02.978] [    6.731580]  dump_stack_lvl+0xb0/0xf4
[13.34.02.978] [    6.731589]  dump_stack+0x18/0x34
[13.34.02.978] [    6.731596]  __lock_acquire+0x934/0x1920
[13.34.02.979] [    6.731602]  lock_acquire+0x16c/0x394
[13.34.02.979] [    6.731608]  _raw_spin_lock_irqsave+0x88/0x1ec
[13.34.02.979] [    6.731617]  cpufreq_acct_update_power+0x74/0xfc
[13.34.02.979] [    6.731624]  account_system_index_time+0x78/0x8c
[13.34.02.979] [    6.731632]  account_system_time+0x78/0x90
[13.34.02.979] [    6.731638]  account_process_tick+0x44/0xac
[13.34.02.979] [    6.731644]  update_process_times+0x58/0xf0
[13.34.02.979] [    6.731652]  tick_sched_handle+0x30/0x70
[13.34.02.979] [    6.731659]  tick_sched_timer+0x4c/0xac
[13.34.02.979] [    6.731665]  __hrtimer_run_queues+0xe0/0x530
[13.34.02.979] [    6.731672]  hrtimer_interrupt+0xe8/0x244
[13.34.02.979] [    6.731679]  arch_timer_handler_virt+0x34/0x4c
[13.34.02.979] [    6.731688]  handle_percpu_devid_irq+0x8c/0x150
```

```
[13.34.02.979] [   6.731696] handle_domain_irq+0xbc/0x100
[13.34.02.982] [   6.731702] gic_handle_irq+0xe4/0x180
[13.34.02.982] [   6.731710] call_on_irq_stack+0x28/0x54
[13.34.02.982] [   6.731717] do_interrupt_handler+0x54/0x60
[13.34.02.982] [   6.731724] el1_interrupt+0x30/0x124
[13.34.02.982] [   6.731730] el1h_64_irq_handler+0x18/0x24
[13.34.02.982] [   6.731735] el1h_64_irq+0x78/0x7c
[13.34.02.982] [   6.731741] console_unlock+0x2b8/0x610
[13.34.02.982] [   6.731749] vprintk_emit+0x110/0x330
[13.34.02.982] [   6.731754] dev_vprintk_emit+0x140/0x178
[13.34.02.982] [   6.731760] dev_printk_emit+0x58/0x80
[13.34.02.982] [   6.731765] __dev_printk+0x4c/0x68
[13.34.02.982] [   6.731771] _dev_printk+0x54/0x7c
[13.34.02.982] [   6.731776] em_dev_register_perf_domain+0x538/0x5dc
[13.34.02.983] [   6.731784] dev_pm_opp_of_register_em+0x98/0xe0
[13.34.02.983] [   6.731790] cpufreq_register_em_with_opp+0x24/0x34
[13.34.02.983] [   6.731799] cpufreq_online+0x224/0x910
[13.34.02.983] [   6.731806] cpufreq_add_dev+0x74/0x90
[13.34.02.983] [   6.731812] subsys_interface_register+0x104/0x114
[13.34.02.983] [   6.731820] cpufreq_register_driver+0x140/0x290
[13.34.02.983] [   6.731827] dt_cpufreq_probe+0x20c/0x400
[13.34.02.983] [   6.731834] platform_probe+0x68/0xe0
[13.34.02.983] [   6.731841] really_probe.part.0+0x9c/0x31c
[13.34.02.983] [   6.731846] __driver_probe_device+0x98/0x144
[13.34.02.983] [   6.731852] driver_probe_device+0x44/0x120
[13.34.02.983] [   6.731858] __driver_attach+0x88/0x194
[13.34.02.983] [   6.731864] bus_for_each_dev+0x70/0xd0
[13.34.02.983] [   6.731871] driver_attach+0x24/0x30
[13.34.02.983] [   6.731876] bus_add_driver+0x10c/0x200
[13.34.02.985] [   6.731881] driver_register+0x78/0x130
[13.34.03.200] [   6.731887] __platform_driver_register+0x28/0x34
[13.34.03.201] [   6.731894] dt_cpufreq_platdrv_init+0x1c/0x28
[13.34.03.201] [   6.731903] do_one_initcall+0x8c/0x410
[13.34.03.201] [   6.731909] kernel_init_freeable+0x1c8/0x22c
[13.34.03.201] [   6.731916] kernel_init+0x24/0x120
[13.34.03.201] [   6.731922] ret_from_fork+0x10/0x20
```

- 步骤 4 根据堆栈信息或者 debug 信息，走查代码，分析锁的使用情况，查明问题真凶！

2.3.2 系统中断

2.3.2.1 系统中断问题

有如下现象出现就怀疑是某个 CPU 产生过多中断引起：

CPU0 在 rcu_preempt detected stall 以后，一直打印下面的 log，就说明 CPU0 一直在运行 irq_handle 程序引起的 rcu stall:

```
[   C0] watchdog: BUG: soft lockup - CPU#0 stuck for 42785s! [swapper/0:0]
[   C0] Modules linked in: standby_debug(E) ... ...
[   C0] CPU: 0 PID: 0 Comm: swapper/0 Tainted: G        OEL    5.15.119-00002-g30a16a304cf5 #18
[   C0] Hardware name: sun55iw3 (DT)
[   C0] pstate: 40400005 (nZcv daif +PAN -UAO -TCO -DIT -SSBS BTYPE=--)
[   C0] pc : _stext+0xc8/0x4c8
```

```
[   C0] lr : _stext+0x90/0x4c8
[   C0] sp : ffffffc008003f00
[   C0] x29: ffffffc008003f20 x28: ffffffc009323188 x27: 0000000000000000
[   C0] x26: 0000000000000010 x25: 0000000000000000 x24: ffffffc00a879e00
[   C0] x23: ffffffc00a847828 x22: ffffffc00a856a00 x21: 0000000000000000
[   C0] x20: ffffffc00a8750c0 x19: ffffffc00a88e840 x18: ffffffc008005020
[   C0] x17: ffffffc0f4ded000 x16: ffffffc008000000 x15: 5fb3804cabab525d
[   C0] x14: 00000000000001fd x13: 0000000000000004 x12: 000000004d614984
[   C0] x11: 00000000ffffff00 x10: 0000000c98ce0652 x9 : 0000000000000000
[   C0] x8 : 00000000000000e0 x7 : 0000000000000000 x6 : 0000000000000000
[   C0] x5 : ffffffc008003cd0 x4 : 0000000000000000 x3 : 0000000000000177
[   C0] x2 : 0000000000000001 x1 : 0000000000000100 x0 : 0000000c98ce07c9
[   C0] Call trace:
[   C0]  _stext+0xc8/0x4c8
[   C0]  __irq_exit_rcu.llvm.12962629528475653967+0x74/0x104
[   C0]  handle_domain_irq+0xa0/0x120
[   C0]  gic_handle_irq+0x58/0x124
[   C0]  call_on_irq_stack+0x3c/0x70
[   C0]  do_interrupt_handler+0x44/0xa0
[   C0]  el1_interrupt+0x34/0x64
[   C0]  el1h_64_irq_handler+0x1c/0x2c
[   C0]  el1h_64_irq+0x7c/0x80
[   C0]  arch_local_irq_enable+0xc/0x18
[   C0]  default_idle_call+0x40/0x15c
[   C0]  do_idle.llvm.777118578527097947+0xbc/0x140
[   C0]  cpu_startup_entry+0x28/0x2c
[   C0]  kernel_init+0x0/0x1ac
[   C0]  start_kernel+0x0/0x4dc
[   C0]  start_kernel+0x3c4/0x4dc
[   C0]  __primary_switched+0xc8/0x7cb4
```

2.3.2.2 系统中断类问题分析

一般 CPU 软中断的优先级最高，一直高频率触发软中断同样会导致系统无法正常调度到其它 task，导致出现 rct stall。

2.3.2.3 系统中断类问题解决

- 步骤 1

先确认系统中断的产生的数目以及中断处理情况，在 kernel 社区搜索如下的 patch 名称并合入

rcu: Add RCU stall diagnosis information

- 步骤 2

尝试复现该问题，并观察复现问题后，系统 log 的打印

如果 hardirqs number 远远大于 softirqs number，说明该 CPU 出现了异常的 hardirqs 被举起，softirqs 来不及处理如此多的 hardirqs 。这里对下面 log 的理解，需要对中断上半部、下半部、cpu 运行时间有一定的理解。

```
[   C2] rcu: INFO: rcu_preempt detected stalls on CPUs/tasks:
[   C2] rcu:    0-....: (1 GPs behind) idle=f81/0/0x7 softirq=6800/6801 fqs=1496014 last_accelerate: 4c8a/bb07
        dyntick_enabled: 1
[   C2] rcu:          hardirqs   softirqs   csw/system
[   C2] rcu:    number: 1192758890       0         0
[   C2] rcu:    cputime: 10188424    7475368         0   ==> 17663780(ms)
```

- 步骤 3

此时就需要寻找凶手，由于系统 hardirq 数目和类型众多，无法直接捞出具体的凶手，所以可以借助 DS-5 在 kernel 中断服务程序 handle_irq_event_percpu 入口位置打断点，这里可以在 DS5 上显示出中断执行函数的名称。

- 步骤 4

记录下来 rcu stall 以后还在进入 handle_irq_event_percpu 的中断函数命名，并在这些中断函数命名对应中断驱动的 HW irq 触发位置增加中断计数并再次复现问题。再次复现以后，中断计数异常多的驱动就是问题凶手。

- 步骤 5

由驱动 owner 去排查异常中断被频繁举起原因。

2.4 系统异常卡死

问题现象

系统运行过程中界面卡死无法操作且串口、adb 等调试工具无法使用，从 log 中没有有用的错误信息，此类问题归结为系统异常卡死。

问题分析

此类问题比较复杂，可能是：

- 供电异常
- 总线异常
- 多核异常

- 系统 crash
- 系统 block

问题解决

- 步骤 1 无任何异常 log 的问题，可以先查一下挂死时各路电是否正常；如果系统存在异构核如 RV，可以尝试先将异构核关闭，确认是否异构核引起问题。
- 步骤 2 可以先查一下如果有 ds-5 的话，使用 ds-5 调试。

1）使用 csat 访问 dram 地址 0x40000000。

2）使用 csat 尝试去访问过 IO 寄存器区域（即各模块的寄存器空间，特别是 DE/VE/GPU）及 dram 空间，若访问出错，则确定是总线挂死问题，需要检查一下相关模块操作总线的行为是否正确。

3）使用 csat 访问寄存器地址，查看 cpu 频率。以 a50 为例，即访问 0x03001000。

4）使用 csat 访问 pc。以 a50 为例，即访问 0x09410084。

表 2-2: 各个平台的 PC 指针

1689 &1718	1667&1680&1701	1719 & 1728	1721 & 1755	1708	1673
0x014100A0	0x014100A0	0x094100A0	0x09410084	C0:	C0:
0x015100A0	0x015100A0	0x095100A0	0x09412084	0x09410084	0x01530084
0x016100A0	0x016100A0	0x096100A0	0x09414084	0x09412084	0x01532084
0x017100A0	0x017100A0	0x097100A0	0x09416084	0x09414084	0x01534084
				C1:	0x01536084
				0x09c10084	C1:
				0x09c12084	0x01630084
				0x09c14084	0x01632084
					0x01634084
					0x01636084

5）如上述四步实验都是正常访问，即可使用 ds-5 软件直接调试，查看各个 cpu 堆栈及状态。可以使用 ds-5 先看一下异常现场时 cpu 的状态。

6）若连接上 ds-5 根据现象怀疑系统异常卡死是由于多核异常导致的（即 IC 某个核自身异常），可以屏蔽被怀疑的异常核再测试（直接在 dtsi 把该核去掉）。

- 步骤 3 可能是随机性 crash，按随机性 crash 的方法排查。
- 步骤 4 可能是系统 blcok 问题，按系统 block 类的方法排查。

3 系统内存类

3.1 启动内存异常类

3.1.1 dram 初始化失败

问题现象

dram 问题导致的失败，有如下类似 log：

```
try to init dram
DRAM DRIVE INFO: V1.40
DRAM normal_mode value: 00000001
rsb_send_initseq: rsb clk 400Khz -> 3Mhz
PMU: AXP81X
ddr voltage = 1500 mv
ID CHECK VERSION: V0.3
ic use default id
using axp AXP818
init dram fail
```

问题分析与解决

很明显的表示 dram 初始化失败。

初始化失败原因可以按照下列说明进行排查和解决。

- dram 焊接问题导致无法初始化：检查焊接情况。
- 可能是物料不支持导致：查看支持列表。
- dram 配置参数问题导致无法初始化：检查参数，可以尝试修改 sys_config 中的 dram_clk 参数降低 dram 频率到 360M。

3.1.2 内存识别错误

问题现象

内存识别错误导致的失败，有如下类似 log：

```
try to setup mmu
mmu setup ok
try to init dram
DRAM DRIVE INFO: kst
NOT GATE MODE
READ DQS LCDL = 00232323
DRAM Type = 3 (2:DDR2,3:DDR3,6:LPDDR2,7:LPDDR3)
DRAM CLK = 576 MHz
DRAM zq value: 003b3bfb
DRAM dram para1: 10e40200
DRAM dram para2: 00000000
DRAM workmode1: 000009f4
DRAM SIZE = 128 M
odt delay
init dram ok, size=128M
```

问题分析与解决

Log 信息表示 dram 初始化成功，但是经常莫名挂死或者死机。

经常莫名挂死或者死机的原因按照下列说明进行排查和解决。

- dram 大小是否与实际相符
- dram 是否焊接正常
- dram 参数是否正确

3.1.3 axp 识别错误

问题现象

axp 识别错误导致 dram 初始化失败，有如下类似 log：

```
axp22x or axp209 read error
[ERROR DEBUG]: POWER SETTING ERROR!
initializing SDRAM Fail.
```

问题分析与解决

Log 信息表示 axp 初始化失败导致 dram 无法初始化，初始化失败原因按照下列说明进行排查和解决。

- axp 驱动是否已经支持
- 是否 I2C 通信或者是 RSB 通信失败导致
- AXP 焊接是否正常

3.1.4 初始化莫名死机

问题现象

Dram 初始化过程中莫名死机，log 如下：

```
DRAM DRIVE INFO: V1.40
DRAM normal_mode value: 00000001
rsb_send_initseq: rsb clk 400Khz -> 3Mhz
PMU: AXP81X
ddr voltage = 1500 mv
ID CHECK VERSION: V0.3
using ic H8
using axp AXP818
DRAM Type =3 (2:DDR2,3:DDR3,6:LPDDR2,7:LPDDR3)
DRAM CLK =672 MHZ
DRAM zq value: 00003bfb  //dram初始化过程中卡死
```

问题分析

Log 显示，在 dram 初始化过程中卡死，初始化失败原因按照下列说明进行排查。

- 焊接问题，一般是焊接问题导致
- 检查一下支持列表，看是否已经支持该款 dram
- Dram sys_config.fex 参数是否配置正确，可以尝试修改 sys_config 中的 dram_clk 参数降低 dram 频率到 360M。

3.1.5 chipid 不支持

问题现象

chip id 不支持导致，报错 log 如下：

```
beign to init dram
DRAM DRIVE INFO: V1.50
DRAM normal_mode value: 00000001
rsb_send_initseq: rsb clk 400Khz -> 3Mhz
PMU: AXP81X
ddr voltage = 1200 mv
ID CHECK VERSION: V0.3
ic check fail,IC must=H8,but id=0x00000021
start ddr type auto scan mode .
rsb_send_initseq: rsb clk 400Khz -> 3Mhz
PMU: AXP81X
ddr voltage = 1500 mv
init dram fail
```

问题分析

Log 显示 IC 检测失败，导致后续的 dram 初始化失败，初始化失败原因按照下列说明进行排查。

- 检查 SDK 是否支持该颗 IC。一般出现这种情况是不支持该颗 IC，前提是多颗 IC 都无法检测通过。

- 确定 IC 的 ID 烧码是否正常，是否为误放的 IC。

3.2 Linux 内存异常类

3.2.1 Lowmemkiller/Lmkd 类问题

3.2.1.1 关键进程被杀

问题现象

杀死关键进程导致系统异常，可能表现为 android 重启或者某个应用异常退出。

```
[10411.565136] lowmemorykiller: Killing 'init' (1189) (tgid 1189), adj 0,
[10411.565136]   to free 1596kB on behalf of 'memtester' (4130) because
[10411.565136]   cache -1632728kB is below limit 32768kB for oom_score_adj 0
[10411.565136]   Free memory is -20512kB above reserved
```

问题分析

通过搜索 “lowmemorykiller”，可以查找系统触发的 lowmemkiller 的情况，log 会打印出被杀死的进程名和 pid，进而判断是否是因为 lowmemkiller 杀死了系统重要的服务导致异常。

问题解决

可通过调整进程的 adj 进而保护系统的进程在某些情况下被保护：

- 调整相应进程的 oom_score_adj，保护进程不被杀死。

```
/proc/<pid>/oom_score_adj [-1000, 1000]  // -1000，表示该进程不会被杀死
```

- 调整 lowmemkiller parameters
  lowmemkiller 是通过 adj 表和 minfree 表来判断要杀死 adj 等级的进程。

```
/sys/module/lowmemorykiller/parameters/adj
/sys/module/lowmemorykiller/parameters/minfree
```

以 H6 Linux-3.10 为例子：当系统剩余内存低于 226M 时，lowmemkiller 可以选择杀死 oom_score_adj > 906 的进程，以此类推。

```
adj:  0,    100,    200,    300,    900,    906
minfree: 18432(73M), 27648(110M), 36864(147M), 46080(184M), 51200(204M), 56320(226M)
```

3.2.1.2 lowmemorykiller 触发不及时

问题现象

owmemkiller 触发次数太少导致系统内存紧张。该问题时会导致系统因为内存紧张而性能下降问题，例如卡顿等，更严重会导致系统出现 OOM

问题分析

lowmemkiller 是通过 adj 表和 minfree 表来判断要杀死 adj 等级的进程。如果 pagecache 占用超过了 minfree 的档位，系统会放弃杀死该档位的进程，此时则要进行回收 pagecache 相关的手段。

3.2.1.3 LMKD

问题现象

LMKD 思想和 lowmemkiller 类似，只是将策略移植到了 android。出问题时可以在 logcat 中搜索“lmkd”, 排查手法和 lowmemkiller 一样。

3.2.2 Out of memory

问题现象

系统出现 Out of memory，出现死机或者延后死机。

```
[   58.778949] fsck_msdos invoked oom-killer: gfp_mask=0x200da, order=0, oom_score_adj=-1000
[   58.788267] CPU: 1 PID: 5677 Comm: fsck_msdos Tainted: G         O 3.10.65 #1
[   58.796711] Call trace:
[   58.799415] [<ffffffc00008879c>] dump_backtrace+0x0/0x11c
[   58.805569] [<ffffffc000088a3c>] show_stack+0x20/0x30
[   58.811228] [<ffffffc00082159c>] dump_stack+0x1c/0x28
[   58.816783] [<ffffffc00081ef20>] dump_header.isra.12+0x88/0x1b0
[   58.823516] [<ffffffc00015af6c>] out_of_memory+0x244/0x2a4
[   58.829548] [<ffffffc00015f188>] __alloc_pages_nodemask+0x5a0/0x7a4
[   58.836678] [<ffffffc00017b470>] handle_pte_fault+0x164/0x7e4
[   58.843112] [<ffffffc00017cb08>] handle_mm_fault+0x188/0x23c
[   58.849339] [<ffffffc000097474>] do_page_fault+0x118/0x2e0
[   58.855593] [<ffffffc000081234>] do_mem_abort+0x4c/0xac
[   58.861449] Exception stack(0xffffffc012fd3e30 to 0xffffffc012fd3f50)
[   58.868530] 3e20:                                   00000000 00000000 00000000 00000000
[   58.877805] 3e40: ffffffff ffffffff aabe6a64 00000000 12fd3ed0 ffffffc0 00004010 ffffff80
[   58.886970] 3e60: 12fd3ea0 ffffffc0 0008142c ffffffc0 00ec3d80 ffffffc0 00004010 ffffff80
[   58.896121] 3e80: 12fd3ed0 ffffffc0 0000400c ffffff80 00000001 00000000 0000400c ffffff80
[   58.905270] 3ea0: 00000000 00000000 000841d0 ffffffc0 00000000 00000000 00000000 00000000
[   58.914412] 3ec0: 00000000 00000000 00084300 ffffffc0 00000000 00000000 2b158000 00000000
[   58.923564] 3ee0: 02b15800 00000000 00000000 00000000 00000000 00000000 0f000000 00000000
```

```
[  58.932733] 3f00: ffafe454 00000000 02b15800 00000000 f1fd6000 00000000 0000ffff 00000000
[  58.941892] 3f20: 0fffffff 00000000 6ab80000 00000000 00000000 00000000 ffafe3e8 00000000
[  58.951041] 3f40: aabe6a75 00000000 00000000 00000000
[  58.956592] Mem-Info:
[  58.959082] DMA per-cpu:
[  58.962046] CPU    0: hi:  186, btch:  31 usd:  23
[  58.967315] CPU    1: hi:  186, btch:  31 usd:   0
[  58.972762] CPU    2: hi:  186, btch:  31 usd:  17
[  58.978022] CPU    3: hi:  186, btch:  31 usd:  90
[  58.983485] active_anon:105575 inactive_anon:105595 isolated_anon:24
[  58.983485]  active_file:46 inactive_file:117 isolated_file:0
[  58.983485]  unevictable:2117 dirty:0 writeback:3 unstable:0
[  58.983485]  free:1902 slab_reclaimable:2372 slab_unreclaimable:4563
[  58.983485]  mapped:464 shmem:73 pagetables:826 bounce:0
[  58.983485]  free_cma:132
[  59.019299] DMA free:7404kB min:6644kB low:86148kB high:87808kB active_anon:422212kB inactive_anon:422380kB
      active_file:312kB inactive_file:296kB unevictable:8468kB isolated(anon):0kB isolated(file):0kB present:1048576kB
      managed:931860kB mlocked:256kB dirty:0kB writeback:8kB mapped:1900kB shmem:292kB slab_reclaimable:9444
      kB slab_unreclaimable:18212kB kernel_stack:3024kB pagetables:3292kB unstable:0kB bounce:0kB free_cma:592
      kB writeback_tmp:0kB pages_scanned:297 all_unreclaimable? no
[  59.067132] lowmem_reserve[]: 0 0 0
[  59.071040] DMA: 409*4kB (UEMC) 78*8kB (UEMC) 17*16kB (UMC) 2*32kB (M) 16*64kB (M) 3*128kB (M) 0*256kB 0*512
      kB 0*1024kB 0*2048kB 1*4096kB (R) = 8100kB
/*链表类型：U－MIGRATE_UNMOVABLE
          M－MIGRATE_MOVABLE
          E －MIGRATE_RECLAIMABLE
        H －MIGRATE_HIGHATOMIC
        I －MIGRATE_ISOLATE
[  59.086124] 2509 total pagecache pages
[  59.090419] 219 pages in swap cache
[  59.094252] Swap cache stats: add 82108, delete 81889, find 14958/17373
[  59.101715] Free swap  = 0kB
[  59.104877] Total swap = 131068kB
[  59.120462] 262144 pages RAM
[  59.123629] 11819 pages reserved
[  59.127170] 838 pages shared
[  59.130517] 231846 pages non-shared
[  59.134348] [ pid ]   uid  tgid  total_vm      rss   nr_ptes  swapents  oom_score_adj name
/* 各个进程占用内存情况：
        PID   UID  进程地址空间的大小 使用物理内存的大小(页面数)
[  59.143190] [ 1248]    0  1248     789     190     3     21       -1000 ueventd
[  59.152243] [ 1724]  1036  1724    2755     138     8     228      -1000 logd
[  59.160992] [ 1732]    0  1732    1336       0     5     101      -1000 debuggerd
......
[  59.390402] [ 5677]    0  5677  328529  209207    469   26897      -1000 fsck_msdos
[  59.399622] [ 6394]  1003  6394    4245     187     10     3       -1000 bootanimation
[  59.409398] [ 6457]    0  6457     306      28     4      0       -1000 app_process
```

问题分析

Linux 内存不足的时候会触发 Out of memory（简称 OOM），通过 kill process 来缓解内存的不足，该机制可能会杀死某些应用导致系统异常，通过搜索 “Out of memory” 可以搜索到是否触发了 OOM。当系统发生 OOM 时，则表示系统的内存在某段时间已经 “触底” 了。

问题解决

可采取如下排查手段。

- 步骤 1-检查是否为某个进程因为内存泄漏等原因占用内存过多，导致系统内存异常？

OOM 会将异常时各个进程占用内存的情况打出来，如上面 log 标红之处，包括进程名、pid、uid、虚拟内存大小、物理内存大小和 oom_score_adj 等。该问题我关注物理内存占用大小：[第四个数字]/256 MBytes。如上面 log：fsck_msdos 的进程占用了 209207/256M 内存，导致了系统内存异常。

```
[  59.390402] [ 5677]    0 5677  328529  209207    469   26897      -1000 fsck_msdos
```

- 步骤 2-检查是否为不可杀进程太多导致内存占用过多？

如果没有明显内存占用异常的进程，查看一下 oom_score_adj 为负值的进程。该值越小越不容易被系统杀死，包括 OOM 和 lowmemkiller，值为-1000 则不会被杀死。一旦该类进程太多，则会导致系统紧张甚至出现 OOM。proc 调整进程 adj 接口：

```
·/proc/<pid>/oom_score：表示该进程的oom分数，分数越高越容易被OOM杀死，为0表示不会被kill
·/proc/<pid>/oom_score_adj：points += adj *(totalpages/1000)，并且当等于-1000时，OOM略过该进程
```

- 步骤 3-检查是否是内存预留出错，例如 CMA 太小导致 OOM？

分析是何种 page 的紧张导致 OOM，判断系统该种页面的预留是否正常？通过分析 gfp_mask 和 rder 参数。

```
gfp_mask：表示需要的内存类型；
order:表示需要的内存页面数，为2^order个页面。
```

可以分析 fsck_msdos 进程需要 (2^0 = 1) 个 movable dma 的页面申请不到导致问题出现。

```
[  58.778949] fsck_msdos invoked oom-killer: gfp_mask=0x200da, order=0, oom_score_adj=-1000
```

然后通过 OOM 的 log 可判断该类页面的剩余，然后调整内存分布。

- 步骤 4-检查是否是水位线设置问题导致 OOM 触发多余频繁？

适当降低 min watermask，降低 OOM 的频率，proc 接口如下：

```
cat /proc/sys/vm/min_free_kbytes
```

3.2.3 CMA 类问题

CMA，连续内存管理器。项目中出现的 CMA 内存问题各式各样，这里列举几种常见的问题，并给出分析的思路和方法。

3.2.3.1 CMA 内存失败

问题现象

分配 CMA 内存失败，导致播放视频失败或者 OOM。

问题分析

可以从下面几个方向去排查：

- 系统预留 CMA 是否充裕？通过/proc/meminfo 可以查询系统 CMA 的总量是否满足最大场景下 cma 内存需求。

```
CmaTotal:      65536 kB
```

- CMA 使用是否已经用尽？各个模块使用情况以 linux-4.4 为例子，通过 cat /sys/kernel/debug/ion/heaps/cma。
- CMA 或者出现泄漏？通过/sys/kernel/debug/ion/heaps/cma，可以看到 cma 是否出现泄漏以及泄漏具体的进程 PID。具体的分析手段和案例可以参照一号通《Android 系统内存泄漏排查指导文档.pdf》第 3.2.2 章节。

3.2.4 内存泄漏类

系统剩余内存充足，但出现大块内存申请失败，导致视频播放失败甚至 OOM 卡死。在开发过程中因为编码的疏漏, 未对申请内存进行释放，就会出现内存泄露。内存泄露在开发测试中是很难被发现的，高强度的老化测试下才会把这些隐藏的 bug 暴露出来。

问题现象

有如下现象之一就可以怀疑是系统内存泄露问题:

- 长时间老化出现的系统卡顿
- 内存不足触发大量的 lowmemorykiller 或者 oom
- 内存分配失败

问题分析

内核内存泄漏主要有以下几种：

- slab 内存泄漏

- ion 内存泄漏
- 私有页内存泄漏

问题解决

- Ion 内存泄漏?

具体参照一号通《Android 系统内存泄漏排查指导文档.pdf》。

- Slab 内存泄漏?

可通过打开内存节点，进行问题定位。

```
CONFIG_DEBUG_SLAB=y
CONFIG_DEBUG_SLAB_LEAK=y
CONFIG_SLABINFO=y
```

另外通过 cat /proc/slabinfo 查看 slab 具体信息，具体的排查手段可以参考一号通《Android 系统内存泄漏排查指导文档.pdf》。

- 其他页面内存泄漏?

使用 cpu_monitor 进行内存监控，抓出具体泄漏的地方，具体参照一号通《Android 系统内存泄漏排查指导文档.pdf》。

3.2.5 内存分配失败

出现如下 log

```
ksdioirqd/mmc1 invoked oom-killer: gfp_mask=0xc2cc0(GFP_KERNEL|GFP_NOWARN|GFP_COMP|GFP_NOMEMALLOC),
        order=3, oom_score_adj=0
 CPU: 7 PID: 811 Comm: ksdioirqd/mmc1 Tainted: G        OE     5.15.137-homlet-h728-android14-v0.9 #1
 Hardware name: sun55iw3 (DT)
 Call trace:
  dump_backtrace.cfi_jt+0x0/0x8
  show_stack+0x20/0x30
  dump_stack_lvl+0x6c/0x9c
  dump_header+0x58/0x25c
  oom_kill_process+0xb4/0x270
  out_of_memory+0x1e4/0x374
  __alloc_pages_slowpath+0xba4/0x120c
  __alloc_pages+0x18c/0x2dc
  kmalloc_order+0x54/0x1ac
  kmalloc_order_trace+0x38/0x158
  __kmalloc_track_caller+0x2b8/0x3ec
  __alloc_skb+0x138/0x2b8
```

```
__netdev_alloc_skb+0x158/0x214
aicwf_sdio_readframes+0x48/0xc0 [aic8800_fdrv]
aicwf_sdio_hal_irqhandler+0x3bc/0x510 [aic8800_fdrv]
process_sdio_pending_irqs+0x88/0x238
sdio_irq_thread+0x94/0x1d0
kthread+0x174/0x1e0
ret_from_fork+0x10/0x20
```

提示 order=3，GFP_KERNEL|GFP_NOWARN|GFP_COMP|GFP_NOMEMALLOC 属性 page 无法申请，就表示出现了内存分配失败问题。

出现上述的 log 后，紧接着会打印下面的重要信息

- Mem-Info 信息，这里面包括了各种内存的使用情况，检查是否有部分特定内存占用过大的情况
- ZONE 的信息，这里面可以看到不同类型的 memory zone 不同 page size 的剩余数量
- Tasks state (memory values in pages) 这里面可以看到所有 task 使用虚拟内存和 RSS 的情况，这里面还可以关注 oom_score_adj 信息，数值越大，表示占用的内存越多，其中-1000 表示数值很低，基本不会被 kill
- oom-kill 和 Out of memory 会打印出因为占用内存过高，在 OOM 发生的时候被 kill 的 task 信息

此类问题绝大部分场景都是因为内存泄漏或者某个 driver 或者 task 占用过多物理内存引起，此时可以按照对 log 的分析，来分辨是哪一种 case.

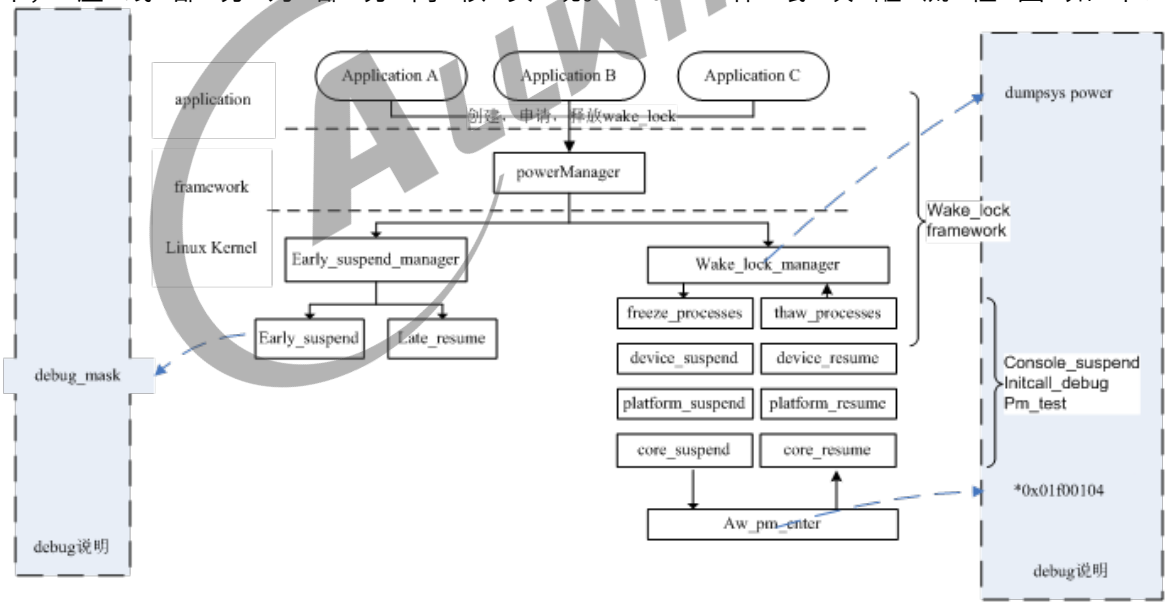
4 休眠唤醒类

4.1 休眠流程概述

4.1.1 Linux

休眠唤醒指系统进入低功耗和退出低功耗模式，一般称之为 Standby。休眠过程由应用发起，经由内核的电源管理框架来进行休眠唤醒管理工作。

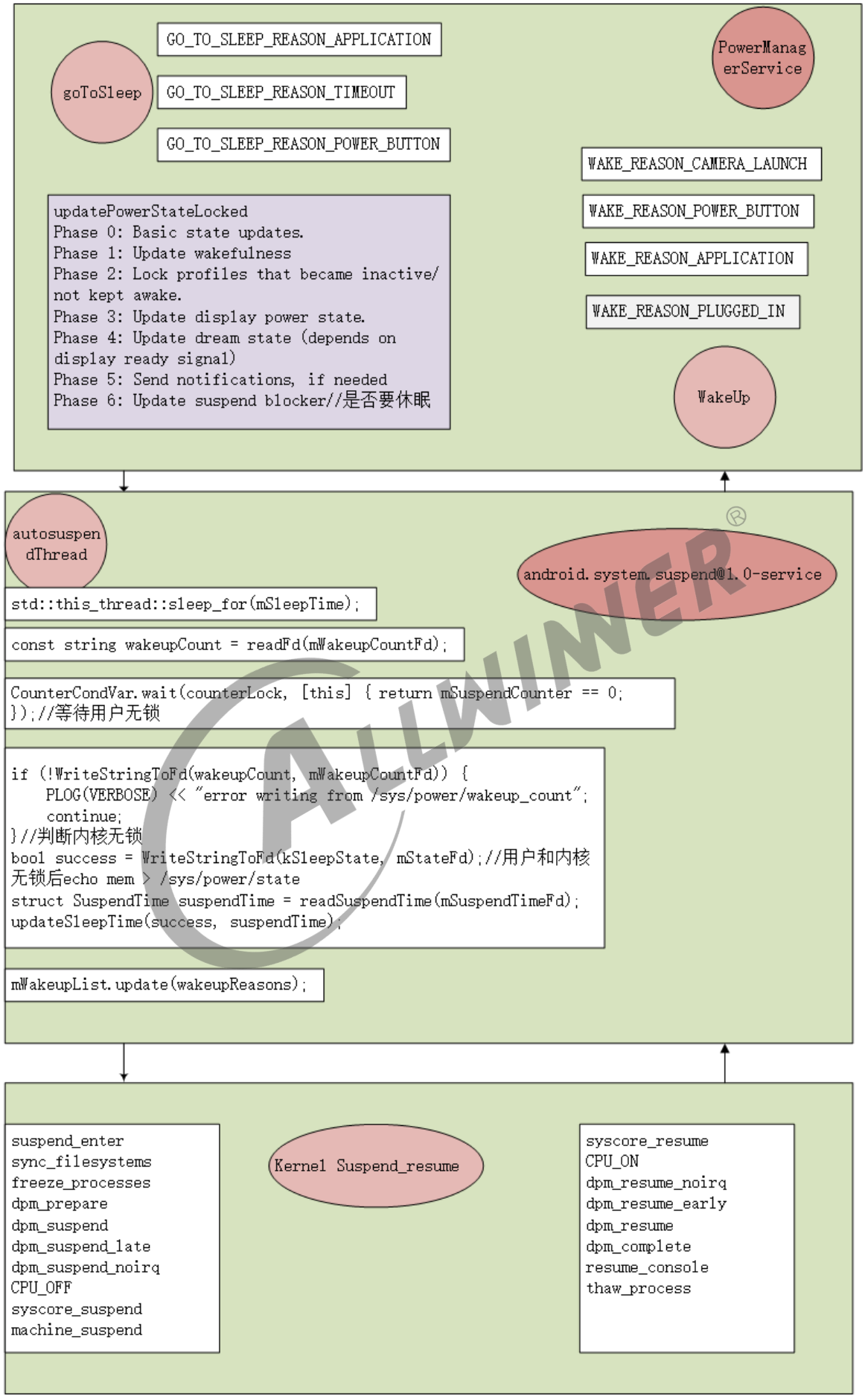
1. 如果存在 CPUS（一颗集成在 IC 内部的对电源进行管理的 openrisc 核，是 SOC 内置的超低功耗硬件管理模块），最终会传递到到 CPUS。因此休眠唤醒类出现问题的可能为应用层、内核层、CPUS 层。
2. 如果不存在 CPUS，则 CPU 进入 WFI。休眠唤醒流程图如下，虚线部分为部分内核实现。Linux 休眠唤醒流程图如下：



4.1.2 Android

Android 固件的情况下，休眠流程由 Android 层发起。Android 层有一个 auto sleep 的线程，检查 Android 和 linux 下是否有持锁。当 Android 和 linux 的锁都释放时，auto sleep 线程会再等待 500ms，如果 500ms 后依旧还未有线程或其他驱动拿到阻止休眠的

锁，则 Android 层就会下发休眠指令，触发 linux 的休眠。Android 休眠唤醒流程图如下：



4.2 常见问题现象

- 系统休眠异常：
  - 无法休眠
  - 休眠速度慢
- 系统唤醒异常：
  - 被错误唤醒
  - 无法被唤醒
  - 唤醒速度慢
  - 唤醒后异常
- 休眠状态异常
  - 供电状态异常
  - 休眠功耗异常
- 休眠唤醒的过程中挂掉/卡死

技巧

该类问题涉及的常见的调试节点和通用调试方法比较多，因此，将它们整理成两个单独的小节放在了最后。

4.3 通用排查方法

休眠唤醒一般分为两个步骤：1. Android 等待锁清空并下发休眠指令 2. Linux 完成整个休眠唤醒流程

流程标志内容	标志性打印信息
进入 Linux 休眠流程	PM: suspend entry (deep)
Linux 休眠流程完成	printk: Suspending console(s) (use no_console_suspend to debug)
Linux 唤醒完成	PM: suspend exit

例：Linux休眠的打印信息

```
[ 347.756120][ T4630] PM: suspend entry (deep)
[ 347.794825][ T4630] Filesystems sync: 0.033 seconds
[ 347.800335][ T4630] Freezing user space processes ... (elapsed 0.002 seconds) done.
[ 347.811009][ T4630] OOM killer disabled.
[ 347.815431][ T4630] Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
[ 347.826136][ T4630] printk: Suspending console(s) (use no_console_suspend to debug)
```

因此首先需要先快速区分休眠唤醒问题是发生在 Android 阶段还是 Linux 阶段可按照下图进行快速

排查

图 4-1

4.3.1 问题分类

首先需要对问题进行分类。（如果已经确认问题类别，可以直接跳到本小节的最后一步）

- 步骤 0：

无法休眠、休眠时间长的情况下，检查休眠唤醒 log，查看问题发生时是否有PM: suspend entry (deep)等 Linux 休眠唤醒标志，如果没有则为 Android 上层的问题。

- 步骤 1：

打开休眠唤醒调试信息、打开 DPM_WATCHDOG 配置。在内核配置中打开 DPM_WATCHDOG 配置:

说明

具体方法查看倒数最后一节的通用调试手段。

- 步骤 2：

触发 kernel 进入休眠或触发 android 进入休眠。触发 kernel 休眠：（在 android 环境下，不建议使用）

```
echo mem > /sys/power/state
```

触发 Android 休眠：

```
input keyevent 26
```

- 步骤 3：

问题分类的具体过程如下：

- 查看系统休眠过程的打印信息。

1. 如果打印信息没有完整打印到【sunxi scp】的位置，说明系统休眠异常，直接跳到本小节的最后一步。
2. 如果打印信息能完整打印到【sunxi scp】的位置，但确认休眠后系统供电没有关闭，或获取休眠唤醒 RTC 状态码不是 CPUS 等待唤醒的状态，说明系统休眠异常，直接跳到本小节的最后一步。
3. 如果确认休眠后系统供电已经关闭，说明系统休眠正常。

```
echo mem > /sys/power/state
====================【freezer】====================
[  63.231459] PM: suspend entry 1970-01-01 06:03:15.956997236 UTC
[  63.239639] PM: Syncing filesystems ... done.
[  63.251156] PM: Preparing system for sleep (mem)
[  63.259123] Freezing user space processes ... (elapsed 0.001 seconds) done.
[  63.267998] Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
[  63.277370] PM: Suspending system (mem)

====================【devices】====================
[  63.282317] calling  sunxi_cpuidle.0+ @ 860, parent: platform
[  63.282321] calling  mmc0:0001+ @ 28, parent: mmc0
[  63.294005] call sunxi_cpuidle.0+ returned 0 after 0 usecs
[  63.300122] calling  sunxi_ths_combine_1+ @ 860, parent: platform
......
[  64.902372] [ohci1-controller]: sunxi_ohci_hcd_suspend
[  64.908092] call 1c19000.ohci1-controller+ returned 0 after 5586 usecs
[  64.915379] calling  soc+ @ 860, parent: platform
[  64.920606] call soc+ returned 0 after 0 usecs
[  64.925534] calling  reg-dummy+ @ 860, parent: platform
[  64.931332] call reg-dummy+ returned 0 after 0 usecs
[  64.936870] PM: suspend of devices complete after 1654.643 msecs
[  64.944666] PM: late suspend of devices complete after 1.126 msecs
[  65.034365] PM: noirq suspend of devices complete after 82.830 msecs
```

```
====================【processors】====================
[  65.041414] Disabling non-boot CPUs ...
[  65.046961] CPU1: shutdown
[  65.050107] CPU1 killed.
[  65.054262] CPU2: shutdown
[  65.057317] CPU2 killed.
[  65.061389] CPU3: shutdown
[  65.064447] CPU3 killed.

====================【core】====================
[  65.067913] PM: Calling sched_clock_suspend+0x0/0x30
[  65.071212] PM: Calling timekeeping_suspend+0x0/0x258
[  65.071212] PM: Calling irq_gc_suspend+0x0/0x70
[  65.071212] PM: Calling fw_suspend+0x0/0x14
[  65.071212] PM: Calling cpu_pm_suspend+0x0/0x18

====================【sunxi scp】====================
```

- 查看系统休眠中的状态。

1. 如果确认休眠后系统供电、休眠后 dram 供电、休眠后非系统供电不符合预期设计，说明系统休眠中状态异常，直接跳到本小节的最后一步。
2. 如果确认上述情况符合预期设计，说明系统休眠中状态正常。

- 查看唤醒过程的打印信息。

1. 如果打印信息没有完整打印到【suspend exit】的位置，说明系统唤醒异常，直接跳到本小节的最后一步。
2. 如果打印信息能完整打印到【suspend exit】的位置，但控制台不能使用或模块功能异常或其他系统异常，说明系统唤醒异常，直接跳到本小节的最后一步。

```
====================【core】====================
[  65.071212] platform wakeup, standby wakesource is:0x0
[  65.071212] PM: Calling cpu_pm_resume+0x0/0x10
[  65.071212] PM: Calling irq_gc_resume+0x0/0x6c
[  65.071212] PM: Calling irq_pm_syscore_resume+0x0/0x8
[  65.071212] PM: Calling timekeeping_resume+0x0/0x268
[  65.071212] PM: Calling sched_clock_resume+0x0/0x4c

====================【processors】====================
[  65.084619] Enabling non-boot CPUs ...
[  65.090828] CPU1 is up
[  65.095533] CPU2 is up
[  65.101004] CPU3 is up

====================【devices】====================
[  65.104145] calling  pio+ @ 860, parent: soc
[  65.108887] call pio+ returned 0 after 14 usecs
[  65.114312] calling  twi0+ @ 860, parent: soc
[  65.119139] call twi0+ returned 0 after 4 usecs
[  65.124176] calling  twi1+ @ 860, parent: soc
[  65.129035] call twi1+ returned 0 after 36 usecs
[  65.135031] PM: noirq resume of devices complete after 31.294 msecs
```

```
[  65.142786] PM: early resume of devices complete after 0.712 msecs
......
[  65.221071] calling  sunxi_cpuidle.0+ @ 860, parent: platform
[  65.221075] call sunxi_cpuidle.0+ returned 0 after 0 usecs
[  65.254065] usb usb2: root hub lost power or was reset
[  65.254086] call 1c19000.ohci1-controller+ returned 0 after 65763 usecs
[  65.254111] calling  usb2+ @ 6, parent: 1c19000.ohci1-controller
[  65.300196] call usb1+ returned 0 after 125628 usecs
[  65.400217] disp_resume finish
[  65.400221] call disp+ returned 0 after 208402 usecs
[  65.530170] call usb2+ returned 0 after 269582 usecs
[  66.640800] [ohci0-controller]: is disable, can not resume
[  66.640805] call 1c14000.ohci0-controller+ returned 0 after 6 usecs
[  66.640844] PM: resume of devices complete after 1491.206 msecs

====================【freezer】====================
[  66.641511] PM: Finishing wakeup.
[  66.642902] Restarting tasks ... done.
[  66.668369] PM: suspend exit 1970-01-01 23:12:06.453640709 UTC
[  66.641511] PM: Finishing wakeup.
[  66.642902] Restarting tasks ... done.
[  66.668369] PM: suspend exit 1970-01-01 23:12:06.453640709 UTC

====================【suspend exit】====================
```

- 步骤 4：

根据问题分类的结果，进行相应的处理：系统休眠异常、系统唤醒异常、系统休眠中状态异常。

4.3.2 系统休眠异常

4.3.2.1 系统无法休眠

- 休眠流程发起异常

1. logcat -s PowerManagerService 查看是否有正常发起 Going to Sleep。

```
PowerManagerService: Powering off display group due to power_button (groupId= 0, uid= 1000)...
PowerManagerService: Going to sleep due to power_button (uid 1000)...
PowerManagerService: Dozing...
```

2. dumpsys power 与正常机器对比 Power Manager State 查看是否有异常。
3. getevent -l 查看按键是否有正常上报。

dumpsys input | grep -i keyevent 查看按键是否有上报。

4. 阅读源码查看 PhoneWindowManager 是否接收了 POWER_KEY 但没有处理。

- 阻止休眠（可能是 wake_lock、irq 频繁打断、某个模块返回 fail 等阻止休眠。）

1. 如果系统休眠未完成就返回，且控制台可以使用：查看 **kernel wake_lock** 状态、查看 **android wake_lock** 状态、查看 **wakeup_source** 状态，确认有模块持有 **wake_lock** 阻止休眠。

   参考系统无法休眠案例或按照模块休眠唤醒异常处理。

2. 如果打印信息在【devices】和【processors】之间的位置，有明显的模块休眠错误信息，且尝试卸载此模块后休眠就正常了，说明是此模块的休眠流程存在异常。

   按照模块休眠唤醒异常处理。

3. 如果系统休眠未完成就返回，且无明显的错误信息：查看系统中断信息，发现有模块中断状态明显异常，如中断次数不合理地过多等等，且尝试卸载此模块后休眠就正常了，说明是此模块的休眠流程存在异常。

   按照模块休眠唤醒异常处理。

4. 否则，按照非模块休眠唤醒异常处理。

- 休眠过程挂掉

1. 如果打印信息卡在【devices】和【processors】之间的位置，且多试几次都是稳定卡在某个模块的休眠过程、dpm watchdog 触发信息显示卡在某个模块、尝试卸载此模块后休眠就正常了、查看和设置休眠唤醒测试模式为 freezer 正常而 devices 异常，说明是此模块的休眠流程存在异常。

   按照模块休眠唤醒异常处理。

2. 否则，按照非模块休眠唤醒异常处理。

4.3.2.2 系统休眠速度慢

- kernel 阶段休眠速度慢定位方法

1. 方法一：根据 log 信息查看 kernel 阶段休眠总耗时

步骤 1：

控制台执行如下命令：

```
echo 0 > /proc/sys/kernel/printk
echo Y > /sys/module/printk/parameters/console_suspend
echo "+5" >/sys/class/rtc/rtc0/wakealarm
echo mem > /sys/power/state
```

步骤 2：

休眠唤醒后，输入 dmesg，查看打印信息, 找出 “PM: suspend entry”、“Disabling non-boot CPUs/CPUX killed” 相关打印的时间戳，它们分别对应 kernel 开始休眠、结束休眠的时间点 (单位 S) kernel 休眠总耗时 = 646.011944 - 645.096897

```
[  645.096897] PM: suspend entry (deep)
[  645.106140] Filesystems sync: 0.009 seconds
……
[  646.011417] Disabling non-boot CPUs ...
[  646.011944] CPU1 killed.
```

2. 方法二：根据 log 信息找出耗时的模块

步骤 1：

控制台执行如下命令：

```
echo 8 > /proc/sys/kernel/printk
echo N > /sys/module/printk/parameters/console_suspend
echo Y > /sys/module/kernel/parameters/initcall_debug
echo 1 > /sys/power/pm_print_times
echo "+5" >/sys/class/rtc/rtc0/wakealarm
echo mem > /sys/power/state
```

步骤 2：

根据步骤 1 输出的 log 信息可以获取每个模块休眠耗时（单位 us）：如：pwm 模块休眠耗时 = 124 us；pinctrl 模块休眠耗时 = 6273 us

```
[ 1757.511553] sunxi_pwm 2000c00.pwm: calling sunxi_pwm_suspend+0x0/0xf0 @ 978, parent: soc@3000000
[ 1757.521497] sunxi_pwm 2000c00.pwm: sunxi_pwm_suspend+0x0/0xf0 returned 0 after 124 usecs
[ 1757.531151] sun8iw20-pinctrl 2000000.pinctrl: calling sun8iw20_pinctrl_suspend_noirq+0x0/0x50 @ 978, parent:
        soc@3000000
[ 1757.543293] sun8iw20-pinctrl 2000000.pinctrl: pinctrl suspend
[ 1757.549721] sun8iw20-pinctrl 2000000.pinctrl: sun8iw20_pinctrl_suspend_noirq+0x0/0x50 returned 0 after 6273 usecs
```

如发现某个模块耗时比较长，可跟模块相关负责人确认耗时是否正常。

3. 方法三：使用 IO 口输出翻转波形，用逻辑分析仪或示波器抓取波形，测量时间

找一个空闲的 GPIO 作为信号线，连接到逻辑分析仪或示波器，在唤醒的程序流程中调用 IO 翻转接口，GPIO 初始化及翻转接口参考代码如下（以 AW1859 SOC PE14 为例）：

kernel 阶段参考代码如下：

```
#define IO_CONFIG_REG           (0x020000C4)
#define IO_CONFIG_SHIFT         (24)
#define IO_CONFIG_MASK          (0xF << IO_CONFIG_SHIFT)
#define IO_SELECT_OUTPUT_CONFIG    (0x1 << IO_CONFIG_SHIFT)

#define IO_DATA_REG             (0x020000D0)
#define IO_DATA_SHIFT           (14)

static void init_io_config(void)
```

```
{
    volatile u32 value = 0x00;

    value = readl(ioremap(IO_CONFIG_REG, 4));
    value &= (~IO_CONFIG_MASK);
    value |= IO_SELECT_OUTPUT_CONFIG;
    writel(value, ioremap(IO_CONFIG_REG, 4));

    value = readl(ioremap(IO_DATA_REG, 4));
    value &= (~(0x1 << IO_DATA_SHIFT));
    writel(value, ioremap(IO_DATA_REG, 4));
}

void plat_pm_io_flip(void)
{
    volatile u32 value = 0x00;

    value = readl(ioremap(IO_DATA_REG, 4));
    value ^= (0x1 << IO_DATA_SHIFT);
    writel(value, ioremap(IO_DATA_REG, 4));
}
```

4.3.2.3 系统不预期休眠

1. 需在不能休眠的场景下加入应用锁或者驱动锁，阻止不预期的休眠
2. 检查误触发休眠流程的原因

4.3.3 系统唤醒异常

4.3.3.1 系统无法唤醒

- 唤醒源问题

1. 确认唤醒源是否正确设置，参考系统不能被唤醒案例或设置唤醒源。
2. 否则，按照非模块休眠唤醒异常处理。

- 唤醒过程挂掉

1. 如果打印信息卡在【devices】和【freezer】之间的位置，且多试几次都是稳定卡在某个模块的唤醒过程、dpm watchdog 触发信息显示卡在某个模块、尝试卸载此模块后唤醒就正常了、查看和设置休眠唤醒测试模式为 freezer 正常而 devices 异常，说明是此模块的唤醒流程存在异常，按照模块休眠唤醒异常处理。
2. 否则，按照非模块休眠唤醒异常处理。

4.3.3.2 系统异常唤醒

- 休眠后，系统非预期地被唤醒。

1. 查看唤醒源并检查设置唤醒源，确认此唤醒源是被错误设置，参考系统被错误唤醒案例或按照模块休眠唤醒异常处理。
2. 否则，按照非模块休眠唤醒异常处理。

4.3.3.3 系统唤醒速度慢

- kernel 阶段休眠速度慢定位方法

1. 方法一：根据 log 信息查看 kernel 阶段唤醒总耗时

步骤 1：

控制台执行如下命令：

```
echo 0 > /proc/sys/kernel/printk
echo Y > /sys/module/printk/parameters/console_suspend
echo "+5" >/sys/class/rtc/rtc0/wakealarm
echo mem > /sys/power/state
```

步骤 2：

休眠唤醒后，输入 dmesg，查看打印信息。找出 “Enabling non-boot CPUs …”、“PM: suspend exit” 相关打印的时间戳，它们分别对应 kernel 开始唤醒、结束唤醒的时间点 (单位 S) kernel 唤醒总耗时 = 647.091127 - 646.012385

```
[  646.012385] Enabling non-boot CPUs ...
[  646.013432] CPU1 is up
……
[  647.091052] Resume caused by IRQ 25, 7090000.rtc
[  647.091127] PM: suspend exit
```

2. 方法二：根据 log 信息找出耗时的模块

步骤 1：

控制台执行如下命令：

```
echo 8 > /proc/sys/kernel/printk
echo N > /sys/module/printk/parameters/console_suspend
echo Y > /sys/module/kernel/parameters/initcall_debug
echo 1 > /sys/power/pm_print_times
echo "+5" >/sys/class/rtc/rtc0/wakealarm
echo mem > /sys/power/state
```

步骤 2：

根据步骤 1 输出的 log 信息可以获取每个模块唤醒耗时（单位 us）：如：pwm 模块唤醒耗时 = 259 us；pinctrl 模块休眠耗时 = 6176 us。

```
[ 1757.590095] sun8iw20-pinctrl 2000000.pinctrl: calling sun8iw20_pinctrl_resume_noirq+0x0/0x58 @ 978, parent:
      soc@3000000
[ 1757.602138] sun8iw20-pinctrl 2000000.pinctrl: pinctrl resume
[ 1757.608445] sun8iw20-pinctrl 2000000.pinctrl: sun8iw20_pinctrl_resume_noirq+0x0/0x58 returned 0 after 6176 usecs
[ 1757.620564] sunxi_pwm 2000c00.pwm: calling sunxi_pwm_resume+0x0/0x14c @ 978, parent: soc@3000000
[ 1757.630645] sunxi_pwm 2000c00.pwm: sunxi_pwm_resume+0x0/0x14c returned 0 after 259 usecs
```

如发现某个模块耗时比较长，可跟模块相关负责人确认耗时是否正常。

3. 方法三：使用 IO 口输出翻转波形，用逻辑分析仪或示波器抓取波形，测量时间（同系统休眠速度慢方法三）

- android 阶段唤醒耗时定位方法

步骤 1：

休眠唤醒后, 控制台执行如下命令：

```
logcat | grep PowerManagerService
```

步骤 2：

根据步骤 2 的 log 可获取如下信息：1. 按键上报时间点

```
01-20 14:41:28.014  2060  2184 I PowerManagerService: Waking up from Asleep
```

2. 亮屏时间点

```
01-20 14:41:28.175  2060  2103 W PowerManagerService: Screen on took 459 ms
```

android 唤醒耗时 = 14:41:28.175 - 14:41:28.014

4.3.3.4 系统唤醒后异常

- 唤醒后模块异常

1. 唤醒后，出现模块功能异常，如 wifi 不能用、摄像头不能用等，按照模块休眠唤醒异常处理。
2. 否则，按照非模块休眠唤醒异常处理。

- 唤醒后系统异常

1. 唤醒后，出现控制台或系统定时器等系统模块的功能异常，按照非模块休眠唤醒异常处理。

4.3.4 系统休眠中状态异常

4.3.4.1 供电状态异常

- 系统供电状态异常

1. 如果确认休眠后系统供电、休眠后 dram 供电不符合预期设计，按照非模块休眠唤醒异常处理。

- 非系统供电状态异常

1. 如果确认休眠后非系统供电不符合预期设计，按照模块休眠唤醒异常处理。

4.3.4.2 休眠功耗异常

1. 确认休眠后系统供电、休眠后 dram 供电不符合预期设计：按照非模块休眠唤醒异常处理
2. 确认休眠后非系统供电不符合预期设计：按照模块休眠唤醒异常处理。
3. 进行系统分量功耗测量，确认是哪一路供电、哪一个模块的功耗偏大。如果是某个模块供电的功耗偏大，按照模块休眠唤醒异常处理；如果是系统供电的功耗偏大，按照非模块休眠唤醒异常处理。

4.4 通用处理流程

4.4.1 模块休眠唤醒异常

模块休眠唤醒异常，就是由模块本身引起的休眠唤醒问题。建议处理流程如下。

- 步骤 1：

查看和设置休眠唤醒测试模式为 devices，可以测试模块驱动的 suspend/resume 函数。这个阶段的问题，模块负责人一般可以自行解决。

- 步骤 2：

查看和设置休眠唤醒测试模式为 core，可以测试 kernel 休眠唤醒的大部分流程，除了系统级硬件资源（如系统供电、系统时钟、引脚等）。这个阶段的问题，模块负责人需要比较熟悉本模块的休眠唤醒的软件流程，合理正确地使用和管理相关软件资源（如锁、空指针、内存、clk、regulator 等），才能解决问题。

- 步骤 3：

查看和设置休眠唤醒测试模式为 none，可以测试休眠唤醒的完整流程。这个阶段的问题，模块负责人需要很熟悉本模块的休眠唤醒的软硬件流程，非常清楚模块本身休眠唤醒需要做什么工作、保留什么资源、关闭什么资源、恢复什么资源、对外部有什么依赖等。

常见的硬件资源有 regulator、gpio、pll、osc、模块自身的寄存器，以下举例逐一分析。

实例分析一之 regulator。某个模块在系统休眠下去时，需要保留自身供电，但实测发现该路供电被关闭了。原因可能是，该模块没有使用 regulator_get 和 regulator_enable 等接口，对自身使用的供电进行管理。

实例分析二之 gpio。某个模块需要使用 gpio，但休眠唤醒后，发现 gpio 相关功能失效了。原因可能是，该模块没有在自身 suspend/resume 函数中对 gpio 进行合适的挂起/恢复操作，而系统休眠唤醒的掉电会使原有配置丢失。

实例分析三之 pll。某个模块需要使用 pll，但休眠唤醒后，发现 pll 失效了。原因可能是，该模块没有在自身 suspend/resume 函数中对 pll 进行合适的挂起/恢复操作，而系统休眠唤醒的掉电会使原有配置丢失。

实例分析四之模块自身的寄存器。某个模块休眠唤醒后，自身功能异常了。原因可能是，该模块没有在自身 suspend/resume 函数中对模块寄存器进行合适的挂起/恢复操作，而系统休眠唤醒的掉电会使原有配置丢失。

综上，模块休眠唤醒需要遵循几个原则：

（1）如果模块在系统休眠后不需要使用或仅部分功能使用，模块休眠时应该尽可能关电、关时钟、关其他资源，甚至关闭模块自身；或者起码也要进入低功耗模式。

（2）模块唤醒时，与休眠流程相对应，应该自行恢复电、时钟、其他资源和模块寄存器，必要时甚至要重新初始化模块，而不能假定这些系统资源是可用的，模块必须自行管理这些资源。

- 步骤 4：

若上述步骤均通过，说明系统休眠唤醒的通路是正常的。对于不适用上述处理流程的问题，模块负责人一般也可以自行解决，不再赘述。

4.4.2 非模块休眠唤醒异常

非模块休眠唤醒异常，就是系统级的休眠唤醒问题。

休眠唤醒功能，涉及操作系统、应用组件、内核、模块驱动和底层硬件，属于系统级功能。休眠唤醒功能的稳定性，是产品稳定性中最关键的一项质量要求，全志对外发布 SDK 是经过了严格的全面老化测试。

当客户端拿到 SDK 经过开发定制，运行在自己的产品工程样机上，发现休眠唤醒的基本功能出现异常时（例如系统不能正常休眠或者唤醒），很大可能是工程开发过程中，软件和硬件变化等因素引起。常见问题有 dram 物料更换、电源方案修改、sys_config 修改、dts 修改、menuconfig 修改等。建议处理流程如下：

- 步骤 1：

检查工程环境的问题。确认使用的工程环境，对比 SDK 包没有非预期的或不确定的改动，保证环境是干净的。如果问题解决，则说明是工程环境引入的问题，方案开发人员可以自行解决。

- 步骤 2：

检查板级配置。对比 SDK 包默认的板级方案参考配置，sys_config 和 dts 配置是否正确无误，供电部分是否有改动、dram 参数部分是否有改动、改动是否合理。

- 步骤 3：

检查板子硬件设计。对比 SDK 包的硬件参考设计原理图，检查系统供电设计是否存在不同、上下电时序是否不同、这些不同是否合理、是否经过方案硬件设计人员审核。

- 步骤 4：

检查系统稳定性。休眠唤醒功能，是系统性功能，必须在系统稳定的基础上，才能保证休眠唤醒功能的正常。客户端样机是否通过简单的 reboot 重启测试、memtester 测试、DDR 物料测试、老化测试（如 Android 系统的捕鱼达人、在线播放视频、播放本地视频，循环录像等场景）。

- 步骤 5：

若上述步骤均未发现异常，或遇到不适用上述处理流程的问题，则需要反馈相关负责人。

4.5 常见问题定位

4.5.1 系统无法休眠

4.5.1.1 系统持锁无法休眠

说明

关键词：系统持锁

问题现象

系统持锁，suspend 失败。

问题分析

suspend 失败，可能是系统持锁阻止休眠。

问题解决

- 步骤一：安卓查看是否有持锁相关信息。

```
dumpsys power | grep PART
```

- 步骤二：内核中是否有相关持锁信息。

> cat /sys/kernel/debug/wakeup_sources

查看 active_since 项，若对应模块不为 0，则该模块一直阻止系统进入休眠，查看该模块是否异常。

> cat /sys/power/wake_lock

查看是否有安卓申请的锁。

4.5.1.2 Android 系统持锁无法休眠

说明

关键词：Android 系统持锁

问题现象

定时休眠到时后，屏幕亮屏，串口可以输入，系统无法休眠。

问题分析

系统无法休眠，确认 Android 是否支持，是否禁止定时休眠。

1. dumpsys suspend_control_internal 查看当前锁类型

```
console:/sys/power # dumpsys suspend_control_internal

 ------------------------------------------------------------------------
```

```
|
 ------------------------------------------------------------------------
| NAME                  | PID   | TYPE   | STATUS   | ACTIVE COUNT |
 ------------------------------------------------------------------------
| ApmOutput             |   389 | Native | Inactive |       226 |
| PowerManagerService.WakeLocks |   571 | Native | Active   |       637 | //应用阻止休眠锁
| PowerManagerService.Broadcasts |   571 | Native | Inactive |        75 | //广播锁
| PowerManager.SuspendLockout   |   571 | Native | Active   |        38 | //禁止autdoSupend锁
| PowerManagerService.Display   |   571 | Native | Active   |        74 | //亮屏锁
| suspend_stats_lock    |   214 | Native | Inactive |         5 |
| IdleMaint             |   197 | Native | Inactive |        13 |
| ApmAudio              |   389 | Native | Inactive |        31 |
```

2. dumpsys power 关注以下几点。

```
mStayOn=true //当为true的时，插入适配器无法屏幕超时休眠，需要按POWER键或者应用发起休眠
Wake Locks: size=0
PARTIAL_WAKE_LOCK          'SleepTest-cpu' ACQ=-1s855ms (uid=10125 pid=2607) //通过这个看是哪个应用持的
        PARTIAL_WAKE_LOCK阻止休眠
Suspend Blockers: size=4
 PowerManagerService.WakeLocks: ref count=0
 PowerManagerService.Display: ref count=1
 PowerManagerService.Broadcasts: ref count=0
 PowerManagerService.WirelessChargerDetector: ref count=0 //任意一个不等于0都不会休眠
```

另外 screen off timeout 表示休眠时间，可通过该值判断你设置的定时时间是否正确；

如果是 androidN ，screen off timeout 还取决于 Sleep timeout：Sleep timeout 的值比 Screen off timeout 小的时候，系统取 Sleep timeout 当做定时休眠的时间，当 Sleep timeout 为-1 时，也表示系统无法定时休眠。

```
Sleep timeout: -1 ms  //只有androidN平台无该该值，定时休眠时间
Screen off timeout: 1800000 ms  //定时灭屏时间
Screen dim duration: 7000 ms   //屏保时间
```

常见场景：eng 固件开发阶段为了测试长时间老化的测试项，会禁止系统定时进入休眠。

问题解决

确认是 Android 系统持锁阻止休眠，方案开发人员可以自行解决。

4.5.1.3 休眠流程发起异常

说明

关键词：Android 休眠流程发起异常

问题现象

机器无法进入 Linux 休眠流程。

问题分析

没有 Linux 休眠标志信息，可能是 Android 未发起休眠流程。

问题排查

- 步骤一：查看是否有正常发起 Going to Sleep。

输入指令：

```
logcat -s PowerManagerService
```

```
PowerManagerService: Powering off display group due to power_button (groupId= 0, uid= 1000)...
PowerManagerService: Going to sleep due to power_button (uid 1000)...
PowerManagerService: Dozing...
```

- 步骤二：与正常机器对比 Power Manager State 查看是否有异常。

输入指令：

```
dumpsys power
```

- 步骤三：查看按键是否有正常上报。

输入指令：

```
dumpsys input | grep -i keyevent
```

- 步骤四：阅读源码查看 PhoneWindowManager 是否接收了 POWER_KEY 但没有处理。

4.5.1.4 系统休眠过程中被唤醒

说明

关键词：休眠过程被唤醒

问题现象

系统休眠过程中被唤醒，提示唤醒源为 NETLINK 或其他模块。

排查方法

dumpsys suspend_control_internal 查看 WakeupInfo

```
Wakeups:唤醒的信息
WakeupInfo{name: 352 axp2202_irq_chip, count: 14} //Resume caused by IRQ 352, axp2202_irq_chip 从深度睡眠中唤醒
    的
WakeupInfo{name: Abort: Last active Wakeup Source: rwnx_rx_wakelock, count: 7} //之前wakeup_source处于active，但
    现在已经deactive了
WakeupInfo{name: 391 7000000.rtc, count: 36}
WakeupInfo{name: Abort: Pending Wakeup Sources: axp2202-battery, count: 2}//处于active的wakeup_source
WakeupInfo{name: Abort: Last active Wakeup Source: rwnx_pwrcrl_wakelock, count: 1}
WakeupInfo{name: Abort: Pending Wakeup Sources: NETLINK, count: 6}
WakeupInfo{name: Abort: Last active Wakeup Source: eventpoll, count: 1}
WakeupInfo{name: Abort: Pending Wakeup Sources: [timerfd], count: 2}
WakeupInfo{name: Abort: Pending Wakeup Sources: event0, count: 1}
WakeupInfo{name: Abort: Pending Wakeup Sources: usb_connecting, count: 5}
```

或者直接查看 log

```
Resume caused by IRQ 325, xradio_irq
Last active Wakeup Source: rwnx_rx_wakelock
Pending Wakeup Sources: NETLINK
```

问题分析

系统休眠过程中被唤醒，若提示唤醒源为 NETLINK，可能是 NETLINK 唤醒源为应用层 epoll 唤醒，当系统休眠过程中，如果有 epoll_wait 调用，并且接收到 epoll 消息，则系统唤醒。通过在查询 event 消息或在 epoll_wait 消息中打印是哪个消息，查看是哪里发送的 uevent 消息，判断是哪个驱动或者内核发送消息，从而定位问题。

问题解决

确认唤醒源为 NETLINK 或其他模块，定位问题，方案开发人员可以自行解决，或者找相应的模块负责人解决。

- 实例一

在 A100 项目中，系统休眠过程中被唤醒，提示为 NETLINK 唤醒。通过排查发现，为应用层调用 epoll_wait 等待电源消息，休眠过程中 nand 驱动调用 regulator_put 函数，导致发送应用层一个 regulator_add 消息，从而系统被唤醒。修改 nand 驱动 regulator_put 的错误使用，解决此问题。

- 实例二

在 A523 项目中，系统休眠过程中，提示为 NETLINk 唤醒，如下 log 所示：

```
[  434.158846][ T3894] psci: CPU7 killed (polled 0 ms)
[  434.166215][ T3894] PM: PM: Pending Wakeup Sources: NETLINK
[  434.172680][ T3894] Enabling non-boot CPUs ...
[  434.178154][    T0] Detected VIPT I-cache on CPU1
[  434.178197][    T0] GICv3: CPU1: found redistributor 100 region 0:0x0000000003480000
[  434.178258][    T0] CPU1: Booted secondary processor 0x0000000100 [0x412fd050]
[  434.179833][ T3894] CPU1 is up
[  434.205823][    T0] Detected VIPT I-cache on CPU2
[  434.205865][    T0] GICv3: CPU2: found redistributor 200 region 0:0x000000000034a0000
[  434.205918][    T0] CPU2: Booted secondary processor 0x0000000200 [0x412fd050]
[  434.207859][ T3894] CPU2 is up
[  434.233843][    T0] Detected VIPT I-cache on CPU3
[  434.233881][    T0] GICv3: CPU3: found redistributor 300 region 0:0x000000000034c0000
[  434.233930][    T0] CPU3: Booted secondary processor 0x0000000300 [0x412fd050]
[  434.236148][ T3894] CPU3 is up
```

图 4-2: NETLINK wakeup event 0

分析思路：

首先是确定目标，即找出哪个模块发送的 NETLINK event，但从正面无法查到具体哪个模块，不过可从侧面找出是哪个模块发送了 NETLINK event，即在发送 NETLINK event 最底层 __netlink_sendskb() 接口加 dump_stack() 接口（因为休眠无法用 ftrace），如下图所示：

```
1261 extern int get_suspend_status(void);
1262
1263 static int __netlink_sendskb(struct sock *sk, struct sk_buff *skb)
1264 {
1265         int len = skb->len;
1266
1267         if (get_suspend_status())
1268                 dump_stack();
1269
1270         netlink_deliver_tap(sock_net(sk), skb);
1271
1272         skb_queue_tail(&sk->sk_receive_queue, skb);
1273         sk->sk_data_ready(sk);
1274         return len;
1275 }
1276
```

图 4-3: NETLINK wakeup event 1

由于很多很多模块都会发送 NETLINK event，故需要在休眠场景过滤掉不关心的事件，可通过如下标志状态过滤，即将进入休眠时设置休眠标志，在 __netlink_sendskb() 调用 dump_stack() 接口前做判断，如下图所示：

```
560 static atomic_t is_in_suspend;
561
562 int get_suspend_status(void)
563 {
564         return atomic_read(&is_in_suspend);
565 }
566 EXPORT_SYMBOL_GPL(get_suspend_status);
567
568 static int enter_state(suspend_state_t state)
569 {
570         int error;
571
572         trace_suspend_resume(TPS("suspend_enter"), state, true);
573         if (state == PM_SUSPEND_TO_IDLE) {
 NORMAL  kernel/power/suspend.c
```

图 4-4: NETLINK wakeup event 2

```
566 EXPORT_SYMBOL_GPL(get_suspend_status);
567
568 static int enter_state(suspend_state_t state)
569 {
570         int error;
571
572         trace_suspend_resume(TPS("suspend_enter"), state, true);
573         if (state == PM_SUSPEND_TO_IDLE) {
574 #ifdef CONFIG_PM_DEBUG
575                 if (pm_test_level != TEST_NONE && pm_test_level <= TEST_CPUS) {
576                         pr_warn("Unsupported test mode for suspend to idle, please
577                         return -EAGAIN;
578                 }
579 #endif
580         } else if (!valid_state(state)) {
581                 return -EINVAL;
582         }
583         if (!mutex_trylock(&system_transition_mutex))
584                 return -EBUSY;
585
586         if (state == PM_SUSPEND_TO_IDLE)
587                 s2idle_begin();
588
589         if (sync_on_suspend_enabled) {
590                 trace_suspend_resume(TPS("sync_filesystems"), 0, true);
591                 ksys_sync_helper();
592                 trace_suspend_resume(TPS("sync_filesystems"), 0, false);
593         }
594
595         pm_pr_dbg("Preparing system for sleep (%s)\n", mem_sleep_labels[state]);
596         pm_suspend_clear_flags();
597         error = suspend_prepare(state);
598         if (error)
599                 goto Unlock;
600
601         if (suspend_test(TEST_FREEZER))
602                 goto Finish;
603
604         trace_suspend_resume(TPS("suspend_enter"), state, false);
605         pm_pr_dbg("Suspending system (%s)\n", mem_sleep_labels[state]);
606         pm_restrict_gfp_mask();
607
608         atomic_set(&is_in_suspend, 1);
609         error = suspend_devices_and_enter(state);
610         atomic_set(&is_in_suspend, 0);
611
612         pm_restore_gfp_mask();
613
614  Finish:
615         events_check_enabled = false;
 NORMAL  kernel/power/suspend.c
```

图 4-5: NETLINK wakeup event 3

添加完 debug 信息后，复现后即可找出哪个模块发送的 NETLINK event，调用栈关系如下图所示：

```
[  434.020614][   T57] CPU: 7 PID: 57 Comm: cpuhp/7 Tainted: G        W  OE     5.15.41-dirty #24
[  434.030243][   T57] Hardware name: sun55iw3 (DT)
[  434.035398][   T57] Call trace:
[  434.038901][   T57]  dump_backtrace.cfi_jt+0x0/0x8
[  434.044253][   T57]  show_stack+0x1c/0x2c
[  434.048731][   T57]  dump_stack_lvl+0x68/0x98
[  434.053598][   T57]  __netlink_sendskb+0xc0/0x100
[  434.058852][   T57]  netlink_broadcast_deliver+0x80/0xe4
[  434.064785][   T57]  do_one_broadcast+0x204/0x310
[  434.070040][   T57]  netlink_broadcast_filtered+0xdc/0x218
[  434.076170][   T57]  netlink_broadcast+0x1c/0x2c
[  434.081324][   T57]  uevent_net_broadcast_untagged+0xd0/0x150
[  434.087744][   T57]  kobject_uevent_net_broadcast+0x104/0x154
[  434.094162][   T57]  kobject_uevent_env+0x28c/0x318
[  434.099613][   T57]  kobject_uevent+0x18/0x28
[  434.104479][   T57]  device_del+0x2ec/0x514
[  434.109151][   T57]  thermal_cooling_device_unregister+0x1f4/0x234
[  434.116058][   T57]  cpufreq_cooling_unregister+0x24/0x48
[  434.122089][   T57]  cpufreq_offline+0x19c/0x398
[  434.127246][   T57]  cpuhp_cpufreq_offline+0x14/0x28
[  434.132792][   T57]  cpuhp_invoke_callback+0x224/0x9d8
[  434.138529][   T57]  cpuhp_thread_fun+0x16c/0x1d8
[  434.143784][   T57]  smpboot_thread_fn+0x2fc/0x424
[  434.149138][   T57]  kthread+0x170/0x1dc
[  434.153517][   T57]  ret_from_fork+0x10/0x20
[  434.158846][ T3894] psci: CPU7 killed (polled 0 ms)
[  434.166215][ T3894] PM: PM: Pending Wakeup Sources: NETLINK
[  434.172680][ T3894] Enabling non-boot CPUs ...
[  434.178154][    T0] Detected VIPT I-cache on CPU1
[  434.178197][    T0] GICv3: CPU1: found redistributor 100 region 0:0x0000000003480000
[  434.178258][    T0] CPU1: Booted secondary processor 0x0000000100 [0x412fd050]
[  434.179833][ T3894] CPU1 is up
[  434.205823][    T0] Detected VIPT I-cache on CPU2
[  434.205865][    T0] GICv3: CPU2: found redistributor 200 region 0:0x00000000034a0000
[  434.205918][    T0] CPU2: Booted secondary processor 0x0000000200 [0x412fd050]
[  434.207859][ T3894] CPU2 is up
[  434.233843][    T0] Detected VIPT I-cache on CPU3
[  434.233881][    T0] GICv3: CPU3: found redistributor 300 region 0:0x00000000034c0000
[  434.233930][    T0] CPU3: Booted secondary processor 0x0000000300 [0x412fd050]
[  434.236148][ T3894] CPU3 is up
[  434.262107][    T0] Detected VIPT I-cache on CPU4
[  434.262129][    T0] GICv3: CPU4: found redistributor 400 region 0:0x00000000034e0000
[  434.262160][    T0] CPU4: Booted secondary processor 0x0000000400 [0x412fd050]
```

图 4-6: NETLINK wakeup event 4

从调用栈信息可以看出是在注销 cpufreq cooling device 时发送的 NETLINK event，都是内核原生框架的代码，走读整个调用过程的源码，可以通过设置 kobject 中 uevent_suppress 值控制 event 是否发送，如下图所示：

```
457 int kobject_uevent_env(struct kobject *kobj, enum kobject_action action,
458                         char *envp_ext[])
459 {
460         struct kobj_uevent_env *env;
461         const char *action_string = kobject_actions[action];
462         const char *devpath = NULL;
463         const char *subsystem;
464         struct kobject *top_kobj;
465         struct kset *kset;
466         const struct kset_uevent_ops *uevent_ops;
467         int i = 0;
468         int retval = 0;
469
470         /*
471          * Mark "remove" event done regardless of result, for some subsystems
472          * do not want to re-trigger "remove" event via automatic cleanup.
473          */
474         if (action == KOBJ_REMOVE)
475                 kobj->state_remove_uevent_sent = 1;
476
477         pr_debug("kobject: '%s' (%p): %s\n",
478                  kobject_name(kobj), kobj, __func__);
479
480         /* search the kset we belong to */
481         top_kobj = kobj;
482         while (!top_kobj->kset && top_kobj->parent)
483                 top_kobj = top_kobj->parent;
484
485         if (!top_kobj->kset) {
486                 pr_debug("kobject: '%s' (%p): %s: attempted to send uevent "
487                          "without kset!\n", kobject_name(kobj), kobj,
488                          __func__);
489                 return -EINVAL;
490         }
491
492         kset = top_kobj->kset;
493         uevent_ops = kset->uevent_ops;
494
495         /* skip the event, if uevent_suppress is set*/
496         if (kobj->uevent_suppress) {
497                 pr_debug("kobject: '%s' (%p): %s: uevent_suppress "
498                                  "caused the event to drop!\n",
499                                  kobject_name(kobj), kobj, __func__);
500                 return 0;
501         }
502         /* skip the event, if the filter returns zero. */
503         if (uevent_ops && uevent_ops->filter)
504                 if (!uevent_ops->filter(kset, kobj)) {
505                         pr_debug("kobject: '%s' (%p): %s: filter function "
 NORMAL  lib/kobject_uevent.c
```

图 4-7: NETLINK wakeup event 5

接下来需要找出在哪个模块哪个时机设置该变量，通过阅读代码，发现在 cpufreq 驱动 suspend 和 resume 接口设置该变量是最佳的，在 suspend 接口先保存默认值，再把该变量设置为 true，resume 时恢复默认值，如下图所示：

```
1589 static int cpufreq_offline(unsigned int cpu)
1590 {
1591         struct cpufreq_policy *policy;
1592         int ret;
1593
1594         pr_debug("%s: unregistering CPU %u\n", __func__, cpu);
1595
1596         policy = cpufreq_cpu_get_raw(cpu);
1597         if (!policy) {
1598                 pr_debug("%s: No cpu_data found\n", __func__);
1599                 return 0;
1600         }
1601
1602         down_write(&policy->rwsem);
1603         if (has_target())
1604                 cpufreq_stop_governor(policy);
1605
1606         cpumask_clear_cpu(cpu, policy->cpus);
1607
1608         if (policy_is_inactive(policy)) {
1609                 if (has_target())
1610                         strncpy(policy->last_governor, policy->governor->name,
1611                                 CPUFREQ_NAME_LEN);
1612                 else
1613                         policy->last_policy = policy->policy;
1614         } else if (cpu == policy->cpu) {
1615                 /* Nominate new CPU */
1616                 policy->cpu = cpumask_any(policy->cpus);
1617         }
1618
1619         /* Start governor again for active policy */
1620         if (!policy_is_inactive(policy)) {
1621                 if (has_target()) {
1622                         ret = cpufreq_start_governor(policy);
1623                         if (ret)
1624                                 pr_err("%s: Failed to start governor\n", __func__);
1625                 }
1626
1627                 goto unlock;
1628         }
1629
1630         if (cpufreq_thermal_control_enabled(cpufreq_driver)) {
1631                 cpufreq_cooling_unregister(policy->cdev);
1632                 trace_android_vh_thermal_unregister(policy);
1633                 policy->cdev = NULL;
1634         }
1635
1636         if (has_target())
1637                 cpufreq_exit_governor(policy);
NORMAL  drivers/cpufreq/cpufreq.c
```

图 4-8: NETLINK wakeup event 6

```
311 static struct cpufreq_driver dt_cpufreq_driver = {
312         .flags = CPUFREQ_NEED_INITIAL_FREQ_CHECK |
313                  CPUFREQ_IS_COOLING_DEV,
314         .verify = cpufreq_generic_frequency_table_verify,
315         .target_index = set_target,
316         .get = cpufreq_generic_get,
317         .init = cpufreq_init,
318         .exit = cpufreq_exit,
319         .online = cpufreq_online,
320         .offline = cpufreq_offline,
321         .register_em = cpufreq_register_em_with_opp,
322         .name = "cpufreq-dt",
323         .attr = cpufreq_dt_attr,
324         .suspend = sunxi_cpufreq_suspend,
325         .resume = sunxi_cpufreq_resume,
326 };
327
 NORMAL  drivers/cpufreq/cpufreq-linux-5.15/cpufreq-dt.c
```

图 4-9: NETLINK wakeup event 7

```
1: d/c/c/cpufreq-dt.c
251 static int sunxi_cpufreq_suspend(struct cpufreq_policy *policy)
252 {
253         struct device *dev = NULL;
254         struct cpufreq_cdev_info *info = &cdev_info;
255         unsigned int suspend_policy_cnt = info->suspend_policy_cnt;
256
257         cpufreq_generic_suspend(policy);
258
259         if (!policy->cdev)
260                 return 0;
261
262         dev = &(policy->cdev->device);
263         ++suspend_policy_cnt;
264         if (suspend_policy_cnt > CPUFREQ_CDEV_NUM) {
265                 pr_err("%s, unsupport policy num!\n", __func__);
266                 return -EINVAL;
267         } else {
268                 info->uevent_info[suspend_policy_cnt - 1].uevent_suppress =
269                         dev_get_uevent_suppress(dev);
270                 info->uevent_info[suspend_policy_cnt - 1].policy = policy;
271         }
272
273         info->suspend_policy_cnt = suspend_policy_cnt;
274         dev_set_uevent_suppress(dev, true);
275
276         return 0;
277 }
278
279 static int sunxi_cpufreq_resume(struct cpufreq_policy *policy)
280 {
281         struct device *dev = NULL;
282         unsigned int i = 0;
283         struct cpufreq_cdev_info *info = &cdev_info;
284         unsigned int suspend_policy_cnt = info->suspend_policy_cnt;
285
286         if (!policy->cdev)
287                 return 0;
288
289         dev = &(policy->cdev->device);
290
291         for (i = 0; i < suspend_policy_cnt; i++) {
292                 if (policy == info->uevent_info[i].policy) {
293                         dev_set_uevent_suppress(dev,
294                                 info->uevent_info[i].uevent_suppress);
295                         break;
296                 }
297         }
298
299         if (i >= suspend_policy_cnt) {
300                 pr_err("%s, policy err!\n", __func__);
NORMAL  drivers/cpufreq/cpufreq-linux-5.15/cpufreq-dt.c
```

图 4-10: NETLINK wakeup event 8

- 实例三

在 A733 项目中，Android 系统出现无法休眠下去的问题，log 如下所示：

```
03.583662][  T622] PM: PM: Pending Wakeup Sources: NETLINK
03.583819][ T4384] PM: Some devices failed to suspend, or early wake event detected
03.583844][ T4384] sunxi:g2d_sunxi:[INFO]: [G2D]: g2d_resume succesfully
03.584611][ T4384] NSI_PMU 2020000.nsi-controller: resume okay
03.584864][ T4384] sunxi:sunxi_mmc_host-4021000.sdmmc:[INFO]: dat3_imask 0
evice VINTF manifest.
```

图 4-11: NETLINK SCSI 0

从 log 可以看出 NETLINK 阻止系统休眠，所以需要按照实例二的方法找出是哪个模块发送了 NETLINK event，添加 debug 信息 (Linux-6.6 内核版本)：

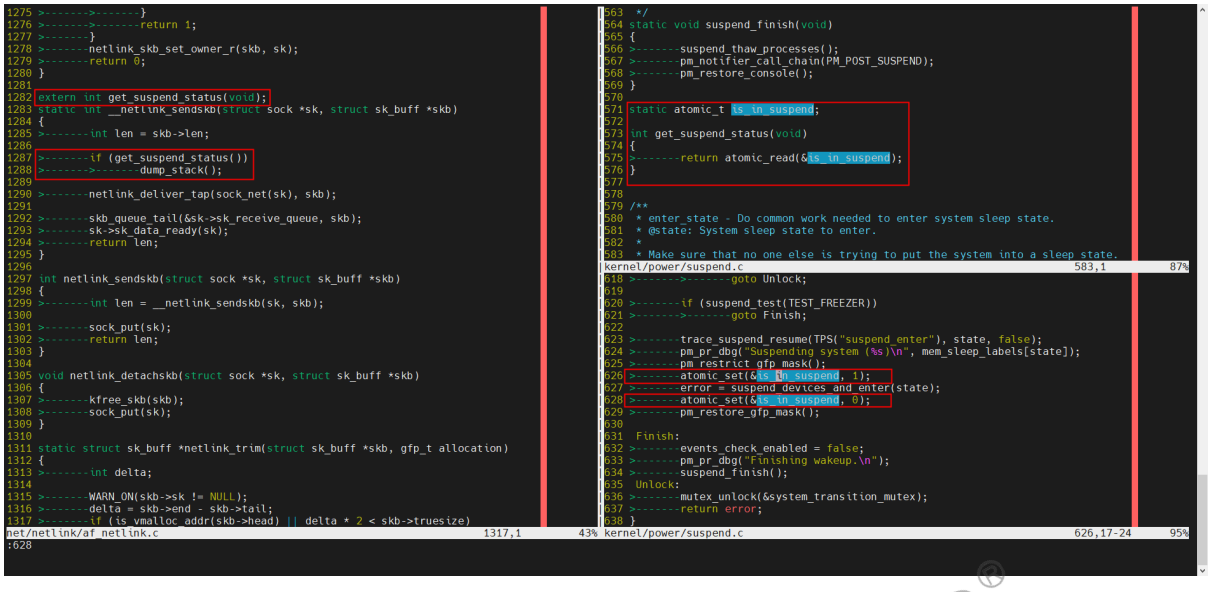


图 4-12: NETLINK SCSI 1

打上补丁后，用新的固件进行休眠测试，log 如下：

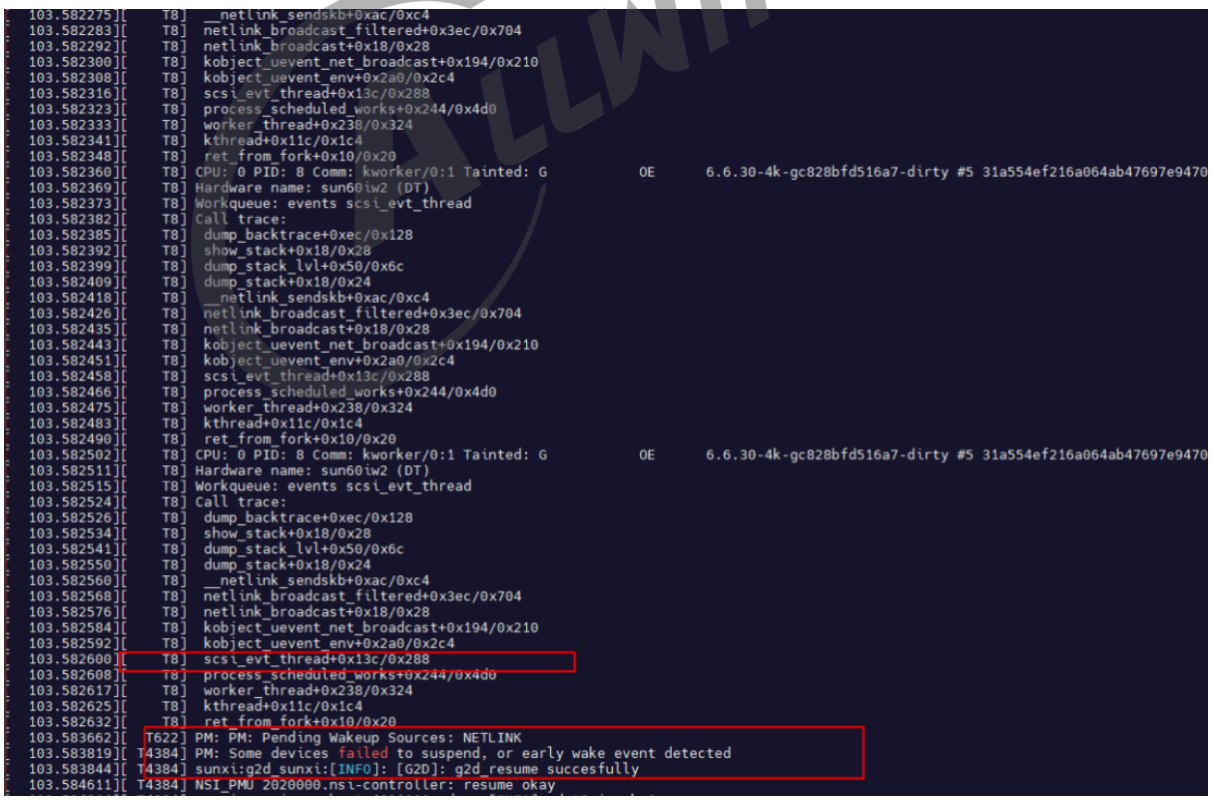


图 4-13: NETLINK SCSI 2

从 log 可以看出是 scsi 模块发起 NETLINK event，导致系统无法休眠下去，由存储模块负责人解

决。

4.5.2 系统无法唤醒

4.5.2.1 休眠后无法唤醒

说明

关键词：唤醒

问题现象

系统休眠后无法唤醒。

问题分析

系统休眠后无法唤醒，原因可能有：

- 模块唤醒失败

唤醒源是通过产生中断将系统唤醒，如果唤醒失败可能是设置的唤醒源模块没有正常支持唤醒功能，此时可以进入 cpus 的 while 循环前后打印相关模块寄存器的值。然后将打印值给模块负责人进行分析。

- 唤醒后显示异常，误判唤醒失败。

需要：确认屏幕是否亮屏

```
dumpsys display | grep mScreenState
```

和检查 screencap 查看是否有图像

- 模块休眠失败。查看是否模块休眠失败，输入以下命令，确认是否在内存休眠唤醒模块出异常。

```
echo N > /sys/module/printk/parameters/console_suspend echo 1 > /sys/power/pm_print_times
```

- 如果是走 input 的设备，输入以下命令，确认按键事件是否有上报。

```
getevent -l
```

- 若上述无异常打印，则认为是 Linux 后的阶段出现异常。不断电重启系统，将启动时候的 RTC 寄存器的信息发给相关负责人，根据 RTC 寄存器信息判断。

```
·UaUUUUUUUUUUUUUUaUUUUUUUUUUUUUUUaUUaUaUUUUUUUUUUUUUUU[335]HELLO!·BOOT0·is·starting!
·[338]sbrom·commit·:·88bc314602bda0eaab36b49faf376a6835daf2ee

·[356]rsb_send_initseq:·rsb·clk·400Khz·->·3Mhz
·[360]PMU:·AXP858
·[362]set·pll·start
·[372]try·to·probe·rtc·region
·[375]rtc[0]·value·=·0x00000000
·[378]rtc[1]·value·=·0x00000000
·[381]rtc[2]·value·=·0x00000000
·[384]rtc[3]·value·=·0xf3f37006
·[387]rtc[4]·value·=·0x00000000
·[390]rtc[5]·value·=·0x00000000
·[393]DRAM·BOOT·DRIVE·INFO:·V0.6·
·[396]DRAM·CLK·=672·MHZ
·[398]DRAM·Type·=7·(3:DDR3,4:DDR4,7:LPDDR3,8:LPDDR4)
```

图 4-14: RTC 寄存器的信息·例

问题解决

- 显示异常。确认是显示异常，将问题交给显示相关负责人。
- 模块休眠失败。确认是模块休眠失败，方案开发人员可以自行解决。
- input 事件没有上报。确认是 input 事件没有上报，检查 input 设备和相关驱动。
- Linux 后的阶段出现异常。将复位重启时的 RTC 寄存器信息发给相关负责人。

4.5.2.2 唤醒源不支持唤醒

说明

关键词：唤醒源

问题现象

休眠后，唤醒源无法唤醒系统，串口没有输出。

问题分析

休眠后，唤醒源无法唤醒，可能是唤醒源不支持。

- cpus 休眠后异常。当出现如下打印时表示 Linux 休眠已经完毕，此时唤醒不了，则可能是 cpus 退出休眠失败，或者唤醒源不对。

```
[ 3465.885063] PM: noirq suspend of devices complete after 16.487 msecs
[ 3465.892225] Disabling non-boot CPUs ...
......
```

通过以下手段可以判断 cpus 休眠后是否正常运行，以下命令表示休眠后 cpus 过一定时间软件自动唤醒。

```
echo 1000 > /sys/module/suspend/parameters/time_to_wakeup //休眠 1000ms 后自动唤醒
```

如果串口能正常打印，wake up source 为 0x400000，则表示 cpus 是正常运行的，这时应该排查一下系统是否支持相应的唤醒源。

- 唤醒源不支持。确认唤醒源不支持的情况。

问题解决

- cpus 休眠后异常。将复位重启时的 RTC 寄存器信息发给相关负责人。
- 唤醒源不支持。将唤醒源的情况发给相关负责人。

4.5.2.3 红外遥控器不能唤醒系统

说明

关键词：红外唤醒

问题现象

红外遥控不能唤醒系统。

问题分析

红外遥控唤醒需要配置唤醒源。

问题解决

红外遥控器默认支持 NEC 红外协议遥控器唤醒，也支持 RC5 红外协议遥控器唤醒，但是需要在 sys_config.fex 进行配置，配置如下：

```
;----------------------------------------------------------------------------
;ir --- infra remote configuration
;ir_protocol_used : 0 = NEC / 1 = RC5
;----------------------------------------------------------------------------
[s_cir0]
```

```
s_cir0_used      = 1
ir_used          = 1
ir_protocol_used   = 0 //置为0支持NEC红外协议遥控唤醒，配置为0支持RC5红外协议遥控唤醒
```

对于同一协议的红外遥控器，能支持唤醒的个数也是有限的，具体在 sys_config.fex 配置 s_cir0 节点。

```
ir_power_key_code0 = 0x57    //遥控POWER值
ir_addr_code0      = 0x9f00  //遥控设备码
ir_power_key_code1 = 0x1a
ir_addr_code1      = 0xfb04
```

4.5.2.4 USB 设备不能唤醒系统

说明

关键词：USB 唤醒

问题现象

USB 不能唤醒系统。

问题分析

USB 需要设置为唤醒源。

问题解决

USB 设备唤醒需要系统支持 USB_STANDBY，需要在 sys_config.fex 配置 usbc 节点。需要注意:

```
[usbc0]
usb_wakeup_suspend = 1
```

1 表示支持 USB0 唤醒，0 表示屏蔽该唤醒源。

4.5.2.5 Hdmi_cec 不能唤醒系统

说明

关键词：HDMI 唤醒

问题现象

HDMI 不能唤醒系统。

问题分析

HDMI 需要设置为唤醒源。

问题解决

在 sys_config.fex 配置 hdmi 节点。

```
[hdmi]
hdmi_cec_support = 1
hdmi_cec_super_standby = 1
```

4.5.2.6 cpus 退出休眠失败

说明

关键词：cpus 唤醒

问题现象

休眠后，无法唤醒，串口没有输出。

问题分析

可能是 cpus 退出休眠失败。

如果通过写 time_to_wakeup 命令，系统没法正常唤醒，则考虑是 cpus 退出休眠失败的了。这时需要短接 reset 脚重启系统（注意不是完全断电，完全断电将无法保留 RTC 值），然后将 boot 阶段打印的 RTC 码值发送给相关负责人，定位问题。

```
[2341]HELLO! pmu_init stub called!
[2645]set pll start
[2648]set pll end
[2649]try to probe rtc region
[2652]rtc[0] value = 0x00000000
[2655]rtc[1] value = 0x000000e0
[2658]rtc[2] value = 0xf1f18000
[2661]rtc[3] value = 0x0000000f
[2663]rtc[4] value = 0x00000000
[2666]rtc[5] value = 0x00000000
```

问题解决

- 确认是否 dram 错误。
- 确认是否上下电时序错误。
- 将复位重启时的 RTC 寄存器信息发给相关负责人。

4.5.2.7 dram 退出自刷新失败

说明

关键词：dram

问题现象

系统唤醒时，dram 退出自刷新失败，导致系统无法唤醒。

问题分析

dram 退出自刷新失败的常见原因。

- dram 类型选择错误。
- dram 驱动版本问题。
- dram 驱动宏控制有误。
- dram 参数问题。
- dram 颗粒问题。
- dram 供电问题。

问题解决

- dram 类型选择错误。确认清楚 dram 类型，选择正确的 bin。
- dram 驱动版本问题。与相关负责人确认清楚驱动版本。
- dram 驱动宏控制有误。检查驱动中 dram 类型相关的宏。
- dram 参数问题。与相关负责人确认 dram 参数是否正确；确认 bin 中的 dram 参数是否正确。
- dram 颗粒问题。与方案硬件负责人和相关负责人确认 dram 颗粒是否不良。
- dram 供电问题。确认休眠后 dram 供电是否正确。

4.5.2.8 上下电时序错误导致无法唤醒

说明

关键词：上下电时序

问题现象

客案的电源方案相比标案有改动，特别是系统供电、dram 供电，导致系统无法唤醒。

问题分析

需要保证休眠唤醒上下电时序的正确。上下电时序常见错误。

- 上下电顺序错误。如标案要求休眠过程中，VDD-CPU 先掉电，VDD-SYS 后掉电；唤醒则相反。确认客案是否满足要求。
- 上电后供电未稳定。如标案要求上电后需要适当延时，确保 VDD-CPU、VDD-SYS、VCC-PLL 等供电稳定后，系统才能进行唤醒流程。确认客案是否满足要求。

问题解决

- 上下电顺序错误。保证上下电顺序符合标案要求。
- 上电后供电未稳定。保证上电后供电稳定符合标案要求。

4.5.2.9 系统唤醒后又自动进入休眠

说明

关键词：自动进入休眠

问题现象

使用闹钟或者电源按键或者遥控唤醒系统后，屏幕黑屏，串口有输出，过会系统又自动进入休眠，表现为好像系统没有唤醒一样。

问题分析

Android 系统有自动休眠的机制，当没有模块（如显示、USB 等）持锁，系统超时后会自动进入休眠。这就是系统唤醒后又自动进入休眠的原因。

休眠唤醒后，控制台执行命令logcat | grep PowerManagerService，可以看出唤醒原因、亮屏耗时、休眠原因等信息。

```
08-24 00:08:17.536   498   595 I PowerManagerService: Waking up from Asleep (uid=1000, reason=
        WAKE_REASON_POWER_BUTTON, details=android.policy:POWER)...
08-24 00:08:18.140   498   563 W PowerManagerService: Screen on took 1124 ms
08-24 00:08:27.018   498   563 I PowerManagerService: Going to sleep due to timeout (uid 1000)...
08-24 00:08:27.021   498   563 I PowerManagerService: Sleeping (uid 1000)...
08-24 00:09:18.456   498   595 I PowerManagerService: Waking up from Asleep (uid=1000, reason=
        WAKE_REASON_POWER_BUTTON, details=android.policy:POWER)...
08-24 00:09:18.854   498   563 W PowerManagerService: Screen on took 856 ms
08-24 01:25:49.758   498   595 I PowerManagerService: Going to sleep due to power_button (uid 1000)...
08-24 01:25:49.763   498   563 I PowerManagerService: Sleeping (uid 1000)...
08-24 01:26:47.299   498  6062 I PowerManagerService: Waking up from Asleep (uid=1000, reason=
        WAKE_REASON_APPLICATION, details=android.server.am:TURN_ON:turnScreenOnFlag)...
08-24 01:26:47.690   498   563 W PowerManagerService: Screen on took 391 ms
```

- 按键或者遥控驱动是否上报 power key 事件。首先确认模块驱动功能正常，可以正常上报事件；再确认模块在唤醒系统时，也有上报事件。
- 显示驱动是否亮屏。在有上报事件的前提下，确认显示驱动是否有亮屏。
- 系统是否超时自动休眠。确认是否因为自动休眠阈值时间太小，导致系统超时自动休眠。

问题解决

- 按键或者遥控驱动没有上报 power key 事件。如果确认某些驱动模块没有上报事件，方案开发人员可以自行解决，或者找相应的模块负责人解决。

- 显示驱动没有亮屏。如果确认在有上报事件的前提下显示驱动没有亮屏，方案开发人员可以自行解决，或者找相应的模块负责人解决。
- 系统超时自动休眠。如果是因为自动休眠阈值时间太小，导致系统超时自动休眠，可以尝试修改自动休眠阈值时间：

```
78    // Create non-HW binder threadpool for SuspendControlService.
79    sp<android::ProcessState> ps{android::ProcessState::self()};
80    ps->startThreadPool();
81
82    sp<SystemSuspend> suspend =                                    改为500
83        new SystemSuspend(std::move(wakeupCountFd), std::move(stateFd), 100 /* maxStatsEntries */,
84                          100ms /* baseSleepTime */, suspendControl, false /* mUseSuspendCounter*/);
85    status_t status = suspend->registerAsService();
86    if (android::OK != status) {
87        LOG(FATAL) << "Unable to register system-suspend service: " << status;
88    }
89
90    joinRpcThreadpool();
91    std::abort(); /* unreachable */
NORMAL  /home1/panzhijian/Workspace   H616_Q/android/system/hardware/interfaces/suspend/1.0/default/main.cpp
```

图 4-15: Android 11 对应修改

```
57
58 static constexpr uint32_t kDefaultMaxSleepTimeMillis = 60000;
59 static constexpr uint32_t kDefaultBaseSleepTimeMillis = 100;
60 static constexpr double kDefaultSleepTimeScaleFactor = 2.0;
61 static constexpr uint32_t kDefaultBackoffThresholdCount = 0;
62 static constexpr uint32_t kDefaultShortSuspendThresholdMillis = 0;      改为500
63 static constexpr bool kDefaultFailedSuspendBackoffEnabled = true;
64 static constexpr bool kDefaultShortSuspendBackoffEnabled = false;
65
NORMAL  /home1/panzhijian/Workspace/android12   system/hardware/interfaces/suspend/1.0/default/main.cpp
```

图 4-16: Android 12 对应修改

4.5.3 系统异常唤醒

4.5.3.1 系统被定时器唤醒

说明

关键词：定时器唤醒

问题现象

休眠后，自动被唤醒，过会自动进入休眠，屏幕黑屏，串口有输出。

问题分析

系统休眠后自动被唤醒，原因可能是，某些应用或者后台进程，通过设置闹钟的方式，定时唤醒系统。

当出现如下打印，表示 Linux 已经休眠完成，准备进入 CPUS 休眠阶段：

```
[ 3465.885063] PM: noirq suspend of devices complete after 16.487 msecs
[ 3465.892225] Disabling non-boot CPUs ...
......
```

当出现以上打印后自动唤醒，则查看如下打印：

```
[ 3466.063570] wake up source:0x80000  //H3 linux4.4平台
[21676.174594] [pm]platform wakeup, standby wakesource is:0x100000  //H5/H6 linux-3.10平台
后面的数字代表唤醒源，根据数字定位唤醒源，定位唤醒源后再判断为何被唤醒。
WAKEUP_SRC is as follow:
CPUS_WAKEUP_LOWBATT             bit 0x1000
CPUS_WAKEUP_USB                 bit 0x2000
CPUS_WAKEUP_AC                  bit 0x4000
CPUS_WAKEUP_ASCEND              bit 0x8000
CPUS_WAKEUP_DESCEND             bit 0x10000
CPUS_WAKEUP_IR                  bit 0x80000
CPUS_WAKEUP_ALM0                bit 0x100000
CPUS_WAKEUP_HDMI_CEC            bit 0x100000
```

常见场景：android 某些应用或者后台进程，会通过设置闹钟的方式，定时唤醒系统，当判断唤醒源为 0x100000 时，大多数为该原因导致。

问题解决

确认是某些应用或者后台进程设置闹钟定时唤醒系统，方案开发人员可以自行解决。

4.5.3.2 系统被其他唤醒源唤醒

说明

关键词：异常唤醒

问题现象

休眠后，被异常唤醒。

问题分析

系统休眠后被异常唤醒，原因可能是，被其他非预期的唤醒源唤醒。

查看唤醒源。对应的代码路径在：lichee/linux4.9/include/linux/power/aw_pm.h。

```
/* the wakeup source of assistant cpu: cpus */
#define CPUS_WAKEUP_HDMI_CEC   (1<<11)
#define CPUS_WAKEUP_LOWBATT    (1<<12)
#define CPUS_WAKEUP_USB        (1<<13)
#define CPUS_WAKEUP_AC         (1<<14)
#define CPUS_WAKEUP_ASCEND     (1<<15)
#define CPUS_WAKEUP_DESCEND    (1<<16)
#define CPUS_WAKEUP_SHORT_KEY  (1<<17)
#define CPUS_WAKEUP_LONG_KEY   (1<<18)
#define CPUS_WAKEUP_IR         (1<<19)
#define CPUS_WAKEUP_ALM0       (1<<20)
#define CPUS_WAKEUP_ALM1       (1<<21)
```

```
#define CPUS_WAKEUP_TIMEOUT    (1<<22)
#define CPUS_WAKEUP_GPIO       (1<<23)
#define CPUS_WAKEUP_USBMOUSE   (1<<24)
#define CPUS_WAKEUP_LRADC      (1<<25)
#define CPUS_WAKEUP_WLAN       (1<<26)
#define CPUS_WAKEUP_CODEC      (1<<27)
#define CPUS_WAKEUP_BAT_TEMP   (1<<28)
#define CPUS_WAKEUP_FULLBATT   (1<<29)
#define CPUS_WAKEUP_HMIC       (1<<30)
#define CPUS_WAKEUP_POWER_EXP  (1<<31)
#define CPUS_WAKEUP_KEY        (CPUS_WAKEUP_SHORT_KEY | CPUS_WAKEUP_LONG_KEY)
```

查看关键打印：

```
platform wakeup, standby wakesource is:0x800000
```

此时对应的是 gpio 唤醒。

问题解决

确认是其他非预期的唤醒源唤醒系统，方案开发人员可以自行解决。

4.5.3.3 系统被非预期频繁唤醒

说明

关键词：频繁唤醒

问题现象

休眠后，系统被频繁唤醒，过会又自动进入休眠，反反复复，导致功耗大、待机短。

问题分析

系统休眠后自动被唤醒，最大可能是，某些应用或者后台进程，通过设置闹钟的方式，定时唤醒系统；或者某些驱动模块如网络，作为唤醒源，通过中断唤醒系统。

首先查看唤醒源，确认清楚是什么模块引起频繁唤醒。如果唤醒源是某些驱动模块，则可以确认是这些模块引起问题；如果唤醒源是 RTC 模块，则需要进一步确认是什么应用或后台进程设置了闹钟。

- 安卓查看闹钟相关信息，通过 Top Alarms 信息，可以定位出是哪些应用或者后台进程设置闹钟：

```
dumpsys alarm

......
 Top Alarms:
  +3m18s343ms running, 0 wakeups, 30 alarms: u0a98:com.google.android.gms
    *alarm*:com.google.android.gms/.vision.DependencyBroadcastReceiverProxy
  +1m14s294ms running, 0 wakeups, 24 alarms: 1000:android
```

```
  *alarm*:TIME_TICK
 +39s128ms running, 5 wakeups, 5 alarms: u0a98:com.google.android.gms
  *walarm*:com.google.android.gms.gcm.ACTION_CHECK_QUEUE
 +34s679ms running, 9 wakeups, 9 alarms: u0a98:com.google.android.gms
  *walarm*:com.google.android.intent.action.GCM_RECONNECT
 +31s782ms running, 0 wakeups, 1 alarms: 1000:android
  *alarm*:com.android.server.action.NETWORK_STATS_POLL
 +27s825ms running, 1 wakeups, 1 alarms: u0a54:com.android.providers.calendar
  *walarm*:com.android.providers.calendar.intent.CalendarProvider2
 +6s612ms running, 1 wakeups, 1 alarms: u0a121:com.google.android.calendar
  *walarm*:com.google.android.calendar.intent.action.CHECK_NOTIFICATIONS
 +140ms running, 0 wakeups, 1 alarms: u0a87:com.google.android.configupdater
  *alarm*:com.google.android.configupdater.DELAYED_FLAG_COMMIT
 +118ms running, 1 wakeups, 1 alarms: 1000:android
  *walarm*:EventConditionProvider.EVALUATE
 +114ms running, 1 wakeups, 1 alarms: 1000:android
  *walarm*:ScheduleConditionProvider.EVALUATE
.....
```

如之前有遇到过是 WiFiConnectivityManager 设置闹钟，周期性唤醒系统。

```
05      UID u0a17: -51m40s296ms  Next allowed:-51m35s296ms (+5s0ms)
06    mUseAllowWhileIdleShortTime: [u0a9 u0a12 u0a16 u0a27 u0a29 u0a41 u0a53 u0ai90
07
08    Top Alarms:
09      +212ms running, 0 wakeups, 52 alarms: 1000:android
10        *alarm*:TIME_TICK
11      +193ms running, 29 wakeups, 29 alarms: 1073:com.android.networkstack
12        *walarm*:DhcpClient.eth0.KICK
13      +147ms running, 0 wakeups, 2 alarms: 1000:android
14        *alarm*:com.android.server.action.NETWORK_STATS_POLL
15      +94ms running, 20 wakeups, 20 alarms: 1000:android
16        *walarm*:WifiConnectivityManager Schedule Periodic Scan Timer
17      +51ms running, 1 wakeups, 1 alarms: u0a17:com.android.providers.calendar
18        *walarm*:com.android.providers.calendar.intent.CalendarProvider2
19      +40ms running, 1 wakeups, 1 alarms: 1000:android
20        *walarm*:*job.deadline*
21      +11ms running, 0 wakeups, 1 alarms: u0a10:com.android.providers.tv
22        *alarm*:com.android.providers.tv.intent.CLEAN_UP_EPG_DATA
23      +9ms running, 2 wakeups, 2 alarms: 1000:android
24        *walarm*:WifiConnectivityManager Schedule Watchdog Timer
25      +6ms running, 1 wakeups, 1 alarms: 1000:android
26        *walarm*:*job.delay*
27
28    Alarm Stats:
```

图 4-17: Top Alarms 信息

问题解决

如果确认是某些驱动模块通过中断唤醒系统，方案开发人员可以自行解决，或者找相应的模块负责人解决；如果确认是某些应用或者后台进程设置闹钟定时唤醒系统，方案开发人员可以修改应用解决。

4.5.3.4 系统异常亮屏

说明

关键词：异常亮屏

问题现象

休眠后，系统被唤醒，屏幕异常亮起。

问题分析

系统休眠后自动被唤醒，有可能是，某些应用或者后台进程，通过设置闹钟的方式，定时唤醒系统，但屏幕没有亮起来，这可能是正常的。但是屏幕如果异常亮起，很可能是不正常的。

例如，按 power 按键系统唤醒屏幕亮起，这是正常的；设置定时闹钟到点后系统唤醒屏幕亮起铃声响起，这是正常的。又如，某些模块如 WiFi，设置闹钟周期性唤醒系统屏幕亮起，这是不正常的。

首先查看屏幕亮起的原因，Android 屏幕亮起的原因有以下这些，如 power 按键、应用等：

```
/**
 * Wake up reason code: Waking for an unknown reason.
 * @hide
 */
public static final int WAKE_REASON_UNKNOWN = 0;

/**
 * Wake up reason code: Waking up due to power button press.
 * @hide
 */
public static final int WAKE_REASON_POWER_BUTTON = 1;

/**
 * Wake up reason code: Waking up because an application requested it.
 * @hide
 */
public static final int WAKE_REASON_APPLICATION = 2;

/**
 * Wake up reason code: Waking up due to being plugged in or docked on a wireless charger.
 * @hide
 */
public static final int WAKE_REASON_PLUGGED_IN = 3;

/**
 * Wake up reason code: Waking up due to a user performed gesture (e.g. douple tapping on the
 * screen).
 * @hide
 */
public static final int WAKE_REASON_GESTURE = 4;

/**
 * Wake up reason code: Waking up due to the camera being launched.
 * @hide
```

```
 */
public static final int WAKE_REASON_CAMERA_LAUNCH = 5;

.....
```

再通过 logcat | grep PowerManagerService 命令，确认屏幕亮起的原因。如下就是 power 按键引起屏幕亮起。

```
02-26 17:58:31.550  2253  2375 I PowerManagerService: Waking up from Asleep (uid=1000, reason=
        WAKE_REASON_POWER_BUTTON, details=android.policy:POWER)...
02-26 17:58:31.551  2253  2253 W UsageStatsService: Event reported without a package name, eventType:15
02-26 17:58:31.555  225[ 2477.571730] stk_ps_enable_store: Enable PS : 1
3  2295 I DisplayPowerController: Blocking screen on until initial contents have been drawn.
02-26 17:58:31.560  3284  3284 D KeyguardClockSwitch: Updating clock: 5□58
02-26 17:58:31.575  2253  2291 E KernelCpuSpeedReader: Failed to read cpu-freq: /sys[ 2477.597457]
        disp_runtime_resume
/devices/system/cpu/cpu0/cpufreq/stats/time_in_state: open faile[ 2477.605844] stk3x1x_enable_ps: HT=1700,LT=1500
d: ENOENT (No such file or directory)
02-26 17:58:31.582  2253 [ 2477.618845] stk3x1x_enable_ps: ps input event=0, ps code = 1712
 2295 V DisplayPowerController: Brightness [102] reason changing[ 2477.627229] [DISP] lcd_clk_config,line:665:
 to: 'manual', previous reason: [ 2477.627231] disp 0, clk: pll(408000000),clk(408000000),dclk(68000000) dsi_rate
        (68000000)
[ 2477.627231]     clk real:pll(408000000),clk(408000000),dclk(102000000) dsi_rate(150000000)
'screen_off'.
02-26 17:58:31.582  1852  1852 D SensorBase: Could not open (write-only) SysFs attribute "/sys/class/input/input5/
        ps_poll_delay" (Permission denied).
02-26 17:58:31.591  1850  1850 I AW_PowerHAL: ==SET_INTER 1
02-26 17:58:31.591  1850  1850 I AW_PowerHAL_Platform: ==NORMAL MODE==
02-26 17:58:31.594  2253  2280 I DisplayManagerService: Display device changed state: "Built-in Screen", ON
02-26 17:58:31.594  1861  1861 D SurfaceFlinger: Setting power mode 2 on display 0
02-26 17:58:31.608  1846  1846 D sunxihwc: setPowerMode: device(lcd0) power mode: On
02-26 17:58:31.657  3284  3284 D StatusBar: disable<e i a s b H R c s > disable2<q i n >
02-26 17:58:31.666  2253  6159 I BroadcastQueue: Delay finish: com.google.android.gms/.vision.
        DependencyBroadcastReceiverProxy
02-26 17:58:31.657  3284  3284 D StatusBar: disable<e i a s b H R c s > disable2<q i n >
02-26 17:58:31.683  3890 25443 I Vision  : Requesting optional module download of ocr.
02-26 17:58:31.685  3890 25443 I Vision  : Checking for download completion for 2491245 -- ocr
02-26 17:58:31.698  2253  2295 I DisplayPowerController: Unblocked screen on after 142 ms
```

确认清楚屏幕亮起是什么原因。如果屏幕亮起的原因是某些驱动模块上报事件，则可以确认是这些模块引起问题；如果屏幕亮起的原因是某些应用，则需要进一步确认是什么应用。

问题解决

如果确认屏幕亮起的原因是某些驱动模块上报事件，方案开发人员可以自行解决，或者找相应的模块负责人解决；如果确认屏幕亮起的原因是某些应用，方案开发人员可以修改应用解决。

4.5.4 休眠唤醒过程中挂掉

4.5.4.1 分阶段过程挂掉

说明

关键词：阶段失败

问题现象

在 Linux 某个阶段出现的休眠或者唤醒失败。

问题分析

定位具体是在哪个阶段出现的休眠或者唤醒失败。

打开内核选项

```
CONFIG_PM_DEBUG=y
```

查看具体失败在哪个阶段

1. echo freezer > /sys/power/pm_test 查看 freezer 阶段是否 ok
2. echo devices > /sys/power/pm_test 查看 freezer/devices 阶段是否 ok
3. echo platform > /sys/power/pm_test 查看 freezer/devices/platform 阶段是否 ok
4. echo processors > /sys/power/pm_test 查看 freezer/devices/platform/processors 阶段是否 ok
5. echo core > /sys/power/pm_test 查看 freezer/devices/platform/processors/core 阶段是否 ok
6. echo mem > /sys/power/state 测试分阶段休眠唤醒

问题解决

确认是哪个阶段出现的休眠或者唤醒失败，方案开发人员可以自行解决。

4.5.4.2 DE 异常访问 dram 导致唤醒失败

说明

关键词：唤醒失败

问题现象

在 Linux 某个阶段出现唤醒失败。

问题分析

定位具体是在哪个阶段出现的唤醒失败。

打开内核选项

```
CONFIG_PM_DEBUG=y
```

问题解决

有些方案不带显示功能的，uboot 开启显示，但 kernel 显示相关配置项关闭的话，唤醒过程中访问 ddr 系统会卡死，原因是 uboot 开启显示时，DE 硬件会一直读取 ddr 内存空间 logo 数据，系统休眠时因为 DE 没走正常的 suspend 休眠流程，导致 de 一直访问 ddr，而此时 ddr 已进入自刷新状态，ddr 自刷新时会关闭所有访问通道，此时访问 ddr 会导致 ddr 异常，故如果方案不带显示功能，uboot 和 kernel 必须同时关闭。

4.5.5 其他休眠唤醒流程问题

4.5.5.1 休眠唤醒流程阻塞

说明

关键词：休眠唤醒阻塞

问题现象

在 Linux 某个阶段出现休眠/唤醒后无响应。

问题分析

定位具体是在哪个阶段出现的阻塞。

1. 打开内核配置。

```
CONFIG_MISC_FILESYSTEMS=y
CONFIG_PSTORE=y
CONFIG_DPM_WATCHDOG=y
DPM_WATCHDOG_TIMEOUT=120
```

2. 进行休眠唤醒

问题解决

当pm核心去调用驱动的suspend/resume的回调函数时，它会设置一个定时器来
监视回调函数的执行，如果回调长时间没有执行完毕的话(当前定时为1s/CONFIG_HZ*
CONFIG_DPM_WATCHDOG_TIMEOUT=1s/250*120=480ms)， 定时器函数会调用panic让系统挂掉，
然后就可以从panic的堆栈信息中获取对应的模块

4.5.5.2 休眠唤醒慢

说明

关键词：休眠唤醒慢

问题现象

在 Linux 某个阶段出现休眠唤醒速度慢的现象。

问题分析

可以先检查 log，看下是安卓部分休眠耗时长还是 Linux 中休眠耗时长。在 PM: suspend entry (deep) 之前，都是安卓端在处理休眠操作，之后是 Linux 内核处理休眠操作。

Linux 端：

打开所有休眠配置，然后看各模块休眠时间，如果有耗时过长的问题，找对应模块负责人优化：

```
su
echo 8 > /proc/sys/kernel/printk
echo N > /sys/module/printk/parameters/console_suspend
echo Y > /sys/module/kernel/parameters/initcall_debug
echo 1 > /sys/power/pm_print_times
```

安卓端：

当设备进入休眠状态时，安卓系统会发送一个系统广播，告知所有应用程序设备即将进入休眠。应用程序可以在接收到该广播后进行相应的理，例如保存当前状态、关闭不必要的服务等。但是安卓端某些应用长时间会持锁，所以会导致休眠时间大大延长。下面的命令可以打出持锁的应用 pid：

```
dumpsys power | grep -i "wake lock" -A10
```

问题解决

安卓：根据打出的 pid，可以看出是什么模块或应用持锁，导致休眠时间延长，然后寻找负责安卓的同事解决该问题。Linux：通过打印可以得知模块的耗时，针对唤醒时间较长的模块分析对应 suspend/resume 函数。

4.5.5.3 休眠唤醒后屏幕不亮

说明

关键词：休眠唤醒屏幕不亮

问题现象

在 Linux 某个阶段出现休眠唤醒屏幕不亮的现象。

问题分析

1. 确认屏幕是否亮屏：dumpsys display | grep mScreenState
2. screencap 查看是否有图像

4.6 常用调试节点

休眠唤醒遇到问题，有一些调试节点比较通用。通过这些节点，尽可能的输出信息，并从输出信息中，提取出有价值信息。

4.6.1 查看和设置 kernel wake_lock 状态

wake_lock 节点，可以用来阻止系统进入休眠。/sys/power/wake_lock 节点：可以用来查看 wake_lock 状态，定位应用层阻止系统休眠的情况。当遇到系统不能进入休眠，可能是某个应用持有 wake_lock 锁，导致系统不能进入休眠。

实例如下：

查看 wake_lock 状态，此时显示模块持有 wake_lock 锁。

```
cat /sys/power/wake_lock
PowerManager.SuspendLockout PowerManagerService.Display
```

/sys/power/wake_lock 和/sys/power/wake_unlock 节点，可以用来获取和释放 wake_lock。通过 test_wake_lock 获取或释放 wake_lock，可以方便调试休眠相关功能。实例如下：

设置 test_wake_lock 持有 wake_lock 锁，此时 test_wake_lock 和显示模块持有 wake_lock 锁；设置 test_wake_lock 释放 wake_lock 锁，此时显示模块持有 wake_lock 锁。

```
echo test_wake_lock > /sys/power/wake_lock
cat /sys/power/wake_lock
PowerManager.SuspendLockout PowerManagerService.Display test_wake_lock

echo test_wake_lock > /sys/power/wake_unlock
cat /sys/power/wake_lock
PowerManager.SuspendLockout PowerManagerService.Display
```

4.6.2 查看 android wake_lock 状态

查看与电源相关的信息。可以用于定位因为持有 wake_lock 阻止系统休眠的情况。实例如下：

```
dumpsys power
```

上述命令的输出信息较多，如果只关注 wake_lock 状态，可以对输出信息进行过滤。实例如下。

```
dumpsys power | grep -A 10 "Wake Locks"
Wake Locks: size=0

Suspend Blockers: size=4
  PowerManagerService.WakeLocks: ref count=0
  PowerManagerService.Display: ref count=1
  PowerManagerService.Broadcasts: ref count=0
  PowerManagerService.WirelessChargerDetector: ref count=0

Display Power: state=ON
```

4.6.3 查看 wakeup_sources 状态

查看内核持有的 wakeup_sources 状态。可以用于定位因为持有 wake_lock 阻止系统休眠的情况。确定 active_since 的值 value 是否为 0；若为非 0，表明该 wakeup_source 已经阻止系统进入休眠 value ms。

实例如下：

小机插入 usb。此时 usb_connecting 阻止系统休眠 8897ms。

```
cat /sys/kernel/debug/wakeup_sources
name                 active_count event_count wakeup_count expire_count active_since
PowerManagerService.WakeLocks 17        17        0         0         0
PowerManager.SuspendLockout   1         1         0         0         661598
PowerManagerService.Display   1         1         0         0         661604
usb_connecting           1         4         0         0         8897
```

小机拔出 usb。此时 usb_connecting 没有阻止系统休眠。

```
cat /sys/kernel/debug/wakeup_sources
name                 active_count event_count wakeup_count expire_count active_since
PowerManagerService.WakeLocks 17        17        0         0         0
PowerManager.SuspendLockout   1         1         0         0         661598
PowerManagerService.Display   1         1         0         0         661604
usb_connecting           1         4         0         0         0
```

如果 wakeup_sources 节点不存在，可以先确认 kernel 配置选项 CONFIG_DEBUG_FS 为 y，并执行如下命令。

```
mount -t debugfs none /sys/kernel/debug
```

4.6.4 设置是否挂起控制台（仅供调试）

console_suspend 节点，可以用来控制休眠唤醒流程中，是否允许控制台输出。该值为 N，表示打印信息通过控制台输出；该值为 Y，表示挂起控制台，此后打印信息不再通过控制台输出。在调试休眠唤醒问题时，一般设置为 N，用来定位问题。

实例如下：

```
echo N > /sys/module/printk/parameters/console_suspend
```

4.6.5 设置 initcall_debug

initcall_debug 节点，可以用来控制是否打印各个设备名称，调用开始和结束时刻；linux-3.10 及之后的版本，打印的是 syscore 对应的设备。

实例如下：

```
echo Y > /sys/module/kernel/parameters/initcall_debug
```

4.6.6 设置内核打印等级

printk 节点，用于设置内核的打印等级。通常在调试问题时，设置打印等级为 8。

实例如下：

```
echo 8 > /proc/sys/kernel/printk
```

4.6.7 查看系统中断信息

interrupts 节点，用于查看系统的中断信息。

实例如下：

```
cat /proc/interrupts
       CPU0       CPU1       CPU2       CPU3
 17:    1401        150        903       1758  wakeupgen  22 Level    sun4i_timer0
 18:       0          0          0          0    GIC-0   29 Level    arch_timer
 19:    1720        489        557        827    GIC-0   30 Level    arch_timer
 54:       1          0          0          0  wakeupgen  24 Level    rtc
 55:       0          0          0          0  wakeupgen  5  Level    RemoteIR_RX
 56:       0          0          0          0  wakeupgen  53 Level    cedar_dev
 57:     847          0          0          0  wakeupgen  1  Level    uart0
 58:   29494          0          0          0  wakeupgen  7  Level    twi0
 59:       0          0          0          0  wakeupgen  8  Level    twi1
```

4.6.8 设置 pm_print_times

pm_print_times 节点，可以用来控制是否打印各个设备名称，调用开始和结束时刻。该值为 1，休眠唤醒时会打印各个设备名称，调用开始和结束时刻。

实例如下：

```
echo 1 > /sys/power/pm_print_times
```

需要注意，Android11（linux-5.4）及之后的版本，由于 GKI 限制没有打开 CONFIG_PM_ADVANCED_DEBUG、CONFIG_PM_DEBUG 等内核配置，所以默认不支持本节点；有调试需求可以自行打开配置。

4.6.9 设置 pm_debug_messages

pm_debug_messages 节点，用来控制休眠唤醒的调试打印信息。linux-4.9 及之前的内核版本不支持本节点。该值为 1，休眠唤醒时会打印相关调试信息。

实例如下：

```
echo 1 > /sys/power/pm_debug_messages
```

需要注意，Android11（linux-5.4）及之后的版本，由于 GKI 限制没有打开 CONFIG_PM_ADVANCED_DEBUG、CONFIG_PM_DEBUG 等内核配置，所以默认不支持本节点；有调试需求可以自行打开配置。

4.6.10 触发 kernel 进入休眠

在 android 环境下，不建议使用。

实例如下：

```
echo mem > /sys/power/state
```

4.6.11 触发 android 进入休眠

实例如下：

```
input keyevent 26
```

4.6.12 查看和设置休眠唤醒测试模式

pm_test 节点，可以用来查看和设置休眠唤醒测试模式。

实例如下：

设置 pm_test 为 devices，并触发 kernel 进入休眠。

```
cat /sys/power/pm_test
[none] core processors platform devices freezer

echo devices > /sys/power/pm_test

cat /sys/power/pm_test
none core processors platform [devices] freezer

echo mem > /sys/power/state
```

各个测试模式的意义如下。

```
freezer：测试freezer阶段的休眠唤醒流程，延时后自动唤醒。
devices：测试freezer/devices阶段的休眠唤醒流程，延时后自动唤醒。
platform：测试freezer/devices/platform阶段的休眠唤醒流程，延时后自动唤醒。
processors：测试freezer/devices/platform/processors阶段的休眠唤醒流程，延时后自动唤醒。
core：测试freezer/devices/platform/processors/core阶段的休眠唤醒流程，延时后自动唤醒。
none：测试完整的休眠唤醒流程，等待唤醒源唤醒。
```

pm_test_delay 节点，可以用来设置测试模式的自动唤醒延时时间，单位是 s。

实例如下：

```
echo 10 > /sys/module/suspend/parameters/pm_test_delay
```

4.6.13 设置自动定时唤醒（仅供调试）

- linux-3.10 内核版本，设置自动唤醒的延时时间，单位是 ms。

```
echo 1000 > /sys/module/suspend/parameters/time_to_wakeup
```

或

```
echo 1000 > /sys/module/pm/parameters/time_to_wakeup_ms
```

- linux-4.9 内核版本，设置自动唤醒的延时时间，单位是 ms。

```
echo 1000 > /sys/power/sunxi_debug/time_to_wakeup_ms
```

或

```
echo 1000 > /sys/module/pm/parameters/time_to_wakeup_ms
```

- linux-5.4 内核版本，设置自动唤醒的延时时间，单位是 ms。

```
echo 1000 > /sys/class/sunxi_standby/time_to_wakeup_ms
```

4.6.14 设置 RTC 定时唤醒

- linux-4.9 内核版本，设置 RTC 定时唤醒的延时时间，单位是 s。

```
echo "+5" >/sys/class/rtc/rtc0/wakealarm
```

如果 wakealarm 节点不存在，可以先确认 kernel 配置 RTC 模块相关选项为 y，并执行如下命令，查找节点的具体路径；或咨询 RTC 模块负责人。

```
find / -name *wakealarm*
```

4.6.15 查看唤醒源

- linux-3.10 内核版本，查看唤醒时的打印信息，来确定唤醒源。

```
platform wakeup, standby wakesource is:0x10
```

或

```
wake up source:0x80000
```

如果出现 wakesource is:0x00，就可能是异常唤醒。

```
platform wakeup, standby wakesource is:0x00
```

- linux-4.9 内核版本，查看唤醒时的打印信息，来确定唤醒源。

```
platform wakeup, standby wakesource is:0x10
```

如果出现 wakesource is:0x00，就可能是异常唤醒。

```
platform wakeup, standby wakesource is:0x00
```

或
查看唤醒时的 pm_system_irq_wakeup 打印信息，来确定唤醒源。

```
[  34.590072] psci: CPU2 killed.
[  34.617351] Invalid sched_group_energy for CPU0
[  34.617356] Invalid sched_group_energy for Cluster0
[  34.617973] CPU3: shutdown
[  34.617996] psci: CPU3 killed.
[  34.618526] pm_system_irq_wakeup: 345 triggered rtc
[  34.618543] Enabling non-boot CPUs ...
[  34.618905] Detected VIPT I-cache on CPU1
[  34.618955] Invalid sched_group_energy for CPU1
[  34.618958] CPU1: update cpu_capacity 1024
[  34.618962] CPU1: Booted secondary processor [410fd034]
```

或
pm_wakeup_irq 节点，可以用来查看唤醒源。

实例如下：

54 是 RTC 模块的中断号，所以唤醒源是 RTC 模块。

```
cat /sys/power/pm_wakeup_irq
54

cat /proc/interrupts
       CPU0     CPU1     CPU2     CPU3
 17:    1401      150      903     1758  wakeupgen  22 Level    sun4i_timer0
 18:       0        0        0        0   GIC-0  29 Level    arch_timer
 19:    1720      489      557      827   GIC-0  30 Level    arch_timer
 54:       1        0        0        0  wakeupgen  24 Level    rtc
 55:       0        0        0        0  wakeupgen   5 Level    RemoteIR_RX
 56:       0        0        0        0  wakeupgen  53 Level    cedar_dev
 57:     847        0        0        0  wakeupgen   1 Level    uart0
Err:       0
```

如果出现错误信息，就可能是异常唤醒。

```
cat /sys/power/pm_wakeup_irq
cat: read error: No data available
```

4.6.16 设置唤醒源

- linux-4.9 内核版本，需要在 dts 中设置 wakeup-source 属性，具体参考相关驱动开发文档。

4.6.17 设置 dram 校验功能（仅供调试）

系统唤醒时，dram 需要退出自刷新模式，可以设置 dram 校验功能，对 dram 某一区域数据进行校验。若校验出错，则说明唤醒后 dram 数据有误，停止唤醒流程，并报出 “dram crc error” 等类似错误信息。实例如下。

- linux-4.9 内核版本，设置 dram 校验功能。其中 “1 0x40000000 0x100000”：第一个参数 1，表示打开 dram 校验功能；第二个参数 0x40000000，表示校验区域的起始地址；第三个参数 0x100000，表示校验区域的长度，单位是 byte。

```
echo 1 0x40000000 0x100000 > /sys/power/sunxi_debug/dram_crc_paras
```

或

```
echo 1 0x40000000 0x100000 > /sys/power/aw_ex_standby/dram_crc_paras
```

或
先查找节点的具体路径，再按照上述的格式，输入命令的参数。

```
find / -name dram_crc_paras
echo 1 0x40000000 0x100000 > /具体路径/dram_crc_paras
```

- linux-5.4 内核版本，设置 dram 校验功能。各个参数同 linux-4.9 内核版本。

```
echo 1 0x40000000 0x100000 > /sys/class/sunxi_standby/dram_crc_paras
```

4.7 通用调试手段

4.7.1 打开休眠唤醒调试信息

实例如下：

```
su
echo 8 > /proc/sys/kernel/printk
echo N > /sys/module/printk/parameters/console_suspend
echo Y > /sys/module/kernel/parameters/initcall_debug
echo 1 > /sys/power/pm_print_times

# 对于linux-5.4及之后内核版本
echo 1 > /sys/power/pm_debug_messages
```

4.7.2 打开 DPM_WATCHDOG 配置

设置看门狗检测设备驱动休眠唤醒的死锁。当系统卡在设备驱动休眠唤醒过程，则看门狗被触发并引起系统崩溃，打印出相关信息。：

打开内核配置。

```
CONFIG_MISC_FILESYSTEMS=y
CONFIG_PSTORE=y
CONFIG_DPM_WATCHDOG=y
DPM_WATCHDOG_TIMEOUT=120
```

打开系统调试打印。

```
su
echo 8 > /proc/sys/kernel/printk
```

实例如下。可以看出系统卡在 axp803_ac_power_suspend 函数中。

```
[  19.079590] PM: suspend entry 2020-11-25 20:36:48.435201383 UTC
[  19.086276] PM: Syncing filesystems ... done.
[  19.099417] PM: Preparing system for sleep (mem)
[  19.106708] Freezing user space processes ... (elapsed 0.001 seconds) done.
[  19.115838] Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
[  19.125374] PM: Suspending system (mem)
[  19.133743] sunxi-mmc sdc2: sdc set ios:clk 0Hz bm PP pm OFF vdd 0 width 1 timing LEGACY(SDR12) dt B
[  23.519822] axp803_usb_power: current limit not set: usb adapter type
[  23.519822]
[  50.143513] axp803-ac-power-supply axp803-ac-power-supply.0: **** DPM device timeout ****
[  50.152709] Call trace:
[  50.155466] [<ffffff80080856b8>] __switch_to+0xb0/0xcc
```

```
[  50.161245] [<ffffff80085d69bc>] __schedule+0x2ac/0x444
[  50.167118] [<ffffff80085d6bd8>] schedule+0x84/0xa0
[  50.172602] [<ffffff80085da304>] schedule_timeout+0x2c0/0x2f4
[  50.179062] [<ffffff80085da390>] schedule_timeout_uninterruptible+0x18/0x20
[  50.186893] [<ffffff80080ecb80>] msleep+0x18/0x24
[  50.192183] [<ffffff800846711c>] axp803_ac_power_suspend+0x10/0x14
[  50.199135] [<ffffff8008350674>] platform_pm_suspend+0x44/0x4c
[  50.205693] [<ffffff8008359d04>] dpm_run_callback+0x4c/0x94
[  50.211957] [<ffffff800835a4bc>] __device_suspend+0x240/0x2ec
[  50.218417] [<ffffff800835b9d8>] dpm_suspend+0xc0/0x1f4
[  50.224290] [<ffffff800835bdb8>] dpm_suspend_start+0x5c/0x68
[  50.230654] [<ffffff80080d791c>] suspend_devices_and_enter+0x68/0x180
[  50.237895] [<ffffff80080d7c48>] pm_suspend+0x214/0x298
[  50.243768] [<ffffff80080d6714>] state_store+0xb8/0xdc
[  50.249546] [<ffffff800828a6e0>] kobj_attr_store+0x14/0x24
[  50.255716] [<ffffff80081d3898>] sysfs_kf_write+0x44/0x50
[  50.261785] [<ffffff80081d28a0>] kernfs_fop_write+0x14c/0x18c
[  50.268246] [<ffffff800816d70c>] __vfs_write+0x1c/0xf4
[  50.274021] [<ffffff800816e38c>] vfs_write+0xc4/0x160
[  50.279700] [<ffffff800816f368>] SyS_write+0x44/0x88
[  50.285281] [<ffffff8008083200>] el0_svc_naked+0x34/0x38
[  50.291253] Kernel panic - not syncing: axp803-ac-power-supply axp803-ac-power-supply.0: unrecoverable failure
[  50.291253]
[  50.304166] CPU: 3 PID: 0 Comm: swapper/3 Not tainted 4.9.170 #5
[  50.310916] Hardware name: sun50iw10 (DT)
[  50.315418] Call trace:
[  50.318166] [<ffffff8008088b70>] dump_backtrace+0x0/0x23c
[  50.324235] [<ffffff8008088dc0>] show_stack+0x14/0x1c
[  50.329915] [<ffffff80082887b4>] dump_stack+0x90/0xb0
[  50.335594] [<ffffff80081224b4>] panic+0x128/0x358
[  50.340978] [<ffffff8008359bf0>] dpm_watchdog_set+0x0/0x68
[  50.347144] [<ffffff80080ecbb0>] call_timer_fn.isra.4+0x24/0x78
[  50.353799] [<ffffff80080ecc8c>] expire_timers+0x88/0xa8
[  50.359770] [<ffffff80080ecd48>] run_timer_softirq+0x9c/0x154
[  50.366229] [<ffffff800808116c>] __do_softirq+0x114/0x210
[  50.372298] [<ffffff800809e128>] irq_exit+0x88/0xc8
[  50.377782] [<ffffff80080dbea4>] __handle_domain_irq+0xb0/0xec
[  50.384339] [<ffffff8008080ec4>] gic_handle_irq+0x64/0xa0
```

4.7.3 休眠后系统供电

表 4-2: 休眠后系统供电情况

	VDD-CPU	VDD-SYS	VCC-PLL	VCC-IO	reserved	reserved
H3	N/A	N/A	N/A	N/A		
T3	power down	power down	N/A	power down		
T7	power down	power down	N/A	N/A		
A50	power down	power down	N/A	N/A		
H6	N/A	N/A	N/A	N/A		
V5	power down	power down	N/A	N/A		
V316	power down	power down	N/A	N/A		
R328	N/A	N/A	N/A	N/A		

	VDD-CPU	VDD-SYS	VCC-PLL	VCC-IO	reserved	reserved
V459	power down	power down	N/A	power down		
H3P	power down	power down	N/A	N/A		
A100	power down	power down	N/A	N/A		
H616	power on	power on	N/A	N/A		
T507	power down	power down	N/A	N/A		
A523	power down	power down	N/A	N/A		

4.7.4 休眠后 dram 供电

表 4-3: 休眠后 dram 供电情况

	VDD-DRAM	VCC-DRAM	VDD18-DRAM	VDD18-LPDDR	VPP-DRAM
ddr2	power down	1.8V	power down	N/A	N/A
ddr3	power down	1.5V	power down	N/A	N/A
ddr4	power down	1.2V	power down	N/A	2.5V
lpddr2	power down	1.2V	N/A	1.8V	N/A
lpddr3	power down	1.2V	N/A	1.8V	N/A
lpddr4	power down	1.1V	N/A	1.8V	N/A

各路供电说明见下，仅供参考，具体以方案硬件负责人的说法为准。

表 4-4: 各路供电说明

名称	电压范围 (V)	供电能力 (A)	说明
VCC-DRAM	1.0~1.8	1A	主控 IO 供电，super standby 时不可掉电外部 DRAM 颗粒供电，super standby 时不可掉电
VDD-DRAM	0.9~1.0	350mA	DRAM 控制器逻辑（主控）电路供电,，super standby 时可掉电。此电压在有些项目中，会和 VDD-SYS 并在一起要求 DRAM 初始化完成后此路不能调压
VDD18-DRAM	1.8	20mA	主控 1V8 路供电（PLL），super standby 时可掉电
VDD18-LPDDR	1.8	30mA	LPDDR2/LPDDR3/LPDDR4 颗粒 core 供电，super standby 时不可掉电
VPP-DRAM（***DDR4***）	2.5	60mA	DDR4 颗粒的激活电压，super standby 时不可掉电

名称	电压范围 (V)	供电能力 (A)	说明
SVREF	VCC_DRAM/2		DDR 颗粒存储单元 0/1 参考电压

4.7.5 休眠唤醒上下电时序

休眠唤醒的上下电时序，需要满足标案的要求。若客案的电源方案相比标案有改动，则需要确认是否符合要求，具体以方案硬件负责人的说法为准。

实例分析一之上下电顺序。在系统唤醒时，发现 cpux 卡在 brom 中无法启动；如果休眠后系统供电不掉电，则正常。原因可能是，系统休眠唤醒的上下电顺序错误。尝试将下电顺序调整为“VDD-CPU -> VCC-PLL -> VDD-SYS”，上电顺序则相反，问题解决。

实例分析二之上电延时。在某方案中需要使用 gpio 控制 VDD-SYS 上下电，在系统唤醒时，发现 cpus 卡在 r_timer 的延时函数中；如果使用标案的 PMU 上下电或休眠后系统供电不掉电，则正常。原因可能是，gpio 上电后没有保证供电稳定，就进行后续恢复的操作，导致 r_timer 控制器异常。尝试 gpio 控制 VDD-SYS 上电后，延时 5ms，问题解决。

4.7.6 获取休眠唤醒 RTC 状态码

RTC 状态码，是系统在休眠唤醒过程中，每经过一个阶段，就对 RTC 模块的寄存器进行修改，用来记录休眠唤醒流程到达哪个阶段的状态标志。一般用于定位休眠唤醒过程中系统卡死的问题，可以从 RTC 状态码得出系统在具体哪个阶段卡死。当休眠唤醒过程中系统卡死，保证系统不掉电，短接 reset 脚复位系统，boot 阶段将打印 RTC 码值；将 RTC 状态码反馈相关负责人。

```
[2341]HELLO! pmu_init stub called!
[2645]set pll start
[2648]set pll end
[2649]try to probe rtc region
[2652]rtc[0] value = 0x00000000
[2655]rtc[1] value = 0x000000e0
[2658]rtc[2] value = 0xf1f18000
[2661]rtc[3] value = 0x0000000f
[2663]rtc[4] value = 0x00000000
[2666]rtc[5] value = 0x00000000
```

以某个平台为例，说明 RTC 状态码的意义。仅供参考，具体以 CPUs 实际代码为准。另外，RTC 状态码属于内部调试信息，无需过多关注，将 RTC 状态码反馈相关负责人即可。

表 4-5: 休眠唤醒 RTC 状态码

基值 0	基值 1	子值	具体意义	对应函数
0xf3f30000	0x1000	0x00	进入 standby 流程	standby_entry()
0xf3f30000	0x1000	0x01	初始化 standby 相关的 dts 配置	standby_dts_parse()
0xf3f30000	0x2000	0x00	准备进入 standby init 流程	standby_process_init()
0xf3f30000	0x3000	0x01	进入 standby init 流程	standby_process_init()
0xf3f30000	0x3000	0x02	CPUs 关闭 core0	cpucfg_cpu_suspend()
0xf3f30000	0x3000	0x03	CPUs 关闭各个 device	device_suspend()
0xf3f30000	0x3000	0x04	CPUs 关闭 CPUx 相关时钟资源	cpu_pll_off()
0xf3f30000	0x3000	0x05	CPUs 让 DRAM 进入自刷新	dram_suspend()
0xf3f30000	0x3000	0x06	CPUs 切换/关闭系统相关时钟资源	clk_suspend()
0xf3f30000	0x3000	0x07	CPUS 复位 SYS 域/使能 ISO	system_suspend()
0xf3f30000	0x3000	0x08	预留的时钟操作	clk_suspend_late()
0xf3f30000	0x3000	0x09	操作 rtc_vccio	rtc_vccio_det_suspend()
0xf3f30000	0x3000	0x0A	CPUs 操作 PMU 掉电	dm_suspend()
0xf3f30000	0x4000	0x00	离开 standby init 流程	standby_process_init()
0xf3f30000	0x5000	0x00	等待唤醒	wait_wakeup()
0xf3f30000	0x6000	0x00	进入 standby exit 流程	standby_process_exit()
0xf3f30000	0x7000	0x01	进入 standby exit 流程	standby_process_exit()
0xf3f30000	0x7000	0x02	CPUs 操作 PMU 上电	dm_resume()
0xf3f30000	0x7000	0x03	操作 rtc_vccio	rtc_vccio_det_resume()
0xf3f30000	0x7000	0x04	CPUs 打开 DCXO	clk_resume_early()
0xf3f30000	0x7000	0x05	CPUs 关闭 ISO/SYS 域复位无效	system_resume()
0xf3f30000	0x7000	0x06	CPUs 恢复/切换系统相关时钟资源	clk_resume()
0xf3f30000	0x7000	0x07	CPUs 让 DRAM 退出自刷新	dram_resume()
0xf3f30000	0x7000	0x08	CPUs 恢复 CPUx 相关时钟资源	cpu_pll_on()
0xf3f30000	0x7000	0x09	CPUs 打开各个 device	device_resume()
0xf3f30000	0x7000	0x0A	CPUs 打开 core0	cpucfg_cpu_resume()
0xf3f30000	0x7000	0x0B	CPUs 等待 core0 唤醒	wait_cpu0_resume()
0xf3f30000	0x8000	0x00	离开 standby exit 流程	standby_process_exit()

表 4-6: 关机重启 RTC 状态码

基值 0	子值	具体意义	对应函数
0xa000	0x101	上层下达了关机指令	sys_op()
0xa000	0x102	上层下达了重启指令	sys_op()
0xa000	0x103	从 uboot 下达了关机指令，且不进行充电判断	sys_op()
0xa000	0x201	进入 pmu 关机函数	xxx_pmu_shutdown
0xa000	0x202	进入 pmu 重启函数	xxx_pmu_reset
0xa000	0x203	进入 pmu 关机充电函数	xxx_pmu_charging_reset
0xa000	0x300	即将写 pmu 关机	xxx_pmu_shutdown

基值 0	子值	具体意义	对应函数
0xa000	0x301	即将写 pmu 重启	xxx_pmu_reset
0xa000	0x401	快充相关重启处理	xxx_pmu_reset
0xa000	0x402	快充相关关机处理	xxx_pmu_reset
0xa000	0x501	快充相关关机充处理	xxx_pmu_reset

如果 RTC 状态码为 0xf3f35000，表示 “等待唤醒”；如果 RTC 状态码为 0xa301，表示 “pmu 重启”。

4.7.7 休眠唤醒测试用例

- linux-4.9 内核版本的休眠唤醒测试用例。

通过设置自动定时唤醒，进行休眠唤醒的完整通路测试。实例如下：

```
echo 2000 > /sys/power/sunxi_debug/time_to_wakeup_ms
echo mem > /sys/power/state
```

通过设置 RTC 定时唤醒，进行休眠唤醒的完整通路测试。实例如下：

```
echo "+5" >/sys/class/rtc/rtc0/wakealarm
echo mem > /sys/power/state
```

自动化测试脚本。实例如下，仅供参考：

```
echo 2000 > /sys/power/sunxi_debug/time_to_wakeup_ms
while [ 1 -eq 1 ]; do
  echo mem > /sys/power/state
  sleep 2
done
```

4.7.8 系统分量功耗测量

所谓的系统分量功耗，就是指各路供电、各个模块供电的具体功耗，与之相对的是整机功耗。它们的关系如下公式所示：

整机功耗 = 各路供电的功耗总和（各个模块供电的功耗总和）* 电源转化效率

实例如下：如下图所示，DCDCA、ALDO1 等是 PMU 的各路供电，VDD-CPU、VCC-USB 等是模块供电。

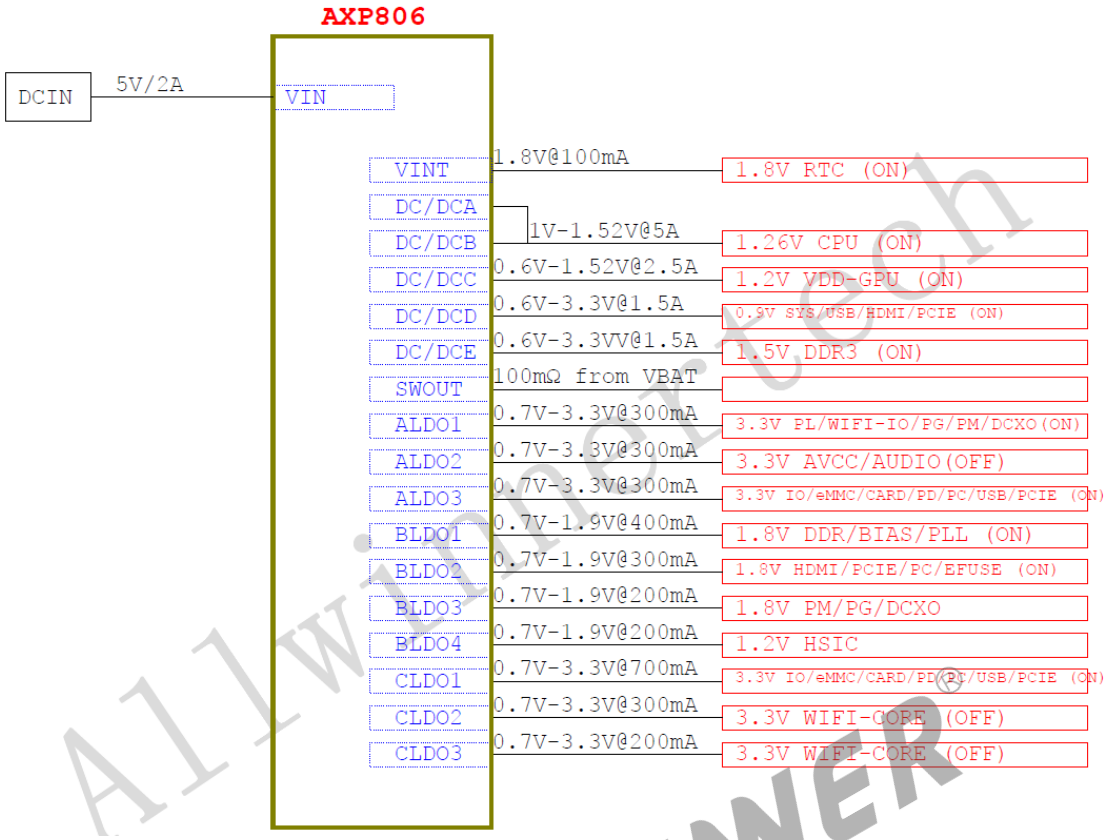


图 4-18: AXP806 供电图

如下表格，是各路供电的分量功耗。

表 4-7: 各路供电的分量功耗

测试场景	测试量	ALDO1	ALDO2	ALDO3	DLDO1
主界面 (DRAM 频率 372MHz)	电压/V	1.8	1.80	3.0	2.8
电流/mA	9.7	28.2	0.13	0.34	0.34
功耗 mW	17.46	50.76	0.39	0.95	0.95

如下表格，是各个模块供电的分量功耗。

表 4-8: 各个模块的分量功耗

测试场景	测试量	Battery	VDD-CPUA	VDD-SYS	VCC-DRAM
主界面 (DRAM 频率 372MHz)	电压/V	3.7	0.84	0.9	1.2
电流/mA	380.0	36.3	188.5	97.0	97.0
功耗 mW	1406.0	30.5	169.7	116.4	116.4

当出现休眠功耗异常的情况，一般都只是知道整机功耗偏大，这个时候是不能解决问题的；还需

要进行分量功耗测试，这样才能定位到具体是哪一路供电、哪一个模块的功耗偏大，之后就可以有针对性的排查。

5 重启类

5.1 pmu 掉电重启

5.1.1 pmu 过流掉电

说明

关键词：掉电，重启

问题现象

- 系统突然就没有打印输出，而且也不能输入，并且 PS、VDD-CPU、VDD-SYS、VCC-IO、AVCC 其中的一路或多路供电已经掉电。
- 系统突然重启，而且没有相关的错误信息，并且上述供电有掉落的现象。

问题分析

pmu 过流保护的原因有：

- pmu 供电设置输入限流
- 超过 pmu 极限负载。整板或模块实际所需供电大于 pmu 供电限流配置。
- pmu 高温限流。pmu 温度过高，引发过流保护。pmu 温度超过 105 度就会开始限流。
- pmu DCDC 或 LDO 过压、欠压

当出现 pmu 异常掉电时，可以按 reset 键复位或先长按电源键关机后短按电源键开机，系统重新启动，u-boot 阶段打印出开关机源寄存器信息，可以确定 pmu 异常掉电原因。具体操作详见《Tina_Linux_ 电源管理开发指南》。

问题解决

- pmu 设置供电限流。设置 ACIN 或 VBUS 的输入限流配置。具体操作详见《Tina_Linux_ 电源管理开发指南》。
- 超过 pmu 极限负载降低板级负载
- pmu 高温限流。pmu 高温限流是一个自动保护功能，可以采用散热片的方式物理降温，避免 pmu 高温。
- pmu DCDC 或 LDO 过压、欠压降低该路供电负载

5.1.2 外部电源掉电

说明

关键词：掉电，重启

问题现象

- 系统突然就没有打印输出，而且也不能输入；ACIN 或 VBUS 已经掉电。
- 系统突然重启，而且没有相关的错误信息；上述供电有掉落的现象。

问题分析

- 如果板级实际所需供电大于外部供电，可能会引发外部电源的过流保护，导致整板掉电及重启。
- 由于其他原因，如外挂 MCU 的异常处理，使外部电源关闭，导致 pmu 和主控掉电。

问题解决

- 提高外部电源供电或降低板级负载。
- 排查外挂 MCU 的异常情况

5.2 内核异常重启

问题分析

内核 panic 是会重启系统的，这个功能可以都过 menuconfig 来配置。

```
#内核panic就重新
CONFIG_PANIC_TIMEOUT=y
```

问题解决

分析内核 panic 的原因并解决。

6 温控类

6.1 cpu 高温或过温关机

说明

关键词：高温、过温关机

问题现象

- 在某场景下，cpu 温度一直比较高，摸上去发烫。
- 在某场景下，由于 cpu 温度过高，触发了软件的过温关机。

问题分析

cpu 温度一直比较高，会影响芯片寿命；
当 cpu 温度高于设定的过温关机温度，就会触发软件进行过温关机。

因为

```
cpu温度=机体温度+cpu温升
机体温度=环境温度+机体温升

注：机体温度是指，样机内部空间的温度，可以理解成是机体温升+环境温度。当样机不存在密封的机体结构时，机体温度等
    于环境温度。
```

所以

```
cpu温度=环境温度+机体温升+cpu温升
```

因此，cpu 温度取决于 cpu 温升和机体温升。温升取决于目标运行中发热情况和散热情况。

- cpu 温升

1. 发热。cpu、gpu、isp、de、ve、dram 等高功耗模块，是否负载、频率、电压等异常，导致发热过大。
2. 散热。可以考虑是否芯片散热较差。

- 机体温升

1. 发热。显示屏、充电、喇叭、通信模组、DRAM 颗粒等大功率器件，是否发热异常。

2. 散热。可以考虑是否机体散热较差。

分析步骤：

- 步骤 1

详细描述测试场景。包括但不限于：环境温度、场景描述、问题现象、散热情况。如：

```
1、环境温度：85度。
2、场景描述：在高温箱内，运行环视全景应用及算法，CVBS或HDMI或AHD显示，输出四个摄像头视频和UI加算法，无录像
    和其他操作。
3、问题现象：死机，T517内部温度log显示最高144度。
4、散热情况：铸铝散热外壳，硅胶连接铸铝散热外壳。
```

- 步骤 2

对 cpu 发热进行排查。

1、该场景的 cpu 负载、频率、温度等情况。使用如下命令，采集 60s 数据。

```
cpu_monitor -s 1000
```

2、该场景的 dram 带宽使用情况。使用如下命令，采集 60s 数据：

```
mtop -m
```

3、该场景各个进程对 cpu 的使用情况。使用如下命令，观察 60s 数据：

```
top
```

检查上述测试数据是否不合理或不符合预期，如 cpu 负载较大、某个 master 对 dram 带宽使用较多、某个进程对 cpu 使用较多等。

4、该场景各路供电的电压、开关状态情况。使用如下命令

```
mount -t debugfs none /sys/kernel/debug
cat /sys/kernel/debug/regulator/regulator_summary
```

检查各路供电状态是否不合理或不符合预期，具体找方案硬件负责人或相应的模块负责人。

5、该场景各个时钟、总线、pll 的频率、开关状态情况。使用如下命令

```
mount -t debugfs none /sys/kernel/debug
cat /sys/kernel/debug/clk/clk_summary
```

检查各个时钟状态是否不合理或不符合预期，具体找方案硬件负责人或相应的模块负责人。

6、该场景的 gpu、isp、de、ve、dram 等高功耗模块的电压、频率、功耗、工作模式等，检查是否不合理或不符合预期，具体找相应的模块负责人。

- 步骤 3

对机体发热进行排查。

该场景的显示屏、充电、喇叭、通信模组、DRAM 颗粒等大功率器件的工作模式、功耗、温度等，检查是否不合理或不符合预期，具体找相应的模块负责人。

- 步骤 4

对机体散热进行排查。

检查机体散热是否不合理或不符合预期，具体找方案硬件负责人。

问题解决

- cpu 发热

1. 如果确认该场景的 cpu 负载较大、某个 master 对 dram 带宽使用较多等情况不合理或不符合预期。方案开发人员可以自行解决，或者找熟悉该场景的系统人员协助解决。
2. 如果确认各路供电状态不合理或不符合预期。方案开发人员可以自行解决，或者找相应的模块负责人解决。
3. 如果确认各个时钟状态不合理或不符合预期。方案开发人员可以自行解决，或者找相应的模块负责人解决。
4. 如果确认是 gpu、isp、de、ve、dram 等高功耗模块的电压、频率、功耗、工作模式等不合理或不符合预期。方案开发人员可以自行解决，或者找相应的模块负责人解决。
5. 对于 cpu 模块，根据具体场景，可以适当降低性能，如降低工作电压和频率、增加 v-f 表的低频点，从而降低功耗、减少发热。也可以降低 cpu 温控的阈值温度和目标温度，适当降低高温性能，加大对 cpu 温度的控制力度，并尽量控制 cpu 在较低温度。
6. 电压和频率、使用低性能模式等，从而降低功耗、减少发热。方案开发人员可以自行解决，或者找相应的模块负责人解决。
7. 对于其他一般模块，根据具体场景，可以适当降低性能、甚至关闭模块，从而降低功耗、减少发热。具体找相应的模块负责人解决。

- cpu 散热

1. 增加芯片的散热片、加强芯片封装的散热设计、加强 PCB 的散热设计。具体找方案硬件负责人解决。

- 机体发热

1. 如果确认是显示屏、充电、喇叭、通信模组、DRAM 颗粒等大功率器件的工作模式、功耗、温度等不合理或不符合预期。方案开发人员可以自行解决，或者找相应的模块负责人解决。

2. 对于显示屏、充电、喇叭、通信模组、DRAM 颗粒等大功率器件，根据具体场景，可以适当使用低功耗模式，如降低显示屏亮度、限制充电电流、降低音量等，从而降低功耗、减少发热。方案开发人员可以自行解决，或者找相应的模块负责人解决。

- 机体散热

1. 增加机体的散热片，考虑风冷、水冷、液冷等主动散热方式。具体找方案硬件负责人解决。

- 其他

1. 如果发热合理且符合预期，但散热无法进一步改善，而且对性能要求不能降低，说明该场景 cpu 温度高的问题无法解决。

著作权声明

商标声明

免责声明