



Unisoc Confidential For hiar

# Kernel Stability 调试指导手册

文档版本  
发布日期

V1.5  
2020-07-28

**版权所有 © 紫光展锐科技有限公司。保留一切权利。**

本文件所含数据和信息都属于紫光展锐所有的机密信息，紫光展锐保留所有相关权利。本文件仅为信息参考之目的提供，不包含任何明示或默示的知识产权许可，也不表示有任何明示或默示的保证，包括但不限于满足任何特殊目的、不侵权或性能。当您接受这份文件时，即表示您同意本文件中内容和信息属于紫光展锐机密信息，且同意在未获得紫光展锐书面同意前，不使用或复制本文件的整体或部分，也不向任何其他方披露本文件内容。紫光展锐有权在未经事先通知的情况下，在任何时候对本文件做任何修改。紫光展锐对本文件所含数据和信息不做任何保证，在任何情况下，紫光展锐均不负任何与本文件相关的直接或间接的、任何伤害或损失。

请参照交付物中说明文档对紫光展锐交付物进行使用，任何人对紫光展锐交付物的修改、定制化或违反说明文档的指引对紫光展锐交付物进行使用造成的任何损失由其自行承担。紫光展锐交付物中的性能指标、测试结果和参数等，均为在紫光展锐内部研发和测试系统中获得的，仅供参考，若任何人需要对交付物进行商用或量产，需要结合自身的软硬件测试环境进行全面的测试和调试。非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

Unisoc Confidential For hiar

# 紫光展锐科技有限公司



# 前言

## 概述

本文档介绍了各种 kernel crash 问题，如 bug on, lockup, watchdog panic 等问题，以及 sysdump 的抓取和 crash 工具命令，串口调试方法。

## 读者对象


本文档主要适用于紫光展锐客户对 kernel stability 问题进行分享和调试。

## 缩略语

缩略语	英文全名	中文解释
sysdump	system dump	系统 dump
rd	Read	读取
bt	Back Trace	回溯

## 符号约定

在本文中可能出现下列标志，它所代表的含义如下。

符号	说明
 说明	用于突出重要/关键信息、补充信息和小窍门等。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害。

## 变更信息

文档版本	发布日期	修改说明
V1.0	2019-05-15	第一次正式发布。
V1.1	2019-05-22	增加缩略语说明，对概览做修订。

文档版本	发布日期	修改说明
V1.2	2019-08-28	适配 SL8563
V1.3	2020-01-03	适配 SL8521
V1.4	2020-04-15	1. 文档名由《Unisoc Kernel Stability & Debug Tools》修改为《Kernel Stability 调试指导手册》。 2. 修改优化文档结构、文档内容、文档样式、图表等。
V1.5	2020-07-28	1. 增加 2.4 章节，适配 UDX710，UDS710。 2. 增加 3.1 章节相关 UDX710，UDS710 内容。

## 关键字

Kernel、System dump、Crash、Panic。

Unisoc Confidential For hiar

# 目 录

1 Kernel Stability 问题简介 .....	1
1.1 重启的模式 .....	1
1.2 重启原理 .....	1
1.3 panic 函数的调用方式 .....	1
1.3.1 驱动调用 .....	2
1.3.2 组合键调用 .....	2
1.3.3 die 函数调用 .....	2
1.3.4 BUG_ON 过程调用 .....	3
1.3.5 hardlockup&softlock 调用 .....	3
1.3.6 Watchdog 调用以及 wdgreboot .....	3
2 抓取 system dump 流程 .....	4
2.1 system dump 抓取流程 .....	4
2.2 system dump 抓取界面 .....	4
2.3 sysdump 文件列表 .....	5
2.4 UDX710 sysdump 位置 .....	5
3 Crash 工具简介 .....	6
3.1 分析 system dump 流程 .....	6
3.2 常用的命令介绍 .....	7
3.2.1 显示 log 命令 .....	7
3.2.2 ps 命令 .....	7
3.2.3 bt 命令 .....	8
3.2.4 runq 命令 .....	8
3.2.5 irq 命令 .....	9
3.2.6 dataType pointer 命令 .....	9
3.2.7 compare 命令 .....	10
3.2.8 rd 命令 .....	10
3.3 Crash 常见问题分析 .....	11
4 串口 log 抓取方法 .....	20
5 参考文档 .....	21

## 图目录

---

图 2-1 system dump 抓取流程图 .....	4
图 2-2 system dump 抓取界面图 .....	5

Unisoc Confidential For hiar

# 1 Kernel Stability 问题简介

此章节主要探讨 kernel 的重启问题。Kernel 遇到异常时会重启，但要注意不是所有重启都是由于 kernel 引起的。我们可以从重启模式来判断是什么类型的重启。

## 1.1 重启的模式

如需确认重启模式，可从下面两种 log 中获取：

1. 重启后的 slog 中的 misc/cmdline.log ( 或者 adb 登入后查看 proc/cmdline)中的关键字 androidboot.mode。
2. ylog 中 snapshot/phone.info 中的 ro.bootmode。

具体从关键字来做模式判断方法如下：

- unknown, special 或者没有值，则为正常开机重启，上层触发的重启(如 android 看门狗重启，systemserver 重启，第三方应用发起的重启)或者手动长按 power 键重启。
- panic/ap wdgreboot 等值则是 kernel 发生问题(包括按下组合键)的重启，硬狗重启。
- 从重启前后 kernel log 原始时间戳，如[852605.423283]的连续性上也可以进一步验证，如果它们是连续的，则是上层重启，因为 kernel 重启的话会重置时间戳。

## 1.2 重启原理

Kernel panic 引起的重启点在：

```
void panic(const char *fmt, ...)
{...
    #ifdef CONFIG_SPRD_SYSDUMP
        sysdump_enter(0, buf, NULL);
    #endif
    ...
    emergency_restart();
    ...}
```

可以看出如果 sysdump 功能开启的话，会先执行 system\_dump，然后重启。

其它如 die 函数，BUG\_ON 过程也会有先调用 sysdump\_enter 后再做重启的操作。

## 1.3 panic 函数的调用方式

Kernel panic 函数调用方式比较多，本节主要根据代码情况进行描述其调用方式。具体情况如下面章节所述。

### 1.3.1 驱动调用

```
static void    init sirfsoc prima2 timer init(struct device node *np){
...
    sirfsoc_timer_base = of_iomap(np, 0);
    if (!sirfsoc timer base)
        panic("unable to map timer cpu registers\n");
...
}
```

### 1.3.2 组合键调用

对于定屏/黑屏，adb 不可连接的情况，可以尝试组合键(大部分情况下是按住上下侧键，同时点击两下 power 键)，如果 kernel 没有彻底死，可以触发 panic 流程：

```
void sprd_debug_check_crash_key(unsigned int code, int value)
{
...
    if (volup p && voldown p) {
...
        if (code == KEY_POWER) {
            ...
            if (loopcount == 2)
                panic("Crash Key");
            ...
        }
...
}
```

### 1.3.3 die 函数调用

由页错误引起 die 函数调用进而触发 panic

```
static
void do_kernel_fault(struct mm_struct *mm, unsigned long addr, unsigned int fsr,
struct pt_regs *regs)
{
...
    pr_alert("Unable to handle kernel %s at virtual address %08lx\n",
            (addr < PAGE_SIZE) ? "NULL pointer dereference" :
            "paging request", addr);
    show_pte(mm, addr);
    die("Oops", regs, fsr);
...
}

void die(const char *str, struct pt_regs *regs, int err)
{
...
    if (bug_type != BUG_TRAP_TYPE_NONE)
        str = "Oops - BUG";
    if (die(str, err, regs))
        sig = 0;
...
}
```



```
oops_end(flags, regs, sig);
}

static void oops_end(unsigned long flags, struct pt_regs *regs, int signr)
{
    //这里的 sysdump_enter 在 kernel4.4 分支上已经去掉，统一通过下面的 panic 函数调用
sysdump_enter
#ifdef CONFIG_SPRD_SYSDUMP
    sysdump_enter(1, "oops", regs);
#endif
    ...
    panic("Fatal exception");
    ...
}
```

### 1.3.4 BUG\_ON 过程调用

```
#define BUG_ON(condition) do { if (unlikely(condition)) BUG(); } while (0)
```

BUG()根据实现不同，有的会是非法指令有的直接调用 **panic** 函数重启

BUG\_ON 类型的 log 中直接会有行号输出，一般直接分析出错位置逻辑即可。

### 1.3.5 hardlockup&softlock 调用

hardlockup 和 softlock 的实现在 kernel/kernel/watchdog.c。

- **hardlockup** 的原理是检测某 cpu 中断是否长时间被关闭。如果是，则认为发生 **hardlockup**。
- **Softlockup**: 某 cpu 调度发生异常。判据是每个 cpu 上有一个 **watchdog** 线程，如果某 cpu 上此线程长时间得不到调度，说明此 cpu 的抢占被长时间关闭。

发生 **hardlockup** 或者 **softlock** 不一定会导致重启。它依赖于以下的配置：

```
CONFIG_BOOTPARAM_HARDLOCKUP_PANIC_VALUE
CONFIG_BOOTPARAM_SOFTLOCKUP_PANIC_VALUE
```

如果它们被置位，则当它们发生时触发 **panic**，否则只是打印一条警告 log。后续有可能调度又恢复正常，但更大可能是发生定屏死机。

### 1.3.6 Watchdog 调用以及 wdgreboot

上面讲过的 **softlock** 和 **hardlock** 用到了 **watchdog** 线程和定时器。

本节所讲的是另外一种机制。

kernel/drivers/watchdog/sprd\_wdt\_sys.c

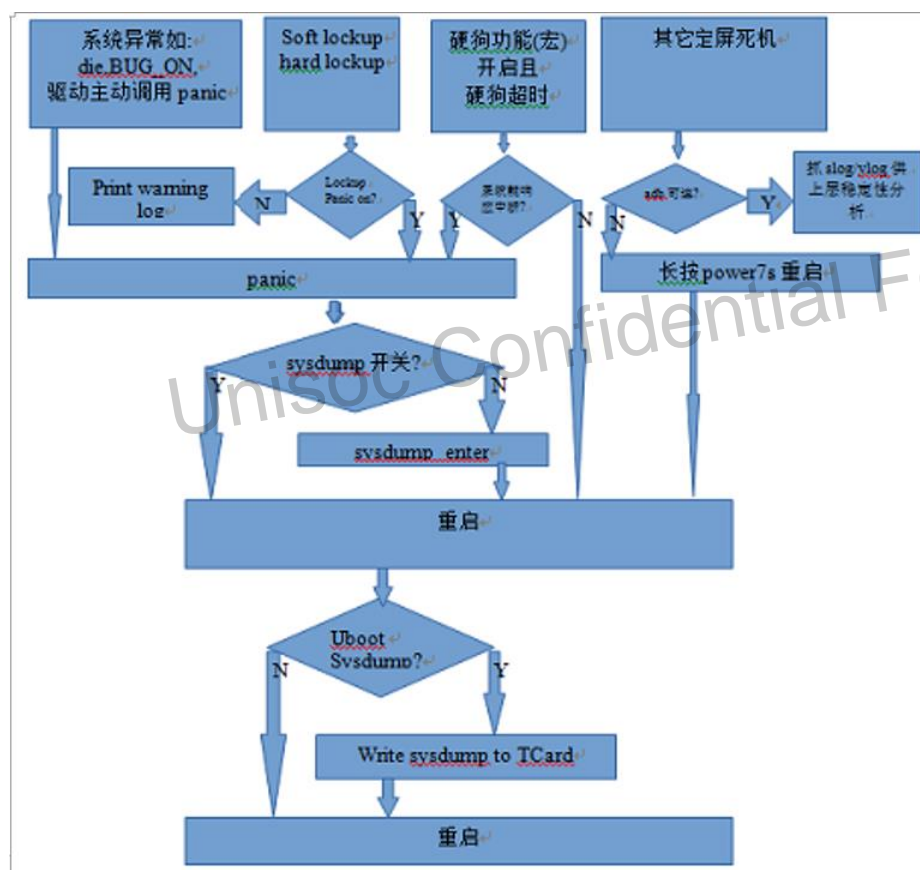
基本思想是有一个喂狗线程(**watchdog\_feeder**)去定期喂狗，并且使能硬狗。狗在特定时间内没被喂的话，则触发硬狗中断，如果此时系统能相应中断的话，则触发 **panic** 调用(重启模式是 **panic**)。如果此时系统已经响应不了中断，当重启倒计时到期，则由硬狗强制重启机器(重启模式是 **ap wdgreboot**)。

# 2 抓取 system dump 流程

## 2.1 system dump 抓取流程

panic 类型的 system dump 会先走 kernel 流程中 `systemdump_enter` 函数中，它将更多相关信息存储起来，以便供后续 uboot 过程进一步存到 T 卡的 `sysdump` 文件中。`watchdog` 超时且系统已经不响应中断时由硬件直接重启后在 uboot 流程中存储 `sysdump` 到 T 卡。长按 `power` 键重启同样由硬件直接重启到 uboot 抓 `sysdump` 文件（根据项目配置不同，可能长按 `power` 键还要外加音量键）。

图2-1 system dump 抓取流程图



对于没有屏的项目，请参见《DUMP2PC 使用指南》。

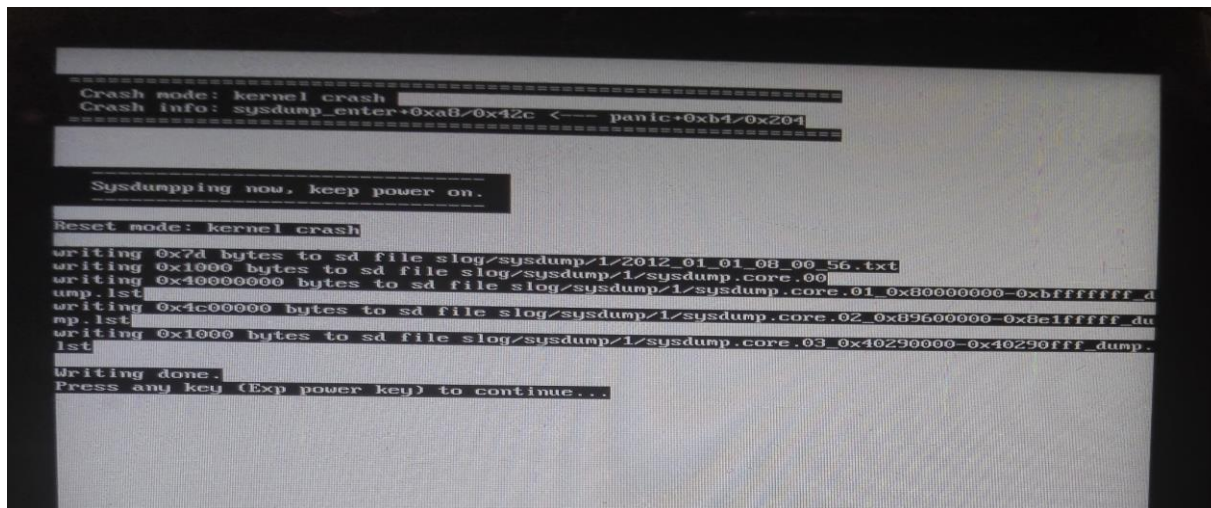
## 2.2 system dump 抓取界面

system dump 抓取完成界面如下图 2-2 所示。此时按下除 `power` 键以外的键即可继续启动机器到 `idle` 界面。

## 说明

此行为只针对有屏的项目。

图2-2 system dump 抓取界面图



## 2.3 sysdump 文件列表

以下为抓取到的 system dump 文件夹内容 (文件个数和名称根据不同平台, 不同 dump 类型可能会有变化), 其中 sysdump.core.\*是真正的 dump 文件组成部分:

- 2017\_09\_30\_14\_56\_50.txt
- sysdump.core.00
- sysdump.core.04\_0x15300000-0x5c157fff\_dump.lst
- sysdump.core.08\_0xe6040000-0xe607ffff\_dump.lst
- sysdump.core.01\_0x00000000-0x007fffff\_dump.lst
- sysdump.core.05\_0x5fd00000-0xbfafffff\_dump.lst
- sysdump.core.09\_0x13900000-0x148fffff\_dump.lst
- sysdump.core.02\_0x0e000000-0x132fffff\_dump.lst
- sysdump.core.06\_0x00800000-0x07fffff\_dump.lst vmlinux
- sysdump-checksum.txt
- sysdump.core.03\_0x14a01000-0x14afffff\_dump.lst
- sysdump.core.07\_0x13300000-0x138fffff\_dump.lst

## 2.4 UDX710 sysdump 位置

UDX710/UDS710 项目 sysdump 在 ylog 中位置:

ylog/modem/md\_memory\_xxx.mem

另外需要留意该 dump 文件不包含 elf head 信息。

# 3 Crash 工具简介

## 3.1 分析 system dump 流程

为了分析 **sysdumpe** 文件，必须有 **crash** 工具，**vmlinux** 文件启动 **crash** 工具的步骤：

步骤 1 把 T 卡上的所有 **sysdumpe** 文件拷贝到工作目录，把它们合成一个文件：

```
cat sysdump.core.0* > all
```

对于 **UDX710/UDS710** 芯片平台的 **dump** 而言，由于不包含 **elf head** 信息，所以需要和用手工制作的头文件合成完整的 **dump**。

假设文件头名字为 **sysdump.core.00**，**UDX710** 的 **dump** 名字为 **md\_memory\_xxx.mem**，按照前面描述，需要使用以下命令进行合成。

```
cat sysdump.core.00 md_memory_xxx.mem > all
```

步骤 2 把和 **sysdumpe** 文件对应的 **vmlinux** 文件也拷贝到相同目录。

步骤 3 **crash** 工具和命令行格式可能会随着平台不同而变化，如：

```
crash arm -m phys base=0x80000000 vmlinux all
crash x86_64 -m phys base=0x34200000 vmlinux all
crash_arm64 -m phys_offset=0x80000000 vmlinux all
```

如果 **vmlinux** 和 **sysdumpe** 文件对应的 **kernel** 版本不符，则 **crash** 工具会报错，此时可以执行以下命令：

```
strings vmlinux | grep gcc
strings all | grep gcc
```

分别得到 **vmlinux** 和 **all(sysdumpe)** 的版本，进而获取正确的 **vmlinux** 以供分析。例如：

```
vmlinux:
Linux version 3.10.65 (root@xxx) (gcc version 4.8 (GCC) ) #2 SMP PREEMPT Wed Jul 26 12: 12: 43
CST 2017
all:
Linux version 3.10.65 (root@xxx) (gcc version 4.8 (GCC) ) #1 SMP PREEMPT Wed Jul 26 16: 00: 24
CST 2017
```

进入 **crash** 工具后可以用 **help** 命令显示出 **crash** 所支持的所有命令：

```
crash_x86_64> help
*          files          mod          search          union
alias      foreach       mount      set              vm
ascii      fuser          net        shal             vmm_log
bt         gdb           p          sig              vtop
btop       help            ps         struct           waitq
compare    ipc            pte        swap             whatis
dev        irq           ptob       sym              wr
dis        kmem         ptov       sys              q
eval       list          rd         task
```

```
exit          log          repeat        timer
extend        mach         runq          tree
```

想看其中某个命令的详细功能和用法可以执行 `help commandxxx`，如：

```
crash x86 64> help dis
NAME
    dis - disassemble

SYNOPSIS
    dis [-rfludxs][-b [num]] [address | symbol | (expression)] [count]

DESCRIPTION
    This command disassembles source code instructions starting (or ending) at
    a text address that may be expressed by value, symbol or expression:
        -r (reverse) displays all instructions from the start of the
            routine up to and including the designated address.
        -f (forward) displays all instructions from the given address
```

----结束

## 3.2 常用的命令介绍

### 3.2.1 显示 log 命令

功能描述：

把 kernel log 最后时间点的数据显示出来。

实例：

```
crash x86 64> log | more
[248859.059313] c3 ion_system_heap_allocate, size: 1785856, time: 6606 us
[248859.062868] c0 ion_system_heap_allocate, size: 3022848, time: 8612 us
[248859.115919] c0 ion_system_heap_allocate, size: 1785856, time: 38396 us
[248859.148993] c2 ion_system_heap_allocate, size: 1785856, time: 5706 us
[248859.167655] c2 [SensorHub]batch_set 925: buf=1 0 200 0

[248859.167681] c2 [SensorHub]batch_set 937: handle = 1, rate = 200, enabled
= 0
[248859.167920] c2 [SensorHub]active_set 902: buf=1 1
```

备注：

可以用 `log > log.txt` 存成文件用 `ue` 或其它文本工具进一步研究。

### 3.2.2 ps 命令

功能描述：

查看系统的线程情况。

实例：

```
crash x86 64> ps | more
  PID   PPID  CPU   TASK                ST  %MEM   VSZ   RSS  COMM
>    0      0   0  ffffffff8220b580  RU   0.0     0     0  [swapper/0]
>    0      0   1  ffff8800bbf367c0  RU   0.0     0     0  [swapper/1]
>    0      0   2  ffff8800bbf50000  RU   0.0     0     0  [swapper/2]
>    0      0   3  ffff8800bbf514c0  RU   0.0     0     0  [swapper/3]
>    0      0   4  ffff8800bbf52980  RU   0.0     0     0  [swapper/4]
>    0      0   5  ffff8800bbf53e40  RU   0.0     0     0  [swapper/5]
>    0      0   6  ffff8800bbf55300  RU   0.0     0     0  [swapper/6]
...
```

备注：

None

### 3.2.3 bt 命令

功能描述：

查看 cpu 上当前线程堆栈。

实例：

```
crash x86 64> bt
PID: 19413 TASK: ffff8800581e3e40 CPU: 1 COMMAND: "Binder: 3613 13"
#0 [ffff88002f60b968] panic at ffffffff81b70625
#1 [ffff88002f60b9e8] oops_end at ffffffff81007a79
#2 [ffff88002f60ba08] no context at ffffffff8103eae4
#3 [ffff88002f60ba60] bad area noseaphore at ffffffff8103edab
#4 [ffff88002f60baa8] bad area noseaphore at ffffffff8103ef2e
#5 [ffff88002f60bab8] do page fault at ffffffff8103f1ae
#6 [ffff88002f60bb10] do page fault at ffffffff8103f5dc
#7 [ffff88002f60bb20] page fault at ffffffff81b827d2
[exception RIP: set_cpus_allowed_ptr+71]
RIP: ffffffff8107e347 RSP: ffff88002f60bbd8 RFLAGS: 00010086
RAX: 00000000d4d4d4d4 RBX: ffff88002f60bce0 RCX: 0000000000015f80
RDX: 0000000000000001 RSI: 0000000000000000 RDI: ffff8800b65cb0a8
```

备注：

None

### 3.2.4 runq 命令

功能描述：

查看 cpu 上运行队列中线程列表。

### 实例：

```
crash x86 64> runq
CPU 0 RUNQUEUE: ffff8800bf615f80
  CURRENT: PID: 0      TASK: ffffffff8220b580  COMMAND: "swapper/0"
  RT PRIO ARRAY: ffff8800bf616118
    [no tasks queued]
  CFS RB ROOT: ffff8800bf616050
    [no tasks queued]
CPU 1 RUNQUEUE: ffff8800bf655f80
  CURRENT: PID: 19413  TASK: ffff8800581e3e40  COMMAND: "Binder: 3613 13"
  RT PRIO ARRAY: ffff8800bf656118
    [no tasks queued]
  CFS RB ROOT: ffff8800bf656050
    [no tasks queued]
```

### 备注：

以上示例 `cpu` 上除了当前正执行线程外，没有其它可运行线程。

## 3.2.5 irq 命令

### 功能描述：

查看中断情况。

### 实例：

```
crash x86 64> irq -s
      CPU0      CPU1      CPU2      CPU3      CPU4      CPU5      CPU6      CPU7
33:         0         0         0         0         0         0         0         0  VPIC
41:    45735    43201    2355         9         0         0         0         0  VPIC sprd serial1
43:    954551   193774   235980   236842         0         0         0         0  VPIC
sprd serial3
49:    1977203   396688   397925   396224         0         0         0         0  VPIC
e7d00000.i2c
50:    243566    56718    57682    58224         0         0         0         0  VPIC
e7e00000.i2c
52:         0         0         0         0         0         0         0         0  VPIC sprd-
mailbox_target
```

### 备注：

None.

## 3.2.6 dataType pointer 命令

### 功能描述：

在知道某个数据类型地址情况下，用可理解形式显示数据类型内容。

实例：

```
crash_x86_64> task_struct ffff8800581e3e40 -x
struct task_struct {
  state = 0x0,
  stack = 0xffff88002f608000,
  usage = {counter = 0x2},
  flags = 0x400140,
  ptrace = 0x0,
  wake_entry = {next = 0x0}, }
```

备注：

None.

### 3.2.7 compare 命令

功能描述：

比较 vmlinux 里的代码段和 dump 里的代码段是否一致。

实例：

```
crash_x86_64> compare
<dump.text>0xfffffffff810039c0: 0x0102b9e08944ea89 0x300f20eac148c000
0x0102b9e88944ea89 0x300f20eac148c000
<dump.text>0xfffffffff810039d0: 0x00b1e9d86348c031 0xfffffeac3c7480000
0x00b1e9d86348c031 0xfffffeac3c7480000
<dump.text>0xfffffffff810039e0: 0x415c415bd88948ff 0x478b48c35d5e415d
0x415c415bd88948ff 0x478b48c35d5e415d
<dump.text>0xfffffffff810039f0: 0x00b84808508b4808 0xf700007fffffffff0
0x00b84808508b4808 0xf700007fffffffff0
<dump.text>0xfffffffff81010200: 0x2e6666aaebfffbff 0x0000000000841f0f
0x2e6666aaebfffbff 0x0000000000841f0f
```

备注：

发现不一致的话可能发生内存 bit 反转，或者内存覆盖，此情况需要检查 ddr 问题。

### 3.2.8 rd 命令

功能描述：

读取某地址开始的多个数据。

实例：

```
crash_x86_64> rd ffff8800581e3e40 4
ffff8800581e3e40: 0000000000000000 ffff88002f608000
ffff8800581e3e50: 0040014000000002 0000000000000000
```



备注：

None。

## 3.3 Crash 常见问题分析

### log 关键字

获取方式：找到相关 sysdump 后，进入 crash 界面把 log 导出来，然后搜索下列关键字：

- oops
- panic
- lockup
- BUG
- watchdog

### BUG\_ON 类型问题

BUG\_ON 类型的 panic 是比较好分析的，因为 log 明确指出了出错函数的位置。如下列所示：

步骤 1 导出 log 信息：

```
[ 4.399536] c0 kernel BUG at /home/jjj/workjjj/C393_SPR_251/ich.code/kernel/kernel/timer.c:732!
[ 4.408294] c0 Internal error: Oops - BUG: 0 [#1] PREEMPT SMP ARM
[ 4.414398] Modules linked in:
[ 4.417419] c0 CPU: 0 PID: 94 Comm: sprd_hotplug Tainted: G      W   3.10.65 #1
[ 4.424987] c0 task: ef2a9100 ti: ee95a000 task.ti: ee95a000
[ 4.430633] c0 PC is at mod_timer+0x80/0x160
[ 4.434906] c0 LR is at icn85xx_ts_interrupt+0x2c/0x68
[ 4.439239] c3 sprdbat: sprdbat_charge_works-start
[ 4.439270] c3 sprdbat: sprdbat_charge_works---vbat_vol 3805, ocv: 3776, bat_current: 117
[ 4.439300] c3 sprdchg get chg cur rawdata * 50+300=450
[ 4.439300] c3 sprdbat: enter sprdbat_auto_switch_cur avg_cur=0, chg_cur=450
[ 4.439300] c3 sprdbat: chg_end_vol_l: 0x105e
[ 4.439971] c0 sprdbat: cv state: 0x88c, item: 0x23
[ 4.439971] c3 sprdbat: chg_cur: 426, chg_ave_current: 53
[ 4.439971] c3 sprdbat: chg_log: time: 4, health: 1, state: 1, stopflags0x: 0, chg_s_time: 2, temp: 200
[ 4.440063] c3 sprdbat: chg_log: chgcur_type: 450, cccv: 27, vchg: 4734, cvstate: 0, cccv_cal: 1
[ 4.495208] c0 pc : [<c002b8dc>]   lr : [<c02ca7f8>]   psr: 60000193
[ 4.495208] c0 sp : ee95be00 ip : 00000000 fp : 00000000
[ 4.507202] c0 r10: c07c24c8 r9 : 00000000 r8 : 00000000
[ 4.512695] c0 r7 : ffff8c90 r6 : 000000e9 r5 : c08123a8 r4 : c08123e0
[ 4.519470] c0 r3 : 00000000 r2 : 00000000 r1 : ffff8c90 r0 : c08123e0
[ 4.526245] c0 Flags: nZCv IRQs off FIQs on Mode SVC 32 ISA ARM Segment kernel
[ 4.533905] c0 Control: 10c5387d Table: 8000406a DAC: 00000015
...
[ 5.317413] c0 [<c002b8dc>] (mod_timer+0x80/0x160) from [<c02ca7f8>]
(icn85xx_ts_interrupt+0x2c/0x68)
```

```
[ 5.326629] c0 [<c02ca7f8>] (icn85xx_ts_interrupt+0x2c/0x68) from [<c00797cc>]
(handle_irq_event_percpu+0x2c/0x168)
[ 5.337036] c0 [<c00797cc>] (handle_irq_event_percpu+0x2c/0x168) from [<c0079944>]
(handle_irq_event+0x3c/0x5c)
[ 5.347137] c0 [<c0079944>] (handle_irq_event+0x3c/0x5c) from [<c007c258>]
(handle_level_irq+0xd8/0xf0)
[ 5.356506] c0 [<c007c258>] (handle_level_irq+0xd8/0xf0) from [<c0079160>]
(generic_handle_irq+0x20/0x30)
...
```

## 步骤 2 找到出错位置相关代码：

```
static inline int
__mod_timer(struct timer_list *timer, unsigned long expires,
bool pending_only, int pinned){
...
}
```

说明调用此处，timer 的结构的功能成员为空，而根据功能逻辑这里不能为 null。为空则触发 BUG\_ON。

## 步骤 3 接下来分析为什么 function 会为 null。

```
BUG_ON(!timer->function);
...
}
```

根据调用栈和相关代码逻辑有如下分析：

```
static irqreturn_t icn85xx_ts_interrupt(int irq, void *dev_id)
=>add_timer(&_st_up_evnet_timer); //add_timer 里调用 mod_timer
static int icn85xx_request_irq(struct icn85xx_ts_data *icn85xx_ts)
err = request_irq(icn85xx_ts->irq, icn85xx_ts_interrupt, IRQ_TYPE_LEVEL_LOW, "icn85xx_ts",
icn85xx_ts); //请求中断
//中断可能很快就来了，进而执行到 icn85xx_ts_interrupt
static int icn85xx_ts_probe(struct i2c_client *client, const struct i2c_device_id *id)
=>
err = icn85xx_request_irq(icn85xx_ts); //这里请求中断，中断请求后很可能马上就相关中断了，最终调用到
mod timer
.....
init_timer(&_st_up_evnet_timer); //这里才初始化 timer
_st_up_evnet_timer.function = icn85xx_force_read_up_event; //这里才填充 function，可能来不及了
```

## 步骤 4 根据分析得出，把 timer 的初始化操作放到 icn85xx\_request\_irq 之前可解决问题。

----结束

## do\_page\_fault -> panic

这种类型的 panic 可能要费不少努力，kernel 遇到了一个非法地址。分析方法如下列所示：

## 步骤 1 导出 log 信息：

```
[248904.928365] c0 BUG: unable to handle kernel paging request at 0000000628dfbca0
...
[248904.928469] c1 task: ffff8800581e3e40 ti: ffff88002f608000 task.ti: ffff88002f608000
[248904.928475] c0 RIP: 0010: [<ffffffffff8107e347>] [<ffffffffff8107e347>]
set cpus allowed ptr+0x47/0x210
[248904.928487] c1 RSP: 0018: ffff88002f60bbd8 EFLAGS: 00010086
```

```
[248904.928493] c1 RAX: 00000000d4d4d4d4 RBX: ffff88002f60bce0 RCX: 0000000000015f80
[248904.928501] c1 RDX: 0000000000000001 RSI: 0000000000000000 RDI: ffff8800b65cb0a8
[248904.928507] c1 RBP: ffff88002f60bc30 R08: 0000000000000001 R09: 0000000000000000
[248904.928513] c1 R10: 0000000000000000 R11: 0000000000000000 R12: 0000000000015f80
[248904.928519] c1 R13: ffff8800b65cb0a8 R14: ffff8800b65ca980 R15: 0000000000015f80
[248904.928554] c1 FS: 00007b865ddb14e8(0000) GS: ffff8800bf640000(0000) knlGS: 00007b866cb04200
[248904.928561] c1 CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[248904.928574] c1 CR2: 0000000628dfbca0 CR3: 00000000aa03e000 CR4: 00000000001026e0
[248904.928605] c1 DR0: 0000000000000007 DR1: 0000000000000001 DR2: 0000000000000006
[248904.928612] c1 DR3: 00000000001026e0 DR6: 0000000080050033 DR7: 0000000000000010
```

步骤2 输入下面命令，得到如下信息：

```
crash_x86_64> bt
PID: 19413 TASK: ffff8800581e3e40 CPU: 1 COMMAND: "Binder: 3613_13"
#0 [ffff88002f60b968] panic at ffffffff81b70625
#1 [ffff88002f60b9e8] oops_end at ffffffff81007a79
#2 [ffff88002f60ba08] no_context at ffffffff8103eae4
#3 [ffff88002f60ba60] __bad_area_nosemaphore at ffffffff8103edab
#4 [ffff88002f60baa8] bad_area_nosemaphore at ffffffff8103ef2e
#5 [ffff88002f60bab8] do_page_fault at ffffffff8103f1ae
#6 [ffff88002f60bb10] do_page_fault at ffffffff8103f5dc
#7 [ffff88002f60bb20] page_fault at ffffffff81b827d2
[exception RIP: __set_cpus_allowed_ptr+71]
RIP: ffffffff8107e347 RSP: ffff88002f60bbd8 RFLAGS: 00010086
RAX: 00000000d4d4d4d4 RBX: ffff88002f60bce0 RCX: 0000000000015f80
RDX: 0000000000000001 RSI: 0000000000000000 RDI: ffff8800b65cb0a8
RBP: ffff88002f60bc30 R8: 0000000000000001 R9: 0000000000000000
R10: 0000000000000000 R11: 0000000000000000 R12: 0000000000015f80
R13: ffff8800b65cb0a8 R14: ffff8800b65ca980 R15: 0000000000015f80
ORIG_RAX: ffffffff8107e347 CS: 0010 SS: 0018
#8 [ffff88002f60bc38] set_cpus_allowed_ptr at ffffffff8107e51b
#9 [ffff88002f60bc48] cpuset_attach at ffffffff810ede27
#10 [ffff88002f60bc90] cgroup_taskset_migrate at ffffffff810e9745
#11 [ffff88002f60bcd0] cgroup_migrate at ffffffff810e9982
#12 [ffff88002f60bd58] cgroup_attach_task at ffffffff810e9a6d
```

步骤3 结合反汇编代码，结构体偏移，寄存器的值推导：

```
crahme/wsys/Build/userdebug/P953F10/kernel/kernel/sched/core.c: 1250
0xffffffff8107e300 <__set_cpus_allowed_ptr>: push %rbp
0xffffffff8107e301 <__set_cpus_allowed_ptr+0x1>: mov %rsp, %rbp
0xffffffff8107e304 sh_x86_64> dis -lx __set_cpus_allowed_ptr:
/<__set_cpus_allowed_ptr+0x4>: push %r15
0xffffffff8107e306 <__set_cpus_allowed_ptr+0x6>: push %r14
0xffffffff8107e308 <__set_cpus_allowed_ptr+0x8>: push %r13
0xffffffff8107e30a <__set_cpus_allowed_ptr+0xa>: push %r12
0xffffffff8107e30c <__set_cpus_allowed_ptr+0xc>: lea 0x728(%rdi), %r13
/*raw_spin_lock_irqsave(&p->pi_lock, *flags) 取task_struct.pi_lock, offset=0x728; rdi 就是 struct
task struct *p; */
0xffffffff8107e313 <__set_cpus_allowed_ptr+0x13>: push %rbx
0xffffffff8107e314 <__set_cpus_allowed_ptr+0x14>: mov %rdi, %r14 //r14 持有 p
0xffffffff8107e317 <__set_cpus_allowed_ptr+0x17>: sub $0x30, %rsp
0xffffffff8107e31b <__set_cpus_allowed_ptr+0x1b>: mov %edx, 0xc(%rsp)
0xffffffff8107e31f <__set_cpus_allowed_ptr+0x1f>: mov %rsi, 0x10(%rsp)
0xffffffff8107e324 <__set_cpus_allowed_ptr+0x24>: mov $0x15f80, %rdx
/home/wsys/Build/userdebug/P953F10/kernel/kernel/sched/sched.h: 1762
```

```

0xffffffff8107e32b <__set_cpus_allowed_ptr+0x2b>:      mov    %rdx, %r12
/home/wsys/Build/userdebug/P953F10/kernel/kernel/sched/sched.h: 1780
0xffffffff8107e32e <__set_cpus_allowed_ptr+0x2e>:      mov    %rdx, %r15
0xffffffff8107e331 <__set_cpus_allowed_ptr+0x31>:      mov    %r13, %rdi
0xffffffff8107e334 <__set_cpus_allowed_ptr+0x34>:      callq  0xffffffff81b800c0
<_raw_spin_lock_irqsave>
0xffffffff8107e339 <__set_cpus_allowed_ptr+0x39>:      mov    %rax, (%rsp)
/home/wsys/Build/userdebug/P953F10/kernel/include/linux/sched.h: 3194
0xffffffff8107e33d <__set_cpus_allowed_ptr+0x3d>:      mov    0x8(%r14), %rax
//task_struct.stack; r14 + 8 =ffff8800b65ca988; rd ffff8800b65ca988: ffff880034840000 => rax 就是 thread_info
/home/wsys/Build/userdebug/P953F10/kernel/kernel/sched/sched.h: 1762
0xffffffff8107e341 <__set_cpus_allowed_ptr+0x41>:      mov    %r12, %rcx
0xffffffff8107e344 <__set_cpus_allowed_ptr+0x44>:      mov    0x10(%rax), %eax
//thread_info.cpu;ffff880034840000+10= ffff880034840000; rd ffff880034840010 : d5d5d5d4d4d4d4d4
0xffffffff8107e347 <__set_cpus_allowed_ptr+0x47>:      add    -0x7dc6ea00(, %rax, 8), %rcx
//00000000d4d4d4d4 x 8 - 7dc6ea00 = 0000000628dfbca0; rax 异常
////////////////////////00000000d4d4d4d4 x 8 - 7dc6ea00 = 0000000628dfbca0
////////////////////////rax 异常
crash_x86_64> rd 0xffff880034840000 300
ffff880034840000:  d4d4d4d4d4d4d4d4 d4d4d4d4d4d4d4d4
ffff880034840010:  d5d5d5d4d4d4d4d4 d7d6d6d6d5d5d5d5
ffff880034840020:  d7d7d7d7d7d7d7d7 d7d7d7d7d7d7d7d7
ffff880034840030:  d8d8d8d7d7d7d7d7 dadad9d9d9d9d9d8
ffff880034840040:  dadadadadadadada dadadadadadadada
ffff880034840050:  dadadadadadadada dadadadadadadada
ffff880034840060:  dadadadadadadada dcdbdbdbdadadada
ffff880034840070:  dcdcdcdcdcdcdcdc ddddddcdcdcdcdcdc
ffff880034840080:  dededededededede dfdfdedededededede
ffff880034840090:  e0e0e0e0e0e0e0df e1e1e0e0e0e0e0e0
ffff8800348400a0:  e2e2e2e2e2e2e2e1 e2e2e2e2e2e2e2e2
ffff8800348400b0:  e2e2e2e3e3e3e3e3 e0e1e1e1e1e1e1e1
ffff8800348400c0:  e0e0e0e0e0e0e0e0 e1e0e0e0e0e0e0e0
ffff8800348400d0:  e2e2e1e1e1e1e1e1 e3e3e3e3e3e3e2e2
ffff8800348400e0:  e3e3e3e3e3e3e3e3 e2e3e3e3e3e3e3e3
...

```

看起来这段内存很有规律。怀疑是内存覆盖。

步骤 4 查看 kernel 最后时间点有很多 audio/video 相关 log，最后找 video team 定位为媒体播放的一个 bug。

----结束

## soft lockup 类型问题

针对 soft lockup 类型问题，分析方法如下列所示：

步骤 1 导出 log 信息：

```

[196922.832097] c0 sensor id: 10, vol: 0xde, temp: 67700
[196922.959680] c2 NMI watchdog: BUG: soft lockup - CPU#2 stuck for 22s! [kworker/2: 0: 2379]
[196922.968189] c0 Modules linked in: mttty marlin2 fm wdtee armtz(O) mali kbase(O) [last
unloaded: sprdwl ng]
[196922.968228] c2 CPU: 2 PID: 2379 Comm: kworker/2: 0 Tainted: G W O 4.4.49+ #1
[196922.968247] c0 Workqueue: events_freezable thermal_zone_device_check

```

```
[196922.968260] c2 task: ffff880011aa94c0 ti: ffff880011fbc000 task.ti: ffff880011fbc000
[196922.968273] c0 RIP: 0010: [<ffffffff810d8be0>] [<ffffffff810d8be0>]
smp_call_function_single+0x90/0x140

crash_x86_64> dis -lx smp_call_function_single
/home/wsys/Build/userdebug/P953F10/kernel/kernel/smp.c: 273
...
/home/wsys/Build/userdebug/P953F10/kernel/kernel/smp.c: 275
0xffffffff810d8b6f <smp_call_function_single+0x1f>: movq $0x0, 0x18(%rsp) //这里 rsp 应该对
csd_stack
0xffffffff810d8b78 <smp_call_function_single+0x28>: movq $0x0, (%rsp)
0xffffffff810d8b80 <smp_call_function_single+0x30>: movq $0x0, 0x8(%rsp)
0xffffffff810d8b89 <smp_call_function_single+0x39>: movq $0x0, 0x10(%rsp)
0xffffffff810d8b92 <smp_call_function_single+0x42>: movl $0x3, 0x18(%rsp)
...
/home/wsys/Build/userdebug/P953F10/kernel/include/linux/compiler.h: 218
0xffffffff810d8bd5 <smp_call_function_single+0x85>: mov 0x18(%rsp), %eax //RSP: 0000:
ffff880011fbf808 should be call_single_data
/home/wsys/Build/userdebug/P953F10/kernel/kernel/smp.c: 110
0xffffffff810d8bd9 <smp_call_function_single+0x89>: test $0x1, %al
0xffffffff810d8bdb <smp_call_function_single+0x8b>: je 0xffffffff810d8be9
<smp_call_function_single+0x99>
0xffffffff810d8bdd <smp_call_function_single+0x8d>: nop
/home/wsys/Build/userdebug/P953F10/kernel/arch/x86/include/asm/processor.h: 562
0xffffffff810d8bde <smp_call_function_single+0x8e>: pause
/home/wsys/Build/userdebug/P953F10/kernel/include/linux/compiler.h: 218
0xffffffff810d8be0 <smp_call_function_single+0x90>: mov 0x18(%rsp), %edx

-----
108static void csd_lock_wait(struct call_single_data *csd)
109{
110 while (smp_load_acquire(&csd->flags) & CSD_FLAG_LOCK)
111     cpu_relax();
112}

-----
560static always inline void rep_nop(void)
561{
562 asm volatile("rep; nop" : : : "memory");
563}
271int smp_call_function_single(int cpu, smp_call_func_t func, void *info,
272                             int wait)
273{
302 if (wait)
303     csd_lock_wait(csd); //应该是执行到这里挂住
}

RSP: 0000: ffff880011fbf808 //它对应 275
struct call_single_data csd_stack = { .flags = CSD_FLAG_LOCK | CSD_FLAG_SYNCHRONOUS };
crash_x86_64> call_single_data ffff880011fbf808
struct call_single_data {
    llist = {
        next = 0x0
    },
    func = 0xffffffff81410210 <__rdmsr_on_cpu>,
    info = 0xffffffff880011fbf858,
```

```
flags = 3    //这里是 3 导致一直挂在 csd_lock_wait 里
}
```

步骤 2 分析代码逻辑，发现 cpu2 发送的服务请求挂到 percpu 变量 call\_single\_queue 里面。

```
crash_x86_64> call_single_queue
PER-CPU DATA TYPE:
  struct llist_head call_single_queue;
PER-CPU ADDRESSES:
[0]: ffff8800bf617240
[1]: ffff8800bf657240
[2]: ffff8800bf697240
[3]: ffff8800bf6d7240
[4]: ffff8800bf717240    //cpu4
[5]: ffff8800bf757240
[6]: ffff8800bf797240
[7]: ffff8800bf7d7240
```

只有 cpu4 的 call\_single\_queue 里有 value，所以推测是 cpu2 发到 cpu4 了

```
crash_x86_64> runq -c 4
CPU 4 RUNQUEUE: ffff8800bf715f80
CURRENT: PID: 0    TASK: ffff8800bbf52980
COMMAND: "swapper/4"
RT PRIO ARRAY: ffff8800bf716118
[no tasks queued]
CFS RB ROOT: ffff8800bf716050
[120] PID: 7677    TASK: ffff8800738514c0    COMMAND: "m.android.music"

crash_x86_64> bt ffff8800bbf52980
PID: 0    TASK: ffff8800bbf52980    CPU: 4    COMMAND: "swapper/4"
#0 [ffff8800bbf63e60] __schedule at ffffffff81b79c02
#1 [ffff8800bbf63e78] cpuidle_enter_state at ffffffff8176738c
#2 [ffff8800bbf63ec0] cpuidle_enter at ffffffff81767932
#3 [ffff8800bbf63ed0] call_cpuidle at ffffffff81099856
#4 [ffff8800bbf63ee8] cpu_startup_entry at ffffffff81099b46
#5 [ffff8800bbf63f38] start_secondary at ffffffff81030a50
```

步骤 3 cpu2 需要 cpu4 服务，cpu4 却长时间在休眠状态。这种问题需要对主管任务调度模块进行深入分析。

----结束

## 存取寄存器挂死问题

此节示范一个死机后手动 dump 的分析过程。当从 sysdump 的 kernel log 里面看不出什么异常线索下，可以看看程序是否挂死在某个寄存器存取操作上。分析前先介绍下面关键全局变量：

```
struct sprd_debug_regs_access *sprd_debug_last_regs_access;
```

### 说明

如果 crash 工具中没有此变量，说明此项目不支持这个机制。

它表示系统最后一次存取寄存器是否完成的信息。此结构的信息如下：

```
crash_arm> sprd_debug_regs_access -ox
struct sprd_debug_regs_access {
    [0x0] unsigned long vaddr; //表示此寄存器虚拟地址
    [0x4] unsigned long stack;
    [0x8] unsigned long pc;
    [0xc] unsigned long time;
    [0x10] unsigned int status; //寄存器操作有没有完成, 1: 完成, 0: 没有完成
    [0x14] u32 value;
}
SIZE: 0x18 //此结构体大小是 0x18 字节, 不同体系结构大小不一样.
```

每个 cpu 都有一个此结构体。下面举例说明：

步骤 1 输入下面命令，得到如下信息：

```
crash_arm> sprd_debug_last_regs_access
sprd debug last regs access = $1 = (struct sprd debug regs access *) 0xf00da000
此问题基于 4 核架构
cpu0 对应的结构体地址为 0xf00da000
cpu1 对应的结构体地址为 0xf00da000+0x18=0xf00da018
cpu2 对应的结构体地址为 0xf00da018+0x18=0xf00da030
cpu3 对应的结构体地址为 0xf00da030+0x18=0xf00da048
Cpu0:
crash_arm > sprd_debug_regs_access 0xf00da000 -x
struct sprd_debug_regs_access {
    vaddr = 0xf007e030,
    stack = 0xe31dbe1c,
    pc = 0xc0011b54,
    time = 0xffff97a3,
    status = 0x1,
    value = 0x0
}
Cpu1:
crash_arm> sprd_debug_regs_access 0xf00da018 -x
struct sprd_debug_regs_access {
    vaddr = 0xf0016004,
    stack = 0xed32ddf8,
    pc = 0xc045fd60,
    time = 0xffff97a3,
    status = 0x0, //只有 cpu1 的寄存器操作没有完成，可能是存取此寄存器时被挂住了
    value = 0x0
}
Cpu2:
crash_arm> sprd_debug_regs_access 0xf00da030 -x
struct sprd debug regs access {
    vaddr = 0xf007e030,
    stack = 0xe32a9e0c,
    pc = 0xc0011b54,
    time = 0xffff97a3,
    status = 0x1,
    value = 0x0
}
Cpu3:
crash_arm> sprd_debug_regs_access 0xf00da048 -x
struct sprd debug regs access {
    vaddr = 0xf007e030,
    stack = 0xe327dde4,
```

```
pc = 0xc0011b54,
time = 0xfffff97a3,
status = 0x1,
value = 0x0
}
```

从 `cpu` 的 `status` 值分析得出, `cpu1` 值异常, 怀疑 `cpu1` 寄存器被挂住了。

步骤 2 转换 `cpu1` 的地址为物理地址并查芯片手册找到具体寄存器。

```
crash32> vtop 0xf0016004 //此命令把虚拟地址转化为物理地址
VIRTUAL PHYSICAL
F0016004 60100004 //查芯片手册, 此地址对应一个 gpu 模块寄存器
PAGE DIRECTORY: c0004000
PGD: c0007c00 => ac00a811
PMD: c0007c00 => ac00a811
PTE: ac00a058 => 60100653
PAGE: 60100000
PTE PHYSICAL FLAGS
60100653 60100000 (PRESENT|DIRTY|YOUNG|WRITE)
```

步骤 3 此问题转 `gpu` 模块分析, 后由 `gpu` 模块解决。

----结束

## 中断过多问题

此节是另外一例死机手动 `dump` 的分析过程。当死机发生后, 可以查看系统的中断情况。下面举例说明:

步骤 1 输入下面命令, 得到如下信息:

```
crash_arm> irq -s
CPU0
52: 0 GIC sprd codec dp
60: 50098 GIC local timer
...
168: 118962 irq-d-gpio sci gpio
169: 1433573 irq-d-gpio sci gpio
170: 281678 irq-d-gpio sci gpio
171: 280478 irq-d-gpio sci gpio
```

从 `log` 中发信时钟中断发生了 50098 次, 而几个 `gpio` 中断发生的次数远大于时钟中断, 这是很不正常的。

步骤 2 查看一下这些中断属于哪个模块。

```
crash_arm> irq 169
IRQ IRQ_DESC/_DATA IRQACTION NAME
169 c0844080 dc4f7b80 "sci_gpio"
crash_arm> irqaction dc4f7b80 -x
struct irqaction {
    handler = 0xc0306b50 <sci_keypad_gpio_isr>,
    dev_id = 0xc0907ff0 <GPIO_COLS+8>,
    percpu_dev_id = 0x0,
    next = 0x0,
    thread_fn = 0x0,
```



```
thread = 0x0,  
irq = 0xa9,  
flags = 0x4008,  
thread_flags = 0x0,  
thread_mask = 0x0,  
name = 0xc073206d "sci_gpio",  
dir = 0xdc4e9b00  
}
```

发现这是一个键盘驱动，问题转到键盘驱动团队研究解决。

----结束

Unisoc Confidential For hiar

# 4 串口 log 抓取方法

有些项目没有 T 卡接口或者在开机过程 **sysdump** 模块还没有初始化时就定屏或者重启。有的干脆就还没有进入 **kernel** 流程，**uboot** 阶段就出了问题等。这些情况下不方便抓取 **sysdump**，此时可以抓取串口 **log** 分析。

对于休眠后无法唤醒，**adb** 口不响应。可以接上串口，动态观察按键后，串口 **log** 窗口什么反应。结合 **sysdump** 一起分析。由于优先级低的 **log** 在串口上没有输出，所以需要调整串口 **log level** 设置。可以通过代码修改或者动态修改。下面就是 2 种不同修改方法：

- 修改代码：

代码位置：项目相关的 **dts** 文件，如 **kernel/arch/arm/boot/dts/xxxx.dts**：(xxxx 的具体名称根据项目实际来)。

```
bootargs = "loglevel=1 console=ttyS...
```

- 上述 **loglevel=1** 调整为 **loglevel=7** 或者 **loglevel=8**。

- 修改完成后编译 **bootimage** 并更新到手机中。

- 动态修改：

- 手机运行过程中动态修改方法：

```
echo 7 > /proc/sys/kernel/printk //此修改仅仅对本次开机有效，重启后失效。
```

- 对于开机过程很早就出问题的情况，只能通过修改代码来修改 **log level**。

## 说明

在某些情况下，**loglevel** 等级变高导致跟踪数据增多，会引起流程异常。

另外由于休眠后，默认情况下 **printk** 中会使能控制台休眠，**log** 就不会输出到控制台（即当前设备的串口）。为了在休眠时可以看到更多 **log** 信息，需要 **disable** 控制台的休眠。

**Disable** 控制台休眠可以通过代码修改或者动态修改来实现。下面就是 2 种不同修改方法：

- 修改代码：

在 **bootargs** 中增加字段 “**no\_console\_suspend**”

```
bootargs = "loglevel=1 no_console_suspend
```

编译并替换 **bootimage**。

- 动态修改：

```
echo 0 > /sys/module/printk/parameters/console_suspend
```

# 5

## 参考文档

---

1. 《DUMP2PC 使用指南》

Unisoc Confidential For hiar