目 录

[第1章 WDT硬件设计 1](#_Toc448141707)

[1.1 WDT原理 1](#_Toc448141708)

[1.1.1 原理 1](#_Toc448141709)

[1.1.2 种类 1](#_Toc448141710)

[1.2 Mediatek WDT硬件设计 1](#_Toc448141711)

[1.2.1 结构 1](#_Toc448141712)

[1.2.2 dual mode 2](#_Toc448141713)

[第2章 WDT软件设计 4](#_Toc448141714)

[2.1 设计原理 4](#_Toc448141715)

[2.2 WDT驱动 5](#_Toc448141716)

[2.2.1 WDT驱动分布 5](#_Toc448141717)

[2.2.2 kernel WDT驱动 5](#_Toc448141718)

[2.2.3 睡眠/唤醒 5](#_Toc448141719)

[2.2.4 重启 6](#_Toc448141720)

[2.2.5 喂狗函数 6](#_Toc448141721)

[2.2.6 开关WDT 6](#_Toc448141722)

[2.2.7 结语 6](#_Toc448141723)

[2.3 WDK驱动 6](#_Toc448141724)

[2.3.1 WDT驱动 6](#_Toc448141725)

[2.3.2 对外接口 6](#_Toc448141726)

[2.3.3 初始化 7](#_Toc448141727)

[2.3.4 喂狗 8](#_Toc448141728)

[2.3.5 CPU hotplug 8](#_Toc448141729)

[2.4 超时触发流程 9](#_Toc448141730)

[2.4.1 32bit kernel FIQ触发流程 9](#_Toc448141731)

[2.4.2 64bit kernel FIQ触发流程 11](#_Toc448141732)

[2.4.3 FIQ step 13](#_Toc448141733)

[2.4.4 结语 13](#_Toc448141734)

[第3章 分析方法 14](#_Toc448141735)

[3.1 HWT 14](#_Toc448141736)

[3.1.1 开机30s超时 14](#_Toc448141737)

[3.1.2 kernel运行过程中30s超时 14](#_Toc448141738)

[3.2 HW REBOOT 15](#_Toc448141739)

[3.2.1 HW reboot原因 15](#_Toc448141740)

[3.2.2 HW reboot调试信息 15](#_Toc448141741)

[3.2.3 分析方法 18](#_Toc448141742)

[3.2.4 结语 19](#_Toc448141743)

[第4章 案例分析 20](#_Toc448141744)

[4.1 HWT-kernel初始化时间长(1) 20](#_Toc448141745)

[4.1.1 基本信息 20](#_Toc448141746)

[4.1.2 分析过程 20](#_Toc448141747)

[4.1.3 解决方法 21](#_Toc448141748)

[4.1.4 结语 21](#_Toc448141749)

[4.2 HWT-kernel初始化时间长(2) 21](#_Toc448141750)

[4.2.1 基本信息 21](#_Toc448141751)

[4.2.2 分析过程 21](#_Toc448141752)

[4.2.3 解决方法 23](#_Toc448141753)

[4.3 HWT-lk执行时间长 23](#_Toc448141754)

[4.3.1 基本信息 23](#_Toc448141755)

[4.3.2 分析过程 23](#_Toc448141756)

[4.3.3 解决方法 25](#_Toc448141757)

[4.4 HWT-中断频繁触发 25](#_Toc448141758)

[4.4.1 基本信息 25](#_Toc448141759)

[4.4.2 分析过程 25](#_Toc448141760)

[4.4.3 解决方法 29](#_Toc448141761)

[4.4.4 结语 29](#_Toc448141762)

[4.5 HWT-大量DEVAPC violation 29](#_Toc448141763)

[4.5.1 基本信息 29](#_Toc448141764)

[4.5.2 分析过程 30](#_Toc448141765)

[4.5.3 解决方法 31](#_Toc448141766)

[4.5.4 结语 31](#_Toc448141767)

[4.6 HWT-SWT卡死 31](#_Toc448141768)

[4.6.1 基本信息 31](#_Toc448141769)

[4.6.2 分析过程 31](#_Toc448141770)

[4.6.3 解决方法 33](#_Toc448141771)

[4.6.4 结语 33](#_Toc448141772)

[4.7 HWT-死锁卡死 34](#_Toc448141773)

[4.7.1 基本信息 34](#_Toc448141774)

[4.7.2 分析过程 34](#_Toc448141775)

[4.7.3 解决方法 37](#_Toc448141776)

[4.7.4 结语 37](#_Toc448141777)

[4.8 HWT-死循环卡死 37](#_Toc448141778)

[4.8.1 基本信息 37](#_Toc448141779)

[4.8.2 分析过程 37](#_Toc448141780)

[4.8.3 解决方法 40](#_Toc448141781)

[4.8.4 结语 41](#_Toc448141782)

[4.9 HWT-CPU卡死 41](#_Toc448141783)

[4.9.1 基本信息 41](#_Toc448141784)

[4.9.2 分析过程 41](#_Toc448141785)

[4.9.3 解决方法 42](#_Toc448141786)

[4.9.4 结语 42](#_Toc448141787)

[4.10 HWT-rt throttle引起卡死 42](#_Toc448141788)

[4.10.1 基本信息 42](#_Toc448141789)

[4.10.2 分析过程 42](#_Toc448141790)

[4.10.3 解决方法 45](#_Toc448141791)

[4.10.4 结语 45](#_Toc448141792)

[4.11 HW REBOOT-HW故障 45](#_Toc448141793)

[4.11.1 基本信息 45](#_Toc448141794)

[4.11.2 分析过程 45](#_Toc448141795)

[4.11.3 解决方法 47](#_Toc448141796)

[4.11.4 结语 47](#_Toc448141797)

[第5章 附录 48](#_Toc448141798)

[5.1 相关FAQ 48](#_Toc448141799)

# WDT硬件设计

## WDT原理

### 原理

 在手机Soc Chip中，里面的AP跑着linux操作系统软件，而任何软件都可能存在各种问题，如果遇到了这些异常，软件可能陷入死循环，导致手机变成“砖头”，如果没有其他硬件辅助，那么只能断电（拔电池）然后重新开机才行。为了避免出现这种情况，芯片内部增加了一个看门狗模块，这个模块专门检测CPU运行状态，只要出现卡死就复位系统。

WDT全称是watchdog timer，就是看门狗模块，看门狗其实就是一个可以在一定时间内被复位的计数器。当看门狗启动后，计数器开始自动计数，经过一定时间，如果计数没有被复位，计数器达到指定数值就会发出复位信号，很多设备包括CPU接到这个信号而复位重启（俗称“被狗咬”）。为了保证看门狗不发出复位信号，就需要在看门狗允许的时间间隔内对看门狗计数器清零（俗称“喂狗”），计数器重新计数。如果系统正常并保证按时“喂狗”，那么就相安无事。一旦程序故障卡死，没有“喂狗”，系统“被咬”复位。

### 种类

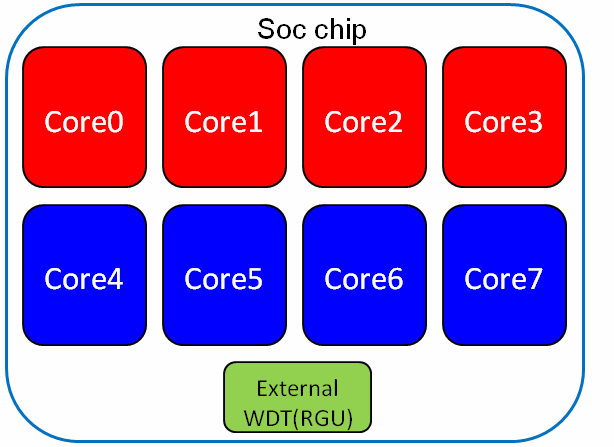
   嵌入式系统中主要可以分为两种类型的看门狗：

* Soc芯片内部集成WDT，这是Soc常用的设计。当然PC上可能用独立的看门狗芯片。
* 软件模拟看门狗，只要有个timer就可以模拟。

## Mediatek WDT硬件设计

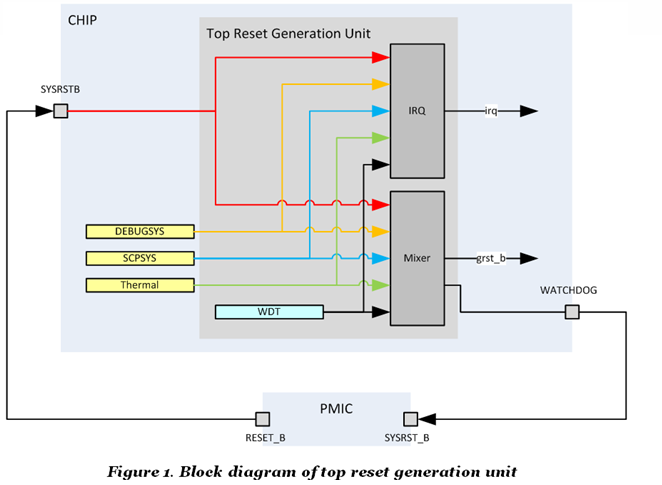
### 结构

在现有Soc芯片中，只有1个看门狗硬件模块，这个模块被包含在一个叫RGU的模块里。在早期平台中，比如MT6577，MT6575除了RGU里的WDT外，每个CPU都有一个local WDT，可以保证每个CPU不卡死。而MT6577之后就只有RGU里的WDT了，如图：



在只有一个WDT的情况下如何保证每个CPU不卡死呢？这里需要软件设计使它都可以喂狗才行。软件设计部分将在下一章节介绍。

     RGU全称是reset generation unit，在datasheet有如下框图：



可以看到WDT在RGU里只是其中一个模块，还有其他模块可以产生复位信号，比如thermal，当温度过高会触发IRQ或直接复位，这也是一种硬件保护措施。复位信号经过Mixer后会分出2个信号，grst\_b会复位芯片内部模块，还有1个信号通过芯片管脚WATCHDOG复位外部芯片。有时为了debug，也可以测量WATCHDOG pin脚（正常是高电平，有复位信号时低电平），看这次重启是否是WDT等触发的。

### dual mode

 WDT功能，大家可以看datasheet里的寄存器介绍，比如包含了

* WDT\_MODE：功能开关
* WDT\_LENGTH：超时计数
* WDT\_RESTART：写该寄存器触发计数清0
* WDT\_STA：状态
* WDT\_INTAVEL：reset信号长度
* WDT\_SWRST：写该寄存器直接触发复位

    这里介绍一个dual mode功能，通常CPU发生卡死，我们需要知道卡死位置，分析原因，然后改进，而不希望直接复位。设计的方法是：WDT超时后先不发复位信号，而是先送出IRQ（如上面的框图），WDT IRQ通常配置成CPU的FIQ，如果CPU只是软件卡死（内核死锁，中断过多等），会响应该FIQ，然后我们在FIQ里收集异常信息和CPU寄存器信息，然后再走正常的panic流程。重启后我们就有信息分析此次WDT timeout的原因了。为了保证IRQ发出后，CPU不卡死，WDT再次计时，如果在panic流程又卡死了，就会由WDT发出复位信号复位。

    简单讲：WDT超时先发出IRQ，WDT重新计时，如果再次超时WDT直接发出复位信号。可以看到WDT超时后分2个阶段，第1阶段发出IRQ用于收集异常信息，第2阶段直接复位。通常我们在第1阶段就完成异常信息收集并通过WDT\_SWRST寄存器完成复位，而不用等到第2阶段的。这种情况我们称为**HWT**（Hardware Watchdog Timeout）

    事实上任何事情总有意外，比如CPU因为PMIC提供的电压低，或硬件故障等原因导致WDT超时发出的IRQ在CPU端得不到响应，WDT只能通过第2阶段复位芯片。这种情况我们称为**HW reboot**。HW reboot通常和硬件有较大关系。比如：

* 硬件故障，电压或频率异常
* 总线卡死

# WDT软件设计

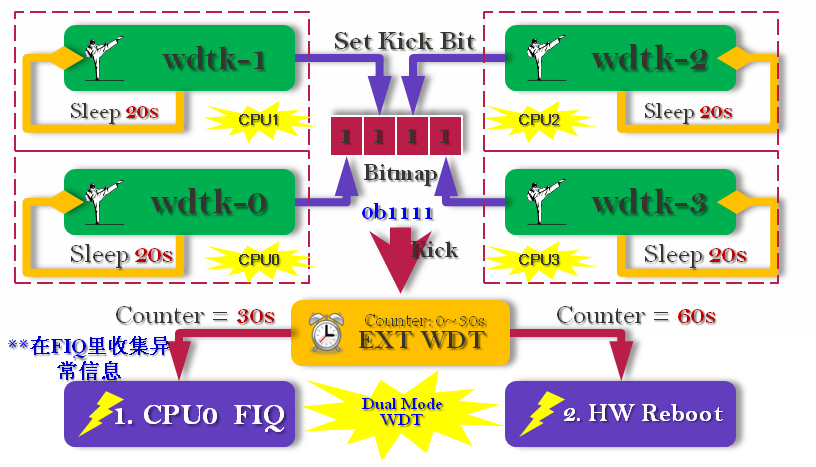
## 设计原理

  上一章节有讲Soc芯片内部只有1个WDT模块，但CPU有多颗，为了保证每个CPU卡死都可以顺利复位，就必须借助软件。WDT本质是个计数器并且支持dual mode，目前我们将超时时间（第1阶段超时）设定为**30s**，在发出IRQ之后超时时间（第2阶段超时）不可设定，各个平台可能不一样，一般在30s～60s之间。

    配置好后，还需要设计喂狗模块，这是重中之重。流程如下：

* bit位代表CPU的喂狗状态：CPU核数每个平台都不太一样，我们用bit位代表1个CPU，比如4个CPU就用4个bit，我们用kick\_bit变量表示。
* 喂狗进程负责喂狗：在喂狗模块初始化时针对每个CPU创建的对应的内核实时进程（优先级最高）并绑定到对应的CPU上（也就是说这个进程只运行在指定CPU上，不会发生迁移），这些进程喂狗（其实就是将kick\_bit对应的bit位置1而已，并没有真正喂狗）后进入睡眠，喂狗间隔是**20s**，如此循环反复。
* 真正喂外部WDT狗：任何一个喂狗进程在喂狗之后，会检查kick\_bit变量，如果该值等于cpus\_kick\_bit变量（这个变量的每个bit代表对应CPU online情况，如果CPU online，对应bit置1），表示所有CPU都已喂狗，这时才真正喂外部WDT狗，同时将kick\_bit清0，进入下一轮喂狗流程，如此循环反复。

    用一张框图总结上面的流程（以4核CPU为例）：



 这里的wdtk-x就是对应CPU的喂狗进程。以上的一套设计可以保证各个CPU卡死都可以通过看门狗复位。

    喂狗间隔是20s，而超时时间是30s，也就是说最长能容忍卡住的时间是10s（卡一小会还是可以的），超过这个时间，系统就会复位了。这里还有问题，由于喂狗进程之间没有同步，是否有可能存在刚开始一起喂狗，之后渐渐出现先后呢？误差肯定有的，但在任何30s时间里，喂狗进程都会喂1次狗的（因为喂狗间隔20s，每个CPU肯至少会喂1次狗）。而只要CPU卡死超过10s就复位了。

    多核CPU不可能一直都是online，系统会根据负载做hotplug，在某颗CPU power down/power up时，会更新cpus\_kick\_bit变量并将kick\_bit变量清0，同时喂外部WDT狗。

    当只剩1个CPU时，并且该CPU要进入睡眠，此时kernel进入suspend状态，WDT模块当然也要关闭的，唤醒时再重启开启。那睡眠后的流程卡死怎么办？那已超出WDT管辖的范围了，需要用其他硬件保证，不在该文章讨论范围。

## WDT驱动

### WDT驱动分布

android系统中，启动分好几个阶段：preloader、lk、kernel、android。因此每个阶段都要配置WDT，才能保证不卡死。这里我们介绍kernel阶段的WDT驱动，其他阶段是类似的。

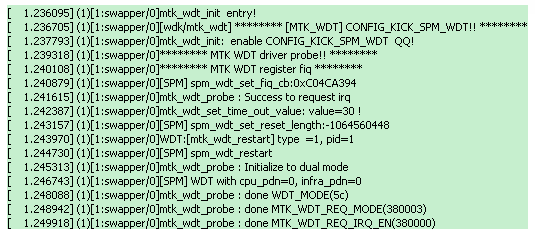
### kernel WDT驱动

流程是：[module\_init](http://10.16.20.15/l1mp6/s?defs=module_init&project=kernel-3.10)([mtk\_wdt\_init](http://10.16.20.15/l1mp6/s?refs=mtk_wdt_init&project=kernel-3.10)) => [mtk\_wdt\_init](http://10.16.20.15/l1mp6/s?refs=mtk_wdt_init&project=kernel-3.10)() => [mtk\_wdt\_probe](http://10.16.20.15/l1mp6/s?refs=mtk_wdt_probe&project=kernel-3.10)()。

    在[mtk\_wdt\_probe](http://10.16.20.15/l1mp6/s?refs=mtk_wdt_probe&project=kernel-3.10)()函数里，做了如下几件事：

* 注册WDT IRQ处理函数[wdt\_fiq](http://10.16.20.15/l1mp6/xref/kernel-3.10/drivers/misc/mediatek/wdt/mt6572/mtk_wdt.c#wdt_fiq)()，kernel本身是没有使用FIQ的，因此WDT FIQ发生时一定会被处理，除非FIQ被关闭或硬件故障了。由于64bit kernel默认支持ATF（ARM Trusted Firmware），FIQ为security interrupt，kernel是无法处理security interrupt的，因此FIQ会路由到ATF里处理。在后面的流程中会详细介绍。
* 设定WDT超时时间30s。
* 调用[mtk\_wdt\_mode\_config](http://10.16.20.15/l1mp6/s?defs=mtk_wdt_mode_config&project=kernel-3.10)()初始化WDT，这里打开的dual mode。

这样就完成WDT的初始化了，WDT初始化是比较早的，基本上3s内就完成了（kernel时间）。以下是72KK版本的uart log中搜索wdt结果：

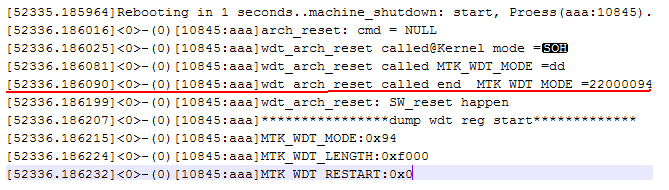


### 睡眠/唤醒

在系统进入睡眠时，WDT要被关闭，不过不是在WDT suspend里关闭WDT，因为suspend不是睡眠最后的步骤。这里会在sleep模块通过调用WDK的wd\_suspend\_notify()来间接调用mtk\_wd\_suspend()来关闭WDT的。同理唤醒时也是早于resume流程就被sleep模块通过WDK的resume\_notify()调用mtk\_wd\_resume()重新启动WDT。

### 重启

系统重启（比如adb reboot）是通过WDT来完成复位的，在驱动里提供了[wdt\_arch\_reset](http://10.16.20.15/l1mp6/s?refs=wdt_arch_reset&project=kernel-3.10)()函数，这个函数通过写WDT\_SWRST寄存器完成软件复位。基本上重启的log都长这样：



  红线以上是正常的log，写完WDT\_SWRST寄存器基本就复位了，但有些平台并不是立即生效，需要过几百毫秒才完成复位，因此有可能看到红线以下的log，这是正常的。

### 喂狗函数

  WDT启动后就开始计时了，如果没人喂狗就会触发FIQ，然后在硬复位。因此需要提供了[**mtk\_wdt\_restart**](http://10.16.20.15/l1mp6/s?refs=mtk_wdt_restart&project=kernel-3.10)()函数，调用一次就将计数清0。

    如果有驱动需要长时间关中断运行，比如开机时的TP固件升级，就需要在里面添加喂狗动作，防止HWT。

### 开关WDT

有时为了调试，我们需要关闭看门狗，驱动里也提供了开启/关闭函数：[mtk\_wdt\_enable](http://10.16.20.15/l1mp6/s?refs=mtk_wdt_enable&project=kernel-3.10)()。如果你要关闭WDT，可以直接在代码里调用这个函数，那么WDT就永久关闭了。

    在WDK驱动里还提供了用户接口，可以很方便在adb shell里关闭/打开WDT。

### 结语

WDT驱动比较简单，基本都是控制WDT硬件寄存器达到所需效果。重点还在于WDK驱动(喂狗模块)。

## WDK驱动

### WDT驱动

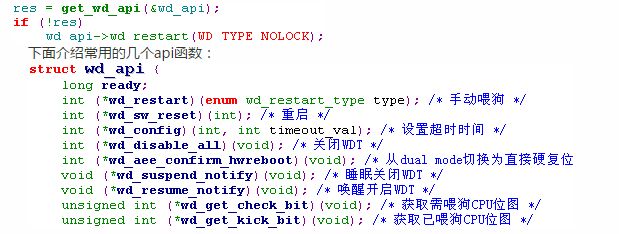
wdk是喂狗模块，由于只有一个WDT，而需要保护多个CPU不卡死，根据前面设计的原理来设计wdk驱动，驱动具体位置：

    kernel-3.10/drivers/misc/mediatek/wdk，里面主要有wd\_common\_drv.c/wd\_api.c

    wd\_api.c负责对外接口，wd\_common\_drv.c负责喂狗工作。

### 对外接口

[**get\_wd\_api**](http://10.16.20.15/l1mp6/s?refs=get_wd_api&project=kernel-3.10)()可以获取到通用的wdk api结构体，里面提供了WDT和WDK所需的操作。使用方法如下(手动喂狗)：



### 初始化

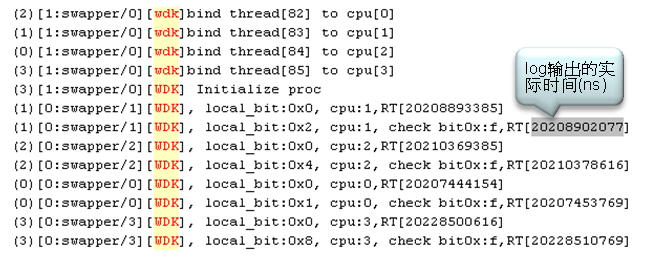
流程1：[arch\_initcall](http://10.16.20.15/l1mp6/s?defs=arch_initcall&project=kernel-3.10)(**[init\_wk\_check\_bit](http://10.16.20.15/l1mp6/s?refs=init_wk_check_bit&project=kernel-3.10)**) => [**init\_wk\_check\_bit**](http://10.16.20.15/l1mp6/s?refs=init_wk_check_bit&project=kernel-3.10)() => [**wk\_cpu\_update\_bit\_flag**](http://10.16.20.15/l1mp6/s?refs=wk_cpu_update_bit_flag&project=kernel-3.10)()。完成cpus\_kick\_bit、kick\_bit这2个重要变量初始化。cpus\_kick\_bit表示需喂狗CPU位图，kick\_bit表示已喂狗CPU位图，每个bit对应CPU。

    流程2：[late\_initcall](http://10.16.20.15/l1mp6/s?defs=late_initcall&project=kernel-3.10)(**[init\_wk](http://10.16.20.15/l1mp6/s?refs=init_wk&project=kernel-3.10)**) => [**init\_wk**](http://10.16.20.15/l1mp6/s?refs=init_wk&project=kernel-3.10)() => [**mtk\_rgu\_sysfs**](http://10.16.20.15/l1mp6/s?refs=mtk_rgu_sysfs&project=kernel-3.10)()、[**start\_kicker\_thread\_with\_default\_setting**](http://10.16.20.15/l1mp6/s?refs=start_kicker_thread_with_default_setting&project=kernel-3.10)()、[**wk\_proc\_init**](http://10.16.20.15/l1mp6/s?refs=wk_proc_init&project=kernel-3.10)()、[register\_cpu\_notifier](http://10.16.20.15/l1mp6/s?defs=register_cpu_notifier&project=kernel-3.10)(&[cpu\_nfb](http://10.16.20.15/l1mp6/s?defs=cpu_nfb&project=kernel-3.10))。

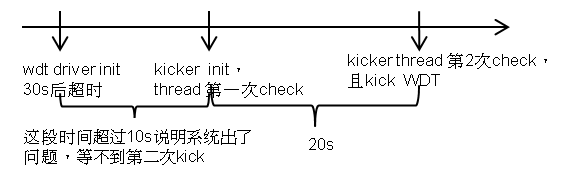
* 通过[**mtk\_rgu\_sysfs**](http://10.16.20.15/l1mp6/s?refs=mtk_rgu_sysfs&project=kernel-3.10)()创建了sysfs文件系统节点。
* 通过[**start\_kicker**](http://10.16.20.15/l1mp6/s?refs=start_kicker&project=kernel-3.10)()创建per CPU的喂狗进程wdtk-x，这些进程都是**系统中优先级最高的进程**，执行的核心函数是[**kwdt\_thread**](http://10.16.20.15/l1mp6/s?refs=kwdt_thread&project=kernel-3.10)()，里面实现了前面提到的喂狗框架。
* 通过[**wk\_proc\_init**](http://10.16.20.15/l1mp6/s?refs=wk_proc_init&project=kernel-3.10)()创建了proc文件系统节点/proc/wdk。我们可以通过它来设定超时时间、喂狗时间、打开/关闭WDT。格式如下（如关闭WDT）：echo 0 20 30 0 0 > /proc/wdk
  + 参数1：目前没用到，可以忽略，直接写0。
  + 参数2：喂狗间隔，默认20，单位秒。
  + 参数3：超时时间，默认30，单位秒。
  + 参数4：目前没用到，可以忽略。
  + 参数5：打开/关闭WDT。
* 根据当前online CPU状况更新cpus\_kick\_bit并注册CPU hotplug回调函数，因为CPU会根据负载上电/下电，因此需要及时更新cpus\_kick\_bit、kick\_bit。

    需要注意的是：

* WDT初始化和WDK初始化是处于kernel初始化的不同阶段，WDT通过module\_init()完成，而WDK通过late\_initcall()，中间隔着各种驱动，因此如果有些驱动执行时间太长，会导致开机30s HWT的问题，这个在后面章节提到。
* 在**KK1.MP11以前的版本**启动喂狗进程时，会直接调用到kwdt\_thread()，里面会设置一遍kick\_bit，最后等于cpus\_kick\_bit然后喂WDT，但实际上第1次kick并没有累加起来，查看log：



* 看到所有wdk进程起来后，都会设置kick\_bit(就是local\_bit)，但是没有累加起来，无法等于0xf，也就无法喂WDT，真正喂狗是在20s之后，从时间轴来看：



* 这是为什么呢？原因是[**start\_kicker**](http://10.16.20.15/l1mp6/s?refs=start_kicker&project=kernel-3.10)()里没创建完进程，就会调用[**wk\_cpu\_update\_bit\_flag**](http://10.16.20.15/l1mp6/s?refs=wk_cpu_update_bit_flag&project=kernel-3.10)()将kick\_bit清0，因此无法累加。**KK1.MP11及之后的版本就没有再调用**[**wk\_cpu\_update\_bit\_flag**](http://10.16.20.15/l1mp6/s?refs=wk_cpu_update_bit_flag&project=kernel-3.10)()，因此wdk初始化完就会完成第1次喂狗。

### 喂狗

[**kwdt\_thread**](http://10.16.20.15/l1mp6/s?refs=kwdt_thread&project=kernel-3.10)()完成喂狗动作，里面就是一个循环：

* 如果需要更新超时时间，则在这里调用wdt函数更新。
* 更新对应CPU的kick\_bit，如果等于cpus\_kick\_bit则清除kick\_bit并喂WDT。
* 显示UTC时间，主要用于kernel和android时间同步，这条log非常重要，如果要看android log当时对应的kernel log，就可以通过这条log找到大致的位置。由于WDK是每20s跑一次，因此这条log也是每20s输出一次，如下：[   25.876493]<1>[thread:187][RT:25867046616] 2015-01-01 00:10:38.35106 UTC; android time 2015-01-01 00:10:38.35106
* 睡眠20s

    各个CPU的[**kwdt\_thread**](http://10.16.20.15/l1mp6/s?refs=kwdt_thread&project=kernel-3.10)()没有同步，因此各自喂狗的时间不定，但只要喂狗间隔不超过30s就没有事。

### CPU hotplug

    由于有注册CPU hotplug回调函数，因此在某颗CPU上电时，会将cpus\_kick\_bit对应的bit置位并同时喂狗。下电时，会清除cpus\_kick\_bit对应bit位并同时喂狗。

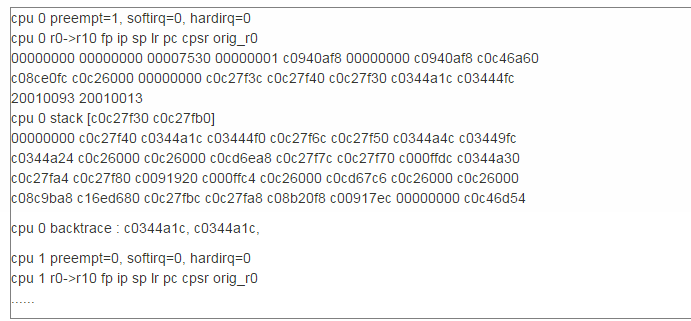
## 超时触发流程

如果喂狗进程没有及时喂狗，WDT就会超时触发FIQ，而分析解决HWT的问题很依赖WDT FIQ能输出什么信息。因此必须要熟悉WDT FIQ流程。32bit/64bit kernel处理FIQ方式不一样，因此要分开讲解。

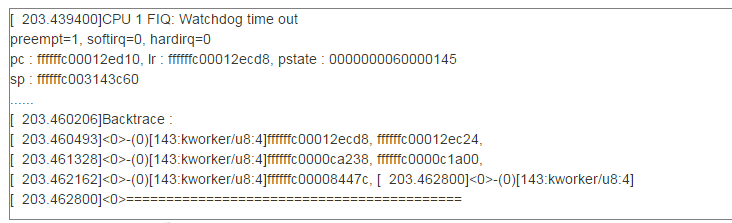
### 32bit kernel FIQ触发流程

 在WDT驱动里有注册了WDT FIQ中断处理函数[**wdt\_fiq**](http://10.16.20.15/l1mp6/s?refs=wdt_fiq&project=kernel-3.10)()，当WDT发出中断后，kernel流程如下：

* \_\_vectors\_start + 0x1C：这是异常向量表FIQ入口，所有FIQ中断都会导向这里。
* [fiq\_glue](http://10.16.20.15/l1mp6/s?defs=fiq_glue&project=kernel-3.10)()：FIQ入口直接调用了这个函数，此时处于FIQ mode，用独立的栈，只要还在这个独立的栈里就无法调用printk输出log，原因是printk使用了current()，会从栈底取出thread\_info结构体，而该独立的栈咩有thread\_info结构体。
* [**fiq\_isr**](http://10.16.20.15/l1mp6/s?refs=fiq_isr&project=kernel-3.10)()：这个函数更新了fiq\_step。后面会继续讲fiq\_step内容。
* [**wdt\_fiq**](http://10.16.20.15/l1mp6/s?refs=wdt_fiq&project=kernel-3.10)()：到了WDT注册的中断函数了，这里还是无法使用printk，因此只能调用[**aee\_wdt\_printf**](http://10.16.20.15/l1mp6/s?refs=aee_wdt_printf&project=kernel-3.10)()打印到buffer里。这里印出了最重要的信息kick和check(对应的kick\_bit和cpus\_kick\_bit)，比如：kick=0x00000007,check=0x0000000f。得到kick和check就知道是哪些CPU没有喂狗了，接下来重点关注没喂狗的CPU调用栈等信息。
* [**aee\_wdt\_fiq\_info**](http://10.16.20.15/l1mp6/s?refs=aee_wdt_fiq_info&project=kernel-3.10)()：这个函数首先将栈切换为当前进程栈，就可以用printk了。另外当前CPU的寄存器打印到per CPU buffer里。最后只允许一个CPU继续往下走，其他CPU直接死循环等待重启了。
* [**aee\_wdt\_irq\_info**](http://10.16.20.15/l1mp6/s?refs=aee_wdt_irq_info&project=kernel-3.10)()：这个函数将输出重要log
  + 喂狗，防止超时。
  + 停止其他CPU。
  + 将per CPU buffer输出到last\_kmsg。注意：只输出到last\_kmsg，kernel log是看不到的。last\_kmsg里的log如下：



* 如果后面流程卡住了，那么能参考的信息只有last\_kmsg了，因此这个信息尤为重要，里面包含CPU寄存器和调用栈。
* 如果没有卡住，那么在kernel log也可以看到这些信息，类似如下：



* + 我们重点看CPU Backtrace部分，这些都是函数地址，需要将地址转换为函数名和所在文件、行号。则需要用到arm-linux-android-addr2line（用于32位）/aarch64-linux-android-addr2line（用于64位，详情请查看quick start => Native Exception问题深入解析 => 进阶篇: coredump分析 => GNU tools）和vmlinux（必须是当时一起编译出来的，如果后面有更新过，addr2line得出的结果可能错误）。比如：

*./prebuilts/gcc/linux-x86/aarch64/aarch64-linux-android-4.9/bin/aarch64-linux-android-addr2line -Cfe vmlinux 0xffffffc00014e68c => ring\_buffer\_unlock\_commit()*

    全部转换后就可以看到完整的调用栈了，有助于我们分析哪里卡住。

* 调用mt\_aee\_dump\_sched\_traces()输出中断信息。这个信息只有eng版本才打开(user版本打开方法请参考[FAQ15102]如何调试IRQ引起的HWT？)。这个有什么用呢？HWT有一个原因是IRQ太过频繁导致，因此IRQ信息能直接看出原因
  + 当前CPU中断信息，可以看到上次IRQ/SoftIRQ/tasklet/hrtimer/SoftTimer信息，如果HWT时调用栈在IRQ里，还可以看到该IRQ的信息，可以**判断执行是否太久**。
  + IRQ Status会印出所有中断在Dur时间段里触发次数，**如果太多就有问题了**（比如>1000次每秒）。



* 将aee wdt buffer打印到last\_kmsg，之前是per CPU buffer，这次是aee wdt buffer：



，这里才能从last\_kmsg看到kick/check信息。

* [**BUG**](http://10.16.20.15/l1mp6/s?refs=BUG&project=kernel-3.10)()，之后就是正常的die/panic流程了，所以kernel log总能看到PC is at aee\_wdt\_irq\_info。看到这个信息就可以判断是HWT了。

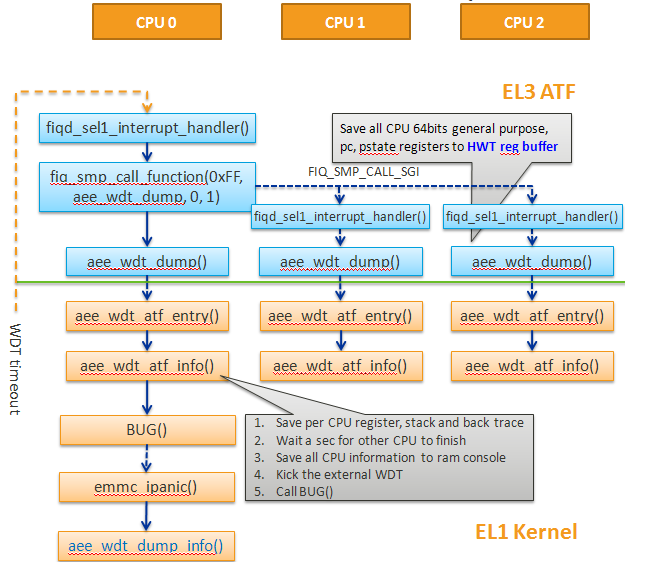
### 64bit kernel FIQ触发流程

    64bit kernel有ATF(ARM Trusted Firmware)，FIQ是security interrupt，必须路由到secure world处理，之后才会返回kernel完成剩下的工作。

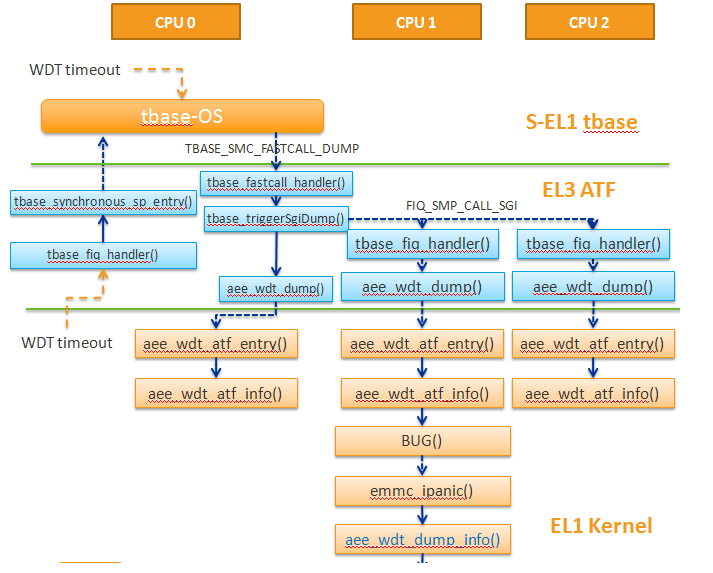
    建议了解下ATF相关内容再来看下面的知识点。

    ATF初始化时就会注册WDT FIQ处理函数，该函数保存per-CPU寄存器信息并返回kernel处理。

    仅有ATF的流程比较简单，由于在ATF时，屏蔽所有中断，因此WDT FIQ只能在EL1 kernel时发生，如下：



如果存在TEE OS则更复杂些，WDT FIQ可能在EL1 kernel或S-EL1 TEE OS触发，最后都会路由到S-EL1 OS处理，然后再返回ATF，最后到EL1 kernel，如下：



  除了底层流程不一样外，kernel流程是差不多的，aee\_wdt\_atf\_info()等效于aee\_wdt\_irq\_info()。因此可以参考上面的内容。

### FIQ step

  kernel发生异常或HWT，当时的系统是不稳定的，无法保证panic或上面的WDT FIQ流程能顺利走完。因此我们使用fiq step标记走到了哪里，如上面的WDT fiq流程里，基本上每走几步就通过aee\_rr\_rec\_fiq\_step()记录特定的数字。在db解开后的SYS\_REBOOT\_REASON、SYS\_LAST\_KMSG和\_\_exp\_main.txt都会有fiq step以供查看，如下：

*WDT status: 2 fiq step: 41  exception type: 1*

    详情可参考：[FAQ14332]SYS\_LAST\_KMSG里的hw\_status和fiq step的含义。

    一般kernel异常，fiq step都不为0，为0表示CPU没有走到这些步骤当中，可能是没异常或CPU卡死跑不了代码。

### 结语

   请结合代码理清一遍流程，知道哪条log从哪个函数打印出来。如果需要加log调试，也知道该加在哪个位置。

    学完WDT/WDK相关内容并不代表就可以独立解决HWT/HW REBOOT的问题了，还需要很多kernel基础知识支撑才行，因为任何模块都不是独立存在，在后面的案例分析就会有更深刻的体会了。

# 分析方法

## HWT

WDT第1阶段超时将触发FIQ，此时系统收到FIQ并处理，最后调用BUG()走正常的kernel exception流程，生成的db类型就是HWT：hardware watchdog timeout。

根据HWT发生的时间点可以分为2类。

### 开机30s超时

 kernel初始化流程：[**start\_kernel**](http://10.16.20.15/l1mp6/s?refs=start_kernel&project=kernel-3.10)() => [**rest\_init**](http://10.16.20.15/l1mp6/s?refs=rest_init&project=kernel-3.10)()，这个过程都是0号进程完成，在rest\_init()就创建了1号进程执行[**kernel\_init**](http://10.16.20.15/l1mp6/s?refs=kernel_init&project=kernel-3.10)()，接着=> [**kernel\_init\_freeable**](http://10.16.20.15/l1mp6/s?refs=kernel_init_freeable&project=kernel-3.10)() => [**do\_basic\_setup**](http://10.16.20.15/l1mp6/s?refs=do_basic_setup&project=kernel-3.10)() => [**do\_initcalls**](http://10.16.20.15/l1mp6/s?refs=do_initcalls&project=kernel-3.10)()。这个过程是单线程的，都是1号线程完成。

    WDT/WDK驱动都在里面初始化，而WDT通常很早就完成初始化，WDK驱动较为靠后，这中间夹着其他驱动，如果其他驱动因为异常而卡较久时间，就有可能到30s都没有喂狗而超时。那原因和解决方法有哪些呢？主要有：

* 驱动初始化时间太长。比如TP需要升级固件，简单的解决方法是手动喂狗。
* 没有配置好或HW故障，打印出很多error log。特别是I2C，出现异常会连续try几次，如果发的cmd较多，会严重拉长初始化时间。还有其他模块如MSDC异常等。因此应该修复这类问题。
* 其他HW故障或代码逻辑卡死。比较复杂，需要具体问题具体分析了。

    初始化时间长的问题，可以通过uart log检索[1:swapper/0]关键字分析就行了。后面有详细例子讲解。

**分析所需材料：**

* db、vmlinux和uart log
* db在sd卡下的mtklog/aee\_exp，或/data/aee\_exp目录下
* vmlinux在codebase里的out/target/product/$proj/obj/kernel\_obj/vmlinux

### kernel运行过程中30s超时

 此时WDK已初始化好，而喂狗进程都是优先级最高的实时进程，其他任何进程都无法卡到喂狗进程。那谁可以卡到喂狗进程呢？换句话说，谁可以影响到系统调度？答案有：

* 长时间关闭中断。这个直接无法调度了，还可以细分为：
  + 关中断的CPU死锁
  + 关中断后陷入循环
* 中断频繁触发。则导致CPU超载，进程基本没有机会被执行到。
* 长时间关抢断。
* 其他HW故障，如总线卡死或HW不稳定。

**分析方法**：

* 获取kick/check bit得知哪些CPU没喂狗，查看last\_kmsg/kernel log看这些CPU的调用栈（《超时触发流程》有讲如何转换CPU Backtrace），看是否卡在锁里。
* 如果怀疑中断过多，则需要eng版本复现抓取db，看isr monitor显示的各个中断触发的频率，判断是哪个中断引起，再找对应的驱动解决。
* 如果发现不符合逻辑引起的卡死，可能是HW不稳定，需要按其他方式处理。

**分析所需材料**

* + db、vmlinux，必要时需要uart log或mobile log。
* db在sd卡下的mtklog/aee\_exp，或/data/aee\_exp目录下
* vmlinux在codebase里的out/target/product/$proj/obj/kernel\_obj/vmlinux

## HW REBOOT

### HW reboot原因

  WDT第1阶段超时将触发FIQ，如果CPU没有响应或CPU响应了但没有在第2阶段超时时间内完成重启，就好导致第2阶段超时发出复位信号复位整个系统，生成的db类型就是HW reboot。

    可以看到HW reboot定义很简单，这样导致了许多问题都导向了HW reboot，而处理问题的方法都不一样。那有哪些异常会导致HW reboot？有如下：

* 多个异常先后发生，这种情况，我们只关注第1个异常就够了，之后的异常可能因为系统已不稳定发生的异常。比如发生KE，在panic flow卡住引起HWT，HWT flow又卡住直接HW reboot了。这种情况，我们只关注KE而不是HW reboot。
* ATF发生panic，ATF层是无法处理任何中断（包括WDT FIQ），里面发生了panic之后都是死循环等待HW reboot。这个不能按HW reboot方式处理，需要看atf log分析解决问题。
* preloader/lk发生panic，和ATF一样，发生HW reboot。但不能按HW reboot的方式处理，这种情况应该看uart log（preloader/lk部分），分析解决问题。
* HW故障，CPU无法正常执行。这个就是纯粹的HW reboot了，也是我们后面要关注的。

### HW reboot调试信息

  基于HW reboot产生原因的复杂性。我们要学会如何区分呢？首先我们要了解哪些信息可以给我们参考。HW reboot没走任何软件流程，是直接复位的，无法知道CPU处于什么状态，因为重启后什么都丢了，不像kernel panic或HWT有详尽的CPU寄存器和调用栈参考。其实HW reboot还是有些硬件模块记录了当时的情况。

**wdt status/fiq step**

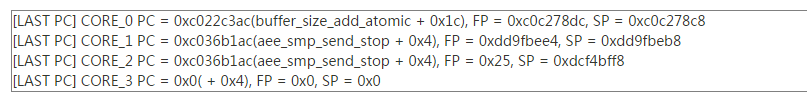
    在preloader初始化WDT前保留了WDT\_STA寄存器的值，并转化为wdt status（或者叫hw\_status），这个值保存在SYS\_REBOOT\_REASON/SYS\_LAST\_KMSG/\_\_exp\_main.txt，和fiq step一起。详情可参考：[FAQ14332]SYS\_LAST\_KMSG里的hw\_status和fiq step的含义。

    wdt status提供了重启的原因，1或5是HW reboot，2是SW reboot等。fiq step的值为0或非0提供了是否走了kernel panic或HWT的流程。如果fiq step不为0，即使wdt status为1或5，我们都不会当成HW reboot来分析。这种情况的原因是panic或HWT流程卡住导致HW reboot了，我们需要参考last kmsg，按普通的kernel panic或HWT分析，找问题的原因。

    如果wdt status为1或5且fiq step为0，那么这个就是HW reboot了。如何调试呢？好像这些信息无法帮我们找到问题。我们继续看还有哪些调试信息。

**last pc**

从82/92及之后的平台，WDT发出复位信号的同时锁存了online CPU的PC/FP/SP。在重启后记录在db的SYS\_REBOOT\_REASON里，格式如下：



有些平台有特别的设定，比如MT6795的PC值有特别的含义：

* 0xa0a0a0a0a0a0a0a0：PC为空，表示没抓到last PC。
* 0xb0b0b0b0b0b0b0b0：抓取last PC失败。

    还有PC所指地址代表的含义大家也要清楚才行，以前平台的AP是扁平结构，PC值所在位置要吗在user space：0～0xBF000000，要吗在kernel space：0xBF000000～0xFFFFFFFF，后面引入64位使问题变复杂了。在aarch64下，PC值可能值NS EL0 user space，也可以是NS EL1 kernel space，或者SEC EL3 ATF及SEC EL1 TEE OS的。因此就需要了解整个框架，才能进一步判断PC来自于哪一层了：

* 64bit kernel space：0xFFFFFF8000000000~0xFFFFFFFFFFFFFFFF。其中0xFFFFFFC000080000开始就是kernel代码起始地址。
* 64bit user space：0～0x7fffffffff，所有app都跑在这里。
* ATF space：不同平台有不同的layout。
  + MT6752/MT6580/MT6795/MT6797的ATF代码段位于0x111000-0x12C400
  + MT6735/MT6753/MT6735M位于0x43001000-0x43020000
  + MT6755位于0x101000-0x120000

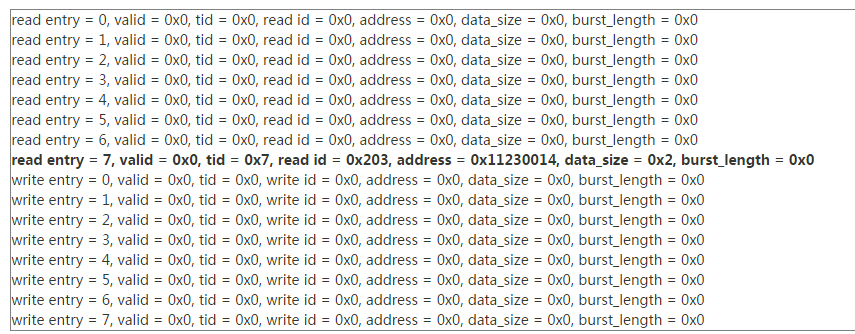
    通过last PC可以知道在最后时刻，各个CPU都在干什么。一般会检查PC所在函数：

* 是否是个死循环。看之前是否关闭了FIQ，如果是，则需要检查代码。
* 正在读写寄存器。读写已关闭clock/power的模块的寄存器会引起bus hang，引起HW reboot。这时要检查读写了哪个模块的寄存器，如果是chip内部模块，直接找Mediatek分析。
* 不在任何space里。那可能跑飞了，多半是HW不稳定引起。需要测量CPU vproc/vsram/Vm等重要电压，看异常发生前是否发生过drop等异常现象。
* 在正常函数里。但没有死循环或其他可疑的地方，这种也可能是HW不稳定，和上面的case一样测量重要电压。

**system tracker**

    上面有提到，如果是读写了已关闭clock/power的寄存器会引起bus hang，最后HW reboot。如何调试这种情况呢？在MT6595及之后引入了system tracker，这个硬件模块会监控bus情况，如果卡死会送一个abort信号给CPU，CPU会进入kernel panic流程。但有时这个bus hang会引起CPU bus卡死，导致CPU也动不了而变成HW reboot。不过bus hang信息还保留在system tracker模块中，重启后会被打包到HW reboot db里。

    如果db里存在SYSTRACKER\_DUMP文件，那一定是发生过bus hang了，该文件内容如下：



    上面记录的地址是物理地址，需要查找datasheet看是哪个模块的寄存器。然后检查代码，看看last pc所在函数是否有可能当时的power/clock被关闭了。

**Low-Power功能标记**

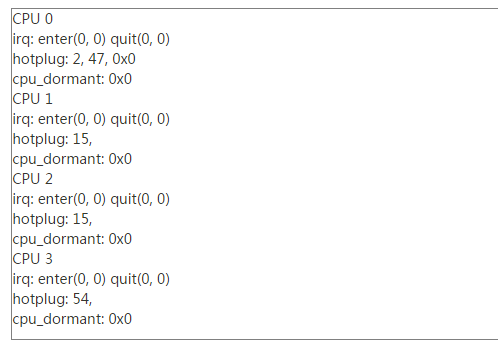
    在SYS\_REBOOT\_REASON不只有wdt status、fiq step和last pc，还有一堆如下内容的信息：



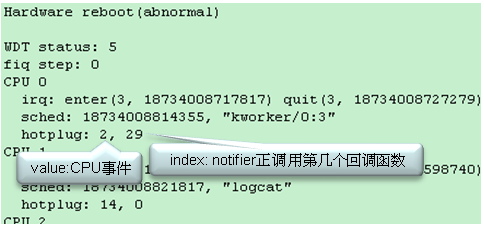
  这些信息显示了系统当时的状态，比如是否进入sodi，当时dvfs档位状态等等。因为系统的复杂性，这些信息也算是提供一些参考吧。

**CPU hot plug**

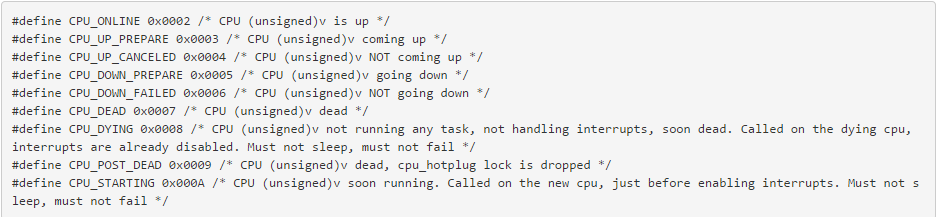
    还有一个重要的信息就是hot plug信息，每个CPU都有，不过CPU0比较特殊，因为不会发生hot plug（在新平台上，CPU0也会发生hot plug），因此有3个参数，其他CPU都只有1个参数：



  CPU0的的参数含义如下：



CPU事件定义在kernel-3.10/include/linux/cpu.h里（2表示CPU\_ONLINE）：



其他CPU参数含义如下（比如54表示CPU已经下电了）：



**last kmsg**

    最后的参考资料是last kmsg，看看是否存在异常。如果没有看到异常，就看最后log停在哪里。猜测和哪个模块相关，然后进一步调试。

### 分析方法

  有了以上的调试信息，我们就可以开始分析了。**纯粹的HW reboot一般和HW相关性较大**。结合以上信息还不足以定位，通常还需要更多信息，比如复现次数等，大致分析如下：



  仅1次复现，如果以上的调试信息看不出问题点就只能后面在关注是否再次发生了。

    可复现的话，需要关注复现路径，发生问题的时间点，结合多次复现的db统一分析，看是否有相似性。

    随机问题，基本导向HW故障，还需要进一步做**信息调查**：



如果软件分析没有任何进展，则需做**HW验证**：



### 结语

 HW reboot是一个复杂的问题，需要积累经验及扩大知识面。还要积累各种调试手段，比如DVFS开关，hotplug开关等，这样调试起来才比较顺手。

# 案例分析

## HWT-kernel初始化时间长(1)

我们来看一个设备异常导致30s HWT的例子。

### 基本信息

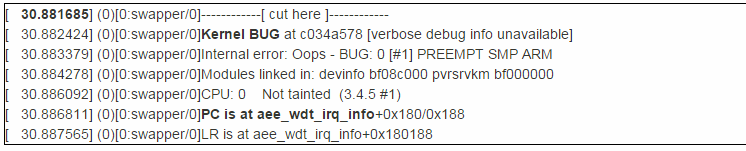
问题：关机状态下插USB不能进入充电模式并且手机会重启

    版本：ALPS.JB2.MP.V1.9

    平台：MT6589

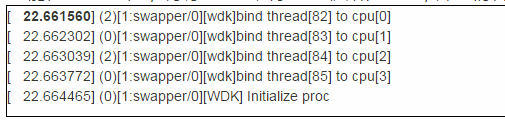
### 分析过程

抓取uart log，如下：



 看到粗体部分就知道是HWT了，而且kernel时间是30s，这是典型的开机30s WDT timeout的case。

    能在30.8s复位，说明0.8s时就将WDT初始化完了，那我们看下log，检查WDK初始化的时间：



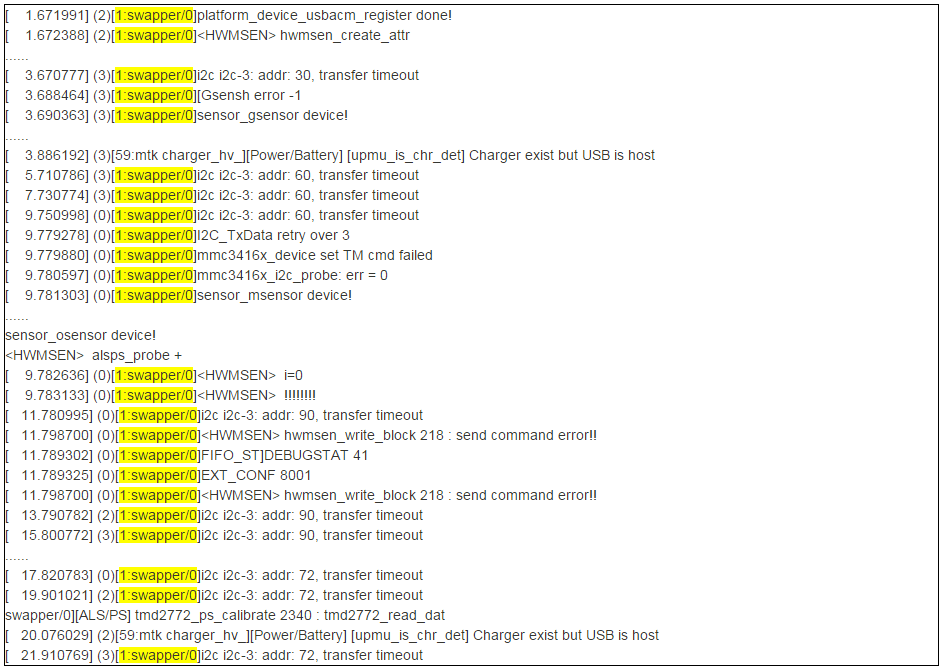
这个明显不对啊，正常情况下kernel 3s左右就完成初始化了，WDK理应也在3s左右，最多不超过10s，如果再晚就会发生30s HWT。而22s肯定会发生HWT，估计在前面某些驱动卡了很长时间了，因此检查初始化流程：

* start\_kernel() -> rest\_init()在rest\_init()创建2个进程跑：kernel\_init()和kthreadd()，其中kernel\_init()之后就加载init，也就是init进程了。
* kernel\_init() -> init\_post() -> Kernel\_init\_done，这条log差不多在开机3s左右要印出来的，但是客户的log：



* 内核中很多驱动通过module\_init()，late\_init()等方法初始化，在内核初始化中在do\_one\_initcal()调用这些注册的函数的。
* kernel\_init() -> do\_basic\_setup() -> do\_initcalls() -> do\_initcall\_level() -> do\_initcall\_level() -> do\_one\_initcal() -> xxx()

    这些函数都是在1号进程里执行（单进程），因此我们只要查看[1:swapper/0]的log(为何是1:swapper/0呢？因为init



可以看到**很多设备初始化失败**（存在i2c error）并且花了20多s的时间。

### 解决方法

这些设备的I2C error全部都要解决，解决这些问题之后kernel就可以在3s内完成初始化，也就不会再发生30s HWT了。

### 结语

解决这类问题，首先要对kernel初始化流程有所了解。否则即使知道HWT，也无法找到问题点。

## HWT-kernel初始化时间长(2)

我们来看一个设备初始化时间长导致30s HWT的例子。

### 基本信息

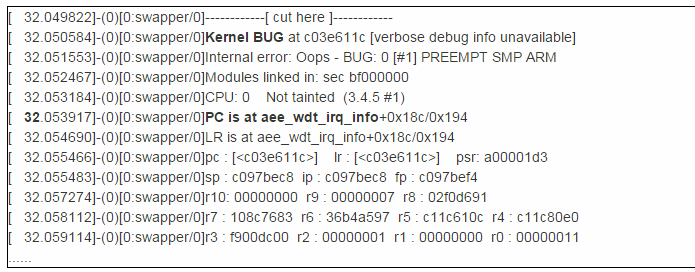
问题：在recovery mode中，会出异常并自动进入normal mode

    版本：ALPS.JB2.MP.V1.2

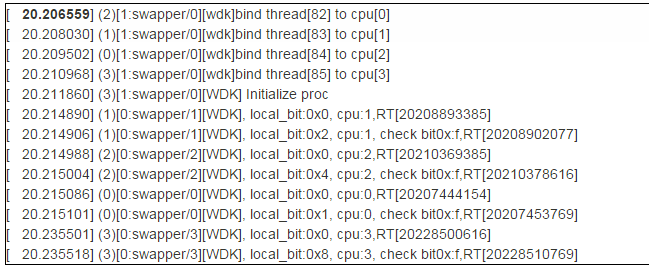
    平台：MT6589

### 分析过程

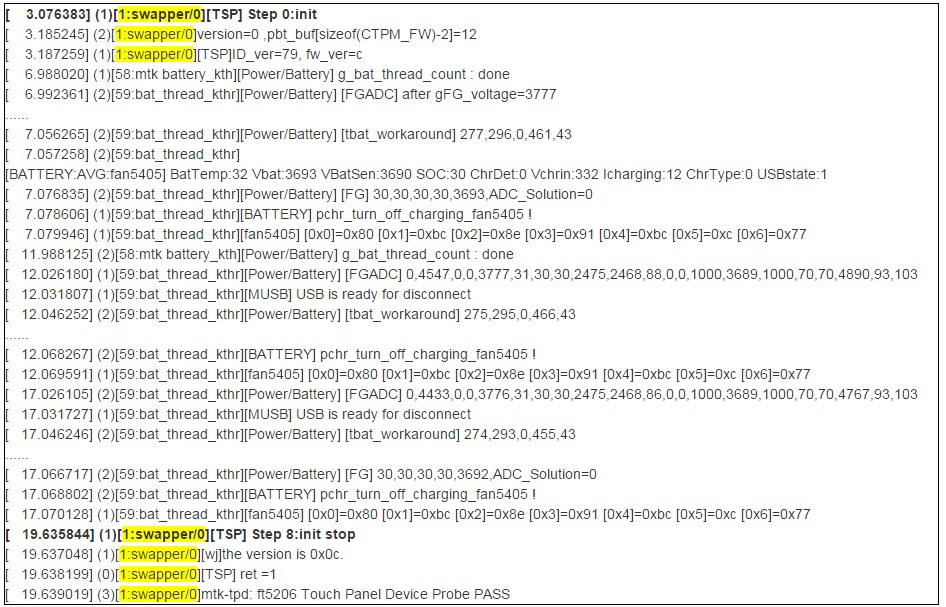
 抓取uart log，如下：



这是典型的开机30s WDT timeout的case，查看CPU0的backtrace，发现已经进入idle了。按照解题思路，需要查看WDK初始化情况，在log中搜索WDK关键字：



可以看到kicker thread在20s左右才创建的，有问题，因此搜索[1:swapper/0]的log：



  看到TSP初始化时间间隔是很长的(16s)，检查TSP驱动，发现在kernel启动之后都会对TP做更新rom的动作，导致花费很长时间。

    至于为什么normal mode下无问题呢，我们再抓取normal log来看，kiker thread也是20s左右创建的，但是之后CPU有plug off的动作，这个动作伴随着kick external WDT，所以才不会再30s时timeout。

### 解决方法

修正TP的驱动，仅在必要时才更新rom，并且更新时主动调用喂狗函数喂狗，避免超时。

## HWT-lk执行时间长

我们来看一个lk执行时间长导致刚进kernel就HWT的例子。

### 基本信息

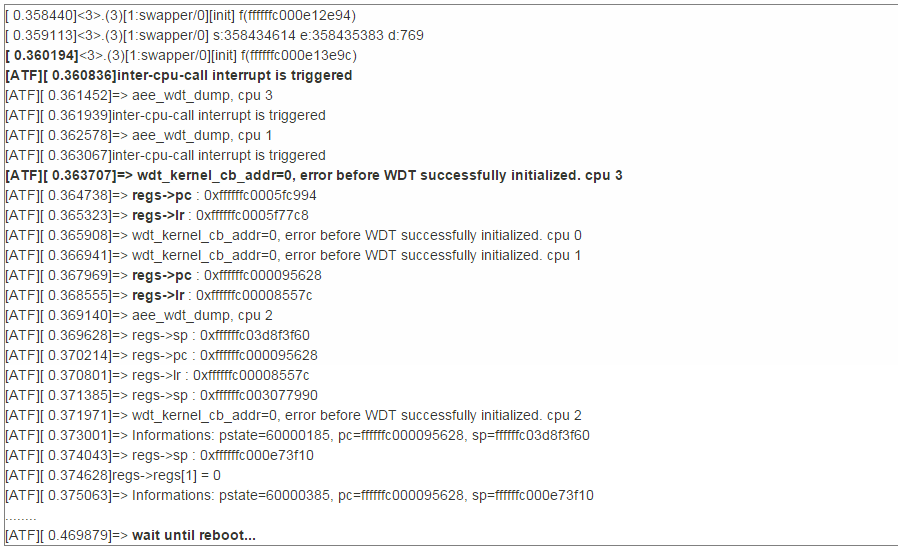
问题：项目刚开始，遇到板子下载软件后重启，几片板子都是这样，没有装外设（屏、TP都不装）

    版本：ALPS.L0.MP8.V2

    平台：MT6735

### 分析过程

抓取uart log，如下：

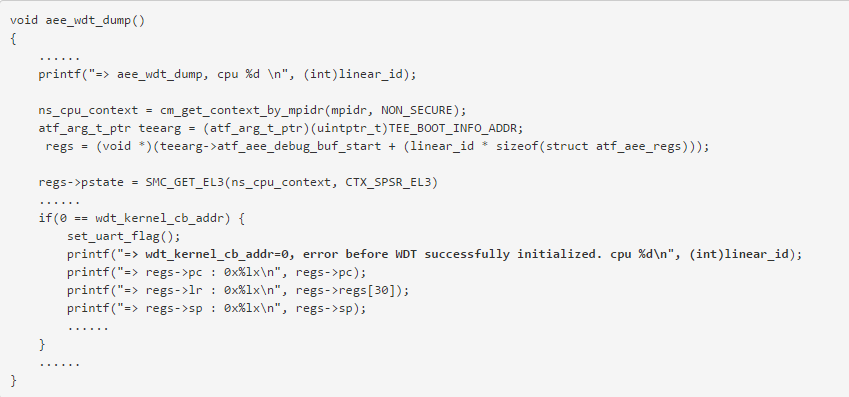


   这是典型的刚进kenrel就发生WDT timeout的case，为什么会这样？

    我们看到log：wdt\_kernel\_cb\_addr=0, error before WDT successfully initialized

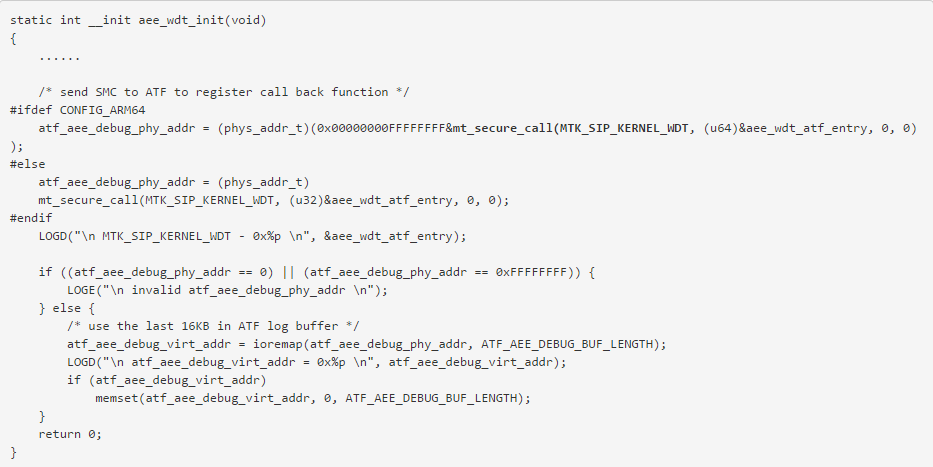
    这个log从哪里来的？代码如下：

vendor/arm/atf-1.0/plat/common/aarch64/plat\_common.c



   ATF是跑在kernel下面的固件，WDT timeout FIQ直接有ATF管理，所以ATF需要将其路由到kernel。通常情况下kernel初始化wdk驱动时会告诉ATF wdt timeout的handler，具体代码在：

kernel-3.10/drivers/misc/mediatek/aee/common/wdt-atf.c

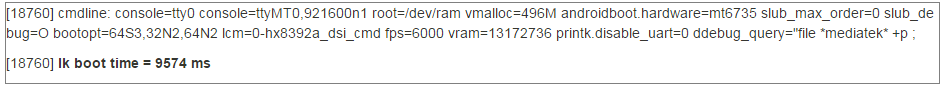


 通过SMC call将aee\_wdt\_atf\_entry地址送给ATF，如果发生HWT，ATF就会回调aee\_wdt\_atf\_entry()，不过看到wdt\_kernel\_cb\_addr=0，表明kernel还没跑到这里。也表明kernel连wdt都还没初始化。那为什么还会timeout？

    首先要了解：preloader、lk、kernel都会设定WDT timeout时间，preloader是16s，lk是10s，kernel是30s，都是在初始化时就设定好的。

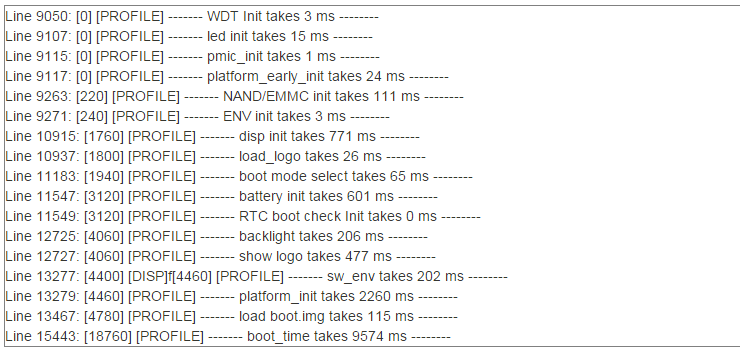
    我们观察到kernel才跑到0.360194s，那时还没跑到kernel初始化wdt的代码，也就是说，这时WDT的timeout时间还是用的是lk的设定时间10s，如果lk设定wdt时到kernel设定wdt前超过10s，就会发生WDT timeout，这个timeout可能在lk阶段，也可能在kernel阶段（kernel 设定wdt前，也表明lk执行时间过长，接近超时），我们看下是否是lk执行时间接近超时呢？

    eng版本有lk profiling，查看log：



    可以看到lk花的时间接近10s了，给kernel到wdt设定的时间不到0.5s，很容易超时的。

    这题的原因是：lk执行时间过长，导致刚到kernel时（还没开始初始化wdt）就发生wdt timeout。需要看lk为何执行时间过久（eng有开lk profiling，搜索PROFILE关键字）：



 就可以看到哪个阶段花的时间长了，如果不够细致，还可以自己添加log分析。

### 解决方法

   通过profile解决异常花时间的模块问题。

    这个case主要知识点是ATF接管WDT FIQ的流程，以及如何识别和分析timeout后印uart log。

## HWT-中断频繁触发

### 基本信息

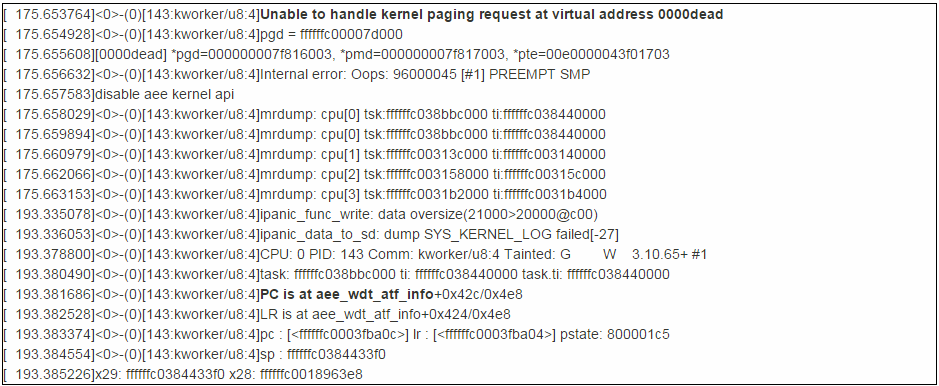
  问题：高概率出现死机

    版本：ALPS.L1.MP3.V1

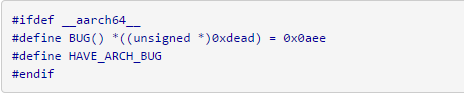
    平台：MT6735M

### 分析过程

抓取db，解开后查看SYS\_KERNEL\_LOG，如下：



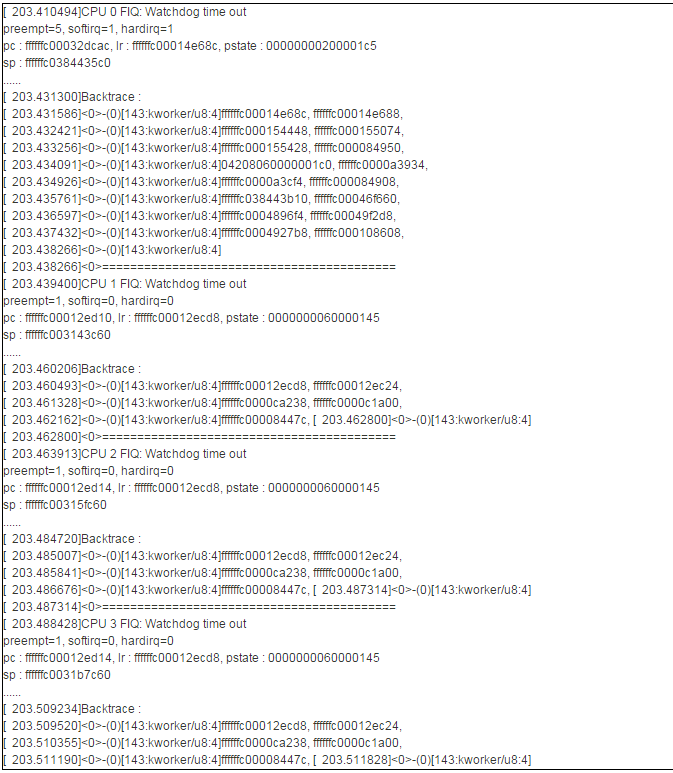
 Unable to handle kernel paging request at virtual address 0000dead的原因是[**aee\_wdt\_atf\_info**](http://10.16.20.15/l1mp6/s?refs=aee_wdt_atf_info&project=kernel-3.10)()最后调用的BUG()定义如下（kernel-3.10/include/asm-generic/bug.h）：



 就是对0xdead赋值触发kernel crash的。看到PC所在的位置就知道是HWT了。这个不是开机30s HWT，因此需要观察kick/check bit：



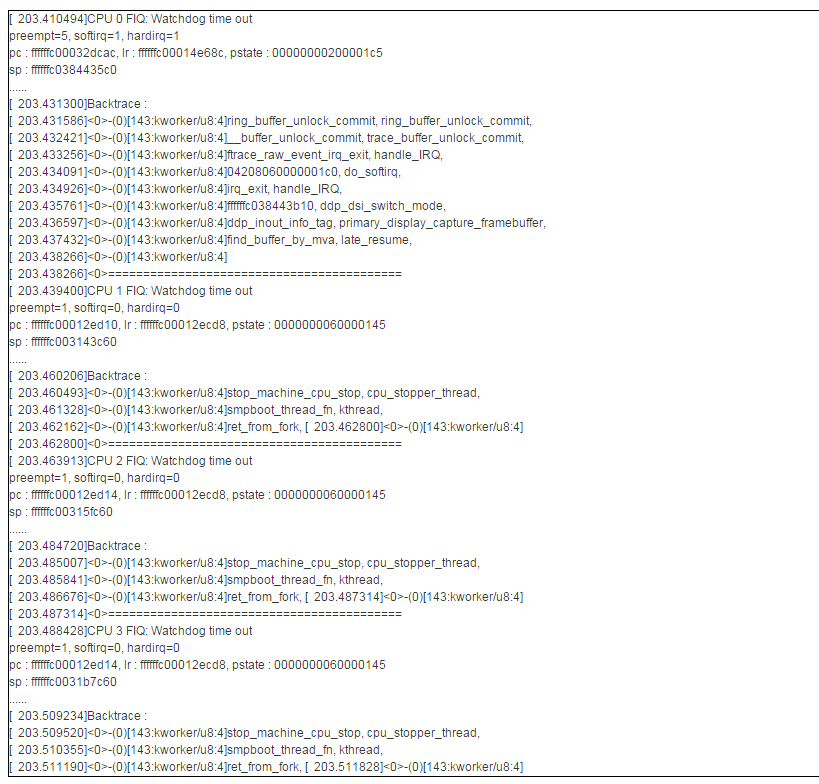
 共有4个CPU（CPU0～3）online，但都没有喂狗。因此需要一一查看CPU调用栈：



    我们重点看CPU Backtrace部分，这些都是函数地址，需要将地址转换为函数名和所在文件、行号。则需要用到aarch64-linux-android-addr2line（详情请查看quick start => Native Exception问题深入解析 => 进阶篇: coredump分析 => GNU tools）和vmlinux（必须是当时一起编译出来的，如果后面有更新过，addr2line得出的结果可能错误）。比如：

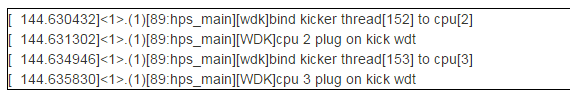
*./prebuilts/gcc/linux-x86/aarch64/aarch64-linux-android-4.9/bin/aarch64-linux-android-addr2line -Cfe vmlinux 0xffffffc00014e68c => ring\_buffer\_unlock\_commit()*

    全部转换后的结果如下：



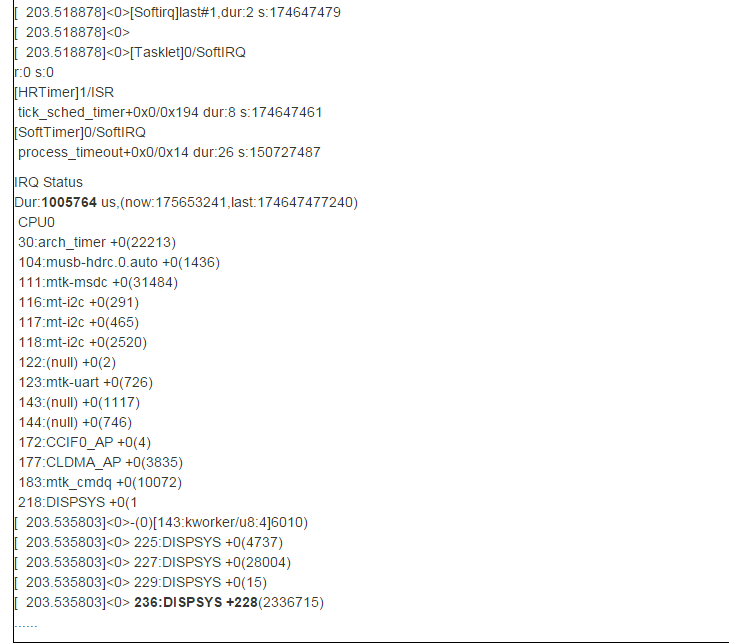
  CPU1～3都处于stop\_machine\_cpu\_stop()，这是kernel stop machine机制，通常在CPU hotplug时会用到，比如CPU1要下电，会将其他CPU进入stop\_machine\_cpu\_stop()停止运行，同时关闭中断，CPU1完成task迁移等动作再恢复。所以问题出在CPU0上。CPU0当时在handle\_IRQ()，正在处理中断，PC所在的位置不像是被卡住的样子。可以排除死锁可能。

    查看最近一次喂狗log：



 不是喂狗进程喂的，而是CPU hotplug喂的，再之前的log就找不到了（SYS\_KERNEL\_LOG大小有限），而且从175s往前推到159s，没有看到CPU1～3的log。估计很早就进入stop\_machine\_cpu\_stop()。

    175s往前推，CPU0倒是很活跃，那为何没有喂狗呢？结合当前的调用栈，怀疑是IRQ过多引起，我们查看下IRQ信息：



 可以看到CPU0正在执行236中断，这个中断属于DISPSYS的，发生异常时，中断已执行了1001419us = 1s。看到IRQ Status最近1005764 us（1s）的时间内，236中断触发了228次。这个频率从该中断行为来看已经不正常了，如果中断执行时间长点，CPU0就无法做任何事，只能不停得处理中断了。这就是该题HWT的原因。

### 解决方法

 问题出在Mediatek的DISP驱动里，需要Mediatek解决并释放patch的。

### 结语

  需要对kernel基本模块有了解，比如stop machine机制。不一定是之前就了解，可以在解决这题的时候，详细查找stop machine机制资料并结合代码学习。

    IRQ状态只有ENG版本才有，如果问题只出在user版本，需要在kernel-3.10/arch/arm64/configs/xxx\_defconfig里设置CONFIG\_MT\_SCHED\_MONITOR=y打开该功能才行。

## HWT-大量DEVAPC violation

### 基本信息

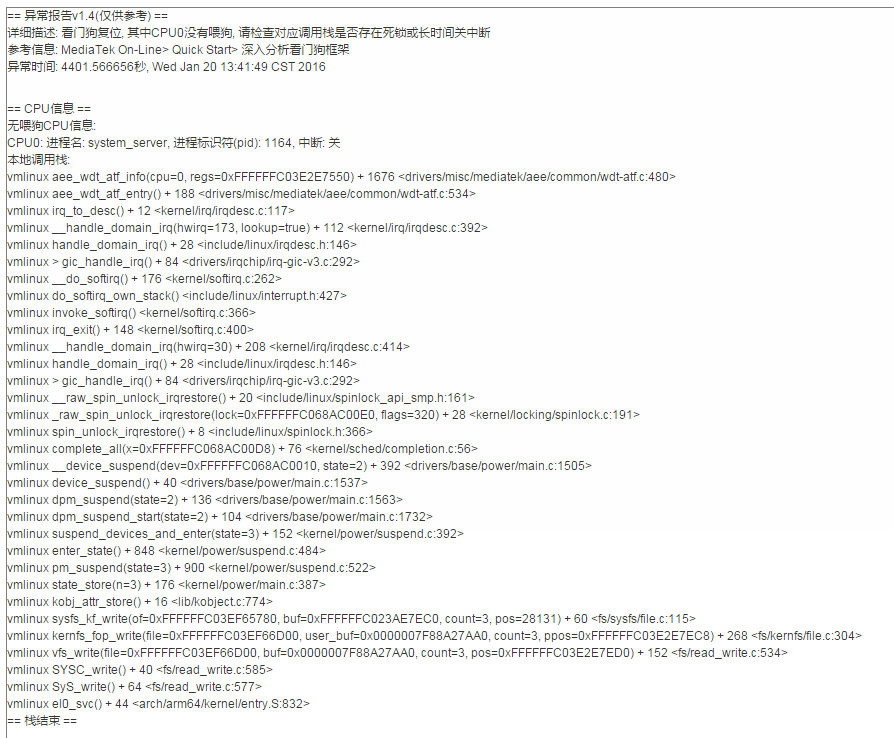
 问题：待机时出现多次 HWT（DEVAPC Access Violation Slave: AUXADC）

    版本：alps-mp-m0.mp9

    平台：MT6797

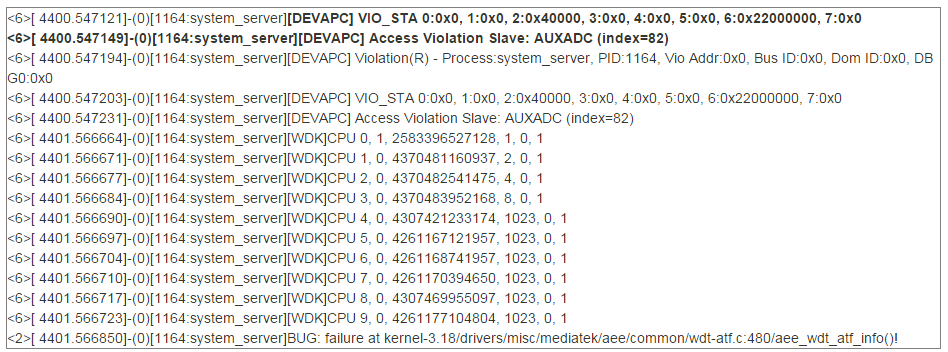
### 分析过程

   用GAT解开db，并结合对应的vmlinux（该文件必须和db一致，具体请看FAQ06985），利用工具（E-Consulter.jar）分析（也可以参考FAQ13941），解析出来的调用栈如下：



   看到调用栈被中断嵌套2次，明显大量中断引起无法正常调度，无法正常喂狗，引起HWT的，可以参考前面的章节《HWT-中断频繁触发》。

    我们看下是否是这样，查看db里面的SYS\_KERNEL\_LOG：



    log中存在大量的[DEVAPC] Access Violation，所以可以不看IRQ统计数据，是否真的很多。那么DEVAPC是什么呢？它的全称是：Device Access Permission Control，管理各个HW模块之前的访问权限。造成访问违例的原因有：

* 访问clock已关闭的HW模块（85%的概率）
* 没有对应读/写权限而去访问HW模块（15%的機率）

    Device APC本身是无法判断哪种原因的，需要深入分析才行。比如这题是AUXADC触发的HWT，需要找MTK分析原因了。

### 解决方法

patch解决（不用当心这样的问题，已修复）。

### 结语

了解log是哪些模块输出的，以及为何输出这些log。然后了解模块的运作（知其然，知其所以然），这样处理问题就得心应手了。

## HWT-SWT卡死

### 基本信息

    问题：使用中发生重启

    版本：ALPS.L0.MP6.V3.20

    平台：MT6752

### 分析过程

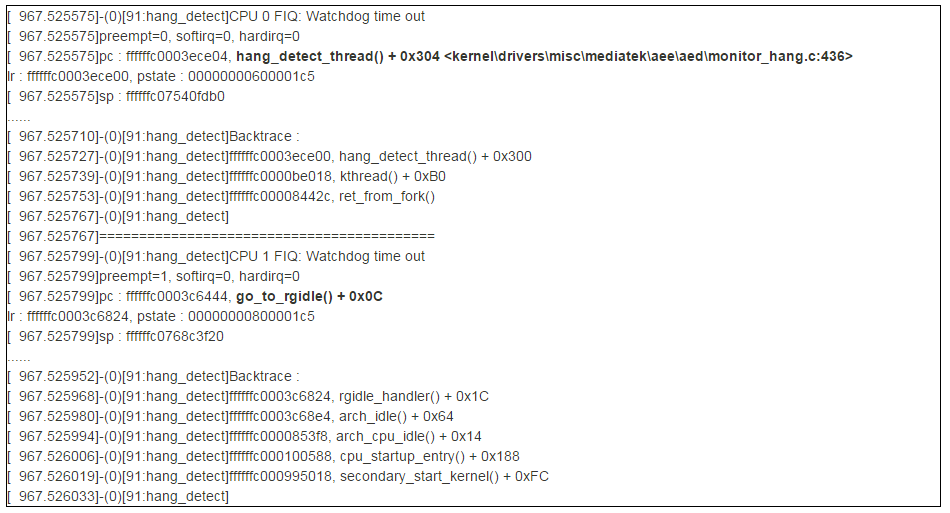
抓取db，解开后查看SYS\_KERNEL\_LOG，如下：



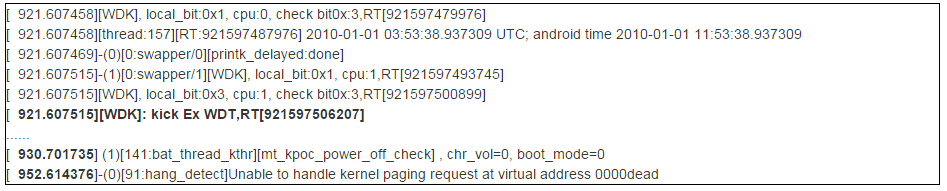
就是BUG()对0xdead赋值触发kernel crash的。看到PC所在的位置就知道是HWT了。这个不是开机30s HWT，因此需要观察kick/check bit：



共有2个CPU（CPU0～1）online，但都没有喂狗。因此需要一一查看CPU调用栈，然后用aarch64-linux-android-addr2line全部转换后的结果如下：



CPU1的PC在go\_to\_rgidle()，这个是CPU无task可运行而进入空闲的函数，表示CPU1处于空闲状态。目前看不出为什么没有喂狗，查看下最近一次喂狗log：



   在921s时喂的狗，超时时刻是952s，30s左右，所以HWT了，理论上921+20=941s会再次喂狗，但从log看到930s到952s没有任何log吐出，CPU肯定发生死循环了。CPU1看不出来，我们看CPU0，前面看到CPU0在kernel\drivers\misc\mediatek\aee\aed\monitor\_hang.c的436行，对应代码如下：



 看到436行是一个while(1);的死循环，而且前面还关闭了中断，直接导致系统无法调度卡死。为什么这里有个死循环呢？

    这是一个功能，hang\_detect进程用于同步SWT的，在android层中，如果发生了SWT，android会发生重启，但如果在SWT过程中因为某些原因卡住，这直接导致手机变成砖头，用户体验很差。因此我们在SWT每走几步就和hang\_detect进程交流，设定超时时间hang\_detect\_counter。如果SWT在某一步中卡住，底层的hang\_detect就会超时，进而跑入上面的代码里，等待HWT。简单来讲：避免SWT卡死，我们将SWT卡死转化为HWT。

    不过这个功能仅在USER版本有效。ENG版本还是卡死，但是系统还在运行，我们可以通过adb shell等接入android，可以查看系统的情况，比如是否是surfaceflinger卡死或EMMC卡死等。

    所以呢，这个问题应该分析SWT卡住的原因，而不是去看HWT了。USER版本直接发生HWT，我们只有db可以看，信息有限，分析问题通常很困难。一般都需要切换为ENG版本复现，问题发生后保留现场，插上USB防止掉电。请Mediatek工程师直接在手机上分析（可以现场分析或远程遥控分析）。

### 解决方法

问题出在SWT，需切换ENG版本复现保留环境分析。

### 结语

 这里需要了解hang detect功能才知道如何下手分析。

    在L1版本之后拿掉了上面的死循环，改为直接调用了BUG()，也就是说，后面版本就将不到这类异常了。

## HWT-死锁卡死

### 基本信息

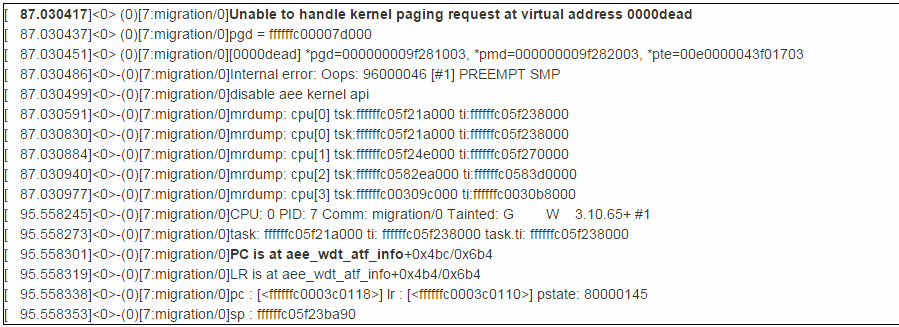
    问题：开机后界面黑屏,按电源键无作用，之后重启

    版本：ALPS.L0.MP8.V2

    平台：MT6735

### 分析过程

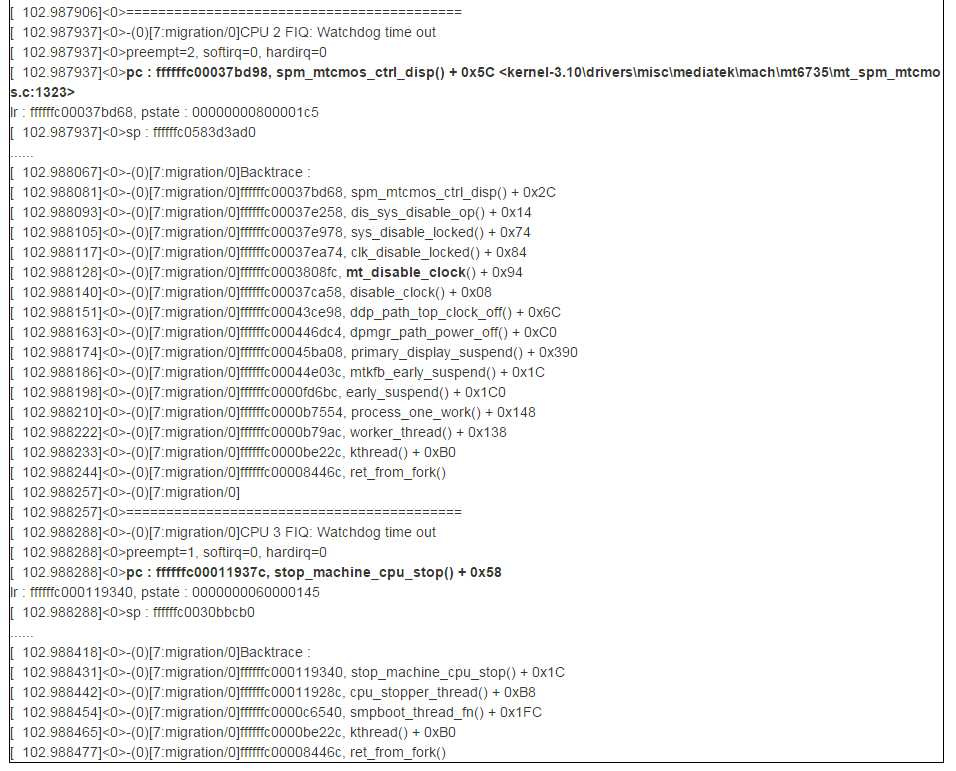
抓取db，db名字就叫db.fatal.00.HWT.dbg，所以是个HWT，解开后查看SYS\_KERNEL\_LOG，如下：



观察kick/check bit：

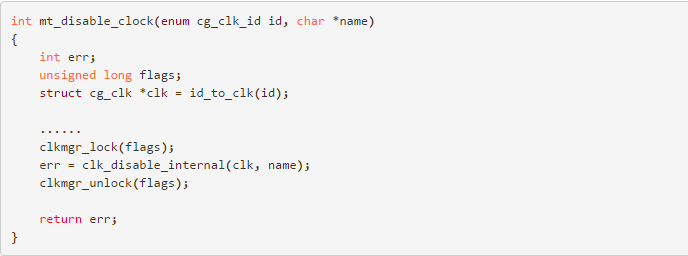


  共有4个CPU（CPU0～3）online，但都没有喂狗。因此需要一一查看CPU调用栈，然后用aarch64-linux-android-addr2line全部转换后的结果如下：



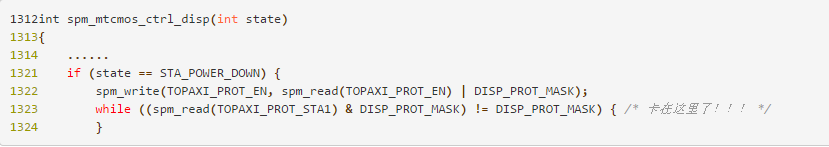
  CPU0和CPU3都处于stop\_machine\_cpu\_stop()，这是kernel stop machine机制，CPU0和CPU3都在等CPU1/CPU2完成工作才能退出stop machine，应该要查看CPU1/CPU2卡在哪里。

    看到CPU1最早也是在stop machine里，不过因为来了中断，中断处理完后又进入softirq处理，里面执行了msdc\_timer\_pm()，要通过mt\_disable\_clock()去关闭clock，当时锁被拿走了，我们看下代码：



 上面的clkmgr\_lock(flags)，用了全局spinlock：clock\_lock。我们看下谁拿走这个锁的。刚好看到CPU2的调用栈也调用了mt\_disable\_clock()，而且还没退出，锁就是它拿了，所以CPU1是在等待CPU2的锁。我们应该看CPU2卡在哪里。

    CPU2是在做suspend，关闭clock，不过关闭clock应该很快才对。我们看下PC所指的代码：



看到1323行是个while循环，就是这个循环导致CPU2一直卡住，CPU1也因此卡住，CPU0/CPU3也因此卡住。

### 解决方法

卡在硬件寄存器状态一直不对，出在Mediatek的DISP power驱动里，需要Mediatek解决的。

### 结语

 HWT一个常见的卡死是死锁，正常情况下拿到锁后尽量尽快执行完所需任务后就释放掉锁，以免持有过久带来一系列问题。

    这题是碰巧CPU2的调用栈明显看出是它拿了锁。如果调用栈无法反应谁持有锁该怎么办？还好**ENG版本**的spinlock有死锁检测机制，持有锁被记录owner和函数，如果获取锁失败，并且在1s内一直无法获取锁，系统会直接印出该锁被谁持有。有助于我们进一步调试。

## HWT-死循环卡死

### 基本信息

    问题：测试中发生一次HWT

    版本：alps-mp-m0.mp7-V1

    平台：MT6755

### 分析过程

 抓取db，db名字就叫db.fatal.00.HWT.dbg，所以是个HWT，用E-Consulter分析，加入vmlinux后，得出**分析报告**如下：



    CPU0~2都没有喂狗，CPU1、CPU2都停在muti\_cpu\_stop，这个是stop machine机制，需要所有在线CPU同步进入multi\_cpu\_stop，所以真正卡住的是CPU0。

    CPU0从调用栈来看，处于idle状态了，为什么没有进入multi\_cpu\_stop？查看CPU0的调度信息：

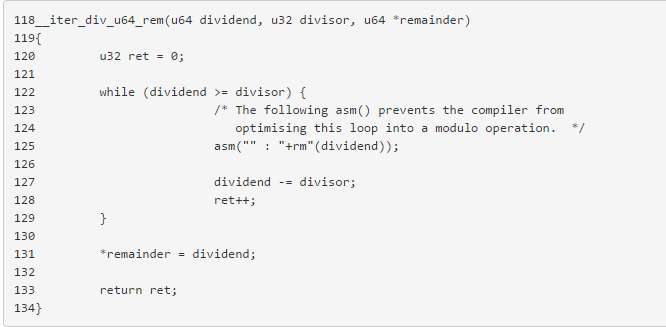


 看到CPU0等待调度的进程已经有71个，更明显看出CPU0卡住了。问题肯定出在CPU0调用栈上，看到

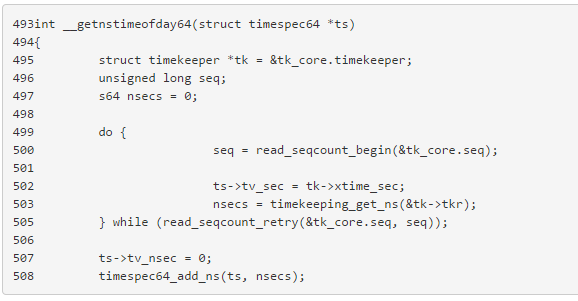


 看到中断是关闭的，所以无法调度，也就说明为何有71个进程等待调度但又没有调度起来了。

 我们仔细分析调用栈，先看\_\_iter\_div\_u64\_rem()函数：



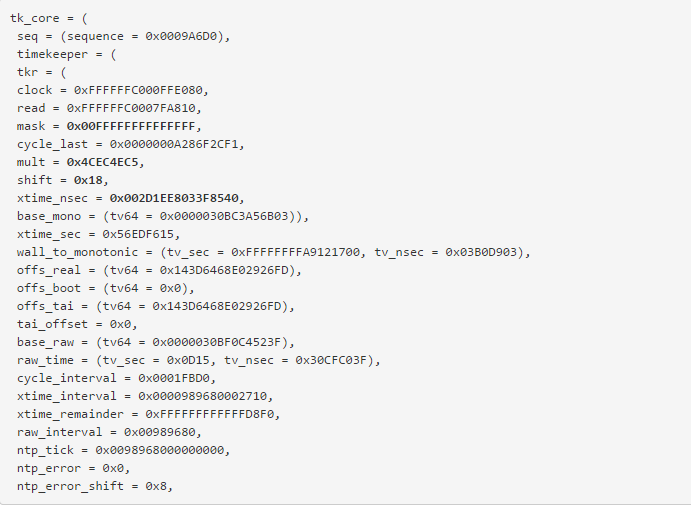
 用debug.cmm加载到trace32，可以看到当时的**dividend** = 0xAC6ABC5F2DB78CA3，**ret** = 0x66FC8BC6。也就是说，这个loop已经跑了**0x66FC8BC6**次了，表示CPU卡在这里很久了。为何会这样？是\_\_getnstimeofday64()传的值nsecs为负值：



   nsecs从timekeeping\_get\_ns来的：



  这里nsec是s64类型的。查看当时tkr（trace32查看变量）：



    很有可能算出负值，引起卡死呢。后面发现kernel已经修复了这个问题：

https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/kernel/time/timekeeping.c?id=35a4933a895927990772ae96fdcfd2f806929ee2

### 解决方法

上kernel patch ALPS02619771。

### 结语

  结合调度信息，调用栈，其他CPU调用栈，一一抽丝剥茧深入分析，最终得出答案。另外原生kernel也可能有问题，不要放弃任何怀疑。

    熟悉使用trace32（可以查看trace32使用教程）、gdb查看全局变量，局部变量等信息，是分析问题的关键。

    熟悉使用E-Consulter，将快速给出初步分析结果，提高效率。

## HWT-CPU卡死

### 基本信息

    问题：SMP引起HWT

    版本：ALPS.L1.MP10

    平台：MT6755

### 分析过程

  抓取db，db名字叫db.fatal.00.KE.dbg，用GAT的logviewer解开db后，查看\_\_exp\_main.txt：



 因此这题其实是HWT，而不是KE，只不过HWT流程没有走完又引起卡住，变成HW reboot。

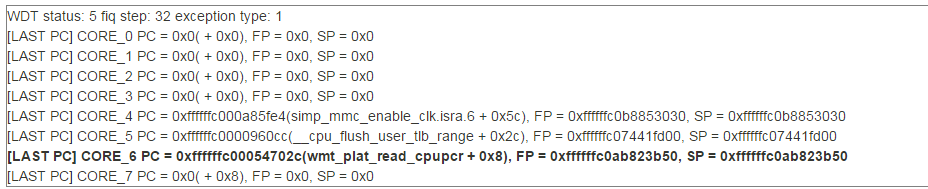
    我们结合对应的vmlinux（该文件必须和db一致，具体请看FAQ06985），利用工具（E-Consulter.jar）分析（也可以参考FAQ13941），解析出来的调用栈如下：



  看到CPU4/5/6都没有喂狗，但只有CPU4有调用栈，而CPU5/6没有调用栈。那么我们先分析CPU4。

    CPU4最后卡住的位置是smp\_call\_function\_single()，这个函数的目的是通知其他CPU做完事后再返回，如果其他CPU没有做完就一直等待。那CPU4是通知哪颗CPU呢？看参数是cpu=6，也就是说，如果CPU6没有做完事，那么CPU4就一直等待。所以要看CPU6在干什么了。

    可惜的是CPU6没有调用栈，无法知道在干什么。其实还有一处信息可以知道CPU6在干什么，那就是last pc，这个信息记录在SYS\_REBOOT\_REASON文件里，内容如下：



    明显CPU6卡在**wmt\_plat\_read\_cpupcr()**里面，而这个函数仅仅是读寄存器而已，所以问题是bus hang引起了这次的HWT。

### 解决方法

 卡在读取硬件寄存器，出在Mediatek的WIFI驱动里，需要Mediatek解决的。

### 结语

这题和《HWT-死锁卡死》不一样的地方是看不到出问题的CPU的调用栈，只能通过last pc知道当时的情况。

## HWT-rt throttle引起卡死

### 基本信息

    问题：长按HOME建弹出快捷方式，按返回键反应迟钝，多操作几次手机会异常重启

    版本：ALPS.L0.MP8.V2.1

    平台：MT6735

### 分析过程

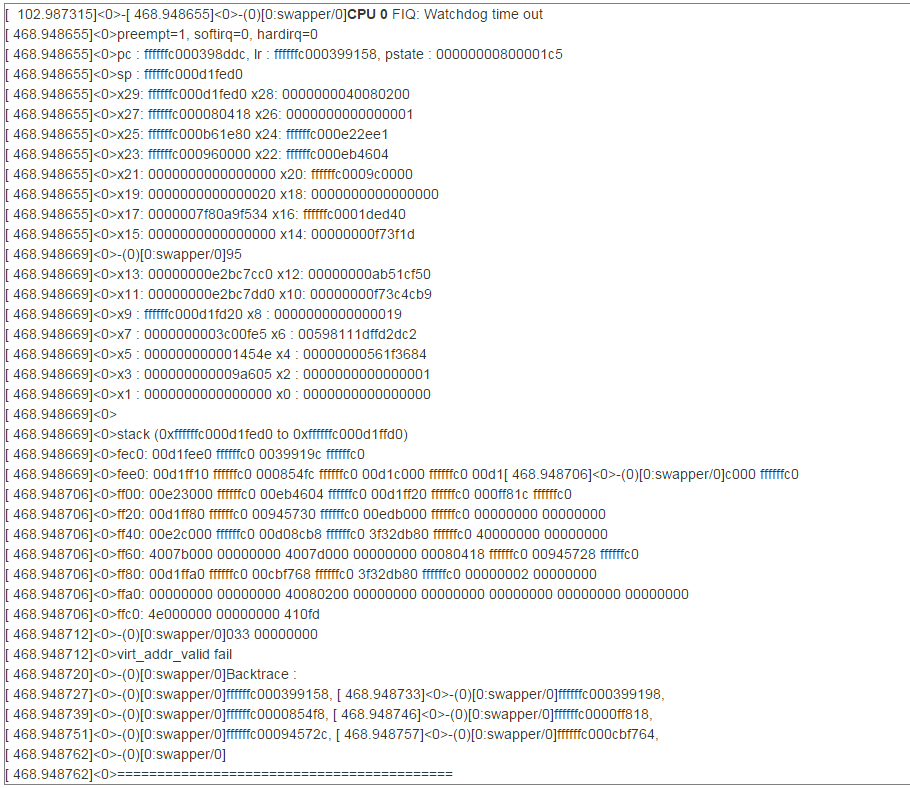
抓取db，db名字就叫db.fatal.00.HWT.dbg，所以是个HWT，用GAT里的log viewer解开后查看SYS\_KERNEL\_LOG，如下：



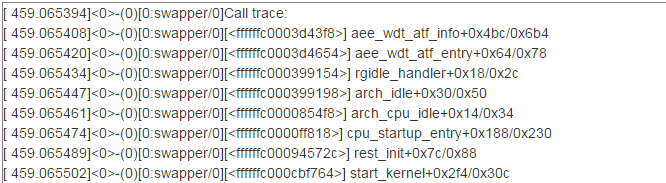
观察kick/check bit：



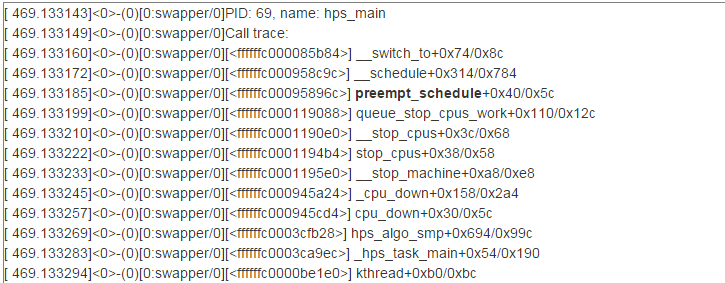
共有2个CPU（CPU0～1）online，但CPU1没有喂狗。因此需要查看CPU1调用栈，查看log发现没有CPU1的调用栈，只有CPU0的：



   而CPU0的调用栈如下：



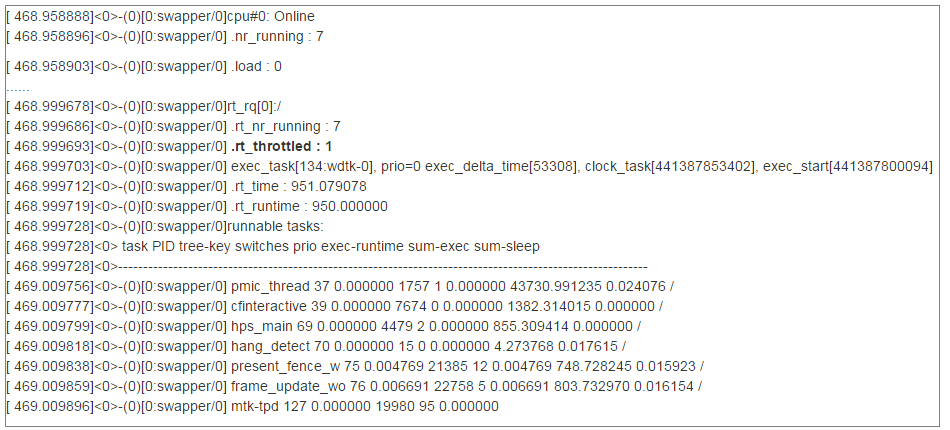
  CPU0处于idle状态，那CPU1去哪里了呢？这时候需要检查前面是否有CPU1活动的log，结果SYS\_KERNEL\_LOG里没看到CPU1的任何log，怀疑是CPU1已被shutdown了，查看hps\_main进程（负责CPU hotplug），刚好有印出来，如下：



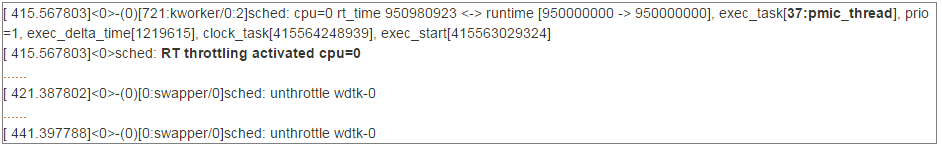
  可以看到CPU down时被抢占了，估计当时在做CPU1 shotdown被抢断，导致check bit没有更新，而CPU1又无法喂狗，引起HWT了。

    需要分析为何hps\_main被抢断。hps\_main是rt thread，除非有高优先级thread进来或被throttled。

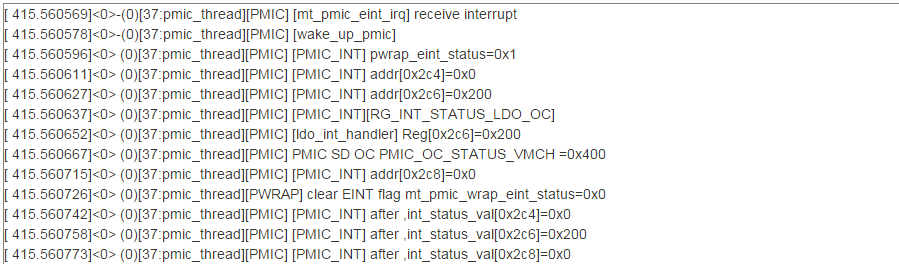
    查看CPU0 runqueue：



  好几个rt thread等待调度，发现rt\_throttled为1（请自行了解rt throttle概念），所以都堵住了，搜索rt\_throttled的原因（搜索throttl关键字）：



 看到throttling的进程是pmic\_thread，原因通常是长时间运行而受到惩罚。搜索有关pmic\_thread的log，发现该线程吐了大量的log：



所以很明显，问题就pmic\_thread吐大量log引起rt throttle，进而影响到hps\_main，产生HWT。

### 解决方法

找出PMIC吐大量log的原因。

### 结语

 HWT本质上就是调度问题，因此滥用实时进程或实时进程长时间占用CPU，有可能引起系统无法及时响应。

    这个问题的解决核心就是要找到谁引起了rt throttle，找到了就好处理了。

    所以往系统里添加实时进程或提升进程优先级为实时都要非常小心。

## HW REBOOT-HW故障

### 基本信息

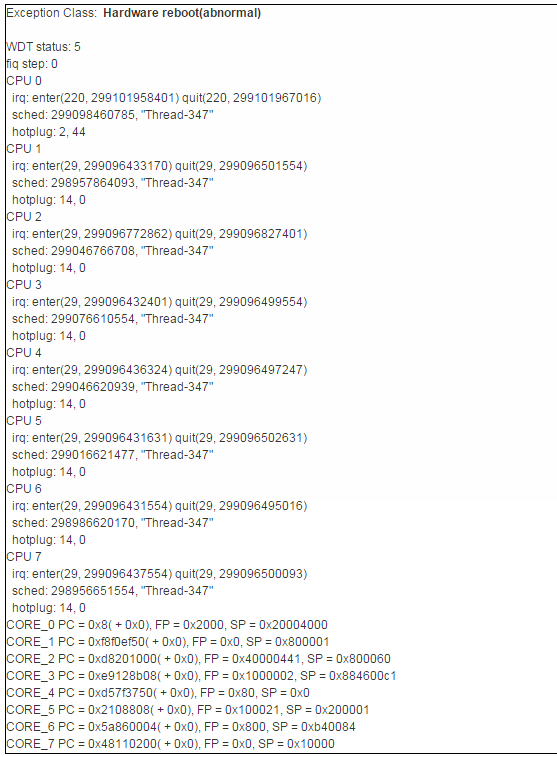
    问题：安兔兔测试浮点运算时重启

    版本：ALPS.JB9.MP.V1.4

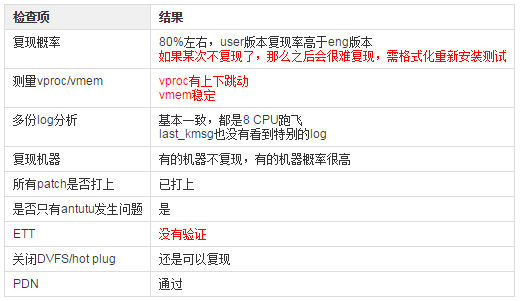
    平台：MT6592

### 分析过程

抓取db，解开后查看\_\_exp\_main.txt，如下：



wdt status为5且fiq step为0，是纯粹的HW reboot，查看last pc，发现PC/FP/SP都是错乱的，估计当时CPU无法正常工作了，怀疑HW故障。先做基本调查：



 由于ETT没有跑过，因此请跑完ETT在确认问题，发现问题依旧。另外vproc确认也不像是有问题的样子。

    另外一条线索，PCB有2版，第1版容易复现，第2版不容易复现，但可惜也没有排查出问题。

    HW问题一般和power/clock相关性较大，因此关闭会影响电压的功能模块验证：



    使用internal buck测试一个晚上，没有复现，基本上断定是ext buck存在问题（92需要ext buck）。接下来需要HW review ext buck电路。HW review ext buck电路之后，检查到buck电感不符合spec，导致buck输出电流不够。则会引起CPU无法正常工作，PC/SP/FP等寄存器乱跑。

### 解决方法

需要更换电感物料。

### 结语

    从db/log分析到最后问题找到，中间的道路是崎岖的。我们要尽量抓住任何可能的疑点（尽量发散），然后一一排查，最后找到问题点。而这过程中使用的调试手段，可能各个平台都不一样，但分析思路都差不多。

    另外还有一点，HW reboot必须可复现才行，否则无法采取任何调试措施。比如这题如果仅复现1次，是根本无法找到问题点的。

# 附录

## 相关FAQ

<https://onlinesso.mediatek.com/_layouts/15/mol/topic/ext/Topic.aspx?MappingId=2d28d0f8-5022-4dd6-b7cd-ce208b6ebb4d>

