**原理**

简介

**什么是HW reboot**

HW reboot是WDT timeout的一种，具体请看：

* MediaTek On-Line> Quick Start> 深入分析看门狗框架

HW reboot本质是WDT timeout后WDT IRQ得不到响应，不得不复位chip的异常。发生的原因有很多：

* 硬件故障
* bus hang
* IC过热
* atf crash
* spm crash
* preloader/lk crash
* HWT过程卡死
* ......

除了明确的分类，如atf/spm在HW reboot大类下会细分外，其他都是HW reboot，需要通过其他信息区分。

**基本分析步骤**

* 分类方面，SpOfflineDebugSuite工具可以辅助判断。
* 通过last pc判断当时CPU处于什么场景，比如是否落在kernel、user space、ATF、TEE等。
  + 这需要对整个系统内存布局的熟悉。
* 查看cpu hotplug/dvfs的档位和状态，判断是否是hotplug、dvfs引起。通过排除法缩小范围。
  + 比如关闭dvfs，使用某一档位。关闭hotplug，单核跑。

**分类**

#### bus hang

如果某个module的clock已经关闭的话，代码还对这个module的寄存器进行读写时，这会导致bug hang，从而最后导致HW reboot. MTK 引入了system tracker去debug  bus hang问题。system tracker 是一个监视bus状态的硬件模块,如果发生了bus hang, system tracker将会发会一个abort 信号给CPU,CPU将会进入kernel panic流程，然而，bug hang经常会导致CPU卡住而导致HW reboot的发生，Bus hang的调试信息会放入system traceker 模块，在下一次重启后我们可以读出这些信息，从而去debug。

当我们解压hw reboot db时，发现SYSTRACKER\_DUMP (O版本之前)或SYS\_LAST\_CPU\_BUS(O版本之后)中有下面的信息，那这个的hw reboot是由bus hang引起的。

|  |
| --- |
| read entry = 0, valid = 0x0, tid = 0x0, read id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 read entry = 1, valid = 0x0, tid = 0x0, read id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 read entry = 2, valid = 0x0, tid = 0x0, read id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 read entry = 3, valid = 0x0, tid = 0x0, read id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 read entry = 4, valid = 0x0, tid = 0x0, read id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 read entry = 5, valid = 0x0, tid = 0x0, read id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 read entry = 6, valid = 0x0, tid = 0x0, read id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 **read entry = 7, valid = 0x0, tid = 0x7, read id = 0x203, address = 0x11230014, data\_size = 0x2, burst\_length = 0x0** write entry = 0, valid = 0x0, tid = 0x0, write id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 write entry = 1, valid = 0x0, tid = 0x0, write id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 write entry = 2, valid = 0x0, tid = 0x0, write id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 write entry = 3, valid = 0x0, tid = 0x0, write id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 write entry = 4, valid = 0x0, tid = 0x0, write id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 write entry = 5, valid = 0x0, tid = 0x0, write id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 write entry = 6, valid = 0x0, tid = 0x0, write id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 write entry = 7, valid = 0x0, tid = 0x0, write id = 0x0, address = 0x0, data\_size = 0x0, burst\_length = 0x0 |

这个**address = 0x11230014**说明是在读取0x11230014寄存器时发后生了bus hang,你可以在SYS\_LAST\_CPU\_BUS找出最后的PC的位置，并找到相关的代码，根据相关spec找到这个寄存器是哪个模块的，最后找出是不是在读取这个寄存器时，这个模块的power或clock已经关闭导致的这个问题。

#### ATF crash

**判断ATF crash**

    查看db解开后是否存在SYS\_ATF\_CRASH，如果有，表示这是ATF crash。用SpOfflineDebugSuite工具（MOL tool搜索关键字下载）也可以识别出来。

**ATF crash种类**

    可以分为2类：

* assert/panic：属于主动报错，检查SYS\_ATF\_CRASH，可以看到类似如下log（ASSERT等关键字）：

[ATF](0)K:[27590.740966]ASSERT: bakery\_get\_ticket : !bakery\_ticket\_number(bakery->lock\_data[me])  
[ATF](2)K:[27590.754518]ERROR: bakery->lock\_data[0]:4d00  
[ATF](2)K:[27590.754633]ERROR: bakery->lock\_data[1]:0  
[ATF](2)K:[27590.754738]ERROR: bakery->lock\_data[2]:d700  
[ATF](2)K:[27590.754847]ERROR: bakery->lock\_data[3]:f902  
[ATF](2)K:[27590.754956]ERROR: bakery->lock\_data[4]:aa00  
[ATF](2)K:[27590.755066]ERROR: bakery->lock\_data[5]:aa00  
[ATF](2)K:[27590.755176]ERROR: bakery->lock\_data[6]:9c00  
[ATF](2)K:[27590.755286]ERROR: bakery->lock\_data[7]:f900  
[ATF](2)K:[27590.755396]ASSERT: bakery\_get\_ticket : !bakery\_ticket\_number(bakery->lock\_data[me])  
E  
Timestamp: 0x00001917f8534561, 27590.741149  
PANIC in EL3 at x30 = 0x0000000000108fbc  
x0 = 0x0000000000000000  
......

* exception：属于代码跑飞/访问错误地址等，检查SYS\_ATF\_CRASH，可以看到类似如下log：

Timestamp: 0x00000003da702889, 16.549685  
Unhandled Exception in EL3.  
x30 = 0x0000000054600f88  
x0 = 0x000000000004f460  
x1 = 0x00000000000494d4  
x2 = 0x0000000000000045  
x3 = 0x000000000000001d  
x4 = 0x0000000000029c20  
x5 = 0x000000000003f000  
x6 = 0x0000000000029cd8  
x7 = 0x0000000000029cfc  
......

用SpOfflineDebugSuite工具（MOL tool搜索关键字下载）可以识别出以上所有类型异常。

**分析建议**

    如果db里存在SYS\_ATF\_RDUMP，那么可以用SpOfflineDebugSuite工具生成cmm脚本。然后用T32分析，否则请提交给MTK处理。

    T32分析atf crash和分析KE类似，如下：

#### 

#### Thermal reboot

查看\_\_exp\_main.txt里的Exception Type字段，比如：

|  |
| --- |
| Exception Class: Hardware Reboot Exception Type: Thermal Reboot  ...... |

告诉你是thermal reboot。用SpOfflineDebugSuite工具（MOL tool搜索关键字下载）也可以识别出来。

在SYS\_REBOOT\_REASON有几个字段和thermal相关，可以检查是否存在异常，类似如下：

thermal\_temp1 = 42  
thermal\_temp2 = 43  
thermal\_temp3 = 41  
thermal\_temp4 = 42  
thermal\_temp5 = 41  
thermal\_temp6 = 0  
thermal\_status: 2  
thermal\_ATM\_status: 0  
thermal\_ktime: 784463835

thermal\_temp1~6代表不同器件的温度，具体定义请查看：

kernel/drivers/misc/mediatek/thermal/mt67xx/inc/mach/mtk\_thermal.h  
里的thermal\_sensor（每个平台可能不一样）  
/\*  
 \* Bank0: CPU-L (TS\_MCU1)  
 \* Bank1: CPU-LL (TS\_MCU2)  
 \* Bank2: CCI (TS\_MCU1 + TS\_MCU2)  
 \* Bank3: GPU (TS\_MCU3)  
 \* Bank4: SoC (TS\_MCU4 + TS\_MCU5)  
 \*/  
enum thermal\_sensor {  
 TS\_MCU1 = 0,  
 TS\_MCU2,  
 TS\_MCU3,  
 TS\_MCU4,  
 TS\_MCU5,  
 TS\_ENUM\_MAX,  
};

thermal\_status、thermal\_ATM\_status表示thermal状态，具体定义请查看：

kernel/drivers/misc/mediatek/thermal/mt67xx/inc/tscpu\_settings.h  
里的thermal\_state、atm\_state（每个平台可能不一样）  
enum thermal\_state {  
 TSCPU\_SUSPEND = 0,  
 TSCPU\_RESUME = 1,  
 TSCPU\_NORMAL = 2,  
 TSCPU\_INIT = 3,  
 TSCPU\_PAUSE = 4,  
 TSCPU\_RELEASE = 5  
};  
enum atm\_state {  
 ATM\_WAKEUP = 0,  
 ATM\_CPULIMIT = 1,  
 ATM\_GPULIMIT = 2,  
 ATM\_DONE = 3,  
};

thermal\_ktime表示前面这些信息更新的kernel时间，单位为us

**参考文档**

DCC上搜索Thermal\_Management关键字可以找到对应的文档

* Thermal\_Management\_MT676X.docx
* thermal management debug SOP V1.0 (CH).pptx

#### lk crash

1.LK 发生assert

检查SYS\_LAST\_LK\_LOG（如果db里有SYS\_PLLK\_LAST\_LOG，则看SYS\_PLLK\_LAST\_LOG。SYS\_PLLK\_LAST\_LOG将被废弃）.

下面的log指示LK的watchdog timeout.

[64128] lk\_wdt\_dump(): watchdog timeout in LK....  
[64129] current\_thread = bootstrap2  
[64129] Dump register from ATF..  
[64130] CPSR: 0x600001f3  
[64130] PC: 0x46027224  
[64130] SP: 0x46096840  
[64131] LR: 0x4600801f  
[64131] mt\_irq\_register\_dump(): do irq register dump  
[64131] GICD\_CTLR: 0x00000012  
[64132] GICD\_IROUTER[0]: 0x00000000, 0x00000000

**分析建议**

lk的timeout时间是10S,需查看是什么原因导致在LK中耗时过长。

下面的log指示LK的发生了assert .

从SYS\_PLLK\_LAST\_LOG中看到：  
  
[529] dtbo\_part\_name is not initialized!  
  
[529] panic (caller 0x48002c3f): ASSERT at (app/mt\_boot/mt\_boot.c:348): 0  
  
[10035] lk\_wdt\_dump(): watchdog timeout in LK....

**分析建议**

对照lk代码，分析LK assert的原因。

2.lk发生data abort

 从uart log有下面相关的log.

[2560] data abort, halting  
  
[2560] r0 0x00000000 r1 0x560bee44 r2 0x560be2a4 r3 0x00000001  
  
[2561] r4 0x00000000 r5 0xf8038000 r6 0x5613d610 r7 0x00000000  
  
[2562] r8 0x66908000 r9 0x7e136000 r10 0x66908000 r11 0x5611f294  
  
[2563] r12 0x5611f294 usp 0x00000000 ulr 0x00000000 pc 0x560710a2  
  
[2563] spsr 0xa0000173  
  
[2564] spsr 0xa0000173 dfsr 0x00000206 dfar 0x66908000  
  
[2564] fiq r13 0x560fb000 r14 0x00000000  
  
[2565] irq r13 0x560fdaf8 r14 0x5607ef82  
  
[2565] \*svc r13 0x5611f200 r14 0x56069a45  
  
[2566] abt r13 0x560faf44 r14 0x5602bf21  
  
[2566] und r13 0x560fb000 r14 0x00000000

**分析建议**

通过pc指针的值和/out/target/product/[project]/obj/BOOTLOADER\_OBJ/build-[project] /lk可以得到当前发生data abort的代码位置，然后再具体再分析具体原因。

#### SPM reboot

**判断SPM reboot**

    查看\_\_exp\_main.txt里的WDT status字段，比如：

WDT status: 16 fiq step: 0  exception type: 0

如果是16，表示这是SPM reboot。用SpOfflineDebugSuite工具（MOL tool搜索关键字下载）也可以识别出来。

**分析建议**

    spm类问题请提交给MTK处理。

#### SSPM reboot

**判断SSPM reboot**

    查看\_\_exp\_main.txt里的WDT status字段，比如：

WDT status: 1028 fiq step: 0  exception type: 0

如果是1024或1028，表示这是SSPM reboot。用SpOfflineDebugSuite工具（MOL tool搜索关键字下载）也可以识别出来。

**SSPM reboot种类**

    可以分为3类：

* assert：检查SYS\_SSPM\_LAST\_LOG，可以看到类似如下log：

Running: ATM  
Assert at 0xf16c

* wdt timeout：检查SYS\_SSPM\_LAST\_LOG，可以看到类似如下log：

Running: IDLE  
WDT T.O from 0x108e

* bus hang：检查SYS\_SSPM\_DATA里的STATUS或AHB\_STATUS的bit4/5/10/13是否全部为0，如果是，则表示存在bus hang，比如：

AHB\_STATUS: 0x0261800

用SpOfflineDebugSuite工具（MOL tool搜索关键字下载）也可以识别出以上所有类型异常。

**分析建议**

    sspm类问题请提交给MTK处理。

#### Preloader Crash

Memory Test Fail

1.Preloader 发生 Memory Test Fail

Uart log中有下面相关的log

[2018/10/25 16:31:54] [MEM] 1st complex R/W mem test fail :FFFFFFFF (start addr:0x60000000)  
  
[2018/10/25 16:31:54] [MEM] 2nd complex R/W mem test fail :FFFFFFFF (start addr:0x80000000, 0x0 @Rank1)  
  
[2018/10/25 16:31:54] [LastDRAMC] 0x10E40C: 0x19870611  
  
[2018/10/25 16:31:54] [LastDRAMC] 0x10E410: 0x0  
  
[2018/10/25 16:31:54] [LastDRAMC] 0x10E414: 0x0  
  
[2018/10/25 16:31:54] [LastDRAMC] 0x10E418: 0x19870611  
  
[2018/10/25 16:31:54] [LastDRAMC] 0x10E41C: 0x0  
  
[2018/10/25 16:31:54] [LastDRAMC] 0x10E420: 0x0  
  
[2018/10/25 16:31:54] [LastDRAMC] 0x10E424: 0x80000000  
  
[2018/10/25 16:31:54] [LastDRAMC] 0x10E428: 0x0  
  
[2018/10/25 16:31:54] [LastDRAMC] 0x10E42C: 0x2101A140  
  
[2018/10/25 16:31:54] [LastDRAMC] 0x10E430: 0xDB77F97E

1.先确定发生的机器是否是单机的，或是跟随机器走的，如果是这样的话，有可能是hw 问题.

2.如果是很多机器出现这种问题，并且又是在做reboot压力测试的话，这种有可能是ddr-reserve mode引起来的问题，可以关闭ddr-reserve mode后再做测试.

3.如果不是上面两种情况的话，需要具体问题具体分析了，请提交eservice.

#### L2 cache error

**如何判断系统是否发生了L2 cache error导致系统重启**

1.解压db, 看last\_cpu\_bus文件是否有这样的内容，比如MT6762，就

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* l2c parity \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[L2C parity] get parity error in mp0

error count = 0x2

index = 0x1c44

bank = 0x100

2.SpOfflineDebugSuite工具分析db提示, 看是否有L2 cache error

**分析建议**

    L2 cache error类问题请提交给MTK处理

#### Dram Gating Error 和 Test Agent Fail

#### 有时候HW reboot的db用SpOfflineDebugSuite分析的结果是Dram Gating Error 或Test Agent Fail，如果没有规律性的话，需要按照下面的流程进行分析。

#### 

**ATF crash案例分析**

#### ATF assert导致HW reboot

**问题背景：**

产线验证，200pcs里有10多pcs重启。

**分析过程：**

拿到5、6份db，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SpOfflineDebugSuite分析，发现是ATF crash：

== 异常报告v1.5(仅供参考) ==  
报告解读: MediaTek On-Line> Quick Start> E-Consulter之NE/KE分析报告解读> KE分析报告  
详细描述: AP读 0x0010FC80地址超时  
参考信息: http://wiki.mediatek.inc/pages/viewpage.action?pageId=93753425  
平台 : MT6797  
版本 : alps-mp-m0.mp9/user build  
异常时间: 0.000000秒, Sat Jan 2 00:06:34 CST 2016  
  
  
== CPU信息 ==  
其他CPU信息:  
 CPU4: 进程名: (null), 进程标识符(pid): 0  
 寄存器异常: SP(0xFFFFFFC096313DB0)/FP(0x0000000000118A70)不在同一内核栈  
 本地调用栈:  
 ...... 0x000000000010FC80()  
 == 栈结束 ==  
  
 CPU5: 进程名: (null), 进程标识符(pid): 0  
 本地调用栈:  
 vmlinux 0xFFFFFFC00019A8F8()  
 == 栈结束 ==  
  
 CPU6: 进程名: (null), 进程标识符(pid): 0  
 本地调用栈:  
 vmlinux 0xFFFFFFC00019AA18()  
 == 栈结束 ==  
  
 CPU7: 进程名: (null), 进程标识符(pid): 0  
 寄存器异常: SP(0xFFFFFFC03E8A7DB0)/FP(0x000000000011A270)不在同一内核栈  
 本地调用栈:  
 ...... 0x000000000010FF3C()  
 == 栈结束 ==  
  
  
这题还需bl31.elf文件, 该文件必须和db匹配。  
详情请查看MOL上的'[FAQ15094]发生ATF CRASH需要提供哪些辅助文件给MTK'

提示AP读 0x0010FC80地址超时，这个地址是SRAM地址，db里有发现SYS\_ATF\_CRASH，因此这题是atf crash。AP读地址超时可以先忽略。

检查SYS\_ATF\_CRASH，log如下：

PANIC in EL3 at x30 = 0x000000000010e69c  
x0 = 0x0000000000000000  
x1 = 0x00000000001149c0  
x2 = 0x00000000000000ed  
x3 = 0x00000000001148a5  
x4 = 0x0000000000007000  
x5 = 0x0000000000000002  
x6 = 0x0000000044600000  
x7 = 0x0000000044600000  
x8 = 0x0000000082000210  
x9 = 0x0000000000000002  
x10 = 0x0000000000000000  
x11 = 0x00000000001151a0  
x12 = 0x0000000000118b80  
x13 = 0x00000000f3d73898  
x14 = 0x000000000011f1c0  
x15 = 0x0000000000109418  
x16 = 0x0000000080000145  
x17 = 0xffffffc000767ca8  
x18 = 0x0000000000000735  
x19 = 0x0000000000125003  
x20 = 0xffffffffc0000000  
x21 = 0x0000000040000000  
x22 = 0x0000000080000000  
x23 = 0x0000000000000001  
x24 = 0x000000000011e3e8  
x25 = 0x000000000011e400  
x26 = 0x0000000000122000  
x27 = 0xffffffc000094000  
x28 = 0xffffffc096310000  
x29 = 0x0000000000118a70  
scr\_el3 = 0x0000000000000735  
sctlr\_el3 = 0x0000000000cd183f  
cptr\_el3 = 0x0000000000000000  
tcr\_el3 = 0x0000000080803520  
daif = 0x00000000000002c0  
mair\_el3 = 0x00000000000004ff  
spsr\_el3 = 0x0000000080000145  
elr\_el3 = 0xffffffc000767ca8  
ttbr0\_el3 = 0x000000000011e3e0  
esr\_el3 = 0x000000005e000000  
far\_el3 = 0x028b084185615467  
spsr\_el1 = 0x00000000600e0010  
elr\_el1 = 0x00000000da577f48  
spsr\_abt = 0x0000000000000000  
spsr\_und = 0x0000000000000000  
spsr\_irq = 0x0000000000000000  
spsr\_fiq = 0x0000000000000000  
sctlr\_el1 = 0x0000000034d5d91d  
actlr\_el1 = 0x0000000000000000  
cpacr\_el1 = 0x0000000000300000  
csselr\_el1 = 0x0000000000000001  
sp\_el1 = 0xffffffc096313db0  
esr\_el1 = 0x000000000fe0241b  
ttbr0\_el1 = 0x0a740000645fe000  
ttbr1\_el1 = 0x000000004181a000  
mair\_el1 = 0x000000ff440c0400  
amair\_el1 = 0x0000000000000000  
tcr\_el1 = 0x00000032b5193519  
tpidr\_el1 = 0x00000000bee2d000  
tpidr\_el0 = 0x0000000000000000  
tpidrro\_el0 = 0x00000000da87b978  
dacr32\_el2 = 0x0000000000000000  
ifsr32\_el2 = 0x0000000000000000  
par\_el1 = 0x0000000000000000  
mpidr\_el1 = 0x0000000080000100  
afsr0\_el1 = 0x0000000000000000  
afsr1\_el1 = 0x0000000000000000  
contextidr\_el1 = 0x0000000000000000  
vbar\_el1 = 0xffffffc000085000  
cntp\_ctl\_el0 = 0x0000000000000005  
cntp\_cval\_el0 = 0x0000002c55959d3f  
cntv\_ctl\_el0 = 0x0000000000000000  
cntv\_cval\_el0 = 0x061804020400c040  
cntkctl\_el1 = 0x00000000000000a6  
fpexc32\_el2 = 0x0000000000000700  
sp\_el0 = 0x0000000000118a70  
cpuectlr\_el1 = 0x0000000000000041  
gic\_hppir = 0x0000000000000000  
gic\_ahppir = 0x0000000000000000  
gic\_ctlr = 0x0000000000000000  
gicd\_ispendr regs (Offsets 0x200 - 0x278)  
 Offset: value  
0000000000000200: 0x0000000000000000  
0000000000000208: 0x0000000000000000  
0000000000000210: 0x0000000000000000  
0000000000000218: 0x0000000000000000  
0000000000000220: 0x0000000000000000  
0000000000000228: 0x0000000000000000  
0000000000000230: 0x0000000000000000  
0000000000000238: 0x0000000000000000  
0000000000000240: 0x0000000000000000  
0000000000000248: 0x0000000000000000  
0000000000000250: 0x0000000000000000  
0000000000000258: 0x0000000000000000  
0000000000000260: 0x0000000000000000  
0000000000000268: 0x0000000000000000  
0000000000000270: 0x0000000000000000  
0000000000000278: 0x0000000000000000  
cci\_snoop\_ctrl\_cluster0 = 0x00000000c0000003  
cci\_snoop\_ctrl\_cluster1 = 0x00000000c0000000  
Dump last LR address : 0x00000000400826a4  
Dump address : 0x0000000000118a70  
......

有看到PANIC in EL3，这个表示是发生了panic，那么需要看谁调用了panic。这时就需要bl31.elf了（一定要同一版本编译出来才行。）

先看x30 = 0x000000000010e69c，X30是LR，看谁调用了panic：

\_\_\_\_\_\_\_\_\_\_\_\_\_\_addr/line|code\_\_\_\_\_|label\_\_\_\_|mnemonic\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|comment\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|  
 39|  
 ZSX:0010E680|A9BF7BFD \_\_assert: stp x29,x30,[SP,#-0x10]! ; x29,x30,[SP,#-16]!  
 ZSX:0010E684|AA0003E1 mov x1,x0  
 41|  
 ZSX:0010E688|D0000020 adrp x0,0x114000  
 39|  
 ZSX:0010E68C|910003FD mov x29,SP  
 41|  
 ZSX:0010E690|91200000 add x0,x0,#0x800 ; x0,x0,#2048  
 39|  
 41|  
 ZSX:0010E694|97FFFFF0 bl 0x10E654 ; printf  
 44|  
 ZSX:0010E698|940004C6 bl 0x10F9B0 ; el3\_panic  
 45|  
 ZSX:0010E69C|14000000\_\_\_\_\_\_\_\_\_\_\_\_b\_\_\_\_\_\_\_0x10E69C

 可以看到是\_\_assert()调用了panic，那么是谁调用了\_\_assert()呢？单单从这个函数看不出什么的。

 这时就需要手动还原调用栈了，前面的log里有将stack值打印到log里，当时的sp为：

* sp\_el0 = 0x0000000000118a70

0x118a70对应的内容如下：

Dump address : 0x0000000000118a70  
0000000000118a80  
000000000010e8c4  
0000000000118ad0  
000000000010ead0  
0000000044640000  
0000000000000d03  
0000000044600000  
0000000000030000  
000000000011e000  
0000000000000003  
000000000fe0241b  
00000000000000af  
0000000000118ae0  
000000000010ef9c  
0000000000118b40  
00000000001093d8  
0000000044640000  
0000000000030000  
0000000000000000  
000000000011d490  
ffffffc001281220  
0000000000000003  
000000000fe0241b  
00000000000000af  
ffffffc000094000  
ffffffc096310000  
ffffffc096313db0  
000000000010f7ec  
ffffffc096313ed0  
ffffffc096313dc0  
00000000da577f48  
00000000ee190f1d

根据stack如何推导调用栈呢，这就需要结合bl31.elf了。首先要知道这个log是在el3\_panic()函数印出来的，也就是说log里的SP是el3\_panic()函数的SP，因此要从el3\_panic()函数开始推导。先看下el3\_panic()函数的汇编：

\_\_\_\_\_\_\_\_\_\_\_\_\_\_addr/line|code\_\_\_\_\_|label\_\_\_\_\_\_|mnemonic\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|comment  
 320|  
 ZSX:0010F9B0|D50041BF el3\_panic: msr SPSel,#0x1 ; SPSel,#1  
 321|  
 ZSX:0010F9B4|9100001F mov SP,x0  
 ZSX:0010F9B8|D53ED040 mrs x0,#0x3,#0x6,c13,c0,#0x2 ; x0, TPIDR\_EL3  
 ZSX:0010F9BC|91008000 add x0,x0,#0x20 ; x0,x0,#32  
\_\_\_\_\_\_\_\_\_\_\_ZSX:0010F9C0|D51ED040\_\_\_\_\_\_\_\_\_\_\_\_\_\_msr\_\_\_\_\_#0x3,#0x6,c13,c0,#0x2,x0\_\_\_;\_\_TPIDR\_EL3,x0  
 ZSX:0010F9C4|F9000401 str x1,[x0,#0x8] ; x1,[x0,#8]  
 ZSX:0010F9C8|910003E1 mov x1,SP  
 ZSX:0010F9CC|F9000001 str x1,[x0]  
 322|  
 ZSX:0010F9D0|1002AB20 adr x0,0x114F34 ; x0,panic\_msg  
 323|  
 ZSX:0010F9D4|9100001F mov SP,x0  
......

 el3\_panic()函数没有压栈，因此看\_\_assert()函数，\_\_assert()函数的SP和el3\_panic()函数的SP相等，我们推导如下：

el3\_panic() <- SP 0x118a70  
\_\_assert() <- SP 0x118a70

我们来模拟\_\_assert()函数压栈过程，首先查看\_\_assert()函数的汇编：

\_\_\_\_\_\_\_\_\_\_\_\_\_\_addr/line|code\_\_\_\_\_|label\_\_\_\_|mnemonic\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|comment\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|  
 ZSX:0010E680|A9BF7BFD \_\_assert: stp x29,x30,[SP,#-0x10]! ; x29,x30,[SP,#-16]!  
 ZSX:0010E684|AA0003E1 mov x1,x0  
 |  
 41|  
 ZSX:0010E688|D0000020 adrp x0,0x114000  
 39|

可以看下SP压了X29,X30对应stack的内容如下：

Dump address : 0x0000000000118a70  
低地址  
0000000000118a80 <- X29  
000000000010e8c4 <- X30  
......  
高地址

000000000010e8c4地址对应的是：

\_\_\_\_\_\_\_\_\_\_\_\_\_\_addr/line|code\_\_\_\_\_|label\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|mnemonic\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|comment  
 ZSX:0010E714|12800104 init\_xlation\_table: movn w4,#0x8 ; w4,#8  
 201|  
 ZSX:0010E718|71000C7F cmp w3,#0x3 ; w3,#3  
 195|  
 ZSX:0010E71C|A9BB7BFD stp x29,x30,[SP,#-0x50]! ; x29,x30,[SP,#-80]!  
 196|  
 ......  
 ZSX:0010E8AC|D0000023 adrp x3,0x114000  
 ZSX:0010E8B0|91270000 add x0,x0,#0x9C0 ; x0,x0,#2496  
 ZSX:0010E8B4|91220021 add x1,x1,#0x880 ; x1,x1,#2176  
 ZSX:0010E8B8|52801DA2 mov w2,#0xED ; w2,#237  
 ZSX:0010E8BC|91229463 add x3,x3,#0x8A5 ; x3,x3,#2213  
 ZSX:0010E8C0|97FFFF70 bl 0x10E680 ; \_\_assert  
 241|  
 ZSX:0010E8C4|110006E3\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_add\_\_\_\_\_w3,w23,#0x1\_\_\_\_\_\_;\_w3,w23,#1

已经找到调用\_\_assert()函数的函数了，是init\_xlation\_table()函数。我们画出堆栈图：

el3\_panic() <- SP 0x118a70  
\_\_assert() <- SP 0x118a70  
init\_xlation\_table() <- SP 0x118a80

接着往下推导，查看init\_xlation\_table()函数的汇编，压入了X20~X26和X29~X30，推导如下：

Dump address : 0x0000000000118a70  
低地址  
0000000000118a80  
000000000010e8c4  
0000000000118ad0 <- X29  
000000000010ead0 <- X30  
0000000044640000 <- X19  
0000000000000d03 <- X20  
0000000044600000 <- X21  
0000000000030000 <- X22  
000000000011e000 <- X23  
0000000000000003 <- X24  
000000000fe0241b <- X25  
00000000000000af <- X26  
......  
高地址

000000000010ead0地址对应的是：

\_\_\_\_\_\_\_\_\_\_\_\_\_\_addr/line|code\_\_\_\_\_|label\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|mnemonic\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|comment  
 ZSX:0010EAAC|A9BF7BFD init\_xlat\_tables: stp x29,x30,[SP,#-0x10]! ; x29,x30,[SP,#-16]!  
 285|  
 ZSX:0010EAB0|90000080 adrp x0,0x11E000  
 ZSX:0010EAB4|90000082 adrp x2,0x11E000  
 283|  
 ZSX:0010EAB8|910003FD mov x29,SP  
 285|  
 ZSX:0010EABC|D2800001 mov x1,#0x0 ; x1,#0  
 ZSX:0010EAC0|9106A000 add x0,x0,#0x1A8 ; x0,x0,#424  
 ZSX:0010EAC4|910F8042 add x2,x2,#0x3E0 ; x2,x2,#992  
 ZSX:0010EAC8|52800023 mov w3,#0x1 ; w3,#1  
 283|  
 285|  
\_\_\_\_\_\_\_\_\_\_\_ZSX:0010EACC|97FFFF12\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_bl\_\_\_\_\_\_0x10E714\_\_\_\_\_\_\_\_\_;\_init\_xlation\_table  
 286|  
 ZSX:0010EAD0|90000080\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_adrp\_\_\_\_x0,0x11E000

已经找到调用init\_xlation\_table()函数的函数了，是init\_xlat\_tables()函数。我们画出堆栈图：

el3\_panic() <- SP 0x118a70  
\_\_assert() <- SP 0x118a70  
init\_xlation\_table() <- SP 0x118a80  
init\_xlat\_tables() <- SP 0x118ad0

接着往下推导，查看init\_xlat\_tables()函数的汇编，压入了X29~X30，推导如下：

Dump address : 0x0000000000118a70  
低地址  
0000000000118a80  
000000000010e8c4  
0000000000118ad0  
000000000010ead0  
0000000044640000  
0000000000000d03  
0000000044600000  
0000000000030000  
000000000011e000  
0000000000000003  
000000000fe0241b  
00000000000000af  
0000000000118ae0 <- X29  
000000000010ef9c <- X30  
......  
高地址

000000000010ef9c地址对应的是：

\_\_\_\_\_\_\_\_\_\_\_\_\_\_addr/line|code\_\_\_\_\_|label\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|mnemonic\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|comment  
 ZSX:0010EF34|A9BA7BFD mt\_icache\_dump: stp x29,x30,[SP,#-0x60]! ; x29,x30,[SP,#-96]!  
 ZSX:0010EF38|910003FD mov x29,SP  
 ZSX:0010EF3C|A90363F7 stp x23,x24,[SP,#0x30] ; x23,x24,[SP,#48]  
 222|  
 ZSX:0010EF40|90000097 adrp x23,0x11E000  
......  
 ZSX:0010EF8C|D2A00402 mov x2,#0x200000 ; x2,#2097152  
 ZSX:0010EF90|528000E3 mov w3,#0x7 ; w3,#7  
 ZSX:0010EF94|97FFFE5E bl 0x10E90C ; mmap\_add\_region  
 231|  
 ZSX:0010EF98|97FFFEC5 bl 0x10EAAC ; init\_xlat\_tables  
 232|  
 ZSX:0010EF9C|52800020\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_mov\_\_\_\_\_w0,#0x1\_\_\_\_\_\_\_\_\_\_;\_w0,#1

已经找到调用init\_xlat\_tables()函数的函数了，是mt\_icache\_dump()函数。我们画出堆栈图：

el3\_panic() <- SP 0x118a70  
\_\_assert() <- SP 0x118a70  
init\_xlation\_table() <- SP 0x118a80  
init\_xlat\_tables() <- SP 0x118ad0  
mt\_icache\_dump() <- SP 0x118ae0

接着往下推导，查看mt\_icache\_dump()函数的汇编，压入了X19~X30，推导如下：

Dump address : 0x0000000000118a70  
低地址  
0000000000118a80  
000000000010e8c4  
0000000000118ad0  
000000000010ead0  
0000000044640000  
0000000000000d03  
0000000044600000  
0000000000030000  
000000000011e000  
0000000000000003  
000000000fe0241b  
00000000000000af  
0000000000118ae0  
000000000010ef9c  
0000000000118b40 <- X29  
00000000001093d8 <- X30  
0000000044640000 <- X19  
0000000000030000 ​<- X20  
0000000000000000 <- X21  
000000000011d490 <- X22  
ffffffc001281220 <- X23  
0000000000000003 <- X24  
000000000fe0241b <- X25  
00000000000000af <- X26  
ffffffc000094000 <- X27  
ffffffc096310000 <- X28  
ffffffc096313db0  
000000000010f7ec  
ffffffc096313ed0  
ffffffc096313dc0  
00000000da577f48  
00000000ee190f1d  
......  
高地址

00000000001093d8地址对应的是：

\_\_\_\_\_\_\_\_\_\_\_\_\_\_addr/line|code\_\_\_\_\_|label\_\_\_\_\_\_\_\_\_\_\_\_|mnemonic\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|comment  
 166|  
 ZSX:001085DC|A9BC7BFD sip\_smc\_handler: stp x29,x30,[SP,#-0x40]! ; x29,x30,[SP,#-64]!  
 ZSX:001085E0|2A0003E5 mov w5,w0  
 178|  
 ZSX:001085E4|528077E0 mov w0,#0x3BF ; w0,#959  
......  
 498|  
 ZSX:001093C0|2A1303E0 mov w0,w19  
 ZSX:001093C4|97FFFC54 bl 0x108514 ; LittleDREQSWEn  
 ZSX:001093C8|17FFFF20 b 0x109048  
 507|  
 ZSX:001093CC|AA1303E0 mov x0,x19  
 ZSX:001093D0|AA1403E1 mov x1,x20  
 ZSX:001093D4|940016D8 bl 0x10EF34 ; mt\_icache\_dump  
 ZSX:001093D8|17FFFF1C\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_b\_\_\_\_\_\_\_0x109048

已经找到调用mt\_icache\_dump()函数的函数了，是sip\_smc\_handler()函数。我们画出堆栈图：

el3\_panic() <- SP 0x118a70  
\_\_assert() <- SP 0x118a70  
init\_xlation\_table() <- SP 0x118a80  
init\_xlat\_tables() <- SP 0x118ad0  
mt\_icache\_dump() <- SP 0x118ae0  
sip\_smc\_handler() <- SP 0x118b40

到这里就不用推导了，因为sip\_smc\_handler()函数是ATF层SMC的系统调用接口。

我们需要回头看assert的原因了，对应init\_xlation\_table()代码：

192static mmap\_region\_t \*init\_xlation\_table(mmap\_region\_t \*mm,  
193 unsigned long base\_va,  
194 unsigned long \*table, unsigned level)  
195{  
......  
232 /\* else Next region only partially covers area, so need \*/  
233  
234 if (desc == UNSET\_DESC) {  
235 /\* Area not covered by a region so need finer table \*/  
236 unsigned long \*new\_table = xlat\_tables[next\_xlat++];  
237 assert(next\_xlat <= MAX\_XLAT\_TABLES); /\* 这里发生assert \*/

这里发生assert的原因是ATF MMU table是预先分配好的内存，通过next\_xlat挨个分配出去，这里assert表示已经超出预先分配的值MAX\_XLAT\_TABLES。

table不够用的原因是有人加mmu mapping region多了。是谁加的呢，从调用栈上看是mt\_icache\_dump()了，这个函数有什么用呢？经过确认是调试用的，会在db里生成SYS\_ICACHE\_DUMP文件，用于调试检查icache是否存在bitflip的。

那这个是否合理呢？看了mt\_ichace\_dump()代码，发现存在race condition的情况：

212int mt\_icache\_dump(unsigned long addr, unsigned long size)  
213{  
214 static int region\_added = 0;  
215 uint64\_t midr\_partnum;  
216 int ret = 0;  
217 midr\_partnum = (read\_midr() >> 4) & 0xFFF;  
218  
219 /\* Log starts here... \*/  
220 //set\_uart\_flag();  
221  
222 if(!region\_added) { /\* 这里存在race condition，可能多个CPU同时进来，导致多次添加mmu region \*/  
223 printf("mmap cache dump buffer : 0x%lx, 0x%lx\n\r", addr, size);  
224  
225 mmap\_add\_region(addr & ~(PAGE\_SIZE\_2MB\_MASK),  
226 addr & ~(PAGE\_SIZE\_2MB\_MASK),  
227 PAGE\_SIZE\_2MB,  
228 MT\_MEMORY | MT\_RW | MT\_NS);  
229  
230 /\* re-fill translation table \*/  
231 init\_xlat\_tables();  
232 region\_added = 1; /\* 这里才标记为1，很晚了。 \*/  
233 printf("mmap cache dump buffer (force 2MB aligned): 0x%lx, 0x%lx\n\r",  
234 addr & ~(PAGE\_SIZE\_2MB\_MASK), PAGE\_SIZE\_2MB);  
235 }

直接将kernel层对应的mt\_icache\_dump()函数注释验证，就再也没出现ATF crash了。

为什么有人频繁调用mt\_icache\_dump()，后来有查到是app经常发生undef inst.异常，在该异常下会调用mt\_icache\_dump()。

**根本原因：**

ATF的mt\_icache\_dump()函数存在race condition导致多次增加mmu region引起assert。

**解决方法：**

使用spin lock保证只有一次进入。

int mt\_icache\_dump(unsigned long addr, unsigned long size)  
{  
 static spinlock\_t dumplock;  
 static int region\_added = 0;  
 uint64\_t midr\_partnum;  
 int ret = 0, has\_added;  
  
 midr\_partnum = (read\_midr() >> 4) & 0xFFF;  
 spin\_lock(&dumplock);  
 has\_added = region\_added;  
 region\_added = 1;  
 spin\_unlock(&dumplock);  
 if(!has\_added) {  
 ......

**结语：**

只有基础扎实才能轻松解决各类不同的问题，像上面的栈推导都是很基础的汇编分析。

涉及知识点

* AAPCS栈推导
* ATF MMU映射架构

#### ATF assert导致HW reboot(2)

**问题背景：**

升级大版本后，开机待机就HW reboot了。

**分析过程：**

拿到2份db，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SpOfflineDebugSuite分析，发现是ATF crash：

== 异常报告v1.5(仅供参考) ==  
报告解读: MediaTek On-Line> Quick Start> E-Consulter之NE/KE分析报告解读> KE分析报告  
详细描述: 发生ATF crash, 请分析SYS\_ATF\_CRASH  
异常时间: 0.000000秒, Tue May 23 09:47:41 CST 2017  
  
== CPU信息 ==  
其他CPU信息:  
 CPU0: 进程名: (null), 进程标识符(pid): 0  
 寄存器异常: SP(0xFFFFFFC0AB277A70)/FP(0x0000000000119BC0)不在同一内核栈  
 本地调用栈:  
 ...... 0x00000000001123E8()  
 == 栈结束 ==  
  
  
这题还需bl31.elf文件, 该文件必须和db匹配。  
详情请查看MOL上的'[FAQ15094]发生ATF CRASH需要提供哪些辅助文件给MTK'

db里有发现SYS\_ATF\_CRASH，因此这题是atf crash。

检查SYS\_ATF\_CRASH，log如下：

[ATF](0)[119.192103]ASSERT: mmap\_add\_region : mm\_last->size == 0  
)  
PANIC in EL3 at x30 = 0x000000000011037c  
x0 = 0x0000000000000000  
x1 = 0x0000000000000001  
x2 = 0x000000006ffa26a0  
x3 = 0x00000000ffffffe0  
x4 = 0x0000000000000031  
x5 = 0x0000000000000003  
x6 = 0x0000000000000000  
x7 = 0xfeff64716e73726f  
x8 = 0x0000000000121b50  
x9 = 0x0000000000000000  
x10 = 0x000000008200ff00  
x11 = 0x0000000000117e80  
x12 = 0x0000000000119c80  
x13 = 0x0000000000000028  
x14 = 0x00000000001219c8  
x15 = 0x0000000000101fe4  
x16 = 0x00000000001082fc  
x17 = 0xffffff800845f930  
x18 = 0x0000000000000735  
x19 = 0x0000000000001000  
x20 = 0x00000000001231f0  
x21 = 0x0000000000123030  
x22 = 0x0000000044a02000  
x23 = 0x0000000044a02000  
x24 = 0xffffff800952a210  
x25 = 0xffffff80090375e0  
x26 = 0x0000000000000040  
x27 = 0xffffff8008d21000  
x28 = 0xffffffc0ab274000  
x29 = 0x0000000000119bc0  
scr\_el3 = 0x0000000000000735  
sctlr\_el3 = 0x0000000000cd183f  
cptr\_el3 = 0x0000000000000000  
tcr\_el3 = 0x0000000080803520  
daif = 0x00000000000002c0  
......

有看到PANIC in EL3，这个表示是发生了panic，那么需要看谁调用了panic，不过这题前面有打印assert，表示有人调用assert引起了panic。通过assert就可以找到对应代码

void mmap\_add\_region(unsigned long base\_pa, unsigned long base\_va, unsigned long size, unsigned attr)  
{  
 mmap\_region\_t \*mm = mmap;  
 mmap\_region\_t \*mm\_last = mm + ARRAY\_SIZE(mmap) - 1;  
 unsigned long pa\_end = base\_pa + size - 1;  
 unsigned long va\_end = base\_va + size - 1;  
  
 assert(IS\_PAGE\_ALIGNED(base\_pa));  
 assert(IS\_PAGE\_ALIGNED(base\_va));  
 assert(IS\_PAGE\_ALIGNED(size));  
  
 if (!size)  
 return;  
  
 /\* Find correct place in mmap to insert new region \*/  
 while (mm->base\_va < base\_va && mm->size) {  
 ++mm;  
 }  
 /\* Make room for new region by moving other regions up by one place \*/  
 memmove(mm + 1, mm, (uintptr\_t)mm\_last - (uintptr\_t)mm);  
  
 /\* Check we haven't lost the empty sentinal from the end of the array \*/  
 assert(mm\_last->size == 0); /\* 这里发生了assert \*/  
 ......  
}

 这个assert表示mmap已经添加到上限！因此要看下谁不停的添加mmap，需要还原调用栈。还原调用栈的方法请参考本文的《ATF assert导致HW reboot》章节

最终我们检查到时候rpmt\_init()不停的添加了mmap，在SYS\_ATF\_CRASH搜索log可以看到：

 Line 659: [ATF](0)[144.675272]INFO: [rpmb\_init] invoked from lk  
 Line 1402: [ATF](0)[115.486522]INFO: [rpmb\_init] invoked from lk  
 Line 1410: [ATF](0)[165.599550]INFO: [rpmb\_init] invoked from lk  
 Line 1768: [ATF](0)[116.828221]INFO: [rpmb\_init] invoked from lk  
 Line 1776: [ATF](0)[134.327828]INFO: [rpmb\_init] invoked from lk  
 Line 2191: [ATF](0)[119.192032]INFO: [rpmb\_init] invoked from lk

rpmt\_init()对应的代码：

int32\_t rpmb\_init(void)  
{  
 uint32\_t phy\_addr = LK\_SHARED\_MEM\_ADDR;  
#ifdef MTK\_RPMB\_DEBUG  
 uint64\_t \*shared\_mem\_ptr;  
 uint64\_t offset = 0;  
  
 set\_uart\_flag();  
 INFO("[%s] in rpmb shared mem smc, phy\_addr:0x%x rpmb\_is\_in\_lk:0x%x!!\n", \_\_func\_\_, phy\_addr, rpmb\_is\_in\_lk);  
#endif  
  
 /\* return err if it's already invoked in lk \*/  
 if (MTK\_RPMB\_LEAVING\_LK == rpmb\_is\_in\_lk) {  
 INFO("[%s] no longer in lk, return immediately\n", \_\_func\_\_);  
 return SIP\_SVC\_E\_NOT\_SUPPORTED;  
 } else if (MTK\_RPMB\_IN\_LK == rpmb\_is\_in\_lk) {  
 INFO("[%s] invoked from lk\n", \_\_func\_\_);  
 }  
 /\* map physcial memory for 4KB size. assert inside mmap\_add\_region if error occurs. no need to check return value in this case \*/  
 mmap\_add\_region((uint64\_t)(phy\_addr & ~(PAGE\_SIZE\_MASK)), (uint64\_t)(phy\_addr & ~(PAGE\_SIZE\_MASK)), PAGE\_SIZE, MT\_DEVICE | MT\_RW | MT\_NS);

所以有人不停的调用rpmb\_init()，对应代码：

uint64\_t mediatek\_sip\_handler(uint32\_t smc\_fid, uint64\_t x1, uint64\_t x2, uint64\_t x3, uint64\_t x4, void \*cookie, void \*handle, uint64\_t flags)  
{  
......  
 /\* Determine which security state this SMC originated from \*/  
 ns = is\_caller\_non\_secure(flags);  
 if (!ns) {  
......  
 } else {  
 switch (smc\_fid) {  
......  
/\* Disable for ep \*/  
#ifdef \_\_MTK\_RPMB  
 case MTK\_SIP\_LK\_RPMB\_INIT\_AARCH32:  
 case MTK\_SIP\_LK\_RPMB\_INIT\_AARCH64:  
 /\* create shared memory for rpmb atf module \*/  
 rpmb\_init();  
 break;  
......  
}

rpmb\_init()是通过SMC call调用下来的。查找整个kernel，都没有代码通过SMC call调用下来，而且通过宏定义MTK\_SIP\_LK\_RPMB\_INIT\_AARCH32明显是从lk调用过来的，而出问题时是在kernel。所以问题就扑朔迷离了。

需要一种方法可以获取陷入EL3前的PC/LR值，拿到PC/LR就可以知道什么地方通过SMC call调用下来。代码添加如下：

int32\_t rpmb\_init(void)  
{  
 int i;  
 uint32\_t phy\_addr = LK\_SHARED\_MEM\_ADDR;  
 cpu\_context\_t \*ns\_cpu\_context; // add this line  
 uint64\_t mpidr, aa,bb,cc,dd; // add this line  
 uint32\_t linear\_id; // add this line  
  
#ifdef MTK\_RPMB\_DEBUG  
 uint64\_t \*shared\_mem\_ptr;  
 uint64\_t offset = 0;  
  
 set\_uart\_flag();  
 INFO("[%s] in rpmb shared mem smc, phy\_addr:0x%x rpmb\_is\_in\_lk:0x%x!!\n", \_\_func\_\_, phy\_addr, rpmb\_is\_in\_lk); #endif  
 init\_count\_rpmb++; // add this line  
 if (init\_count\_rpmb > 2) { // add this block  
 mpidr = read\_mpidr();  
 linear\_id = platform\_get\_core\_pos(mpidr);  
 ns\_cpu\_context = cm\_get\_context\_by\_mpidr(mpidr, NON\_SECURE);  
 aa = SMC\_GET\_EL3(ns\_cpu\_context, CTX\_ELR\_EL3);  
 bb = SMC\_GET\_GP(ns\_cpu\_context, (CTX\_GPREG\_LR));  
 cc = read\_ctx\_reg(get\_sysregs\_ctx(ns\_cpu\_context), CTX\_SP\_EL1);  
 dd = SMC\_GET\_EL3(ns\_cpu\_context, CTX\_SPSR\_EL3);  
 INFO("(%d) pc:<%016lx> lr:<%016lx> sp:<%016lx> pstate=%x\n", (int)linear\_id, aa, bb, cc, (uint32\_t)dd);  
 assert(0);  
 }  
......

重新复现，抓到hw reboot db，解开看SYS\_ATF\_CRASH：

[ATF](0)[112.037433]INFO: rpmb\_init: on cpu0  
[ATF](0)[112.037504]INFO: (0) pc:<ffffff8008463828> lr:<ffffff8008463f2c> sp: pstate=800001c5  
[ATF](0)[112.038987]ASSERT: rpmb\_init : 0

拿到PC，通过对vmlinux进行addr2line，看到是：

wake\_reason\_t spm\_go\_to\_sleep(u32 spm\_flags, u32 spm\_data)  
{  
 ......  
 mt\_secure\_call(MTK\_SIP\_KERNEL\_SPM\_ARGS, SPM\_ARGS\_PCM\_WDT, 0, 0); /\* PC落在这里 \*/  
 ......  
}

查看MTK\_SIP\_KERNEL\_SPM\_ARGS：

#define MTK\_SIP\_KERNEL\_SPM\_ARGS (0x82000228 | MTK\_SIP\_SMC\_AARCH\_BIT)

和MTK\_SIP\_LK\_RPMB\_INIT对应不上呀：

#define MTK\_SIP\_LK\_RPMB\_INIT\_AARCH32 0x82000103  
#define MTK\_SIP\_LK\_RPMB\_INIT\_AARCH64 0xC2000103

只能看下mediatek\_sip\_handler()是如何处理MTK\_SIP\_KERNEL\_SPM\_ARGS的：

......  
 case MTK\_SIP\_KERNEL\_SPM\_ARGS\_AARCH32:  
 case MTK\_SIP\_KERNEL\_SPM\_ARGS\_AARCH64:  
 spm\_args(x1, x2, x3);  
#ifdef MTK\_ICCS\_SUPPORT  
 case MTK\_SIP\_KERNEL\_ICCS\_STATE\_AARCH32:  
 case MTK\_SIP\_KERNEL\_ICCS\_STATE\_AARCH64:  
 rc = iccs\_state(x1 & 0xff, x2 & 0xff, x3 & 0xff);  
 break;  
#endif  
/\* Disable for ep \*/  
#ifdef \_\_MTK\_RPMB  
 case MTK\_SIP\_LK\_RPMB\_INIT\_AARCH32:  
 case MTK\_SIP\_LK\_RPMB\_INIT\_AARCH64:  
 /\* create shared memory for rpmb atf module \*/  
 rpmb\_init();  
 break;  
......

发现MTK\_SIP\_KERNEL\_SPM\_ARGS\_AARCH32、MTK\_SIP\_KERNEL\_SPM\_ARGS\_AARCH64的case下没有break！！执行完后直接跑到rpmb\_init()里了！问题就出在这里。

**根本原因：**

MTK\_SIP\_KERNEL\_SPM\_ARGS\_AARCH32、MTK\_SIP\_KERNEL\_SPM\_ARGS\_AARCH64的case下没有break！

**解决方法：**

增加break

**结语：**

涉及知识点

* EL1/EL3 context获取方法
* ATF MMU映射架构

#### ATF未定义指令异常

**问题背景：**

产线压力测试，出现1台机器多次crash。

**分析过程：**

拿到4份db，用GAT解开db，其中2份是HW reboot，利用工具SpOfflineDebugSuite分析，发现是ATF crash：

== 异常报告v1.7(仅供参考) ==  
报告解读: MediaTek On-Line> Quick Start> NE/KE分析报告解读> KE分析报告  
详细描述: ATF 未定义指令异常，请结合崩溃进程调用栈检查相关代码  
平台 : MT6765  
版本 : alps-mp-o1.mp6/user build  
异常时间: 156.507718秒, Thu Aug 23 18:34:39 CST 2018  
  
  
== CPU信息 ==  
崩溃CPU信息:  
 CPU0: ATF, 中断: 关  
  
== 日志信息 ==  
atf log:  
[ATF](0)[1.090373]ERROR: mediatek\_plat\_sip\_handler: unhandled N-Sec SMC (0x82000230)  
[ATF](6)[3.416833]INFO: kernel time sync 0x[ATF](6)K:[2.573130]ERROR: reset pcm(PCM\_FSM\_STA=0x2048490)  
  
  
如果是ATF crash，麻烦贵司以后提供db的同时提供对应的bl31.elf，我司即可迅速分析，减少不必要的延误。  
这题还需out/target/product/$proj/trustzone/ATF\_OBJ/{debug|release}/bl31/bl31.elf文件, 该文件必须和db匹配, 对应时间戳是:  
Built : 21:45:31, Aug 13 2018  
详情请查看MOL上的'[FAQ15094]发生ATF CRASH需要提供哪些辅助文件给MTK'

根据提示将对应的bl31.elf放入symbols目录，重新跑工具分析：

== 异常报告v1.7(仅供参考) ==  
报告解读: MediaTek On-Line> Quick Start> NE/KE分析报告解读> KE分析报告  
详细描述: ATF 未定义指令异常，请结合崩溃进程调用栈检查相关代码  
平台 : MT6765  
版本 : alps-mp-o1.mp6/user build  
异常时间: 156.507718秒, Thu Aug 23 18:34:39 CST 2018  
  
  
== CPU信息 ==  
崩溃CPU信息:  
 CPU0: ATF, 中断: 关  
  
== 日志信息 ==  
atf log:  
[ATF](0)[1.090373]ERROR: mediatek\_plat\_sip\_handler: unhandled N-Sec SMC (0x82000230)  
[ATF](6)[3.416833]INFO: kernel time sync 0x[ATF](6)K:[2.573130]ERROR: reset pcm(PCM\_FSM\_STA=0x2048490)

看到是未定义指令，直接启动trace32查看，双击debug.cmm：

#### 

可以看到PC指向的指令确实是undef的，不过奇怪的是前后的指令都是正常的，为何就这条指令异常了呢？

我们单独反汇编bl31.elf或再开启一个T32加载bl31.elf：

* data.load.elf bl31.elf

#### 

bl31.elf里面对应的地址里的指令是正常的，我们对比发现差了1个bit：A20203F7 != AA0203F7。

**根本原因：**

指令发生bitflip导致了ATF crash。

**解决方法：**

2个db都是在同样的位置发生bitflip，优先检查DDR，然后才是Soc。

**结语：**

SpOfflineDebugSuit支持ATF debugging，可以方便的还原调用栈等信息，使问题一目了然。

**HW reboot案例分析**

#### TEE驱动大量log导致HW reboot

**问题背景：**

跑camera monkey 出现两笔hw reboot。

**分析过程：**

拿到2份db，用GAT解开db，利用工具E-Consulter分析，发现是HW reboot：

== 异常报告v1.5(仅供参考) ==  
报告解读: MediaTek On-Line> Quick Start> E-Consulter之NE/KE分析报告解读> KE分析报告  
详细描述: 发生HW reboot, 检查CPU是否卡死或硬件故障  
平台 : MT6757  
版本 : alps-mp-m1.mp3/userdebug build  
异常时间: 0.000000秒, Sun Nov 13 11:45:47 CST 2016  
  
== CPU信息 ==  
其他CPU信息:  
 CPU0: 进程名: (null), 进程标识符(pid): 0  
 寄存器异常: SP(0xFFFFFFC0B4FAF220)/FP(0x00000000001119F0)不在同一内核栈  
 本地调用栈:  
 ...... 0x000000000010B790()  
 == 栈结束 ==  
  
 CPU1: 进程名: (null), 进程标识符(pid): 0  
 寄存器异常: SP(0xFFFFFFC063DC3ED0)/FP(0x0000000000112290)不在同一内核栈  
 本地调用栈:  
 ...... 0x000000000010C010()  
 == 栈结束 ==

可以看到CPU0和CPU1都在ATF的范围内

* 知识点：除了35/53系列，其他的ATF都在SRAM里跑，因此地址范围是SRAM的地址范围。

可以通过aarch64-linux-android-addr2line -Cfe bl31.elf 0x000000000010B790 0x000000000010C010

结果如下：

console\_core\_putc  
vendor/mediatek/proprietary/trustzone/atf/v1.2/plat/mediatek/mt6757/drivers/uart/8250\_console.S:145  
wfi\_spill  
vendor/mediatek/proprietary/trustzone/atf/v1.2/services/std\_svc/psci/psci\_entry.S:122

看到CPU0在印log，CPU1在WFI里，一般在WFI里，要吗是idle，要吗是power off了。因此查看CPU1的hotplug状态，检查SYS\_REBOOT\_REASON里的：

CPU 1  
 irq: enter(0, 0) quit(0, 0)  
 hotplug: 54 ==>表示已经power off  
 cpu\_dormant: 0x0

* 知识点：54是什么意思呢？看代码arch\arm64\kernel\smp.c

void cpu\_die(void)  
{  
 unsigned int cpu = smp\_processor\_id();  
  
 aee\_rr\_rec\_hotplug\_footprint(cpu, 51);  
 idle\_task\_exit();  
 aee\_rr\_rec\_hotplug\_footprint(cpu, 52);  
 local\_irq\_disable();  
 aee\_rr\_rec\_hotplug\_footprint(cpu, 53);  
 /\* Tell \_\_cpu\_die() that this CPU is now safe to dispose of \*/  
 complete(&cpu\_died);  
 aee\_rr\_rec\_hotplug\_footprint(cpu, 54);  
 /\*  
 \* Actually shutdown the CPU. This must never fail. The specific hotplug  
 \* mechanism must perform all required cache maintenance to ensure that  
 \* no dirty lines are lost in the process of shutting down the CPU.  
 \*/  
 cpu\_ops[cpu]->cpu\_die(cpu);  
 aee\_rr\_rec\_hotplug\_footprint(cpu, 55);  
 BUG();  
}

可以看到，在cpu\_die函数里，每走一步会埋一个脚印，所以54是脚印，表示已走到这里。这里54表示cpu已power off

正常情况下，CPU1 power off了，那么last pc就为0了，为何显示在0x000000000010C010呢？这说明power off没有走完，因此需要检查hotplug的状态，继续查看SYS\_REBOOT\_REASON：

CPU notifier status: 8, 55, 0x0 ====>8表示CPU\_DYING

* 知识点：8是CPU\_DYING的值，7是CPU\_DEAD，看kernel代码就知道了
* 55表示第几个回调函数，结合开机的log就可以知道回调函数的名称。
* 0x0表示函数地址，这个可以结合vmlinux，通过addr2line定位

正常CPU power off流程CPU\_DOWN\_PREPARE -> CPU\_DYING -> CPU\_DEAD -> CPU\_POST\_DEAD。

CPU1明显没走完。从SYS\_REBOOT\_REASON可以看出：

CPU\_UP\_PREPARE: 176637646266  
CPU\_STARTING: 176637646959  
CPU\_ONLINE: 176637647901  
CPU\_DOWN\_PREPARE: 176640416422  
CPU\_DYING: 176640417095  
CPU\_DEAD: 176640334756  
CPU\_POST\_DEAD: 176640335323

上面的时间单位是us，可以看到CPU\_DYING是最新的时间，表示CPU\_DEAD和CPU\_POST\_DEAD还没走到。

而当前只有CPU0和CPU1活着，因此只能是CPU0将CPU1 power off，而CPU0在console\_core\_putc，在打印uart log。

这时看SYS\_ATF\_LAST，发现大量的如下的log：

[ATF](0)[176700.018474]fiq id [ffffffff]cid[0]  
[ATF](0)[176700.019001]fiq id [ffffffff]cid[0]  
[ATF](0)[176700.019529]fiq id [ffffffff]cid[0]  
[ATF](0)[176700.020057]fiq id [ffffffff]cid[0]  
[ATF](0)[176700.020584]fiq id [ffffffff]cid[0]  
[ATF](0)[176700.021112]fiq id [ffffffff]cid[0]

而这个log在代码里属于TEE驱动打印出来的，因此需要第3方修复这个问题。经过排查发现，这是atf wdt flow没导入引起的问题，因此需要patch：ALPS03021943（M1.MP3版本用TEEI才有问题）

**结语：**

需要多方面的知识将表象的问题一一排除，最终找到root cause。

涉及知识点

* CPU hotplug分析

#### TP使用SPI2接口高概率引起bus hang

**问题背景：**

开发过程中遇到SODI3下HW reboot，存在SPI2 bus hang。

CPU：MT6765

版本：P0.MP3

**分析过程：**

问题容易出现，复现的db稳定在SODI3下SP2 bus hang引起的HW reboot。

== 异常报告v1.7(仅供参考) ==  
报告解读: MediaTek On-Line> Quick Start> NE/KE分析报告解读> KE分析报告  
详细描述: Peri/Infra总线超时, 请检查SYS\_LAST\_CPU\_BUS  
参考信息: RD Window(Hata Tang)  
平台 : MT6765  
版本 : alps-mp-p0.mp3/eng build  
异常时间: 0.000000秒, Wed Jan 2 09:59:47 CST 2019  
  
  
== 平台信息 ==  
-- CPU --  
 L opp: 15( 612mV, 900MHz) 6638.447990s  
 LL opp: 15( 612mV, 400MHz) 6655.000690s  
CCI opp: 15( 600mV, 300MHz) 6655.000690s  
vproc: 612mV, vsram: 850mV  
-- DDR4(4G) --  
vcore dvfs opp: 15(650mV, 1534MHz)  
-- Low Power --  
SPM last pc: 0x562  
State: sodi3/FW

问题可能在SPI2身上，经过调查SPI2被用来接0-flash TP，而这个项目第一次使用SPI2，因此怀疑TP不正常使用SPI2引起了bus hang。在SODI3下bus hang，可以推导出：

* AP已经idle，因此不可能是TP driver传输数据引发，可能是TP slave有误触发SPI2传输。

不过这似乎也不可能，因为SPI和I2C一样，slave属于被动元件，无法主动发起传输。请TP/SPI2工程师进场分析：

* review TP代码
* 加log夹发生异常的位置

刚好客户找到必现方法：不插充电器，快速遮挡红外，百分百死机。有夹到log发现是加载固件时死机了。

再仔细判断场景，AP idle下才会进SODI3，那么为何当时在加载固件？经过分析发现TP driver加载固件会等待SPI2传输完成，这时AP可能进idle的。

那么问题就明显了，一定是SPI2配置有问题导致在SODI3下误动作去读写bus了，和low power工程师讨论这题，直接给出方向：

* 进出SODI3/SODI/DPIDLE是有条件的，进入前会判断某些clk是否关闭，如果是打开则不会进入对应的low power状态。

这里就怀疑到SPI2没有在判断的范围内，经过读取idle\_cond\_mask内容，发现SPI2果然不在判断范围内，那么很有可能是在SPI2工作的时候进入了SODI3，导致SODI3下bus hang了。

将SPI2加入判断范围，对应代码：

kernel-4.9/drivers/misc/mediatek/base/power/spm/mt6765/mtk\_idle\_cond\_check.c

static unsigned int idle\_cond\_mask[NR\_IDLE\_TYPES][NR\_CG\_GRPS] = {  
 [IDLE\_TYPE\_DP] = {  
 0x04000038, /\* MTCMOS, 26:VCODEC,5:ISP,4:MFG,3:DIS \*/  
 0x08040802, /\* INFRA0, 27:dxcc\_sec\_core\_cg\_sta \*/  
 0x00BFB800, /\* INFRA1, 8:icusb\_cg\_sta (removed) \*/  
 0x000000C5, /\* INFRA2 \*/  
 0x3FFFFFFF, /\* MMSYS0 \*/  
 0x00000000, /\* MMSYS1 \*/  
 },  
 [IDLE\_TYPE\_SO3] = {  
 0x04000030, /\* MTCMOS, 26:VCODEC,5:ISP,4:MFG \*/  
 0x0A040802, /\* INFRA0, 27:dxcc\_sec\_core\_cg\_sta \*/  
 0x00BFB800, /\* INFRA1, 8:icusb\_cg\_sta (removed) \*/  
 0x000000D1, /\* INFRA2 \*/ **/\* 修改为0x000002D1 \*/**  
 0x3FFFFFFF, /\* MMSYS0 \*/  
 0x00000000, /\* MMSYS1 \*/  
 },

然后复测，发现还是会死机，抓回来db分析变成了dpidle发生SPI2 bus hang了，同时佐证了之前的分析，因此将SODI,DPIDLE一起修改：

static unsigned int idle\_cond\_mask[NR\_IDLE\_TYPES][NR\_CG\_GRPS] = {  
 [IDLE\_TYPE\_DP] = {  
 0x04000038, /\* MTCMOS, 26:VCODEC,5:ISP,4:MFG,3:DIS \*/  
 0x08040802, /\* INFRA0, 27:dxcc\_sec\_core\_cg\_sta \*/  
 0x00BFB800, /\* INFRA1, 8:icusb\_cg\_sta (removed) \*/  
 0x000000C5, /\* INFRA2 \*/ **/\* 修改为0x000002C5 \*/**  
 0x3FFFFFFF, /\* MMSYS0 \*/  
 0x00000000, /\* MMSYS1 \*/  
 },  
 [IDLE\_TYPE\_SO3] = {  
 0x04000030, /\* MTCMOS, 26:VCODEC,5:ISP,4:MFG \*/  
 0x0A040802, /\* INFRA0, 27:dxcc\_sec\_core\_cg\_sta \*/  
 0x00BFB800, /\* INFRA1, 8:icusb\_cg\_sta (removed) \*/  
 0x000000D1, /\* INFRA2 \*/ **/\* 修改为0x000002D1 \*/**  
 0x3FFFFFFF, /\* MMSYS0 \*/  
 0x00000000, /\* MMSYS1 \*/  
 },  
 [IDLE\_TYPE\_SO] = {  
 0x04000030, /\* MTCMOS, 26:VCODEC,5:ISP,4:MFG \*/  
 0x08040802, /\* INFRA0, 27:dxcc\_sec\_core\_cg\_sta \*/  
 0x00BFB800, /\* INFRA1, 8:icusb\_cg\_sta (removed) \*/  
 0x000000C1, /\* INFRA2 \*/ **/\* 修改为0x000002C1 \*/**  
 0x0F84005F, /\* MMSYS0 \*/  
 0x00000000, /\* MMSYS1 \*/  
 },  
 [IDLE\_TYPE\_RG] = {  
 0, 0, 0, 0, 0, 0},  
};

复测无复现。

**根本原因：**

SPI2之前没有被使用和测试过，没有加入low power场景进入条件判断，导致SPI2工作时进入引起bus hang。

**解决方法：**

将SPI2纳入low power场景进入条件

**结语：**

涉及知识点

* low power flow

mtk\_idle\_cond\_update\_state()在low power判断能否进入前执行。先获取各个clk状态，然后和idle\_cond\_mask求与，如果全部为0表示clk状态允许进入low power场景。

void mtk\_idle\_cond\_update\_state(void) /\* 收集clock状态 \*/  
{  
 int i, j;  
 unsigned int clk[NR\_CG\_GRPS];  
  
 /\* read all cg state (not including secure cg) \*/  
 for (i = 0; i < NR\_CG\_GRPS; i++) {  
 idle\_value[i] = clk[i] = 0;  
  
 /\* check mtcmos, if off set idle\_value and clk to 0 disable \*/  
 if (!(idle\_readl(SPM\_PWR\_STATUS) & idle\_cg\_info[i].subsys\_mask))  
 continue;  
 /\* check clkmux \*/  
 if (check\_clkmux\_pdn(idle\_cg\_info[i].clkmux\_id))  
 continue;  
 idle\_value[i] = clk[i] = idle\_cg\_info[i].bBitflip ? ~idle\_readl(idle\_cg\_info[i].addr) : idle\_readl(idle\_cg\_info[i].addr);  
 }  
  
 /\* update secure cg state \*/  
 update\_secure\_cg\_state(clk);  
  
 /\* update pll state \*/  
 update\_pll\_state();  
  
 /\* update block mask for dp/so/so3 \*/  
 for (i = 0; i < NR\_IDLE\_TYPES; i++) {  
 if (i == IDLE\_TYPE\_RG)  
 continue;  
 idle\_block\_mask[i][NR\_CG\_GRPS] = 0;  
 for (j = 0; j < NR\_CG\_GRPS; j++) {  
 idle\_block\_mask[i][j] = idle\_cond\_mask[i][j] & clk[j];  
 idle\_block\_mask[i][NR\_CG\_GRPS] |= idle\_block\_mask[i][j];  
 }  
}  
  
bool mtk\_idle\_cond\_check(int idle\_type) /\* 检查是否能进入low power场景 \*/  
{  
 bool ret = false;  
  
 /\* check cg state \*/  
 ret = !(idle\_block\_mask[idle\_type][NR\_CG\_GRPS]);  
  
 /\* check pll state \*/  
 ret = (ret && !idle\_pll\_block\_mask[idle\_type]);  
  
 return ret;  
}

**附录**

**lk crash案例分析**

#### lk heap溢出导致crash

**问题背景：**

在lk中增加了两个eMMC固件，固件大小共600KB，固件通过.h文件的形式编译到lk中，手机反复重启。

CPU：MT6765

版本：O1.MP6

**分析过程：**

由于反复重启，因此没有db，直接分析uart log，看到明显的assert：

[72147] panic (caller 0x6002b443): ASSERT at (kernel/thread.c:314): run\_queue\_bitmap != 0

对应代码：

void thread\_resched(void)  
{  
 thread\_t \*oldthread;  
 thread\_t \*newthread;  
  
......  
  
 // should at least find the idle thread  
 ASSERT(run\_queue\_bitmap != 0);

分析其代码逻辑，发现很难推导出为何run\_queue\_bitmap == 0了，继续分析log，看到了其他assert：

[2581] panic (caller 0x6002b3b3): ASSERT at (kernel/thread.c:314): run\_queue\_bitmap != 0  
......  
[SECLIB\_IMG\_VERIFY] malloc memory for heap failed!!  
[1176] panic (caller 0x6002582f): ASSERT at (app/mt\_boot/sec/img\_utils.c:62): 0  
......  
[SECLIB\_IMG\_VERIFY] malloc memory for heap failed!!  
[1176] panic (caller 0x6002582f): ASSERT at (app/mt\_boot/sec/img\_utils.c:62): 0  
......  
[SECLIB\_IMG\_VERIFY] malloc memory for heap failed!!  
[1176] panic (caller 0x6002582f): ASSERT at (app/mt\_boot/sec/img\_utils.c:62): 0

后面的panic可以明显看到是申请内存失败，和添加了600K的fw有关，导致heap size不够，引起分配失败了。

查看heap init代码(lib/heap/heap.c)：

void heap\_init(void)  
{  
 LTRACE\_ENTRY;  
 // set the heap range  
 theheap.base = (void \*)HEAP\_START;  
 theheap.len = HEAP\_LEN;  
......  
}

heap大小有HEAP\_LEN决定：

#define HEAP\_LEN ((size\_t)\_heap\_end - (size\_t)&\_end)

由\_heap\_end和\_end决定，定义在（arch/arm/crt0.s）：

.global \_heap\_end  
\_heap\_end:  
 .int \_end\_of\_ram

在看看\_end和\_end\_of\_ram定义：

[vendor](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/)/[mediatek](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/)/[proprietary](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/proprietary/)/[bootable](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/proprietary/bootable/)/[bootloader](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/proprietary/bootable/bootloader/)/[lk](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/proprietary/bootable/bootloader/lk/)/[arch](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/proprietary/bootable/bootloader/lk/arch/)/[arm](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/proprietary/bootable/bootloader/lk/arch/arm/)/[system-onesegment.ld](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/proprietary/bootable/bootloader/lk/arch/arm/system-onesegment.ld)

4ENTRY(\_start)  
5SECTIONS  
6{  
7 . = %MEMBASE%;  
......  
/\* unintialized data (in same segment as writable data) \*/  
71 . = ALIGN(4);  
72 \_\_bss\_start = .;  
73 .bss : { \*(.bss .bss.\*) }  
74  
75 . = ALIGN(4);   
76 \_end = .;  
77  
78 . = %MEMBASE% + %MEMSIZE%;  
79 \_end\_of\_ram = .;

所以\_end\_of\_ram由MEMBASE + MEMSIZE决定。\_end是放在bss之后。

那么heap size就是就是MEMSIZE - lk大小。由于增加了600K fw，其实就是增大了lk大小。

导致heap size变小，引起malloc fail。我们看看MEMSIZE的定义：

/[vendor](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/)/[mediatek](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/)/[proprietary](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/proprietary/)/[bootable](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/proprietary/bootable/)/[bootloader](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/proprietary/bootable/bootloader/)/[lk](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/proprietary/bootable/bootloader/lk/)/[target](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/proprietary/bootable/bootloader/lk/target/)/$proj/[rules.mk](http://mbjswglx817:8010/alps-mp-o1.mp6/xref/vendor/mediatek/proprietary/bootable/bootloader/lk/target/evb6763_64_ufs_dp/rules.mk)

ifeq ($(MTK\_MLC\_NAND\_SUPPORT), yes)  
 DEFINES += MTK\_MLC\_NAND\_SUPPORT  
 MEMSIZE := 0x00900000 # 9MB  
else ifeq ($(MTK\_TLC\_NAND\_SUPPORT), yes)  
 DEFINES += MTK\_TLC\_NAND\_SUPPORT  
 MEMSIZE := 0x00900000 # 9MB  
else  
 MEMSIZE := 0x00400000 # 4MB  
endif

项目没有定义MTK\_MLC\_NAND\_SUPPORT和MTK\_TLC\_NAND\_SUPPORT，所以MEMSIZE是4MB。

但memory layout给lk预留的是9MB，因此这题的解法是增加MEMSIZE到9MB。

**根本原因：**

添加的600K的fw导致heap size变小，申请内存释放引起panic

**解决方法：**

增加heap size

**结语：**

涉及知识点

* lk heap架构
* lk memory layout

#### lk wdt timeout导致HW reboot

**问题背景：**

9台机器做Monkey测试出现一个hw reboot。

CPU：MT6797

版本：N1.MP9

**分析过程：**

拿到1份db，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SpOfflineDebugSuite分析，发现是HW reboot：

== 异常报告v1.5(仅供参考) ==  
报告解读: MediaTek On-Line> Quick Start> E-Consulter之NE/KE分析报告解读> KE分析报告  
详细描述: 发生HW reboot, 检查CPU是否卡死或硬件故障  
平台 : MT6797  
版本 : alps-mp-n1.mp9/eng build  
异常时间: 0.000000秒, Thu Jan 1 13:36:44 GMT 2015  
  
  
== CPU信息 ==  
其他CPU信息:  
 CPU0: (null), pid: 0, 中断: 关, SVC  
 本地调用栈:  
 ...... 0x0000000046005FA6()  
 == 栈结束 ==

看到0x46005FA6感觉是lk的地址范围，并且看到当前的CPSR为SVC mode。我们检查下kernel的状态，查看SYS\_KERNEL\_LOG：

[45216.468765] (0)[227:bat\_update\_thre]mtk-wdt 10007000.toprgu: shutdown  
[45216.470035] (0)[227:bat\_update\_thre][name:mtk\_wdt&]\*\*\*\*\*\*\*\* MTK WDT driver shutdown!! \*\*\*\*\*\*\*\*  
[45216.470682] (0)[227:bat\_update\_thre][name:mtk\_wdt&]\*\*\*\*\*\*\*\* MTK WDT driver shutdown done \*\*\*\*\*\*\*\*  
[45216.470877] (0)[227:bat\_update\_thre]reboot: Restarting system with command 'DLPT reboot system'  
[45216.470933]-(0)[227:bat\_update\_thre]machine\_restart, arm\_pm\_restart( (null))

发现已restart了，说明kernel已完成重启，那么问题不在kernel。

到这里应该要找lk的symbol文件，分析这个地址0x46005FA6是哪个函数，出了什么问题了。

不过我们还是找找有没有其他线索，先看SYS\_ATF\_LAST，看ATF的状态，在log中间（注意是中间，不是尾巴）找到wdt timeout：

[ATF](0)[11.763968]core 0 is dumped !  
[ATF](0)[11.764004]aee\_wdt\_dump: on cpu0  
[ATF](0)[11.764430](0) pc: lr: sp: pstate: 60000173  
[ATF](0)[11.765582](0) x29: 000000004607a400 x28: 0000000000000000 x27: 0000000000000000  
[ATF](0)[11.766551](0) x26: 0000000000000000 x25: 0000000000000000 x24: 0000000000000000  
[ATF](0)[11.767520](0) x23: 000000004607a400 x22: 0000000000000000 x21: 000000004607a400  
[ATF](0)[11.768489](0) x20: 0000000000000000 x19: 0000000046096840 x18: 000000004600801f  
[ATF](0)[11.769459](0) x17: 000000004607cd3c x16: 0000000046007ff0 x15: 0000000000000000  
[ATF](0)[11.770428](0) x14: 0000000000000000 x13: 0000000000000000 x12: 0000000046096854  
[ATF](0)[11.771397](0) x11: 00000000000001b3 x10: 000000004604b5d4 x09: 000000004604b5c4  
[ATF](0)[11.772367](0) x08: 000000004604b5e8 x07: 0000000051eb851f x06: 00000000000001b4  
[ATF](0)[11.773336](0) x05: 000000000001fbd6 x04: 000000000be4baa1 x03: 0000000010008048  
[ATF](0)[11.774305](0) x02: 000000000be564c1 x01: 000000000001fbd6 x00: 000000000be564c5  
[ATF](0)[11.775276]core 0 is dumped !  
[ATF](0)[11.775694]aee\_wdt\_dump: on cpu0  
[ATF](0)[11.776146](0) pc: lr: sp: pstate: 600001f3  
[ATF](0)[11.777299](0) x29: 000000004607a400 x28: 0000000000000000 x27: 0000000000000000  
[ATF](0)[11.778268](0) x26: 0000000000000000 x25: 0000000000000000 x24: 0000000000000000  
[ATF](0)[11.779237](0) x23: 000000004607a400 x22: 0000000000000000 x21: 000000004607a400  
[ATF](0)[11.780207](0) x20: 0000000000000000 x19: 0000000046096840 x18: 000000004600801f  
[ATF](0)[11.781175](0) x17: 000000004607cd3c x16: 0000000046007ff0 x15: 0000000000000000  
[ATF](0)[11.782145](0) x14: 0000000000000000 x13: 0000000000000000 x12: 0000000046096854  
[ATF](0)[11.783114](0) x11: 00000000000001b3 x10: 000000004604b5d4 x09: 000000004604b5c4  
[ATF](0)[11.784083](0) x08: 000000004604b5e8 x07: 0000000051eb851f x06: 00000000000001b4  
[ATF](0)[11.785052](0) x05: 000000000001fbd6 x04: 000000000be4baa1 x03: 0000000010008048  
[ATF](0)[11.786022](0) x02: 000000000be564c1 x01: 000000000001fbd6 x00: 000000000be564c5

时间是11s，所以问题理清，应该是lk执行太长时间引起。需要lk uart log看哪里执行长了。

我们可以进一步确认问题，检查SYS\_LAST\_LK\_LOG（如果db里有SYS\_PLLK\_LAST\_LOG，则看SYS\_PLLK\_LAST\_LOG。SYS\_PLLK\_LAST\_LOG将被废弃）：

[64128] lk\_wdt\_dump(): watchdog timeout in LK....  
[64129] current\_thread = bootstrap2  
[64129] Dump register from ATF..  
[64130] CPSR: 0x600001f3  
[64130] PC: 0x46027224  
[64130] SP: 0x46096840  
[64131] LR: 0x4600801f  
[64131] mt\_irq\_register\_dump(): do irq register dump  
[64131] GICD\_CTLR: 0x00000012  
[64132] GICD\_IROUTER[0]: 0x00000000, 0x00000000

在N版本，lk可以支持wdt timeout，因为有注册wdt handler，可以输出有些信息，从这里我们再次确认了lk wdt timeout问题。

查看前面的时间戳，发现已经跑了64s，这个不可能，因为lk wdt最多设定10s，怎么回事？检查下前面的log，发现：

[1651] [AUXADC] ch=2 raw=1227 data=539   
[1651] [check\_bat\_protect\_status]: check VBAT=3215 mV with 3450 mV, start charging...   
[1666] [BATTERY:bq24196] charger enable/disable 1 !  
......  
[59332] [AUXADC] ch=1 raw=20943 data=3451   
[59333] [check\_bat\_protect\_status]: check VBAT=3451 mV   
[59333] [check\_bat\_protect\_status]: check VBAT=3451 mV with 3450 mV, stop charging...

原来是低压充电，充到3.45v才开机的，那么在充电过程中，一定有人主动喂狗了，搜索mtk\_wdt\_restart()函数：

void check\_bat\_protect\_status()  
{  
 ......  
 while (bat\_val < BATTERY\_LOWVOL\_THRESOLD) {  
 mtk\_wdt\_restart(); /\* 这里喂狗 \*/  
 ......  
 dprintf(CRITICAL, "[%s]: check VBAT=%d mV with %d mV, start charging... \n", \_\_FUNCTION\_\_, bat\_val, BATTERY\_LOWVOL\_THRESOLD);  
 is\_charging = 1;  
 pchr\_turn\_on\_charging(KAL\_TRUE);  
 thread\_sleep(10000); /\* 这里延时 \*/  
 bat\_val = get\_bat\_sense\_volt(5);  
 dprintf(CRITICAL, "[%s]: check VBAT=%d mV \n", \_\_FUNCTION\_\_, bat\_val);  
 }  
 dprintf(CRITICAL, "[%s]: check VBAT=%d mV with %d mV, stop charging... \n", \_\_FUNCTION\_\_, bat\_val, BATTERY\_LOWVOL\_THRESOLD);  
}

最后一次喂狗后，其实要加上延时时间。

从wdt timeout时间往前推10s，那么就是54s左右，查看当时的log：

[54092] [check\_bat\_protect\_status]: check VBAT=3439 mV with 3450 mV, start charging...   
......  
[59333] [check\_bat\_protect\_status]: check VBAT=3451 mV   
[59333] [check\_bat\_protect\_status]: check VBAT=3451 mV with 3450 mV, stop charging...

也就是说，喂狗后的5s后才退出check\_bat\_protect\_status()函数，那么留给后面流程的时间不多了。

检查下其他分支的代码，发现其他分支在check\_bat\_protect\_status()函数最后再喂了一次狗：

void check\_bat\_protect\_status()  
{  
 ......  
 while (bat\_val < BATTERY\_LOWVOL\_THRESOLD) {  
 mtk\_wdt\_restart(); /\* 这里喂狗 \*/  
 ......  
 dprintf(CRITICAL, "[%s]: check VBAT=%d mV with %d mV, start charging... \n", \_\_FUNCTION\_\_, bat\_val, BATTERY\_LOWVOL\_THRESOLD);  
 is\_charging = 1;  
 pchr\_turn\_on\_charging(KAL\_TRUE);  
 thread\_sleep(10000); /\* 这里延时 \*/  
 bat\_val = get\_bat\_sense\_volt(5);  
 dprintf(CRITICAL, "[%s]: check VBAT=%d mV \n", \_\_FUNCTION\_\_, bat\_val);  
 }  
 mtk\_wdt\_restart(); /\* 这里再次喂狗 \*/  
 dprintf(CRITICAL, "[%s]: check VBAT=%d mV with %d mV, stop charging... \n", \_\_FUNCTION\_\_, bat\_val, BATTERY\_LOWVOL\_THRESOLD);  
}

那么问题就清晰明了，需要在N1.MP9版本也增加喂狗，避免lk wdt timeout。

**根本原因：**

充电流程导致后面的流程没有喂狗引起lk wdt timeout！

**解决方法：**

充电流程check\_bat\_protect\_status()增加喂狗

**结语：**

涉及知识点

* lk wdt架构
* lk memory layout