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：

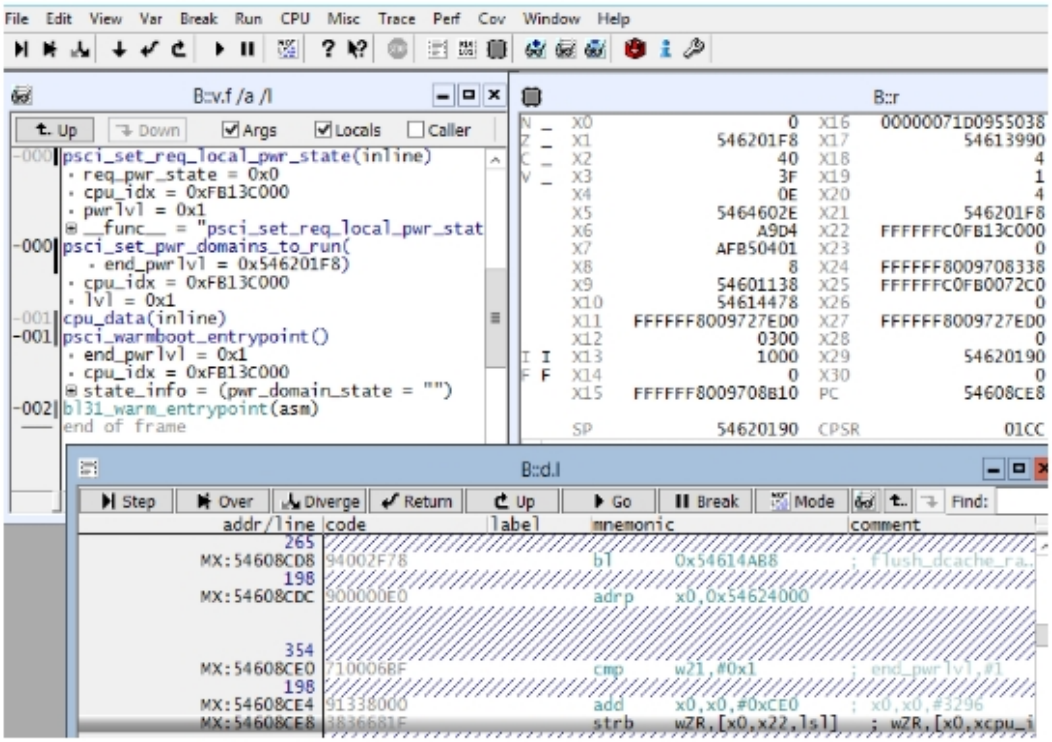
```
Timestamp: 0x00000003da702889, 16.549685

Unhandled Exception in EL3.

x30 = 0x0000000054600f88

x0 = 0x000000000004f460

x1 = 0x0000000000494d4

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

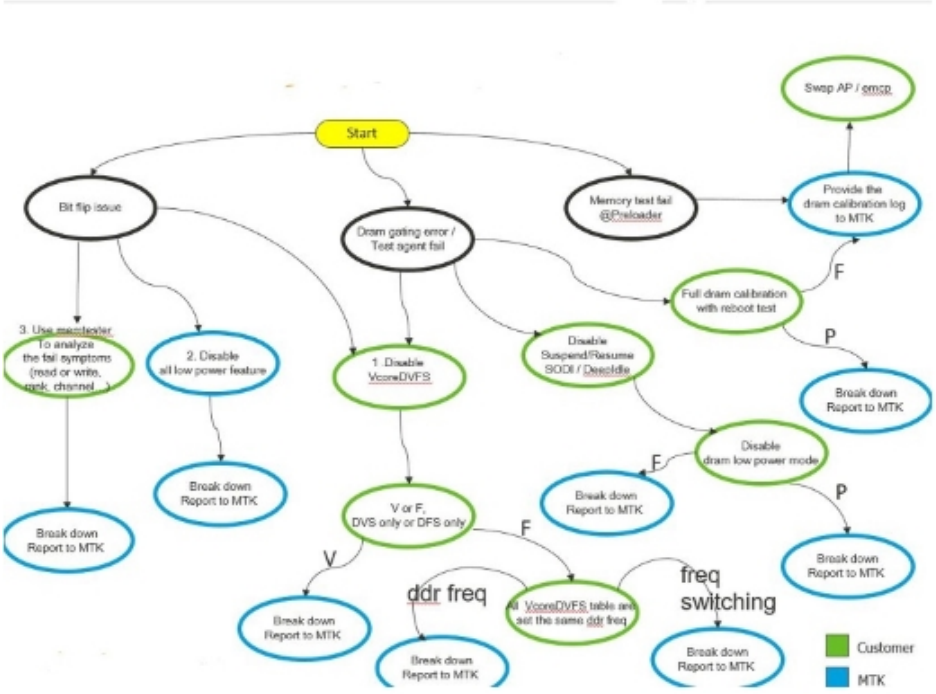
2.SpOfflineDebugSuite 工具分析 db 提示，看是否有 L2 cache error

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 b131.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 = 0xfffffffffc0000000
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
000000000000d03
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 b131.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

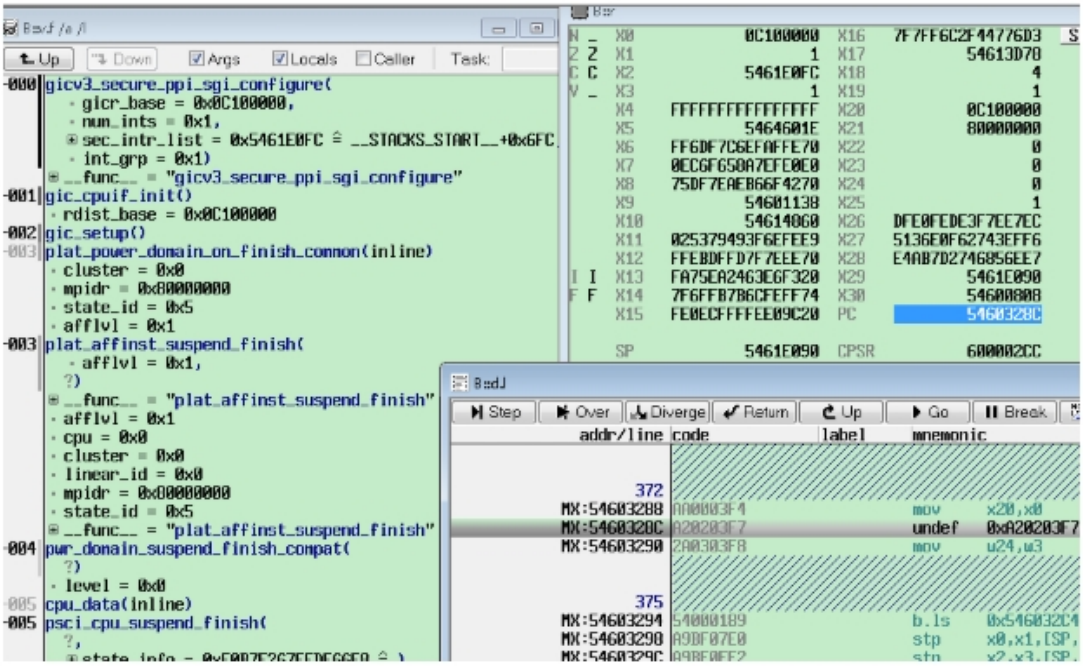
== 日志信息 ==

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)

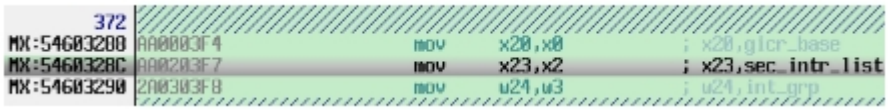
看到是未定义指令，直接启动 trace32 查看，双击 debug.cmm：



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-ml.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 O-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：01.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/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/mediatek/proprietary/bootable/bootloader/lk/target/$proj/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