Qualcomm
TECHNOLOGIES, INC

Qualcomm Crash Debugging Standard Operation Procedure

Debug Manual

80-NM641-1 B

June 6, 2014

Submit technical questions at:
https://support.cdmatech.com/

Confidential and Proprietary – Qualcomm Technologies, Inc.

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Figures

Tables

Revision history

Revision	Date	Description
A	Feb 2014	Initial release
B	June 2014	The secondary release, the main changes: 1. Add some check points in hardware checklist chapter 2. Modify the link of ramparser download place 3. Add one solution of pstore which is supplement of last_kmsg

Note: There is no Rev. I, O, Q, S, X, or Z per Mil. standards.

1 Introduction

1.1 Purpose

This document is aimed to

- Give the whole picture of how many kinds of crashes and how to get dump for analysis on Qualcomm platform.
- Describe what the possible reason is and what the action item could be done to move forward when crash happens.
- Talk about some common debug methods that can be used on real targets.

1.2 Scope

This document is not a solution to crashes, but to describe what to do in the first step and how to analyze from the dump.

The typical issues which are listed in this document cannot cover all kinds of crashes, but the most common issues we met.

This document is applicable to Qualcomm's new Family B platform (8974, 8x26, 8926, 8x10, 8x12, 9x25), also could cover some of Family A platform (8960, 8930, 8064, 9x15).

1.3 Conventions

Function declarations, function names, type declarations, and code samples appear in a different font, e.g., `#include`.

Code variables appear in angle brackets, e.g., `<number>`.

Commands to be entered appear in a different font, e.g., **`copy a:*.* b:`**.

Button and key names appear in bold font, e.g., click **Save** or press **Enter**.

If you are viewing this document using a color monitor, or if you print this document to a color printer, red typeface indicates data types, blue typeface indicates attributes, and green typeface indicates system attributes.

Parameter types are indicated by arrows:

→	Designates an input parameter
←	Designates an output parameter
↔	Designates a parameter used for both input and output

Shading indicates content that has been added or changed in this revision of the document.

1.4 References

Reference documents are listed in Table 1-1. Reference documents that are no longer applicable are deleted from this table; therefore, reference numbers may not be sequential.

Table 1-1 Reference documents and standards

Ref.	Document	
Qualcomm Technologies		
Q1	*Application Note: Software Glossary for Customers*	CL93-V3077-1
Q2	*Software Debug Manual for MSM8960™ Linux Android*	SP80-VN930-5Q
Q3	*Hexagon Programming: Processor Overview*	80-VB419-57
Q4	*QuRT Stability Debug*	80-N2893-1
Q5	*MPSS Debug Guide Hexagon*	80-NC254-17
Q6	*Qualcomm Debug Subsystem (QDSS) – CE Introduction to QDSS*	80-NF515-1
Q7	*QDSS Tracer Non-HLOS API IS*	80-NF515-4
Q8	*QDSS STM Linux API IS*	80-NF515-5

1.5 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://support.cdmatech.com/.

If you do not have access to the CDMATech Support Service website, register for access or send email to support.cdmatech@qti.qualcomm.com.

1.6 Acronyms

For definitions of terms and abbreviations, see [Q1].

2 Crash Overview

Crash issues are very important which will heavily influence user experience. Crash means the UE meets a reset or UI freezes. On Qualcomm platforms, we have some mechanism to get memory dump which contains the whole DDR and internal mem snapshot when reset happens. Then we can use the memory dump for further analysis, such as parsing logs, checking task stacks and global parameter values.

Dump contains most of the information to analyze the crash issue, Chapter 3 will tell us how to get a valid dump and what we should do if we cannot succeed to get a dump finally.

Once we get dumps, firstly, we need to use several Qualcomm tools to parse the memory dump. For most of the issues, we can know the direct reason of crash after this stage.

We divide the crash issues into the following several kinds after this first diagnosis:

2.1 Classify crash

2.1.1 Kernel panic issue

If you enable CONFIG_PANIC_TIMEOUT (the number of seconds to wait after panic before automatically rebooting, it will be disabled if set to 0) and CONFIG_MSM_DLOAD_MODE, the phone will reboot into dload mode if a panic issue happens. For such problem, the detail registers when panic occurs will be printed in the kernel message.

For example,

```
<0>[  279.663195] Kernel panic - not syncing: Fatal exception
<6>[  279.678178] Modules linked in: adsprpc
<6>[  279.681909] CPU: 0    Tainted: G      D W     (3.4.0-svn683 #1)
<6>[  279.687819] PC is at trigger_load_balance+0xa8/0x370
<6>[  279.692763] LR is at trigger_load_balance+0x94/0x370
<6>[  279.697709] pc : [<c01c0058>]    lr : [<c01c0044>]    psr: a0000193
<6>[  279.697712] sp : dbdd9c28  ip : 00000000  fp : dbdd9ce0
<6>[  279.709167] r10: fffff7c6  r9 : 00000040  r8 : 00000000
<6>[  279.714374] r7 : c32d3980  r6 : c0e25dc0  r5 : c32d3980  r4 :
00000000
<6>[  279.720884] r3 : 00000000  r2 : d412b800  r1 : c32d39b8  r0 :
00000000
<6>[  279.727396] Flags: NzCv  IRQs off  FIQs on  Mode SVC_32  ISA ARM
Segment user
<6>[  279.734603] Control: 10c5787d  Table: 34f0806a  DAC: 00000015
```

Also if we set the subsystem restart level to RESET_SOC, then a kernel panic will occur when subsystem dog bite or err_fatal.

```
[  172.765921] SMSM: Modem SMSM state changed to SMSM_RESET.
[  172.771672] Notify: start reset
[  172.771717] Fatal error on the modem.
[  172.771738] modem subsystem failure reason: tdssrchbplmn.c:658:Assertion
(cmd->num_freqs <= TDSL1_MAX_ACQ_CHANNEL_NUM) && (.
[  172.771751] subsys-restart: subsystem_restart_dev(): Restart sequence
requested for modem, restart_level = SYSTEM.
[  172.771770] Kernel panic - not syncing: subsys-restart: Resetting the
SoC - modem crashed.
[  172.771798] [<c010c108>] (unwind_backtrace+0x0/0x11c) from [<c08889bc>]
(panic+0x88/0x1f0)
[  172.771818] [<c08889bc>] (panic+0x88/0x1f0) from [<c015dfe4>]
(subsystem_restart_dev+0x168/0x1ac)
[  172.771837] [<c015dfe4>] (subsystem_restart_dev+0x168/0x1ac) from
[<c01339e8>] (modem_err_fatal_intr_handler+0x3c/0x48)
[  172.771857] [<c01339e8>] (modem_err_fatal_intr_handler+0x3c/0x48) from
[<c01e38a0>] (handle_irq_event_percpu+0x84/0x270)
[  172.771876] [<c01e38a0>] (handle_irq_event_percpu+0x84/0x270) from
[<c01e3ac8>] (handle_irq_event+0x3c/0x5c)
[  172.771894] [<c01e3ac8>] (handle_irq_event+0x3c/0x5c) from [<c01e6550>]
(handle_level_irq+0xe0/0xfc)
[  172.771911] [<c01e6550>] (handle_level_irq+0xe0/0xfc) from [<c01e3280>]
(generic_handle_irq+0x20/0x30)
…..
```

On Family B, subsystem dog bite will also trigger irq to AP, so Linux will have a panic.

The following log is an example.

```
<6>[31905.464319] Watchdog bark! Now = 31905.464315
<6>[31905.464323] Watchdog last pet at 31894.464043
<6>[31905.464328] cpu alive mask from last pet 0-1
<6>[31905.464331] Causing a watchdog bite!cs_rmi4_f12_abs_report: Finger 2:
<4>[31899.699638] status = 0x01
```

2.1.2 Watchdog issue

On Family A Qualcomm platform, we have a hardware watchdog which needs to be pet every ten seconds by default (the value can be changed). If system is unhealthy, the watchdog will bark at first to let software have the last try to control system. Then the bark information and each CPU context will be saved in imem, followed by a system reset and fell into dload mode. When you use the linux-dump-analyzer to analyze the dump, the watchdog bark will be printed together with the CPU context. But when AP watchdog bite on Family A, then it cannot save any information which can be identified as unknown reset.

It has some differences from Family B. On Family B platforms, only secure watchdog bite can reset system. So no matter AP watchdog bark or bite, we still can dump the CPU context.

A typical log which is parsed by linux-dump-parser is as below:

```
------ watchdog state ------
[!!!!] Read aef2c000 from IMEM successfully!
[!!!!] An FIQ occured on the system!
running dump version 1
Core 0 PC: uvc_video_decode_start+578 <c0514e18>
Core 0 LR: uvc_video_decode_start+1d0 <c0514a70>

[<c0514e18>] uvc_video_decode_start+0x578
[<c0515150>] uvc_video_decode_isoc+0x7c
[<c0514750>] uvc_video_complete+0xa8
[<c04580d4>] usb_hcd_giveback_urb+0xb0
[<c046cc80>] ehci_urb_done+0x90
[<c04728dc>] ehci_work+0x49c
[<c0473878>] ehci_irq+0x490
[<c0457604>] usb_hcd_irq+0x30
[<c00d4fe0>] handle_irq_event_percpu+0xb0
```

2.1.3 Userspace reset

In fact, this kind of issue is not a system issue because the chip does not reset but the system server restarts. When such issue happens, customers just see a very quick restart of android UI but the kernel is still in good state. So kernel log is OK but logcat log shows a boot, such as the following event log:

```
11-15 10:27:02.764 I/boot_progress_start(18193): 67055866
11-15 10:27:02.974 I/boot_progress_preload_start(18193): 67056071
11-15 10:27:04.246 I/boot_progress_preload_end(18193): 67057344
```

NOTE: The function **`boot_progress_start ()`** with a big system_server pid > 800 indicates system_server restarts.

2.1.4 Unknown reset

Other reset issues except the above three kinds are all belonging to this type. It also can be divided into different kinds if we look into the technical details. It may be watchdog bite, power sudden lost or failure to enter dload mode. The details will be discussed in Chapter 9.

2.1.5 UI freeze issues

In this scenario, phone will not reset for a long time but the UI is frozen. Customers cannot do any operations and also cannot restart the phone since the key press event cannot be responded. It is a bad experience for end users since it cannot be recovered without pulling out the battery.

2.2 Quickly identify a crash

2.2.1 How to classify a crash quickly

1. Check kernel log to see whether it is panic information.
2. If there is panic information, check whether it is subsystem fatal error or kernel BUG, or kernel page fault.
3. If no panic information, check Trustzone log to see any fatal message. On Family A, AP dog bark will trigger TZ FIQ and print error message, on Family B, AP dog bite will trigger TZ FIQ and TZ will print error message,
4. If no any information, we classify it as unknown reset, please see Chapter 6 about unknown reset
5. We can also refer to the flowchart on Chapter 15.

2.2.2 Getting kernel log

There are different ways to get kernel log when crash happens:

- If uart is enabled and there is uart connect to PC for a console output, you can get kernel log from the uart console. In the latest Family B platform like 8974, it opens in kernel and LK by default. For kernel, you can check it in commandline to grep “cmdline=ttyHSL0”. For LK, you can grep “WITH_DEBUG_UART”.
- From memory dump, please see Chapter 3 to get dumps.
- We can also use apanic method to save panic information into emmc. Please read **solution 00022917** in salesforce.
- There is another kernel feature which is called last_kmsg (it is replaced by pstore on kernel 3.10) that can save the kernel message of last time. It is often used to get the log. Please read **solution 00027971/ Solution 00028866** in salesforce.

2.2.3 Getting Trustzone log

1. The simple method is to open `OCIMEM.bin or DDRCS0` by a hexedit (depends on platform, tz log lies in OCIMEM.bin on 8974 platform but lies in DDRCS0 on 8926), and search string “**Initializing PIL**”, then copy all the text string after it to a text file.
2. We also can get trustzone log by adb, please follow the **solution 00027702**.
3. For more trustzone debugging, please refer Chapter 13.

2.3 The common tools we used for crash analysis

On-target debugging:

- trace32 assisted debugging
- eclipse debugging for the userspace issues
- adb/logcat for live logging

NOTE: Please refer to [Q2] to get more details about this.

Off-target debugging (memory dump debugging):

- Ramparse.py:

  It is a python tool that can extract application system information and imem saved information. The newest version always located at the following link:

  https://www.codeaurora.org/cgit/quic/la/platform/vendor/qcom-opensource/tools/

  The usage of this tool can refer to **solution 00027972**.

- Trace32 scripts

  We can use trace32 scripts when analyzing subsystem dumps, including trustzone, modem and rpm. Please refer to the chapters of each subsystem debugging.

3 Getting Dumps

Dump is the memory snapshot, including the whole system memory and internal memory. Notice that we also need the exactly match debug symbol files together with the dump when you submit a stability case to Qualcomm. The symbols files we often need are vmlinux, RPM_<build_id>.elf, tz.elf, and modem_<buildid>.elf.

3.1 JTAG dump – Saving dump using JTAG

Characteristics:

- Most powerful and basic way to get memory dump
- We can stop the UE at any time and capture memory dump
- Not applicable if the issue is not reproducible with JTAG
- Takes longer than USB dump

Steps:

1. Save complete memory dump (preferred)

   Once UE stops, run **`common\tools\cmm\common\std_savelogs.cmm`**
2. Save memory dump per module
   - Once UE stops, run appropriate script
   - For Apps, run **`common\tools\cmm\apps\std_savelogs_apps.cmm`**
   - For Modem, run **`common\tools\cmm\modem\std_savelogs_modem.cmm`**
   - For LPASS, run **`common\tools\cmm\lpass\std_savelogs_lpass.cmm`**
   - For WCNSS, run **`common\tools\cmm\riva\std_savelogs_riva.cmm`**
   - For RPM, run **`common\tools\cmm\rpm\std_savelogs_rpm.cmm`**

NOTE: Cache flush and saving MMU are done by those scripts

3.2 USB dump – Saving dump through USB using QPST memory debug

Characteristics:

- Most efficient and convenient way to get memory dump
- Transmit memory content through USB cable to PC
- UE needs in Download mode for connecting via USB

- Enable the CONFIG_MSM_DLOAD_MODE in target and prepare environment of USB drivers/QPST before using USB dump.

Steps:

1. Use the Android command to enable Download for kernel panic or hardware reset, because Download mode is controlled by HLOS now.

   ```
   echo 1 > /sys/module/restart/parameters/download_mode
   ```

2. Save USB dump from the device in Download mode
   a. Ensure that the phone is connected to a PC with the appropriate USB cable.
   b. Start the QPST memory debug application on the PC.
   c. Select the appropriate COM port for the phone which is identified as in Download mode.
   d. Click **Get Regions**, which requests the list of memory regions available for upload from the phone.
   e. Select the memory regions to upload to the PC from the list.
   f. Select a location to save the logs, then click **Save To**. The upload begins automatically once the directory is specified.

3.3 SD card dump – Saving dump to SD card

Characteristics:

- Most convenient while field testing, since a USB cable or host PC is not required
- Retrieve ramdump later from the device

Steps:

1. Define FEATURE_BOOT_RAMDUMPS_TO_SD_CARD flag in `cust*.h` file (created from .builds).

   If not enabled by default, this should be enabled. And the `call (boot_ram_dumps_to_sd_card)` should be compiled into the ramdump logic.

2. Define SD_PATH (define SD_PATH "/mmc1/") in `cust*.h` file.

   This also already exists in our build in `.builds/cust*.h`.

3. Place `rdcookie.txt` empty file in FAT partition on removable SD card (at SD_PATH) to enable ram-dump-to-sd card.
4. When SBL3 enters Download mode with above setup, SBL3 will find `rdcookie.txt` at SD_PATH and perform ram-dump-to-sd card, and enter Download mode after ramdump completion.

3.4 Subsystem restart dump – Saving dump during subsystem restart

Characteristics:

- While performing subsystem restart, we can collect dump from subsystem.

- Subsystem dump is saved onto file system.
- Subsystem restart and dump feature should be enabled to use this.

Steps:

1. Enable subsystem restart first. Normally, it will be disabled by default.

   ```
   echo 3 > /sys/module/subsystem_restart/parameters/restart_level
   ```

2. Install Ramdump App to `/system/bin` and change permission.

   ```
   adb remount
   adb push subsystem_ramdump /system/bin
   adb shell chmod 777 /system/bin/subsystem_ramdump
   adb shell sync
   ```

3. Change permission by adb command:

   ```
   chmod 664 /dev/ramdump_*
   ```

4. Enable Ramdump mode in the adb shell while restarting subsystem.
   - echo 1 > /sys/module/subsystem_restart/parameters/enable_ramdumps – enable ramdump when subsystem panic occurs
   - echo 1 > /sys/module/subsystem_restart/parameters/reset_detection – enable ramdump when subsystem reset is detected.

NOTE: This is needed only on MSM8660 platform. For MSM8974 and subsequent chips, enable_ramdump is enabled by default.

5. Execute subsystem_ramdump by running the **`deamon ./system/bin/subsystem_ramdump`** and the rampdump location is `/data/ramdump or /sdcard/ramdump` depending on previous configuration.

NOTE: **Solution 00023342** addresses this procedure.

3.5 Trigger a crash to get dump by software

If modem port is seen from PC, we can trigger each subsystem dump by the following commands from QXDM:

Table 3-1 QXDM command to trigger each subsystem crash

Test case	MPSS	ADSP (LPASS/sensor)	Pronto (WCNSS)	Venus (MMSS)	GSS
Software Error Fatal	send_data 75 37 03 00	send_data 75 37 03 48	send_data 75 37 03 32	send_data 75 37 03 64	send_data 75 37 03 00

Software Exception (Div by 0)	send_data 75 37 03 00 03	send_data 75 37 03 48 03	send_data 75 37 03 32 03	send_data 75 37 03 64 03	send_data 75 37 03 00 03
Software Exception (NULL ptr)	send_data 75 37 03 00 02	send_data 75 37 03 48 02	send_data 75 37 03 32 02	send_data 75 37 03 64 02	send_data 75 37 03 00 02
WD Bite	send_data 75 37 03 00 01	send_data 75 37 03 48 01	send_data 75 37 03 32 01	send_data 75 37 03 64 01	send_data 75 37 03 00 01

If AP side adb can work, we can get the dump by triggering a panic from adb shell:

```
echo ‘c’ > /proc/sysrq-trigger
```

3.6 Trigger a crash to get dump by hardware

Dropping down the PS_HOLD can simulate the sudden hardware reset to get the dump. For Family B, we have some timing requirement, so it is better to use a simple RC circuit to pull down the PS_HOLD.

3.7 Why cannot get USB dump sometimes

Customers often use USB dump because JTAG dump sometimes not so convenient when testing in large scale. But sometimes customers cannot get a valid dump, even worse, cannot enter dload mode.

We can analyze this issue from the flow of UE entering into dload mode. There are some requirements that may be not satisfied which finally cause failure to get a dump.

1. Check multiple software debug macros whether open or not. For example, CONFIG_MSM_DLOAD_MODE.
2. DDR corruption causes the magic number to lose. Then UE falls into normal boot flow when checking the magic number. For such issue, check Chapter 9.2.4 (memory not stable).
3. UE also meets abnormal when reset, it even fails very early before running till the code which checks the dload magic number... For such issue, we may add log in SBL or use trace32 for further debugging. For how to enable uart log in SBL, please refer to **solution 00027701** and **Solution 00027943**.
4. PMIC not setting correctly, it may be a pmic hardware reset and then DDR content is corrupted. It needs pmic engineers to check the related registers setting.

We also have some other ways to assist in debugging except dump in this scenario,

1. Disable watchdog (WDOG_EN pin in hardware) and try to attach trace32 when the phone freezes. Then we use trace32 to check the whole system state on target.
2. Take advantage of some logging mechanism background before crash happens, for example, we can use kernel last_kmsg (it is replaced by pstore on kernel 3.10) method, then it will record the kernel message last time before reset. Please refer to Chapter 2.1.2 .

3. Take advantage of ETB(Family B not useful)/RTB/QDSS (Please refer to Chapter A) debug method to assist in analyzing
4. Involve hardware to measure the power-on-sequence waveforms to find abnormal. Please refer to Chapter 14 (hardware checklist).

4 Kernel Panic Analysis

As described in Chapter 1, when panic happens, the kernel log has already described the register values which describe the exact scenarios. If the information is not complete, we need to check the log buffer to make sure it is uncachable (Please refer to **Solution 00027973**). last_kmsg (pstore on 3.10 kernel) is also a good method to save the logs (**Solution 00027971/Solution 00028866**).

4.1 Common steps

1. Get the whole dump and then use `ramparser.py` to get the system detail logs
2. Check the `dmesg_tz.txt` and click **launch_t32.bat** to launch dumps in trace32 simulator.

4.2 Typical issues

4.2.1 Prefetch abort with the PC at 0x0 or an address that's not in the 0xC0000000-0xCFFFFFFF range

- DDR memory corruption

  Do a **data.list** on the current PC and LR values in T32. For a more advanced check, you can compare the instructions against how they appear in the original vmlinux file, using an eabi-nm tool to generate a readable `objdump.txt` file. If you see discrepancies, it may be DDR issues or notify Qualcomm.

- Cache corruption

  Do a **data.list** on the current LR value, and look at the instruction immediately before the LR value. If the instruction references a register value that can change, the prefetch abort could actually be caused by a software bug. Check the function the LR. If you can identify the tech team who owns this function, file a case to the corresponding tech team. If you've reached this point and are out of ideas, submit a case to Qualcomm CE team.

4.2.2 Unhandled page fault or NULL pointer access

Example:

```
<1>[ 3084.469254] Unable to handle kernel NULL pointer dereference at
virtual address 00000074
<1>[ 3084.477329] pgd = ebf54000
<1>[ 3084.479998] [00000074] *pgd=00000000
<0>[ 3084.483564] Internal error: Oops: 5 [#1] PREEMPT SMP ARM
<6>[ 3084.488855] Modules linked in: wlan(O) adsprpc
```

```
<6>[ 3084.493293] CPU: 1    Tainted: G        W  O  (3.4.0-svn466 #1)
<6>[ 3084.499213] PC is at __mutex_lock_slowpath+0x84/0x360
```

For such issue, we can restore the registers which are printed on kernel message in trace32 simulator. And then we check the exception stack to find what the root cause is. The possible reasons are:

- If the address looks valid, it may be river code issue, which needs to check codes.

  Race condition; (see Chapter A for enabling multiple kernel debug features to check)
- If the address looks invalid, check the memory mapping to make sure it is mapped

  It may be memory bitflip issue; (see Chapter 9.2.4)

  It may be core not stable issue (see Chapter 14)

4.2.3 Trigger the kernel BUG

Example:

```
<2>[ 166.667126] kernel BUG at
/apps_proc/kernel/drivers/platform/msm/ipa/a2_service.c:454!
<0>[ 166.678147] Internal error: Oops - BUG: 0 [#2] PREEMPT ARM
<4>[ 166.683618] Modules linked in: sd8xxx(O) mlan(O)
<4>[ 166.688220] CPU: 0 Tainted: G D W O (3.4.0+ #1)
<4>[ 166.693607] PC is at kickoff_ul_wakeup_func+0xac/0xcc
<4>[ 166.698637] LR is at kickoff_ul_wakeup_func+0xac/0xcc
```

The kernel BUG is set to catch the unexpected scenario, for example, an invalid pointer, an extremely timeout value etc. So we still need to check the dump as unhandled page fault scenario. If it is not expected memory problem, we still need refer to Chapter 9.2.4 to see whether memory unstable. If it shows a register error, then maybe we need refer to Chapter 14 for hardware checklist.

4.2.4 Cache parity error

Both the L1 cache and L2 cache have the capability to detect up to 1 bit of data or a tag-parity error. Both I-Cache and D-Cache are susceptible to data and tag-parity errors if the voltage supply to the cache is instable. The CESR stores fields for I-Cache data-parity errors, I-Cache tag-parity errors, D-Cache data-parity errors, and D-Cache tag-parity errors. Voltage instability or insufficient voltage on the rails powering the respective caches are the only known possible causes for data and tag-parity errors.

It is not 100% fatal. So we normally need to check PVS value and adjust core value by increasing 25mV or 50mV.

4.2.5 L2 master port decode error

This kind of error is the result of an improper register or memory access. That is, the software running on the apps processor could be accessing a register or memory area that does not exist, is not clocked, or is locked down by an XPU.

For Family A, it is possible to use the ‘EBI ERP’ driver to read out the register/memory address.

For Family B, usually, master port decode errors which trigger XPU violations of AHB timeouts can be debugged. Please refer to Chapter 13.

4.2.6 Subsystem error

On Family A chipsets, if a subsystem err_fatal and subsystem restart level is set to RESET_SOC, then a kernel panic will also be trigged. For such issue, we need to check each subsystem debugging (for example, for modem, please refer to Chapter 11).

On Family B, subsystem dog bite will also trigger an AP IRQ.

4.2.7 Out of memory issues

Example:

```
<4>[54062.495330] android.browser invoked oom-killer: gfp_mask=0xc0d0,
order=2, oom_adj=0, oom_score_adj=0
<4>[54062.502349] Mem-info:
<4>[54062.502471] Normal per-cpu:
<4>[54062.502655] CPU 0: hi: 90, btch: 15 usd: 0
<4>[54062.502777] CPU 1: hi: 90, btch: 15 usd: 0
<4>[54062.502868] HighMem per-cpu:
<4>[54062.503082] CPU 0: hi: 186, btch: 31 usd: 0
<4>[54062.503173] CPU 1: hi: 186, btch: 31 usd: 60
<4>[54062.503387] active_anon:87251 inactive_anon:624 isolated_anon:1
<4>[54062.503387] active_file:11946 inactive_file:12064 isolated_file:0
<4>[54062.503387] unevictable:3936 dirty:3 writeback:0 unstable:0
<4>[54062.503387] free:4209 slab_reclaimable:2750 slab_unreclaimable:5916
```

OOM is a mechanism when Linux system meets memory shortage. After this error is printed, the kernel log will also print the memory usage in system. Sometimes it happens due to memory leak and sometimes it is just because of memory allocation too much. For the memory leakage, we may refer to `kernel/Documentation/kmemleak.txt` for further debugging. For the whole system memory usage tuning, we also can adjust the low memory killer parameters to make it work efficiently before OOM is triggered.

4.2.8 Dog bark

On Family B, non-secure dog bark will trigger an irq and kernel will print and panic. Please refer to Chapter 6.

5 AP Watchdog Bark Analysis on Family A

5.1 Common steps

For such issue, sometimes it is caused by panic, then we can refer to Chapter 4 (panic analysis). But it is also more possible caused by something abnormal that blocks the dog petting thread. For example, logging too much sometimes causes such issues.

In most scenarios, the CPU context is valid in imem. Then we can click on **lauch_t32.bat** to restore the crash point to check the kernel thread. And RTB information is also very helpful in such scenario. If the CPU context is not meaningful, we may refer to Chapter 9.2.4 and Chapter 14 to check memory stability and hardware.

5.2 Typical issues

5.2.1 Excessive logging

Example:

```
[!!!!] Read b5f05000 from IMEM successfully!
[!!!!] An FIQ occured on the system!
running dump version 1
Core 0 PC: go_wfi+4 <c0023c10>
Core 0 LR: msm_pm_idle_enter+70 <c0054e88>
[<c0023c10>] go_wfi+0x4
[<3>] (No symbol for address 3)+0x0

Core 1 PC: msm_hsl_console_putchar+118 <c034d22c>
Core 1 LR: uncached_logk_pc+20 <c0077b98>
[<c034d22c>] msm_hsl_console_putchar+0x118
[<c0345898>] uart_console_write+0x40
[<c034dd28>] msm_hsl_console_write+0xc4
[<c007c238>] __call_console_drivers+0xac
[<c007ca68>] call_console_drivers+0xbc
[<c007cf20>] console_unlock+0x6c
[<c02ccd50>] do_fb_ioctl+0x31c
[<c013e6f0>] do_vfs_ioctl+0x240
[<c013e75c>] sys_ioctl+0x34
[<c000df00>] ret_fast_syscall+0x0
```

If there is a stack symptom with a lot of log output (messages are printed out every 100th or 1000th of a second), it maybe cause watchdog bark.

5.2.2 Obvious driver-specific function in current CPU stack

For example,

```
------ watchdog state ------
[!!!!] Read 82a95000 from IMEM successfully!
[!!!!] An FIQ occured on the system!
running dump version 1
Core 0 PC: _raw_spin_lock_irqsave+5c <c07465e8>
Core 0 LR: pagevec_lru_move_fn+64 <c00fa738>

[<c07465e8>] _raw_spin_lock_irqsave+0x5c
[<c00fa738>] pagevec_lru_move_fn+0x64
[<c00fa9a0>] __lru_cache_add+0x9c
[<c010e094>] do_wp_page+0x674

Core 1 PC: _raw_spin_lock_irqsave+5c <c07465e8>
Core 1 LR: pagevec_lru_move_fn+64 <c00fa738>
```

Open up T32 and examine the context on each core. You can load each core's context with the core0_regs.cmm, core1_regs.cmm, core2_regs.cmm, and core3_regs.cmm scripts, respectively.

If we can see the watchdog bark having an obvious driver-specific function just before the **__irq_svc…wdog_bark_handler** part of the stack, please grep for the function or message and look for places where execution could possibly spin for a long period or even worse, with the irq disabled. If you can identify the code responsible, file a case to the corresponding tech team.

If the stack traces on one or more cores end at or near a spinlock or mutex call, it could be a deadlock/livelock scenario, which is caused either by incorrectly written code or by a corrupted lock.

5.2.3 Nothing obvious in the stack trace

1. Check the kernel workqueue farther down in the `dmesg_TZ` file.

   If you see a lot of **`pending workqueue`** functions and one or two **`Workqueue BUSY`** functions, one of these might be blocking the workqueue. Grep for the function or message, and look for places where execution could possibly spin for a long period.

2. If workqueue also seems normal, maybe system is too busy

   If you find no function blocking the workqueue, try to use function trace to see whether there is a lot of hard irqs or soft irqs. You can try to increase the watchdog bark time sometimes.

3. Check for XPU violations

   XPU violations sometimes also cause issues, please refer to Chapter 13.

6 AP Non-Secure Watchdog Bark Analysis on Family B

On Family B only secure watchdog bite can reset the SOC. When AP non-secure watchdog bark happens, it will trigger a kernel panic such as following:

```
<6>[22299.499652] Watchdog bark! Now = 22299.499638
<6>[22299.503018] Watchdog last pet at 22288.499307
<6>[22299.507351] cpu alive mask from last pet 0-1
<6>[22299.511596] Causing a watchdog bite!
```

Then we can analysis it which is similar as Chapter 5.

7 AP Non-Secure Watchdog Bite Analysis on Family B

Unlike the non-secure watchdog resetting chip on Family A, AP non-secure watchdog bite will be caught by trustzone on Family B. In this scenario, trustzone handler will dump the CPU context and then reset. If the CPU context seems correct, follow the analysis as Chapter 5. If it looks strange, may refer to Chapter 9. The trustzone diag buffer will be like below:

```
Diag buffer
=========================
….
(8)
(8)
(8)
(8)
Fatal Error: NON_SECURE_WDT
```

8 Userspace Reset

8.1 Common steps

In fact, this is a system server restart and kernel is still healthy. It is also called framework reboot. For such issues, adb should be continuously in connection and work well. So we need to check logcat logs at first. It may be purely java issues and also possible kernel issues.

Some common commands which can be used on shell:

- Merge logcat and kernel log in one file:
  ```
  logcat -f /dev/kmsg &
  cat /proc/kmsg > /data/1.txt &
  ```

- **`bugreport:`** get the system logs and thread status
- **`dumpstate:`** get the system state
- **`procrank:`** print each thread memory usage
- **`top:`** print each thread cpu usage

8.2 Typical issues

The userspace reset is mainly divided into the following several kinds:

8.2.1 Android watchdog fires

System_server's watchdog thread regularly posts monitor messages to check for system responsiveness. Android.server.ServerThread thread queues and processes watchdog thread's messages. When it is timeout in **petting** watchdog, the watchdog thread kills system_server.

Android's watchdog has the following monitors, if one of the following services has problems, then maybe cause android watchdog timeout.

- InputManagerService
- NetworkManagementService
- MountService
- PowerManagerService
- WindowManagerService
- ActivityManagerService

For such issues, it maybe have ANR or some other userspace errors before. If ANR has already happened, a `traces.txt` file will be saved into `/data/anr` folder. It contains most of the stack of dalvik threads. We need to check this file and tombstone to find out what halts the thread. The possible reasons are as below:

- CPU usage is too high. This will cause CPU usage information to be printed in the logcat. Then we need to check whether there is something wrong with that thread
- The dalvik main thread is halted on some native function, then we need to check the driver native method further

Check the `traces.txt` and tombstones to see where it halts. If it halts on some native function, we also need to check driver; if the reason is CPU usage, we also need to check the kernel thread which occupies most of the CPU.

For a debug method, we can also disable Android watchdog (only to debug some watchdog issues), the command is as below:

```
adb shell setprop debug.watchdog.disabled 1
```

8.2.2 Fatal exception in a system process

For this issue, the **`adb logcat –v threadtime`** will print the exact java lines error. We can debug java codes to see what happens.

8.2.3 Native crash in a system process

For the native crash issues, the tombstones file will be generated on phone's filesystem in `/data/tombstones` folder when issues happen.

The following is common ways to debug such issues:

- Check logcat and tombstone, use the following command to recover the stack.
  ```
  $ANDROID_BUILD_TOP/development/scripts/stack
  –symbols-dir=out/target/product/<board>/symbols <tombstone_file>
  ```
- Enable multiple dalvik debug and bionic debug property, such as dalvik.debug.oom, libc.debug.malloc
- Try to find similar problems in android opensource forum or google maillist since it is owned by google.

8.2.4 Surfaceflinger native crash

The debug method is similar as native crash above in a system process. And if the tombstone points to a native function in kernel, we maybe need to check display related drivers.

8.2.5 Excessive JNI references

JNI is Java Native Interface. JNI Global references are created to prevent garbage collection of Java objects. CheckJNI can be enabled to prevent JNI problems in **engg/non-user** builds.

The JNI Global Reference Limit in code is 2001 by default. If exceeding this number, the Dvm Abort (dalvik process gets killed) will happens.

For a debug method, we can add HPROF heap dump when this happens. And then we can inspect JNI reference table in logcat and heap dumps in the form of `.hprof` files in `/data`. It is mainly java application issue.

You can refer to http://android-developers.blogspot.com/2011/03/memory-analysis-for-android.html for more details.

9 Unknown Reset

If we meet reset with no obvious clue and then does not fall into the previous kinds, we called it unknown reset. For such issues, it is mostly systematic and difficult to solve. Sometimes we need to try a lot of methods, get dumps for each method and then find out the real culprit.

9.1 Common steps

1. Provide the whole memory dump together with `RPM/TZ/AP` symbol file.
2. Refer Chapter 12 and Chapter 13 to get RPM/TZ logs
3. Check reboot reason via PMIC_PON.BIN and RST_STAT.BIN if available
4. Check whether it is related with suspend/sleep, if it is, we may try with modifying the sleep level to locate the issue
5. Check the hardware checklist (Chapter 14) if the unknown reset is hardly to reproduce

9.2 Typical issues

9.2.1 Non-secure AP watchdog bite on Family A

For Family A, RPM watchdog and AP watchdog both have the capability to reset system, so we may disable one to quickly identify which core causes the issue. If disable apps watchdog, then the phone will freeze instead of reset, it is surely non-secure apps watchdog bite.

1. For the Family A non-secure watchdog bite, firstly, we still need to check the debugging method in Chapter 5, especially the workqueue information.
2. Check global parameters and RTB information (see Chapter A) to know what the final action cores are doing.
   a. Notice whether the irq/fiq is disabled in last action
   b. Check whether it is sleep related, if yes, please refer to Chapter 9.2.5.
3. If we have ETB information (see Chapter A), then load ETB information to trace32 simulator to see the last instructions cores are doing.
4. If still no clue and problems easily to be reproduced, we can disable apps watchdog and then try to attach trace32 on target for further debugging.
5. If still no clues, please involve hardware guys to see whether voltage is correct according to Chapter 14.

9.2.2 Non-secure RPM watchdog bite in Family A

It is not common since RPM is seldom changed by customers. Anyway, if it happens, it is very possible related with memory and hardware. Please check Chapter 12, Chapter 9.2.4 and Chapter 14.

9.2.3 Secure watchdog bite in Family B

For Family,B only secure watchdog can reset the system, RPM log and TZ log are very important to such issues. Furthermore, for some difficult issues, we also need to enable QDSS software event and hardware event to check more.

1. Check the RPM global parameters to see which subsystem is alive.
2. If RPM log has err_fatal information, then refer to Chapter 12
3. If the trustzone log shows XPU error or bus timeout error, please refer to Chapter 13.
4. If the application processor is alive, check the kernel message, RTB information, ETB information to know what actions each cores are doing.
5. If the RPM global parameters show application processor is still in process of shutdown or bringup, then refer to Chapter 9.2.5. In this case, we need review the hardware checklist (Chapter 13)
6. If we have no result till this step, it may be bus hung, we can enable QDSS to see whether it can get some information. Please refer to Chapter A.

9.2.4 Memory stability checking

The memory should be suspected if we meet some strange issues, for example, the global parameter is not expected; the crash happens after customers use another MCP, always related with some specific board., etc.

Basically, the memory stable issues may be in the following aspects:

- Memory timing issues when operate at ultra low or high frequencies
- Memory voltage out of spec in some scenario
- Memory voltage and frequency out of sync when phones dynamically change frequency and voltage

For such issues, we have some simple methods to identify the issues:

- Disable DCVS to have a test
- Prevent sleep or power collapse to have a test
- We can also have some tools to test the memory stability
  - Use trace32 for simple memory test: *d.test <range>*
  - Use Qualcomm scripts to test the whole memory
  - Use QblizzardEx package
  - Use Alarmtest package to test the sleep/wake sequence of DDR

The above tools can give an overall coverage and basic test of DDR, but sometimes we still need to co-work with memory vendor to test the memory with special instruments.

9.2.5 Sleep/suspend related issues

It is typical for such issues happens during the stage of sleep/suspend since there is a lot of system behaviors happening during this timeslot, such as voltage adjustment, CPU/cache re-initialization, frequency changing etc.

For such issues, we may check the issue with the following steps:

1. Check application processor kernel message and RTB information to see whether it is already programmed SPM registers. If not, then application cores should still online and problem maybe lie in the suspend thread, we need to focus on software side. If it has already programmed SPM registers, we may need to check what state AP cores stay.
2. If application side already show it is offline, we need to check the trustzone kernel counters to see whether it halts on trustzone.

   PCEntry is incremented when SCM call is made into TZ to enter power collapse.

   PCExit is incremented when the WFI in the above SCM returns immediately (i.e. we do not enter power collapse)

   WarmEnty is incremented when TZ begins executing after power collapse.

   WarmExit is incremented when execution is about to be handed off to HLOS after executing the warm boot code.

   For Krait 0, you can assume that TZ thinks the core is online if the `PCEntry = PCExit + WarmExit`. Otherwise, TZ thinks it's power collapsed.

   For all other Krait cores, you can assume that TZ thinks the core is online if the `PCEntry + 1 = PCExit + WarmExit`. Otherwise, TZ thinks it's power collapsed.

   - The WarmEntry and WarmExit counters are used to determine when execution enters and leaves the TZ while resuming from power collapse (a "warm boot").
   - If the WarmEntry and WarmExit counters for any of the cores differ, it means halt on trustzone side, please refer to Chapter 9, Trustzone debugging.
3. Check the global symbol ee status in RPM dump to see what subsystem wakes or sleeps. If trustzone shows core is power-collapsed but RPM already got the SPM acknowledge of wakeup, then the issue maybe lie in L2 cache and power rail. Please contact Qualcomm and refer to Chapter 13.

Notice that for issues related with suspend/sleep, they are mostly related with frequency/voltage scaling, unexpected timing issues at low voltage (especially on board with bad PDN). So we need to involve hardware experts to hardware checklist (Chapter 14) as soon as possible.

9.2.6 Bus hung

For bus hung issues, it is hard to debug, so please contact Qualcomm as soon as possible.

Mainly we use the following methods to identify this kind of issues:

- Use oscilloscope to measure the waveforms of vdd_cx, if it is bus hung, it keeps at the same level for the whole watchdog bite time no matter how long we change the watchdog bite time.
- Disable watchdog hardware pin, then phone should freeze. At this time, we can try to see whether we lucky enough to attach trace32, we can access multiple registers to guess which bus hung if success.

- For Family B chips, most of the bus timeout issues will be caught by trustzone. Please refer to Chapter 13. And we can also enable QDSS for further debugging, please see Chapter A.

9.2.7 Pmic and thermal issues

Sometimes the pmic and thermal will also cause system reset. For such issues, let's follow the following steps:

1. Check the GCC_RESET_STATUS register value for any potential sign of a thermally-induced reset:

   Open up the `RST_STAT.BIN` file in the RAM dump folder with a hex editor, and look at the first byte in the file.

   - If the value of this register is 0x00, it means the device most likely reset from something other than a non-secure or secure watchdog bite. This could include kernel panics, subsystem resets, or a reset triggered by the PMIC.
   - If the value of this register is 0x02, it means the device reset due to a non-secure or a secure watchdog bite. Check all possible causes for a watchdog bark or bite if you've reached this point.
   - If the value of this register is 0x08, it means the device overheated and is automatically reset after exceeding a temperature threshold. Notify the launch team if this is the case.

2. Check the PMIC reset registers for signs of a PMIC or user-induced reset:

   Open up the `PMIC_PON.BIN` file in the RAM dump folder with a hex editor.

   - If the **PON_WARM_RESET_REASON_1** register or the 3rd byte of the `PMIC_PON.BIN` file is 0x80, it means the target is intentionally reset by someone holding down the power button for 15 seconds. This is also designed to send the device to download mode as a result.
   - Also check the **PON_REASON and POFF_REASON** to see whether USB_CHG, SMPL, UVLO etc, comparing with the PMIC software manual.

3. If we still have no results, then need hardware team to grasp the waveforms to check the PS_HOLD and PON signals. Then we can identify whether it is pmic caused reset or MSM reset first.

10 Freeze Issues

10.1 Common steps

In this scenario, phone will not reset for a long time but the UI is already frozen. At first, we need to check whether watchdog is enabled or not. Both kernel and android have their own watchdogs, and it is expected to reset when something abnormal happens.

10.2 Typical issues

If hardware watchdog has already been enabled but UI is still frozen, then it may be divided into several kinds of issues:

- Try to check the UI freeze whether it will get better (just very slow response) or never been recovered. If it can be recovered, then it is not freeze issue, but need to check kernel behavior that whether some issues lie in scheduling, such as real time tasks or too much interrupt.
  - Check the logcat to see whether some **ANR** happens, please refer to Chapter 5.2.1 for analysis
  - Check the logcat to see whether CPU usage is too high, and use **top** and function **`trace`** for further debugging
- Check whether adb daemon can work or not, if adb port can be recognized and adb command can be parsed, then it is possible a framework issue or some kernel thread deadlocked.
  - Use adb to get the kernel log, logcat log and bugreport
  - Analyze the bugreport to see whether some service is deadlock, especially Surfaceflinger and other display related threads
  - Check the logcat to see whether CPU usage is too high
  - Check the logcat to see whether it is due to memory killer always working and service is restarting
  - If still no result, you can trigger a panic **`echo c > /proc/sysrq-trigger`** via adb shell to get a dump for further analyzing.
- If some USB device can be recognized when plug in PC but adb cannot work, it means kernel is not working normally but watchdog is still pet regularly.
  - Try to attach the trace32 to check what cores are doing
  - If fails to attach trace32 on krait, attach it on RPM side and get a dump.
  - If kernel log is still working via uart, add more logs to reproduce and check
  - Use QDSS to set proper event for debugging, please contact Qualcomm.
- If no device can be found when plug in PC, it is a scenario that watchdog cannot bite but kernel cannot response interrupt and still can pet dog. It seldom occurs and we only can rely

on trace32 or QDSS for further debugging if we really meet this issue. But in most cases, it may be pmic issue that fails to reset the phone, we need to refer to Chapter 14 for hardware checklist.

11 Modem Crash Debug

For Family A and Family B, the modem processor is QDSP6, the RTOS is BLAST/QURT. For QDSP6 processor overview, refer to [Q3].

For general modem crash, you will see the modem crash info on the kernel log,

```
<3>[26704.104390] SMSM: Modem SMSM state changed to SMSM_RESET.
<3>[26704.104431] Notify: start reset
<3>[26704.468632] Watchdog bite received from modem software!
<3>[26704.468682] modem subsystem failure reason: ipcmem.c:728:No free
memory for requested size [188]. Last pool checked: pool n.
<6>[26704.468714] subsys-restart: subsystem_restart_dev(): Restart sequence
requested for modem, restart_level = RELATED.
<6>[26704.471006] subsys-restart: subsystem_shutdown(): [e15c8000]:
Shutting down modem
```

For modem crash debug, we have two detailed documents. Please refer to [Q4] and [Q5]:

12 RPM Debug

To debug RPM side crash issue, we usually need to load and analyze the RPM dump, parse out RPM log and NPA log, restore the call stack to locate the code line where the crash happens.

Most of the time, RMP side crash is caused by err_fatal which is trapped by RPM codes. And we also find DDR freqency/voltage setting/PVS feature could cause stability problem sometimes. These scenarios will be addressed here.

12.1 RPM dump load/analyze

To load Family B RPM dumps into a T32 simulator:

1. Open a T32 simulator and do sys.up.
2. Use load.cmm or manually load as belows:

   ```
   d.load.binary CODERAM.BIN ~~0xfc100000~~   0x100000
   d.load.binary DATARAM.BIN  ~~0xfc190000~~   0x190000
   d.load.binary MSGRAM.BIN   0xfc428000
   d.load.elf  \\elf_file_path\RPM.elf /nocode
   ```

   Once these steps are done(with the matching elf and dump), OEM could start to parse/analyze on RPM logs(RPM log/NPA log) and debugging variables
3. Extract an RPM external log

   ```
   do rpm_proc\core\power\ulog\scripts\ULogDump.cmm <path to your
   directory>
   ```
4. Extract an NPA log

   ```
   do rpm_proc\core\power\npa\scripts\NPADump.cmm <path to your directory>
   ```
5. Translate rpm log into readable format

   ```
   python rpm_proc\core\power\rpm\debug\scripts\rpm_log.py -f “RPM External
   Log.ulog” -n “NPA Log.ulog” > rpm_parsed.txt
   ```

   Additional switches are –r, which prints raw (sclk value in hex format) timestamps.

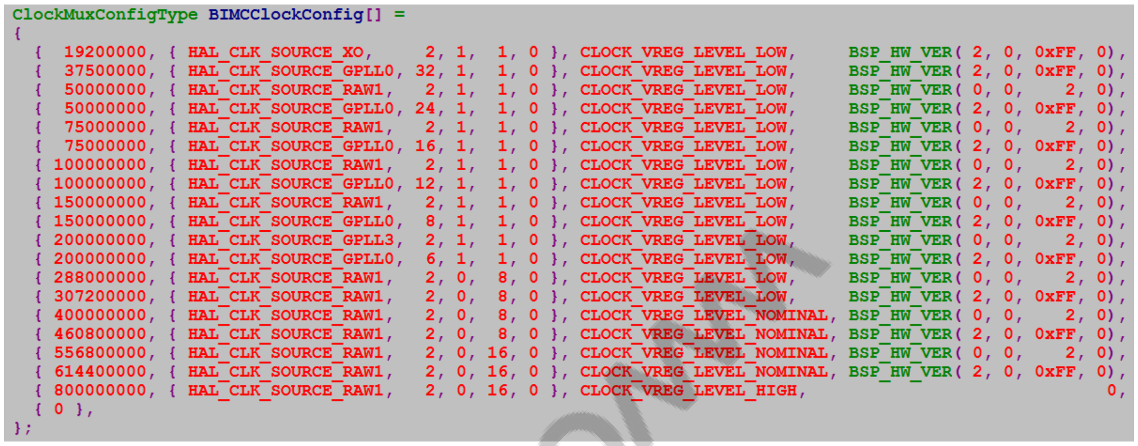
12.2 Err_fatal related crash

For the err_fatal casued crash issue, we could check the parsed RPM log, it will indicate why the crash happens. Then we can use **rpm_restore_from_core.cmm** script to restore its call stack. With the debug information, we could figure out where exactly the crash happens.

12.3 DDR related crash

Comparing to DDR2, DDR3 becomes much more critical on timing. Usually we could try lower its freq to see whether this issue could get improved/solved, to identify whether it’s DDR related.

On Family A, we can do this from DDR’s NPA callback. On Family B ,we need to achieve this as below:



```
ClockMuxConfigType BIMCClockConfig[] =
{
  {  19200000, { HAL_CLK_SOURCE_XO,     2, 1,  1, 0 }, CLOCK_VREG_LEVEL_LOW,     BSP_HW_VER( 2, 0, 0xFF, 0),
  {  37500000, { HAL_CLK_SOURCE_GPLL0, 32, 1,  1, 0 }, CLOCK_VREG_LEVEL_LOW,     BSP_HW_VER( 2, 0, 0xFF, 0),
  {  50000000, { HAL_CLK_SOURCE_RAW1,   2, 1,  1, 0 }, CLOCK_VREG_LEVEL_LOW,     BSP_HW_VER( 0, 0,    2, 0),
  {  50000000, { HAL_CLK_SOURCE_GPLL0, 24, 1,  1, 0 }, CLOCK_VREG_LEVEL_LOW,     BSP_HW_VER( 2, 0, 0xFF, 0),
  {  75000000, { HAL_CLK_SOURCE_RAW1,   2, 1,  1, 0 }, CLOCK_VREG_LEVEL_LOW,     BSP_HW_VER( 0, 0,    2, 0),
  {  75000000, { HAL_CLK_SOURCE_GPLL0, 16, 1,  1, 0 }, CLOCK_VREG_LEVEL_LOW,     BSP_HW_VER( 2, 0, 0xFF, 0),
  { 100000000, { HAL_CLK_SOURCE_RAW1,   2, 1,  1, 0 }, CLOCK_VREG_LEVEL_LOW,     BSP_HW_VER( 0, 0,    2, 0),
  { 100000000, { HAL_CLK_SOURCE_GPLL0, 12, 1,  1, 0 }, CLOCK_VREG_LEVEL_LOW,     BSP_HW_VER( 2, 0, 0xFF, 0),
  { 150000000, { HAL_CLK_SOURCE_RAW1,   2, 1,  1, 0 }, CLOCK_VREG_LEVEL_LOW,     BSP_HW_VER( 0, 0,    2, 0),
  { 150000000, { HAL_CLK_SOURCE_GPLL0,  8, 1,  1, 0 }, CLOCK_VREG_LEVEL_LOW,     BSP_HW_VER( 2, 0, 0xFF, 0),
  { 200000000, { HAL_CLK_SOURCE_GPLL3,  2, 1,  1, 0 }, CLOCK_VREG_LEVEL_LOW,     BSP_HW_VER( 0, 0,    2, 0),
  { 200000000, { HAL_CLK_SOURCE_GPLL0,  6, 1,  1, 0 }, CLOCK_VREG_LEVEL_LOW,     BSP_HW_VER( 2, 0, 0xFF, 0),
  { 288000000, { HAL_CLK_SOURCE_RAW1,   2, 0,  8, 0 }, CLOCK_VREG_LEVEL_LOW,     BSP_HW_VER( 0, 0,    2, 0),
  { 307200000, { HAL_CLK_SOURCE_RAW1,   2, 0,  8, 0 }, CLOCK_VREG_LEVEL_LOW,     BSP_HW_VER( 2, 0, 0xFF, 0),
  { 400000000, { HAL_CLK_SOURCE_RAW1,   2, 0,  8, 0 }, CLOCK_VREG_LEVEL_NOMINAL, BSP_HW_VER( 0, 0,    2, 0),
  { 460800000, { HAL_CLK_SOURCE_RAW1,   2, 0,  8, 0 }, CLOCK_VREG_LEVEL_NOMINAL, BSP_HW_VER( 2, 0, 0xFF, 0),
  { 556800000, { HAL_CLK_SOURCE_RAW1,   2, 0, 16, 0 }, CLOCK_VREG_LEVEL_NOMINAL, BSP_HW_VER( 0, 0,    2, 0),
  { 614400000, { HAL_CLK_SOURCE_RAW1,   2, 0, 16, 0 }, CLOCK_VREG_LEVEL_NOMINAL, BSP_HW_VER( 2, 0, 0xFF, 0),
  { 800000000, { HAL_CLK_SOURCE_RAW1,   2, 0, 16, 0 }, CLOCK_VREG_LEVEL_HIGH,                              0,
  { 0 },
};
```

Figure 12-1 RPM code of BIMCClockConfig

12.4 Voltage related crash

Sometimes, customers report the phone may crash in deep sleep mode with small probability, we could try increase **cx_sleep_voltage_uv/mx_sleep_voltage** to see whether it works. Especially in the process of sleep/resume, improper setting could result in failure to wakeup.

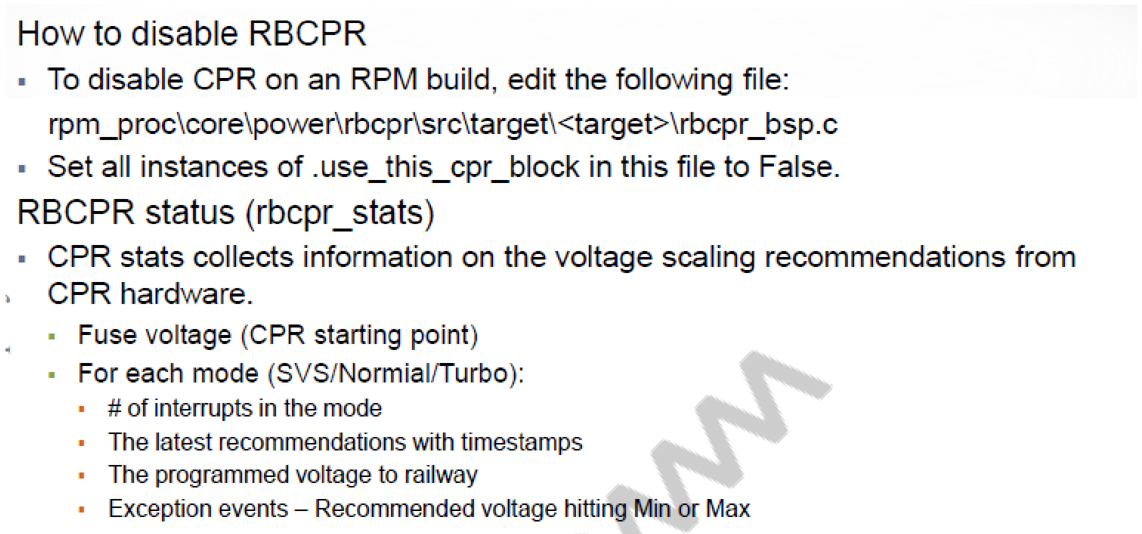
```
static sleep_target_config_data_t temp_sleep_target_config_data =
{
  /* mx_turbo_voltage_uv   */ 1050000,
  /* mx_nominal_voltage_uv */ 950000,
  /* mx_sleep_voltage_uv   */ 675000,

  /* cx_turbo_voltage_uv   */ 1050000,
  /* cx_nominal_voltage_uv */ 900000,
  /* cx_low_voltage_uv     */ 700000,
  /* cx_sleep_voltage_uv   */ 600000

};
```

- cx_sleep_voltage_uv, this value may vary on different chips, it’s decided dynamically during RPM initialization, PVS related
- I.

12.5 RBCPR related crash

The quickest way to identify an issue RBCPR related or not, is to disable this feature.

If confirmed, then move further the debugging.



How to disable RBCPR

- To disable CPR on an RPM build, edit the following file:
  rpm_proc\core\power\rbcpr\src\target\<target>\rbcpr_bsp.c
- Set all instances of .use_this_cpr_block in this file to False.

RBCPR status (rbcpr_stats)

- CPR stats collects information on the voltage scaling recommendations from CPR hardware.
  - Fuse voltage (CPR starting point)
  - For each mode (SVS/Normial/Turbo):
    - # of interrupts in the mode
    - The latest recommendations with timestamps
    - The programmed voltage to railway
    - Exception events – Recommended voltage hitting Min or Max

Figure 12-2 Disable RBCPR

13 Trustzone Debug

13.1 TZ dump analysis

We have some methods to analysis the trustzone dump based on USB dump.

- TzDumpParser.py

  `TzDumpParser.py msm8960 .\IMEM_C.bin .\IMEM_A.bin TzDump.txt`

  `TzDumpParser.py msm8974 .\MSGRAM.bin .\DDRCS0.bin TzDump.txt`

  `TzDumpParser_v2.py msm8974 .\OCIMEM.bin .\OCIMEM.bin TzDump.txt` (8974v2 platform)

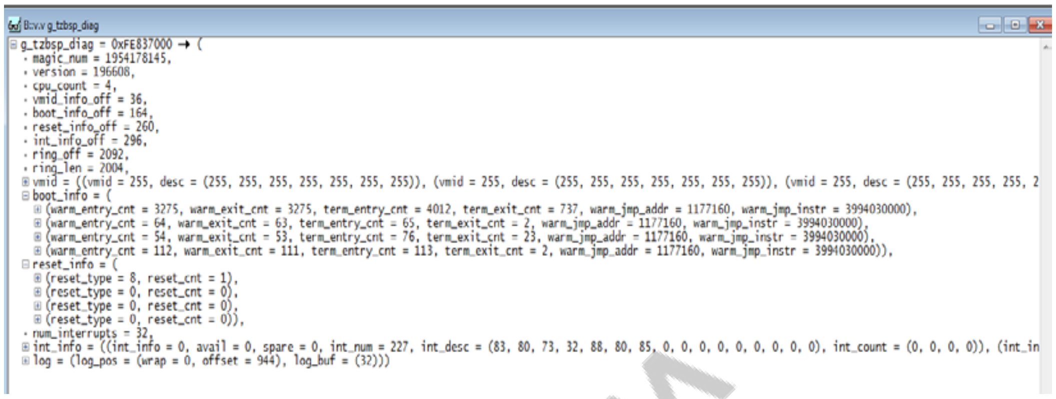
  Tzdump.txt will include tzlog and other useful information.

- Load TZ dumps to T32sim manually

  a. Load TZ backup memory
     - sys.resettarget
     - sys.cpu krait
     - sys.up
     - do load.cmm
     - d.save.b tz_from_ddr.bin 0x0fd16000++0x3A000
     - d.load.b tz_from_ddr.bin 0xfe806000
     - d.load.elf tz.elf /nocode /noclear

  b. Enable trustzone mmu mapping
     - PER.S C15:0x2 %LONG mmu_l1_tt
     - mmu.reset
     - mmu.scan
     - mmu.on

  c. Get TZ log and other information
     - v.v g_tzbsp_diag



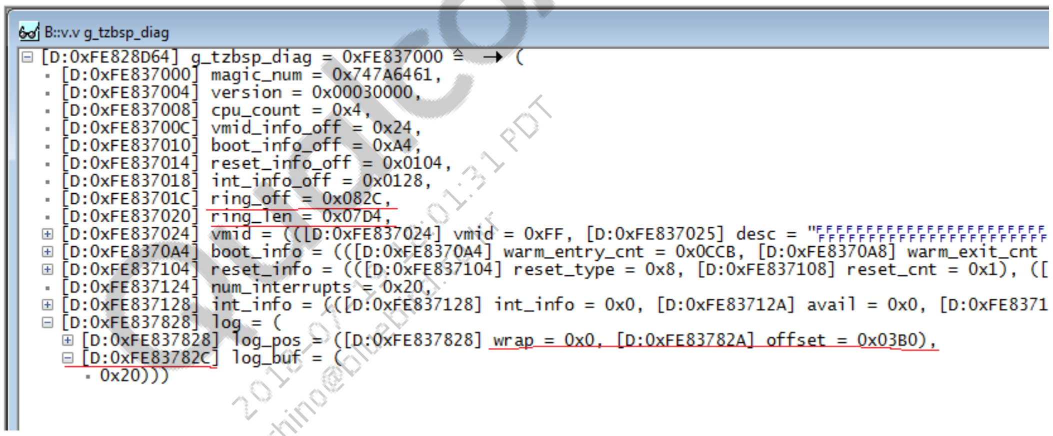
B::v.v g_tzbsp_diag

```
⊟ g_tzbsp_diag = 0xFE837000 → (
  · magic_num = 1954178145,
  · version = 196608,
  · cpu_count = 4,
  · vmid_info_off = 36,
  · boot_info_off = 164,
  · reset_info_off = 260,
  · int_info_off = 296,
  · ring_off = 2092,
  · ring_len = 2004,
  ⊞ vmid = ((vmid = 255, desc = (255, 255, 255, 255, 255, 255, 255)), (vmid = 255, desc = (255, 255, 255, 255, 255, 255, 255)), (vmid = 255, desc = (255, 255, 255, 255, 2
  ⊟ boot_info = (
    ⊞ (warm_entry_cnt = 3275, warm_exit_cnt = 3275, term_entry_cnt = 4012, term_exit_cnt = 737, warm_jmp_addr = 1177160, warm_jmp_instr = 3994030000),
    ⊞ (warm_entry_cnt = 64, warm_exit_cnt = 63, term_entry_cnt = 65, term_exit_cnt = 2, warm_jmp_addr = 1177160, warm_jmp_instr = 3994030000),
    ⊞ (warm_entry_cnt = 54, warm_exit_cnt = 53, term_entry_cnt = 76, term_exit_cnt = 23, warm_jmp_addr = 1177160, warm_jmp_instr = 3994030000),
    ⊞ (warm_entry_cnt = 112, warm_exit_cnt = 111, term_entry_cnt = 113, term_exit_cnt = 2, warm_jmp_addr = 1177160, warm_jmp_instr = 3994030000)),
  ⊟ reset_info = (
    ⊞ (reset_type = 8, reset_cnt = 1),
    ⊞ (reset_type = 0, reset_cnt = 0),
    ⊞ (reset_type = 0, reset_cnt = 0),
    ⊞ (reset_type = 0, reset_cnt = 0)),
  · num_interrupts = 32,
  ⊞ int_info = ((int_info = 0, avail = 0, spare = 0, int_num = 227, int_desc = (83, 80, 73, 32, 88, 80, 85, 0, 0, 0, 0, 0, 0, 0, 0, 0), int_count = (0, 0, 0, 0)), (int_in
  ⊞ log = (log_pos = (wrap = 0, offset = 944), log_buf = (32)))
```

- get tzlog

  d.save.b tzlog.txt &(location of g_tzbsp_diag->log->log_buf) ++ring_len

  d.save.b tzlog.txt 0xFE83782C++0x0704



B::v.v g_tzbsp_diag

```
⊟ [D:0xFE828D64] g_tzbsp_diag = 0xFE837000 ≙ → (
  · [D:0xFE837000] magic_num = 0x747A6461,
  · [D:0xFE837004] version = 0x00030000,
  · [D:0xFE837008] cpu_count = 0x4,
  · [D:0xFE83700C] vmid_info_off = 0x24,
  · [D:0xFE837010] boot_info_off = 0xA4,
  · [D:0xFE837014] reset_info_off = 0x0104,
  · [D:0xFE837018] int_info_off = 0x0128,
  · [D:0xFE83701C] ring_off = 0x082C,
  · [D:0xFE837020] ring_len = 0x07D4,
  ⊞ [D:0xFE837024] vmid = (([D:0xFE837024] vmid = 0xFF, [D:0xFE837025] desc = "FFFFFFFFFFFFFFFFFFFFFF
  ⊞ [D:0xFE8370A4] boot_info = (([D:0xFE8370A4] warm_entry_cnt = 0x0CCB, [D:0xFE8370A8] warm_exit_cnt
  ⊞ [D:0xFE837104] reset_info = (([D:0xFE837104] reset_type = 0x8, [D:0xFE837108] reset_cnt = 0x1), ([
  · [D:0xFE837124] num_interrupts = 0x20,
  ⊞ [D:0xFE837128] int_info = (([D:0xFE837128] int_info = 0x0, [D:0xFE83712A] avail = 0x0, [D:0xFE8371
  ⊟ [D:0xFE837828] log = (
    ⊞ [D:0xFE837828] log_pos = ([D:0xFE837828] wrap = 0x0, [D:0xFE83782A] offset = 0x03B0),
    ⊟ [D:0xFE83782C] log_buf = (
      · 0x20)))
```

13.2 XPU violation debugging

If TZ XPU violation happens, you can see TZ log like below:

```
xpu: ISR begin
XPU ERROR: Non Sec!!
xpu:>>> [1] XPU error dump, XPU id 3 (BIMC_MPU0)<<<
 xpu: uErrorFlags: 00000002
xpu:  HAL_XPU2_ERROR_F_CLIENT_PORT
 uBusFlags: 00080521
xpu:  HAL_XPU2_BUS_F_ERROR_AC
xpu:  HAL_XPU2_BUS_F_APROTNS
xpu:  HAL_XPU2_BUS_F_AOOO
xpu:  HAL_XPU2_BUS_F_ABURST
xpu:  HAL_XPU2_BUS_F_NONSECURE_RG_MATCH
 xpu: uPhysicalAddress: 0d3bb8a8
 xpu: uMasterId: 00000000, uAVMID   : 00000003
 xpu: uATID    : 00000000, uABID    : 00000002
 xpu: uAPID    : 00000000, uALen    : 00000000
 xpu: uASize   : 00000002, uAPReqPriority   : 00000000
 xpu: uAMemType: 00000000
Fatal Error: XPU_VIOLATION
```

How to analysis the XPU violation log:

1. XPU name and ID can be found in the first line of dump

2. uPhysicalAddress which indicates the address that is illegally accessed
   a. If the XPU ID listed is either "BIMC_MPU0" or "BIMC_MPU1", this means an XPU violation happens at a DDR memory address, check **`go/memorymap`** to see what region of memory is being accessed.
   b. If not so, this is an *offset* value onto some other register base; look for a register whose name matches up with the XPU ID
3. uAVMID indicates the VMID of requesting master that causes the access; see next page for VMID list, VIMD map for msm8974 is defined as below;

```
#define TZBSP_VMID_NOACCESS     0
#define TZBSP_VMID_RPM          1
#define TZBSP_VMID_TZ           2
#define TZBSP_VMID_AP           3
#define TZBSP_VMID_MSS          4
#define TZBSP_VMID_LPASS        5
#define TZBSP_VMID_WLAN         6
#define TZBSP_VMID_CP           7
#define TZBSP_VMID_VIDEO        8
#define TZBSP_VMID_DEHR         9
#define TZBSP_VMID_OCMEM_DM     10
```

4. Look up the **uAPID**, **uABID**, and **uMasterId**, determine what tries to do the access. (These correspond to the **PID**, **BID**, and **MID**, respectively).
5. From the previous interrupt, get the core happening Xpu violation, and then check the corresponding RTB log, to check the last **LOGK_READL** or **LOGK_WRITEL** event. see if it corresponds to a register access that looks uncommon or related to a particular hardware block or subsystem

For XPU violation, it's usually critical and we need to involve many internal documents to narrow the error place. Please raise case to QCT if you find such issues.

13.3 AHB timeout debugging

Take msm8974 as sample, 8974 has a new hardware debug feature for detecting bus hangs on most AHB buses. This can be used to identify which address typically (a memory-mapped register access) fails to be accessed (usually because a necessary clock is disabled).

When a hang is detected, hardware forcefully completes the transaction (without actually reading/writing the intended address) and raises an interrupt to which TZ software subscribes. TZ then logs the syndrome information for analysis.

When a timeout is detected, TZ will log the event in the following format. It can be retrieved from the `TZParsedLog*` text file in a parsed ramdump.

```
ABT CNOC_2 ID: 0x00004000
ABT CNOC_2 ADDR0: 0x00501004
ABT CNOC_2 ADDR1: 0x00000000
ABT CNOC_2 HREADY: 0xfffffff7
ABT CNOC_2 Slaves: 5
ABT MMSS_0 ID: 0x00004000
ABT MMSS_0 ADDR0: 0x00001004
ABT MMSS_0 ADDR1: 0x00000000
ABT MMSS_0 HREADY: 0xfffffffb
ABT MMSS_0 Slaves: 5
Fatal Error: AHB_TIMEOUT
```

From the above, we can immediately see that the failing access generates timeouts at both CNOC_2 and MMSS_0, so the access must have traversed both the Config NOC and MMSS NOC buses.

We can also see that the CNOC_2 physical address is 0x00501004, and the MMSS_0 physical address is 0x00001004. Note the similarity between these addresses, and how their bottom bits are the same. This is expected. With every bus the access goes over, a few of the upper bits are stripped off and used to determine the next bus the access should be routed through on its way to its destination. For the purposes of debugging, focus on the ABT messages associated with the most specific address- in this case, the ones for MMSS_0:

```
ABT MMSS_0 ID: 0x00004000
ABT MMSS_0 ADDR0: 0x00001004
ABT MMSS_0 ADDR1: 0x00000000
ABT MMSS_0 HREADY: 0xfffffffb
ABT MMSS_0 Slaves: 5
```

The next step is to determine what register lives at offset 0x1004 from the MMSS_0 base. And judge what causes the AHB timeout, we need to involve many Qualcomm internal docs, which will not be released to customers later.

So the best way when you find AHB timeout error is to raise a case to Qualcomm .

14 Hardware Checklist

14.1 When need to involve hardware team

The hardware checklist should be involved and checked when we meet the following issues:

- Issues only happens on particular version of chip and software cannot find anything abnormal
- Issues only happens on particular version of customer hardware PCB and software cannot find anything abnormal
- Issues only happens when using a particular MCP and software cannot find anything abnormal
- Software has found memory not stable
- Software has found cores don't behavior normally

14.2 Hardware checklist

Basically, from the software perspective, the hardware checklist should include the following items:

- Whether the clock is stable and especially the crystal is stable if we find cores behavior abnormal, especially, whether there is a clock glitch, whether there is a reflection on clock path, whether need increase drive strength of clock pad.
- Whether memory voltage is stable and whether it is out of spec when memory bitflip issue happens, especially the issue disappears when we disable DCVS.
- For special memory issue, we need to check whether the memory voltage and memory frequency are out of sync
- For sleep/wake issue, we need to check whether core voltage is enough to bring it up, whether the setup time is out of spec. Also we need to monitor both VDD_CX and VDD_MX to make sure it is correct.
- For some reset issues, we also need to check the power on sequence waveforms, for example, PS_HOLD, PON_RESET
- If we point the issue lies in particular driver, sometimes we need hardware check the controller waveforms, for example, SDCC, USB etc.
- For platforms that has CPR hardware, we may try to increase floor voltage or aggressively, disable CPR for some unknown reset.
- For some unknown reset scenario that application processor failed to bringup, we need to check SPM delay time, clock and vdd_cx voltage.

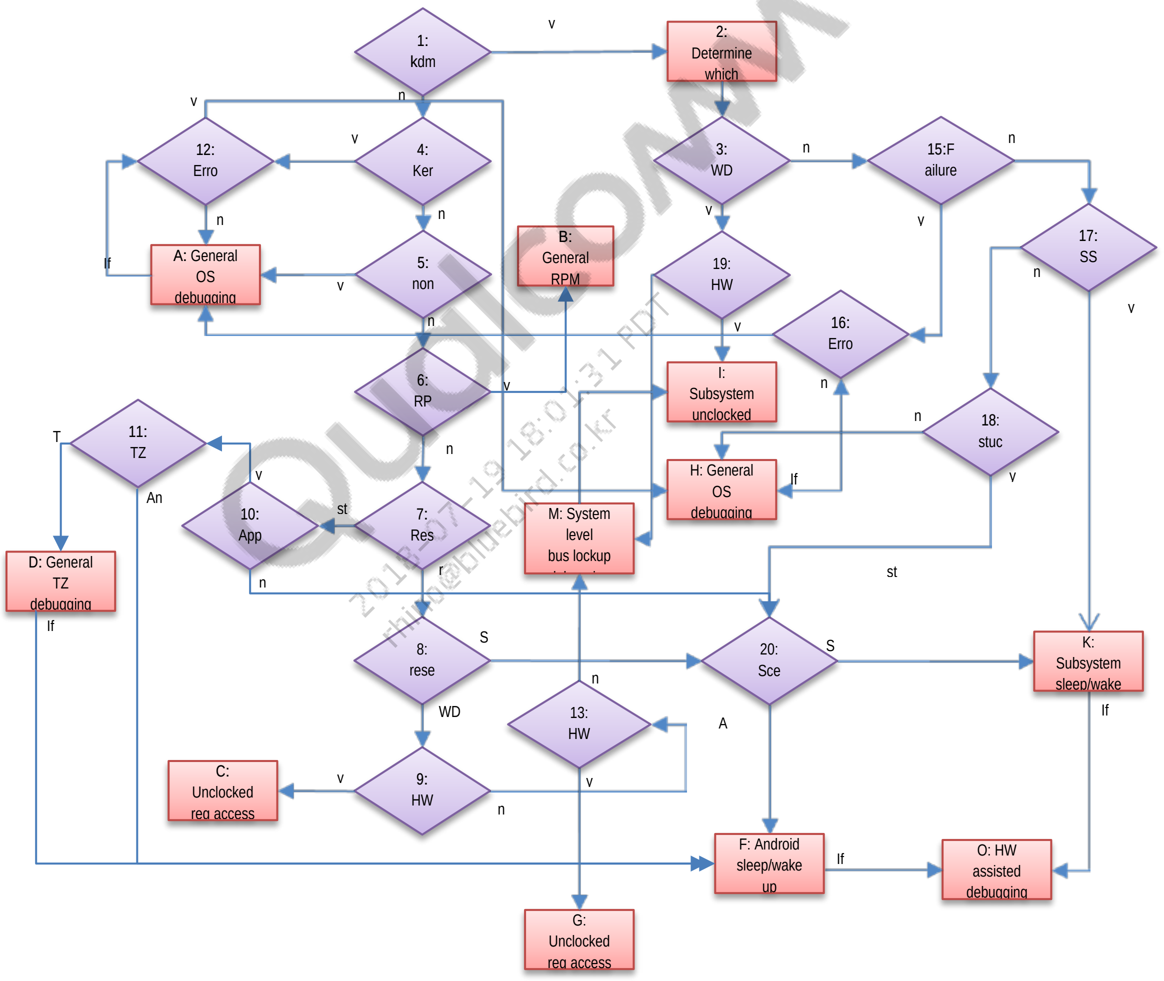
14.3 Chip category related

Also some issue may be related with chip category.

- For the issues that happen on particular PVS part, we may increase the vdd of krait core for a try
- For the issues that happen on some old chips with RCPCR enabled, we can try to disable it again. Please refer to Chapter 12.5_RBCPR_related_crash.

15 Debug Flow Chart

15.1 Flowchart



15.2 Detailed explanation of flowchart

1. Check subsystem failure (Steps 1 to 3).

   Check subsystem failure first. Upon subsystem failure, you will see the following error messages from kdmsg (Android kernel debug message):

   - Modem error fatal/WD bark/exception – “**Modem SMSM state changed to SMSM_RESET. Probable err_fatal on the modem.**”
   - Modem WD bite – “**WD bite received from modem Software!**”
   - WCNSS Error Fatal/DogBark/Exceptions – “**SMSM reset request received from Riva**”
   - WCNSS WD bite – “**WD bite received from Riva**”
   - LPASS error fatal/exception/WD bite – “**WD bite received from Q6!**”
   - LPASS not responsive – “**apr_tal:open timeout**” “**APR: Unable to open handle**”

2. Check kernel message for Android kernel panic (Step 4).

   - Check kernel panic from the kernel debug messages. If there is a kernel panic, you can find the following messages easily:
     - Exception – “**Kernel panic - not syncing: Fatal exception**”
     - Others – “**Kernel panic - not syncing: xxxxxx**” where xxxxxx varies based on failures
   - Nonsecure Krait WD bark is now handled by apps (different from MSM8960), so you will see a kernel panic message for that.

3. Check Krait secure WD bark/nonsecure WD bite (Step 5).

   - Both apps secure WD bark and nonsecure WD bite are handled by TZ.
   - Use Linux parser to check the information.

4. Check RPM error (Step 6).

   Check RPM error, since Krait may not get that in the debug message. You can find the following error messages in the RPM log in case of RPM error:

   20183.020111: rpm_error_fatal (pc: 0x00033718) (a: 0x00902fd0) (b: 0x00000000) (c: 0x00000000)

5. Check reset or stuck (Step 7).

   If there are no errors in the kernel debug message or RPM log, this may mean that the error handler(s) does not work properly on the issue. There are two known situations in which the error handler cannot run.

   - First is a device reset, since there is no time for software to do anything in hardware reset.
   - Second is a device lockup. If somehow the device does not crash, but also does not run at all, we call the device locked up for some reason. The common problem is the CPU core goes bad while waking up from power-collapse due to improper voltage or clocks.

   However, you cannot distinguish reset or being stuck completely by logs, so you need to rely on the tester s’ information on the issue because testers know if the device is stuck or reset before taking the dump.

6. Check SMPL or WD (Step 8).

   These are two common causes for reset, SMPL and WD.

   - The first one is hardware WD (secure WD bite). In case of WD reset, WDT will collect debug information.
   - In addition to WDT, reset reason will be captured by QDSS.

7. Check unclocked register access from Krait (Step 9).

   There are two common situations that lead to Krait WD bite: unclocked register access and bus hung

   - In case of unclocked register access from Krait, RTB will help, since it provides any hardware register access that may cause a bite. PFT will also help.
   - In addition to RTB, there may be a clock status on those hardware blocks by hardware events.
   - If there is no suspicious hardware register access from Krait, this might be a bus hang situation. Look into syndrome information from bus hang detection.

8. Check apps is in the middle of entering sleep or waking up (Step 10).

   a. Check if apps is in the middle of entering power collapse or waking up from power collapse. If it is in the middle of a sleep operation, look into sleep-related modules.

   b. If not in the middle of a sleep operation, check if apps get stuck due to a software reason (less likely); if not, look into other hardware factors.

   c. Meanwhile, the RPM log will tell a core's expected status (alive or not).

9. Determine TZ or Android (Steps 10 and 11).

   - Check whether TZ or Android is running at the moment using TZ counter. TZ counters are saved at TZBSP_SHARED_IMEM_DIAG_ADDR + 0xA4.
   - In addition to TZ counter, we can also use PFT to locate which instruction is executed.

10. Check MPROC-related error (Step 12/16).

    a. Since all subsystems eventually fail in case of another core's error, determine which core gets the error first and focus on that.

    b. Checking SMEMLOG usually helps figure out which core gets the error and stops first.

11. Check RPM or Krait for WD bite reset (Step 13).

    a. Ideally, RPM bite is handled by apps.

    b. If apps cannot handle it, a secure WD bite will happen eventually.

12. Check unclocked register access from RPM (Step 14).

    a. The RPM register value can be restored, which is saved by RPM PBL. We can check if there is hardware register access at the moment, which may cause a WD bite.

    b. Syndrome information from bus hang detection may help in this situation.

13. Check subsystem failure (Step 15).
    a. Each subsystem has its own debug message; failures can be checked from those logs first in case of a software error/exception.
    b. For Modem/LPASS, there are F3 messages.
    c. For WCNSS, debug message is logged through kernel debug message.
14. Check being stuck (Step 17).

    If there is no error information in the log, two known situations are the subsystem gets stuck or the error handler in the subsystem does not run properly. A distinction needs to be made.
15. Check whether subsystem is in the middle of wakeup (Step 17).

    A common situation of being stuck is in the middle of waking up the core; check this possibility first by looking at the RPM log, as well as the internal variable. If it gets stuck in the middle of waking up, look into details of the sleep operations.
16. Check whether subsystem gets stuck or reset due to hardware (Step 18).

    Sometimes a subsystem gets stuck or reset due to a nonsoftware reason. In this case, we may suspect a hardware malfunction like voltage fluctuation. Note that it is hard to distinguish if the software causes this, so all software possibilities need to be ruled out first.
17. Check hardware register access from subsystem (Step 19).
    a. If the subsystem gets a WD bite, check whether the subsystem have unclocked register access. In case of a Hexagon WD bite, NMI will be triggered and collect debug information. If no hardware unclocked register access from the subsystem is found, check bus hang.
    b. ETM of subsystem and bus hang syndrome information may help in this situation.
18. Check if SMPL is an apps-related behavior or other subsystem behavior (Step 20).

    In case of SMPL, narrow down which core's behavior is related to the SMPL. Those SMPLs might be hardware-related issues, which will provide a clue where to look from a hardware perspective.

A Appendix

A.1 Trace32 usage

Trace32 is a common used tool for on-target debugging. Each release notes will talk about how to open the trace32 shortcut and how to attach it and debug. The folder `<meta build>\common\tools\cmm` also has a lot of scripts that can be used on trace32.

A.2 How to enable debugfs in kernel

Debugfs is a very useful filesystem that can be used for module/kernel status checking.

1. **CONFIG_DEBUG_FS** needs to be enabled in kernel config
2. In adb shell, **`mount –t debugfs debugfs /sys/kernel/debug`**
3. **`cd /sys/kernel/debug`**, then we can see the debugfs directory tree can be used.

A.3 How to use the multiple debug config in kernel

Kernel has a very strong debug mechanism which can be used to debug mutex, spinlock, memory leak, interrupt latency etc. The detailed document can be referred to is [Q2] and the `<android root>/kernel/Documentation`.

A.4 RTB explanation

Please refer to Solution **00024343**, **00024789** and **00024350**.

A.5 ETM explanation and usage

Please refer to Solution **00027974**.

A.6 GDB to analyze application coredump

Please refer to Solution **00028327** (how to get coredump), We can use the arm-eabi-gdb to analyze dump and all the gdb commands can be used.