RPM Debug Manual

80-NM128-1 A

February 5, 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

1 Introduction

1.1 Purpose

This document provides information about software debugging logs, tools, key structures, and techniques applicable to the MSM8974 family chipset RPM.

At the time of document creation, the MSM8974 family includes the MSM8974, MDM9x25, MSM8x26, MSM8x10, APQ8084, and MPQ8092 chipsets. Additional chipsets are being added to the family.

1.2 Scope

This document provides software engineers with the necessary information for debugging the MSM8974 family chipset Resource Power Manager (RPM).

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>`.

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 Reference documents that are no longer applicable are deleted from this table; therefore, reference numbers may not be sequential.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	*Resource Power Manager User Guide*	80-N6955-1
Q3	*Presentation: RPM Debug Overview*	80-NA157-9

1.5 Technical assistance

For assistance or clarification on information in this guide, 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@qualcomm.com.

1.6 Acronyms

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

2 RPM Overview

2.1 Hardware overview

Figure 2-1 is a top-level example block diagram of the RPM.

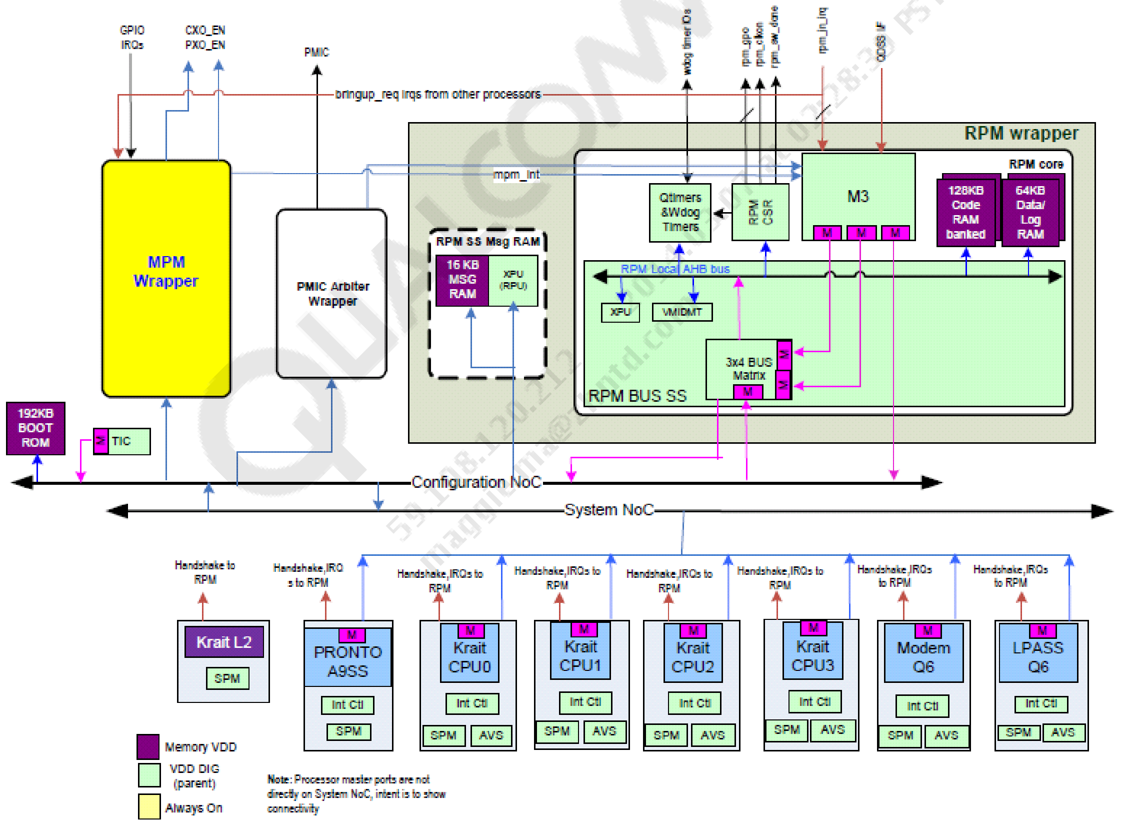

Figure 2-1 RPM example block diagram

2.1.1 RPM processor

The RPM core consists of:

- Cortex-M3 processor with integrated Nested Vectored Interrupt Controller (NVIC)
- 128 KB multibank code RAM
- 64 KB multibank data/log RAM
- 16 KB message RAM

The RPM processor has a native Advanced High-performance Bus (AHB) Lite interface and built-in ETM/ITM and DAP support. The processor target frequency is 100 MHz. The Cortex-M3 supports a native SWFI instruction, which allows the processor to turn off its own clock when the clock is not needed.

The Cortex-M3 processor uses a Harvard architecture, enabling simultaneous instruction fetch with data load/store. It supports thumb2, single-cycle 32-bit multiply, and hardware divide.

2.1.2 AHB

The RPM AHB is synchronous to the RPM processor. It instantiates an aPU, which is configured to protect the code RAM and mPU. The code RAM and mPU are configured to protect different predecoded address spaces within the RPM. The RPM AHB has Cortex-M3 as the default master for the bus. The RPM AHB allows byte, half-word, and word accesses for the code and data/log RAM. The RPM AHB allows only word accesses for CSR registers.

2.1.3 Boot ROM

The RPM boot ROM block houses the actual ROM that the RPM processor uses to boot out of on system reset, along with the AHB logic to access it through the RPM AHB bus. The main purpose of the boot ROM is to store the Primary Boot Loader (PBL). The PBL is the first piece of code that the chip executes and is responsible for initial hardware setup to a point where further bootup can proceed from the Flash. The PBL must reside in an internal boot ROM to guarantee that the chip always boots from a known trusted code.

2.1.4 Code RAM

The RPM instantiates a 128 KB code RAM. It is a single-port compiler memory. The RPM code RAM operates at a clock frequency that is synchronous to the RPM AHB. The RPM code RAM allows single-cycle read access and is 32 bits wide. The Cortex-M3 core data accesses can spill into code RAM space if the software chooses, though this is at a loss of optimal performance.

2.1.5 Data/log RAM

The RPM instantiates a 64 KB data/log RAM. This RAM is a single-port compiler memory. The RPM data RAM operates at a clock frequency that is synchronous to the RPM AHB. The RPM data RAM allows single-cycle read access and is 32 bits wide. The Cortex-M3 core instruction accesses can spill into data RAM space if the software chooses, though this is at a loss of optimal performance.

2.1.6 Message RAM

The RPM message RAM provides memory for sending messages to and from the RPM core. The messaging masters, shown in Table 2-1, use this memory to communicate with the RPM.

Table 2-1 RPM messaging masters

Chipset	MSM8974	MDM9x25
Messaging masters	▪ APSS (Krait) ▪ Modem ▪ LPASS ▪ WCNSS	▪ APSS (A5) ▪ Modem ▪ LPASS

2.1.7 Interrupt controller

The Cortex-M3 has a built-in NVIC with a configurable number of interrupts. The RPM is configured to use 64 interrupts. Sources external to the RPM include MPM, SPM, messaging, and nonmessaging masters. Sources internal to the RPM include general-purpose timers, WDOG timer bark, and CTI interrupts. The RPM interrupt controller provides priority queuing of interrupts and supports both edge-sensitive and level-sensitive interrupts.

The NVIC can be fully accessed only from Privileged mode, but interrupts can pend in User mode if they are enabled via the Configuration Control Register (CCR). Any other User mode access causes a bus fault. All NVIC registers are accessible using byte, half-word, and word, unless otherwise stated. All NVIC registers and system debug registers are little-endian, regardless of the endian state of the processor.

2.1.8 Timers

The RPM instantiates the QTimer with two frames, one for the kernel and one for the user. The QTimer keeps track of real-time in every power mode.

The RPM also has a WDOG timer running on the Sleep clock, with a configurable bark and bite expiration.

2.1.9 CSR

The RPM Control/Status Register (CSR) provides IPC and GPO registers to generate IPC interrupts and general-purpose pulses respectively. The RPM CSR provides a WFI_CONFIG register to generate requests to turn off the RPM bus clocks and CHIP_SLEEP_EN, also known as SW_DONE, to signal the MSM Power Manager (MPM) that the MSM is ready for sleep. WDOG timer software interface registers, test bus configuration registers, and other miscellaneous use registers are supported.

2.2 RPM software overview

The RPM software topology is shown in Figure 2-2.

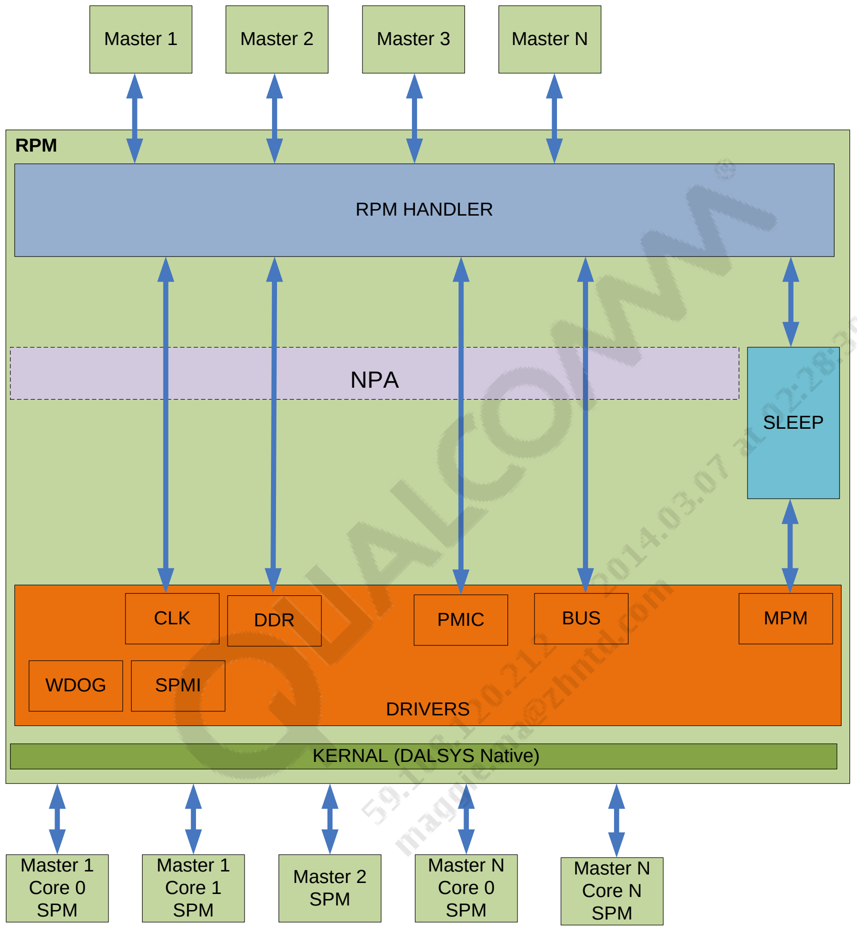

Figure 2-2 RPM software topology

2.2.1 Kernel

The kernel for the RPM supports:

- Interrupts
- Intlock, priorities, and configuration
- Busy waits
- Timers
- SWFI
- Reduced code size

The kernel is implemented using DalSys to the metal. This provides the RPM with a lightweight kernel and allows easier driver porting.

2.2.2 RPM handler

The RPM handler abstracts the RPM message protocol away from other software. The driver handles the RPM-side client and server portions of the RPM messaging and hands data to the rest of the drivers via callbacks.

2.2.3 Drivers

Drivers for each resource are supported by the RPM register with the RPM handler to request notification when requests are received for the resource that the driver controls. Upon receiving this notification, the drivers perform arbitration that must be performed between the new request and previous requests from other masters. Based on the arbitration results, the driver determines how to modify the hardware resource.

2.2.3.1 NPA

A driver may use Node Power Architecture (NPA) to represent resources controlled by the driver. NPA is a generic framework that allows nodes to represent resources. Each node has clients and is responsible for aggregating workload requirements on their resources while optimizing power usage. The nodes make up a distributed graph, allowing one node to be a client of another node.

2.2.3.2 Clock driver

The clock driver consists of two parts, one part resides on each master and the other part resides on the RPM.

The RPM clock driver directly handles aggregating requests from each master for the systemwide clock resources controlled by the RPM. The driver also handles RPM-specific clocks.

2.2.3.3 Bus arbiter driver

The bus arbiter driver consists of multiple parts, one part resides on each master and one part resides on the RPM.

The RPM bus arbiter driver takes bus arbiter settings as requests from the different masters in the system and aggregates them to represent the frequency-independent system settings. The frequency required to meet the settings is calculated from these settings.

Using the calculated value, the bus arbiter driver makes a request of the RPM clock driver. The clock driver request sets a floor for the frequency at which the buses/FABRICs can operate. The system settings are then converted into frequency-dependent settings and the driver configures the bus arbiter hardware with those settings.

2.2.3.4 PMIC driver

The PMIC driver consists of two parts, one part resides on each master and the other part resides on the RPM.

The RPM PMIC driver directly aggregates requests from each master for the systemwide PMIC resources controlled by the RPM.

2.2.3.5 WDOG driver

The WDOG is a fail-safe for incorrect or stuck code. If a register is not written within a specific time period, an interrupt occurs that allows the software to attempt to recover. If the register is still not written within a specific time period, the system resets.

For the RPM subsystem, only the scheduler has the capability to pet the WDOG and reset the WDOG counter. There is no software task monitoring other software tasks for stuck conditions. Dealing with the WDOG during scheduler operations should meet the fail-safe needs. The software does not have to explicitly interact with the WDOG in most scenarios.

The WDOG also supports a freeze that stops the countdown for scenarios where the scheduler does not run within a given timeframe. Examples of scenarios requiring freeze support include long memory or hardware accesses. In most cases, those scenarios should be avoided by the software design. Freeze is supported when the RPM enters Sleep mode.

The RPM watchdog driver cannot reset the system.

2.2.3.6 MPM driver

The MPM driver is used to program the MPM hardware block during systemwide sleep. This driver resides on the RPM and is responsible for programming the MPM to perform:

- Vdd_Dig retention – Puts the systemwide power rail in the Retention state
- Vdd_Dig collapse – Puts the systemwide power rail in the Collapse state
- Vdd_Mem retention – Puts the systemwide power rail in the Retention state
- Vdd_Mem collapse – Puts the systemwide power rail in the Collapse state
- CXO shutdown – Turns off CXO

The driver must support programming of the MPM timer hardware and the MPM interrupt controller.

The MPM driver must also initialize several configuration registers to correctly handle the hardware combination and configuration needs of the system.

3 RPM Build Instructions

3.1 Tools preparation

To make the RPM build, the licensee must have the tools described in this chapter.

Python 2.x

As a general rule, any 2.x version should work. To verify the version:

1. Go to Start→Run→cmd.exe.
2. At the prompt, run **python –v**.

If the version is not on the 2.6.x line, the licensee should obtain and run the most current 2.6.x Windows installer from http://www.python.org/download.

Nothing should be leveraged from the 3.x line, as it is not backward-compatible.

After installation, execute a command prompt and verify that Python installed into the path correctly. Sometimes it does not install correctly, depending upon the selections made during installation. To verify:

3. Go to Start→Run→cmd.exe.
4. At the prompt, run **python**.
   a. If nonapplicable information followed by a >>> prompt appears, Python installed correctly. Press **Ctrl+Z** followed by **Enter** to exit.
   b. If an error appears, add C:\Python26 and C:\Python26\Scripts to the correct path.

SCons 1.2.0

Later versions should work. SCons 1.2.0 is the most current version at the time of this document release.

1. Verify C:\Python26\Scripts is in the path. Add it if it is not.
2. Verify the SCons version.
   a. Go to Start→Run→cmd.exe.
   b. At the prompt, run **scons –v**.

Make note of the information that follows engine:. If an error occurs when the version check is run or the engine version is less than 1.2.0, obtain and run the Windows installer from the Stable column of http://www.scons.org/download.php.

ARMCT 5.01 B94

ARMCT 5.01 Build 94 is required for building the MSM8974 chipset family RPM.

3.2 Build process

3.2.1 RPM build

To build RPM in a Windows environment:

1. cd <path_to_rpm_source>\build.
2. Run build_<target>.bat, e.g., Build_8974.bat.

   The RPM image, RPM.elf, and the final image, RPM_hash_nonsec.mbn, are generated in the rpm_proc\core\bsp\rpm\build folder.
3. To make clean, run **Build_<target>.bat –c** in the build directory.

To build RPM in a Linux environment:

1. Set up the build enviroment.
2. Run **./build_<target>.sh** in the build directory.

3.2.2 Known build issues

If a compiling error with Subprocess.Popen occurs, ensure ARMCT5 is installed and remove rpm_proc/build/compiler_cache.pkl from the build.

4 RPM Debugging Overview

4.1 Trace32 scripts

4.1.1 Save RAM dump – rpm_dump.cmm

If the RPM crashes and there is no expert available, capture the state of the session for later analysis.

```
do rpm_dump.cmm \\location\to\put\logs
```

4.1.2 Load RAM dump – rpm_load_dump.cmm

If rpm_dump.cmm was used to capture the state of an RPM session, it can be tedious to manually restore the many dumps. Use the following script to accelerate this process:

```
do rpm_load_dump.cmm \\location\of\logs
```

4.1.3 Restore a crash – rpm_restore_core.cmm

If the RPM crashes, it is likely that it dumped its core and then moved on to another crash management code. To restore the RPM to a point that is as close as possible to the point of the crash, use the following script to load the core dump:

```
do rpm_restore_core.cmm
```

NOTE: This script requires that memory dumps and symbols are loaded before it can be used.

4.1.4 Parse log – rpm_parse_faults.cmm

An RPM crash may be due to a software fault. If so, the core dump may include useful information about the fault that occurred. To analyze this information, run:

```
do rpm_parse_faults.cmm
```

NOTE: This script requires that memory dumps and symbols are loaded before it can be used.

4.1.5 Examine the preempted process – rpm_m3_unstack.cmm

A dead RPM may indicate that an executing process was preempted by a software fault handler or interrupt. To examine the interrupted process, navigate to the top of the fault handler and run:

```
do rpm_m3_unstack.cmms
```

Since the core dump generally occurs at the top of the fault handler, running this script immediately after rpm_restore_core.cmm usually places you at the faulting instruction.

4.2 Getting the RPM logs using T32

The RPM has a log that is very useful for determining what has occurred on the RPM. This is the primary source of debugging reset and hang problems and can also be useful for looking at performance issues.

While attached to the RPM in the Break state, run the following commands in T32:

```
do rpm_proc\core\power\ulog\scripts\ULogDump.cmm <path to your directory>
do rpm_proc\core\power\npa\scripts\NPADump.cmm <path to your directory>
```

This places the RPM external log and the NPA log, which only contains dump information, into the log directory. The RPM external log requires use of a parsing tool to interpret. To run the ROM external log, run the following command from the log directory:

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

Additional switches are –r, which print raw (hex sclk value) timestamps.

4.3 Getting the RPM logs using Hansei script

Hansei is a tool for parsing interesting debug information straight out of the RAM dumps. This should enable faster triage turnaround and provides more information with less effort.

Hansei is included with the RPM release, located at rpm_proc\core\bsp\rpm\scripts\hansei.

4.3.1 Prerequisites

4.3.1.1 Python 2.7.x

Python 2.7.x must be installed. If the Python version is unknown, open a command window and run python -V.

4.3.1.2 Pyelftools library

A version of the pyelftools library that supports the ARM compiler tools must be installed. Unfortunately, this is not supported by the mainline version of pyelftools.

The version can be retrieved from https://bitbucket.org/pplesnar/pyelftools-pp.

After the pyelftools source code is extracted, install it by running the command prompt, changing to the source directory, and running python setup.py install.

4.3.2 Usage

The script accepts a few options. This information can also be retrieved by running hansei.py –h.

```
usage: hansei.py [-h] [--elf rpm.elf] [--output logs_path] [--verbose]
                 [--parser parse_path] [--parser_rel]
                 dumpfile [dumpfile ...]
```

4.3.2.1 Output

- rpm-summary.txt – Contains general information about the health of the RPM, including the core dump state and fault information
- rpm-log.txt – Postprocessed RPM external log from the MSM8960 family chipsets
- rpm-rawts.txt – Same log as rpm-log.txt but processed with the raw timestamp option for a hexadecimal left column in QTimer ticks
- npa-dump.txt – Standard NPA dump format without inaccurate timestamps of the MSM8960 family chipsets or T32-dumped versions
- ee-status.txt – Contains information about the subsystems and their cores that are active or sleeping
- reqs_by_master/* – Folder containing a file for each execution environment, detailing the current requests EE has in place with the RPM
- reqs_by_resource/* – Folder structure containing a folder for each of the resource types registered with the RPM server; under that folder, there is a file containing the requests to each resource of that type
- mpm.txt – Dump of the current MPM registers
- railway.txt – Dump of the state of railway, including voter lists
- sleep_stats.txt – Dump of sleep statistics shared with all masters

Example

```
python hansei.py -e rpm.elf -o . rpm_code_ram.bin rpm_data_ram.bin
rpm_msg_ram.bin
```

5 Debugging Railway

5.1 Check railway status

```
Railway.rail_state[x]; x= rail# (mx=0/cx=1/gfx=2)
```

5.2 Check railway voters

- Follow voter_list_head and voter_link to see all voters
- Voter ID
  - `master` number (0: APPS, 1: MODEM, 2: QDSP, 3: RIVA)
  - typedef enum

```
typedef enum
{
    RAILWAY_SVS_VOTER_ID = 100,
    RAILWAY_RPM_CX_VOTER_ID,
    RAILWAY_RPM_MX_VOTER_ID,
    RAILWAY_DDR_TRAINING_VOTER_ID,
    RAILWAY_RPM_BRINGUP_VOTER,
    RAILWAY_RPM_INIT_VOTER,
    RAILWAY_CLOCK_DRIVER_VOTER_ID,
    RAILWAY_CPR_SETTLING_VOTER,
} railway_voter_id; Railway
```

5.3 Change railway settings

To tune the railway voltage settings, modify the default_uv of the following structure in railway_config.c:

```
    static const railway_config_data_t temp_config_data =
    {
        .rails     = (railway_rail_config_t[])
        {
            {
            //Must init Mx first, as voting on the other rails will cause
  Mx changes to occur.
                .vreg_name = "vddmx",
                .quantize_to_corner = true,
                .supports_sw_mode = true,
                .maximum_for_automode = RAILWAY_NOMINAL,
                .vreg_type = RPM_SMPS_B_REQ,
                .vreg_num  = 1,

                .pm_rail_id = PM_RAILWAY_MX,
                .pmic_step_size = 12500,
                .initial_corner = RAILWAY_SUPER_TURBO,
                .default_uvs =
                {
                    0,          //RAILWAY_NO_REQUEST
                    675000,     //RAILWAY_RETENTION
                    950000,     //RAILWAY_SVS_KRAIT
                    950000,     //RAILWAY_SVS_SOC
                    950000,     //RAILWAY_NOMINAL
                    1050000,    //RAILWAY_TURBO
                    1050000,    //RAILWAY_SUPER_TURBO
                },
                .supported_corners = (railway_corner[])
                {
                  RAILWAY_RETENTION,
                  RAILWAY_SVS_SOC,
                  RAILWAY_NOMINAL,
                  RAILWAY_SUPER_TURBO,
                },
                .supported_corners_count = 4,
            }
```

6 Debugging RPM RBCPR

6.1 RBCPR status

RPM collects RBCPR information from the CPR hardware and stores it in rbcpr_stats:

- Fuse voltage, the CPR starting point
- For each mode, e.g., SVS/Nominal/Turbo, the number of interrupts in the mode
- Most current recommendations with timestamps
- Programmed voltage to railway
- Exception events – Recommended voltage hitting Min or Max

6.2 CPR debug

6.2.1 Enable/disable CPR at runtime

RBCPR is enabled by default during rpm_init() and can be disabled/enabled at runtime in rpm_ctl.c by using the DISABLE bit in RPM_CTL.

6.2.1.1 Enable/disable CPR with the DISABLE bit in RPM_CTL

To enable/disable CPR with the DISABLE bit in RPM_CTL:

- Set the bit − Disable CPR
- Clear the bit − Enable CPR

6.2.1.2 Enable/disable CPR in the Linux kernel

To enable/disable CPR in the Linux kernel:

- Disable CPR

```
adb shell “echo 8 > /sys/module/rpm_resources/mode/rpm_ctl”
```

- Enable CPR

```
adb shell “echo 0 > /sys/module/rpm_resources/mode/rpm_ctl”
```

6.2.2 Retrieve the RBCPR log

RBCPR statistics are used to collect information about the voltage scaling recommendations from the RBCPR hardware:

- Fuse voltage, the CPR starting point
- For each mode, e.g., SVS/Nominal/Turbo
  - Number of mode interrupts
  - Most current recommendations with timestamps
  - Programmed voltage to railway
- Exception events – Recommended voltage hitting Min or Max
- Mode and voltage of the last interrupt
- Ability to turn on/off statistics

RBCPR statistics are placed in a dedicated location of the RPM MSG RAM to make it always available for the HLOS to read and send through a diagnostic mechanism.

On the Android side, the debugfs can be mounted to read the RBCPR information.

```
mount -t debugfs none /sys/kernel/debug
cat /sys/kernel/debug/rpm_rbcpr
```

Example

```
:RBCPR Platform Data (upside_steps: 1)(downside_steps:2)(svs_voltage:
1050000)(nominal_voltage: 1162500)(turbo_voltage: 1287500)
:RBCPR Stats (status_counter: 8) (current_corner:
RBCPR_CORNER_TURBO)(current_timestamp: 0x2646943) (railway_voltage:1100000)
:       RBCPR Corner Data (name: RBCPR_CORNER_SVS) (efuse_adjustment: -
37500)(programmed_voltage: 912500(isr_counter:1)(min_counter: 0)(max_counter:0)
:               Voltage History[0] (voltage: 0) (timestamp: 0x0)
:               Voltage History[1] (voltage: 0) (timestamp: 0x0)
:               Voltage History[2] (voltage: 912500) (timestamp: 0x12b209)
:       RBCPR Corner Data (name: RBCPR_CORNER_NOMINAL) (efuse_adjustment: -
37500)(programmed_voltage: 1112500)(isr_counter: 4)(min_counter:
0)(max_counter:0)
:               Voltage History[0] (voltage: 1062500) (timestamp: 0x10ca11)
:               Voltage History[1] (voltage: 1087500) (timestamp: 0x10ca1c)
:               Voltage History[2] (voltage: 1112500) (timestamp: 0x10ca26)
:       RBCPR Corner Data (name: RBCPR_CORNER_TURBO) (efuse_adjustment: -
37500)(programmed_voltage: 1100000)(isr_counter: 3)(min_counter:
1)(max_counter:0)
:               Voltage History[0] (voltage: 1162500) (timestamp: 0x1d5ac)
:               Voltage History[1] (voltage: 1125000) (timestamp: 0x13fd6a)
:               Voltage History[2] (voltage: 1087500) (timestamp: 0x14170e
```

7 Debugging RPM Sleep

7.1 RPM sleep statistics

RPM Sleep Stats is defined as:

```
typedef struct sleep_stats_type
{
 uint32  stat_type;
 uint32  count;
 uint64  last_entered_at;
 uint64  last_exited_at;
 uint64  accumulated_duration;
 uint32  client_votes;
 uint32  reserved[3];
} sleep_stats_type;
```

Static sleep_stats_type* sleep_stats, the sleep_stats is an array of size 2:

```
sleep_stats[0] for CXO Shutdown
sleep_stats[1] for VDD Minimization
```

- Count field – Shows how many times RPM entered the associated sleep mode
- last_entered_at field – Shows the timestamp of when RPM entered the associated sleep mode
- last_exited_at field – Shows the timestamp of when RPM exited from the associated sleep mode
- Accumulated_duration field – Shows how long RPM has been in the associated sleep mode since it booted

7.2 Disabling RPM sleep

Set sleep_allow_low_power_modes = FALSE and build RPM. This disables RPM sleep, i.e., halt, xo_shutdown, and vdd_min.

7.3 Disabling PMIC watchdog

Set pmic_wdog_enable = 0 and build RPM. This prevents the PMIC watchdog from firing.