ACKNOWLEDGEMENT

By utilizing this website and/or documentation, I hereby acknowledge as follows:

Effective October 1, 2012, QUALCOMM Incorporated completed a corporate reorganization in which the assets of certain of its businesses and groups, as well as the stock of certain of its direct and indirect subsidiaries, were contributed to Qualcomm Technologies, Inc. (QTI), a wholly-owned subsidiary of QUALCOMM Incorporated that was created for purposes of the reorganization.

Qualcomm Technology Licensing (QTL), the Company's patent licensing business, continues to be operated by QUALCOMM Incorporated, which continues to own the vast majority of the Company's patent portfolio. Substantially all of the Company's products and services businesses, including QCT, as well as substantially all of the Company's engineering, research and development functions, are now operated by QTI and its direct and indirect subsidiaries[1]. Neither QTI nor any of its subsidiaries has any right, power or authority to grant any licenses or other rights under or to any patents owned by QUALCOMM Incorporated.

No use of this website and/or documentation, including but not limited to the downloading of any software, programs, manuals or other materials of any kind or nature whatsoever, and no purchase or use of any products or services, grants any licenses or other rights, of any kind or nature whatsoever, under or to any patents owned by QUALCOMM Incorporated or any of its subsidiaries. A separate patent license or other similar patent-related agreement from QUALCOMM Incorporated is needed to make, have made, use, sell, import and dispose of any products or services that would infringe any patent owned by QUALCOMM Incorporated in the absence of the grant by QUALCOMM Incorporated of a patent license or other applicable rights under such patent.

Any copyright notice referencing QUALCOMM Incorporated, Qualcomm Incorporated, QUALCOMM Inc., Qualcomm Inc., Qualcomm or similar designation, and which is associated with any of the products or services businesses or the engineering, research or development groups which are now operated by QTI and its direct and indirect subsidiaries, should properly reference, and shall be read to reference, QTI.

[1] The products and services businesses, and the engineering, research and development groups, which are now operated by QTI and its subsidiaries include, but are not limited to, QCT, Qualcomm Mobile & Computing (QMC), Qualcomm Atheros (QCA), Qualcomm Internet Services (QIS), Qualcomm Government Technologies (QGOV), Corporate Research & Development, Qualcomm Corporate Engineering Services (QCES), Office of the Chief Technology Officer (OCTO), Office of the Chief Scientist (OCS), Corporate Technical Advisory Group, Global Market Development (GMD), Global Business Operations (GBO), Qualcomm Ventures, Qualcomm Life (QLife), Quest, Qualcomm Labs (QLabs), Snaptracs/QCS, Firethorn, Qualcomm MEMS Technologies (QMT), Pixtronix, Qualcomm Innovation Center (QuIC), Qualcomm iSkoot, Qualcomm Poole and Xiam.

Resource Power Manager

User Guide

80-N6955-1 D

July 25, 2012

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

Qualcomm Confidential and Proprietary

QUALCOMM Incorporated
5775 Morehouse Drive
San Diego, CA 92121-1714
U.S.A.

Contents

Figures

Tables

Revision history

Revision	Date	Description
A	Aug 2011	Initial release
B	Dec 2011	Added reference documents (Table 1-1) and Chapter 5
C	May 2012	Added APQ8064, Linux build, and scheduling details; updates throughout
D	Jul 2012	Added RPM BRCPR, updated build RPM and Table 7-1 RPM address settings

1 Introduction

1.1 Purpose

This document describes the Resource Power Manager (RPM) and how to debug it. The information in this documentation applies to the following chipsets:

- MSM8660
- MSM8960
- MDM9x15
- AQP8064
- MSM8930

1.2 Scope

This document is intended for individuals who must understand the detailed steps regarding blowing the efuse.

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>`.

If you are viewing this document using a color monitor, or if you print this document to a color printer, **red boldface** indicates code that is to be **added**, and ~~blue strikethrough~~ indicates code that is to be replaced or removed.

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

1.4 References

Reference documents, which may include Qualcomm, standards, and resource 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		
Q1	*Application Note: Software Glossary for Customers*	CL93-V3077-1
Q2	*Presentation: MSM8960 Internal Chip Bus (ICB) Driver Architecture*	80-N8454-1
Q3	*Presentation:MSM8960 RPM Scheduling Details*	80-N8454-2

1.5 Technical assistance

For assistance or clarification on information in this guide, submit a case to Qualcomm CDMA Technologies 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 Main hardware overview

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

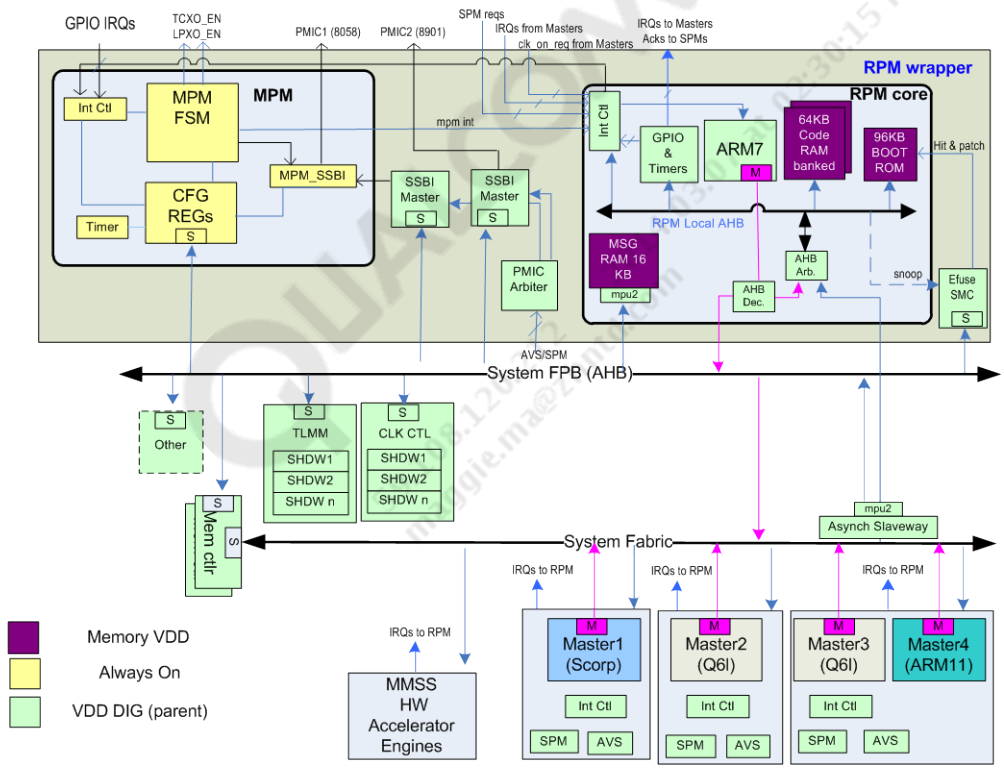


Figure 2-1 RPM hardware top-level block diagram

2.1.1 Processor

The RPM uses an ARM7™ TDMI core as its processor. The ARM7 TDMI is a RISC CPU with 32-bit ARM® and 16-bit thumb instruction sets. For debugging capabilities, JTAG support is included in this core.

2.1.2 Advanced High-performance Bus (AHB)

The RPM block instantiates the system Fast Peripheral Bus (FPB) core to be used as the RPM local AHB. The system FPB core is configured to have two masters and six slaves.

The possible masters are:

- RPM processor
- Test Interface Controller (TIC)
- JTAG2 AHB
- An external master

Out of these four possible masters, the RPM processor, TIC, and JTAG2 AHB are mutually exclusive, i.e., only one of these can be a master at any given time.

The RPM AHB slaves are:

- RPM boot ROM
- RPM code RAM
- RPM interrupt controller
- RPM timers
- RPM Control/Status Registers (CSRs)
- System FPB

There is also a default slave that houses the RPM AHB register interface.

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 Flash. The PBL must reside in an internal boot ROM to guarantee that the chip always boots from a known trusted code.

The RPM currently reserves the first 128 KB in its memory space for boot ROM. The boot ROM resides at the physical address 0x0, since the ARM7 TDMI core used by the RPM can only boot from this address.

2.1.4 Code RAM

The RPM code RAM provides storage for the RPM's code image as well as TCM data. On initial bootup, the PBL in the RPM boot ROM downloads the RPM code image from the external boot device into code RAM. Once the download is complete, the ARM7 starts executing this code image. In addition to the RPM's code image, the RAM also houses the primary warm boot handler, which will be used if power collapse of the primary (C1) power rail is implemented.

2.1.5 Interrupt controller

The RPM interrupt controller block is a wrapper around the QGIC core that it instantiates. The RPM interrupt controller block configures the QGIC to have 64 Shared Peripheral Interrupts (SPIs), five Private Peripheral Interrupts (PPIs), and one Software Triggered Interrupt (STI). All SPIs are connected to the interrupt input ports on the RPM core interface. Three of the PPIs are tied to general purpose timer interrupts, while the fourth PPI is connected to the watchdog (WDOG) bark interrupt. The QGIC requires a minimum of one STI; at this time there are no specific plans for the STI on the RPM.

The RPM interrupt controller also sends out a nonclocked interrupt detection signal, rpm_wakeup_irq.

2.1.6 Timers

The RPM timers block instantiates three general purpose and one WDOG timer. Two of the three general purpose timers are configured to be fast clock timers, i.e., their clock source is either PXO or CXO. The third general purpose timer is configured to be a sleep clock timer. The WDOG timer runs on this sleep clock.

The sleep clock timer provides a coarse resolution timer. A sleep clock (~32 kHz) is supplied as the timer's source, providing a theoretical range from 0.03 ms to approximately 37 hrs.

The general purpose timers are used by the RPM to provide a regular/hardware time base that can be used for dynamic performance monitoring, temperature monitoring, debug, and any other general use purposes, such as delayed power events, etc. The timer can be polled and/or provide an interrupt pulse to the processor via a PPI to RPM_INT_CTRL. The counter is a 32 bit up counter.

The clock source for the two fast clock timers is either PXO or CXO, providing for a more fine-grained timer than the sleep clock timer. For the general purpose timers the signal from PXO or CXO can be further divided by 1, 2, 3, or 4 to provide a range from approximately 50 µs to 3.6 min.

2.1.7 CSR

The RPM CSR block holds all of the RPM control and status registers. These include general purpose control and status registers, as well as the RPM timer block control and status registers.

The RPM SW_DONE register implements a Software Wait For Interrupt (SWFI) functionality for the RPM ARM7 processor. The software can write to this register to indicate that RPM ARM7 is done with its software processing and it is now safe to initiate Sleep mode.

The RPM CLK_OFF register provides the software with a means to turn off the clocks to the various RPM blocks. This register can be used to generate Clock Off requests for gating off the RPM processor, bus, timer0, timer1, timer2, and WDOG timer clocks, as well as the clock to the RPM master interface on the system FPB.

The RPM provides a configurable number of General Purpose Outputs (GPOs) that can be used as interrupt outputs as well as general status pulses. The number of GPOs is determined by setting the NUM_GPO_GROUPS_GENERIC. For the A-family series of chips this is instantiated as 32, but not all 32 are always connected. The number of connected GPOs varies based on the number of execution environments and the debug system. For example, chipsets that contain the QDSS subsystem have more GPOs connected than those that do not, as the QDSS subsystem adds additional debug signals that are asserted to the debugger.

2.1.8 RPM secondary components

2.1.8.1 Message RAM

The RPM message RAM provides memory for sending messages to and from the RPM core. The messaging masters use this memory to communicate with the RPM.

The number of messaging masters varies depending on what chipset for which the RPM software is generated.

Table 2-1 identifies the messaging masters for each of the chips the RPM supports.

Table 2-1 RPM messaging masters for A-family chipsets

Chipset	Messaging masters
MSM8660	▪ D-Scorpion ▪ LPASS ▪ Modem – ARM11™
MSM8960	▪ D-Krait ▪ WCNSS ▪ LPASS ▪ Modem – Hexagon™ SPS
MSM9x15	▪ Sparrow ▪ LPASS ▪ Modem – Hexagon
APQ8064	▪ Kraits ▪ GNSS (disabled for Fusion configuration) ▪ WCNSS ▪ Sensors ARM7 ▪ LPASS

2.1.8.2 Multiprocessor power manager

The MPM block provides a hardware mechanism for putting the system core resources in a low power state when the chip is idle as well as restoring these resources when a wakeup event is detected.

The MPM controls the following system core resources:

- PXO
- CXO
- VDD_Dig Sleep mode
- VDD_Mem Sleep mode

The RPM is the only subsystem capable of controlling the MPM.

2.1.9 RPM input components

2.1.9.1 SPM

The SPM is a subsystem power manager. Its purpose is to manage local power-related resources for a subsystem. For subsystems that have an independent power rail, the SPMs allow a subsystem to enter a Power Collapse and Restore state, independently of any processor outside of the subsystem. For subsystems that do not have an independent power rail, the SPMs support the capability to turn off the Ground to Domain Foot Switch (GDFS) or to turn off the Ground to Domain Head Switch (GDHS), then to restore the state of the subsystem independently of any processor outside of the subsystem.

To provide the capability to maximize the power savings when a master is entering a low power state, the master may notify the RPM that a low power state is being entered and then have its SPM generate an interrupt to the RPM when the master has fully entered the low power state. This allows the RPM to efficiently control the systemwide shared resources needed by the subsystem while the subsystem is in a low power state. The state of the systemwide resources would then be restored by the RPM, before the RPM would remove the subsystem to exit the low power state.

For masters that consist of more than one core, each core is supported by an SPM. This enables the RPM to collect the interrupt signals from the SPM for each core independently and to correctly determine when the master has completely entered a low power state.

A master can prevent the modification of the system wide resources when entering a low power state by configuring the SPM or SPMs to not generate an interrupt to the RPM.

2.1.10 Output components

The hardware blocks that are controlled by the RPM can vary from chipset to chipset. Table 2-2 identifies the specific output hardware block variations for the A-family.

Table 2-2 RPM components for A-family chipsets

Output Component	MSM8960	MSM9x15	MSM8660	APQ8064
Power Management Interface Controller (PMIC)	PM8921	PM8018	▪ PM8058 ▪ PM8901	▪ PM8921 ▪ PM8821
Flexible Advanced Buses and Reusable Interconnect Cores (FABRICs)	Baseline (AFAB+SFAB+ MMFAB)	▪ No AFAB ▪ No MMFAB ▪ Lower freq for SFAB ▪ Hardware freq scaling	Baseline (AFAB+SFAB+ MMFAB)	Baseline (AFAB+SFAB+ MMFAB)
Memory	LPDDR2 (400 MHz)	LPDDR1	LPDDR2 (333 MHz)	▪ LPDDR2 (533 MHz) ▪ PCDDR3 (533 MHz)
Reference clock	▪ PXO ▪ CXO	CXO	▪ PXO ▪ CXO	▪ PXO ▪ CXO

2.1.10.1 Clock controller

Clock controller is the hardware block that controls the clock distribution for the majority of the system.

The RPM controls the system wide resource frequencies through the clock controller. This will affect the frequency of:

- Bus and FABRIC clocks
- DMOV clock
- DDR clock

2.1.10.2 PMIC

The PMIC is the external hardware that controls the voltage rails, the battery, the charger, the Real-Time Clock (RTC), the Housekeeping Analog to Digital Conversion (HKADC), and other miscellaneous components of the hardware. The PMIC is accessed using the Single-wire Serial Bus Interface (SSBI) protocol.

The RPM controls all the shared voltage regulators in the PMIC.

2.1.10.3 Bus arbitration

The bus arbiter is the hardware block that controls the arbitration settings for some portions for the bus FABRICs. The RPM controls all the arbitrations settings in the system.

2.1.10.4 Memory controller

The memory controller is the hardware block that controls the external memory.

The RPM controls interactions with the memory controller and the necessary voltages, and the entry and exit of memory self-refresh.

2.2 RPM software overview

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

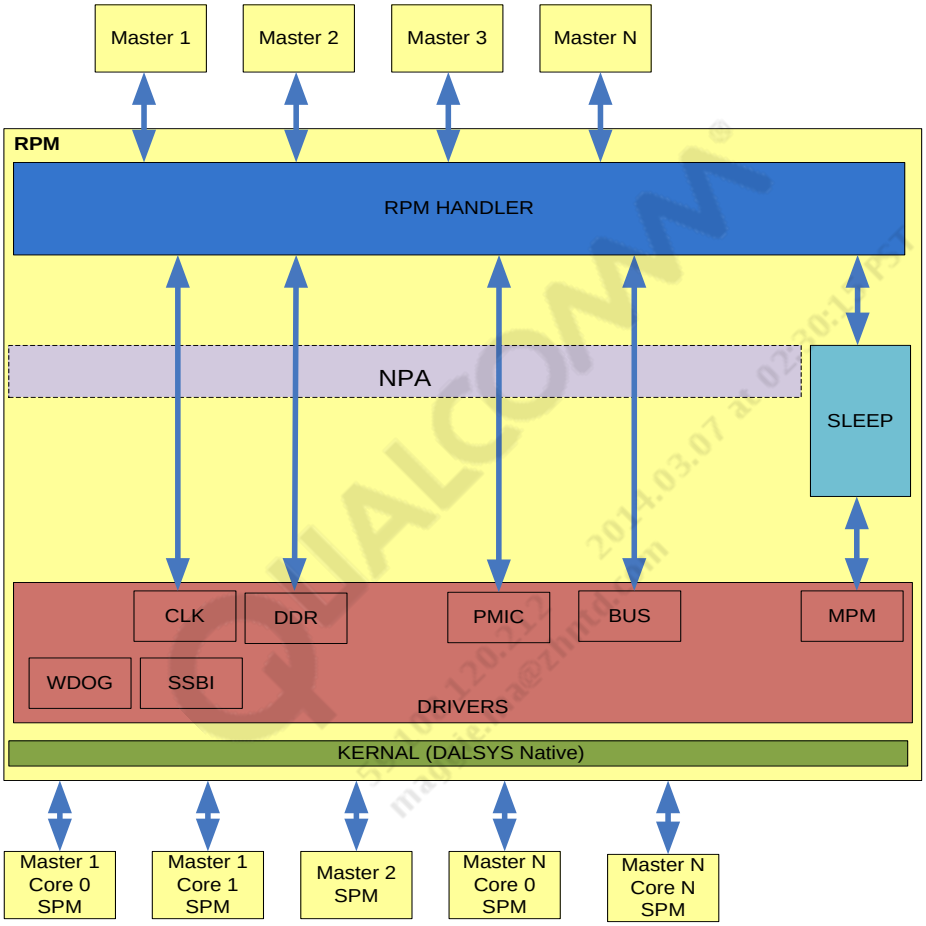


Figure 2-2 RPM software topology

2.2.1 Kernel

The kernel for the RPM will support:

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

The kernel will be implemented using DalSys to the metal. This provides RPM with a lightweight kernel and allows for 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 of the resources supported by the RPM will register with the RPM handler to request notification when requests are received for the resource which the driver controls. Upon receiving this notification, the drivers will perform any arbitration that must be performed between the new request and previous requests from other masters. Based on the results of the arbitration, the driver determines how to modify the hardware resource.

2.2.3.1 NPA

A driver may use the NPA to represent the 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 workloads 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 of the masters and the other part resides on the RPM.

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

2.2.3.3 Bus arbitration driver

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

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

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 of the masters and the other part resides on the RPM.

The RPM PMIC driver directly aggregates requests from each of the masters for any of 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 allowing software to attempt to recover. If the register is still not written within a specific time period, the system will reset.

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

The WDOG also supports a freeze, stopping the countdown, for scenarios where the scheduler will not run within a given timeframe. Examples of scenarios requiring freeze support include long memory or hardware accesses. Such scenarios should be avoided in most cases by software design. Freeze is supported when the RPM goes into Sleep mode.

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 do the following:

- Vdd_Dig retention – Put the systemwide power rail in Retention state
- Vdd_Dig collapse – Put the systemwide power rail in Collapse state
- Vdd_Mem retention – Put the systemwide power rail in Retention state
- Vdd_Mem collapse – Put the systemwide power rail in Collapse state
- PXO shutdown – Turn off PXO
- CXO shutdown – Turn off CXO

The driver must also support programming the MPM timer hardware, as well as the MPM interrupt controller.

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

2.2.3.7 RPM message protocol driver

The RPM message protocol driver abstracts the RPM message protocol away from other subsystem software. The driver handles the RPM-side client and server portions of the RPM messaging. Each of the resource drivers registers with the RPM message protocol driver at initialization time. When a resource request is received by the RPM, it is passed to the registered resource driver for processing.

3 RPM Build Instructions

3.1 Tools preparation

To make the RPM build, the licensee must have the tools described here.

Python 2.x

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

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

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

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

After installing, 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:

1. Start → Run → cmd.exe.
2. At prompt, run **python**.
   a. If you see junk followed by a >>> prompt, Python has installed correctly. Press Ctrl+Z followed by Enter to exit.
   b. If you see an error, 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 latest at the time of this document release.

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

Make note of what follows “engine:”. If an error occurs when running the version check, 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.

RVDS

RVDS 4.0 is required for theMSM8660 RPM build and RVDS 4.1 Build 514 or earlier is required for the MSM8960 and MDM9x15 RPM builds.

To check installation, verify the existence of a folder at c:\apps\RVDS41 or c:\apps\RVCT41.

3.2 Build process

3.2.1 RPM build

NOTE: Numerous changes were made in this section.

To build RPM in a Windows environment:

1. cd <path_to_rpm_source>\build.
2. Execute rvct40.bat to set up the build environment.
3. Run build.bat.

   The RPM image (RPM.elf) and final image (RPM_hash_nonsec.mbn) is generated in the build\rpm\<Target_plaform>\build.
4. To make clean, run the following command in the build directory:

   build –c

To build RPM in a Linux environment:

1. Set up the build environment.
   a. Have the following in your path. The values will change depending on the environment

      ARMLMD_LICENSE_FILE=8224@redcloud:7117@license-wan-arm1

      PYTHONPATH=/usr/bin/python2.6
   b. Change the shell to bash using : chsh and choose bash as your shell
   c. Make the following changes:

      fromdos –f /rpm_proc/build/build.sh

      fromdos –f /rpm_proc/build/build_8960.sh

      fromdos –f /rpm_proc/build/build_common.py

      fromdos –f /rpm_proc/tools/build/scons/SCons/scons.sh

      fromdos –f /rpm_proc/core/boot/ddr/build/msm8960.sconscript
2. Run ./build.sh in the build directory.

3.2.2 RPM secure build

To build a secure RPM:

1. To enable standard CSMS signing, uncomment the following line from build\rpm\<MSMplatform>\build\Sconscript.

   #env.Replace(USES_SECBOOT = 'yes')
2. To enable QDST signing, ensure that you have openSSL on your path locally, and then uncomment the following two lines from build\rpm\8960\build\SConscript:

   #env.Replace(USES_SECBOOT = 'yes')

   #env.Replace(USES_SECBOOT_QDST = 'yes')

3.2.3 General guidelines

When using env.AddLibsToImage or env.AddLibrary, the image tag that RPM publishes is "RPM_IMAGE".

RPM.map (in the build products directory) can be very useful for profiling the size that code is consuming, however, it does not account for heap usage.

When the RPM image size is exceeded, the ARM linker often prints an error message, e.g., section RPM_STACK overlaps section RPM_ZI. This means that code size must be reduced before the image will build successfully.

3.2.4 Known build issues

A known build issue is that a newer python version may no longer support one of the flags in the build. This should be fixed on future releases. The immediate fix for the issue is to comment out the line that has the error.

File: C:\RPM\core\bsp\build\scripts\scons_mod.py

Line: 332

Change description: Add the code shown in **red boldface**.

```
#startupinfo.dwFlags |= subprocess.STARTF_USESHOWWINDOW
```

4 RPM Message

4.1 RPM message infrastructure

RPM messaging is built around three core concepts:

- Masters – These are the subsystems that make requests to the RPM; e.g., on MSM8660, there are three masters:
  - Apps
  - Modem
  - LPASS Hexagon
- Resources – These are the physical components for which the RPM is responsible for controlling, i.e., clocks, voltage rails, etc. Every resource is able to receive independent requests from more than one source.
- Configuration sets (or "sets") – These are a collection of settings (one for each resource) that all fit a logical use case. At the moment, two sets are provided for each master:
  - Active set – This set contains all the settings used when the subsystem is awake.
  - Sleep set – This set contains all the settings used when the subsystem is sleeping (when an SPM handshake occurs).

  The active set is also considered the primary set.

  Every master has two sets that are stored in the RPM. At any given time one of these sets is operating as the selected set.
  - When a set becomes selected, the requests from that set are sent to the drivers for each respective resource.
  - When the selected set does not have a request for a particular resource, the effective request for that master is inherited from the primary set.

  Transitions are made between sets based on triggers. The RPM currently supports two kinds of triggers:
  - SPM handshakes
    - Shutdown handshakes automatically trigger a change to the subsystem's sleep set.
    - Bringup handshakes automatically trigger a change to the subsystem's active set.

- Timers
  - A timed trigger can be set up to cause a set change between two sets after being in the selected set for a certain period of time.
  - The most-used timed trigger is a wakeup timer, i.e., a timed trigger in the sleep set that returns to the subsystem's active set after some time has elapsed.
  - Timed triggers from the sleep set always generate a wakeup interrupt.

RPM messages are designed to fulfill this transaction. RPM message content includes:

- Resource to be modified
- Requested setting for resource
- Set for which the request is designated

RPM sends an interrupt back to the master once the request has been processed. Masters should not use the resource until a response has been received. Each message can support more than one resource request, e.g., when updated, the sleep set can open a transaction, call multiple drivers to allow them to fill in their required sleep settings, then end the transaction. Ending the transaction sends the message to the RPM.

4.2 Aggregation across masters

As described in the previous section, either active set or sleep set can be selected. In the example in Table 4-1, the active set is selected for the master. The sets could include clock settings, regulator settings, and oscillator settings.

Table 4-1 Configuration sets

Resource	Active set	Sleep set	Effective request (active set)
Clock A	200 MHz	0 MHz	200 MHz
Regulator B	1100 mV	[No request]	1100 mV
Oscillator C	[No request]	[No request]	Driver default (generally off)

Once more than one master requests the shared resource, RPM will aggregate the request and set the shared resource to the correct settings, as shown in Figure 4-1.

In Figure 4-1, * denotes the selected set, therefore, the apps-selected set is an active set and requests the 266 MHz FABRIC clock, the modem-selected set is an active set and requests the 50 MHz FABRIC clock, the LPASS-selected set is a sleep set and requests the 27 MHz FABRIC clock. RPM aggregates all the requests and sets the FABRIC clock to 266 MHz.

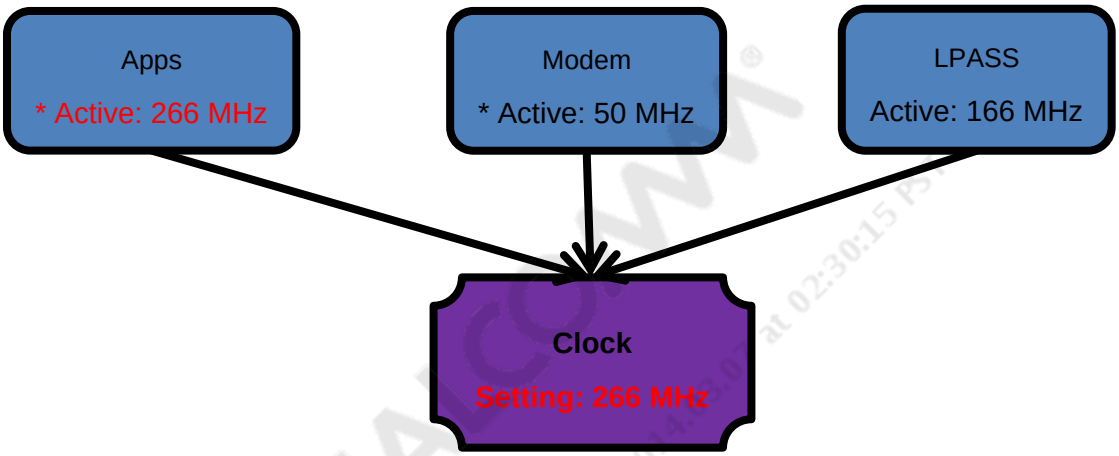


Figure 4-1 Aggregation cross masters for active sets

Once apps- and modem-selected clocks are changed to sleep set, the RPM will aggregate again and set the clocks to the maximum request, i.e., the LPASS sleep clock sets to 27 MHz. See Figure 4-2 for details.

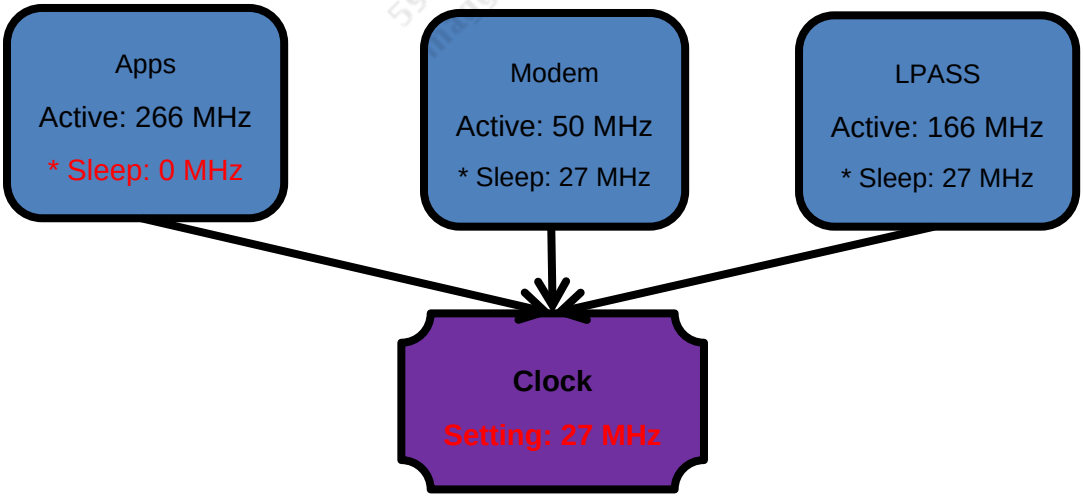


Figure 4-2 Aggregation cross masters for sleep sets

4.3 RPM message RAM

Messaging is the main method for each of the masters in the system to communicate with the RPM. The message RAM is partitioned into several regions, using an RPU with 512-byte granularity. A majority of the partitions are dedicated for master/RPM pairs. These partitions are only accessible by that master and the RPM. See Figure 4-3.

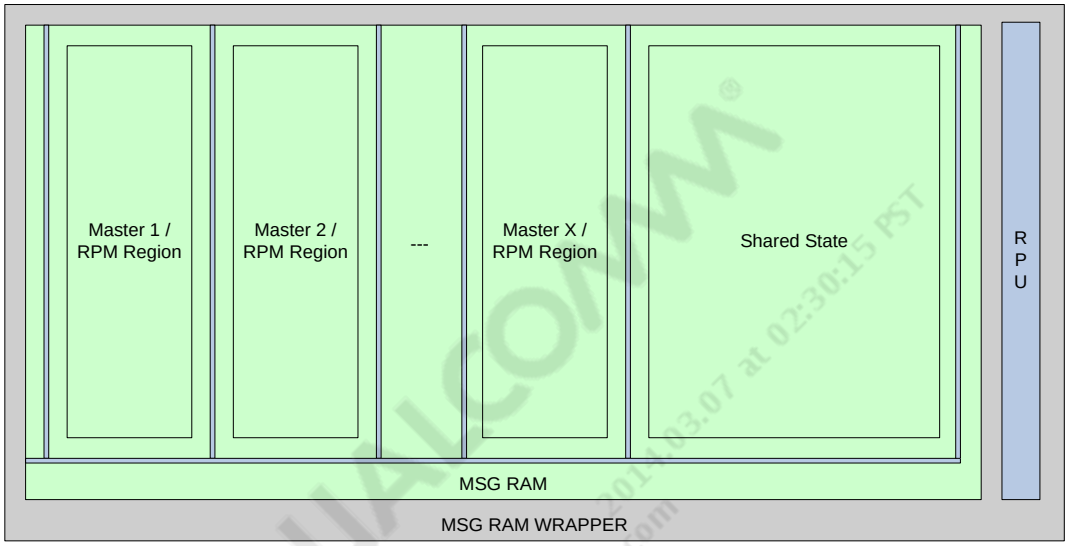


Figure 4-3 RPM message RAM partition

4.3.1 RPM message RAM layout

The message RAM layout determines what areas in the request, acknowledge, and status regions of the message RAM are dedicated to a specific resource.

The mapping is in core\power\rpm\dal\target\<MSMplatform>\RPMMessageRAMLayout.h.

Each resource needs an area allocated in:

- The RPMResourceDataLayout structure – This determines where in a processor's request section the resource goes, and how large of a data buffer is used when requesting to that resource.
- The RPMStatusLayout structure – This determines where a resource's status goes in the RPM's system status section, and how large the status of that resource should be.

  It is important that the overall size of these structures is preserved.
- After any modifications, these structures should remain 1024 bytes in size.
- This may require adding or removing padding with reserved fields in order to preserve the size.
- If size is not preserved, the RPU used to enforce security will not work correctly.

4.3.2 RPM message RAM interface file

The RPM message interface file provides utility functions to allow the RPM code to read and write message RAM in a target-independent manner. The file contains functions that enumerate certain characteristics of the message RAM layout. The file is located at core\power\rpm\dal\target\<MSMplatform>\RPMMessageRAM.c.

There are four regions in this file that must be modified:

- The array ResourceOffsetTable must contain an entry for each possible value in the DAL_rpm_ResourceType enumeration.
  - The entry represents the offset into the request segment that the resource resides at, and is generated from the structure layout.
- The array StatusOffsetTable must contain an entry for each possible value in the DAL_rpm_ResourceType enumeration.
  - The entry represents the offset into the status segment that the resource resides at, and is generated from the structure layout.
  - If a resource does not publish a status, its entry in the array should be 0xFFFF to indicate that there is no valid offset to status for this resource.
- The function RPM_GetResourceSize must be updated to return the correct size of the new resource.
  - The ResourceOffsetTable has a lot of repeated values in it, and therefore is implemented as a switch statement.
  - Using SIZEOF on the structure definition is only required if another appropriate entry does not already exist.
  - In general, any resource of 4 or 8 bytes can be added to the already-existing large blocks of case statements that resolve to those sizes.
- The function RPM_GetStatusSize must be updated to return the correct size of the new resource.
  - The guidelines for the ResourceOffsetTable are the same as with RPM_GetResourceSize.
  - In many cases, status fields are the same size as the resource requests, but the framework does allow them to be of different sizes.
  - The meaning of status fields can be hard to interpret if they are not the same as the requests.

4.4 RPM message example

When resource requirements for a subsystem change, the subsystem must notify the RPM of the change. This notification is accomplished by the subsystem master writing a message into the message RAM and triggering an interrupt to the RPM. Once the RPM receives the interrupt, the RPM reads the message RAM to determine which resource is requested to be changed and to which performance level. The RPM then defines the appropriate performance level for the requested resource and controls the change sequence to update the required resource performance level. Once completed, the RPM indicates the request has been acted upon by writing a message into the message RAM and triggering an interrupt to the subsystem master.

4.4.1 Single resource request

The steps for changing the current requirements on a resource are shown below using the system FABRIC as an example:

Master

1. Write System_Fabric_Clk_Req – Performance Level 3.
2. Write Resource_Ind_Req – System FABRIC.
3. Write Change_Req – Awake set.
4. Trigger M2RPM_0 interrupt.

RPM

1. Receive M2RPM_0 interrupt.
2. Read Change_Req – Awake set.
3. Read Resource_Ind_Req – System FABRIC.
4. Parse Change_Req and Resource_Ind_Req.
5. Read System_Fabric_Clk_Req – Performance Level 3.
6. Clear Resource_Ind_Req.
7. Clear Change_Req.
8. Parse and apply change to system FABRIC.
9. Write System_Fabric_Clk_Ack – Performance Level 3.
10. Write System_Fabric_Clk_Status – Performance Level 4.
11. Write Resource_Ind_Ack – System FABRIC.
12. Write Change_Ack – Awake set.
13. Trigger RPM2M_0 interrupt.

Master

1. Receive RPM2M_0 interrupt.
2. Read Change_Ack – Awake set.
3. Read Resource_Ind_Ack – System FABRIC.
4. Parse Change_Ack and Resource_Ind_Ack.
5. Read System_Fabric_Clk_Ack - Performance Level 3.
6. Read System_Fabric_Clk_Status - Performance Level 4 (optional).
7. Parse and apply acknowledgment.
8. Clear Resource_Ind_Ack.
9. Clear Change_Ack.

Figure 4-4 illustrates this system FABRIC flow.

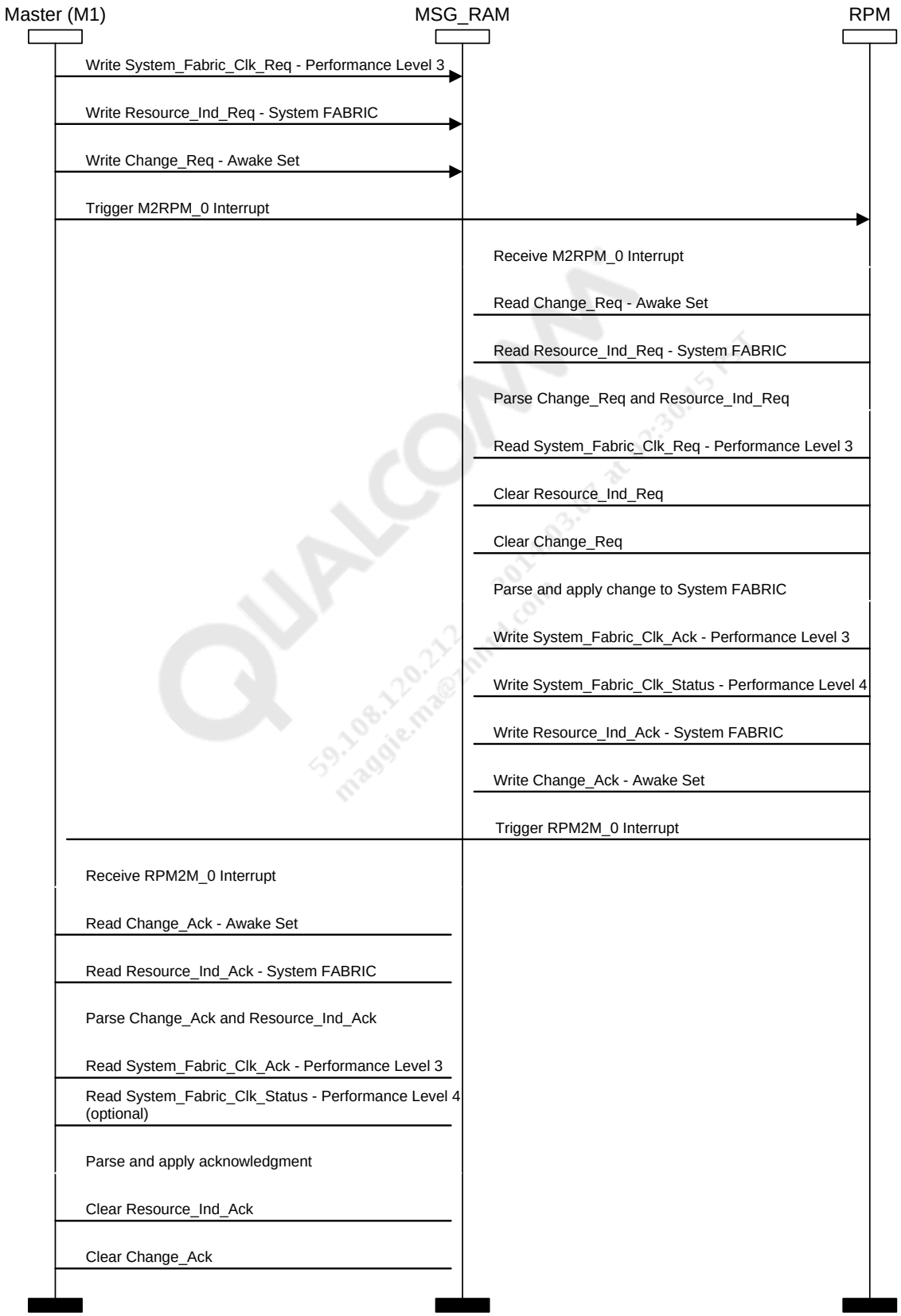


Figure 4-4 Message flow example

4.4.2 Multiple resource request

The steps for changing the current requirements on several resources are shown below. In this example, the request is for changing the system FABRIC, the apps FABRIC, and Vdd_Dig.

Master

1. Write System_Fabric_Clk_Req – Performance Level 3.
2. Write Apps_Fabric_Clk_Req – Performance Level 1.
3. Write Vdd_Dig_Req – Voltage Level 4.
4. Write Resource_Ind_Req – System FABRIC, apps FABRIC, Vdd_Dig.
5. Write Change_Req - Awake set.
6. Trigger M2RPM_0 interrupt.

RPM

1. Receive M2RPM_0 interrupt.
2. Read Change_Req – Awake set.
3. Read Resource_Ind_Req – System FABRIC, apps FABRIC, Vdd_Dig.
4. Parse Change_Req and Resource_Ind_Req.
5. Read System_Fabric_Clk_Req – Performance Level 3.
6. Clear Resource_Ind_Req – System FABRIC.
7. Parse and apply change to system FABRIC.
8. Write System_Fabric_Clk_Ack – Performance Level 3.
9. Write System_Fabric_Clk_Status – Performance Level 4.
10. Write Resource_Ind_Ack – System FABRIC.
11. Check for higher priority message.
12. Parse Change_Req and Resource_Ind_Req.
13. Read Apps_Fabric_Clk_Req – Performance Level 1.
14. Clear Resource_Ind_Req – Apps FABRIC.
15. Parse and apply change to Apps FABRIC.
16. Write Apps_Fabric_Clk_Ack – Performance Level 1.
17. Write Apps_Fabric_Clk_Status – Performance Level 1.
18. Write Resource_Ind_Ack – Apps FABRIC.
19. Check for higher priority message.
20. Parse Change_Req and Resource_Ind_Req.
21. Read Vdd_Dig_Req – Voltage Level 4.
22. Clear Resource_Ind_Req – Vdd_Dig.
23. Clear Change_Req.

24. Parse and apply change to Vdd_Dig.
25. Write Vdd_Dig_Ack – Voltage Level 4.
26. Write Vdd_Dig_Status – Voltage Level 4.
27. Write Resource_Ind_Ack – Vdd_Dig.
28. Write Change_Ack – Awake set.
29. Trigger RPM2M_0 interrupt.

Master

1. Receive RPM2M_0 interrupt.
2. Read Change_Ack – Awake set.
3. Read Resource_Ind_Ack – System FABRIC, apps FABRIC, Vdd_Dig.
4. Parse Change_Ack and Resource_Ind_Ack.
5. Read System_Fabric_Clk_Ack – Performance Level 3.
6. Read System_Fabric_Clk_Status – Performance Level 4 (optional).

NOTE: Reading the status registers is optional; this is provided for the master in case the information is needed.

7. Read Apps_Fabric_Clk_Ack – Performance Level 1.
8. Read Apps_Fabric_Clk_Status – Performance Level 1 (optional).

NOTE: Reading the status registers is optional; this is provided for the master in case the information is needed.

9. Read Vdd_Dig_Ack – Voltage Level 4.
10. Read Vdd_Dig_Status – Voltage Level 4 (optional).

NOTE: Reading the status registers is optional; this is provided for the master in case the information is needed.

11. Parse and apply acknowledgment.
12. Clear Resource_Ind_Ack.
13. Clear Change_Ack.

Figure 4-5 and Figure 4-6 illustrate the multiresource flow.

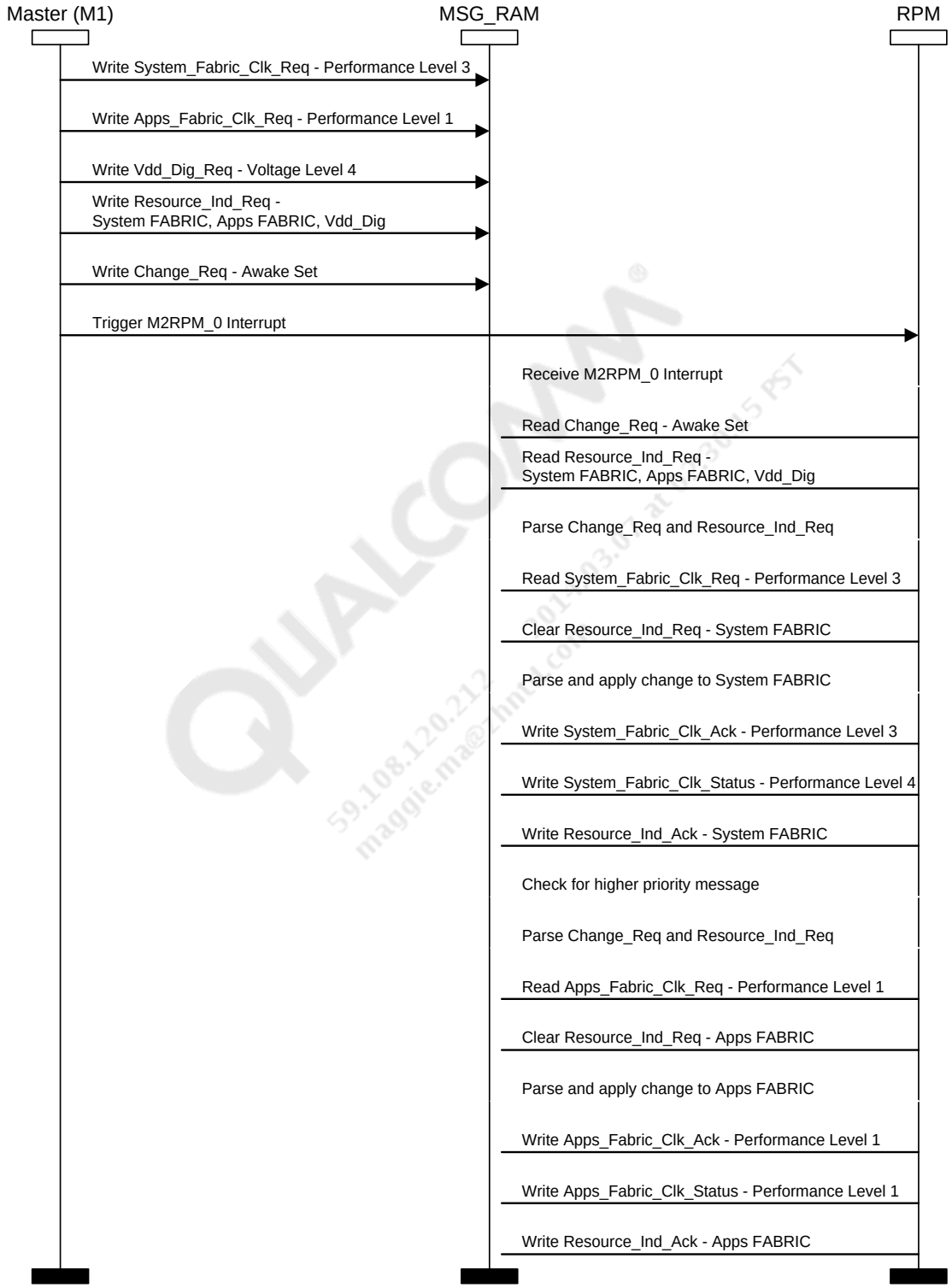


Figure 4-5 Multiresource message flow (1 of 2)

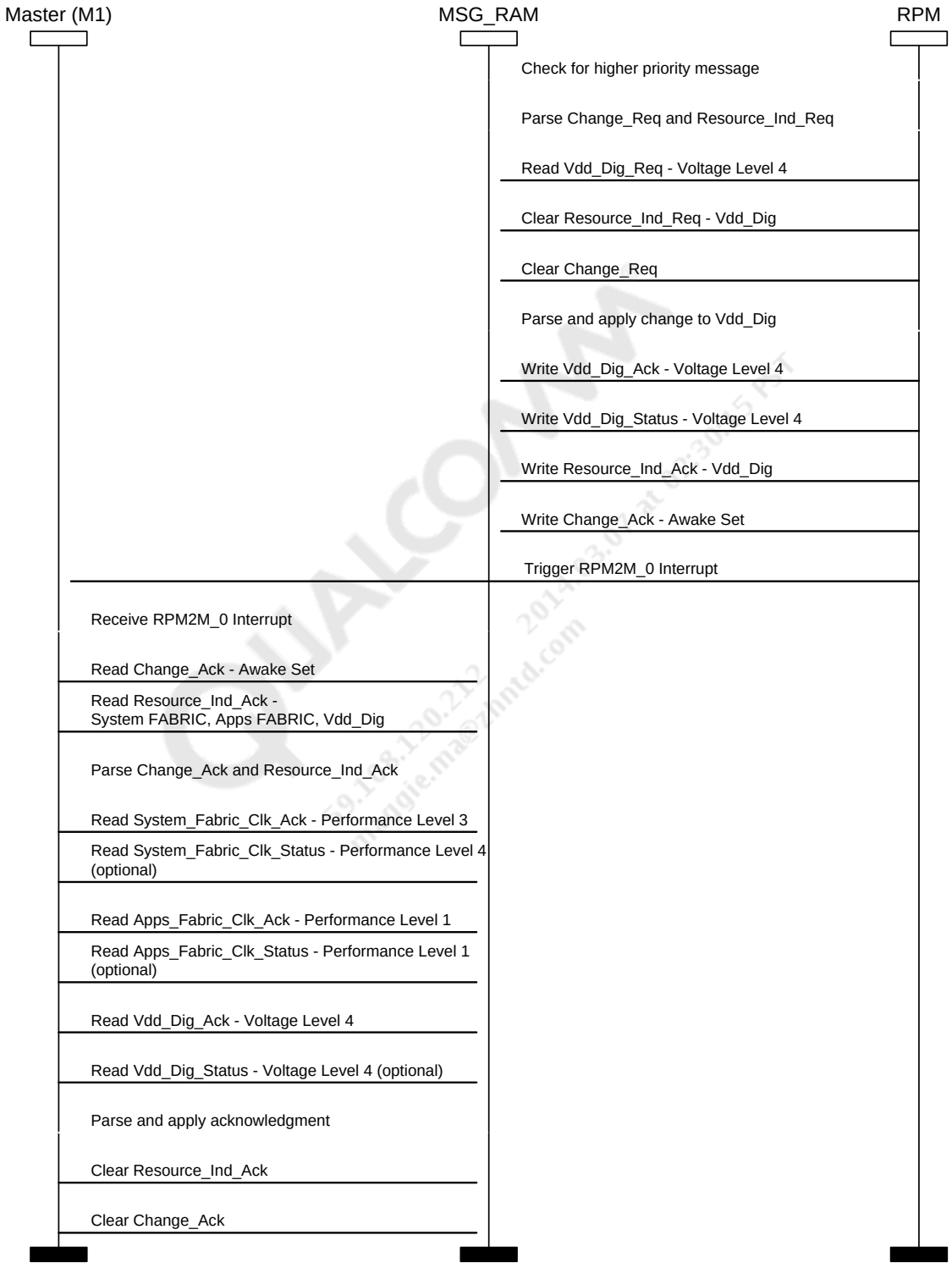


Figure 4-6 Multiresource message flow (2 of 2)

4.5 RPM message driver implementation

The RPM messaging protocol is split into two different DAL drivers:

- DalRPM is the messaging client that runs on master processors.
- DalRPMFW is the messaging server that runs in the RPM firmware.

4.5.1 DALRPM

DalRPM's interface is in core\api\power\DDIRPM.h. The driver is a DAL, which means the first call that must be made to it is DAL_RPMDeviceAttach, which returns a handle to the device. This handle can be thought of as the context when talking to the device; the handle is passed as the first argument of all other driver functions.

DalRPM has two main purposes:

- Sending requests to the RPM; related APIs are:
  - DalRPM_SendRequest
  - DalRPM_CreateMessage
  - DalRPM_AppendRequest
  - DalRPM_SendMessage
  - DalRPM_DestroyMessage

  The target of these requests, i.e., which set they go to, is controlled by DalRPM_ConfigureSet.
- Reading status from the RPM
  - DalRPM_GetStatus

4.5.1.1 DalRPM_SendRequest

This function immediately sends a message to the RPM with a request for a single resource. By default, the function does not return until a response has been received from the RPM.

Prototype

```
DALResult DalRPM_SendRequest(DalDeviceHandle * _h,
                             DAL_rpm_ResourceType Resource,
                             uint32  DataLen,
                             void * Data)
```

→	h	Handle acquired from DAL_RPMDeviceAttach
→	Resource	Resource ID of the resource at which the request is directed
→	DataLen	Length of the resource data buffer to send
→	Data	A pointer to the resource data buffer

Description

The function does not busy wait. Once it has sent its request, it waits on a DALEvent triggered from the ISR invoked by the RPM response interrupt.

Return value

This function returns an error if the resource is unsupported by the RPM, the driver on the RPM rejected the request, or if the data length does not match expectations.

4.5.1.2 DalRPM_ConfigureSet

This function modifies the “context” associated with the handle to the RPM driver.

Prototype

```
DALResult DalRPM_ConfigureSet(DalDeviceHandle * _h,
                              DAL_rpm_ConfigSetType  ConfigSet)
```

→	h	Handle acquired from DAL_RPMDeviceAttach
→	ConfigSet	ID of the configuration set where requests should be directed

Description

After modifying the context associated with the handle passed to this function, all requests sent with the same handle will be directed at the appropriate set.

The licensee can call ConfigureSet any number of times to retarget back to the original set.

4.5.1.3 DalRPM_CreateMessage

The RPM can process requests to multiple resources in a single message. This API creates a message that can have multiple requests added to it before sending.

Prototype

```
DALResult DalRPM_CreateMessage(DalDeviceHandle * _h, DAL_rpm_MessageType *
                               MessageHandle)
```

→	h	Handle acquired from DAL_RPMDeviceAttach
→	MessageHandle	Pointer to a message that will be appended into and eventually sent

Return value

The function returns a handle via the out parameter and passes this message handle to AppendRequest and SendMessage.

4.5.1.4 DalRPM_AppendRequest

This API places a request into an RPM message but does not immediately send the message. SendMessage must be called before the RPM will receive the request.

Prototype

```
DALResult DalRPM_AppendRequest(DalDeviceHandle * _h,
                               DAL_rpm_MessageType  MessageHandle,
                               DAL_rpm_ResourceType  Resource,
                               uint32  DataLen, void * Data)
```

Description

Arguments are the same as in SendRequest, except that in this instance the request is not sent immediately.

4.5.1.5 DALResult DalRPM_SendMessage

This API sends the multirequest message to the RPM.

Prototype

```
DALResult DalRPM_SendMessage(DalDeviceHandle * _h, DAL_rpm_MessageType
                             MessageHandle)
```

→	h	Handle acquired from DAL_RPMDeviceAttach
→	MessageHandler	Pointer to a message that will be sent

Description

The RPM receives all requests contained in the message at the same time.

Sending a message this way has the same overhead as calling SendRequest but can send many more requests at the same time. This function behaves the same as SendRequest in many ways:

- It blocks until the response from the RPM is received.
- The function does not busy wait, instead it yields until the response interrupt allows it to continue.

4.5.1.6 DALResult DalRPM_DestoryMessage

This API cleans up the scratch space in an RPM multipart message when it is no longer needed, which frees up some heap space.

Prototype

```
DALResult DalRPM_DestroyMessage(DalDeviceHandle * _h,
                                DAL_rpm_MessageType  MessageHandle)
```

→	h	Handle acquired from DAL_RPMDeviceAttach
→	MessageHandle	Pointer to a message that will be sent

Description

The message handle should not be used after being destroyed.

4.5.2 DALRPMFW

DalRPMFW's interface is in core\api\power\DDIRPMFW.h. The driver is a DAL, which means the first call that must be made to it is DAL_RPMFWDeviceAttach, which returns a handle to the device.

This handle can be thought of as the context when talking to the device; the handle is passed as the first argument of all other driver functions.

DalRPMFW allows a driver to register itself as the owner of a resource and accepts requests for that resource.

Registration as a driver can follow two paths:

- Register as a "raw" driver
  - Uses DalRPMFW_RegisterDriver
  - Raw drivers receive a callback whenever a new resource request comes in
  - Raw drivers must perform their own aggregation

Example of registering as a raw driver:

```
    static DALSYSEventHandle gRPMFWNotifyEvent;
/* table of resources this driver handles */
static DAL_rpm_ResourceType notificationResources[] =
    { DAL_RPM_RESOURCE_NOTIFICATIONS };
void *RPMFW_Notification_Update(void *pCtxt, uint32
EffectiveImmediately, void *payload, uint32 payloadLen);
    RetVal = DALSYS_EventCreate(DALSYS_EVENT_ATTR_CALLBACK_EVENT,
                                &gRPMFWNotifyEvent, NULL);
    if(DAL_SUCCESS != RetVal) break;
    RetVal = DALSYS_SetupCallbackEvent(gRPMFWNotifyEvent,
```

```
                                                RPMFW_Notification_Update, NULL);
        if(DAL_SUCCESS != RetVal) break;
        /* register internal driver. internal drivers have no client context
    */
        RetVal = DalRPMFW_RegisterDriver(handle, /* context from RPMFW
    attach */

    sizeof(notificationResources)/sizeof(notificationResources[0]),
                                    notificationResources,
                                    gRPMFWNotifyEvent);
        if(DAL_SUCCESS != RetVal) break;
```

- Register as an NPA driver
  - RPMFW creates a client for each master to the registered NPA resource
  - NPA resources get standard NPA aggregation but cost significantly more heap space vs raw
  - NPA resources can be accessed from other local code on the RPM just as easily as by remote requests

Example of registering as an NPA driver:

```
// Registering an NPA driver is quite easy. You simply have to register the
name of your
// resource with the corresponding enumeration that you're claiming
ownership over.
DALDEVICEHANDLE rpm_handle;
assert(DAL_SUCCESS == DAL_RPMFWDeviceAttach(DALDEVICEID_RPMFW,
&rpm_handle));
DalRPMFW_RegisterNPADriver(rpm_handle, DAL_RPM_RESOURCE_CXO, "/xo/cxo");
```

With NPA drivers, the RPMFW framework creates a client for each master that can make a request to that resource. Requests are then received by the NPA driver just like any other request coming in (just like a request from local code).

NPA drivers are very convenient ways to register resources. For small numbers of resources their use is recommended.

However, for large numbers of resources it is possible that the extra heap usage of NPA drivers vs raw drivers can be difficult to fit into the limited size available to the RPM.

5 RPM Scheduling

5.1 Task

An RPM task is an instance of the class Task. The class Task is defined in rpm_task.h and rpm_task.cpp. It has the following interface:

- Get task name
  `const char *get_name() const;`
- Get task priority
  `uint8_t get_priority() const;`
- Set the deadline by which the execution should complete
  `void set_deadline(uint64_t deadline);`
- Get the deadline by which the execution should complete
  `bool get_deadline(uint64_t& deadline) const;`
- Get the estimate of how long this task takes to execute
  `virtual uint64_t get_length() const = 0;`
- Get the time the task should start by in order to finish on time
  `uint64_t get_start() const;`
- Set the time the task should start by in order to finish on time
  `void set_start(uint64_t start_time);`
- Check if the task has immediate work to do
  `virtual bool hasImmediateWork() const = 0;`
- Check if the task has scheduled work to do
  `virtual bool hasScheduledWork() const;`
- Run this task until it completes, stops, or be preempted
  `void execute(volatile bool &preempt, uint64_t stop_time);`

There are two tasks created for each master: one of RPMFWRequestHandler which handles resource requests from the master; and one of RPMFWSetChanger which handles set transitions when the master enters or exits sleep.

The task priority is inherited from its master: MPSS has the highest priority, APPS has the intermediate, and LPASS has the lowest priority.

5.2 Dispatcher

A dispatcher is an object embedded in a task. The dispatcher executes the batches of work associated with the task. The class RPMFWDispatcher is defined in DalRPMFWDispatcher.h and DalRPMFWDispatcher.cpp, which has the following interface:

- Set up a new batch of work to be executed

```
void set_work(const DAL_rpm_ResourceIndType &changed_resources,
              DAL_rpm_ConfigSetType          changed_set);
```

- Get the estimate worst-case execution time

```
uint64_t get_length() const;
```

- Perform the dispatching until all tasks are done, run out of time, or be preempted

```
bool dispatch(volatile bool         &preempt,
              uint64_t               stop_time,
              bool                  &rejected,
              RPMResourceDataLayout &new_req,
              bool                   save_req,
              RPMResourceDataLayout *acknowledgement);
```

5.3 Scheduler

The scheduler is the static instance of the class Sched which implements cooperative multiple tasking. The class Sched is defined in rpm_sched.h and rpm_sched.cpp.

The scheduler has three private members:

- Immediate task list – Tasks with immediate work will be kept in this binary heap.

```
TaskHeap immediateQ_;
```

- Scheduled task list – Tasks with scheduled work will be added into this sorted list.

```
TaskList scheduledQ_;
```

- Preemption flag – The dispatcher checks this flag periodically to see if it should stop current work permaturately and yield the processor for higher priority task.

```
volatile bool preempt_;
```

The scheduler provides the following interfaces:

- Retrieve the single scheduler instance

```
friend Sched &theSchedule();
```

- Add a new task to be scheduled, causing the task be added to the immediate or scheduled list and the preemption flag be updated as needed

```
void schedule_task(Task &new_task,
                   ScheduleType schedule_type = DEFAULT);
```

- Run the next outstanding task, returning only when there is no immediate work to do

```
void run();
```

- Return the time by which the scheduler should run next

```
uint64_t get_next_start() const;
```

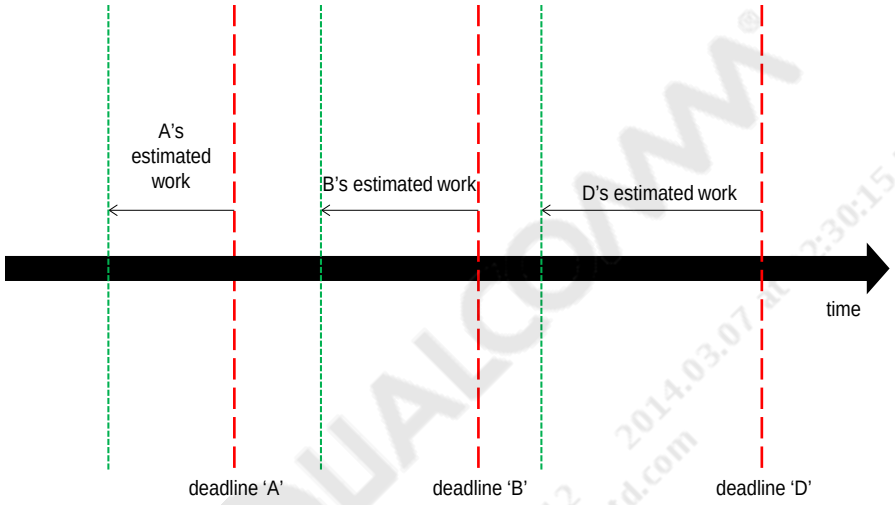
- Return the estimated duration of the scheduler's next run

```
uint64_t get_next_duration() const;
```

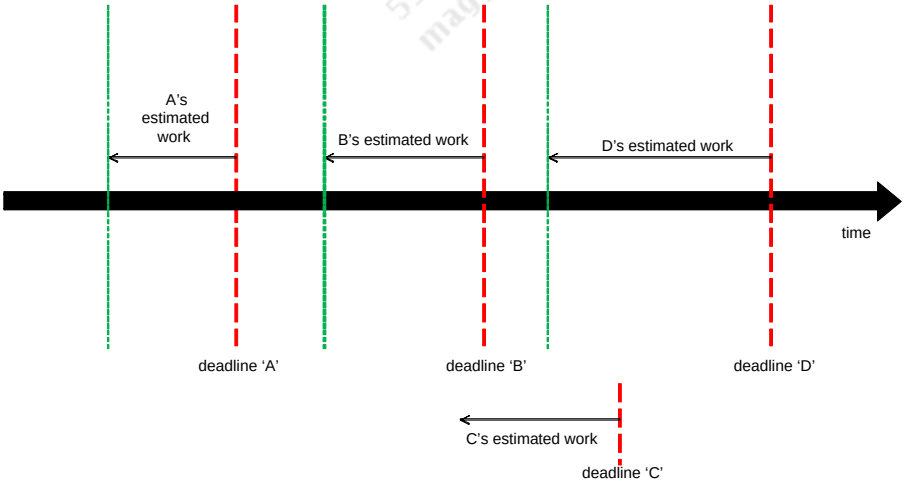
5.4 Schedule collision – Stack up example

Every scheduled conflict has either been explored or is just another permutation of what has been covered. What about immediate work? How should it interact with the scheduled timeline? An example follows.

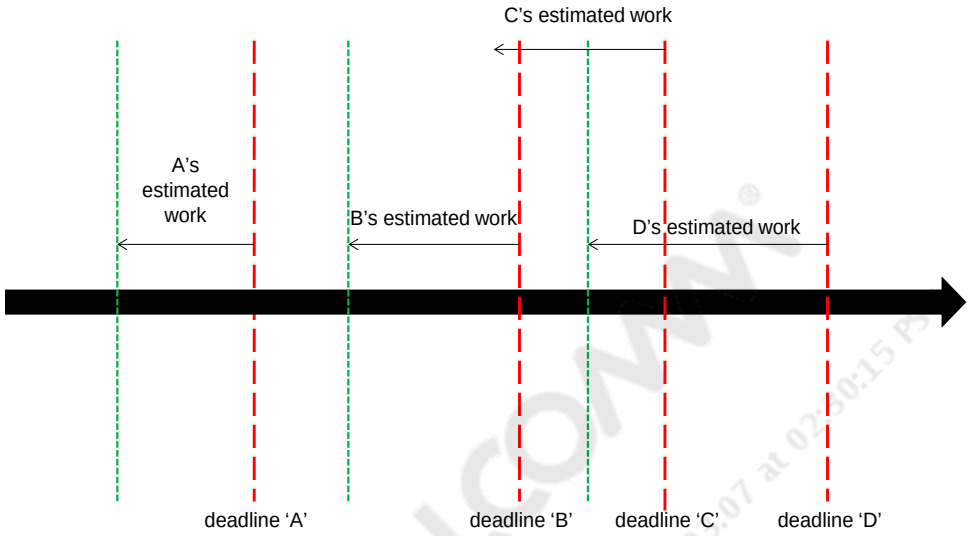
1. Task A, task B, and task D are on the schedule, each with their own deadline – deadline A, deadline B, and deadline D.



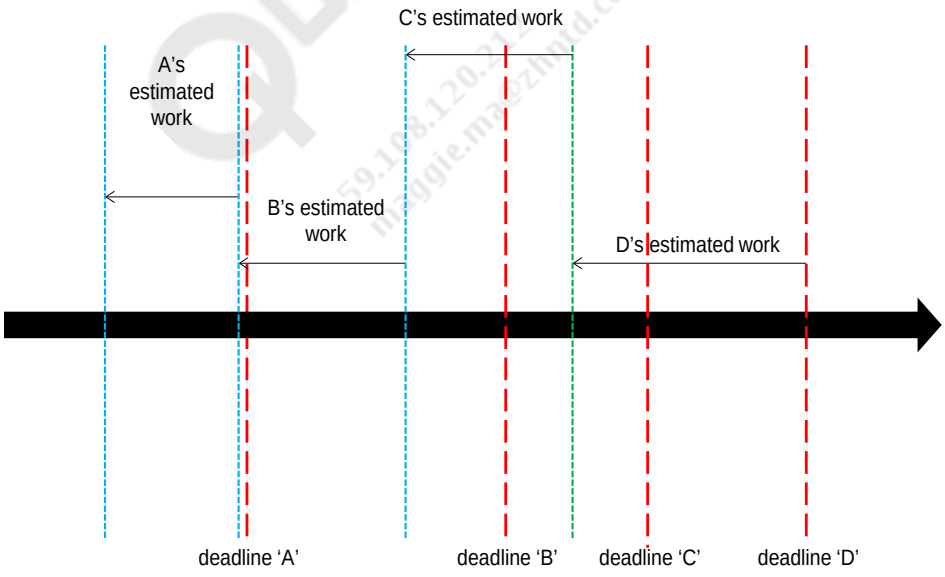
2. A new task, task C, comes in with a new deadline, deadline C.



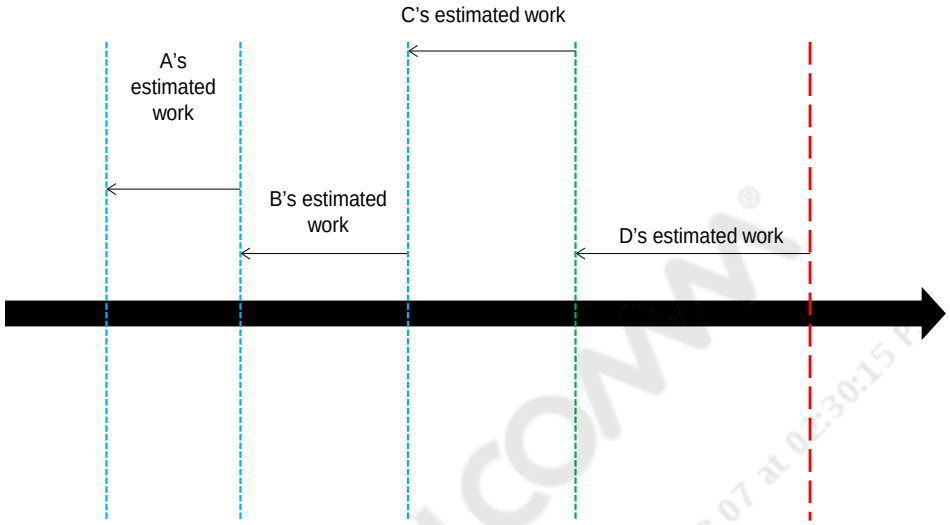
3. Task C is inserted and based on the deadline, task C now conflicts with task D in the queue, so the execution time for task C is pushed out.



4. Now, task A conflicts with task B, so the execution time for task B is pushed out.



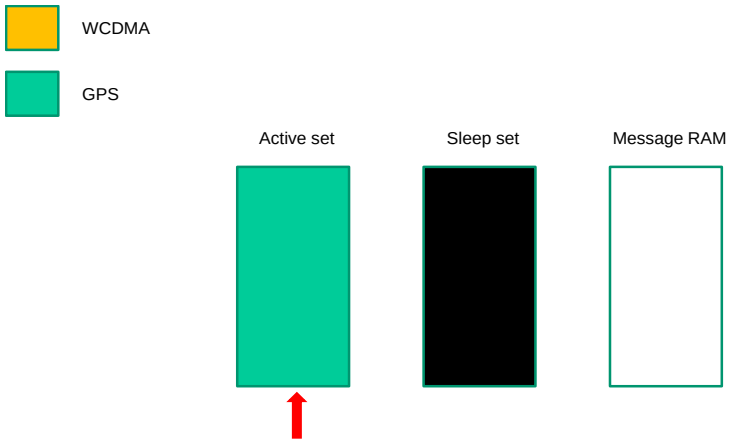
5. Task D's original schedule remains unchanged. The rest of the schedule needed modifications. This is called stack up when the problem occurs, and the resolution is called schedule fix up.



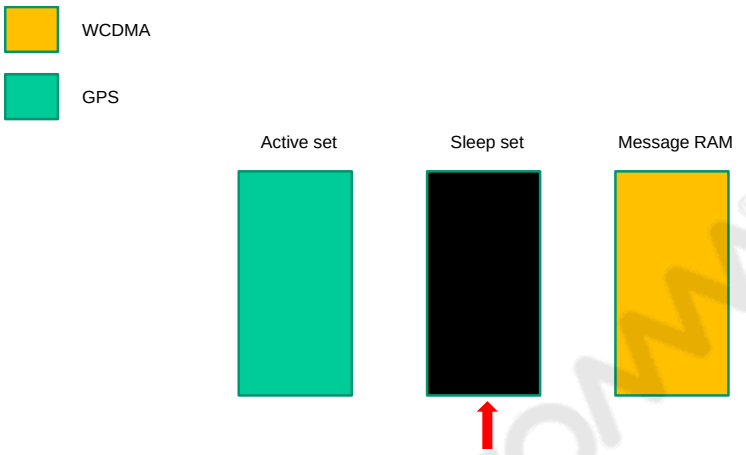
5.5 Concurrency – Next awake set

Wake-up information from the modem goes from time to deadline; missing the deadline is considered failure. Allow the modem to avoid a resource double state change by storing a next awake set into memory before sleep to save the handshake time cost as described in the following steps.

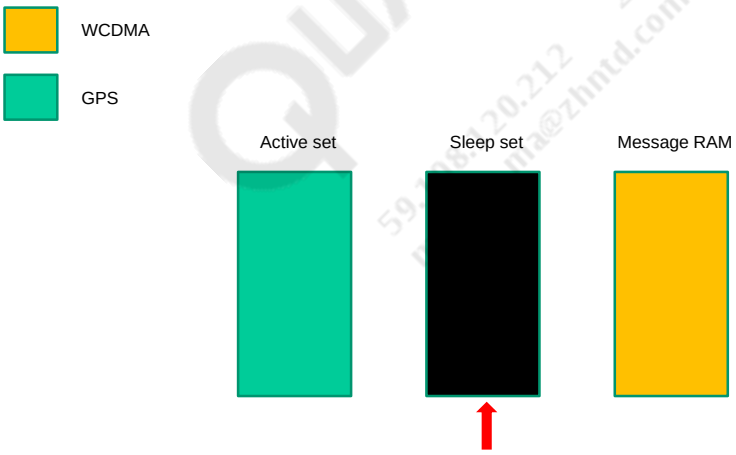
1. When going to sleep, a new sleep set is written and sent to the RPM. The settings for the next wake-up are then left as an unsent request in message RAM.



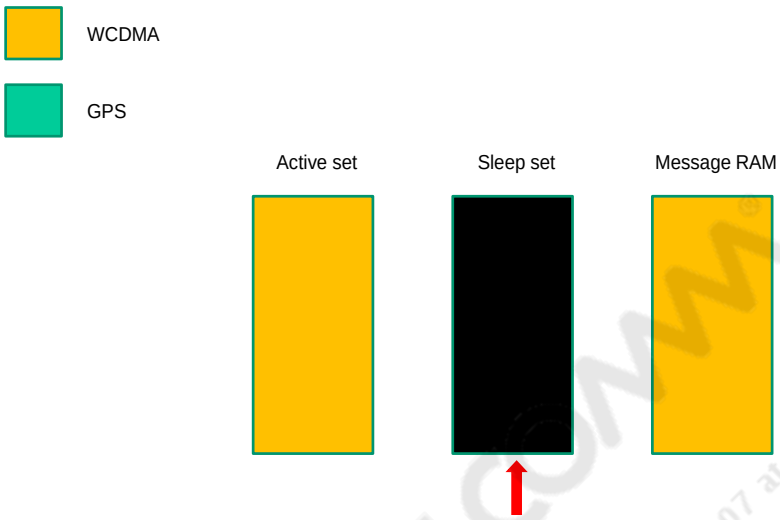
2. Once the RPM detects a sleep transition on a master, it will check to see if there is an unsent message in message RAM.



3. If there is an unsent message, it will schedule the application of that request just before the wake-up of that master.



4. At wake-up, the active set is updated. This allows the active set to be changed ahead of time and to skip double-state changes upon wake-ups.



5.6 Schedule code layout

- Request handling algorithms:
  - DalRPMFWHandler.cpp
  - DalRPMFWHandler.h
- Set transition algorithms:
  - DalRPMFWSetChanger.cpp
  - DalRPMFWSetChanger.h
- Common code between request handling and set transitions – Iterating over a list of resource requests and sending them off to drivers:
  - DalRPMFWDispatcher.cpp
  - DalRPMFWDispatcher.h
- Schedule code – Code that is responsible for keeping track of how long each resource will take to transition:
  - DalRPMFWEstimator.cpp
  - DalRPMFWEstimator.h
- Schedule framework:
  - rpm_sched.cpp
  - rpm_task.cpp
  - rpm_timeservice.cpp
  - task_heap.cpp
  - task_list.cpp

5.7 Example of scheduling with preemption

Figure 5-1 shows an example of scheduling with preemption.

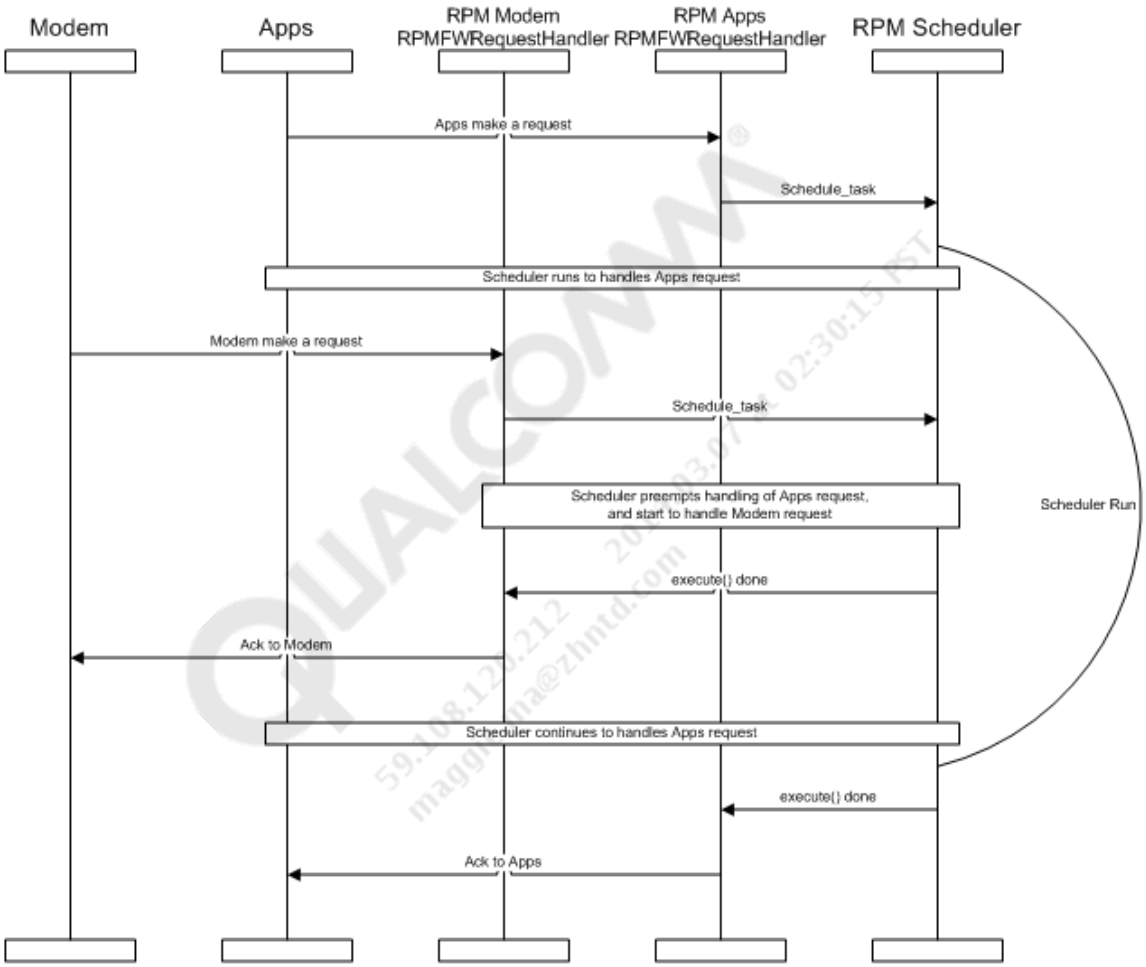


Figure 5-1 Scheduling with preemption

6 RPM BRCPR

NOTE: This chapter was added to this document revision.

6.1 RPM Core Power Reduction (CPR)

CPR is a system that embeds into SoC and is used to control the VDD level of a chip. CPR consists of CPR master core and satellite sensors which are integrated into various blocks on SoC. It utilizes sensors to estimate whether the chip is relatively fast or slow, then produces a result that can be interpreted and used to send a VDD modification command to the PMIC chip. CPR driver has been added in RPM to control the Vdd Dig, Vdd Gpu, etc.

Figure 6-1 illustrates the RBCPR core from RapidBridge. It consists of one master and a number of sensors. There is additional Qualcomm wrapper logic called rbif. The software can program registers in the rbif_csr block, in the Qualcomm rbif. There are no programmable registers in the RapidBridge logic.

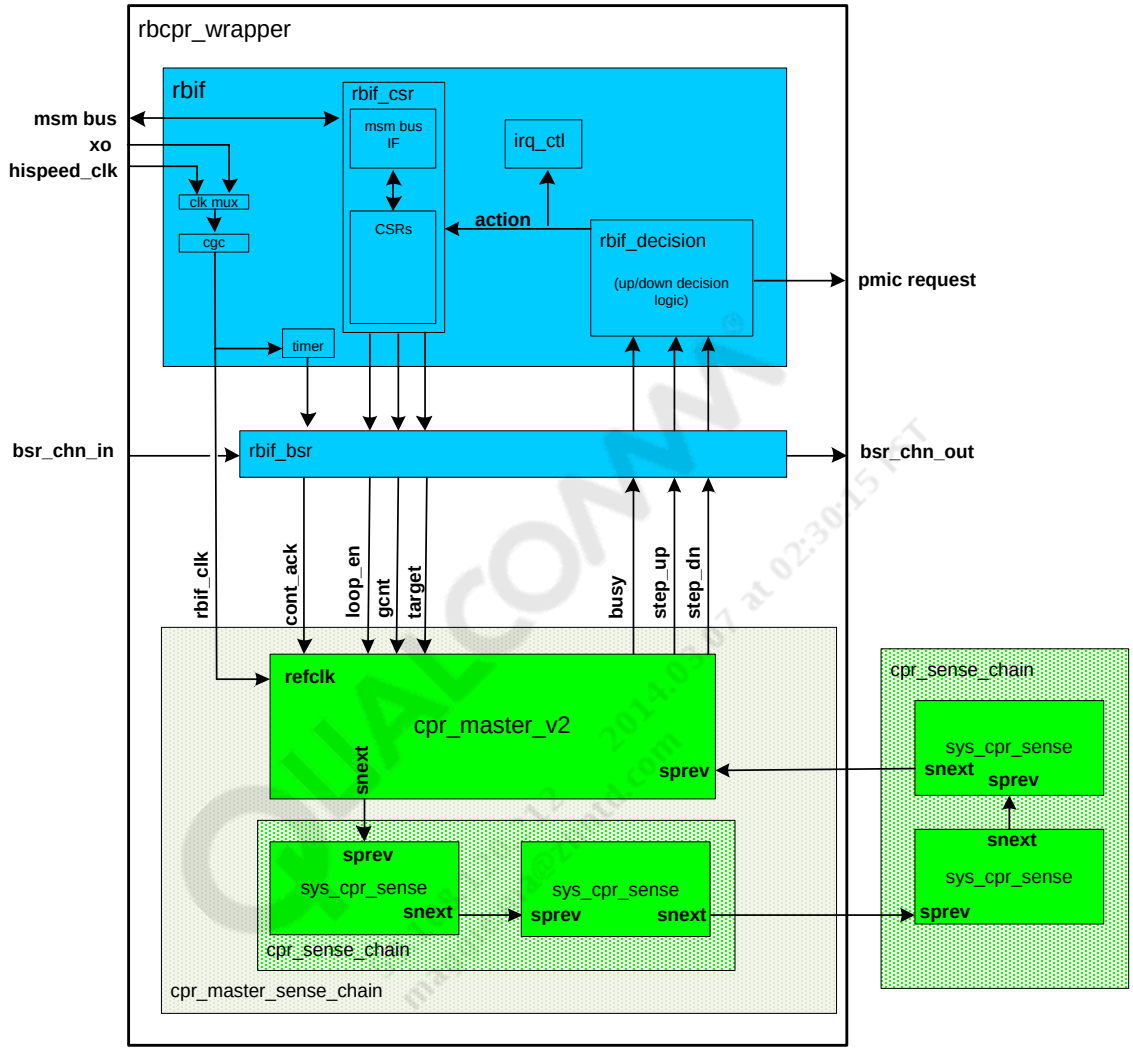


Figure 6-1 RPM message RAM partition

6.2 CPR initialization

The software must configure some registers prior to using RBCPR. One of the register values, STEP_QUOT, determines how many PMIC voltage steps should be taken to compensate for changes in the quotient (QUOT). The QUOT is basically the raw count of the slowest sensor. The QUOT, or raw count, is taken while the cpr_master is counting GCNT cycles of the reference clock. If the silicon is on the slow side, the QUOT value will be lower; if the silicon is on the fast side, the QUOT will be higher.

During RBCPR operation, the QUOT is subtracted from the target value (TARG) to determine an error value. This error value is converted to some number of PMIC steps, by factoring in the STEP_QUOT.

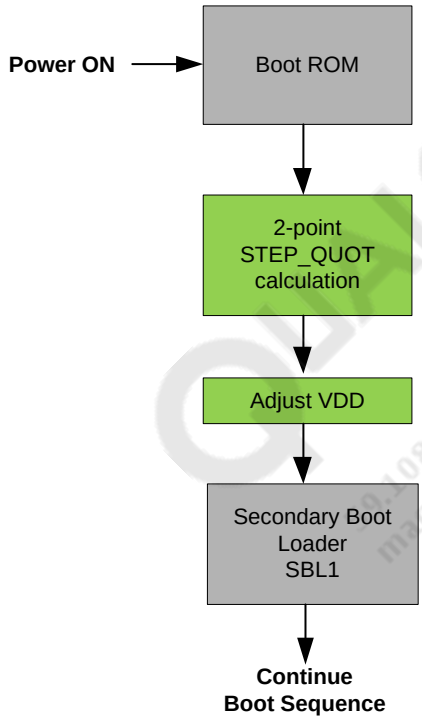


Figure 6-2 2-point STEP_QUOT calculation during boot

Performs an arithmetic calculation in the software, to derive STEP_QUOT:

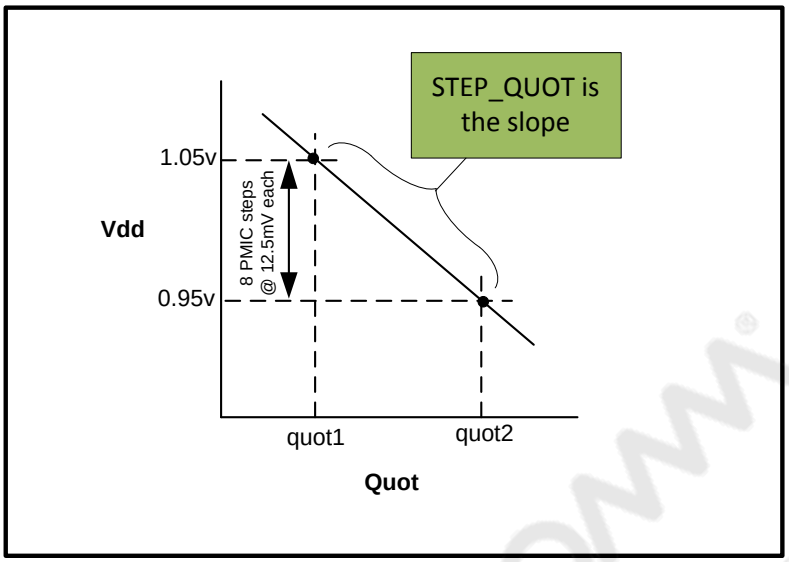


Figure 6-3 STEP_QUOT is how many QUOT units per PMIC step

The reason we divide by 8 is because the two quotients were measured at Vdd points that are 8 PMIC steps apart. Each PMIC step is 12.5 mV, and we measured the quot_1 at 1.05 v, and quot_2 at 0.95 v. The STEP_QUOT tells us how many QUOT units correspond to one PMIC step.

- quot1 – quot2 = delta_quot
- delta_quot / 8 = STEP_QUOT

NOTE: On 8x30 RPM, the 2-point STEP_QUOT calculation has been removed recently.

6.3 CPR measurement and adjustment

This section describes how CPR takes measurements and adjustments.

1. Configure the CPR interrupts
2. Receive CPR interrupt, look at RBCPR_STATUS, and make PMIC adjustment
   a. If step_up is 1, increase PMIC Vdd by one step
   b. If step_down is 1, decrease PMIC Vdd by one step
3. Take another RBCPR measurement

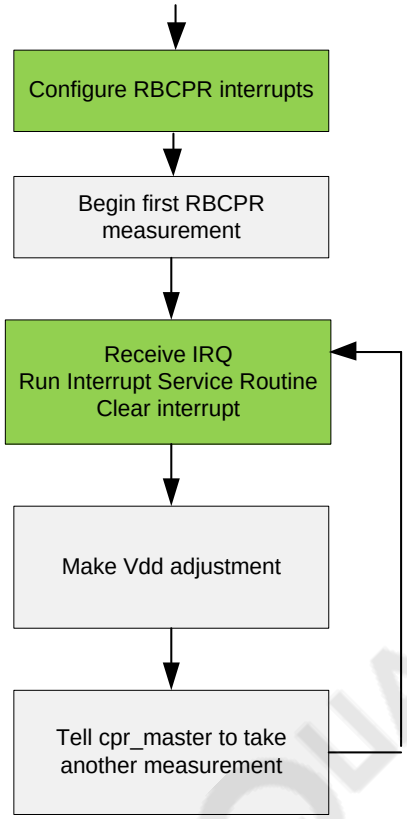


Figure 6-4 CPR measurement/adjustment

6.4 CPR voltage switching

RPM runs in one of three modes:

- Nominal
- Turbo
- SVS

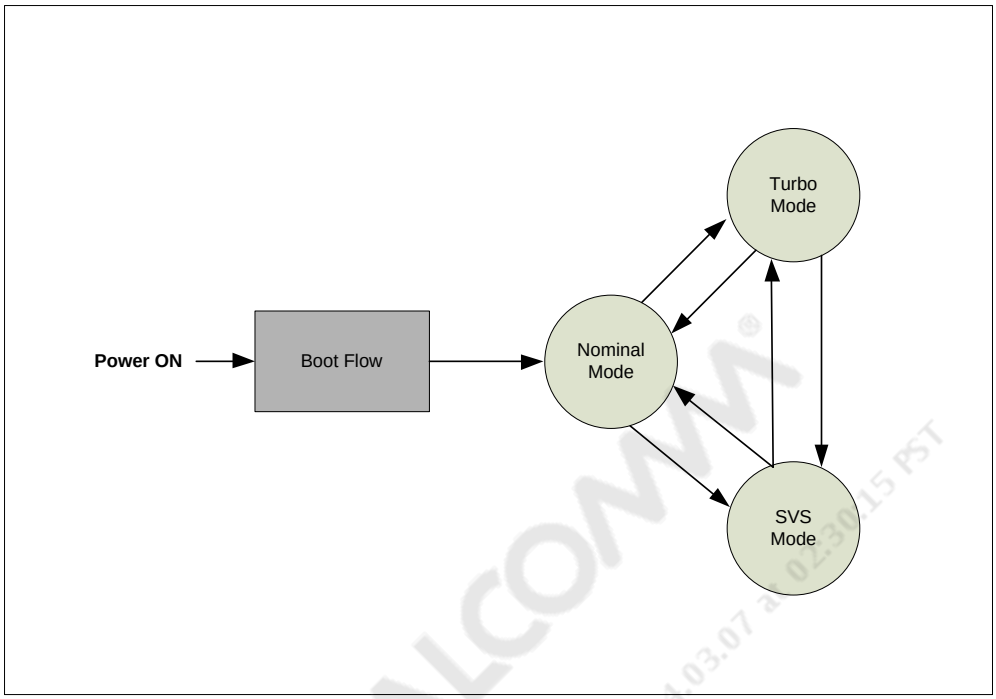


Figure 6-5 CPR measurement/adjustment

Whenever there is a requirement to switch the core voltage for a mode change, RBCPR callbacks are called before and after the voltage to:

- Disable the RBCPR block and interrupts
- Set new Vdd through the software (not RBCPR)
- Perform 2-point STEP_QUOT calculation
- Configure RBCPR with the new GCNT/TARG pairs
- Re-enables the block and interrupts

6.5 CPR driver source code

The source code resides in:

- Folder rpm_proc\core\power\rbcpr\.
  - The read only data is present in HAL_rbcpr_bsp.c and rbcpr_bsp.c, depending on whether the data applies to the hardware register settings or software algorithm respectively.
  - Driver code is in rbcpr.c
- Cx rail voltage is adjusted by the CPR driver. Mx voltage is always set to a static or “safe” voltage for the operating corner.
- The recommendation provided by CPR is in terms of PMIC steps.

  1 step = 12.5 mV

6.6 CPR debug

6.6.1 Enable/disable CPR at runtime

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

6.6.1.1 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

6.6.1.2 Enable/disable CPR in Linux kernel

To enable/disable CPR in 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"

6.6.2 Retrieve RBCPR log

RBCPR stats are to collect information of the voltage scaling recommendations from RBCPR hardware:

- Fuse voltage (CPR starting point)
- For each mode (SVS/Nominal/Turbo):
- # of interrupts in the mode
- The latest 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 stats

RBCPR stats are put at a dedication location of RPM MSG RAM in order to make it always available for HLOS to read and send out through some diagnostic mechanism.

In 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 RPM Debug

7.1 Load RPM RAM dump

To open the RPM RAM dump:

1. Open a T32 simulator and do sys.up.
2. Jump to the dumps directory and load the following:

```
d.load.binary MSGRAM.bin RPM_MSG_RAM_START_ADDRESS
d.load.binary CODERAM.bin RPM_CODE_RAM_START_ADDRESS
```

   - See Section 7.2.3 to find out which RPM build version the customer uses and load the respective RPM ELF.
   - See Section 7.2.4 to retrieve the ULOG log.
   - See Section 7.2.4.2 to retrieve the NPA log.
3. v.v gpRPMFWMaster to get the global firmware setting.
4. Do <RPM Build>\core\power\rpm\dal\scripts\rpm_loadcoredump.cmm to recover the core dump.

7.2 RPM log collection

The RPM publishes a small log into a very limited area of spare message RAM. The physical format of the log is the ULog format used for various other logs.

- A circular buffer – Currently size is limited
- A raw log – Uses a set of IDs and a variable number of parameters per message

To get the RPM RAM dump, licensees must dump the RPM code RAM dump and message the RAM dump. See Table 7-1 for RPS address settings.

Table 7-1 RPM address settings

	MSM8660	MSM8960/8930	MDM9x15	APQ8064
RPM_CODE_START_ADDRESS	0x20000	0x20000	0x20000	0x20000
RPM_CODE_SIZE	0x20000	0x24000	0x24000	0x28000
RPM_MSG_RAM_ADDRESS	0x104000	0x108000	0x108000	0x108000
RPM_MSG_RAM_SIZE	0x4000	0x6000	0x6000	0x6000
RPM_SWVERSION_ADDRESS	0x104008	0x108008	0x108008	0x108008

7.2.1 RPM RAM dump collection with T32

Licensees can save the RPM dumps and send them for further analysis. To save the RPM dumps:

1. Create the issue scenario where RPM dumps are needed.
2. Open ARM7 T32 and attach (sys.m.a).
3. Break T32 and do the following for saving the RPM memory dump:

```
d.save.binary C:\Temp\CODERAM.bin RPM_CODE_START_ADDRESS++RPM_CODE_SIZE
d.save.binary C:\Temp\MSGRAM.bin   RPM_MSG_RAM_ADDRESS++RPM_MSG_RAM_SIZE
```

7.2.2 RPM RAM dump collection with QPST

When the system goes to the SBL error handler, the licensee can retrieve the RPM ram dump with the QPST memory dump application tools.

Once the apps processors detect the system IMEM, the magic number is set to trigger the DLOAD mode; it will not load RPM so RPM MSG RAM will not be overwritten.

7.2.3 RPM version check

If you have the build running, you can check using T32 using this memory location:

```
d.in RPM_SWVERSION_ADDRESS  /LONG
```

For example:

```
d.in 0x00104008 /LONG
```

This would print:

```
SD:00104008 = 00000245
```

where 0x245 → 0x0.0x2.0x45 → 00.02.69. The location is in hex, so 0x245 = build 00.02.69.

7.2.4 Retrieving logs with T32

7.2.4.1 Ulog

```
data.load.elf \\<RPMBUILD>\\build\rpm\<MSMPlatform>\build\RPM.elf /nocode
(loads the symbols)
do \\<RPMBUILD>\core\power\ulog\scripts\ULogDump.cmm
```

7.2.4.2 NPA log

```
data.load.elf \\<RPMBUILD>\\build\rpm\<MSMPlatform>\build\RPM.elf /nocode
(loads the symbols)
do \\<RPMBUILD>\core\power\npa\scripts\NPADump.cmm
```

7.2.4.3 Parsing Ulog

The parsed_output.txt is the readable text format Ulog.

```
Python \\<RPMBUILD>\rpm\scripts\rpm_log.py -f "RPM External Log.ulog"
-n "NPA Log.ulog" > parsed_output.txt
```

7.2.5 Retrieving logs with ADB

7.2.5.1 Ulog

```
adb shell mkdir /data/debug
adb shell mount -t debugfs none /data/debug
adb shell cat /data/debug/rpm_log > “RPM External Log.ulog"

Python \\<RPMBUILD>\rpm\scripts\rpm_log.py -f "RPM External Log.ulog"
> parsed_output.txt
```

7.2.5.2 RPM status

The RPM status shows the number of times that RPM got to XO shutdown/VDD min and how long it stays in the low power stage.

```
adb shell mount -t debugfs none /sys/kernel/debug
adb shell cat /sys/kernel/debug/rpm_stats
```

7.3 Log analysis

7.3.1 RPM log examples

Example 1

The first example log shows that the modem subsystem requests multiresource changes, the apps FABRIC, the system FABRIC, and EBI CLK are updated by the RPM, and the Ack information is sent to MSS:

```
0x4b4eac: rpm_message_int_received (master: "MSS")
0x4b4eae: rpm_servicing_master (master: "MSS")
0x4b4eb2: rpm_driver_dispatch (resource: "Apps Fabric Clock")
0x4b4eb5: rpm_driver_complete (rejected: 0)
0x4b4eb6: rpm_driver_dispatch (resource: "System Fabric Clock")
0x4b4ec1: rpm_driver_complete (rejected: 0)
0x4b4ec3: rpm_driver_dispatch (resource: "EBI1_CLK")
0x4b4ec8: rpm_driver_complete (rejected: 0)
0x4b4ec9: rpm_sending_message_int (master: "MSS")
```

Example 2

In the next example, one of the apps processors used the SAW to go to sleep, and used a mode with RPM handshake:

```
568.205811: rpm_shutdown_req (master: "APSS")
568.205841: rpm_shutdown_ack (master: "APSS")
```

Example 3

In this example, one of the apps processors woke up from a sleep mode that uses an RPM handshake:

```
572.510834: rpm_bringup_req (master: "APSS")
572.510864: rpm_bringup_ack (master: "APSS")
```

Example 4

Both of these are messages that indicate transitions that occur when all of the processors for a subsystem are asleep, either both apps cores, or the modem, or the Hexagon, depending on the value of the master.

```
736.674957: rpm_master_set_transition (master: "APSS") (leaving: "Active
Set") (entering: "Sleep Set")
811.198547: rpm_master_set_transition (master: "APSS") (leaving: "Sleep
Set") (entering: "Active Set")
```

Example 5

In this example, the LPASS master is requesting the RPM for a PXO resource; rejected: 0 implies that the request went through.

```
78.945282: rpm_sending_message_int (master: "LPASS")
78.945404: rpm_message_int_received (master: "LPASS")
78.945465: rpm_servicing_master (master: "LPASS")
78.945557: rpm_driver_dispatch (resource: "PXO")
78.945648: rpm_driver_complete (rejected: 0)
```

Example 6

In this example, the system is entering and exiting VDD minimization; the count is the total number of times this state has been entered, and sleep time is (81-79)=2 sec.

```
79.547623: rpm_vdd_min_enter (count: 10)
81.763244: rpm_vdd_min_exit
```

Example 7

The MSS has requested that the contents of a set be invalidated. This is generally used to clear the set contents.

```
 82.002363: rpm_servicing_master (master: "MSS")
 82.002543: rpm_driver_dispatch (resource: "Request Invalidate")
 82.002684: rpm_driver_complete (rejected: 0)
```

7.3.2 NPA log examples

Example 1

The RPM contains 1 low power node = /node/sleep/uber, which requests the low power resources. PXO is not required in the example.

```
0x4D34A5: npa_issue_required_request (handle: 0x0003ED64) (client: "sleep")
(request: 13) (resource: "/sleep/uber")
Requests are determined by looking at what bits are set: 13 => 1101=> mem
dig pxo cxo
In this example:
Vdd Mem is required
Vdd Dig is required
PXO is not required
CXO is required
```

Example 2

Requests on the /node/sleep/uber usually result in a request to /sleep/LPR to enable/disable.

```
0x4D34A6: npa_issue_required_request (handle: 0x0003ED1C) (client:
"/node/sleep/uber") (request: 1) (resource: "/sleep/lpr")
Requests values are:
0x1 = RPM Halt only
0x2 = XO Shutdown
0x4 = Vdd minimization
0x4D34A7:    request complete (handle: 0x0003ED1C) (sequence: 0x00013F01)
(request state:1) (active state:1)
0x4D34A7:    request complete (handle: 0x0003ED64) (sequence: 0x0002D500)
(request state:13) (active state:13)
```

Example 3

Scorpion 0 requests the system FABRIC run at 157 MHz.

```
: npa_resource (name: "/clk/bus/sfab") (handle: 0x3B48C) (units: KHz)
(resource max: 157250) (active max: 157250) (active state 157250)  (active
headroom: -157250) (request state: 0)
:         npa_client (name: /clk/bus/sfab) (handle: 0x3B994) (resource:
0x3B48C) (type: NPA_CLIENT_REQUIRED) (request: 0)
:         npa_client (name: LPASS) (handle: 0x3BE9C) (resource: 0x3B48C)
(type: NPA_CLIENT_REQUIRED) (request: 0)
```

```
:         npa_client (name: Modem) (handle: 0x3BED4) (resource: 0x3B48C)
(type: NPA_CLIENT_REQUIRED) (request: 0)
:         npa_client (name: Scorpion 0) (handle: 0x3BF0C) (resource:
0x3B48C) (type: NPA_CLIENT_REQUIRED) (request: 1)
:         npa_threshold_event (name: /clk/bus/sfab) (handle: 0x3C5DC)
(resource: 0x3B48C) (lo_threshold: 0) (hi_threshold: 0)
:         npa_change_event (name: ) (handle: 0x3C720) (resource: 0x3B48C)
:         end npa_resource (handle: 0x3B48C)
: npa_resource (name: "/clk/bus/afab") (handle: 0x3
```

Example 4

An example of NPA resource/client is:

```
0x65BD6: npa_resource (name: /core/cpu) (handle: 0x24B7070) (units: MIPS)
(resource max: 480) (active max: 512) (active state 480) (active headroom:
-32) (request state: 480)
0x65BD6: npa_client (name: wl1_npa_mcpu) (handle: 0x24C8A98) (resource:
0x24B7070) (type: NPA_CLIENT_REQUIRED) (request: 122)
0x65BD6: npa_client (name: GPS_CPU_CLIENT) (handle: 0x24C8BA8) (resource:
0x24B7070) (type: NPA_CLIENT_REQUIRED) (request: 0)
0x65BD6: npa_client (name: /core/wiredconnectivity/hsusb) (handle:
0x24C8CB8) (resource: 0x24B7070) (type: NPA_CLIENT_REQUIRED) (request: 480)
0x65BD6: npa_client (name: audio) (handle: 0x24C1BA0) (resource: 0x24B7070)
(type: NPA_CLIENT_REQUIRED) (request: 0)
0x65BD6: npa_client (name: CPU Dynamics Timer) (handle: 0x24B43A8)
(resource: 0x24B7070) (type: NPA_CLIENT_RESERVED2) (request: 0)
0x65BD6: end npa_resource (handle: 0x24B7070)
```

In this log:

- Name: /core/cpu – Use this to connect to the resource.
- Units: MIPS – All required requests are in MIPS.
- Resource Max: 480 – 480 MIPS is the maximum value this target supports.
- Active State: 480 – The resource is currently at 480 MIPS.
- Request State: 480 – This changes per resource, but for /core/cpu this is the sum of all requests in the system.

7.3.3 gpRPMFWMaster

gpRPMFWMaster is the global variable which includes the running status of all master information.

```
typedef struct
{
  /* basic information required at various points */
  DAL_rpmfw_MasterType master_id;

  /* configuration set data */
  DAL_rpm_ConfigSetType        selected_set;
  DAL_rpmfw_ConfigSetDataType  set[DAL_RPM_CONFIG_SET_COUNT];

  /* defer engine data */
  uint32            defer_count;
  uint32            defer_actions;
  DALSYSEventHandle defer_timer;

  /* notification 'driver' data */
  DAL_rpm_ResourceIndType notify_pending;
  DALBOOL                 notify_in_queue;

  /* spm handshake data */
  DAL_rpmfw_SPMStatusType spm_subsystem_status;
  uint32                  spm_num_active_cores;
  uint32                  spm_pending_bringups;

  /* trigger 'driver' data */
  DALSYSEventHandle         triggers_timer;
  DALBOOL                   triggers_timer_expired;

  /* request handler data */
  DAL_rpm_ResourceType resource;
} DAL_rpmfw_MasterDataType;
```

7.3.4 DAL_rpmfw_MasterDataType

This stores all of the states related to each master processor.

```
typedef enum
{
  DAL_RPMFW_MASTER_0 = 0,  /* all targets: APSS */
  DAL_RPMFW_MASTER_1,      /* all targets: MPSS SW */
  DAL_RPMFW_MASTER_2,      /* all targets: LPASS */
```

```
  DAL_RPMFW_MASTER_3,      /* 8960: RIVA */
  DAL_RPMFW_MASTER_4,      /* 8960: DSPS */
  DAL_RPMFW_MASTER_5,      /* 8960: MPSS FW */
  DAL_RPMFW_MASTER_COUNT,
  DAL_RPMFW_MASTER_SIZE = 0x7FFFFFFF
} DAL_rpmfw_MasterType;
```

See Table 7-2 for the bus master for different platforms.

Table 7-2 Bus master for different platforms

	MSM8660	MSM8960/8930	MDM9x15	APQ8064
Master 0	Scorpions	Kraits	Sparrow	Kraits
Master 1	ARM11	Hexagon software	Hexagon software	GNSS
Master 2	LPASS	LPASS	LPASS	WCNSS
Master 3	N/A	WCNSS	N/A	Sensors ARM7
Master 4	N/A	DSPS (not DSPS for MSM8930)	N/A	LPASS
Master 5	N/A	MPSS	N/A	N/A

7.3.5 Selected set

The selected set indicates which configuration set the RPM is using for the master; there are currently two sets.

Active sets for the current resource configuration

In the following example, the master 0 is in the active settings configuration (red text).

```
gpRPMFWMaster = (
  [0] = 0x00039B4C -> (
  master_id = DAL_RPMFW_MASTER_0 = 0 = 0x0,
  selected_set =DAL_RPM_CONFIG_SET_PRIMARY=DAL_RPM_CONFIG_SET_ACTIVE_0 =0x0,
      set = (
        [0] = 0x00039BD4 -> (
          resource = (
            [0] = (data_length = 64 = 0x40, data_valid = 0 = 0x0, data =
0x00039E74),
            [1] = (data_length = 12 = 0x0C, data_valid = 0 = 0x0, data =
0x00039EB4),
            [2] = (data_length = 8 = 0x8, data_valid = 0 = 0x0, data =
0x00039EC0),
            [3] = (data_length = 4 = 0x4, data_valid = 0 = 0x0, data = 0x0),
            [4] = (data_length = 0 = 0x0, data_valid = 0 = 0x0, data = 0x0),
            [5] = (data_length = 4 = 0x4, data_valid = 0 = 0x0, data = 0x0),
            [6] = (data_length = 4 = 0x4, data_valid = 1 = 0x1, data = 0x1),
            [7] = (data_length = 4 = 0x4, data_valid = 1 = 0x1, data = 0x0),
            [8] = (data_length = 4 = 0x4, data_valid = 1 = 0x1, data =
0x0001944C),
```

Sleep sets for the resource configuration to be used during low power mode

The example below shows the master 0 is in the sleep settings configuration (red text).

```
 gpRPMFWMaster = (
    [0] = 0x00039B4C -> (
      master_id = DAL_RPMFW_MASTER_0 = 0 = 0x0,
      selected_set = DAL_RPM_CONFIG_SET_SLEEP = 1 = 0x1,
      set = (
        [0] = 0x00039BD4,
        [1] = 0x0003A130 -> (resource = (
            [0] = (data_length = 64 = 0x40, data_valid = 0 = 0x0, data =
0x0003A3D0),
            …
            [4] = (data_length = 0 = 0x0, data_valid = 0 = 0x0, data = 0x0),
            [5] = (data_length = 4 = 0x4, data_valid = 0 = 0x1, data = 0x0),
            [6] = (data_length = 4 = 0x4, data_valid = 0 = 0x0, data = 0x0),
            [7] = (data_length = 4 = 0x4, data_valid = 0 = 0x0, data = 0x0),
```

7.3.6 Sets

Sets indicate the request (if any) for each resource. Basically, a set is an array indexed by enumeration values of type DAL_rpm_ResourceType, e.g., 5 is CXO.

- data_valid indicates if any request at all has been made on that master/set pair.
- Data is either the literal data (if data_size ≤ 4) or a pointer to the data buffer.
- Data requests are inherited from the active set requests when data_valid = 0 for the sleep set and data_valid = 1 for the active set.

Resource enumeration is defined in DAL_rpm_ResourceType in core\api\power\RPMTypes.h.

The following example shows the resource settings. The master 1 (modem processor) is in sleep configuration and resource type TCXO is voted to sleep by the modem processor.

```
gpRPMFWMaster = (
    [0] = 0x00038548,
    [1] = 0x00039040 -> (
      master_id = DAL_RPMFW_MASTER_1 = 0x1,
      selected_set = DAL_RPM_CONFIG_SET_SLEEP = 0x1,
      set = (
        [0] = 0x00039080,
        [1] = 0x000395DC -> (
          resource = (
            [0] = (data_length = 0x40, data_valid = 0x0, data =
0x0003987C),
```

```
            [1] = (data_length = 0x0C, data_valid = 0x1, data =
0x000398BC),
            [2] = (data_length = 0x8, data_valid = 0x1, data = 0x000398C8),
            [3] = (data_length = 0x4, data_valid = 0x0, data = 0x0),
            [4] = (data_length = 0x0, data_valid = 0x0, data = 0x0),
            [5] = (data_length = 0x4, data_valid = 0x1, data = 0x0),
```

Once the definition of the DAL_rpm_ResourceType is known, the licensee can obtain the configuration of certain resource settings, such as the voltage that the modem processor is voting for Vdd_Dig, e.g.:

```
v.v   (pm_npa_vreg_smps_type*) gpRPMFWMaster[1].set[1].resource[0x30].data
```

In this example:

- gpRPMFWMaster[1] – 1 references the modem
- set[1] – 1 references sleep set
- resource[0x30] – 0x30 → 48 references Vdd_Dig resource

In this example, the modem is voting for Vdd_Dig minimization (500 mv).

```
   (pm_npa_vreg_smps_type*)gpRPMFWMaster[1].set[1].resource[0x30].data =
        0x0003D8D8 -> (
                mvol = 500 = 0x01F4,
                ip = 0x0,
                fm = 0x0,
                pc = 0x0,
                pf_low = 0x0,
                pd = 0x0,
                ia = 0x0,
                freq = 0x0,
                freq_clk_src = 0x0,
                reserved = 0x0)
```

7.3.7 SPM variable

spm_subsystem_status indicates the status of the SPM for that master, e.g., going to sleep/ sleeping/waking/awake.

For multicore processors, spm_num_active_cores indicates how many cores are awake. For other processors, the number is 1 or 0 based on the single processor.

spm_pending_bringups indicate how many cores are waiting to be awakened.

The following example shows that two apps processors are in the Wakeup state and LPASS is in the Sleep state.

```
gpRPMFWMaster = (
    [0] = 0x00039B4C -> (
      master_id = DAL_RPMFW_MASTER_0 = 0 = 0x0,
      …,
      spm_subsystem_status = DAL_RPMFW_SPM_AWAKE = 0 = 0x0,
      spm_num_active_cores = 2 = 0x2,
      spm_pending_bringups = 0 = 0x0,
      triggers_timer = 0x000381C4,
      triggers_timer_expired = 0 = 0x0),
    [1] = 0x0003A68C,
    [2] = 0x0003B1CC -> (
      master_id = DAL_RPMFW_MASTER_2 = 2 = 0x2,
      …,
      spm_subsystem_status = DAL_RPMFW_SPM_SLEEPING = 2 = 0x2,
      spm_num_active_cores = 0 = 0x0,
      spm_pending_bringups = 0 = 0x0,
      triggers_timer = 0x0003E364,
      triggers_timer_expired = 0 = 0x0))
```

7.4 Adding RPM logs

The interface to the RPM external log is provided in core\power\rpm\inc\rpm_log.h. To add a log message:

1. #define a new log ID to use in rpm_log.h.
2. Wherever you need to log the message, #include “rpm_log.h” and call the following macro:

   `RPM_LOG_EVENT(YOUR_LOG_ID, your_data1, your_data2)`

   The number of data elements can be completely variable, but each argument has a cost and, therefore, <4 arguments is recommended.

   A best practice is to have the arguments in each position always mean the same thing for a given ID, i.e., the first argument is always the master ID, the second argument is always the resource, etc.

3. rpm_log.py can then be easily extended to parse the new ID as required:

   core\power\rpm\dal\scripts\rpm_log.py

   For example, in lpr_definition_uber.c go to sleep, the MPM register is dumped:

```
void dump_mpm(void)
{
  RPM_LOG_EVENT(MPM_VDD_CFG,    HWIO_IN(MPM_PMIC_VDD_CFG),
HWIO_IN(MPM_PMIC_VDD_CFG_1), HWIO_IN(MPM_PMIC_VDD_CFG_2),
HWIO_IN(MPM_PMIC_VDD_CFG_3));
  RPM_LOG_EVENT(MPM_INT_CONFIG, HWIO_IN(MPM_INT_POLARITY_1),
HWIO_IN(MPM_INT_POLARITY_2), HWIO_IN(MPM_DETECT_CTL_1),
HWIO_IN(MPM_DETECT_CTL_2));
  RPM_LOG_EVENT(MPM_POWER_CFG,  HWIO_IN(MPM_LOW_POWER_CFG),
HWIO_IN(MPM_LOW_POWER_STATUS), HWIO_IN(MPM_INT_EN_1),
HWIO_IN(MPM_INT_EN_2));
}
```

4. In rpm_log.py, add following code to extend the MPM register parsing:

```
class MPMVddCfg:
        __metaclass__ = Parser
        id = 0xF0
        def parse(self, data):
             return 'mpm_vdd_cfg (MPM_PMIC_VDD_CFG: 0x%08x)
(MPM_PMIC_VDD_CFG_1: 0x%08x) (MPM_PMIC_VDD_CFG_2: 0x%08x)
(MPM_PMIC_VDD_CFG_3: 0x%08x)' \
                          % (data[0], data[1], data[2], data[3])
class MPMIntConfig1:
        __metaclass__ = Parser
        id = 0xF4
        def parse(self, data):
             return 'mpm_int_config_1 (MPM_INT_EN_1: 0x%08x) (MPM_INT_EN_2:
0x%08x) (MPM_INT_POLARITY_1: 0x%08x) (MPM_INT_POLARITY_2: 0x%08x)' \
                          % (data[0], data[1], data[2], data[3])
class MPMPowerCfg:
        __metaclass__ = Parser
        id = 0xF2
        def parse(self, data):
             return 'mpm_power_cfg (MPM_LOW_POWER_CFG: 0x%08x)
(MPM_CXO_POWER_RAMPUP_TIME: 0x%08x) (MPM_PXO_POWER_RAMPUP_TIME: 0x%08x)' \
                          % (data[0], data[1], data[2])
```

7.5 Enabling Tramp logs

By default, the Tramp logs are disabled in RPM. In the MSM8960 default software, the RPM Tramp was not enabled. The following instructions show how to enable the RPM Tramp log.

1. Add the red lines into rpm_proc\core\systemdrivers\tramp\build\SConscript.

```
elif env.has_key('RPM_IMAGE'):
  env.Append(CPPDEFINES = ['FEATURE_TRAMP_LOG'])
  TRAMP_SOURCES += [
    '${BUILDPATH}/platform/qgic/tramp_qgic_log.c',
  ]
```

2. Because of the limited coderam, the number of log entries has to be reduced to a range of 50 to 200 in rpm_proc\core\systemdrivers\tramp\src\platform\qgic\tramp_qgic_log.c.

   In the MSM8960 1045 RPM release, 100 entries seems to be appropriate; a larger number than that causes booting issues.

```
#define MAX_TRAMP_LOG_ENTRIES 100
```

3. T32 command to view the RPM tramp log:

```
v.v tramp_log
```

7.6 Stopping RPM log from the apps kernel

On the RPM side, register rpm_log_go_silent with TRAMP_MSS_SW_GP_LOW_IRQ. When the IRQ occurs, the function will be called to stop the RPM logging.

```
tramp_register_isr(TRAMP_MSS_SW_GP_LOW_IRQ, rpm_log_go_silent, 1);
```

Also, modify the corresponding QGIC ID (from +27 to +25) in rpm_proc\core\systemdrivers\tramp\src\platform\qgic\8960\tramp_qgic_bsp_rpm.c.

```
248c248
<     TRAMP_QGIC_BSP_SPI_OFFSET +27,                  /* QGIC id */
---
>     TRAMP_QGIC_BSP_SPI_OFFSET +25,                  /* QGIC id */
```

Whenever the apps kernel wants to stop the RPM logging, it triggers the interrupt using the following command:

```
writel(0x1, MSM_APCS_GCC_BASE+0x8);
```

7.7 Debugging the RPM MPM

MPM is a hardware state machine block with the following states:

```
1.IDLE:      MPM in Idle and can accept the command
             transit on HWIO_OUT(RPM_SW_DONE, HWIO_RMSK(RPM_SW_DONE));
2.FREEZE_IO: Assert Freeze/Clamp
3.POWER_OFF: Send SSBI cmds and MSM in sleep state
4.POWER_ON:  Send SSBI cmds and warm up the power rails
5.POWER_UP:  Send SSBI cmds and warm up the power rails
6.BOOT:      De-assert preclamp, deassert RESET to RPM to warm boot.
             Once in this stage, RPM starts to execute code.
7.UNCLAMP:   Deassert Clamp, transit on MPM_HWIO_OUT(HARDWARE_RESTORED, 1);
8.UNFREEZE:  Deassert Freeze
1.IDLE
```

rob_mark_event has the record of the MPM state transition as shown below; the customer can check event_data[4] and event_data[5] to see if the MPM status machine has correctly executed.

```
typedef enum
{
  ROB_EVENT_VDD_MIN_ENTER,
  ROB_EVENT_VDD_MIN_EXIT,
  ROB_EVENT_XO_SD_ENTER,
  ROB_EVENT_XO_SD_EXIT,
  ROB_EVENT_MPM_PRE_SW_DONE,
  ROB_EVENT_MPM_POST_SW_DONE,
  ROB_EVENT_RPM_HALT_ENTER,
  ROB_EVENT_RPM_HALT_EXIT,
  ROB_EVENT_RPM_HALT_PRE_CLKOFF,
  ROB_EVENT_RPM_HALT_POST_CLKOFF,

  ROB_NUM_EVENTS
} rob_event_type;
```

If not, dump the following registers to global variables (or into RPM log) to find out what happened.

```
LOW_POWER_STATUS
MPM_INT_EN_1
MPM_INT_STATUS_1
MPM_INT_EN_2
MPM_INT_STATUS_2
```

7.8 Debugging the SPM

7.8.1 SPM introduction

SPM is a subsystem power manager. Its purpose is to manage local, power-related resources for a subsystem:

- Independent (of any other processor outside subsystem) power collapse and restore for a processor (which has an independent voltage rail or is a tool with Headswitch or Footswitch) in a subsystem.
- Functional VDD minimization, i.e., minimal voltage at which the subsystem is still operational, but at a very low frequency (performance).
- Dormant VDD minimization, i.e., retention mode; the system does not require a reset, but it is nonfunctional.
- Voltage control is done through handshaking with PMIC arbiter (generic/physical PMIC protocol independent).
- Turn on/off clamps around processor core, caches (memories, e.g., L2).
- Turn on/off foot switches/header switches for GDHS macros (including memories). For memories, the header switch (pmos) on memory arrays on Vdd_mem, peripheries have footer switch (nmos) on Vdd_dig.
- Initiating a reset or with a reset from the clock controller.
- Handshake with the clock controller or turning on/off clocks. The clock controller can be a local clock controller or a global clock controller. Note that RPM can also take care of clocks based on a message from the processor and then after a handshake from SPM.

7.8.2 Triggering SPM

The SPM status machine is triggered after RPM sets up the shared resource and triggers the internal GPO interrupt via HWIO_OUTI(RPM_GPO_WDCLRn, idx, pin) in RPMFW_SetBringupAck(). If this command has been executed but the master does not bring up as expected, then SPM needs to be debugged.

7.8.3 Debugging SPM

The SPM status register can be checked via T32 or log in the RPM or TrustZone when the issue occurs.

MSM8960 example for the SPM status register:

```
0x0201200C L2 SPM STS
0x0208900C Core 0 SPM STS
0x0209900C Core 1 SPM STS

0x02012020 L2 SPM CTL
0x02089020 Core 0 SPM CTL
0x02099020 Core 1 SPM CTL
```

SPM_CMD_ADDR in the STS register indicates the last SPM command executed.

SPM_START_ADDR in the SAW2_SPM_CTL register indicates the start of the SPM command.

Example of a typical SPM command that executes 00-0x22 command sequence and SPM_CMD_ADDR=0x22 means:

```
0x0208900C : 0x00003022 (SAW2_STS_0)
                            ---------------------------
                               [15] SHTDWN_REQ        0
                               [14] SHTDWN_ACK        0
                               [13] BRNGUP_REQ        1
                               [12] BRNGUP_ACK        1
                              [11:10] PMIC_STATE      0
                                [9:8] RPM_STATE       0
                                  [7] AVS_STATE       0
                                 [6:0] SPM_CMD_ADDR   0x22
                            ---------------------------

0x02089020 : 0x00000001 (SAW2_SPM_CTL)
                            ---------------------------
                             [10:4] SPM_START_ADDR 0x00
                                [3] ISAR              0
                              [2:1] WAKEUP_CONFIG     0
                                [0] SPM_EN            1
                            ---------------------------
```

SPM0 has correctly executed the preset command.

Example of a typical SPM command sequence:

```
------------------- Summary of SPM0----------------------
      Last SPM command executed: [22]
Start address for SPM sequence: [00]
--------------------------------------------------

------------------- Summary of SPM1 ----------------------
      Last SPM command executed: [25]
Start address for SPM sequence: [18]
--------------------------------------------------

------------------- Summary ----------------------
      Last SPM command executed: [19]
Start address for SPM sequence: [14]
--------------------------------------------------
```

7.8.4 SPM command sequence details

```
0x02089080 : CMD0:0x03 CMD1:0x0F CMD2:0x0F CMD3:0x0F
  [00] 0x03  SLP with no RPM handshake
  [01] 0x0F  End of program
  [02] 0x0F  End of program
  [03] 0x0F  End of program

0x02089084 : CMD0:0x00 CMD1:0x24 CMD2:0x54 CMD3:0x10
  [04] 0x00  Toggle output 0, wait 0 ticks (clockGate)
  [05] 0x24  Toggle output 2, wait 2 ticks (clampCPU)
  [06] 0x54  Toggle output 5, wait 2 ticks (clampPLL)
  [07] 0x10  Toggle output 1, wait 0 ticks (duptag_HS)

0x02089088 : CMD0:0x09 CMD1:0x03 CMD2:0x01 CMD3:0x10
  [08] 0x09  PMIC operation: off
  [09] 0x03  SLP with no RPM handshake
  [0A] 0x01  PMIC operation: normal
  [0B] 0x10  Toggle output 1, wait 0 ticks (duptag_HS)

0x0208908C : CMD0:0x54 CMD1:0x30 CMD2:0x0C CMD3:0x24
  [0C] 0x54  Toggle output 5, wait 2 ticks (clampPLL)
  [0D] 0x30  Toggle output 3, wait 0 ticks (corePOR)
  [0E] 0x0C  Toggle output 0, wait 32 ticks (clockGate)
  [0F] 0x24  Toggle output 2, wait 2 ticks (clampCPU)

0x02089090 : CMD0:0x30 CMD1:0x0F CMD2:0x0F CMD3:0x0F
  [10] 0x30  Toggle output 3, wait 0 ticks (corePOR)
  [11] 0x0F  End of program
  [12] 0x0F  End of program
  [13] 0x0F  End of program

0x02089094 : CMD0:0x00 CMD1:0x24 CMD2:0x54 CMD3:0x10
  [14] 0x00  Toggle output 0, wait 0 ticks (clockGate)
  [15] 0x24  Toggle output 2, wait 2 ticks (clampCPU)
  [16] 0x54  Toggle output 5, wait 2 ticks (clampPLL)
  [17] 0x10  Toggle output 1, wait 0 ticks (duptag_HS)

0x02089098 : CMD0:0x09 CMD1:0x07 CMD2:0x01 CMD3:0x0B
  [18] 0x09  PMIC operation: off
  [19] 0x07  SLP with RPM handshake
  [1A] 0x01  PMIC operation: normal
  [1B] 0x0B  Wait for event [0] (L2_ready)
```

```
0x0208909C : CMD0:0x10 CMD1:0x54 CMD2:0x30 CMD3:0x0C
  [1C] 0x10  Toggle output 1, wait 0 ticks (duptag_HS)
  [1D] 0x54  Toggle output 5, wait 2 ticks (clampPLL)
  [1E] 0x30  Toggle output 3, wait 0 ticks (corePOR)
  [1F] 0x0C  Toggle output 0, wait 32 ticks (clockGate)

0x020890A0 : CMD0:0x24 CMD1:0x30 CMD2:0x0F CMD3:0x0F
  [20] 0x24  Toggle output 2, wait 2 ticks (clampCPU)
  [21] 0x30  Toggle output 3, wait 0 ticks (corePOR)
  [22] 0x0F  End of program
  [23] 0x0F  End of program

0x020890A4 : CMD0:0x0F CMD1:0x0F CMD2:0x0F CMD3:0x0F
  [24] 0x0F  End of program
  [25] 0x0F  End of program
  [26] 0x0F  End of program
  [27] 0x0F  End of program
```

7.9 Debugging why the system cannot sleep

To debug why the system cannot go to sleep, set the breakpoint at the xo_shutdown_exit() and retrieve the Ulog. The following example shows that GPIO 123 is the culprit that keeps the RPM from going to XO shutdown:

```
84.007935: rpm_xo_shutdown_enter (count: 367)
84.010345: rpm_xo_shutdown_exit (total duration: 28513)
84.010376: rpm_mpm_wakeup_ints (interrupts: "[timetick] | [gp crci trigger / gpio 123]")
```

7.10 Debugging the modem late wake-up issue

The modem had an err_fatal because it missed the paging time slot. By looking at the gpRPMMaster record, it looked like RPM woke up the modem late by 4.3 ms (Task::end_ - Task::deadline_ = 138 sclks = 4.3 ms). Further investigation indicated the root cause was the frequent and short apps sleep/wake-up overlaps with the modem wake-up timeline, which delayed the modem wake-up.

```
gpRPMFWMaster = (
    0x0003C568 -> (
      master_id = DAL_RPMFW_MASTER_0 = 0x0,
      changer = 0x000420AC -> (
        Task::deadline_ = 0x00000001005C0FF7,
        Task::start_    = 0x00000001005C0E6F,
        Task::end_      = 0x00000001005C0FE9,
```

```
0x0003CC4C -> (
  master_id = DAL_RPMFW_MASTER_1 = 0x1,
    Task::deadline_ = 0x00000001005C0FCA,
    Task::start_    = 0x00000001005C0F6F,
    Task::end_      = 0x00000001005C1054,
```

7.11 Allowing RPM to enter SWFI and XO shutdown

When the RPM is configured to allow XO shutdown and halt, ARM7 halts the daisy chain and the entire system will halt. To prevent this, by default be_gentle_to_the_daisy_chain is set to 1. To enable RPM low power modes, disconnect all T32s except for the RPM T32.

To use the RPM T32 set:

```
v be_gentle_to_the_daisy_chain=0
```

7.12 Disabling RPM halt for debug purposes

Legacy ARM cores (ARM7, ARM11) resynchronize the JTAG RTCK signal to their respective processor clocks. The result is that halting the RPM clock, e.g., during sleep, causes a “no RTCK” condition on the JTAG daisy chain. Therefore, if reliable JTAG is required for any master, the RPM cannot halt, as the RPM is on the daisy chain.

Two methods of working around this are built into the system:

1. In a command window, connect to the Linux side and enter:

   ```
   echo 0 > /sys/module/rpm_resources/enable_low_power/rpm_cpu
   ```

   This method applies only to the MSM8660 and MSM8960 chipsets.

2. Create an /nv/items_files/sleep/core0/sleep_config.ini file.
   a. This file must be placed at /nv/items_files/sleep/core0/sleep_config.ini (using EFS Explorer):
   b. Contents of file:

   ```
   [rpm.halt]
   disable=1
   ```

7.13 Disabling the RPM watchdog

Licensees can make a change to core\power\rpm\fakedrivers\dog.c to comment the dog-related register settings to disable the RPM watchdog.

8 RPM Customization and Highlights

8.1 DDR driver

The boot loader DDR driver must sync with the RPM-side DDR driver. All the RPM DDR drivers share the same DDR device table with the boot loader. The boot loader detects and chooses the best-matched parameter settings during boot time and saves these parameters to system IMEM. When the RPM starts run, it retrieves the ddr_device_table[] from the system IMEM directly, so no DDR change is required for the RPM.

8.2 NPA driver

The system FABRIC is initialized by clkrgm_rpm_bsp_data[] in the code core/systemdrivers/clkregim/src/proc/rpm/<MSMplatform>/clkrgm_rpm_bsp.c. The FABRIC clock is inited to the maximum configuration during boot.

Licensees can change the global variable based on their requirements.

8.3 MPM driver

The MPM core/power/mpm/hal/bsp/source/<MSMplatform>/BSPmpm.c include the settings of the XO warm-up timer settings and other related register settings.

Based on the selected XO, licensees can tune the XO warm-up timer here.

8.4 Cx/Mx managment

Cx (Vdd_DIG) and Mx(Vdd_MEM) scaling is managed by RPM. Table 8-1 lists the status for the different A-family platforms.

Table 8-1 Status for A-family platforms

	MSM8960/8930/8064 (S3,LDO24)	MDM9x15 (S1, LDO9)
Vdd Cx sleep	0.65	0.65
Vdd Mx sleep	0.75	0.75
Vdd Cx awake	1.15	1.15
Vdd Mx awake	1.15	1.15
Vdd Cx SVS	0.95	0.95
Vdd Mx SVS	1.05	1.05
Vdd Cx Nominal	1.05	1.05
Vdd Mx Nominal	1.05	1.05
Vdd Cx Turbo	1.15	1.15

	MSM8960/8930/8064 (S3,LDO24)	MDM9x15 (S1, LDO9)
Vdd Mx Turbo	1.15	1.15
VDD Cx force SVS	NA	0.95
VDD Cx Force SVS	NA	0.95

Vdd_Mem/Dig scaling and interfaces:

- AP MP to RPM – PMIC driver on AP MP aggregates load requests and calls into RPM register interface
- LPASS Q6 to RPM – Does not support PMIC NPA; calls into RPM driver to control Vdd_Mem voltage rail
- Modem Q6 to RPM – Uses PMIC NPA to request Vdd_Mem/Dig scaling
- RPM PMIC driver – Sees no difference between calls from MARM PMIC NPA vs Linux PMIC driver (non-NPA)

8.5 Clock management

RPM also manage the XO. Table 8-2 lists the platforms in the A-family and the different XO/sleep clocks.

Table 8-2 A-family XO/sleep clocks

Platforms	XO/sleep clocks	Comments
APQ8064	LPXO 27 MHZ	Both PXO and CXO can be shutdown.
	TCXO 19.2 MHZ	
	Sleep clock (32 K)	
MSM8960	LPXO 27 MHZ	LPXO can be turned off in VDD mins, TCXO buffer will turn it off; LPXO is managed by MPM and TCXO is managed by RPM software.
	TCXO 19.2 MHZ	
MSM8930	Same as MSM8960	Same as MSM8960.
MDM9x15	CXO (19.2 MHZ)	No LPXO, CXO can be shut down.
fusion3 (APQ8064 + MDM9x15)	LPXO 27 MHZ	MDM9x15 and APQ8064 can go to XO shutdown and VDD min individually. For fusion APQ8064, when there is an active voice call, the APQ8064 needs to provide the clock via PXO for the SLIM bus between APQ, MDM, and the voice codec. In this case, only CXO can be shutdown.
	TCXO 19.2 MHZ	
	Sleep clock (32 K)	
	CXO (19.2 MHZ)	

8.6 Latency recalculation

There is support for dynamic calculation of the sleep mode latencies. A recurring issue has been that any change in the sleep path changes the timing, and therefore, the hard-coded latency numbers need to be changed. Instead, RPM should profile itself during sleep paths and come up with estimates that are based on actual runtime execution. For the platforms that do not have this functionality, customers need to add the corresponding delay in the vdd_min_latency() and xo_shutdown_latency() once latency has been added in the enter/exit of the lower power mode.

A Additional Information

A.1 NPA

Node Power Architecture (NPA) is the framework for managing dynamic resources with the goals of system transparency and interface consistency. It is a mechanism used by resource owners to expose their resource, but all resources are still owned by their respective teams.

Each NPA log entry contains:

- Handle – Unique identifier to be used to see when requests start and complete
- Client – Master or RPM-based resource making the request
- Request – Resource state being requested
- Resource – Resource to which the client would like the request to be applied

Entries in the log are paired.

The example below shows that the modem requests the apps FABRIC run on the 155 MHz and active state in the request. The complete message shows the apps FABRIC runs at 174 MHz after RPM aggregates the request.

```
0x4D351C: npa_issue_required_request (handle: 0x0003D7B0) (client: "Modem")
(request: 155000) (resource: "/clk/bus/afab")
0x4D3524:    request complete (handle: 0x0003D7B0) (sequence: 0x001A6000)
(request state:155000) (active state:174667)
```

Table A-1 lists the NPA resource being used; N means the node is not used in the platform.

Table A-1 NPA resources

NPA nodes						
Node name	**Resource name**	**Owner proc**	**Notes**	**MSM8660**	**MSM8960**	**MDM9x15**
/node/xo/cxo	/xo/cxo	RPM		Y	Y	Y
/node/xo/pxo	/xo/pxo	RPM		Y	Y	N
/node/xo/mxo	/xo/mxo	RPM		Y	Y	N
/node/clk/cpu	/clk/cpu	Each		Y	Y	N
/node/clk/bus/afab	/clk/bus/afab	RPM	Apps FABRIC	Y	Y	
/node/clk/bus/sfab (For MSM8960 only)	/clk/bus/sfab	RPM	System FABRIC, isosync to apps FABRIC	Y	Y	Y

NPA nodes						
Node name	**Resource name**	**Owner proc**	**Notes**	**MSM8660**	**MSM8960**	**MDM9x15**
/node/clk/bus/mmfab	/clk/bus/mmfab	RPM	MM FABRIC	Y	Y	N
	/clk/mem/smi	RPM	SMI is 2x the MM FABRIC generally	Y	N	N
/node/clk/bus/dfab	/clk/bus/dfab	Scorpion	Daytona FABRIC	Y	Y	Y
/node/clk/bus/sfpb	/clk/bus/sfpb	RPM	System FPB	Y	Y	Y
/node/clk/bus/cfpb	/clk/bus/cfpb	RPM	Chip FPB	Y	Y	Y
/node/clk/bus/mmfpb	/clk/bus/mmfpb	Scorpion		Y	Y	N
/node/clk/mem/ebi1	/clk/mem/ebi1	RPM		Y	Y	Y

A.2 Common sleep nodes and LPR

This section describes the common sleep nodes and LPR.

A.2.1 CXO/PXO

This node is used to control the state of the CXO (on the modem) or the PXO (on the Hexagon or apps) clock. Clients can register with this node to issue and cancel requests for the CXO/PXO clock to be in a certain state. The following are the states that are supported by this node:

- Active – Client requires that the CXO/PXO clock stay active
- Off – Clients are OK with turning the CXO/PXO clock off during idle

This node registers with the sleep task as an LPR, since it can only be put into its low power mode when the CPU is halted. At this time, only one low power mode is supported by this node:

- Off – CXO/PXO clock will be turned off during idle if conditions allow

If a client wants to affect this node, the client should use the name /xo/cxo or /xo/pxo.

A.2.2 VDD Dig

This node is used to control the state of the systemwide digital power rail during sleep. Clients can register with this node to issue and cancel requests for the rail to be in a certain state during sleep. Clients can only make requests through this node that affect whether the node is allowed to go to VDD minimization. If a client would like to request that VDD dig go to a certain voltage during active modes, they must go through the PMIC node.

NOTE: For interrupt latching, the interrupt controller driver automatically takes care of ensuring that the proper voltage level is requested in order to detect the necessary interrupts during sleep. This node operates in units of millivolts.

This node registers with the sleep task as an LPR, since it can only be put into its low power mode when the CPU is halted. The following low power modes are supported by this node:

- Minimization Level 0 – Dig power rail is lowered to its lowest possible retention voltage during Idle if conditions allow
- Minimization Level 1 –Dig power rail is lowered to its retention voltage, without going below 0.75 V, during Idle if conditions allow

If a client wants to affect this node, the client should use the name /rail/vdd_dig.

A.2.3 VDD Mem

This node is used to control the state of the memory power rail. Clients can register with this node to issue and cancel requests for the rail to be in a certain state. Currently, clients can only make requests through this node that affect whether the node is allowed to go to VDD minimization. If a client would like to request that VDD mem go to a certain voltage, they must go through the PMIC node. The following are the states that are supported by this node:

- Active – Client requires that the mem rail stay at its active voltage during Sleep
- Retention – Client is OK with the mem rail going to its retention voltage during Idle

This node registers with the sleep task as an LPR, since it can only be put into its low power mode when the CPU is halted. The following low power mode is supported by this mode:

- Retention – Mem power rail is lowered to its lowest possible retention voltages during Idle if conditions allow

If a client wants to affect this node, the client should use the name /rail/vdd_mem.

A.2.4 Core VDD

This node is used to control the state of the core power rail. The states that the core power rails can enter varies per processor:

- Modem – Foot-switched
- Apps core 0 – Off, minimized
- Apps core 1 – Off, minimized
- Hexagon – Off
- RPM – No low power states (always on)

Clients can register with this node to issue and cancel requests for the core rail to be in a certain state. The following are the states that are supported by this node:

- Active – Client requires that the core rail stay powered on, or minimized (if applicable to the processor)
- Off – Client is OK with the core rail being gated off or powered off (depending on the processor) during Idle

This node registers with the sleep task as an LPR, since it can only be put into its low power mode when the CPU is halted. The following low power mode is supported by this mode:

- Off – The core rail will be foot-switched off or powered off during idle if conditions allow. When the rail is switched on again, the software will start executing at the reset vector and will go through warm boot to restore the processor back to its previous context.

If a client wants to affect this node, the client should use the name /core/cpu/vdd.