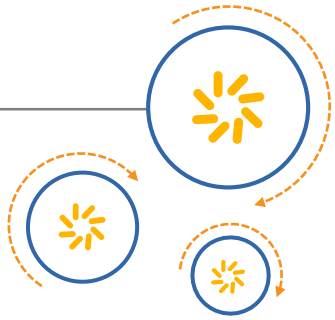
Qualcomm Technologies, Inc.



Resource Power Manager (RPM.BF)

User Guide

80-NA157-15 F

August 4, 2015

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

Revision history

Revision	Date	Description
A	July 2012	Initial release
B	October 2012	Numerous changes were made to this document; it should be read in its entirety
C	November 2013	Added MDM9x35
D	April 2014	Numerous changes were made to this document; it should be read in its entirety
E	April 2014	Updated Sections 1.1 and 1.2
F	August 2015	Numerous changes were made to this document; it should be read in its entirety

Contents

Figures

Tables

1 Introduction

1.1 Purpose

This document describes the Resource Power Manager (RPM) for chipsets that use the RPM.BF subsystem.

This document is intended for OEMs, who need to understand details about RPM.BF.

1.2 Conventions

Function declarations, function names, type declarations, attributes, and code samples appear in a different font, for example, `#include`.

Code variables appear in angle brackets, for example, `<number>`.

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

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

If you are viewing this document using a color monitor, or if you print this document to a color printer, **red 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.

1.3 Technical assistance

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

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

2 RPM overview

2.1 Hardware overview

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

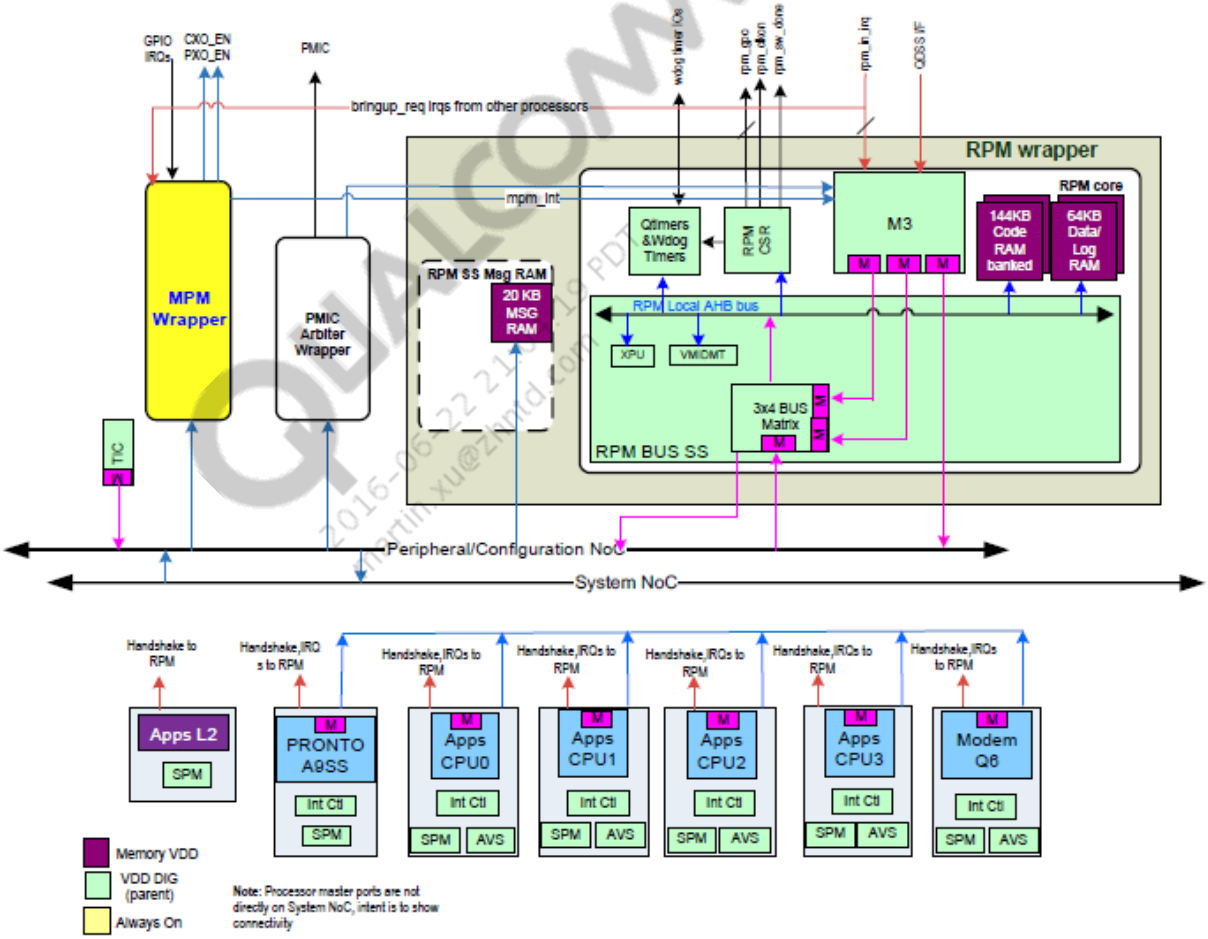


Figure 2-1 RPM example block diagram

2.1.1 RPM processor

The RPM uses a Harvard architecture based Cortex-M3 as its ARM processor. The processor is a low-power processor that features low gate count, low interrupt latency, and low-cost debug. The processor implements the ARMv7-M architecture.

The RPM core consists of:

- Cortex-M3 processor with integrated NVIC
- 144 KB multibank code RAM
- 64 KB multibank data/log RAM
- 24 KB message RAM

The RPM processor has a native Advanced High-performance Bus (AHB) Lite interface and built-in ETM/ITM and DAP support. The processor target frequency is 150 MHz. The Cortex-M3 supports a native SWFI instruction, which allows the processor to turn off its own clock when the clock is not needed.

NOTE: For MSM8916/MSM8939/MSM8929/MSM8909 RPM is not the boot processor. APSS acts as the boot processor.

2.1.2 AHB

The RPM AHB is synchronous to the RPM processor. The RPM AHB has Cortex-M3 as the default master for the bus. The RPM AHB allows byte, half-word, and word accesses for the code and data/log RAM. The RPM AHB allows only word accesses for CSR registers.

2.1.3 Code RAM

RPM code RAM serves as an instruction memory for the RPM processor. On boot-up, the RPM software image is downloaded to code RAM. The 3x2 matrix system access port allows the RPM image to be downloaded and authenticated by the Apps processor. The M3 then jumps to the code RAM and start executing the software image. The code RAM also houses the primary warm boot handler, which is the first piece of code to execute on a warm boot-up.

RPM code RAM sits at address 0x0000_0000 in the RPM address map. The RPM instantiates a 144 KB code RAM. 128 KB is used for RPM code ram and remaining 16 KB is used for common DDR libraries for SBL and RPM. The RPM code RAM operates at a clock frequency that is synchronous to the RPM AHB.

2.1.4 Data/log RAM

RPM data/log RAM serves as data and log memory for RPM processor. Data RAM is used for stack pointer for Interrupt vector table. Also it can be used for storing the log after debug.

The RPM data RAM sits at address 0x0009_0000 in the RPM address map.The RPM instantiates a 64 KB data/log RAM. The RPM data RAM operates at a clock frequency that is synchronous to the RPM AHB.

2.1.5 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. See Table 2-1 for examples.

Table 2-1 Example RPM messaging masters

Chipset	MSM8916/MSM8939/MSM8909
Messaging masters	▪ APSS (ARM) ▪ Modem ▪ WCNSS

2.1.6 Interrupt controller

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

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

2.1.7 Timers

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

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

2.1.8 CSR

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

2.1.9 Page select scheme

Some of the chipsets require having DDR memories above the traditional 4 GB range. Inherently, Cortex- M3 can see only a 4 GB space, because it supports only 32-bit addresses. To see the system memory above 4 GB of the chipsets that supports Large Physical Address Extension (LPAE), RPM extends the Cortex-M3 32-bit address from system interface to 36-bits with a page select scheme. With the help of the page select scheme, the Cortex-M3 system interface can see any 1 GB range of the entire system memory map at any given time.

The page select scheme slides this 1 GB DDR portal range across the entire 4-GB or 64-GB system memory based on the chipset that supports LPAE (Large Physical Address Extension) or non-LPAE. To support this scheme, a 6-bit RPM_PAGE_SELECT register is added for RPM. This register helps hardware in mapping the 32-bit address from the Cortex-M3 system interface into a 36-bit address.

2.2 RPM software overview

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

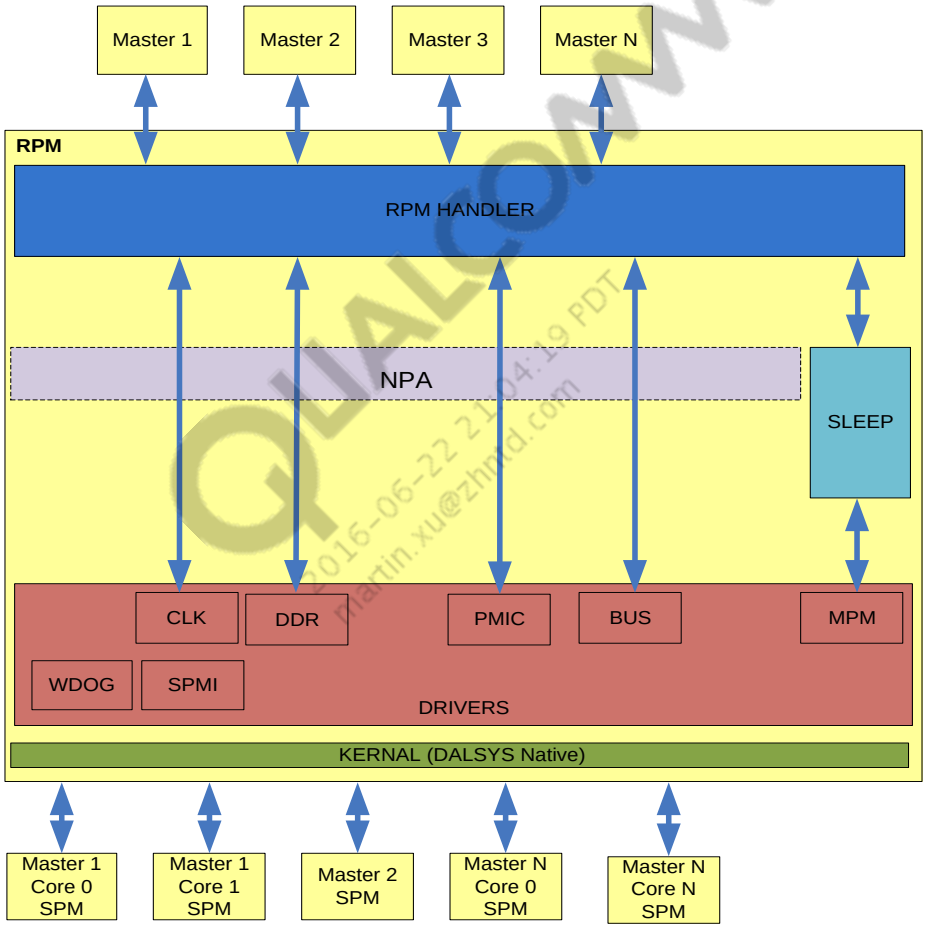


Figure 2-2 RPM software topology

2.2.1 Kernel

The kernel for the RPM supports:

- Interrupts
- Intlock, priorities, and configuration
- Busy waits
- Timers
- SWFI

2.2.2 RPM handler

The RPM handler abstracts the RPM messages from sub systems. This driver handles the 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 have to register with RPM handler to request notifications. Upon receiving this notification, the drivers perform arbitration between the new request and previous requests from other masters. Based on the arbitration results, the driver determines how to modify the hardware resource.

2.2.3.1 NPA

NPA driver uses the NPA to represent resources controlled by the driver (for example, /sleep/uber, /clk/bimc). NPA is a generic framework that allows nodes to represent resources. Each node has clients and is responsible for aggregating workload requirements on their resources while optimizing power usage. The nodes make up a distributed graph, allowing one node to be a client of another node.

2.2.3.2 Clock driver

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

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

2.2.3.3 Bus arbitration driver

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

The RPM bus arbiter driver takes bus arbiter settings as requests from the different masters in the system and aggregates them to represent the frequency-independent system settings. 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 multiple parts. One part resides on each master and the other part resides on the RPM.

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

2.2.3.5 WDOG driver

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

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

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

The RPM watchdog driver cannot reset the system.

2.2.3.6 MPM driver

The MPM driver is used to program the MPM hardware block during system wide sleep. This driver resides on the RPM and is responsible for programming the MPM to do the following:

- Vdd_Dig retention – Puts the system-wide power rail in the Retention state
- Vdd_Mem retention – Puts the system-wide power rail in the Retention state
- CXO shutdown – Turns off CXO

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

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

3 RPM build instructions

See the appropriate software user manual or build and integration documentation for detailed instructions of building the RPM subsystem. Additional information may also be found in the software release notes.

NOTE: To build the RPM, install ARM Compiler Tools 5.01 update 3 (build 94).

4 RPM scheduling

4.1 Task

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

- 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 is preempted:

```
void execute(volatile bool &preempt, uint64_t stop_time);
```

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

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

4.2 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 is kept in this binary heap.

```
TaskHeap immediateQ_;
```

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

```
TaskList scheduledQ_;
```

- Preemption flag – The dispatcher checks this flag periodically to see if it should stop current work prematurely 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;
```

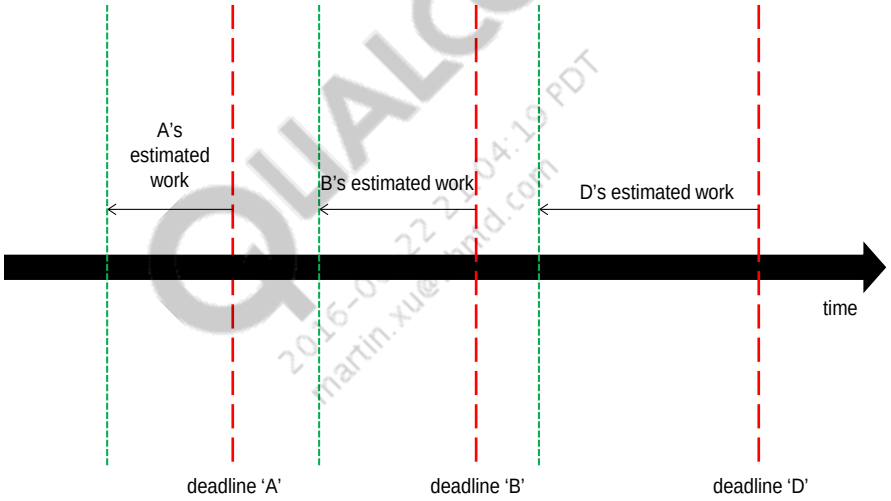
- Return the estimated duration of the scheduler's next run:

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

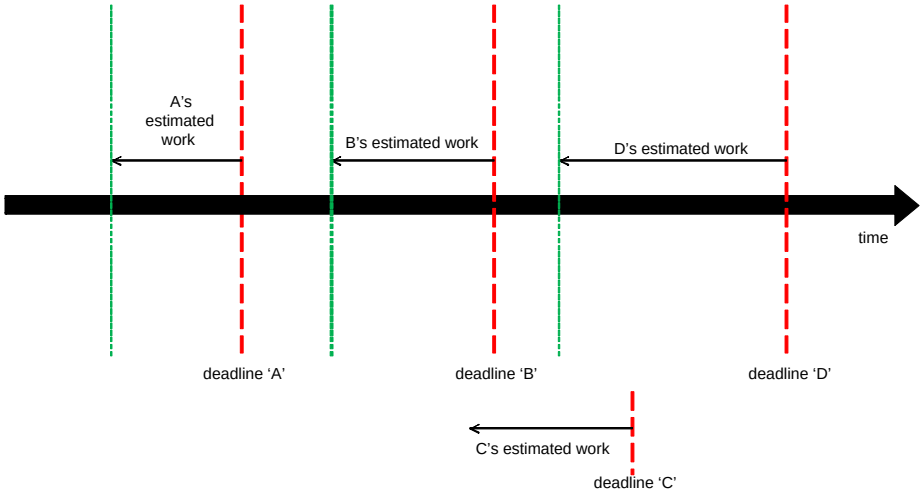
4.3 Schedule collision – Stack up example

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.

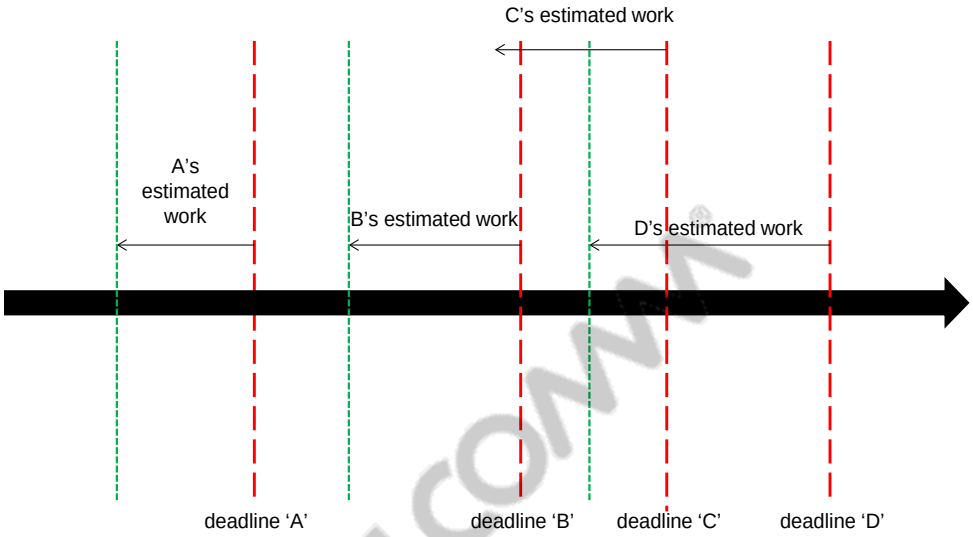
1. Task A, task B, and task D are on the schedule, each with their own deadline – deadline A, deadline B, and deadline D.



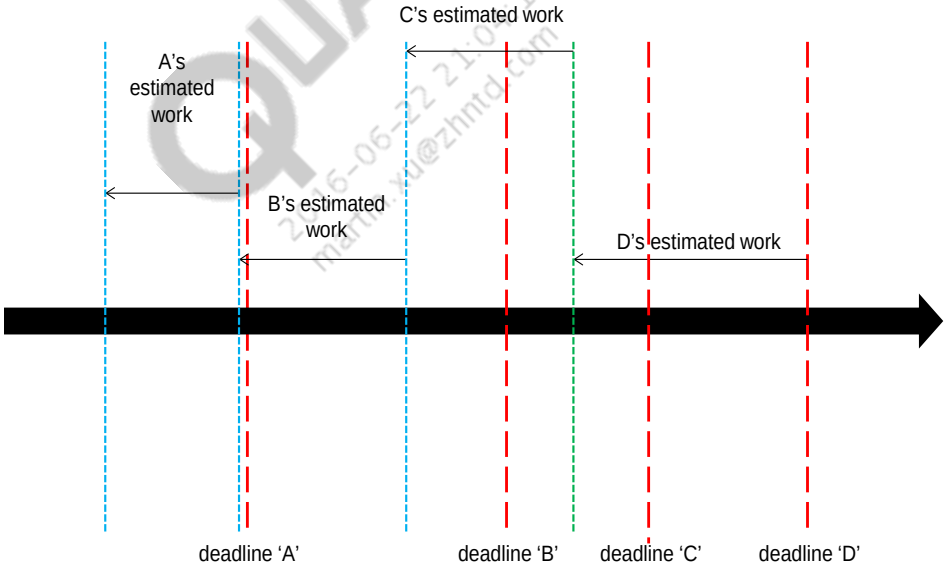
2. A new task, task C, comes in with a new deadline, deadline C.



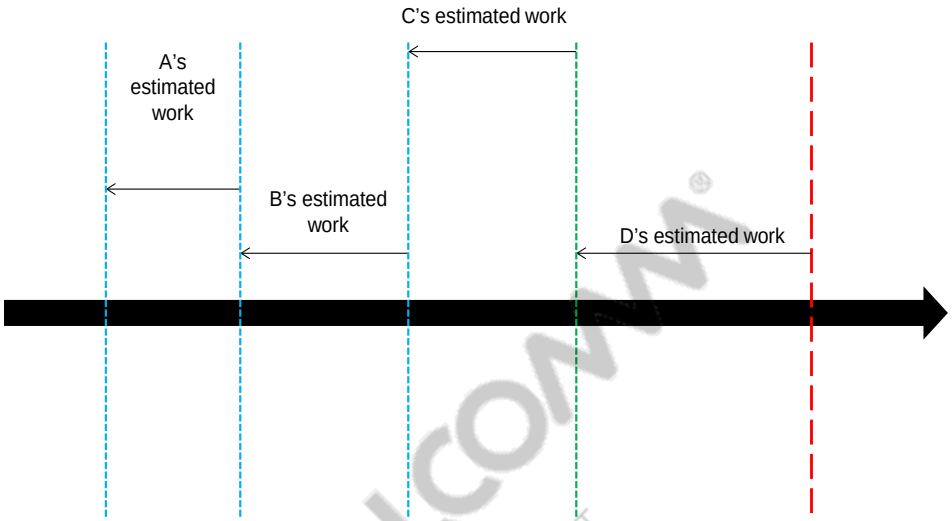
3. Task C is inserted; based on the deadline, task C now conflicts with task D in the queue, so the execution time for task C is pushed out.



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



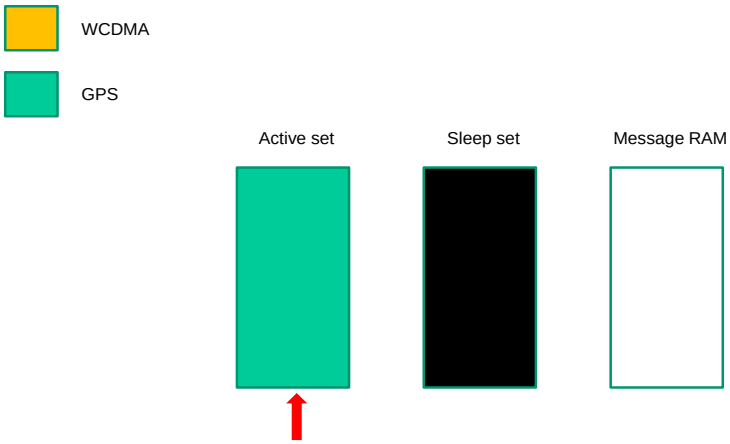
4. Task D's original schedule remains unchanged. The rest of the schedule needed modifications. This is called stack up when the problem occurs, and the resolution is called schedule fix up.



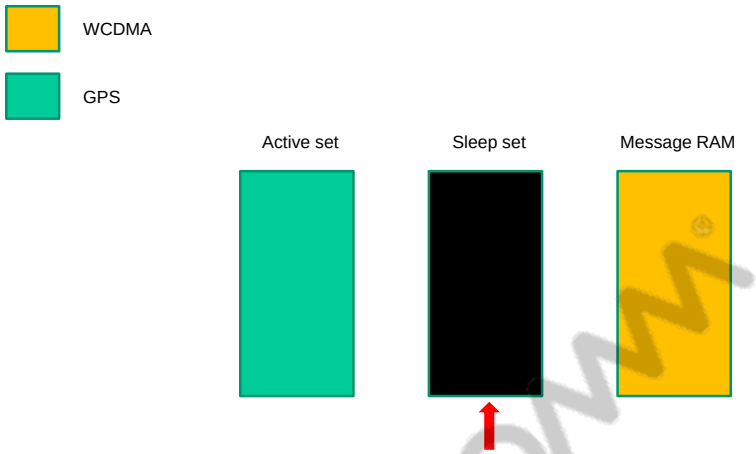
4.4 Concurrency – Next awake set

Wake-up information from the modem goes from time to deadline; missing the deadline is considered a 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.

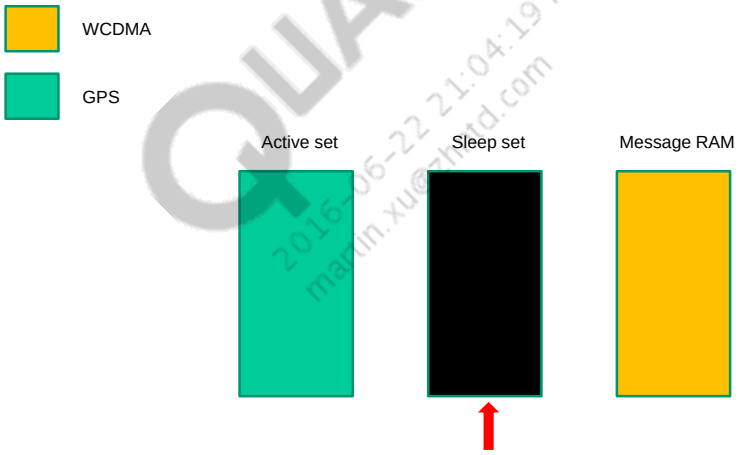
1. When going to sleep, a new sleep set is written and sent to the RPM. The settings for the next wake-up are then left as an unsent request in message RAM.



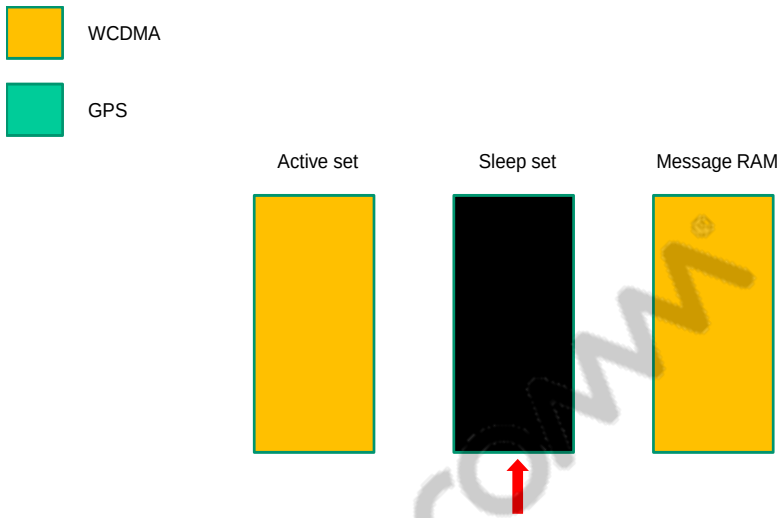
2. Once the RPM detects a sleep transition on a master, it checks to see if there is an unsent message in message RAM.



3. If there is an unsent message, it schedules 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.



4.5 Schedule code layout

- Request handling algorithms:
  - Rpm_handler.cpp
  - Rpm_handler.h
- Set transition algorithms:
  - Rpm_set_changer.cpp
  - Rpm_set_changer.h
- Schedule code – Code that is responsible for keeping track of how long each resource will take to transition:
  - Rpm_estimator.cpp
  - Rpm_estimator.h
- Schedule framework:
  - rpm_sched.cpp
  - rpm_task.cpp
  - task_heap.cpp
  - task_list.cpp

4.6 Example of scheduling with preemption

Figure 4-1 shows an example of scheduling with preemption.

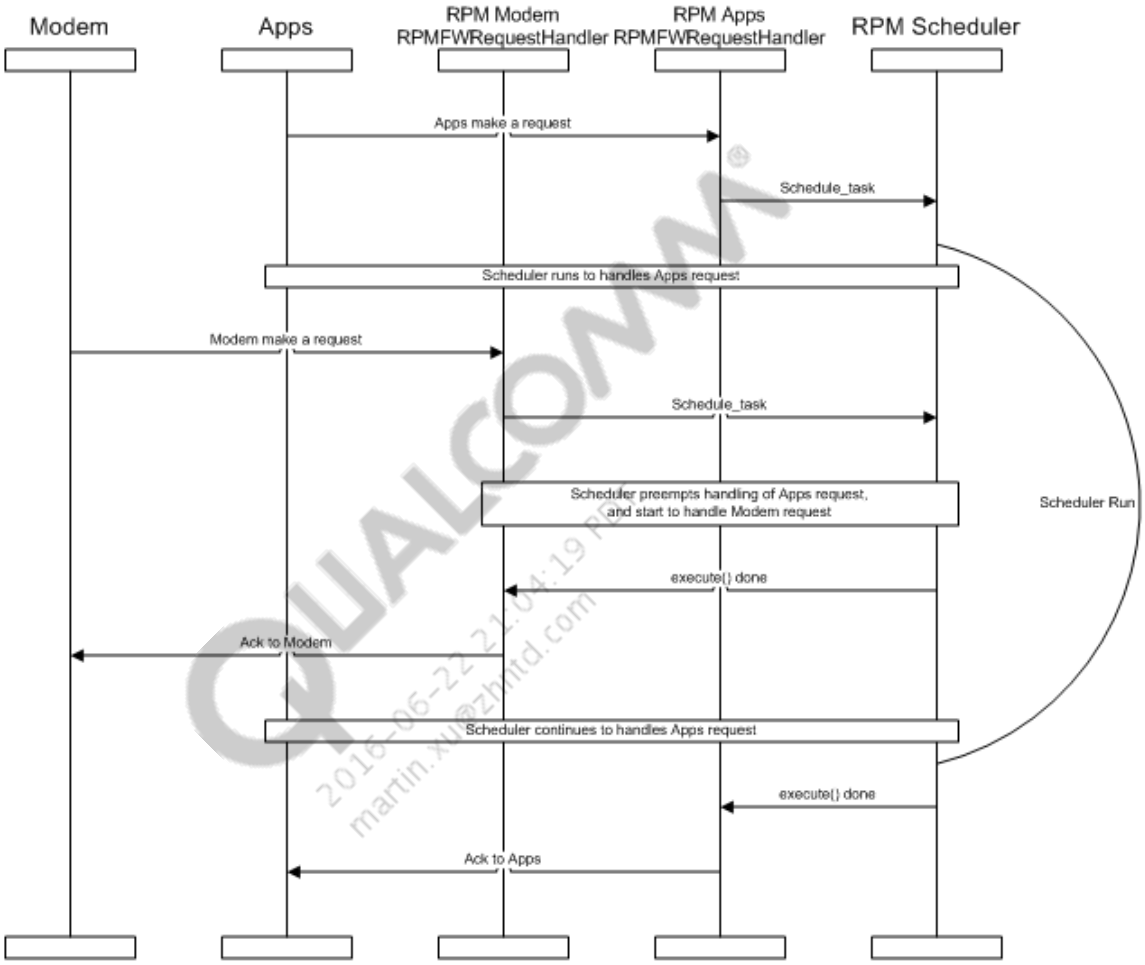


Figure 4-1 Scheduling with preemption

5 RPM messaging

5.1 RPM message infrastructure

RPM messaging is built around three core concepts:

- Masters – These are the subsystems that make requests to the RPM. The messaging masters for different targets are given in Table 2-1.
- Resources – These are the physical components that the RPM is responsible for controlling:
  - Clocks
  - Bus arbitration
  - PMIC
  - Chip sleep wakeup interrupts
  - Processor wakeup deadlines
- Configuration sets – These are a collection of settings, one for each resource, that fits a logical use case. Three sets are currently provided for each master:
  - Active set – This set contains the settings used when the subsystem is awake.
  - Sleep set – This set contains the settings used when the subsystem is sleeping (when an SPM handshake occurs).
  - Next active set – This set contains the settings used when the subsystem is awake and before the active set.

5.2 RPM message RAM

Messaging is the main method for each system master 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 5-1.

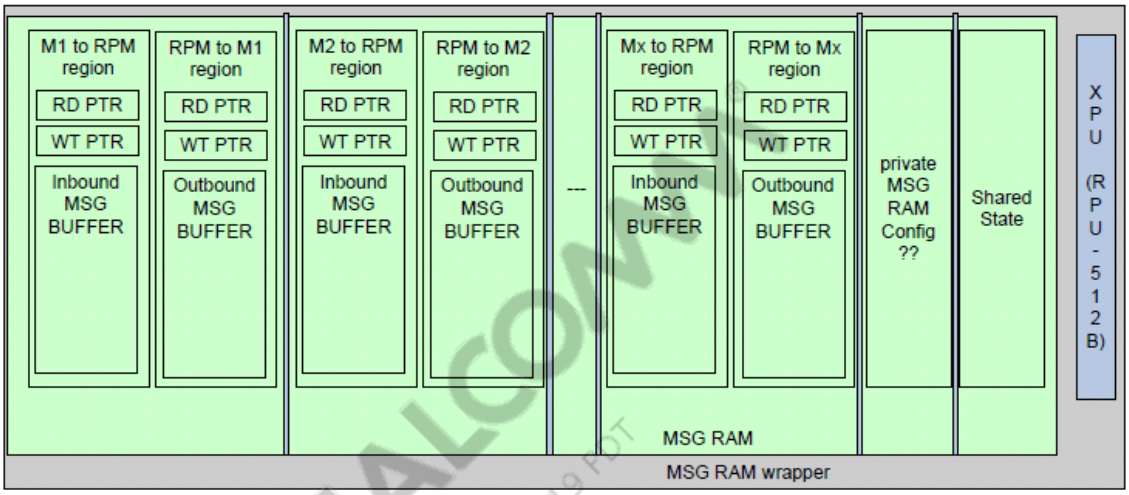


Figure 5-1 RPM message RAM partition

5.3 Transport layer – SMD Lite

SMD Lite is a message-oriented FIFO transport layer selected to replace the previous register emulation.

SMD Lite is a lightweight inter processor communication layer available on the most recent chips. It is focused on the mechanism of moving data between processors and tries to remain entirely ignorant of the policy of what that data means and of higher-level constructs, for example, queuing.

The SMD Lite API is documented in the smd_lite.h header file that is available with the build. Since the API has evolved over time, the header file that ships with the build is considered to be the canonical documentation. It defines the structures and functions in Sections 5.3.1 and 5.3.2.

5.3.1 SMD Lite data structures

The SMD Lite data structures are:

- smdl_handle_type
- smdl_iovec_type
- smdl_event_type
- smdl_sig_type
- smdl_callback_t

5.3.2 SMD Lite functions

The SMD Lite functions are:

- smdl_init()
- smdl_open()
- smdl_close()
- smdl_rx_peek()
- smdl_tx_peek()
- smdl_read()
- smdl_write()
- smdl_readv()
- smdl_writev()

5.4 Message format – Key Value Pairs (KVPs)

The messages are formatted as KVP. The KVP format was selected for better compatibility and expandability, low overhead, and because it can be incrementally parsed easily.

At a high level, a KVP structure is simply:

- What you are talking about
- Blob of data that you are saying

At a lower level, this is just 2+n words of data.

- One word for key – Generally used as a 4-byte string (uv\0\0, clk\0, etc.)
- One word for length – Describes how many bytes follow
- Blob of data

This structuring is repeated recursively to build full messages.

Internally, a request to an RPM is a concatenated array of KVP objects. KVP objects are simple in their layout in memory:

```
<4 bytes of key> <4 bytes of length> <0..* bytes of value>
```

Therefore, a request to the microvolts key (“uv”) of a regulator might be laid out in memory, e.g., shown byte-by-byte:

```
75 76 00 00 | 04 00 00 00 | e0 c8 10 00
```

The pipe symbol (|) separates the fields for visual effect. The first field is the little-endian rendering of 0x7675, which is the transliteration of “uv” as explained above. The second field indicates that the request is a 4-byte data buffer. The final field is the 4-byte data buffer containing a little-endian rendering of the value 1,100,000 µV (or 1.1 V).

5.4.1 Example of request to change LDO3 voltage and current

```
{
    “req\0” : {
        {“rsrc” : “ldo\0”}
        {“id”   : 3}
        {“set”  : 0}
        {“data” : {
              {“uv\0\0” : 1100000}
              {“mA\0\0” : 130}
          }
        }
    }
}
```

5.4.2 KVP usage

In practice, there is a kvp module which is used to build these buffers of KVP objects. A brief overview of the API follows.

First, a kvp instance is created. KVP instances contain an internally managed, dynamically expanding buffer that holds the requests and their data.

```
kvp_t *cxo_request = kvp_create(12);
```

The magic number 12 above is a hint about how much data will be placed in the stream. This example case uses 12; if you know in advance what your requests will look like, use the following rules to pick a guideline number:

- 8 bytes of overhead per call to kvp_put(), plus however many bytes of data you are “putting,” rounded up to a 4-byte boundary on each put()
- 0 is always an acceptable hint; the kvp object always reallocates a larger buffer as required

Next, put the KVP into the buffer:

```
unsigned nCXORequirement = 1;
 unsigned units_Enab = 0x62616e45; // = "Enab" -- the units for CXO
 kvp_put(cxo_request, units_Enab, sizeof(nCXORequirement),
&nCXORequirement);
```

Every time you pass a kvp object to an RPM function call, its contents are “consumed.” To reuse the kvp object later on, make sure to take the following cleanup actions:

- To reuse the buffer to resend the exact same data, call kvp_reset(<your object>).
- To reuse the buffer to send /different/ data, call kvp_clear(<your object>). Do not forget to use kvp_put() to add new data before sending.

Of course, if there is no further required use of the buffer, kvp_destroy(<your object>) can be used to clean it up entirely.

6 RPM railway

6.1 Railway-related components

6.1.1 RPM server

The RPM server routes requests from masters and internal clients to the correct resource in the RPM.

6.1.2 PMIC driver

The PMIC driver processes voltage transition requests from masters and internal clients on the RPM. All knowledge of the power grid for the specific target is contained within the PMIC driver.

6.1.3 Railway

- Manages the Vdd_Mx, Vdd_Cx, rails and their dependencies
  - Vdd_Mx must be greater than or equal to Vdd_Cx at all times
- Provides notifications to registered clients when one of the ‘managed’ rails changes voltage
- Provides notification to the sleep module regarding whether it is possible to go into Vdd minimization based on the aggregated votes on Vdd_Cx and Vdd_Mx
- Allows the Core Power Reduction (CPR) driver to override the default corner values with the dynamic values that are recommended by the CPR (Vdd_Cx)
- Provides NPA nodes to allow the clock driver to vote on voltages for Vdd_Cx.
- Allows internal RPM clients to make non suppressible requests for voltages on any of the managed rails; these non-suppressible requests influence whether Vdd minimization can occur.

6.1.4 SVS

- Tracks the Vdd_Cx voltage and votes on the RPM CPU clock to be maximum speed that is available at the current Vdd_Cx voltage
- Can also vote to raise Vdd_Cx in order to boost the CPU frequency should the current scheduler workload make that the sensible thing to do

6.1.5 Clock driver

The clock driver manages all of the shared clocks on the system; it votes via NPA to raise voltages as required to scale clock frequencies.

6.1.6 CPR driver

The CRP driver manages the CPR hardware for the Vdd_Cx rail, provides a table of dynamic corner voltages to the railway, and updates this table (per rail) as the CPR hardware notifies it that the voltage should be raised/lowered to maintain timing.

6.1.7 DDR driver

The DDR driver manages the DDR settings and updates the DDR controller hardware as Vdd_Cx and Vdd_Mx change.

6.1.8 Sleep

Sleep code is executed whenever the RPM is idle. The sleep code will put the RPM into halt/Vdd minimization/XO shutdown depending on a number of factors:

- How long it expects the RPM to be idle
- The state of the masters; any master being active prevents XO shutdown and Vdd minimization
- Depending on what railway votes for the sleep target of various rails, Vdd minimization may or may not be possible

6.2 Interfaces and dependencies

Figure 6-1 illustrates the interfaces and dependencies of the RPM railway.

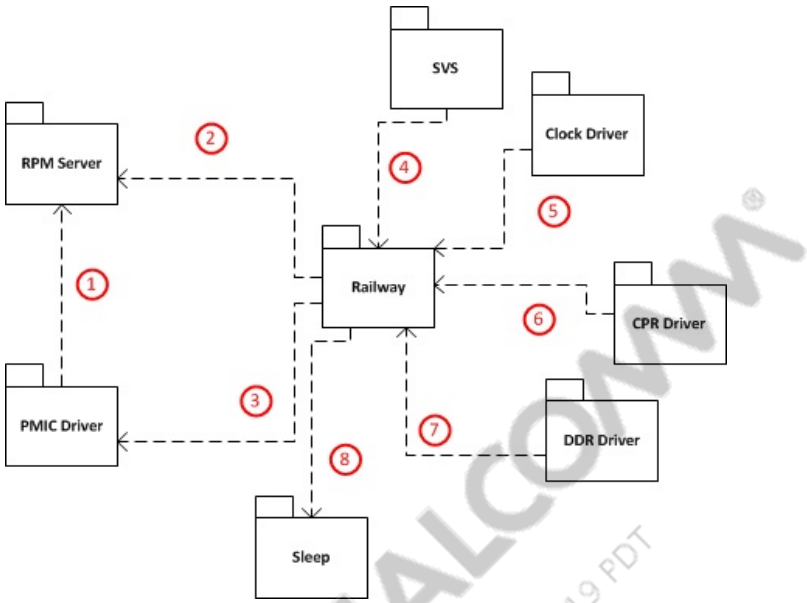


Figure 6-1 Interfaces and dependencies

The following numbered steps correspond to Figure 6-1:

1. The PMIC driver registers with the RPM server to receive all PMIC requests (RPM_SMPS_A_REQ, RPM_LDO_A_REQ, etc.).
2. Railway registers with the RPM server to receive PMIC requests for specific resources, i.e., Vdd_Cx, Vdd_Mx. These override the registration done by the PMIC driver; the PMIC driver receives no notification for votes on these resources. For these resources, the RPM server routes both the xlate and apply functions to the override, i.e., railway.
3. Railway makes requests for voltages on the rails it manages directly to the PMIC driver. The PMIC driver is responsible for understanding the power grid of the specific target being run on and adjusting the parent regulators as required.
4. SVS registers with railway for notifications on when Vdd_Cx changes. It also uses the railway API for voting for Vdd_Cx up when it wants to boost the CPU speed.
5. Clock driver registers with railway for notifications on when Vdd_Cx changes. It also votes on /pmic/client/clk_regime_dig nodes for when it wants to change voltages on Vdd_Cx. These nodes are implemented by the railway component.
6. CPR registers with railway for notifications when either Vdd_Cx change (so that it can reinitialize the CPR hardware to the new corner). It also uses an API on railway to initiate a voltage change when the dynamic CPR-derived voltage for the current corner has been updated.
7. DDR driver registers with railway for notifications on when Vdd_Mx change.
8. Railway is responsible for updating the /sleep/uber node depending on whether there are any non-suppressible votes for Vdd_Cx, which would prevent Vdd minimization.

7 RPM RBCPR

7.1 RPM CPR

CPR is a system that embeds into the SoC and is used to control the VDD level of a chip. CPR consists of a CPR master core and satellite sensors that are integrated into blocks on the 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. The CPR driver was added in the RPM to control the Vdd DIG.

The Rapid Bridge Core Power Reduction (RBCPR) core consists of one master and a number of sensors, shown in green in Figure 7-1. There is additional QTI wrapper logic called rbif, shown in blue in Figure 7-1. The software can program registers in the rbif_csr block, in the QTI rbif. There are no programmable registers in the Rapid Bridge logic.

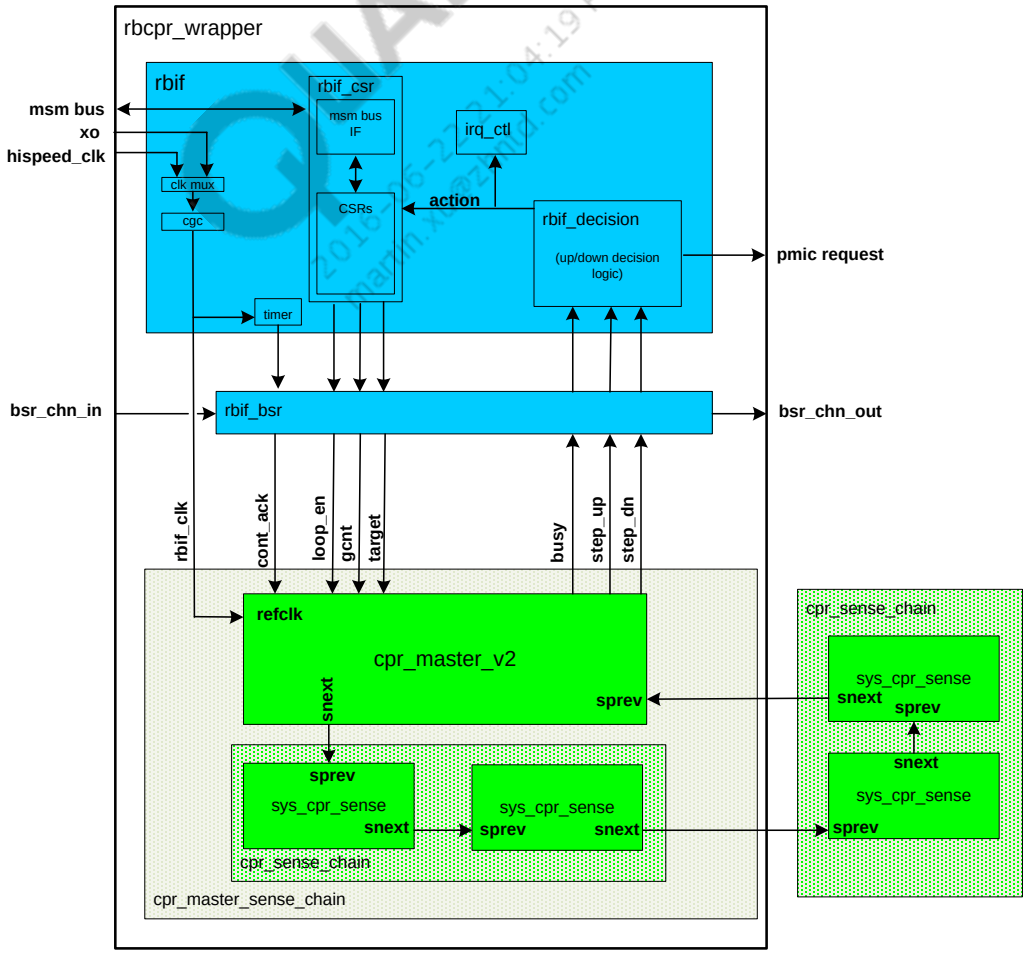


Figure 7-1 RBCPR core with master and sensors

7.2 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 and 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 is lower; if the silicon is on the fast side, the QUOT is 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. See Figure 7-2.

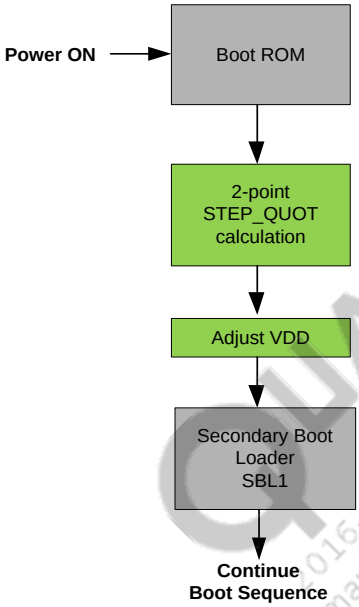


Figure 7-2 Two-point STEP_QUOT calculation during boot

RPM performs an arithmetic calculation in the software to derive STEP_QUOT as shown in Figure 7-3.

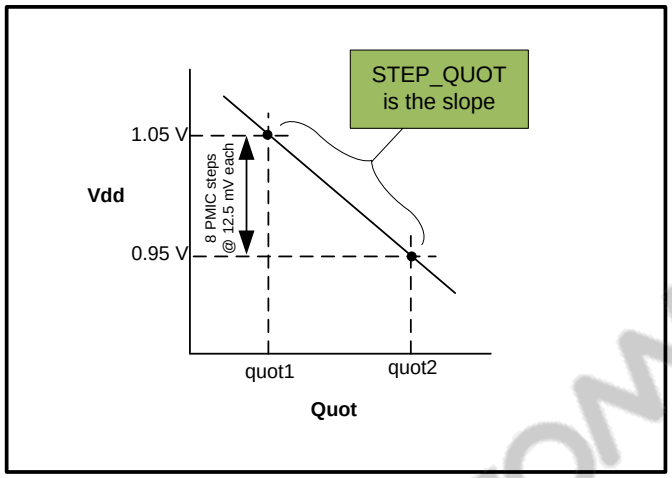


Figure 7-3 STEP_QUOT is the number of QUOT units per PMIC step

Division by 8 is used because the two quotients were measured at Vdd points that are eight PMIC steps apart. Each PMIC step is 12.5 mV; quot_1 was measured at 1.05 V and quot_2 at 0.95 V. The STEP_QUOT shows how many QUOT units correspond to one PMIC step.

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

7.3 CPR measurement and adjustment

This section describes how the CPR takes measurements and makes adjustments.

1. Configure the CPR interrupts.
2. Receive the CPR interrupt, look at RBCPR_STATUS, and make the PMIC adjustment.
   a. If step_up is 1, increase the PMIC Vdd by one step.
   b. If step_down is 1, decrease the PMIC Vdd by one step.
3. Notify CPR to take another RBCPR measurement. See Figure 7-4.

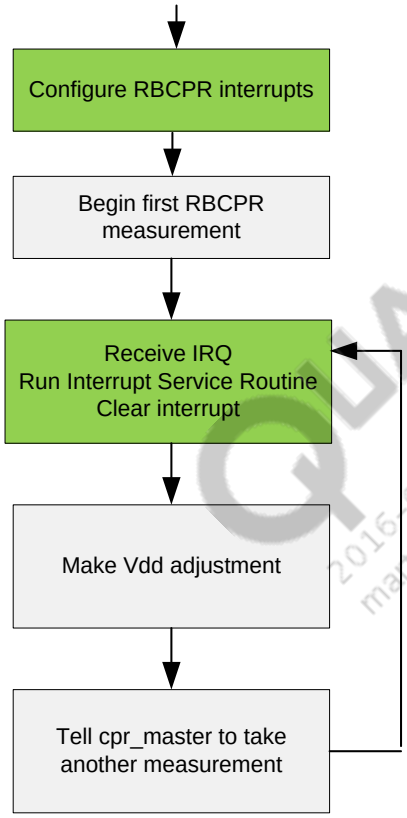


Figure 7-4 CPR measurement/adjustment

7.4 CPR voltage switching

The RPM runs in one of three modes:

- Nominal
- Turbo
- SVS

Figure 7-5 shows the CPR voltage switching modes.

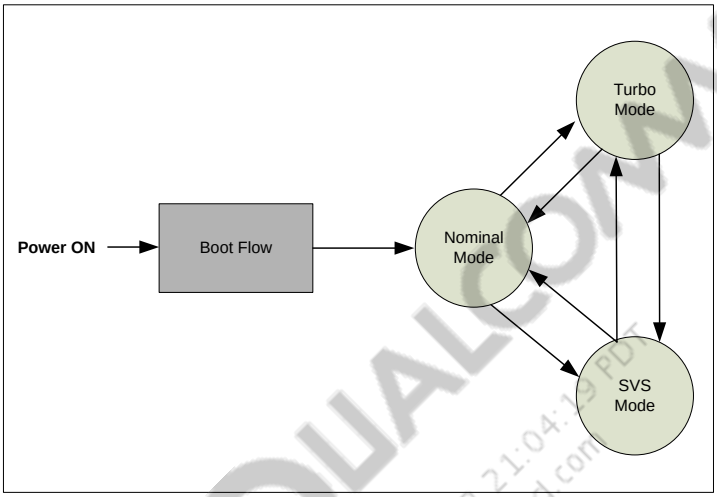


Figure 7-5 CPR voltage switching flow

When 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 two-point STEP_QUOT calculation
- Configure RBCPR with the new GCNT/TARG pairs
- Re-enable the block and interrupts

7.5 CPR driver source code

The CPR driver source code resides in the folder rpm_proc\core\power\rbcpr\ and has the following properties:

- 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 the software algorithm respectively.
- The 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:

  One step = 12.5 mV

7.6 CPR debug

7.6.1 Enable/disable CPR at runtime

RBCPR is enabled by default during rpm_init().

7.6.1.1 Enable/disable CPR

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.

To enable/disable CPR, replace all the instances of the variable rbcpr_enablement with RBCPR_DISABLED in the file rbcpr_bsp.c located at /rpm_proc/core/power/rbcpr/src/target/8916 <Target>

…..

```
~~.rbcpr_enablement=RBCPR_ENABLED_CLOSED_LOOP,~~
.rbcpr_enablement= RBCPR_DISABLED,
```

……

7.6.2 Retrieve the RBCPR log

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

- Fuse voltage (CPR starting point)
- For each mode (SVS/Nominal/Turbo):
  - Number of mode interrupts
  - 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 statistics

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

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

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

Example

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

8 RPM debugging

8.1 Trace32 scripts

8.1.1 Save RAM dump – rpm_dump.cmm

If the RPM crashes and there is no expert available, use the following script to capture the state of the session for later analysis:

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

8.1.2 Load RAM dump – rpm_load_dump.cmm

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

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

8.1.3 Restore a crash – rpm_restore_core.cmm

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

```
do rpm_restore_core.cmm
```

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

8.1.4 Parse log – rpm_parse_faults.cmm

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

```
do rpm_parse_faults.cmm
```

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

8.1.5 Examine the preempted process – rpm_m3_unstack.cmm

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

```
do rpm_m3_unstack.cmms
```

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

8.2 Getting the RPM log

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

8.2.1 Using T32

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

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

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

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

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

9 Software events

Software events are used to replace RPM log events in RPM.AF. Table 9-1 lists how to add the software event entry.

9.1 RPM SWEvent range table

Table 9-1 reflects the enumeration in core/api/debugtrace/tracer_event_ids.h.

Table 9-1 Software event entry

ID range	Tech area
0	RESERVED
1 to 63	DDR
64 to 191	BUS
192 to 319	RPM
320 to 383	SLEEP
384 to 511	CLOCK
512 to 639	PMIC
640 to 649	OCMEM
650 to 669	RAILWAY
670 to 689	RBCPR
700 to 709	SYSTEM_DB
710 to 1023	RESERVED

9.2 Adding RPM SWEvents

9.2.1 Add new SWEvent to tech area SConscript

Add the new SWEvent to the bottom of the list of SWEvents in the SConscript, but before the last event. The format is ['NEW_SWEVENT', 'description of new software event: param1 %d'].

An example of adding RPM_MASTER_SET_TRANSITION_COMPLETE to core/power/rpm/build/Sconscript is:

```
if 'USES_QDSS_SWE' in env:
  QDSS_IMG = ['QDSS_EN_IMG']
  events = [['RPM_BOOT_STARTED=192','rpm boot started'],
     ['RPM_BOOT_FINISHED','rpm boot finished'],
     ['RPM_BRINGUP_REQ','rpm_bringup_req: (master %d) (core %d)'],
     ['RPM_BRINGUP_ACK','rpm_bringup_ack: (master %d) (core %d)'],
     ['RPM_SHUTDOWN_REQ','rpm_shutdown_req: (master %d) (core %d)'],
     ['RPM_SHUTDOWN_ACK','rpm_shutdown_ack: (master %d) (core %d)'],
     ['RPM_TRANSITION_QUEUED','rpm_transition_queued: (master %d) (status
%d) (deadline: %d)'],
     ['RPM_MASTER_SET_TRANSITION','rpm_master_set_transition: (master %d)
(fromSet %d) (toSet: %d)'],
     ['RPM_MASTER_SET_TRANSITION_COMPLETE','rpm_set_transition_complete:
(master %d)'],
     ['RPM_LAST=319','rpm last placeholder'],
  ]
  env.AddSWEInfo(QDSS_IMG, events)
```

9.2.2 Add SWEVENT calls for new SConscript

Include swevent.h and make calls of the form SWEVENT(<enum>, params...).

For example:

```
#include "swevent.h"
...
SWEVENT(RPM_MASTER_SET_TRANSITION_COMPLETE, master_id);
```

9.2.3 Add SWEVENT RAM post-parsing

1. Add parsing details to core/power/rpm/debug/scripts/<tech area>_parser.py.
2. ID is a hex value which matches the swevent ID.
   RPM_MASTER_SET_TRANSITION_COMPLETE = 200 = 0xc8.
3. The class should return a string describing the SWEvent.
4. Data is parseable with various functions, described in core/power/rpm/debug/scripts/target_data.py.

   An example of adding a parsing class for RPM_MASTER_SET_TRANSITION_COMPLETE is:

```
class RPMTransitionComplete:
   __metaclass__ = Parser
   id = 0xC8
   def parse(self, data):
         return 'rpm_master_set_transition_complete (master: %s)' %
get_master_name(data[0])
```

10 RPM master user guide

10.1 Send requests to RPM

Assuming CXO = (clock, 0), to turn it on right now, submit the following Active mode RPM driver request:

```
unsigned rpm_post_request(rpm_set_type set, rpm_resource_type resource,
unsigned resource_id, kvp_t *kvps)

ex: rpm_post_request (RPM_ACTIVE_SET, RPM_CLOCK_REQ, 0, cxo_request);
```

RPM_CLOCK_REQ is a clk string. cxo_request is the KVP buffer constructed as described in Section 5.4.2.

rpm_post_request returns a message ID. Message IDs uniquely identify the changes that your post request generated to the RPM. Sometimes the message ID can be 0. This may happen when the request was redundant.

API rpm_barrier(<message ID>) with the ID returned by rpm_post_request is used to wait for the completion of that message (and all preceding messages).

To chain multiple messages and wait for them all to complete, use the following idiom:

```
unsigned last_id = 0, id;
id = rpm_post_request(...);
last_id = id ? id : last_id;
id = rpm_post_request(...);
last_id = id ? id : last_id;
id = rpm_post_request(...);
last_id = id ? id : last_id;
rpm_barrier(last_id);
```

This sequence invokes three RPM requests and then waits for all three to complete. It guarantees that all of the requests have completed, regardless of set request or redundance.

A References

A.1 Acronyms and terms

Acronym or term	Definition
AHB	Advanced High-performance Bus
CCR	Configuration Control Register
CPR	Core Power Reduction
CSR	Control/Status Register
KVP	Key Value Pairs
LPAE	Large Physical Address Extension
MPM	MSM Power Manager
NVIC	Nested Vectored Interrupt Controller
RBCPR	Rapid Bridge Core Power Reduction
RPM	Resource Power Manager