Freescale Semiconductor
White Paper

Document Number: QORIQPMWP
Rev. 0, 12/2014

QorIQ Power Management

Convention

This document uses the `Courier New` font to identify commands, explicit command parameters, code examples, expressions, data types, and directives.

Contents

1 Power Management (PM) introduction

As embedded processing becomes more pervasive, compute power becomes more complex, and electronic systems consume more power over time, controlling or reducing these systems' power consumption is critical to avert cost and environmental concerns. Power Management technologies will play a more important role in addressing this issue.

1.1 Power consumption composition

This figure shows a typical diagram of CMOS, which is extensively used in semiconductor products.

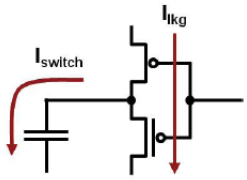


Figure 1. CMOS diagram

The CMOS power consumption can normally be grouped into two categories:

- Static Power Consumption
- Dynamic Power Consumption

Below is a formula for calculating the rough power consumption of a CMOS-based semiconductor product. The left side of the formula is the Dynamic Power Consumption and the right side of the formula is the Static Power Consumption.

$$E = \int_0^t (CV_{dd}^2 f_c + V_{dd} I_{lkg})\, dt$$

From this formula, we can see that the higher the working frequency (performance), the higher the dynamic power consumption. The more functions we are supporting, the more transistors are in use, which means more static and dynamic power is needed. Therefore, as a rule of thumb: power consumption increases with more performance or functional capacity provided.

Figure 2 shows the base concept for Power Management technologies.

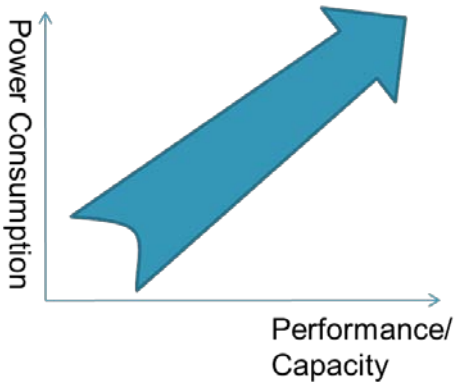


Figure 2. Power consumption and performance/capacity

1.2 Power Management definition

- The state in which the system is not operating in full performance/capacity to save power is called a low-power state.
- Power management (PM) is a technology used on some electrical appliances that turns off the power or switches the system to a low-power state when inactive.
- The key to power management is matching the runtime performance/capacity with runtime workload requirements. Turn off everything else or slow down to what is necessary to reduce the runtime power consumption.
- Power management can NOT reduce maximum power consumption when the system is fully loaded. Therefore, the board designer still has to provision a power supply that needs to support the peak demands of the system according to the hardware specifications.

2 QorIQ PM features

2.1 CPU core low-power states

This table shows the QorIQ hardware low-power states in the CPU core level (thread/core/cluster), which covers both Power Architecture® and ARM® architecture.

Table 1. QorIQ hardware low-power states in the CPU core level

States	*PH00*	*PH10*	*PW10*	*PH15*	*PW15*	*PH20*	*PW20*	*PW30*	*PCL10**	*PCL30*
Other names	Run	Doze	Wait	Nap		Drowsy	Drowsy		Stop	
ARM names	Normal				WFI/WFE	Core retention	Core dynamic retention	Core shutdown	L2WFI	Cluster shutdown

Instruction fetch	On	Stopped	Stopped	Stopped	Stopped	Stopped	Stopped	Stopped	Stopped	Stopped
Core clock	On	On	On	Off	Off	Off	Off	Off	Off	Off
Core Voltage	On	On	On	On	On	Retention	Retention	Off	Retention/On	Off
Core L1 Cache Snoop	On	On	On	Off	On	Off(PPC) On(ARM)	Off(PPC) On(ARM)	Off	Off	Off
Cluster Clock	On	On	On	On	On	On	On	On	Off	Off
Cluster L2 Cache Snoop	On	On	On	On	On	On	On	On	Off	Off
Available on	All	E500V2 E500MC E5500	E500V2 E500MC E5500	E500V2 E500MC E5500	A7 A9 A53 A57	E6500	E6500 A53 A57	A7 A9 A53 A57	E6500 A7 A53 A57	A7 A53 A57

Other core PM features include:

Table 2. Other PM features

Feature	Description	Available on
Drowsy Altivec	Disable Altivec co-processor when the Altivec instructions are not being used	E6500

2.2 SOC/system-wide low-power states

This table shows the QorIQ low-power states in the SoC or system-wide.

Table 3. QorIQ low-power states in the SoC or system-wide

SoC state	*LPM00*	*LPM20*	*LPM35*
System state	**Run**	**Sleep**	**Deep Sleep**
Core state	PH00	PH20	PH30
Core Time base	On	Stopped	Off
On-chip devices	On	Clock gated	Powered off by board

Core Voltage	On	On	Off
DDR controller	On	On	Powered off by board
Exception devices	On	On	On
DDR memory	On	Self-refresh	Self-refresh
On-board devices	On	Depending on board implementation, normally On.	Powered off by board
Available on	All	All (Except SoC with StarCore)	LS1021, T1040, P1022, MPC8536, MPC8313

2.3 Other PM features

Other PM features on the SoC include:

Table 4. Other PM features on the SoC

Feature	Description	Available on
Dynamic Frequency Scaling (DFS)	Changing the CPU frequency at runtime.	P4080, P5020, P5040, P2041, P3041, T4240, T2080, T1040, LS1021
Cascade PM	Using only part of the queues in DPAA when the receiving I/O is not fully loaded. This will help make part of the cores in the system become idle and possible to enter low-power states.	SoC with DPAA Qman
Auto Response	Automatically respond to certain types of network traffic in deep sleep in order to stay longer in the low-power state. Need the Fman microcode update to support to feature.	SoC with DPAA Fman and requires AR enabled microcode
Thermal diode	Diode for external thermal monitor to measure the internal temperature of CPU.	All
TMU	Internal thermal monitor to directly get the temperature of CPU.	TBD

There are also PM features/components outside of the SoC on the board level, as listed in the following table.

Table 5. PM features outside of the SoC

Feature	Description	Available on
Deep sleep board control logic	Cooperate with SoC to prepare the whole system for deep sleep entrance and resuming the whole system back to normal on wakeup.	P1022DS, T1040QDS, T1042RDB, LS1021QDS
Power Monitor	Monitor chip to measure the power consumption of certain power rail.	P5020QDS, T4240QDS, T1040QDS, T2080QDS, and etc.
Thermal Monitor	Monitor chip to measure the CPU temperature.	P5020QDS, T4240QDS, T1040QDS, T2080QDS, and etc.
Configurable Voltage regulator	Voltage regulator that supports changing of voltage.	P5020QDS, T4240QDS, T1040QDS, T2080QDS, and etc.

3 Linux PM features

Linux already provides different frameworks for Power Management to make system software cooperate with hardware low-power states. We have to follow the existing frameworks to make QorIQ low-power states work correctly on the Linux operating system. Linux also provides software features that can make the hardware low-power states work more intelligently and more efficiently.

3.1 Linux PM frameworks

Figure 3 demonstrates different PM-related frameworks that are currently (as of kernel 3.12) in the Linux system. Most of the Linux PM frameworks reside in kernel space and can be categorized into four groups: Linux device model extension, PM core features, CPU management features and miscellaneous features. Some of these frameworks interact with other frameworks, as shown by the blue line arrows in Figure 3. Some frameworks can interact with user space almost all through the kernel /sysfs interface.

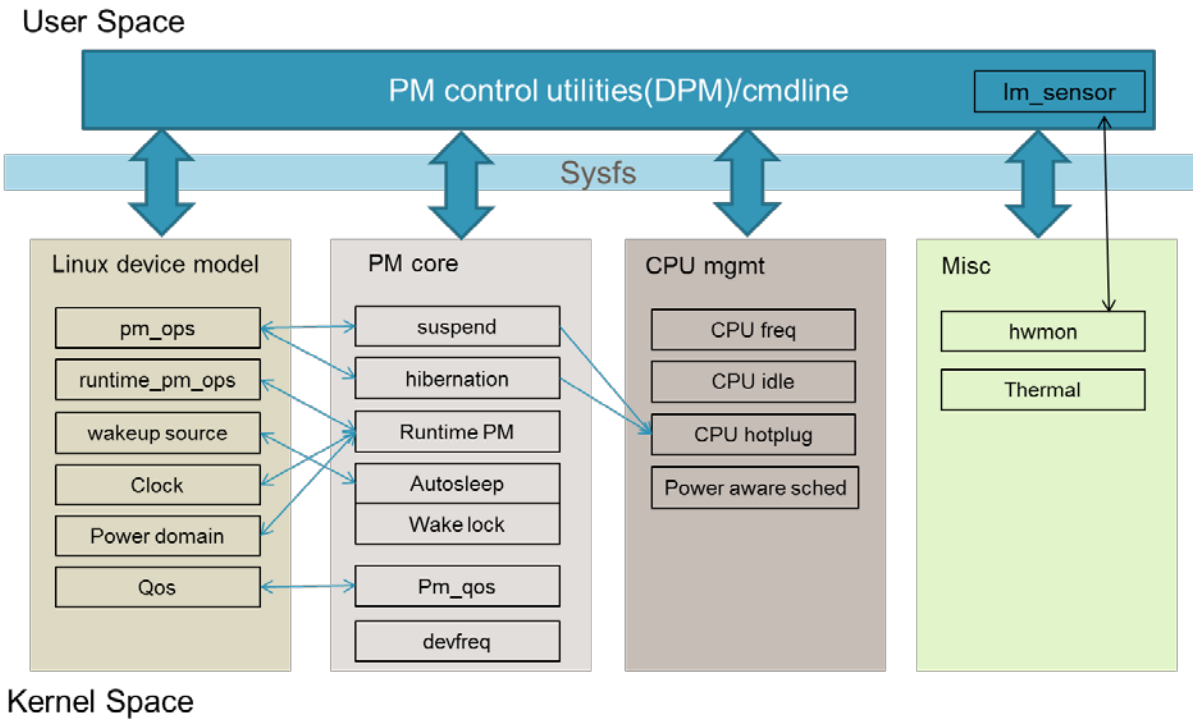


Figure 3. PM frameworks currently in the Linux system

3.2 PM features in Freescale Linux SDK

This section briefly introduces the software PM features included in the Freescale Linux SDK (as of SDK1.7). We explain some of them in detail in the following sections.

3.2.1 Static features

Static features, listed below, are PM features that trigger low-power states and/or can be woken up manually by the user or a user application:

- System Suspend (static triggered)
- Freeze all the user space applications and put the system into a system wide low-power state
  - Sleep (standby)
  - Deep sleep (suspend to RAM)
  - Hibernation (suspend to disk)
- Static wakeup mechanism
  - Wakeup on LAN (magic packet)
  - Wakeup on GPIO, external interrupt, USB, SD, timer
- CPU hotplug
- CPU frequency with performance/powersave/userspace governors

3.2.2 Dynamic features

Dynamic features, listed below, are PM Features that trigger low-power states and/or can be woken up automatically without intervention of user or user application (by operating system or hardware).

By operating system:

- CPU idle
- CPU frequency with on-demand/conservative governors (available on SDK1.7)

By hardware:

- System suspend with dynamic wakeup mechanisms
  - Wakeup on user-defined packets
  - Auto response
- Cascaded PM
- Drowsy Altivec

3.2.3 SDK support matrix

This table shows different Linux PM frameworks, which are supported on different platforms, as well as supported hardware features that are associated with these frameworks.

Table 6. Linux PM frameworks

Linux framework	Hardware feature	P4080	T4240	T1040	LS1021
System suspend	*Sleep*	Y	Y	Y	Y
	Deep Sleep	NA	NA	Y	Y
	Hibernation	Y	Y	Y	Y
	Wake on LAN	Y	Y	Y	Y
	Auto response	N	N	Y	N
CPU idle	*PW10*	Y	Y	Y	NA
	PW15	NA	NA	NA	Y
	PW20	NA	Y	NA	NA
CPU hotplug	*PH15*	Y	N	Y	NA
	PW15	NA	NA	NA	Y
	PH20	NA	Y	NA	NA
	PCL10	NA	Y	NA	N
CPU freq	*DFS*	Y	Y	Y	Y
hwmon	*On board monitors*	Y	Y	Y	Y
Misc	*Cascade PM*	Y	Y	Y	NA
	Drowsy Altivec	NA	Y	NA	NA

4 Linux system suspend feature

Linux suspend is a framework for putting the whole system into a suspend state wherein almost nothing works in order to achieve aggressive power saving. Linux supports three suspend states:

- standby – Using QorIQ **sleep** hardware low-power state
- mem – Using QorIQ **deep sleep** hardware low-power state
- disk – **Powers off** all hardware (suspend to disk/hibernation)

Wakeup from a suspend state can be triggered through various system interrupts and events. The usable wakeup events on each SoC could be different. The wakeup events include:

- MPIC timer
- Ethernet magic packet
- Ethernet user defined packet
- USB plug/unplug events
- SD card plug/unplug events
- External interrupts
- GPIO events

4.1 Usage

- Kernel build options required:
  - `CONFIG_SUSPEND` (needed by all three suspend states)
  - `CONFIG_HIBERNATION` (only needed by disk state)
- Different suspend states can be triggered through sysfs interface by using different commands listed below:

```
#echo standby > /sys/power/state
#echo mem > /sys/power/state
#echo disk > /sys/power/state
```

4.2 Deep sleep high level process

Figure 4 shows the typical flow of deep sleep entrance and wakeup on recent chips (for example, T1040 and LS1021). The blocks in orange are actions owned by the system software, and blocks in green are actions owned by hardware (board and SoC).

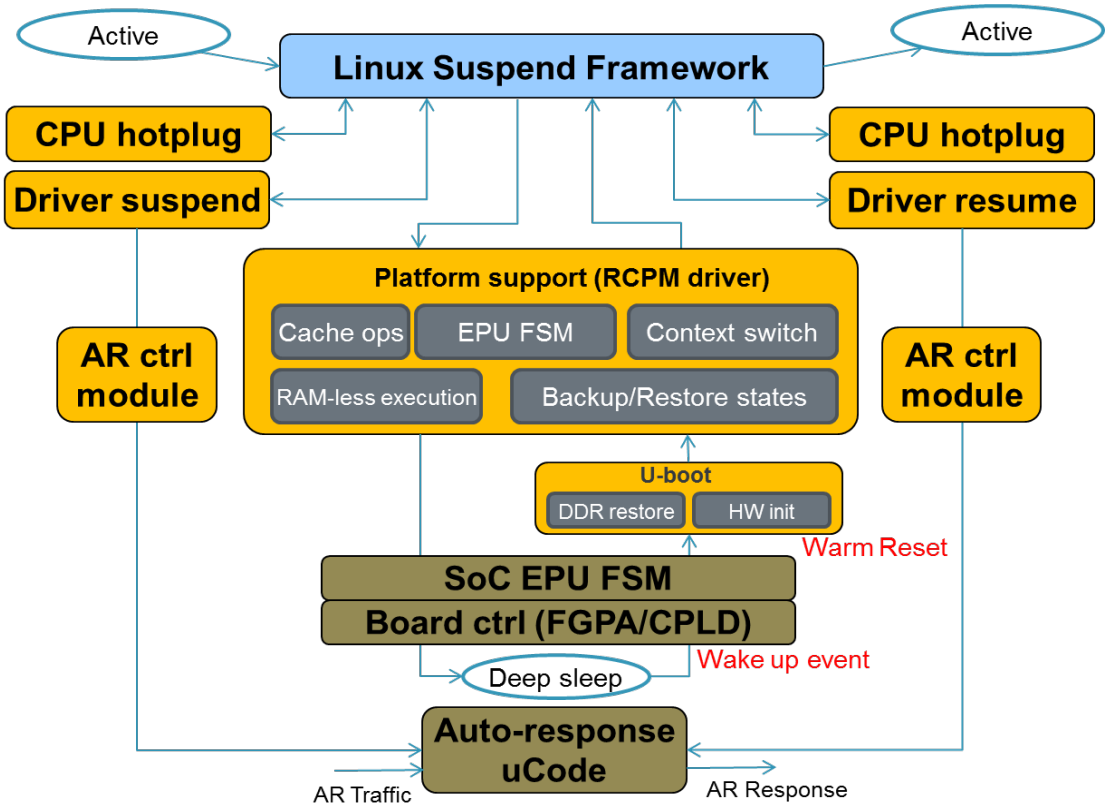


Figure 4. Deep sleep entrance and wakeup flow (typical)

The following procedure outlines deep sleep entrance:

1. User/application triggers deep sleep.
2. Linux freezes all the on-going tasks.
3. Linux calls suspend callbacks of each device driver to backup device states and stop device properly.
4. Linux puts all the non-booting CPUs offline.
5. RCPM driver backs up states of booting CPU in DDR, puts DDR into self-refresh, initializes and trigger the deep sleep state machine.
6. Deep sleep state machine puts the SoC into LPM35 low-power state and triggers board action.
7. Board control logic powers off most power rails of the SoC and shuts down unused devices on the board.
8. Deep sleep is entered.

Similarly, deep sleep wakeup can be performed with the following procedure:

1. Deep sleep state machine observes wakeup event and triggers board action.
2. Board control logic powers-on the SoC and on-board devices that were powered off.
3. Deep sleep state machine triggers a warm reset, and CPU starts running boot code.
4. Boot loader finds out this is a deep sleep restore, initializes the DDR controller, pulls DDR out of self-refresh, and jumps to the Linux resume code in DDR.

5. RCPM driver restores backed up core states and cleans up deep sleep state machine.
6. Linux brings all the non-booting CPU online.
7. Linux calls resume callbacks of each device driver to restore device states and kick starts the device.
8. Linux resumes all the frozen tasks.
9. System is back to normal.

4.3 Deep sleep detailed sequence

This section explains the detailed process that happens during the deep sleep entrance and wakeup.

4.3.1 Deep sleep entrance process

All steps below happen in Linux kernel space:

1. Sync file systems.
2. Suspend prepare:
    a. Prepare console.
    b. Send `PM_SUSPEND_PREPARE` notification.
    c. Disable user-mode helper.
    d. Freeze processes and kernel thread (freezable only, normal kthreads are non-freezable, freezable kthread need to be specifically configured).
3. Set `gfp_allowed_mask` to avoid I/O in malloc.
4. Platform specific `suspend_ops->begin()`.
5. Suspend console.
6. Device PM suspend start:
    a. Call `.prepare()` callbacks of device drivers.
    b. Call `.suspend()` callbacks of device drivers with async callbacks scheduled.
7. Platform-specific `suspend_ops->prepare()`.
8. Device PM suspend end:
    a. Call `.suspend_late()` callbacks.
    b. Disable all interrupts except ones with `IRQF_NO_SUSPEND` in IRQ controller.
    c. Pause cpuidle framework.
    d. Call `.suspend_noirq()` callbacks of device drivers with interrupt disabled.
9. Platform specific `suspend_ops->prepare_late()`.
10. Disable all CPUs other than the booting CPU.
11. Arch disable all interrupts (clear EE) – no return for interrupt triggered wakeup.
12. Call `suspend()` callbacks for system core devices, such as ktime.
13. Last check on pending wakeup events.
14. Call platform-specific `suspend_ops->enter()` //Using T1040 as example, other silicon are similar:

a. Mask interrupts in RCPM (POINT of NO RETURN).
b. Backup content of DDR which will be damaged in DDR training.
c. Setup resume pointer to SCFG_SPARECR2 register.
d. Enable warm reset in SCFG_DPSLPCR register.
e. Setup state machine in the EPU.
f. Backup core states into DDR memory.
g. Flush CPC cache.
h. Setup environment for running code without RAM.
i. Load the code into icache and start running the code in cache.
j. Put DDR into self-refresh.
k. Notify board control logic(FPGA/CPLD) to be ready for deep sleep entrance (monitor shared pins for deep sleep event) through writing board control logic register.
l. Assert MCKE isolation (GPIO1[29]) so that DDR MCKE signal is driven by board instead of SoC.
m. Trigger EPU state machine to take the SoC into LPM35 state.
n. EPU asserts board isolation (EVT_B[9]) to board control logic, to isolate SoC I/O pins with board.
o. EPU de-asserts POWER_EN to notify the board control logic (FPGA/CPLD) to shut down power rails for cores and most I/O.
p. Deep sleep is entered.

Note:

COLOR for common steps in Linux suspend framework

COLOR for platform specific steps

COLOR for board-related steps that need board design collaboration

4.3.2 Deep sleep wakeup process

1. Deep sleep wakeup process before u-boot:
    a. EPU detects wakeup event.
    b. EPU asserts POWER_EN to notify the board control logic (FPGA/CPLD) to turn on power rails.
    c. EPU waits POWER_OK to be asserted by the board.
    d. EPU issues device warm reset.
    e. PBL loads RCW and PBI.
    f. Secure boot authentication is bypassed.
2. Deep sleep wakeup process in u-boot:
    a. Start running u-boot, and u-boot finds out this is deep sleep restore by checking CRSTSR[WDRFR] .

b. Notify board control logic (FPGA/CPLD) to be out of deep sleep state through writing board control logic register.
c. Disable MCKE isolation on board.
d. Re-initialize DDR controller with by-pass mode, do the DDR training in specific address.
e. Bring DDR out of self-refresh.
f. Restore DDR content damaged by DDR training.
g. Get the kernel re-entry point from `SCFG_SPARECR2` and jump to it.

3. Deep Sleep wakeup process transfer to Linux kernel space for all steps below:
4. Platform specific setup (leftover for `suspend_ops->enter()`):
    a. Setup non-booting CPUs and MMU.
    b. Restore core states from memory.
    c. Disable warm reset.
    d. Clean up EPU state machine.
5. Call `resume()` callbacks for system core devices (such as ktime).
6. Arch enable all interrupts (set EE).
7. Enable all CPUs other than the booting CPU.
8. Platform specific `suspend_ops->wake()`.
9. Device PM resume start:
    a. Call `.resume_noirq()` callbacks.
    b. Enable interrupts on the IRQ controller.
    c. Resume cpuidle framework.
    d. Call `.resume_early()` callbacks.
10. Platform specific `suspend_ops->finish()`.
11. Device PM resume end:
    a. Call `.resume()` callbacks with async callbacks scheduled.
    b. Call `.complete()` callbacks.
12. Resume console.
13. Platform specific `suspend_ops->end()`.
14. Set `gfp_allowed_mask` to allow I/O in malloc.
15. Suspend finish:
    a. Thaw processes.
    b. Call `PM_POST_SUSPEND` notifiers.
    c. Restore console.

Note:

COLOR for common steps in Linux suspend framework

COLOR for platform specific steps

COLOR for board-related steps that need board design collaboration

4.4 Timing and order of device callbacks

Each device driver can register different callbacks which will be called at different timings. We can refer to the detailed sequence introduced above for the exact timing in the whole process. Due to the dependency of different devices or drivers, maintaining a particular order when invoking callbacks of different device drivers in a same timing is critical. For the same timing, all the devices will be traversed for available callback according to the order of an existing `dpm_list` data structure in Linux.

4.4.1 Order of different timing and traverse order in each timing

- Entrance:
  - `Prepare()` – dpm_list order
  - `Suspend()`– reverse dpm_list order
  - `Suspend_late()`– reverse dpm_list order
  - `Suspend_noirq ()`– reverse dpm_list order
- Wakeup:
  - `Resume_noirq()`– dpm_list order
  - `Resume_early()`– dpm_list order
  - `Resume()`– dpm_list order
  - `Complete()` – reverse dpm_list order

4.4.2 Order of dpm_list

The order of dpm_list is critical for the order of callbacks in the same timing. The dpm_list is created according to the order of device creation with `device_add()` routine during the Linux boot up and runtime hotplug. As a general rule, the bus device or parent device needs to be registered before other devices on the bus or before any child devices; dependent devices need to be registered before the device that is having a dependency. Below shows how the code impacts the actual dpm_list at runtime.

- For platform devices, which are added by device tree probe, the order is the same as the device node defined in the device tree. If a device node is defined multiple times in dts (overriding), take the order of first definition.
- For bus devices, which are added by bus probe, the order depends on probe timing and the order of driver in the kernel Makefile:
  - Initcall level of the bus probing
  - Makefile order in the same level
- Hotplugged devices are registered after devices that are present during boot up.

As a reference, the embedded file below is an example of dpm_list on T1040:

t1040-dpm-list.txt

4.5 Callbacks in different domains

PM callbacks can be defined in multiple domains for a device in the Linux device hierarchy (pm_domain, type, class, bus, and driver) for each callback timing. Only one of these callbacks is actually called for a device in each callback timing. The first available callback in the domain list below is used, while other callbacks in later domains are ignored:

1. `dev->pm_domain`
2. `dev->type->pm`
3. `dev->class->pm / dev->class->suspend`
4. `dev->bus->pm / dev->bus->suspend`
5. `dev->driver->pm`

4.6 Suspend debugging

Linux system suspend is a complex process. Any issue in the process can make the suspend fail completely. Below are a few debugging methods to help find potential issues:

- Enable console output during deep sleep entrance/wakeup process
- Put `no_console_suspend` in the kernel boot parameter (include in bootargs env setting in U-Boot).
- `CONFIG_PM_DEBUG` kernel option
- Enabling this kernel option and recompiling the kernel includes verbose logging into the kernel that could provide useful information during debugging. You might also need to increase the Linux console loglevel to make the debugging output visible on console.
  `# echo 7 > /proc/sys/kernel/printk`
- ASLEEP LED on board
- Normally there is an ASLEEP LED on the reference boards to reflect the SoC's ASLEEP signal. This LED is lit up by the hardware to indicate that sleep/deep sleep is entered on the SoC. If it is not lit up, the hardware is not in sleep or deep sleep state.

4.7 Linux wakeup framework

4.7.1 Userspace interfaces

Enable wakeup rollback

The following command allows the device to enter deep sleep with rollback mechanism enabled, which will rollback the entrance process when a Linux wakeup event is received from the issuing of the command until the calling of platform specific deep sleep entrance code in the final stage.

```
#echo `cat /sys/power/wakeup_count ` > /sys/power/wakeup_count &&
echo mem > /sys/power/state
```

Device wakeup enable/disable

The following commands enables or disables a specific device as a Linux wakeup source.

```
#echo enable > /sys/device/.../power/wakeup
#echo disable > /sys/device/.../power/wakeup
```

4.7.2 Kernel APIs

The following APIs are kernel APIs that should be used by the device drivers to register as a Linux wakeup source to the system or to trigger a Linux wakeup event.

`int device_init_wakeup(struct device *dev, bool val);` *//Initialize the wakeup framework for a device*

`void device_set_wakeup_capable(struct device *dev, bool capable);` *//Set device wakeup capability*

`int device_set_wakeup_enable(struct device *dev, bool enable);` *//Enable/disable device wakeup event*

`void pm_wakeup_event(struct device *dev, unsigned int msec);` *//Trigger a device wakeup event for rollback*

4.7.3 Avoid losing wakeup event using the wakeup_count mechanism

The wakeup framework introduces a new global sysfs attribute, `/sys/power/wakeup_count`, associated with a running counter of wakeup events and a helper function, `pm_wakeup_event()`, that may be used by kernel subsystems to increment the wakeup events counter.

`/sys/power/wakeup_count` may be read from or written to by user space. Reads always succeed and return the current value of the wakeup events counter. Writes, however, only succeed if the written number is equal to the current value of the wakeup events counter. If a write is successful, it causes the kernel to save the current value of the wakeup events counter and to compare the saved number with the current value of the counter at certain points of the subsequent suspend (or hibernate) sequence. If the two values don't match, the suspend sequence is aborted just as though a wakeup interrupt happened. Reading from `/sys/power/wakeup_count` again turns that mechanism off.

4.8 Deep sleep auto-response

Auto-Response is a Freescale-specific feature that can process and respond to certain types of packets (for example, ICMP, SNMP, and so on) so that the device appears online to other networked devices but doesn’t need to be woken up from deep sleep to process these packets.

4.8.1 Auto response enablement flow

This figure shows the flow chart for entering deep sleep with the Auto-Response feature enabled.

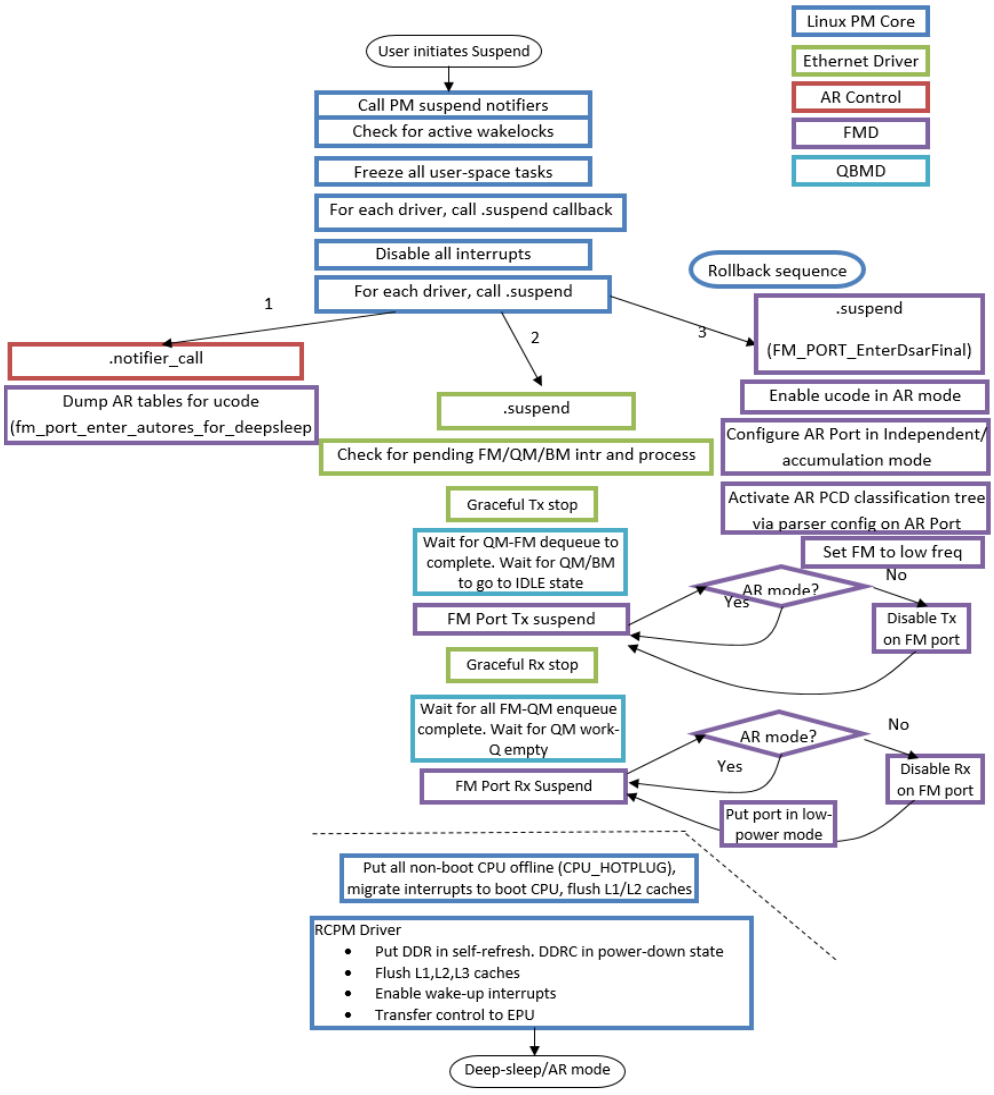


Figure 5. Entering deep sleep with Auto-Response enabled

4.8.2 DPAA drivers and callbacks involved

The hardware configuration for auto-response mode is done automatically during deep sleep entrance and wakeup making use of the device driver callbacks. Listed below are all the DPAA Ethernet related driver files. Files that are highlighted in bold font are the files that include a PM callback.

`drivers/net/ethernet/freescale/xgmac_mdio.c`

`drivers/net/ethernet/freescale/fsl_pq_mdio.c`

`drivers/net/ethernet/freescale/fman/src/wrapper/lnxwrp_fm.c : suspend()`

`drivers/net/ethernet/freescale/fman/src/wrapper/lnxwrp_fm_port.c`

drivers/net/ethernet/freescale/dpa/dpaa_eth_shared.c: suspend()

drivers/net/ethernet/freescale/dpa/offline_port.c: suspend()

drivers/net/ethernet/freescale/dpa/dpaa_eth_generic.c

drivers/net/ethernet/freescale/dpa/dpaa_eth_macless.c

drivers/net/ethernet/freescale/dpa/mac.c

drivers/net/ethernet/freescale/dpa/dpaa_eth_proxy.c: suspend()

drivers/net/ethernet/freescale/dpa/dpaa_eth.c : suspend()

drivers/staging/fsl_qbman/qman_config.c : suspend_noirq()

drivers/staging/fsl_qbman/bman_config.c : suspend_noirq()

drivers/staging/fsl_qbman/qman_high.c : pm_domain->suspend_noirq()

drivers/staging/fsl_qbman/bman_high.c: pm_domain->suspend_noirq()

drivers/staging/fsl_pme2/pme2_ctrl.c: suspend()

5 CPU idle feature

The CPU idle feature dynamically puts a CPU core in low-power states when there is nothing left to be done on that CPU, and automatically wakes up the core when there is a task to be done. This feature normally uses light weight low-power states with small latency and can be woken up by timer and IPI interrupts.

5.1 Usage

- The feature is enabled by default. There is no kernel configure option for it.
- It can be turned off by adding "`powersave=off`" to kernel bootcmd (bootargs env setting if U-Boot is used)
- On e6500 cores, CPU idle enters the PW10 state by default. We can configure the system to enter the PW20 state by using the command:
  `#echo 1 > /sys/devices/system/cpu/cpuX/pw20_state`
- For e500v2 cores, CPU idle enters the DOZE state by default. We can configure the system to enter the NAP state by using the command:
  `#echo 1 > /proc/sys/kernel/powersave_nap`

5.2 Single state CPU idle implementation

1. There is a never-ending task running on each CPU (swapper).
2. `cpu_idle()` is invoked when there is nothing else to do.
3. Calls platform power saving routines if available (for example, `ppc_md.power_save()` for Power Architecture®).

4. Enters corresponding low-power state depending on the configuration mentioned in the Usage section above.
5. Exits the low-power state when there is an interrupt pending to the core.

5.3 Multi-state CPU idle support (not supported yet)

There is a new CPU idle framework added to Linux to choose a suitable low-power state to enter when system is in idle condition.

The CPU idle subsystem uses governors to determine how a low-power state is chosen. Currently there are two governors for CPU idle:

- ladder – Enters deeper low-power state step by step (used on jiffies based system).
- menu – Calculates and selects a state by considering previous knowledge of the system load and the following aspects (used on tick-less system):
  - Energy breakeven point
  - Performance impact
  - Latency tolerance

For example, the CPU idle governor can choose from the following low-power states on e6500:

- PW10
- PW20
- PCL10

6 CPU hotplug feature

CPU hotplug is a feature that allows removal and insertion of a CPU statically into the Linux system during runtime.

6.1 Usage

- Kernel build option required: CONFIG_HOTPLUG_CPU
- Runtime control through sysfs: `/sys/devices/system/cpu/*`
  - To remove a CPU from the system:
    ```
    #echo 0 > /sys/devices/system/cpu/cpuX/online
    ```
  - To insert a CPU from the system:
    ```
    #echo 1 > /sys/devices/system/cpu/cpuX/online
    ```

6.2 Detailed removal process

The actual removal process needs to be split into two parts. One part is done on the core issuing the removal command, which could be on any running core. The other part is running on the core to be removed.

Running on the core issuing the command:

1. Send `CPU_DOWN_PREPARE` notification to all in-kernel interested modules
2. Migrate all processes to other CPU(s)
3. Migrate all interrupts to a new CPU
4. Migrate timers/bottom half/tasklets to a new CPU
5. Call an architecture specific routine `__cpu_disable()`
6. Send `CPU_DEAD` notification to all in-kernel modules
7. `__cpu_disable()` calls platform specific `cpu_disable()`
    a. Clear bit in the online cpu mask
    b. Migrate IRQ

Running on the core to be removed:

1. Finish current scheduling time slice.
2. Because all tasks and interrupts have been migrated to other cores, the core will have nothing left to do and enter `cpu_idle()`.
3. `cpu_idle()` checks `cpu_mask` and find out that the current CPU is set to offline.
4. Calls platform specific CPU disable code (for example, `ppc_md.cpu_die() = smp_85xx_mach_cpu_die()` for Power Architecture®)
    a. Flush and disable L1 cache
    b. Actually set the CPU to Low-power state (for example, PH20 state for e6500 and Nap for earlier cores)

6.3 Compatibility issues

Normal Linux applications and drivers should be correctly running on any online CPU. But sometimes applications and drivers are designed to perform a specific task on a specific CPU in order to achieve better performance. These applications and drivers need to take action when CPU hotplug is triggered.

For kernel space drivers: add callbacks to the notification chain by using `register_cpu_notifier()` for the following events:

a. `CPU_ONLINE`
b. `CPU_UP_PREPARE`
c. `CPU_DOWN_PREPARE`
d. `CPU_DEAD`

For user space applications or drivers: Process/thread affinity is automatically be updated by the system if original affinity setting cannot be met after CPU hotplug operation. Applications can get the notification of CPU hotplug event by using the Linux *uevent* interface through *udevd* utility or */sbin/hotplug* utility.

7 CPU freq feature

CPU freq, also known as Dynamic Frequency Scaling (DFS), enables changes to the working frequency of CPU/cores at runtime. There are different use cases that require different frequency change strategies. Linux CPU freq framework defines different governors to represent these use cases. The governors can also be grouped into static governors and dynamic governors:

- **performance** (static) – Sets the CPU statically to the highest frequency within the borders of `scaling_min_freq` and `scaling_max_freq`.
- **powersave** (static) – Sets the CPU statically to the lowest frequency within the borders of `scaling_min_freq` and `scaling_max_freq`.
- **userspace** (static) – Allows the user, or any userspace program running with UID "`root`", to set the CPU to a specific frequency by making a sysfs file "`scaling_setspeed`" available in the CPU-device directory.
- **ondemand** (dynamic) – Sets the CPU frequency depending on the current usage. Increase to the highest frequency if the load is higher than threshold. Decrease to the lowest frequency if the load is not there.
- **conservative** (dynamic) – Also sets the CPU frequency depending on the current usage. It gracefully increases and decreases the CPU speed rather than jumping to max speed the moment there is any load on the CPU.

7.1 Usage

The following kernel configuration options should be enabled in order to use the CPU freq feature:

- `CONFIG_CPU_FREQ` – the option needed for all CPU freq features
- `CONFIG_MPC85xx_CPUFREQ` – enables the driver for JOG hardware feature, required for MPC8536 and P1022
- `CONFIG_QORIQ_CPUFREQ` – enables the driver for QorIQ DFS hardware feature, required for other QorIQ P, T and LS series chips
- `CONFIG_ CPU_FREQ_GOV_*` – enables the support for respective governors

There are different commands used for different governors:

- **performance** (static)
  ```
  # echo performance >
  /sys/devices/system/cpu/cpuX/cpufreq/scaling_governor
  ```
- **powersave** (static)
  ```
  # echo powersave >
  /sys/devices/system/cpu/cpuX/cpufreq/scaling_governor
  ```
- **userspace** (static)
  There are multiple commands needed when using userspace governor.
  1. Change the governor to userspace

```
# echo userspace >
/sys/devices/system/cpu/cpuX/cpufreq/scaling_governor
```

2. Get available frequencies of cpuX

```
# cat
/sys/devices/system/cpu/cpuX/cpufreq/scaling_available_frequencies
   1199999 599999 299999 799999 399999 199999 1066666 533333
266666
```

3. Change the core frequency to 533 MHz

```
# echo 533333 >
/sys/devices/system/cpu/cpuX/cpufreq/scaling_setspeed
```

4. Verify the frequency setting

```
# cat /sys/devices/system/cpu/cpuX/cpufreq/scaling_cur_freq
   533333
```

- **ondemand** (dynamic)

```
# echo ondemand >
/sys/devices/system/cpu/cpuX/cpufreq/scaling_governor
```

- **conservative** (dynamic)

```
# echo conservative >
/sys/devices/system/cpu/cpuX/cpufreq/scaling_governor
```

7.2 DFS hardware feature

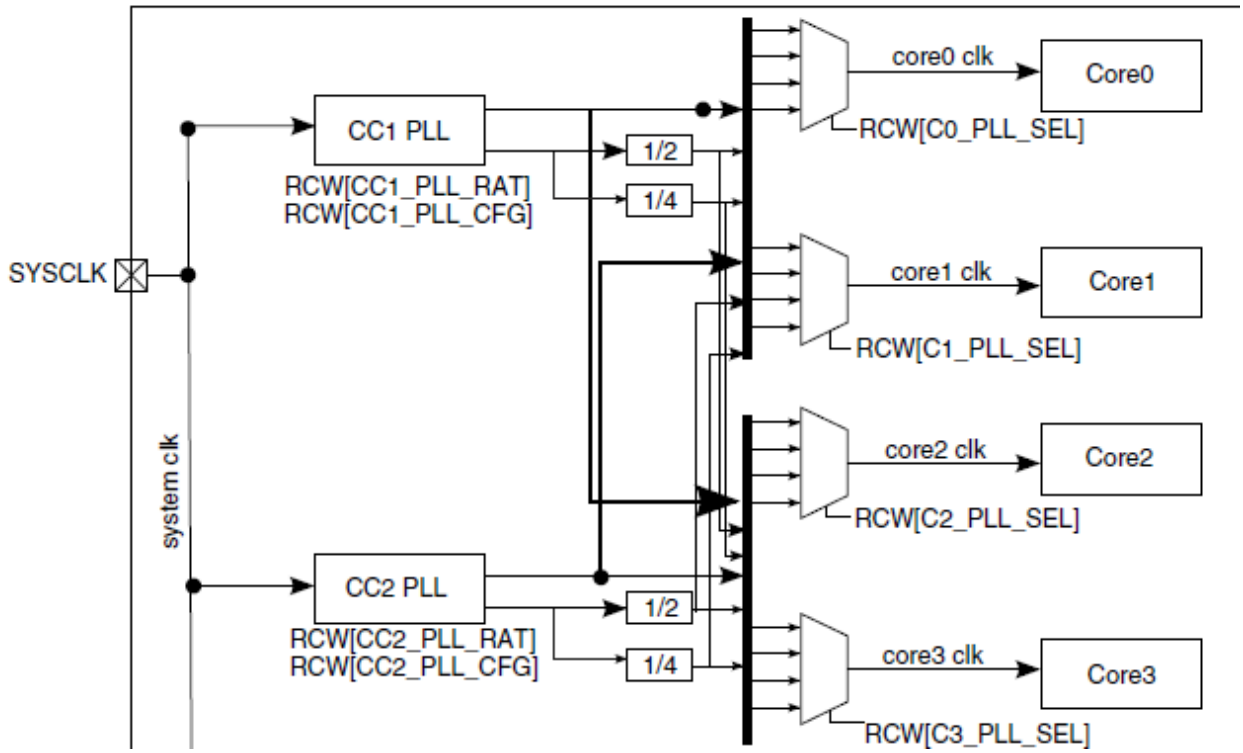


Figure 6. DFS hardware feature diagram

The DFS hardware feature depends on the hardware capability to switch the core clock among different PLLs without introducing clock glitch.

8 How to use PM technologies effectively

Each PM feature has different benefits and different side effects. Some features can naturally be suitable to certain kind of use cases and not be suitable for other use cases. It is critical to choose the correct set of PM feature(s) for a specific application in order to achieve optimal power saving without having unacceptable side effects.

8.1 The pros and cons of PM features

8.1.1 Benefit: power saving

The main purpose of using PM technologies is to save power. However, different static PM features have different power savings on different hardware platforms. Also, dynamic PM features or policy/governor have different power savings on different use cases. Knowing the specific set of enabled PM technologies that work well on certain hardware platforms helps you have a better understanding of how much power can actually be saved.

8.1.2 Side effect: latency

One of the major side effects of using PM technologies is the additional latency introduced. The hardware can enter low-power states when all or part of the system is in idle. But when a new request comes, normally the system needs to be restored to the normal working state before serving the new request. The restore from low-power states to normal working state requires time from both hardware and system software. Both aspects contribute to the latency for serving the new request.

There are different definitions of latency from the user's point of view:

- Response latency – Time between request arrival and system starting to deal with the request and provide initial response. Critical to UI based use case or RT use case.
- Completion latency – Time between request arrival and completion of the request.
- Speed up latency – Time between request arrival and the time when the performance/capacity has increased to the point with which the request can be serviced with QoS requirements met.

8.1.3 Side effect: performance

Some static PM features, such as static CPU freq and CPU hotplug reduce the system capability if enabled and impact peak performance. Some dynamic PM features also introduce overhead on the cores for doing the load monitoring and might also impact peak performance.

8.1.4 Rule of thumb

There is a rule of thumb to understand the pros and cons of different PM features. If a PM feature provides more power saving, it normally introduces more latency or more performance impact. An analysis of the benefits and drawbacks is discussed in the Benchmarking section below.

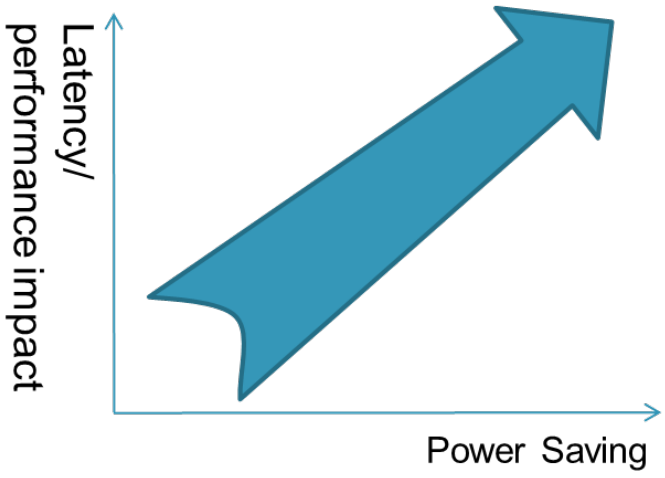


Figure 7. Power saving and the latency/performance impact

8.2 Benchmarking

8.2.1 Power consumption measurement

We have included the power measurement solution by having power monitoring chip designed in some of our development boards. This figure shows how we measure the power consumption and gather the consumption readings.

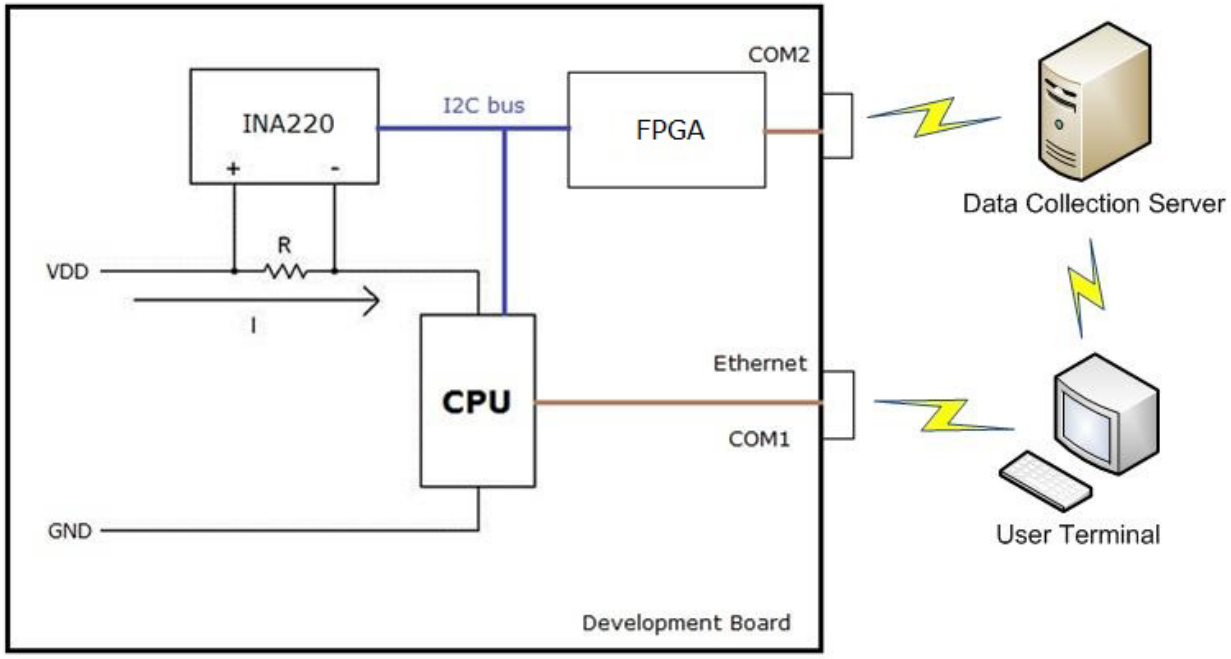


Figure 8. Measuring power consumption

The on-board power monitor chip is connected to an I^2C bus. Reading it with CPU would be the easiest way, but that introduces additional load to the CPU and impact accuracy of power consumption measurement. The FPGA available on most Freescale QDS reference boards provides OCM mechanism, which can read the I^2C power monitor periodically and accumulate the readings. The readings can be exported through OCM console to an external server for further process.

8.2.2 Latency measurement

Hardware latency measurement

We can use an oscilloscope to catch the timing of certain PM-related signals and calculate the latency. Below is an example of measuring the hardware wakeup latency of P1022 sleep.

For resume process, the hardware part begins before the software part. Usually the system is woken up by an interrupt. The start point is the IRQ signal. Take P1022DS as an example: the event button is the wakeup source and is connected to IRQ8. The end point is the ASLEEP signal.

This figure shows the scope snapshot for a resume-from-sleep latency test on P1022DS. The yellow line is the IRQ8, and the red line is the ASLEEP signal.

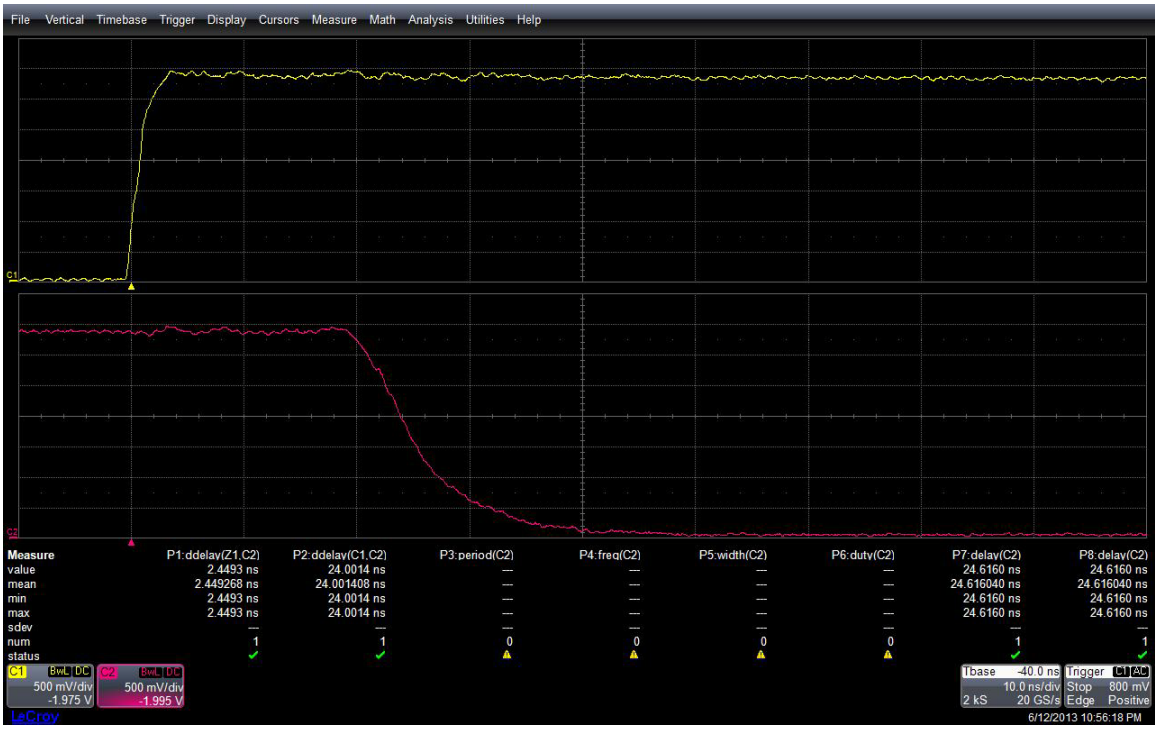


Figure 9. Resume-from-sleep latency test on P1022DS

From the snapshot on oscilloscope we can see the hardware latency of resume is about 20 ns.

Software latency measurement

There are several ways of getting the latencies in software:

1. **Using standard debugging information to get device suspend/resume latency**

   Linux kernel provides a standard mechanism using hrtimer for driver-specific suspend/resume latency measurement.

   There are 3 steps for using this mechanism:

   1. Enable `CONFIG_PM_DEBUG` in kernel configure, recompile the kernel, and boot up using the new kernel
   2. Echo 1 to `/sys/power/pm_print_times` file.
   3. Increase the console log level to debug.
   4. Use command to trigger system suspend and wakeup the system.

   After that, you will find additional information printed during the entrance of system suspend and wake up from suspend. The information includes not only the overall latency of all devices in certain stage, but also how long each device used in that stage. Below is an example of the console output.

```
PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.01 seconds)
done.
Freezing remaining freezable tasks ... (elapsed 0.01
seconds) done.
PM: prepare suspend of devices complete after 1.071 msecs
calling  0.0:02+ @ 3210, parent: fffdf0054.mdio-mux-emi1
call 0.0:02+ returned 0 after 1 usecs
calling  0.0:01+ @ 3210, parent: fffdf0054.mdio-mux-emi1
call 0.0:01+ returned 0 after 0 usecs
calling  fffdf0054.mdio-mux-emi1+ @ 3210, parent:
fffdf0000.board-control
call fffdf0054.mdio-mux-emi1+ returned 0 after 0 usecs
...
PM: suspend of devices complete after 3547.976 msecs
PM: late suspend of devices complete after 0.876 msecs
calling  0001:01:00.0+ @ 3210, parent: 0001:00:00.0
call 0001:01:00.0+ returned 0 after 79 usecs
calling  0001:00:00.0+ @ 3210, parent: pci0001:00
call 0001:00:00.0+ returned 0 after 30 usecs
calling  0000:00:00.0+ @ 3210, parent: pci0000:00
call 0000:00:00.0+ returned 0 after 33 usecs
PM: noirq suspend of devices complete after 26.721 msecs
Disabling non-boot CPUs ...
Enabling non-boot CPUs ...
CPU1 is up
...
CPU23 is up
calling  0000:00:00.0+ @ 3210, parent: pci0000:00
call 0000:00:00.0+ returned 0 after 51 usecs
calling  0001:00:00.0+ @ 3210, parent: pci0001:00
call 0001:00:00.0+ returned 0 after 36 usecs
calling  0001:01:00.0+ @ 3210, parent: 0001:00:00.0
```

```
call 0001:01:00.0+ returned 0 after 79 usecs
PM: noirq resume of devices complete after 26.737 msecs
PM: early resume of devices complete after 0.594 msecs
calling  ffe124000.localbus+ @ 3210, parent: none
call ffe124000.localbus+ returned 0 after 1 usecs
...
calling  0.0:01+ @ 3890, parent: fffdf0054.mdio-mux-emi1
call 0.0:01+ returned 0 after 0 usecs
calling  0.0:02+ @ 3890, parent: fffdf0054.mdio-mux-emi1
call 0.0:02+ returned 0 after 0 usecs
PM: resume of devices complete after 8425.230 msecs
PM: complete resume of devices complete after 0.864 msecs
Restarting tasks ... done.
```

The resolution of the latency is at microsecond level. If the time consumed by the device is very short it just outputs 0 usecs. For a more accurate latency measurement, please use the Time Base measurement.

2. Using Cyclictest to measure task wakeup latency (with certain PM features enabled)

The document and how-to of the utility can be found at:

- https://rt.wiki.kernel.org/index.php/Cyclictest
- http://people.redhat.com/williams/latency-howto/rt-latency-howto.txt

The tool was designed to measure Linux task wakeup latency. It can also be used to measure the impact of PM features on the task wakeup latency when tested with PM features enabled.

The working mechanism to measure the latency introduced by the CPU idle feature is demonstrated in the following figure:

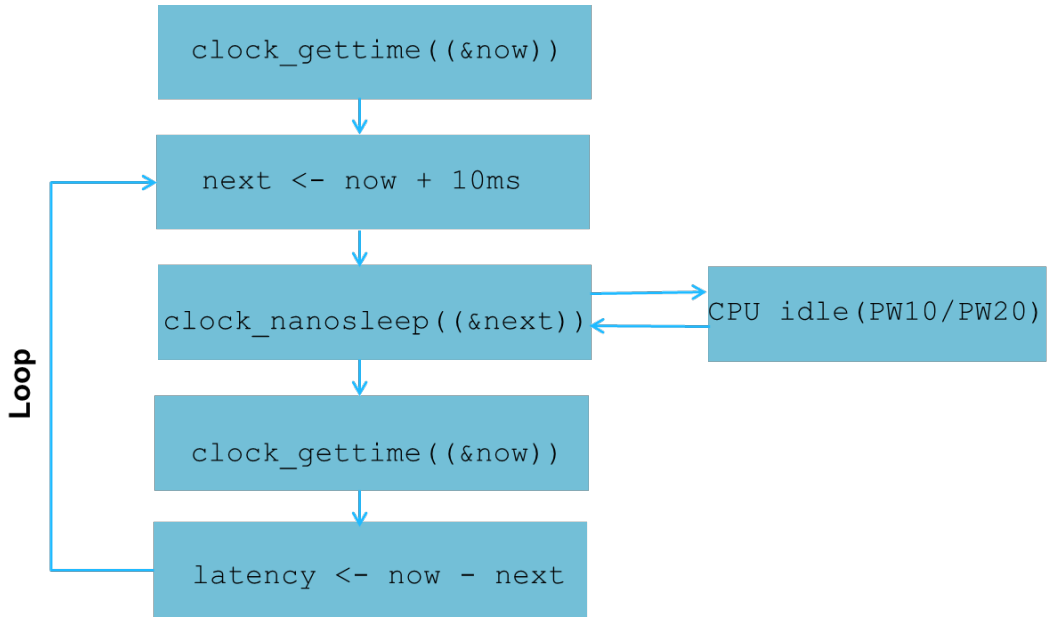


Figure 10. Working mechanism to measure latency

3. Measure latency between two customized points
   a. Using hrtimer API

The Linux hrtimer subsystem in kernel can be used for all software latency measurements. This subsystem is based on Decrementer interrupt and precision is the same as Time base on PowerPC.

Use `ktime_get()` API before and after the measurement points. The difference between the two readings is the latency. Be advised that the `ktime_get()` API also introduces some latency, so if you want to get latency in nanosecond resolution you need to use Time Base or ATB based measurement.

b. Using Time Base (TB) registers

The Time Base is a hardware counter driven at an implementation-dependent frequency. For example, on P1022DS the incremental frequency of the time base is CCB frequency divided by eight and the precision of TB measurement is at the 10 ns level. See the silicon reference manual to get the detailed frequency of TB.

By reading TB registers before and after the test process, we can get the latency result using this formula:

(TB_after – TB_before) / TB frequency

This is not a standard time measuring mechanism in Linux kernel and is only available on Power Architecture®.

c. Using Alternate Time Base (ATB) registers

The most precise latency tests count CPU cycles. The ATB registers record the number of CPU cycles since system startup. By reading this register before and after the measurement points, we can get an accurate latency result. For example, the overhead of reading ATB registers is 8 cycles on P1022DS. The real latency could be calculated using this formula:

(ATB_after – ATB_before – 8) / core frequency

Then we can get the real latency time according to the CPU working frequency.

The hardware limitation of this method is that ATB counting could be affected by low-power states or frequency changing. As in the e6500 core, ATB does not increment when it's in low-power states PW20, PH20, or PH30. In PH30, the value of the ATB resets to 0 when the core is reset to exit PH30. Also, if CPU frequency scaling feature is using dynamic governors, such as ondemand and conservative, this method should be avoided.

Another limitation of this method is that ATB counting is not synchronized among different processors. An ATB reading should only be done on the same processor for both start and end points.

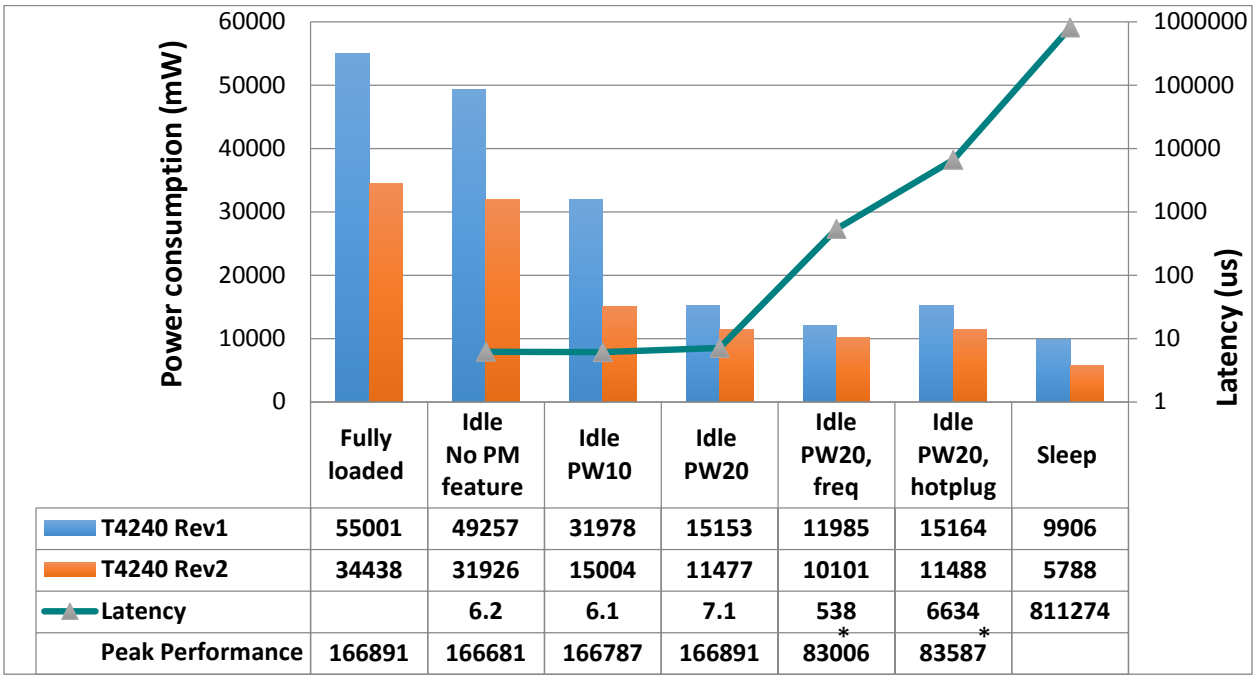
This is also not a standard time measuring mechanism in Linux kernel and is only available on Power architecture.

8.2.3 Performance measurement

The peak performance impact of these features can be measured by running system benchmarking tools, such as CoreMark, lmbench, Dhystone, and so on.

8.2.4 Sample benchmark for T4240QDS

The chart below shows the benchmark result we got for T4240 in our test environment. Note that the consumption measured on another system could be different from the results below, as the environment temperature impacts power consumption and even each individual chip of the same model has its own power characteristics. You need to check the silicon data sheet for the guaranteed power consumption when designing your power supply and cooling system. The power benchmark here is useful to understand the approximate percentage of power savings when a specific PM feature is used.



	Fully loaded	Idle No PM feature	Idle PW10	Idle PW20	Idle PW20, freq	Idle PW20, hotplug	Sleep
T4240 Rev1	55001	49257	31978	15153	11985	15164	9906
T4240 Rev2	34438	31926	15004	11477	10101	11488	5788
Latency		6.2	6.1	7.1	538	6634	811274
Peak Performance	166891	166681	166787	166891	83006*	83587*	

Figure 11. T4240 benchmark results

Hardware configurations of the benchmark:

- T4240 rev1, 6GB DDR@800MHz, CCB@667MHz, CPU@1.67GHz.
- T4240 rev2, 6GB DDR@933MHz, CCB@733MHz, CPU@1.67GHz.

Notes about the measurement done:

- Latency: The latency of freq and hotplug cases is the speed up latency while latency of other cases is the response latency.
- Peak Performance: The highest CoreMark score we can get when certain PM features are enabled. The power consumption of idle cases is measured without this CoreMark running.

The naming of cases in the chart is very simple; further explanation of the cases can be found below:

- Fully loaded: The system is fully utilized with the Coremark tool
- Idle: There is no application running on the system
- No PM features: No PM feature is enabled on the system, which means the Linux system is actually running an idle loop without entering any low-power states.
- PW10/PW20: CPU idle Linux PM feature is enabled, and it is configured to enter PW10 and PW20, respectively.
- Freq: All the cores have used the CPU freq Linux PM feature to reduce to the half frequency.
- Hotplug: Half of the cores are removed using the CPU hotplug Linux feature.
- Sleep: System has entered sleep low-power state using the Linux suspend PM feature.

8.3 Understand your use case

The PM features try to match the load with hardware capability. It is critical to understand the characteristics of the system workload for your specific use case or application. Optimal power saving without impact on the Quality of Service is achieved only when the most suitable PM feature set for your use case is determined.

8.3.1 Load triggering mechanism

First, understand the trigger of workload in your use case. This helps determine if the workload is predictable or not from the user or system point of view. Listed below are a few common categories for the trigger of workload:

- System Application
  - Autonomous application running on the system
  - Characteristic: Load predictable to System
  - Example products: Monitoring(NVR), automatic control, consumer electronics
- Direct User Input
  - Human through Human-machine interface
  - Example trigger: Buttons/switches, Graphic UI, Plug/unplug, Console
  - Characteristic: Load predictable to User or System maintainer
  - Example products: PC, consumer electronics, printing & imaging
- Incoming load from Network
  - Remote peer connected through network
  - Example trigger: New packets, New transactions
  - Characteristic: Load Unpredictable to System or System maintainer
  - Example products: server, networking equipment

8.3.2 Load profile a certain time

The workload at a certain point of time in a system can be defined into the following states:

- Performance wise
  - Fully loaded
  - Partially loaded
  - Completely idle
- Function wise when loaded
  - Partial feature used
  - Full feature used

8.3.3 Load profile within a period of time

The workload can also be characterized within a period of time. Listed below are a few load profiles in a real system:

- Sustained Load: Load that maintains a moderate level of utilization and persists for some time

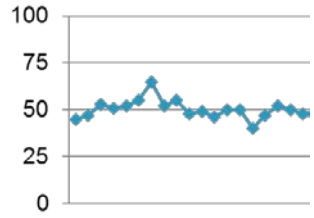


Figure 12. Sustained load

- Burst load: Load requires high utilization when running but completely idle when not running

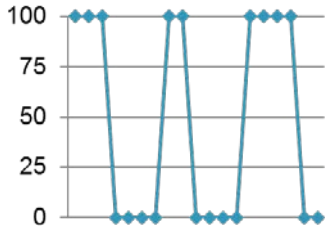


Figure 13. Burst load

- Mixed load: Mixture of burst load and sustained load in a system

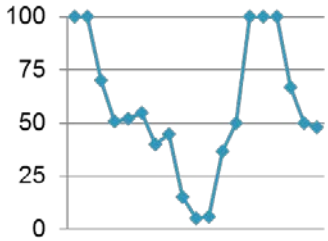


Figure 14. Mixed load

8.3.4 Load Type in Linux System

Workload can also be split into different categories in Linux system according to its running context and task property:

- Kernel space

  System calls – This is in process context and normally associates with I/O generated by application.

  - Kernel threads – This is normally for maintenance work or deferred work in kernel.
  - Interrupts – This is in IRQ context and normally associates with incoming I/O.
- User space
  - Real-time tasks – Processes with real-time scheduling attributes
  - Normal tasks – Processes without real-time scheduling attributes

8.3.5 Quality of Service (QoS) requirements

There are normally expectations on quality of service depending on use case or application. These expectations should be considered when choosing PM technologies. The common QoS expectation could include the following:

- Latency requirement
  - Low latency (RT)
  - Medium latency (Networking)
  - High latency (Human interactive)
- Throughput requirement
- Performance requirement

8.4 Match PM features with use case

Now we will try to use decision trees and a matrix to help choose the correct PM technologies based on the understanding of your use case.

8.4.1 Dynamic PM vs. static PM

First, we need to consider if we want to use static PM features or dynamic PM features.

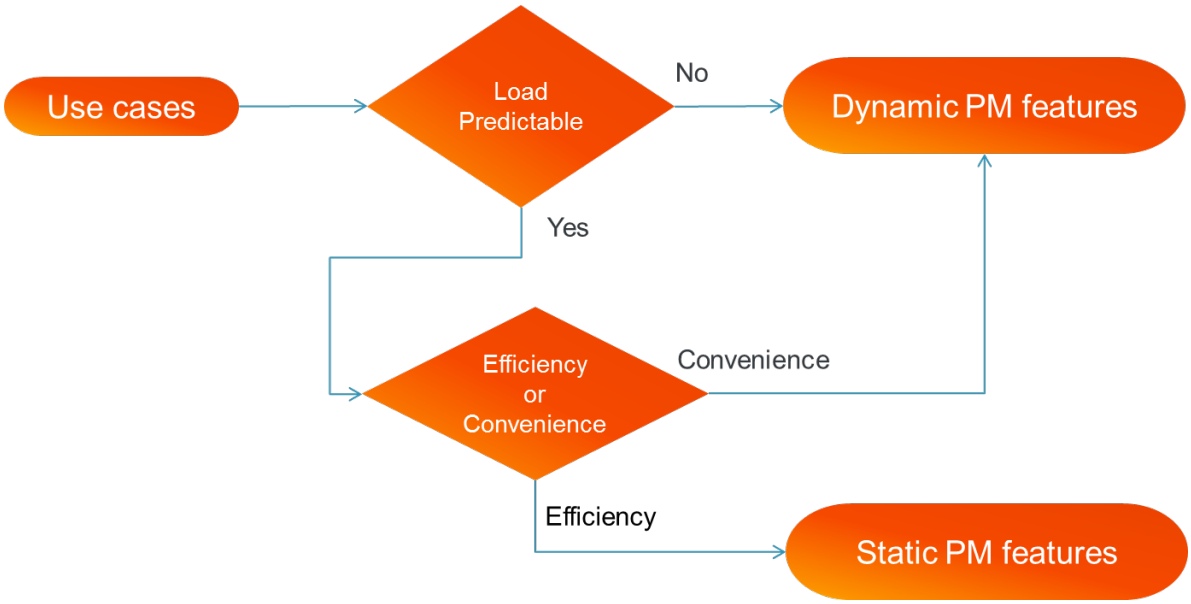


Figure 15. Use cases: static vs. dynamic PM features

Non-predictable load can only benefit from dynamic PM features. If the load can be predicted, normally static PM features are more efficient than the counterpart in dynamic PM features. However, dynamic PM provides more convenience by requiring no change or very little change to the software to take the benefit of power saving.

Predictable load to the application can choose application controlled static PM features and predictable load to the user can choose user controlled static PM features.

8.4.2 Choose static PM features

If we decided to use static PM features, according to the profile of load, the user or application can choose to trigger PM features based on the state machine below.

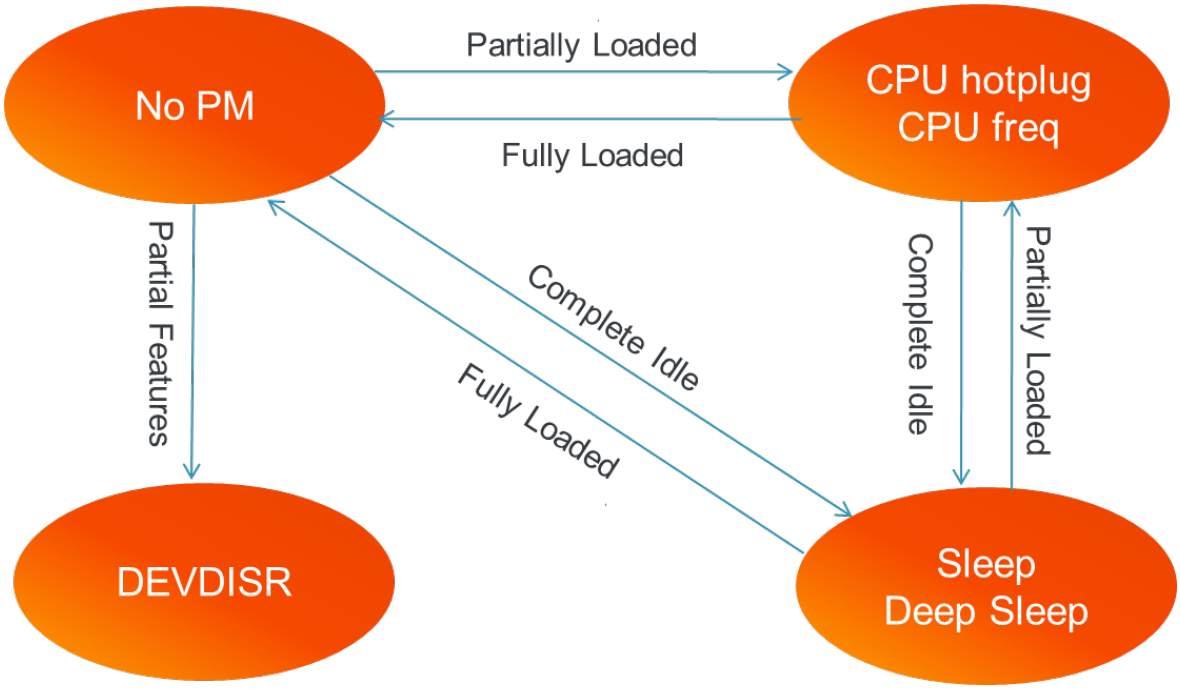


Figure 16. State machine for static PM features

When the system is completely idle, we can choose to enter system-wide low-power states, such as sleep and deep sleep, in which most parts of the system stop working. When the system is partially idle, we can use CPU freq and CPU hotplug features, in which the system is still functioning but with lower peak performance. When the system is only using part of the functionalities provided by the hardware, we can choose to disable the blocks that are not in use.

8.4.3 Choose dynamic PM features

If we decided to use dynamic PM features, the matrix below can help determine the list of features for different load trigger and different load profile.

Table 7. Load trigger and load profile features

	Network Triggered	Application Triggered	Mixed
Burst Load	Autosleep + WoL Autosleep + AR CPUidle CPUfreq (on-demand)	CPUidle CPUfreq (on-demand)	Autosleep + WoL Autosleep + AR CPUidle CPUfreq (on-demand)

Sustained Load	Cascade PM + CPUidle CPUfreq (conservative)	Power Aware scheduler CPUfreq (conservative)	Cascade PM + CPUidle Power Aware scheduler CPUfreq (conservative)
Mixed	Cascade PM + CPUidle CPUfreq (on-demand) CPUfreq (conservative)	CPUidle CPUfreq (on-demand) CPUfreq (conservative)	Cascade PM + CPUidle CPUfreq (on-demand) CPUfreq (conservative)

8.4.4 Filtering with latency requirement

You might get more than one option for PM features. The last step is to match the latency requirement with the latency of the PM features matching your load profile. The benchmark result of different PM features might come with the SDK. Below are the approximate levels of latencies on T4240 as an example:

- Response latency

Table 8. Response latency

PM features	T4240 Latency (us)
No PM(CPU idle disabled)	6.200
CPU idle PW10	6.106
CPU idle PW20	7.098
sleep	811,274 [1]
hibernation	12,601,000 [2]

1. Standard setting, variable with different devices enabled
2. Standard setting, variable with different hard disk and memory size

- Speed up latency

Table 9. Speed up latency

PM features	T4240 Latency (us)
Static CPUfreq	17 ~ 98
CPU hotplug	6,634
Dynamic CPUfreq ondemand	9,531 ~ 19,911 [1]
Dynamic CPUfreq conservative	9,531 ~ 209,911 [1]

1. These are measured with the default setting of the governors. There are tunable parameters for each governor, which impacts the latency.

9 Summary

Increasing power consumption is a big challenge for the entire IT industry. Power management technologies are very helpful in reducing the overall power consumption of electronic devices by matching the runtime capability with runtime workload. Freescale QorIQ networking products provide plenty of hardware low-power states and hardware Power Management (PM) features. These hardware PM capabilities can be utilized in Linux by various standard Linux PM frameworks to achieve both static and dynamic power saving in the system.

Different Linux PM features, different configurations, and different combinations result in different power saving characteristics and side effects on latency and performance. It is critical to match the PM features with your specific use case to achieve the best power saving without unacceptable impact on latency and performance. By benchmarking the PM features available and analyzing the characteristics of system work load, we can get a better match using the principles discussed in this white paper.

10 Revision history

This table provides a revision history for this white paper.

Table 10. Revision history

Rev. Number	Date	Description
0	12/2014	Initial public release

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: QORIQPMWP
Rev. 0
12/2014