



# PXA1826 Openwrt

## Power Management Guide

Not Approved by Document Control.  
For Review Only

MARVELL INTERNAL USE ONLY

DO NOT DISTRIBUTE

FOR <CUSTOMER> USE ONLY

Doc. No. MV-Sxxxxx-xx Rev. y

June 21, 2015, Preliminary

CONFIDENTIAL

Document Classification: Proprietary Information



## Document Conventions

**Note:** Provides related information or information of special importance.**Caution****Caution:** Indicates potential damage to hardware or software, or loss of data.**Warning:** Indicates a risk of personal injury.

## Document Status

Doc Status: Preliminary

Technical Publication: Int. Rev. 0.xx

This document is based on template# MV-S200005-05.

For more information, visit our website at: <http://www.marvell.com>

## Disclaimer

No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose, without the express written permission of Marvell. Marvell retains the right to make changes to this document at any time, without notice. Marvell makes no warranty of any kind, expressed or implied, with regard to any information contained in this document, including, but not limited to, the implied warranties of merchantability or fitness for any particular purpose. Further, Marvell does not warrant the accuracy or completeness of the information, text, graphics, or other items contained within this document.

Marvell products are not designed for use in life-support equipment or applications that would cause a life-threatening situation if any such products failed. Do not use Marvell products in these types of equipment or applications.

With respect to the products described herein, the user or recipient, in the absence of appropriate U.S. government authorization, agrees:

- 1) Not to re-export or release any such information consisting of technology, software or source code controlled for national security reasons by the U.S. Export Control Regulations ("EAR"), to a national of EAR Country Groups D:1 or E:2;
- 2) Not to export the direct product of such technology or such software, to EAR Country Groups D:1 or E:2, if such technology or software and direct products thereof are controlled for national security reasons by the EAR; and,
- 3) In the case of technology controlled for national security reasons under the EAR where the direct product of the technology is a complete plant or component of a plant, not to export to EAR Country Groups D:1 or E:2 the direct product of the plant or major component thereof, if such direct product is controlled for national security reasons by the EAR, or is subject to controls under the U.S. Munitions List ("USML").

At all times hereunder, the recipient of any such information agrees that they shall be deemed to have manually signed this document in connection with their receipt of any such information.

Copyright © 1999–2015. Marvell International Ltd. All rights reserved. M Logo, Marvell, Moving Forward Faster, Alaska, Link Street, Prestera, Virtual Cable Tester, Yukon, Datacom Systems On Silicon, AnyVoltage, DSP Switcher, Feroceon, ZX, ZXSTREAM, ARMADA, Qdeo & Design, QuietVideo, TopDog, TwinD, and Kinoma are registered trademarks of Marvell or its affiliates. Avanta, Avastar, Carrierspan, DragonFly, HyperDuo, HyperScale, Kirkwood, LinkCrypt, Marvell Smart, The World As You See It, Turbosan, and Vmeta are trademarks of Marvell or its affiliates.

Patent(s) Pending—Products identified in this document may be covered by one or more Marvell patents and/or patent applications.

---

# Table of Contents

<b>1</b>	<b>About This Document .....</b>	<b>5</b>
1.1	Purpose.....	5
1.2	Acronyms and Abbreviations .....	5
<b>2</b>	<b>Introduction .....</b>	<b>6</b>
<b>3</b>	<b>Opportunistic Sleep.....</b>	<b>8</b>
3.1	Suspend-to-RAM .....	8
3.2	Wake Source and Wakelock.....	10
<b>4</b>	<b>PM QoS .....</b>	<b>14</b>
<b>5</b>	<b>cpufreq.....</b>	<b>16</b>
5.1	CPUfreq QoS .....	18
5.1.1	PM QoS user space interface .....	19
<b>6</b>	<b>devfreq for DDR.....</b>	<b>20</b>
6.1	devfreq PM QoS interface .....	20
<b>7</b>	<b>cpuidle .....</b>	<b>22</b>
7.1	cpuidle QoS Constraints.....	25
7.2	Tickless idle.....	26
<b>8</b>	<b>runtime PM.....</b>	<b>27</b>
8.1	Runtime PM V.S. autosleep.....	28
8.2	Hardware v.s. Software .....	29
<b>9</b>	<b>Thermal Management.....</b>	<b>30</b>
<b>10</b>	<b>WiFi Power Management.....</b>	<b>31</b>



## List of Tables

Table 3: Acronyms and Abbreviations ..... 5

Table 3: Revision History ..... 32

## List of Figures

No table of figures entries found.

# 1 About This Document

## 1.1 Purpose

This document explains the design of PXA1826 power management mechanisms and their usage guide.

It contains the low-power modes, the dynamic voltage and frequency change, the QoS constraints for both of them, and the thermal management. What's more, the wifi uAP power management is also addressed.

## 1.2 Acronyms and Abbreviations

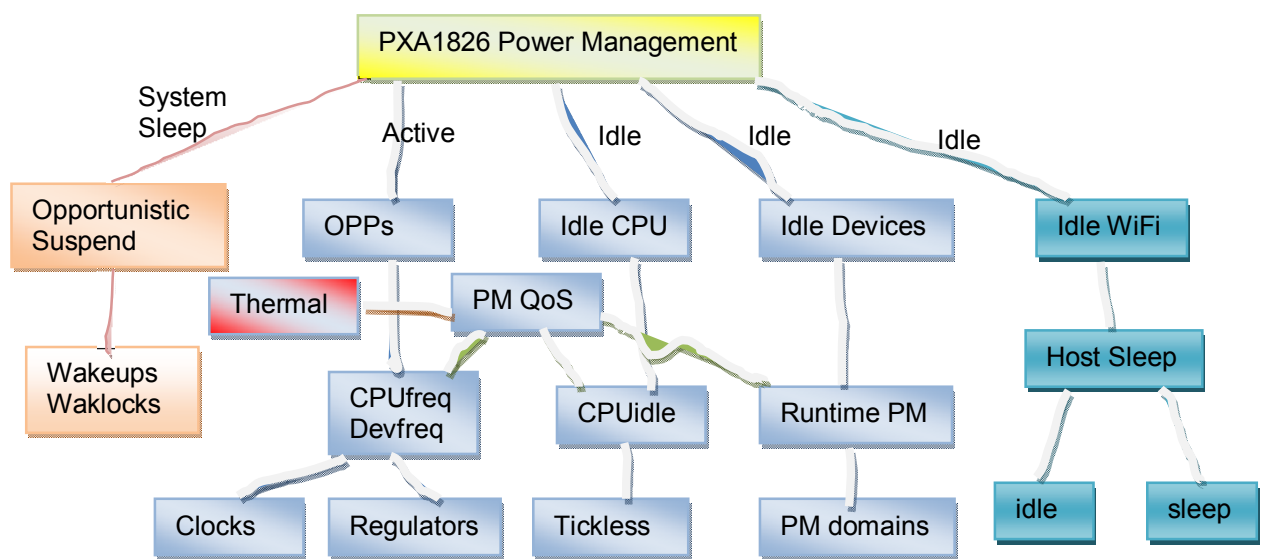
**Table 1: Acronyms and Abbreviations**

Acronym	Description
UE	User equipment, target
AP	Application processor
CP	Communication processor
OPP	Operating Performance Point

## 2 Introduction

This document includes the power and thermal mechanisms supported and all information required to use them in kernel space and user space (if applicable).

Power management is an evolving part of Linux kernel. PXA1826 solution is based on openwrt with kernel 3.10, below is the main infrastructure of the whole PXA1826 power management solution.



Power usage has to be managed throughout all the components of the system: kernel, low-level software infrastructure, and applications. Here are the various building blocks :

- system suspend to RAM and resume, autosleep based on opportunistic suspend mechanism.
- cpufreq and devfreq (with clock framework, regulators framework), supporting to choose the best one among all the voltage and frequency operating modes (OPPs) supported by PXA1826 system, including CPU and DDR.
- cpuidle, when your CPU is idle, it can switch to deeper and deeper sleep modes, consuming less power.
- tickless idle, stop periodic tick when idle, only wakes for next "event" or interrupt
- runtime power management (power domain)
- pm QoS
- thermal framework

- WiFi subsystem power management and its host-sleep mechanism for interaction to PXA1826

# 3 Opportunistic Sleep

When there is opportunity to go to low power state (“opportunistic” sleep), system tries to hit suspend. It’s an aggressive suspend strategy, where a hardware block should be powered on only when needed.

Drivers enter low power states as part of system-wide low-power states (suspend-to-RAM). The device tree is walked in a bottom-up order to suspend devices. A top-down order is used to resume those devices. The ordering of the device tree is defined by the order in which devices get registered: a child can never be registered, probed or resumed before its parent; and can’t be removed or suspended after that parent. The policy is that the device tree should match hardware bus topology.

The device, bus, and class drivers collaborate in entering low power states, have their own suspend/resume methods, power down hardware and software subsystems, and wake up without loss of data; some drivers manage wakeup events, which let the system to wake up.

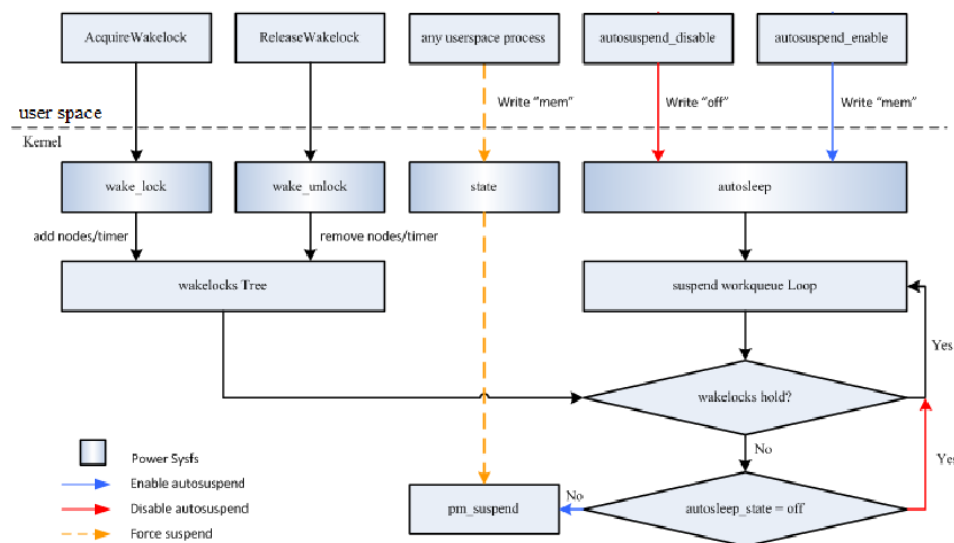
Wake up events, e.g., could be:

- external event on GPIO
- Onkey, wifi client association
- timer alarm

## 3.1 Suspend-to-RAM

Knowing in advance that the whole system is not going to be used in the near future, turn off everything (possibly by force) except for the RAM chips.

Clocks may be gated except for low frequency clock in wakeup domain. Voltages may be off, except for RAM and wakeup domain.





Takes longer then going to idle.

Takes time for clock generators, power regulators to stabilize on resume.

Freezing/thawing of threads.

When the system goes into a sleep state, each device's driver is asked to suspend the device. Wakeup-enabled devices will usually stay partly functional in order to wake the system.

When the system leaves that low-power state, the device's driver is asked to resume it by returning it to full power. The suspend and resume operations always go in paired, and both are multi-phase operations.

**Suspend Sequence**(pm\_suspend() kernel/power/suspend.c):

1. Call notifiers (while user space is still there).
2. Freeze tasks.
3. 1st phase of suspending devices (.suspend() callbacks).
4. Disable device interrupts.
5. 2nd phase of suspending devices (.suspend\_noirq() callbacks).
6. Disable non-boot CPUs (using CPU hot-plug).
7. Turn interrupts off.
8. Execute system core callbacks.
9. Turn off the CPU.

Freezing of tasks is needed to prevent below actions:

- filesystems changes while system state being saved;
- memory allocation to threads, as about 50% of RAM is needed to create hibernation image;
- userspace processes and some kernel threads interaction with suspended devices.

Power off mode is the last step in suspend sequence when power from all power domains except of a wakeup power domain.

**Resume Sequence:**

1. Wakeup signal.
2. Run boot CPU's wakeup code.
3. Execute system core callbacks.
4. Turn interrupts on.
5. Enable non-boot CPUs (using CPU hot-plug).
6. 1st phase of resuming devices (.resume\_noirq() callbacks).
7. Enable device interrupts.
8. 2nd phase of suspending devices (.resume() callbacks).
9. Thaw tasks.

10. Call notifiers (when user space is back).

**How to send system to suspend:**

```
# echo mem > /sys/power/autosleep
```

Note: System never sleeps if there is USB cable or charger connected.

## 3.2 Wake Source and Wakelock

The system stays in suspend state most of the time and should only be awake if absolutely necessary, i.e. if there is at least one system component that remains active. Sometimes it is needed to keep system awake to keep the device responsive to user interaction. Wakelock is used to prevent system from going to suspend, also known as suspend blocker. Defined in simple terms, a wake lock is a binary kernel object that is acquired by a subsystem whenever it needs to keep the system awake. The kernel monitors all wake locks and executes a system suspend only when none of the wake locks are held.. If – at any point during the suspend procedure – any of the subsystems requires the system to stay awake, it would acquire its wake lock which would immediately abort the suspend in progress. The latter mechanism is used in particular by wakeup interrupts to prevent racing with a suspend request currently in progress.

The kernel uses a wakeup\_source object in a device's power management block (struct dev\_pm\_info) to avoid race conditions between wakeup and suspend events. To manipulate the device's wakeup\_source object, the following kernel functions were added:

- device\_init\_wakeup() – when called with enable==1, initialize the device's wakeup\_source, when called with enable==0, disable the device's wakeup\_sorce
- pm\_stay\_away() – notify the system that a device is processing a wakeup event
- pm\_relax() – notify the system that a device is no longer processing a wakeup event
- pm\_wakeup\_event() – notify the system that the device will be processing the wakeup event until timeout

All of these functions have an argument representing the device's struct device object, indicating the device to which a wakeup\_source and wakeup event are associated. The struct wakeup\_source embedded in the device's struct dev\_pm\_info field

The autosleep (a.k.a. opportunistic suspend) functionality will automatically trigger a suspend whenever there are no wakeup sources held. It works in conjunction with driver suspend/resume (runtime suspend and auto-suspend) hooks to implement power-saving modes for the system. As long as there are suspend blockers outstanding, the system will not suspend. Autosleep helps to save power by suspending the entire system whenever nothing is going on. Note that opportunistic suspend can happen even when processes are running in user space. In the absence of a suspend blocker, any computation underway is not considered to be important enough to keep the system awake. This behavior is a form of defense against poorly-written applications which might, otherwise, drain a system's battery in a short period of time.

The kernel also has functions that manipulate the wakeup\_source object directly:

- `wakeup_source_init()` – initialize a wakeup source object
- `wakeup_source_trash()` – de-initialize a wakeup source object
- `__pm_stay_away()` – notify the system that a wakeup event is being processed
- `__pm_relax()` – notify the system that a wakeup event is no longer being processed
- `__pm_wakeup_event()` – notify the system that a wakeup event will be processed until timeout

Here is an example of how wake source is manipulated directly in kernel (The + is the recommended usage, the “-“ items are the old Android implementation usage).

[drivers/staging/android/alarm-dev.c](#)

@@ -25,17 +25,6 @@

#include <linux/alarmtimer.h>  
#include "android\_alarm.h"

/\* XXX - Hack out wakelocks, while they are out of tree \*/

-struct wake\_lock {

- int i;

-};

-#define wake\_lock(x)

-#define wake\_lock\_timeout(x, y)

-#define wake\_unlock(x)

```

-#define WAKE_LOCK_SUSPEND 0
-#define wake_lock_init(x, y, z) ((x)->i = 1)
-#define wake_lock_destroy(x)
-
#define ANDROID_ALARM_PRINT_INFO (1U << 0)
#define ANDROID_ALARM_PRINT_IO (1U << 1)
#define ANDROID_ALARM_PRINT_INT (1U << 2)
@@ -61,7 +50,7 @@ module_param_named(debug_mask, debug_mask, int, S_IRUGO | S_IWUSR | S_IWGRP);

static int alarm_opened;
static DEFINE_SPINLOCK(alarm_slock);
-#static struct wake_lock alarm_wake_lock;
+static struct wakeup_source alarm_wake_lock;
static DECLARE_WAIT_QUEUE_HEAD(alarm_wait_queue);
static uint32_t alarm_pending;
static uint32_t alarm_enabled;
@@ -154,7 +143,7 @@ static long alarm_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
    if (alarm_pending) {
        alarm_pending &= ~alarm_type_mask;
        if (!alarm_pending && !wait_pending)
            wake_unlock(&alarm_wake_lock);
+        __pm_relax(&alarm_wake_lock);
    }
    alarm_enabled &= ~alarm_type_mask;
    spin_unlock_irqrestore(&alarm_slock, flags);
@@ -192,7 +181,7 @@ from_old_alarm_set:
    spin_lock_irqsave(&alarm_slock, flags);
    pr_alarm(10, "alarm wait\n");
    if (!alarm_pending && wait_pending) {
        wake_unlock(&alarm_wake_lock);
+        __pm_relax(&alarm_wake_lock);
        wait_pending = 0;
    }
    spin_unlock_irqrestore(&alarm_slock, flags);
@@ -284,7 +273,7 @@ static int alarm_release(struct inode *inode, struct file *file)
    if (alarm_pending)
        pr_alarm(INFO, "alarm_release: clear "
            "pending alarms %x\n", alarm_pending);
-        wake_unlock(&alarm_wake_lock);
+        __pm_relax(&alarm_wake_lock);
    wait_pending = 0;
    alarm_pending = 0;
}
@@ -302,7 +291,7 @@ static void devalarm_triggered(struct devalarm *alarm)
    pr_alarm(INT, "devalarm_triggered type %d\n", alarm->type);
    spin_lock_irqsave(&alarm_slock, flags);
    if (alarm_enabled & alarm_type_mask) {
        wake_lock_timeout(&alarm_wake_lock, 5 * HZ);
+        __pm_wakeup_event(&alarm_wake_lock, 5000); /* 5secs */
        alarm_enabled &= ~alarm_type_mask;
        alarm_pending |= alarm_type_mask;
        wake_up(&alarm_wait_queue);
@@ -368,15 +357,14 @@ static int __init alarm_dev_init(void)
    alarms[i].u.hrt.function = devalarm_hrthandler;
}

-        wake_lock_init(&alarm_wake_lock, WAKE_LOCK_SUSPEND, "alarm");
-
+        wakeup_source_init(&alarm_wake_lock, "alarm");
    return 0;
}

static void __exit alarm_dev_exit(void)
{
    misc_deregister(&alarm_device);
-        wake_lock_destroy(&alarm_wake_lock);
+        wakeup_source_trash(&alarm_wake_lock);
}

```

---

```
module_init(alarm_dev_init);
```

Wake sources (For history reason, called wake locks in user space) could also be manipulated from user space through the /sys/power interface using the following files:

- /sys/power/wake\_lock – writing a string to this file would create/acquire a wake lock with that name

Take wakelock:

```
# echo mylock > /sys/power/wake_lock
```

Or

```
# echo mylock timeout_ns > /sys/power/wake_lock
```

- /sys/power/wake\_unlock – writing a string to this file would release a wake lock with that name

Release wakelock:

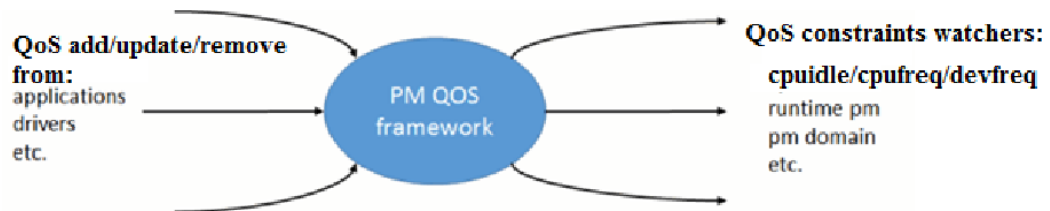
```
# echo mylock > /sys/power/wake_unlock
```

# 4 PM QoS

Power management quality of service.

This interface provides a kernel and user mode interface for registering performance expectations by drivers, subsystems and user space applications on one of the parameters.

For each device a list of performance requests is maintained along with an aggregated target value. The aggregated target value is updated with changes to the request list or elements of the list. Typically the aggregated target value is simply the max or min of the request values held in the parameter list elements.



- PM QoS provides a coordination mechanism between the hardware providing a power managed resource and users with performance needs
- It is a kernel infrastructure to facilitate the communication of latency and throughput needs among devices, system, and users.
- Automatic power management, at the driver level, is enabled with coordinated device throttling given the QoS expectations on that device.

Two different PM QoS frameworks are available:

- PM QoS classes for `cpu_dma_latency`, `memory_throughput`, `network_latency`, `network_throughput`. This type of QoS is extended to support `cpufreq` and `devfreq` constraints in our solution, will addressed in later in `cpufreq` and `devfreq` part.
- the per-device PM QoS framework provides the API to manage the per-device latency constraints.

So each device driver can use PM QoS framework to ask power managed resource. Depending on the QoS class is requested by device driver, the request is delivered by the PM QoS framework to the registered recipients.

Examples:

PM QoS and cpuidle

The `cpu_dma_latency` requests are delivered to the `cpuidle` framework, based on them `cpuidle` makes decision to what next C state it will be allowed to put cpu. To receive that request the `cpuidle` framework have registered with PM QoS framework for that QoS class. The `cpu_dma_latency` is a minimum latency needed by device driver to process requests in time.

PM QoS and platform bus (interconnect).

The memory\_ throughput class frequently used to ask platform bus throughput. The recipient in that case is platform bus and users are device drivers. Depending on the asked maximum throughput in the request list the proper frequency is set for platform bus.

per-device PM QoS and runtime\_pm.

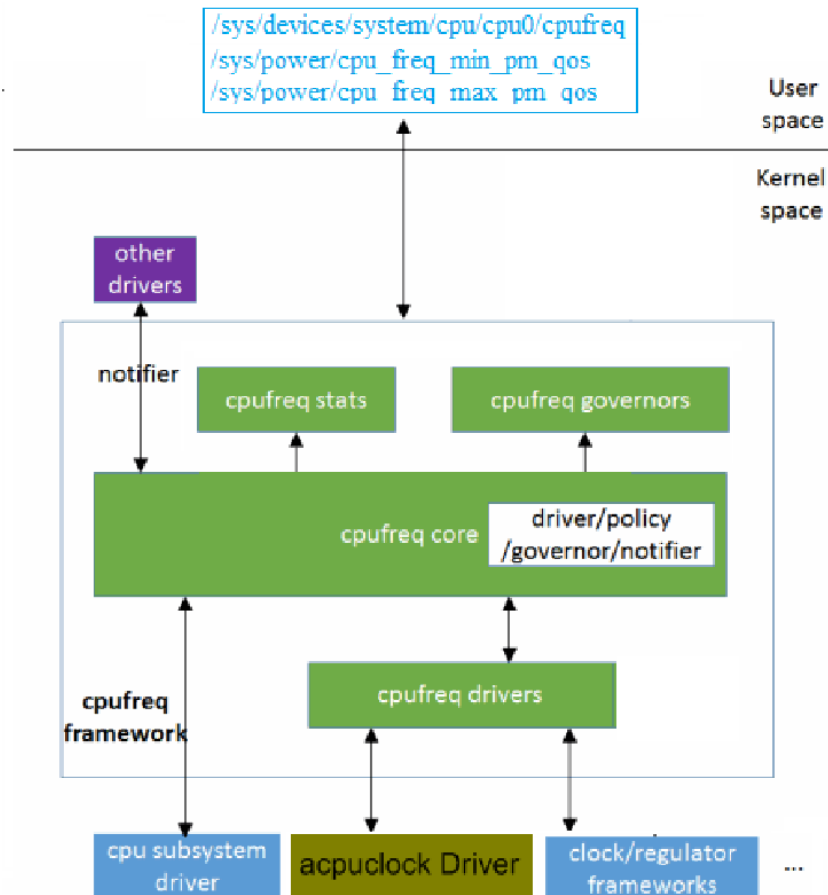
The devices often support a range of runtime power states, which might use names such as "off", "sleep", "idle", "active", and so on. As in case with C states in cpuidle, different power states for device have different latencies based on what the next power state is chosen.

Devices are often included in power domains that means the power for a device is gained/retained only when all devices in that power domain are inactive, in other words they have common power states. Each power state has its own latency and this latency is main criteria to choose the next power state.

As devices are included in power domains, the next power state in power domain is chosen depending on biggest latency requested by all devices in the power domain. The recipient in that case is power domain and users are device drivers.

# 5 cpufreq

cpufreq refers to the kernel infrastructure that implements CPU frequency scaling. This technology enables the operating system to scale the CPU speed up or down in order to save power. CPU frequencies can be scaled automatically depending on the system load, or manually by user space programs.



- Cpufreq framework is used to trigger core freq-chg with low level clock operation support.
- Several common CPUfreq governors(ondemand, interactive, etc) are provided by open source to calculate the workload of the cpu every sampling window and get the max workload as reference for next frequency.
- Several file nodes could be used by userspace to limit the min/max cpu frequency or adjust governor or profiling parameters to get better power and performance balance.
- Kernel side also provides min/max cpu qos constraints for drivers to limit the core frequency.



- Two kinds of cpu frequency qos constraints are provided. Qos min is used to limit the min frequency of cpu; QoS max is used to limit the max frequency of cpu.
- They are both used by kernel driver and have lower priority than usespace requirement from cpufreq governor parameters scaling\_min\_freq and scaling\_max\_freq.
- It is also not recommended for driver to use qos max to limit max cpu freq due to performance impact. Driver/userspace requesting the frequency change in short latency(boosting) can limit the min frequency of cpu via QoS

There are various kernel governors available for use with the cpufreq subsystem (/Documentation/cpu-freq/governors.txt). These governors set the processor frequency based on certain criteria; some dynamically change the frequency as inputs are changed either by the system or the user. Here are several typical governors, and the interactive governor is set as default after system boot up.

Userspace governor: Manual frequencies

Next there is the userspace governor, which allows you to select and set a frequency manually. This governor also works with processor frequency daemons running in userspace to control frequency. This governor is useful for setting a unique power policy that is not preset or available from the other governors; you can also use it to experiment with policies.

Note that the userspace governor itself does not dynamically change the frequency; rather, it allows you or a userspace program to dynamically select the processor frequency.

Ondemand governor: Frequency change based on processor use

The ondemand governor was the first in-kernel governor to dynamically change processor frequency based on processor utilization. The ondemand governor checks the processor utilization and if it exceeds the threshold, the governor will set the frequency to the highest available. If the governor finds the utilization to be less than the threshold, it steps down the frequency to the next available. If the system continues to be underutilized, the governor will continue stepping down the frequency until the lowest available is set.

You can control the range of frequencies available, the rate at which the governor checks utilization on the system, and the utilization threshold.

Interactive Governor: Enhanced Ondemand

Much like the OnDemand governor, the Interactive governor dynamically scales CPU clockspeed in response to the workload placed on the CPU by the user. This is where the similarities end. Interactive is significantly more responsive than ondemand, because it's faster at scaling to maximum frequency.

You can control the range of frequencies available, the rate at which the governor checks utilization on the system, the utilization thresholds, and the frequency step rate.

```
$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_governors
conservative ondemand powersave userspace interactive performance
```

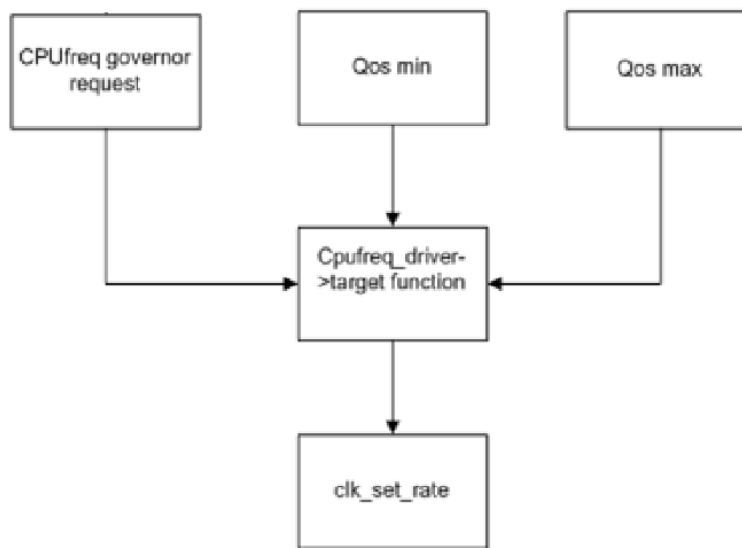
```
$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies
312000 416000 624000 832000 1248000
```

Dynamic voltage and frequency scaling (DVFS) is based on cpufreq/devfreq and integrated to clock framework to trigger voltage change.

Each components(clock node) such as core/ddr that has voltage requirement should register its frequency to voltage requirement at init stage. DVFS will use voltage up-to model to implement and centralize to manage max voltage requirement from all components.

Note: the duty cycle statistics interface is /sys/kernel/debug/pxa/stat/cpu\_dc\_stat

## 5.1 CPUfreq Qos



- Cpufreq governor request/Qos min notifier/Qos max notifier all call cpufreq\_driver->target function to change core frequency.
- Cpufreq\_driver->target function usually will call cpufreq helper function to find a recommend frequency that already considered the policy->min and policy->max. And the same time target function also should get Qos min and max request to made final decision.
- Code is like below, it will make sure final frequency will NOT lower than Qos\_min/policy->min or higher than Qos\_max/policy->max.

```

target_freq = max((unsigned int)pm_qos_request(PM_QOS_CPU_FREQ_MIN),
target_freq);

target_freq = min((unsigned int)pm_qos_request(PM_QOS_CPU_FREQ_MAX),
target_freq);

cpufreq_frequency_table_target(policy, freq_table, target_freq, relation, &index); --
helper function considers policy min&max

target_freq = freq_table[index].frequency;
  
```

There are two sets of QoS interfaces for user space. One is the original misc device based interfaces (e.g. /dev/cpu\_freq\*, /dev/ddr\_devfreq\*) can only have one user at same time. The other is the new developed sysfile interface which can support multiple requests.

### 5.1.1 PM QoS user space interface

```
root@OpenWrt:/# ls /sys/power/cpu_freq_*qos
/sys/power/cpu_freq_max_pm_qos
/sys/power/cpu_freq_min_pm_qos
```

```
root@OpenWrt:/# ls /sys/power/cpu_freq_*unqos
/sys/power/cpu_freq_max_pm_unqos
/sys/power/cpu_freq_min_pm_unqos
```

To add a cpu freq qos constraint in appA,

```
# echo appA 832000 > /sys/power/cpu_freq_min_pm_qos
```

With timeout in ns unit

```
# echo appA 832000 2000000 > /sys/power/cpu_freq_min_pm_qos
```

To update a existing qos constraint,

```
# echo appA 624000 > /sys/power/cpu_freq_min_pm_qos
```

To remove the cpu freq qos constraint (only works correctly for a existing constraint)

```
# echo appA > /sys/power/cpu_freq_min_pm_unqos
```

To check the existing constraints from user space

```
# cat /sys/power/cpu_freq_min_pm_qos
```

To check details of all the active QoS constraints from both user space and kernel space,

```
# cat /sys/kernel/debug/cpufreq_qos
```

# 6

## devfreq for DDR

devfreq is a generic DVFS framework with device-specific OPPs

With OPPs, a device may have multiple operable frequency and voltage sets. However, there can be multiple possible operable sets and a system will need to choose one from them. In order to reduce the power consumption (by reducing frequency and voltage) without affecting the performance too much, a Dynamic Voltage and Frequency Scaling (DVFS) scheme may be used.

DVFS is a technique whereby the frequency and supplied voltage of a device is adjusted on-the-fly. DVFS usually sets the frequency as low as possible with given conditions (such as QoS assurance) and adjusts voltage according to the chosen frequency in order to reduce power consumption and heat dissipation.

Normally, DVFS mechanism controls frequency based on the demand for the device, and then, chooses voltage based on the chosen frequency. devfreq also controls the frequency based on the governor's frequency recommendation and let OPP pick up the pair of frequency and voltage based on the recommended frequency. Then, the chosen OPP is passed to device driver's "target" callback.

The generic DVFS for devices, devfreq, is quite similar with /drivers/cpufreq.

It doesn't support governor dynamic changing, the profile governor of each device is determined by kernel build configuration

Simple\_ondemand governor is used to collect DDR data ratio in one sample window to determine next windows DDR frequency, so that DDR freq is adaptive to its workload

Note: the duty cycle statistics interface is /sys/kernel/debug/pxa/stat/ddr\_dc\_stat

### 6.1 devfreq PM QoS interface

PM QoS framework is extended to support devfreq QoS constraint, it's similar to cpufreq QoS constraints. Here is the example on how to use it in user space.

```
root@OpenWrt:~# ls /sys/power/ddr_devfreq_*_qos
/sys/power/ddr_devfreq_max_pm_qos
/sys/power/ddr_devfreq_min_pm_qos
```

```
root@OpenWrt:~# ls /sys/power/ddr_devfreq_*_unqos
/sys/power/ddr_devfreq_max_pm_unqos
```

---

```
/sys/power/ddr_devfreq_min_pm_unqos
```

To add a ddr freq qos constraint in appA,

```
# echo appA 400000 > /sys/power/ddr_devfreq_min_pm_qos
```

With timeout in ns unit

```
# echo appA 400000 2000000 > /sys/power/ddr_devfreq_min_pm_qos
```

To update a existing qos constraint,

```
# echo appA 312000 > /sys/power/ddr_devfreq_min_pm_qos
```

To remove the ddr freq qos constraint (only works correctly for a existing constraint)

```
# echo appA > /sys/power/ddr_devfreq_min_pm_qos
```

To check the existing constraints from user space

```
# cat /sys/power/ddr_devfreq_min_pm_qos
```

To check details of all the active QoS constraints from both user space and kernel space,

```
# cat /sys/kernel/debug/ddrfreq_qos
```

# 7

## cpuidle

Processor power management can be classified into two classes:

- Processor active – various states a processor can be in while actively executing and retiring instructions(processors running at different frequencies)
- Processor idle – various states a processor can be in while it is idle and not retiring any instructions

If there is no tasks to run, scheduler chooses idle task.

```
for (;;) {  
    do_noting();  
}
```

But it just wastes power. CPUs are capable to be suspended (power down, sleep, wfi instructions). What processor does in idle is architecture dependent. There are usually several levels of sleep varies depending on the hardware, could be for example:

cpu clock off

cpu clock off, voltage reduced to retention voltage

cpu clock off, voltage off, content of memory retained, voltage on memory is on

Processors support multiple processor idle states with varying amounts of power consumed in those idle states. Each such state will have an entry-exit latency associated with it. In typical system usage models, processor(s) spend a lot of their time idling. Thus any power saved when system is idle will have big returns in terms of battery life, heat generated in the system, need for cooling, etc

Each of the cpu idle states (C-states) is characterized by its power consumption and wakeup latency, and also based on preservation of the processor state, while in this C-state. A platform can dynamically change the number of C-states supported, based on different platform parameters such as whether it is running on battery or AC power.

The cpuidle framework consists of two key components:

- A governor that decides the target C-state of the system.
- A driver that implements the functions to transition to target C-state.

cpuidle governors implement the policy side of cpuidle. The kernel allows the existence of multiple governors at any given time, though only one will be in control of a given CPU at any time. When making its decision, the governor should pay attention to the current latency requirements expressed by other code in the system. The mechanism for the registration of these requirements is the "pm\_qos" subsystem. A number of quality-of-service requirements can be registered with this system, but the one most relevant for cpuidle governors is the CPU latency requirement. That information can be obtained with:

```
#include <linux/pm_qos_params.h>
```

```
int max_latency = pm_qos_requirement(PM_QOS_CPU_DMA_LATENCY);
```

The cpuidle driver registers itself with the framework during boot-up and populates the C-states with exit latency, target residency (minimum period for which the state should be maintained for it to be useful) and flag to check the bus activity.

A C-state is used to identify the power state supported through the cpu idle loop. Each C-state is characterized by its:

- Power consumption
- Wakeup latency
- Preservation of processor state while in 'the' state.

The cpuidle governor makes decision to choose C state based on that:

- Scheduler knows when the system is idle, as it had chosen idle task.
- Scheduler knows when the next timer expires.
- Wake up latency is known from PM QoS
- cpu latency is also known

What C states do we have:

```
root@OpenWrt:/# ls /sys/devices/system/cpu/cpu0/cpuidle/
state0 state1 state2 state3
```

The description of the C states on PXA1826:

```
$ cat /sys/devices/system/cpu/cpu0/cpuidle/state*/desc
```

C1: Core internal clock gated

C2: Core power down

D1p: AP idle state

D1: Chip idle state

Corresponds to C1, C2, C3, C4

What is inside of a state:

```
root@OpenWrt:/# ls /sys/devices/system/cpu/cpu0/cpuidle/state3
desc  disable latency name  power  time  usage
```

The name of a state:

```
root@OpenWrt:/# cat /sys/devices/system/cpu/cpu0/cpuidle/state3/name
D1
```

How much time is spent in a state:

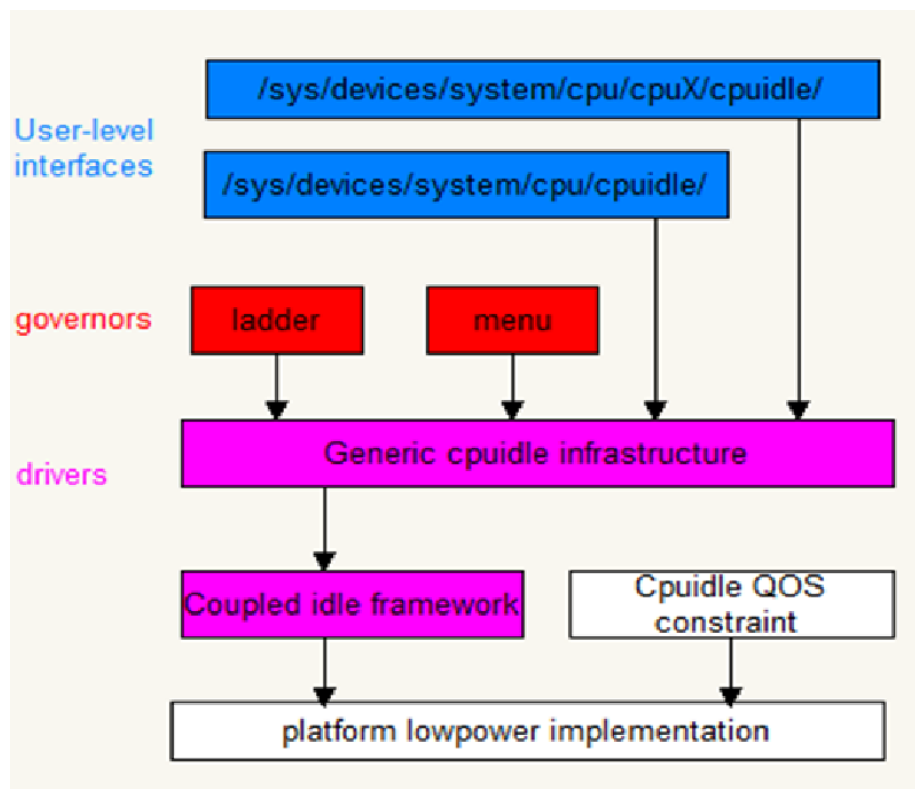
```
# cat /sys/devices/system/cpu/cpu0/cpuidle/state3/time
141650728838
```

How many times we hit a state:

```
#cat /sys/devices/system/cpu/cpu0/cpuidle/state3/usage
747250
```

How to disable a state:

```
# echo 1 > /sys/devices/system/cpu/cpu0/cpuidle/state3/disable
```



- menu and ladder governors aims to select proper low power state when system is idle. Only menu governor is used because the tickles idle is enabled.
  - ladder - steps down or up sleep states one at a time depending on the time spent in the last idle period. It works well with a regular timer tick, but not with dynamic tick.
  - menu - selects sleep state based on expected idle time. Works well with dynamic tick systems.
- cpuidle QoS constraints are used for components that block low power modes.



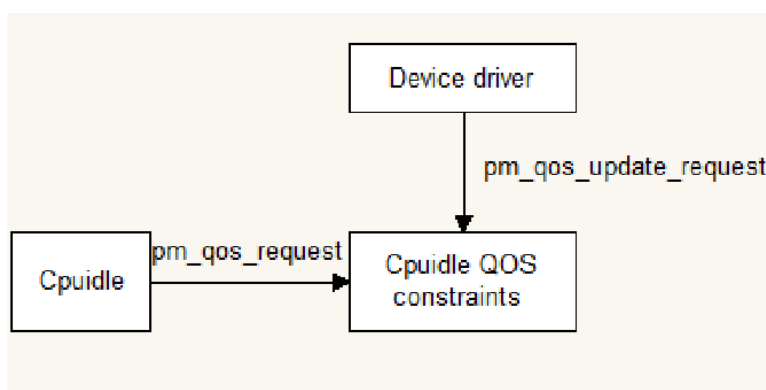
## 7.1 cpuidle QoS Constraints

We have Three kinds of constraints because AXI, DDR and VCTCXO are shut down in order (D1P, D1, D2).

- Peripherals that are accessing AXI fabric need a constraint to block D1P since AXI clock will be off in that or deeper mode. Especially some peripherals interrupts rely on axi fabric, such as USB.
- Peripherals that are accessing DDR need a constraint to block D1 since DDR clock will be off in that or deeper mode.
- Peripherals that use VCTCXO as clock source need a constraint to block D2 since VCTCXO is off in that mode.

PM QoS will be used for these three kinds of constraints.

- One new PM\_QOS\_CPUIDLE\_BLOCK will be added.
- Three different QoS value:  
PM\_QOS\_CPUIDLE\_BLOCK\_AXI\_VALUE ,  
PM\_QOS\_CPUIDLE\_BLOCK\_DDR\_VALUE and  
PM\_QOS\_CPUIDLE\_BLOCK\_VCTCXO\_VALUE.



Devices drivers interface for QoS operations:

pm\_qos\_add\_request

pm\_qos\_update\_request

pm\_qos\_remove\_request

Devices	SDH	USB	SSP	Serial	I2C	LCD	GEU	SPI	KEYPAD
D1P QOS	Yes	Yes	Yes	Yes		Yes	Yes		
D1 QOS					Yes			Yes	Yes

## 7.2 Tickless idle

Tickless kernel, dynamic ticks or NO\_HZ is a config option that enables a kernel to run without a regular timer tick. The timer tick is a timer interrupt that is usually generated HZ times per second, with the value of HZ being set at compile time and varying between around 100 to 1500. Running without a timer tick means the kernel does less work when idle and can potentially save power because it does not have to wake up regularly just to service the timer. Eliminating idle periodic ticks causes kernel process scheduler not do idle balance as frequently as it would do otherwise.

No periodic timer interrupts(“the timer tick”), allows CPU stay in chosen C state until some event happens.

The kernel seeks to avoid processor wakeups by turning off the period timer tick when nothing is happening. Before stopping the clock, the kernel must decide when it should wake up again; this decision involves looking at the timer queue to see when the next timer expires. In the absence of other events (hardware interrupts, for example), the system will sleep until the nearest timer is due. When a CPU goes into idle state, timer framework evaluates the next scheduled timer event and in case that the next event is further away than the next periodic tick, it reprograms the per-CPU clock-event device to this future event. This will allow the idle CPU to go into longer idle sleeps without the unnecessary interruption by the periodic tick.

### Deferrable timers

Many of kernel timers should run as soon as the requested period has expired. Others, however, are less important - to the point that they are not worth waking up the processor. These non-critical timeouts can run some fraction of a second later (when the processor wakes up for other reasons) and nobody will notice the difference.

Deferrable timers work as usual timers when system is busy, and they will fire at the scheduled time. In idle deferrable timers wait in queue until system wakes up due to a non-deferrable timer expires or any other interrupt wakes up the CPU.

# 8

## runtime PM

**Runtime power management (runtime pm)** manages the suspending and resuming of individual system components at run time, which is a framework through which device drivers can implement autonomous power management when idle. Run-time PM allows devices to be automatically idled or auto-suspended upon inactivity, independently of one another instead of having all devices suspended together using the standard static suspend techniques.

Device that are not used during run time could be powered off to save power.

- while the system is running devices may be put to low power states independently of other power management activity.
- a parent device cannot be suspended unless all of its child devices have been suspended
- device local suspend/resume, Fine granularity PM, not like whole system sleep
- user application is not required to be frozen as system sleep
- Transparent to user space

Many devices are able to dynamically power down while the system is still running. This feature is useful for devices that are not being used, and can offer significant power savings on a running system. These devices often support a range of runtime power states, which might use names such as "off", "sleep", "idle", "active", and so on. Those states will in some cases (like PCI) be partially constrained by the bus the device uses, and will usually include hardware states that are also used in system sleep states.

- Subsystems and drivers provide callbacks (struct dev\_pm\_ops)
- Subsystems and drivers handle remote wakeup.
- The core handles concurrency (locking etc.).
- The core takes care of device dependencies (parents vs children).
- The core provides reference counting facilities (detection of idleness).
- The core provides helpers (e.g. pm\_runtime\_suspend()).

e.g.

- USB: please check suspend/resume functions in drivers/usb/gadget/mv\_udc\_core.c
- LCD.
- SDH
- PCIE

Note: cpuidle can be seen as a special case of runtime PM for CPU

## 8.1 Runtime PM V.S. autosleep

System sleep and runtime PM are related to each other, but they are not the same thing. In some situations they may bring the system to the same physical state, but they do that in different ways. Runtime power management (Runtime PM) turns off (stop clock or remove power) hardware components that aren't going to be used in the near future, transparently from the user space's viewpoint. System sleep assumes or knows in advance that the whole system is not going to be used in the near future, turn off everything (possibly by force) except for the RAM chips and wakeup source generators.

In autosleep (*opportunistic suspend*) approach the natural state of the system is a sleep state, in which energy is only used for refreshing memory and providing power to a few devices that can generate wakeup signals. The working state, in which the CPUs are executing instructions and the system is generally doing some useful work, is only entered in response to a wakeup signal from one of the selected devices. The system stays in that state only as long as necessary to do certain work requested by the user. When the work has been completed, the system automatically goes back to the sleep state. This approach can be referred to as *opportunistic suspend* to emphasize the fact that it causes the system to suspend every time there is an opportunity to do so.

To implement it effectively one has to address a number of issues, including possible race conditions between system suspend and wakeup events (i.e. events that cause the system to wake up from sleep states). Namely, one of the first things done during system suspend is to freeze user space processes (except for the suspend process itself) and after that's been completed user space cannot react to any events signaled by the kernel. In consequence, if a wakeup event occurs exactly at the time the suspend process is started, user space may be frozen before it will have a chance to consume the event, which will be delivered to it only after the system is woken up from the sleep state as a result of *another* wakeup event. Unfortunately, on a cell phone the "deferred" wakeup event may be a very important incoming call, so the above scenario is hardly acceptable for this type of device.

For this reason, one may think that it's better not to suspend the system at all and use the *cpuidle* framework for the entire system power management. This approach appears to allow some systems to be put into a low-power state resembling a sleep state. However, it may not guarantee that the system will be put into that state sufficiently often because of applications using busy loops to excess and kernel timers. PM quality of service (QoS) requests may also prevent *cpuidle* from using deep low-power state of the CPUs. Moreover, while only a few selected devices are enabled to signal wakeup during system suspend, the runtime power management routines that may be used by *cpuidle* for suspending I/O devices tend to enable all of them to signal wakeup. Thus the system wakes up from low-power states entered as a result of *cpuidle* transitions relatively more often than from "real" sleep states, so its ability to save energy is limited. This basically means that *cpuidle*-based system power management may not be sufficient to save as much energy as opportunistic suspend on the same system.

If you predict that you will be idle for a long enough period of time then *cpuidle* is perfectly valid for you to hit your deepest sleep state in the *cpuidle* path. (it can be the lowest power state as system suspend). However, it's not that ideal case in reality because of the issues mentioned above.

Thus, it does seem more common for suspend to target a deeper hardware sleep state than the deepest possible *cpuidle* state for a given platform. Then why we need both idle and suspend mechanisms? Answer is that the runtime PM and *cpuidle* help to save power when autosleep is blocked by wakeup sources.

### Runtime PM

The `.runtime_suspend()` callback operates on a device that's already quiescent.

Device driver has control on when to suspend/resume

### System suspend

The `.suspend()` callback's role is to quiescent the device.

The `.suspend_noirq()` callback operates on a device quiescent by `.suspend()`.

Often `.runtime_suspend()` and `.suspend_noirq()` can point to the same routine, while `.suspend()` is specific to system suspend.

## 8.2 Hardware v.s. Software

Another critical thing to understand is the difference of hardware and software power-saving behaviors.

The low-power states that the silicon can achieve which varies in how much power they save with trade-offs such increased wake-up latency, loss of context/cache, etc. WFI is the gateway to low-power states in ARM hardware (from the perspective of the Linux kernel). A plain WFI without any extra steps will gate the CPUs clocks. With some extra steps (programming target power domain state, etc) then WFI can trigger lower voltages supplied by the PMIC/regulators, or total power gating for power domains/island resulting in increased energy savings but costlier wake-up time and loss of context.

The software behavior is what the Linux OS tries to do to save power. Runtime PM and cuilde and system suspend all are such behaviors.

Idle is "oh, the device/cpu is doing nothing, it can enter low power state to save power". Note that idle does not aim to affect the business of the Linux scheduler. E.g. it ideally should not impact performance, as it is only going to target a low power hardware idle state opportunistically based on naturally occurring idle time from the scheduler.

Suspend is "I don't want the whole system to do anything until I press power button, force sleep to save power". It *forces* idleness upon the OS until a wake-up event resumes the OS from suspend. Imagine closing the lid on your laptop while it is running. That is suspend. Processes are frozen regardless of whether we have lots of work scheduled or not. Typically system suspend targeting the deepest hardware idle state, but it doesn't have to.

There is nothing stopping a platform from suspending to RAM and leaving everything powered up and only clock gating the CPUs with a WFI. That is possible and indicates the separation of software and hardware idling. When that happens, we have a bug to be fixed to make them match.

# 9

## Thermal Management

For the definition and basic abstraction concepts (Documentation/thermal/sysfs-api.txt).

- Concepts of thermal zones, trip points and cooling devices.
- Framework to register thermal zone and cooling devices.

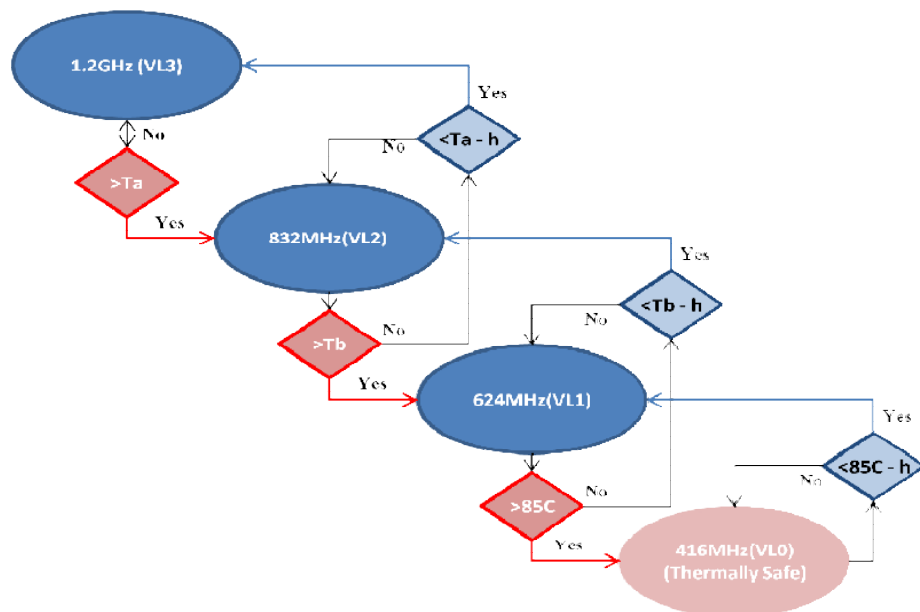
Cooling devices:

- cpufreq: opensource already support it; but it's use scaling\_max which will be over written by others easily, so it's changed to use cpufreq max QoS
- DDRfreq: use ddr\_devfreq QoS to add max constraint.

Bi\_direction cooling governor.

- Driver add both up and down thresholds (T1...t1...)
- governor will get trend by cur\_temperature and last\_temperature;
- governor get next cooling state with (T1, T2,... ) or (t1, t2,...) based on the trend
- governor call framework API, the related cooling instance will be updated

Below diagram takes the cpufreq cooling device as example to explain the bi-direction governor



- Different hardware design may have different thermal characteristics
- No user space governor, all policy decided by kernel space
- Power down when software failed to cool the system.

# 10

## WiFi Power Management

- WiFi uAP will be turned off by host after 10mins in idle mode (configurable via webui). So that wifi chip will go to sleep mode to save power.
  - no timeout when USB/charger exist
  - restart the timer at charger plug in/out events
  - recover the uAP at ONKEY\_EVENT or webUI operation

- Host sleep mechanism

It's the mechanism in wifi driver and wifi firmware to notify each other's idle state and wakeup each other. Host (PXA1826) will be in sleep mode if there is no wifi traffic. Any wakeup event or STA transaction can wake up the host system as long as the wifi chip is not in sleep state (not timeout)

## A Revision History

Table 2: Revision History

Document No and Revision	Int Rev	Description	Date
0.1		Initial version	July 18, 2015
0.2		Add 8.1 and 8.2 to clarify the relationship of idle and suspend, rephrase the runtime pm description.	July 22, 2015





Marvell Semiconductor, Inc.  
5488 Marvell Lane  
Santa Clara, CA 95054, USA  
Tel: 1.408.222.2500  
Fax: 1.408.988.8279  
[www.marvell.com](http://www.marvell.com)

**Marvell. Moving Forward Faster**