Runtime Power Management Framework

for I/O Devices in the Linux Kernel

Rafael J. Wysocki

Faculty of Physics UW / SUSE Labs / Renesas

June 10, 2011

Outline

Why Do We Need a Framework for Device Runtime PM?

Well, there are a few reasons

1. Platform support may be necessary to change the power states of devices.
2. Wakeup signaling is often platform-dependent or bus-dependent (e. g. PCI devices don't generate interrupts from low-power states).
3. Drivers may not know when to suspend devices.
   - Devices may depend on one another (accross subsystem boundaries).
   - No suitable "idle" condition at the driver level.
4. PM-related operations often need to be queued up for execution in future (e. g. a workqueue is needed).
5. Runtime PM has to be compatible with system-wide transitions to a sleep state (and back to the working state).

Device "States"

Runtime PM framework uses abstract states of devices

- **ACTIVE** – Device can do I/O (presumably in the full-power state).
- **SUSPENDED** – Device cannot do I/O (presumably in a low-power state).
- **SUSPENDING** – Device state is changing from ACTIVE to SUSPENDED.
- **RESUMING** – Device state is changing from SUSPENDED to ACTIVE.

Device “States”

Runtime PM framework uses abstract states of devices

ACTIVE – Device can do I/O (presumably in the full-power state).
SUSPENDED – Device cannot do I/O (presumably in a low-power state).
SUSPENDING – Device state is changing from ACTIVE to SUSPENDED.
RESUMING – Device state is changing from SUSPENDED to ACTIVE.

Runtime PM framework is oblivious to the actual states of devices

The real states of devices at any given time depend on the subsystems and drivers that handle them.

Changing the (Runtime PM) State of a Device

Suspend functions

```
int pm_runtime_suspend(struct device *dev);
int pm_schedule_suspend(struct device *dev, unsigned int delay);
```

Changing the (Runtime PM) State of a Device

Suspend functions

```
int pm_runtime_suspend(struct device *dev);
int pm_schedule_suspend(struct device *dev, unsigned int delay);
```

Resume functions

```
int pm_runtime_resume(struct device *dev);
int pm_request_resume(struct device *dev);
```

Changing the (Runtime PM) State of a Device

Suspend functions

```
int pm_runtime_suspend(struct device *dev);
int pm_schedule_suspend(struct device *dev, unsigned int delay);
```

Resume functions

```
int pm_runtime_resume(struct device *dev);
int pm_request_resume(struct device *dev);
```

Notifications of (apparent) idleness

```
int pm_runtime_idle(struct device *dev);
int pm_request_idle(struct device *dev);
```

Reference Counting

Devices with references held cannot be suspended.

Reference Counting

Devices with references held cannot be suspended.

Taking a reference

```
int pm_runtime_get(struct device *dev);  /* + resume request */
int pm_runtime_get_sync(struct device *dev);  /* + sync resume */
int pm_runtime_get_noresume(struct device *dev);
```

Reference Counting

Devices with references held cannot be suspended.

Taking a reference

```
int pm_runtime_get(struct device *dev);  /* + resume request */
int pm_runtime_get_sync(struct device *dev);  /* + sync resume */
int pm_runtime_get_noresume(struct device *dev);
```

Dropping a reference

```
int pm_runtime_put(struct device *dev);  /* + idle request */
int pm_runtime_put_sync(struct device *dev);  /* + sync idle */
int pm_runtime_put_noidle(struct device *dev);
```

Subsystem and Driver Callbacks

include/linux/pm.h

```
struct dev_pm_ops {
        ...
        int (*runtime_suspend)(struct device *dev);
        int (*runtime_resume)(struct device *dev);
        int (*runtime_idle)(struct device *dev);
};
```

Subsystem and Driver Callbacks

include/linux/pm.h

```
struct dev_pm_ops {
        ...
        int (*runtime_suspend)(struct device *dev);
        int (*runtime_resume)(struct device *dev);
        int (*runtime_idle)(struct device *dev);
};
```

include/linux/device.h

```
struct device_driver {
        ...
        const struct dev_pm_ops *pm;
        ...
};
```

```
struct struct bus_type {
        ...
        const struct dev_pm_ops *pm;
        ...
};
```

Wakeup Signaling Mechanisms

Depend on the platform and bus type

1. Special signals from low-power states (device signal causes another device to generate an interrupt).
   - PCI Power Management Event (PME) signals.
   - PNP wakeup signals.
   - USB "remote wakeup".
2. Interrupts from low-power states (wakeup interrupts).

Wakeup Signaling Mechanisms

Depend on the platform and bus type

1. Special signals from low-power states (device signal causes another device to generate an interrupt).
    - PCI Power Management Event (PME) signals.
    - PNP wakeup signals.
    - USB "remote wakeup".
2. Interrupts from low-power states (wakeup interrupts).

What is needed?

1. Subsystem and/or driver callbacks need to set up devices to generate these signals.
2. The resulting interrupts need to be handled (devices should be put into the ACTIVE state as a result).

sysfs Interface

`/sys/devices/.../power/control`

- on – Device is always ACTIVE (default).
- auto – Device state can change.

sysfs Interface

/sys/devices/.../power/control

- on – Device is always ACTIVE (default).
- auto – Device state can change.

/sys/devices/.../power/runtime_status (read-only, 2.6.36 material)

- active – Device is ACTIVE.
- suspended – Device is SUSPENDED.
- suspending – Device state is changing from ACTIVE to SUSPENDED.
- resuming – Device state is changing from SUSPENDED to ACTIVE.
- error – Runtime PM failure (runtime PM of the device is disabled).
- unsupported – Runtime PM of the device has not been enabled.

powertop Support (Since 2.6.36)

Two additional per-device *sysfs* files.

powertop Support (Since 2.6.36)

Two additional per-device *sysfs* files.

`/sys/devices/.../power/runtime_active_time`

Time spent in the ACTIVE state.

powertop Support (Since 2.6.36)

Two additional per-device *sysfs* files.

`/sys/devices/.../power/runtime_active_time`

Time spent in the ACTIVE state.

`/sys/devices/.../power/runtime_suspended_time`

Time spent in the SUSPENDED state.

powertop Support (Since 2.6.36)

Two additional per-device *sysfs* files.

`/sys/devices/.../power/runtime_active_time`

Time spent in the ACTIVE state.

`/sys/devices/.../power/runtime_suspended_time`

Time spent in the SUSPENDED state.

powertop will use them to report per-device "power" statistics.

The Execution of Callbacks

The PM core executes subsystem callbacks

The subsystem may be either a device type, or a device class, or a device type (in this order).

The Execution of Callbacks

The PM core executes subsystem callbacks

The subsystem may be either a device type, or a device class, or a device type (in this order).

Subsystem callbacks (are supposed to) execute driver callbacks

1. The subsystem callbacks are responsible for handling the device.
2. They may or may not execute the driver callbacks.
3. What the driver callbacks are expected to do depends on the subsystem.

Automatic Idle Notifications, System Suspend

The PM core triggers automatic idle notifications

1. After a device has been (successfully) put into the ACTIVE state.
2. After all children of a device have been suspended.

Automatic Idle Notifications, System Suspend

The PM core triggers automatic idle notifications

1. After a device has been (successfully) put into the ACTIVE state.
2. After all children of a device have been suspended.

This causes an idle notification request to be queued up for the device.

Automatic Idle Notifications, System Suspend

The PM core triggers automatic idle notifications

1. After a device has been (successfully) put into the ACTIVE state.
2. After all children of a device have been suspended.

This causes an idle notification request to be queued up for the device.

The PM workqueue is freezable

Only synchronous operations (runtime suspend, runtime resume) work during system-wide suspend/hibernation.

I/O Runtime PM Reference Counting Problem

In general, there is no guarantee that all device runtime PM usage counters will be 0 before (or even during) system suspend.

I/O Runtime PM Reference Counting Problem

In general, there is no guarantee that all device runtime PM usage counters will be 0 before (or even during) system suspend.

For this reason, the I/O runtime PM framework cannot be used directly for suspending devices during system suspend.

I/O Runtime PM Reference Counting Problem

In general, there is no guarantee that all device runtime PM usage counters will be 0 before (or even during) system suspend.

For this reason, the I/O runtime PM framework cannot be used directly for suspending devices during system suspend.

Nevertheless, it generally is possible to use the same PM callback routines for both runtime PM and system suspend/resume at the driver level (not necessarily at the subsystem level).

I/O Runtime PM Reference Counting Problem

In general, there is no guarantee that all device runtime PM usage counters will be 0 before (or even during) system suspend.

For this reason, the I/O runtime PM framework cannot be used directly for suspending devices during system suspend.

Nevertheless, it generally is possible to use the same PM callback routines for both runtime PM and system suspend/resume at the driver level (not necessarily at the subsystem level).

That may or may not be a good idea depending on the platform the driver is designed for.

Remote Wakeup Problem

Runtime PM requires that remote wakeup be set up, if supported, for all devices being suspended (needed for transparency from the user space's perspective).

Remote Wakeup Problem

Runtime PM requires that remote wakeup be set up, if supported, for all devices being suspended (needed for transparency from the user space's perspective).

In the system sleep case that depends on information provided by user space via *sysfs*.

Remote Wakeup Problem

Runtime PM requires that remote wakeup be set up, if supported, for all devices being suspended (needed for transparency from the user space's perspective).

In the system sleep case that depends on information provided by user space via *sysfs*.

Therefore subsystem-level PM callbacks need to work differently during system suspend/resume and during the analogous runtime PM operations.

Remote Wakeup Problem

Runtime PM requires that remote wakeup be set up, if supported, for all devices being suspended (needed for transparency from the user space's perspective).

In the system sleep case that depends on information provided by user space via *sysfs*.

Therefore subsystem-level PM callbacks need to work differently during system suspend/resume and during the analogous runtime PM operations.

This applies to power domain PM callbacks too.

What A Power Management Domain Is

Technically

Power domain is a set of devices sharing power resources (e.g. clocks, power planes).

What A Power Management Domain Is

Technically

Power domain is a set of devices sharing power resources (e.g. clocks, power planes).

From the kernel's perspective

Power management domain is a set of devices whose power management uses the same set of callbacks with common PM data at the subsystem level (not necessarily in one power domain, but mutually dependent).

What A Power Management Domain Is

Technically

Power domain is a set of devices sharing power resources (e.g. clocks, power planes).

From the kernel's perspective

Power management domain is a set of devices whose power management uses the same set of callbacks with common PM data at the subsystem level (not necessarily in one power domain, but mutually dependent).

Representation via `struct dev_power_domain` and derived structures (need to change the name!).

What A Power Management Domain Is

Technically

Power domain is a set of devices sharing power resources (e.g. clocks, power planes).

From the kernel's perspective

Power management domain is a set of devices whose power management uses the same set of callbacks with common PM data at the subsystem level (not necessarily in one power domain, but mutually dependent).

Representation via `struct dev_power_domain` and derived structures (need to change the name!).

If a PM domain object exists for a device, its PM callbacks take precedence over bus type (or device class, or type) callbacks (3.0-rc1).

Power Domains and PM Domains

PM domains are a more general concept (there need not be a power domain for a PM domain object to be useful).

Power Domains and PM Domains

PM domains are a more general concept (there need not be a power domain for a PM domain object to be useful).

Nevertheless, the main intended purpose of PM domains is to support power domains.

Power Domains and PM Domains

PM domains are a more general concept (there need not be a power domain for a PM domain object to be useful).

Nevertheless, the main intended purpose of PM domains is to support power domains.

The current proposal is to add PM domains support for the simple case in which a device can belong to one power domain at a time and there is a clearly defined way to power off and power down a power domain.

Runtime PM Of Power Domains

Observations

1. All devices in a power domain have to be idle so that a shared power resource can be turned off (e.g. clock stopped or power removed).
2. Power is necessary for remote wakeup to work.
3. Latency to turn a power domain on generally depends on all devices in it.

Runtime PM Of Power Domains

Observations

1. All devices in a power domain have to be idle so that a shared power resource can be turned off (e.g. clock stopped or power removed).
2. Power is necessary for remote wakeup to work.
3. Latency to turn a power domain on generally depends on all devices in it.

Thus the PM core should provide means by which:

1. The status of devices in a power domain may be monitored.
2. Decisions to turn power domains off may be made on the basis of (known) device latencies and predicted next usage time (and PM QoS).