Runtime Power Management Framework

for I/O Devices in the Linux Kernel

Rafael J. Wysocki

Faculty of Physics UW / SUSE Labs, Novell Inc.

July 31, 2010

Outline

Goals of Power Management (in a Computer System)

Reduce the amount of energy used

To save the planet, lower power bills, extend battery life, ...

Limit the total power draw

To prevent the maximum capacity from being exceeded.

Provide information and control interfaces

To allow the user(s) to control the system's behavior.

Consumption of Energy

Suppose it takes time T to do certain amount of work and the energy used for doing the work is E.

Consumption of Energy

Suppose it takes time T to do certain amount of work and the energy used for doing the work is E.

Then, we have

$$E = \int_0^T P(t)dt$$

where $P(t)$ is the total power draw at the time t (non-negative, depends on the number of hardware components drawing power at the time t).

Consumption of Energy

Suppose it takes time T to do certain amount of work and the energy used for doing the work is E.

Then, we have

$$E = \int_0^T P(t)dt$$

where $P(t)$ is the total power draw at the time t (non-negative, depends on the number of hardware components drawing power at the time t).

T is a measure of performance

The smaller it is, the faster the work is done.

Reducing Consumption of Energy

Observation

Powering up more hardware components than necessary to do the work does not cause the work to be done faster (i. e. increasing $P(t)$ above certain minimum level does not cause T to shrink).

Reducing Consumption of Energy

Observation

Powering up more hardware components than necessary to do the work does not cause the work to be done faster (i. e. increasing $P(t)$ above certain minimum level does not cause T to shrink).

Strategies

- Turn off hardware components that are not necessary.
- Turn hardware components on as needed (caution required).
- Reduce the capacity of hardware components that are not 100% utilized.

Runtime power management (runtime PM)

Helps to reduce the consumption of energy while work is being done.

Saving Energy vs Performance

Observation

Sacrificing performance makes sense if the reduction of $P(t)$ offsets the growth of T.

Saving Energy vs Performance

Observation

Sacrificing performance makes sense if the reduction of $P(t)$ offsets the growth of T.

Example – CMOS integrated circuits

1. Dissipated power is proportional to the frequency (f) and the square of the voltage (V)
$$E \sim V^2 \cdot f \cdot T$$
2. The time necessary to do the job (T) is inversely proportional to the frequency (f)
$$E \sim V^2$$
3. It makes sense to reduce f if that allows one to reduce V.

What If the System Is Idle?

What does "idle" mean?

The system doesn't do any useful work (the user knows what is useful).

What If the System Is Idle?

What does "idle" mean?

The system doesn't do any useful work (the user knows what is useful).

Idle systems consume energy in vain

1. Put them to sleep.
   - Sleep states (different wakeup times, different power consumption).
   - Substantial energy savings possible if used aggressively.
   - The system is prevented from doing work.
   - Clock sources are shut down (RTC is running).
   - Putting the system to sleep and waking it up takes energy.
2. Turn all hardware components off "dynamically".
   - This can be regarded as a special case of runtime PM for $P_{min}(t) = 0$.

Hardware Capabilities

Processors

- P-states (dynamic reduction of frequency and voltage).
- C-states (CPU "sleep" states).

I/O devices

- Standard low-power states (e. g. ACPI, PCI, USB, ...).
- Direct control of clock and voltage.
- Wakeup signaling (remote wakeup).
- Device-specific capabilities.

Kernel Features Related to Runtime PM

cpufreq

Dynamic CPU frequency and voltage scaling (P-states).

cpuidle + PM QoS

Using CPU C-states to save energy when the CPU is idle (PM QoS is used to specify latency requirements).

Clock framework

I/O devices' clocks manipulation.

I/O runtime PM framework (and USB autosuspend framework)

Using device low-power states to save energy when the devices are "idle".

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).

cpuidle and Runtime PM

The problem of platform-specific driver modifications

- A driver entry point is necessary to suspend a device from a *cpuidle* `->enter()` callback (analogously for resume).
- Without a general framework these driver entry points need to be platform-specific.
- "Special" drivers for SoC (System on a Chip) devices?

SuperH (shmobile)

On-chip devices are suspended from *cpuidle* "context" using the "standard" runtime PM callbacks.

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);
```

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.

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);
};
```

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.

`/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 (2.6.36 material)

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 bus type, or a device type, or a device class (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.

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.
- The PM core takes references on all devices during the "prepare" phase of a system transition (`pm_runtime_get_noresume()` is used).

More Driver & Subsystem Support, Interface to *cpuidle*

There are only a few PCI drivers supporting runtime PM

Only a few subsystems support runtime PM (2.6.35-rc)

- `platform` bus type (weak definitions of callbacks).
- PCI
- USB
- I^2C

Difficulty with "idle" conditions at the driver level

- User space may know when to suspend devices (e. g. video).
- *cpuidle* should be able to schedule the suspend of I/O devices.

Documentation & Headers

Documentation

- `Documentation/power/devices.txt`
- `Documentation/power/runtime_pm.txt`
- `Documentation/power/pci.txt`
- *PCI Local Bus Specification, Rev. 3.0*
- *PCI Bus Power Management Interface Specification, Rev. 1.2*
- *Advanced Configuration and Power Interface Specification, Rev. 3.0b*
- *PCI Express Base Specification, Rev. 2.0*

Headers

- `include/linux/pm.h`
- `include/linux/pm_runtime.h`

Source Code

- drivers/base/power/runtime.c
- drivers/base/power/sysfs.c
- drivers/base/power/generic_ops.c
- drivers/base/power/power.h
- drivers/pci/pci-driver.c
- drivers/pci/pci.c
- drivers/pci/pci-acpi.c
- drivers/pci/pci.h
- drivers/pci/pcie/pme/*
- drivers/acpi/pci_root.c
- drivers/acpi/pci_bind.c

PCI Device Power States

Power states of PCI devices

$D0$ – Full-power state.

$D1$ – Low-power state, optional, minimum exit latency.

$D2$ – Low-power state, optional, 200 μs recovery time.

$D3_{hot}$ – Low-power state, 10 ms recovery time, "sw accessible".

$D3_{cold}$ – Low-power state, no Vcc, 10 ms recovery time.

Devices in low-power states cannot do "regular" I/O

- Memory and I/O spaces are disabled.
- Devices are only allowed to initiate PME.
- Devices in $D3_{cold}$ are not accessible to software (RST# has to be asserted after power has been restored).

Power Management Events (PME)

Out-of-band for "parallel" PCI (PME network)

- PME signals go directly to system core logic.
- May be routed around bridges.
- Platform (e. g. ACPI BIOS) support required.
- Add-on devices problem.
  - Device identification (bridge as a proxy, bus walking required).
  - Hardware vendors (sometimes) "forget" to attach PME lines.

In-band for PCI Express (PME messages)

- Native signaling via interrupts generated by Root Ports.
- Coupled with native PCIe hot-plug (shared interrupt vector).
- ACPI BIOSes may not allow us to use it directly.

PCI Device Probing (2.6.36 material)

drivers/pci/pci-driver.c

```
static long local_pci_probe(void *_ddi)
{
        struct drv_dev_and_id *ddi = _ddi;
        struct device *dev = &ddi->dev->dev;
        int rc;

        /* Unbound PCI devices are always set to disabled and suspended.
         * During probe, the device is set to enabled and active and the
         * usage count is incremented.  If the driver supports runtime PM,
         * it should call pm_runtime_put_noidle() in its probe routine and
         * pm_runtime_get_noresume() in its remove routine.
         */
        pm_runtime_get_noresume(dev);
        pm_runtime_set_active(dev);
        pm_runtime_enable(dev);

        rc = ddi->drv->probe(ddi->dev, ddi->id);
        if (rc) {
                pm_runtime_disable(dev);
                pm_runtime_set_suspended(dev);
                pm_runtime_put_noidle(dev);
        }
        return rc;
}
```

PCI Device Removal (2.6.36 material)

drivers/pci/pci-driver.c

```
static int pci_device_remove(struct device * dev)
{
        struct pci_dev * pci_dev = to_pci_dev(dev);
        struct pci_driver * drv = pci_dev->driver;

        if (drv) {
                if (drv->remove) {
                        pm_runtime_get_sync(dev);
                        drv->remove(pci_dev);
                        pm_runtime_put_noidle(dev);
                }
                pci_dev->driver = NULL;
        }

        /* Undo the runtime PM settings in local_pci_probe() */
        pm_runtime_disable(dev);
        pm_runtime_set_suspended(dev);
        pm_runtime_put_noidle(dev);

        ...
        if (pci_dev->current_state == PCI_D0)
                pci_dev->current_state = PCI_UNKNOWN;
        ...
}
```

PCI Bus Type's Runtime Suspend Routine

drivers/pci/pci-driver.c

```
static int pci_pm_runtime_suspend(struct device *dev)
{
        struct pci_dev *pci_dev = to_pci_dev(dev);
        const struct dev_pm_ops *pm = dev->driver ? dev->driver->pm : NULL;
        pci_power_t prev = pci_dev->current_state;
        int error;

        if (!pm || !pm->runtime_suspend)
                return -ENOSYS;

        error = pm->runtime_suspend(dev);
        suspend_report_result(pm->runtime_suspend, error);
        if (error)
                return error;

        pci_fixup_device(pci_fixup_suspend, pci_dev);

        ...
        if (!pci_dev->state_saved)
                pci_save_state(pci_dev);

        pci_finish_runtime_suspend(pci_dev);

        return 0;
}
```

PCI "Finish Runtime Suspend" Routine

drivers/pci/pci.c

```
int pci_finish_runtime_suspend(struct pci_dev *dev)
{
        pci_power_t target_state = pci_target_state(dev);
        int error;

        if (target_state == PCI_POWER_ERROR)
                return -EIO;

        __pci_enable_wake(dev, target_state, true, pci_dev_run_wake(dev));

        error = pci_set_power_state(dev, target_state);

        if (error)
                __pci_enable_wake(dev, target_state, true, false);

        return error;
}
```

`pci_target_state(dev)` chooses the low-power state to put the device into (i.e. the lowest-power state the device can signal wakeup from).

PCI Wakeup Signaling Preparation

drivers/pci/pci.c

```
int __pci_enable_wake(struct pci_dev *dev, pci_power_t state, bool runtime, bool enable)
{
        int ret = 0;

        ...
        if (enable) {
                int error;

                if (pci_pme_capable(dev, state))
                        pci_pme_active(dev, true);
                else
                        ret = 1;
                error = runtime ? platform_pci_run_wake(dev, true) :
                                  platform_pci_sleep_wake(dev, true);
                if (ret)
                        ret = error;
                if (!ret)
                        dev->wakeup_prepared = true;
        } else {
                ...
        }

        return ret;
}
```

PCI Bus Type's Runtime Resume Routine

Called automatically after a wakeup interrupt has been generated by the platform (e. g. ACPI GPE) or by a PCIe Root Port (native PME).

drivers/pci/pci-driver.c

```
static int pci_pm_runtime_resume(struct device *dev)
{
        struct pci_dev *pci_dev = to_pci_dev(dev);
        const struct dev_pm_ops *pm = dev->driver ? dev->driver->pm : NULL;

        if (!pm || !pm->runtime_resume)
                return -ENOSYS;

        pci_pm_default_resume_early(pci_dev);
        __pci_enable_wake(pci_dev, PCI_D0, true, false);
        pci_fixup_device(pci_fixup_resume, pci_dev);

        return pm->runtime_resume(dev);
}
```

PCI Bus Type's Runtime "Idle" Routine

drivers/pci/pci-driver.c

```
static int pci_pm_runtime_idle(struct device *dev)
{
        const struct dev_pm_ops *pm = dev->driver ? dev->driver->pm : NULL;

        if (!pm)
                return -ENOSYS;

        if (pm->runtime_idle) {
                int ret = pm->runtime_idle(dev);
                if (ret)
                        return ret;
        }

       pm_runtime_suspend(dev);

        return 0;
}
```

Returning an error code from `->runtime_idle()` prevents suspend from happening (alternatively, reference counting can be used).

r8169 PCI Rutime PM: High-Level Description

General idea

Put the device into a low-power state if network cable is not connected.

Steps

1. Enable runtime PM during initialization.
2. Check if the runtime PM is allowed (by user space).
3. Check if the cable is attached initially (schedule suspend if not).
4. React to the changes of the cable status.
   - Start runtime resume if attached.
   - Schedule runtime suspend if detached.
5. Disable runtime PM when the driver is unloaded.

Driver Initialization and Removal (2.6.36 material)

drivers/net/r8169.c

```
static int __devinit rtl8169_init_one(struct pci_dev *pdev, const struct pci_device_id *ent)
{
        ...
        device_set_wakeup_enable(&pdev->dev, tp->features & RTL_FEATURE_WOL);

        if (pci_dev_run_wake(pdev))
                pm_runtime_put_noidle(&pdev->dev);

        ...
}



static void __devexit rtl8169_remove_one(struct pci_dev *pdev)
{
        ...

        if (pci_dev_run_wake(pdev))
                pm_runtime_get_noresume(&pdev->dev);

        ...
}
```

Opening the Device

drivers/net/r8169.c

```
static int rtl8169_open(struct net_device *dev)
{
        struct rtl8169_private *tp = netdev_priv(dev);
        struct pci_dev *pdev = tp->pci_dev;
        int retval = -ENOMEM;

        pm_runtime_get_sync(&pdev->dev);

        ...

        tp->saved_wolopts = 0;
        pm_runtime_put_noidle(&pdev->dev);

        rtl8169_check_link_status(dev, tp, tp->mmio_addr);
out:
        return retval;

        ...
}
```

Closing the Device

drivers/net/r8169.c

```
static int rtl8169_close(struct net_device *dev)
{
        struct rtl8169_private *tp = netdev_priv(dev);
        struct pci_dev *pdev = tp->pci_dev;

        pm_runtime_get_sync(&pdev->dev);

        /* update counters before going down */
        rtl8169_update_counters(dev);

        ...

        pm_runtime_put_sync(&pdev->dev);

        return 0;
}
```

Checking Link Status

drivers/net/r8169.c

```
static void rtl8169_check_link_status(struct net_device *dev,
                                      struct rtl8169_private *tp,
                                      void __iomem *ioaddr)
{
        unsigned long flags;

        spin_lock_irqsave(&tp->lock, flags);
        if (tp->link_ok(ioaddr)) {
                /* This is to cancel a scheduled suspend if there's one. */
                pm_request_resume(&tp->pci_dev->dev);
                netif_carrier_on(dev);
                netif_info(tp, ifup, dev, "link up\n");
        } else {
                netif_carrier_off(dev);
                netif_info(tp, ifdown, dev, "link down\n");
                pm_schedule_suspend(&tp->pci_dev->dev, 100);
        }
        spin_unlock_irqrestore(&tp->lock, flags);
}
```

Suspending the Device

drivers/net/r8169.c

```
static int rtl8169_runtime_suspend(struct device *device)
{
        struct pci_dev *pdev = to_pci_dev(device);
        struct net_device *dev = pci_get_drvdata(pdev);
        struct rtl8169_private *tp = netdev_priv(dev);

        if (!tp->TxDescArray)
                return 0;

        spin_lock_irq(&tp->lock);
        tp->saved_wolopts = __rtl8169_get_wol(tp);
        __rtl8169_set_wol(tp, WAKE_ANY);
        spin_unlock_irq(&tp->lock);

        rtl8169_net_suspend(dev);

        return 0;
}
```

Resuming the Device

drivers/net/r8169.c

```
static int rtl8169_runtime_resume(struct device *device)
{
        struct pci_dev *pdev = to_pci_dev(device);
        struct net_device *dev = pci_get_drvdata(pdev);
        struct rtl8169_private *tp = netdev_priv(dev);

        if (!tp->TxDescArray)
                return 0;

        spin_lock_irq(&tp->lock);
        __rtl8169_set_wol(tp, tp->saved_wolopts);
        tp->saved_wolopts = 0;
        spin_unlock_irq(&tp->lock);

        __rtl8169_resume(dev);

        return 0;
}
```

Notification of Idleness, PM Operations

drivers/net/r8169.c

```
static int rtl8169_runtime_idle(struct device *device)
{
        struct pci_dev *pdev = to_pci_dev(device);
        struct net_device *dev = pci_get_drvdata(pdev);
        struct rtl8169_private *tp = netdev_priv(dev);

        if (!tp->TxDescArray)
                return 0;

        rtl8169_check_link_status(dev, tp, tp->mmio_addr);
        return -EBUSY;
}

static const struct dev_pm_ops rtl8169_pm_ops = {
        ...
        .runtime_suspend = rtl8169_runtime_suspend,
        .runtime_resume = rtl8169_runtime_resume,
        .runtime_idle = rtl8169_runtime_idle,
};
```