Runtime Power Management in the PCI Subsystem

of the Linux Kernel

Rafael J. Wysocki

Faculty of Physics U. Warsaw / SUSE Labs, Novell Inc.

November 4, 2010

Outline

The Goal: Allow PCI Drivers to Use I/O Runtime PM

I/O Runtime PM framework is there in the kernel

PCI device drivers should be able to use it.

I/O Runtime PM framework's requirements

1. Subsystem (i.e. PCI bus type) callbacks have to be implemented.
   - `->runtime_suspend(dev)`
   - `->runtime_resume(dev)`
   - `->runtime_idle(dev)`
2. Wakeup signaling (*remote wakeup*) has to work.
   - Preparing devices to signal wakeup.
   - Receiving and processing wakeup signals from devices.
   - Resuming devices that signaled wakeup.

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".

$D3_{cold}$ – Power off, 10 *ms* (or more) recovery time.

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

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

PCI Bus (Segment) Power States

Power states of PCI bus segments

- $B0$ – Full-power state.
- $B1$ – Vcc applied, no transactions, perpetual idle.
- $B2$ – Vcc applied, clock stopped (low), 50 ms recovery time.
- $B3$ – no Vcc, no clock, RST# required for recovery.

No transactions allowed on bus segments in low-power states

- Devices are only allowed to signal wakeup.
- $B3$ implies $D3_{cold}$ for all devices on the bus segment.
- Bus segment goes to Bn when its bridge is put into Dn.
- Programming the bridge into $D3_{hot}$ need not put the bus into $B3$.
- Power removal puts bus segments into $B3$.

Wakeup Signaling – 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.
  1. Device identification (bridge as a proxy, bus walking required).
  2. 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 PCI Express hot-plug (shared interrupt vector).
- ACPI BIOSes may not allow us to use it directly.

ACPI Device Power States and Methods

Power states of devices defined by ACPI

- *D*0 – Full-power.
- *D*1 – Low-power, optional, exit latency lesser than for *D*2.
- *D*2 – Low-power, optional, exit latency greater than for *D*1.
- *D*3 – Power off.

Device power management methods defined by ACPI (examples)

Power State n (`_PSn`) puts the device into power state *Dn*.

Power State Current (`_PSC`) evaluates to the current device state.

Power Resources n (`_PRn`) returns device power resources for *Dn*.

Device Sleep Wake (`_DSW`) enables or disables the device's wake function.

Power Resources for Wake (`_PRW`) returns wakeup power resources.

ACPI Power Resources and Wakeup Signaling

ACPI methods for manipulating power resources

On (`_ON`) turns the power resource on.

Off (`_OFF`) turns the power resource off.

Status (`_STA`) returns the current On/Off status.

ACPI-based wakeup signaling

Each wakeup-capable device has a dedicated General Purpose Event (GPE) for wakeup signaling, `_PRW` returns its number.

1. Device wakeup signal (e.g. PME) goes to core logic.
2. Core logic activates the device's "wakeup" GPE.
3. GPE causes System Control Interrupt (SCI) to be generated.
4. SCI interrupt handler executes GPE control method (`_Exx`/`_Lxx`).
5. GPE control method schedules system notification for the device.

Linux Kernel and ACPICA

ACPI Component Architecture (ACPICA)

"An operating system (OS)-independent reference implementation of the Advanced Configuration and Power Interface Specification (ACPI). It can be easily adapted to any host OS." (`http://www.acpica.org`)

Linux kernel uses ACPICA

1. ACPI Machine Language (AML) interpreter.
2. Parsing of ACPI configuration tables.
3. Walking ACPI namespace and execution of methods.
4. ACPI Global Lock manipulation.
5. SCI and event handling (GPEs and notifications).

GPE Handling by "Old" ACPICA (Prior to Linux 2.6.34)

No GPE reference counting

Attempts to enable (disable) an already enabled (disabled) GPE had no effect.

"Wakeup" vs "runtime" GPEs

1. If a GPE was pointed to by `_PRW` for at least one device, it was regarded as dedicated for wakeup from system sleep states.
2. "Wakeup" GPEs were not expected to be necessary in the system working state (ACPI *S0*).
3. GPEs that were not pointed to by `_PRW` for any devices were regarded as "runtime" GPEs and were disabled before entering a system sleep state (ACPI *S1*–*S4*).
4. "Run-wake" GPEs were special "runtime" GPEs that might be used for system wakeup.

GPE Handling Changes in Linux 2.6.34 and Later

PCI runtime PM requirements

1. Multiple `_PRW` methods may point to one GPE.
2. GPE should not be disabled as long as at least one device using it is suspended – reference counting is necessary.
3. GPEs pointed to by `_PRW` need to be enabled in the working state.

Resultant changes in ACPICA

1. Reference counting for "runtime" GPEs.
2. GPE "types" not used any more.
3. Setting up GPEs for system wakeup simplified.
4. ACPICA does not execute `_PRW` for all devices any more.
5. Implicit device notification for GPEs without associated `_Lxx`/`_Exx` methods pointed to by `_PRW` (in the works).

Native PCI Express Features and ACPI

Native PCI Express features

Hot-plug : Hotplug events signaled via Port interrupts.

AER (Advanced Error Reporting): Reporting hardware errors.

PME : Wakeup signaling via Root Port interrupts.

Cannot be enabled without requesting control from the ACPI BIOS.

ACPI interface for enabling PCI Express services

1. The kernel is supposed to use _OSC to ask the BIOS for control of the native PCI Express features.
2. The _OSC interface is not suitable to ask for each feature separately.
   - Control of PCI Express Capability Structure is needed for all of them.
   - They may depend on each other.

Changes Related to The Native PCI Express PME

Setting up PCI Express Root Ports

_OSC is used to request control of all the native PCI Express features simultaneously (Linux 2.6.36). This allows the native PCI Express PME driver to be enabled by default.

Multiple system notification handlers per device (ACPICA)

1. On PCI Express hardware PME and hot-plug are often signaled by the ACPI BIOS in the same way (via GPE and system notification).
2. There may be multiple reasons for ACPI system notification per device (PME and hot-plug).
3. It is (now) possible to install multiple system notification handlers per device (Linux 2.6.34).

PCI Native and ACPI-Based Power Management

Transition into a low-power state

Carried out by `pci_finish_runtime_suspend()` called from `pci_pm_runtime_suspend()`.

1. Choose state to put the device into (use ACPI to get information).
2. Program the device to signal wakeup (PME).
3. Set up the device's power resources for signaling wakeup, execute `_DSW` for it and enable its "wakeup" GPE.
4. Program the device to go into a (PCI) low-power state (e.g. $D3_{hot}$).
5. Execute `_PSn` for the device (the ACPI target state should match the PCI target state).
6. Turn off the device's power resources unnecessary for wakeup.

PCI Native and ACPI-Based Power Management Cont.

Transition into the full-power state

Carried out by `pci_pm_runtime_resume()`.

1. Turn on the device's power resources.
2. Execute `_PS0` for the device (to put it into ACPI *D0*).
3. Program the device to go into PCI *D0* (if it's not already there).
4. Disable the device's "wakeup" GPE, execute `_DSW` for it and turn off power resources needed only to signal wakeup.
5. Disable native wakeup signaling (PME) for the device.

There are special cases

1. Legacy PCI devices (native PCI PM not supported).
2. Add-on devices (ACPI PM not available).

Add-On Devices and PME

PCI Express

1. In-band PME messages received by Root Ports.
2. Root Ports generate interrupts (native mechanism) or trigger GPEs (ACPI mechanism).

"Parallel" PCI

1. Out-of-band signals that require separate routing (PME network).
2. Bridges are often used as proxies.
   - PME from a device triggers a GPE associated with an upstream brigde.
   - All devices under the bridge have to be inspected.
3. Hardware vendors sometimes fail to connect the PME lines.
   - Device PME status has to be checked periodically (polling).
   - Code doing that is in the works.

PCI Runtime PM Functions

drivers/pci/pci-driver.c

Subsystem-level (PCI bus type's) callbacks:

- `pci_pm_runtime_suspend()`
- `pci_pm_runtime_resume()`
- `pci_pm_runtime_idle()`

drivers/pci/pci.c

- `__pci_enable_wake()`: Enable/disable PCI wakeup capability.
- `pci_set_power_state()`: Change power state of PCI devices.
- `pci_finish_runtime_suspend()`: Transition to low-power states.
- `pci_dev_run_wake()`: Check if device is wakeup-capable.
- `pci_pme_wakeup_bus()`: PME status inspection and resume.

Functions Related to ACPI Support

drivers/pci/pci-acpi.c

- `acpi_pci_run_wake()`: Enable/disable ACPI-based wakeup.
- `acpi_pci_set_power_state()`: Set ACPI power state.
- `pci_acpi_wake_dev()`: Device wakeup.
- `pci_acpi_wake_bus()`: Bus wakeup (all devices under a bridge).

Conclusion

- PCI Runtime PM framework has been implemented.
- Native PCI PM and ACPI PM are used.
- Wakeup signaling through ACPI GPEs or native PCI Express PME.
- Substantial modifications of ACPICA were necessary (still being worked on).
- ACPI `_OSC` handling reworked.
- Some code still in the works.

References – Documentation

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*
- *ACPICA Project* (`http://www.acpica.org`)
- *I/O Runtime PM Framework Presentation* (`http://events.linuxfoundation.org/slides/2010/linuxcon2010_wysocki.pdf`)

References – Source Code

- include/linux/pm.h
- include/linux/pm_runtime.h
- include/linux/pci.h
- drivers/base/power/runtime.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/*
- drivers/acpi/pci_root.c
- drivers/acpi/pci_bind.c

PCI Device Probing (Linux 2.6.36)

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 (Linux 2.6.36)

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 (Linux 2.6.36)

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,
};
```