PCI Power Management

Copyright (c) 2010 Rafael J. Wysocki <rjw@sisk.pl>, Novell Inc.

An overview of concepts and the Linux kernel's interfaces related to PCI power management. Based on previous work by Patrick Mochel <mochel@transmeta.com> (and others).

This document only covers the aspects of power management specific to PCI devices. For general description of the kernel's interfaces related to device power management refer to Documentation/driver-api/pm/devices.rst and Documentation/power/runtime_pm.rst.

1. Hardware and Platform Support for PCI Power Management

1.1. Native and Platform-Based Power Management

In general, power management is a feature allowing one to save energy by putting devices into states in which they draw less power (low-power states) at the price of reduced functionality or performance.

Usually, a device is put into a low-power state when it is underutilized or completely inactive. However, when it is necessary to use the device once again, it has to be put back into the "fully functional" state (full-power state). This may happen when there are some data for the device to handle or as a result of an external event requiring the device to be active, which may be signaled by the device itself.

PCI devices may be put into low-power states in two ways, by using the device capabilities introduced by the PCI Bus Power Management Interface Specification, or with the help of platform firmware, such as an ACPI BIOS. In the first approach, that is referred to as the native PCI power management (native PCI PM) in what follows, the device power state is changed as a result of writing a specific value into one of its standard configuration registers. The second approach requires the platform firmware to provide special methods that may be used by the kernel to change the device's power state.

Devices supporting the native PCI PM usually can generate wakeup signals called Power Management Events (PMEs) to let the kernel know about external events requiring the device to be active. After receiving a PME the kernel is supposed to put the device that sent it into the full-power state. However, the PCI Bus Power Management Interface Specification doesn't define any standard method of delivering the PME from the device to the CPU and the operating system kernel. It is assumed that the platform firmware will perform this task and therefore, even though a PCI device is set up to generate PMEs, it also may be necessary to prepare the platform firmware for notifying the CPU of the PMEs coming from the device (e.g. by generating interrupts).

In turn, if the methods provided by the platform firmware are used for changing the power state of a device, usually the platform also provides a method for preparing the device to generate wakeup signals. In that case, however, it often also is necessary to prepare the device for generating PMEs using the native PCI PM mechanism, because the method provided by the platform depends on that.

Thus in many situations both the native and the platform-based power management mechanisms have to be used simultaneously to obtain the desired result.

1.2. Native PCI Power Management

The PCI Bus Power Management Interface Specification (PCI PM Spec) was introduced between the PCI 2.1 and PCI 2.2 Specifications. It defined a standard interface for performing various operations related to power management.

The implementation of the PCI PM Spec is optional for conventional PCI devices, but it is mandatory for PCI Express devices. If a device supports the PCI PM Spec, it has an 8 byte power management capability field in its PCI configuration space. This field is used to describe and control the standard features related to the native PCI power management.

The PCI PM Spec defines 4 operating states for devices (D0-D3) and for buses (B0-B3). The higher the number, the less power is drawn by the device or bus in that state. However, the higher the number, the longer the latency for the device or bus to return to the full-power state (D0 or B0, respectively).

There are two variants of the D3 state defined by the specification. The first one is D3hot, referred to as the software accessible D3, because devices can be programmed to go into it. The second one, D3cold, is the state that PCI devices are in when the supply voltage (Vcc) is removed from them. It is not possible to program a PCI device to go into D3cold, although there may be a programmable interface for putting the bus the device is on into a state in which Vcc is removed from all devices on the bus.

PCI bus power management, however, is not supported by the Linux kernel at the time of this writing and therefore it is not covered by this document.

Note that every PCI device can be in the full-power state (D0) or in D3cold, regardless of whether or not it implements the PCI PM Spec. In addition to that, if the PCI PM Spec is implemented by the device, it must support D3hot as well as D0. The support for the D1 and D2 power states is optional.

PCI devices supporting the PCI PM Spec can be programmed to go to any of the supported low-power states (except for D3cold). While in D1-D3hot the standard configuration registers of the device must be accessible to software (i.e. the device is required to respond to PCI configuration accesses), although its I/O and memory spaces are then disabled. This allows the device to be programmatically put into D0. Thus the kernel can switch the device back and forth between D0 and the supported low-power states (except for D3cold) and the possible power state transitions the device can undergo are the following:

Current State | New State
D0 | D1, D2, D3
D1 | D2, D3
D2 | D3
D1, D2, D3 | D0

The transition from D3cold to D0 occurs when the supply voltage is provided to the device (i.e. power is restored). In that case the device returns to D0 with a full power-on reset sequence and the power-on defaults are restored to the device by hardware just as at initial power up.

PCI devices supporting the PCI PM Spec can be programmed to generate PMEs while in any power state (D0-D3), but they are not required to be capable of generating PMEs from all supported power states. In particular, the capability of generating PMEs from D3cold is optional and depends on the presence of additional voltage (3.3Vaux) allowing the device to remain sufficiently active to generate a wakeup signal.

1.3. ACPI Device Power Management

The platform firmware support for the power management of PCI devices is system-specific. However, if the system in question is compliant with the Advanced Configuration and Power Interface (ACPI) Specification, like the majority of x86-based systems, it is supposed to implement device power management interfaces defined by the ACPI standard.

For this purpose the ACPI BIOS provides special functions called "control methods" that may be executed by the kernel to perform specific tasks, such as putting a device into a low-power state. These control methods are encoded using special byte-code language called the ACPI Machine Language (AML) and stored in the machine's BIOS. The kernel loads them from the BIOS and executes them as needed using an AML interpreter that translates the AML byte code into computations and memory or I/O space accesses. This way, in theory, a BIOS writer can provide the kernel with a means to perform actions depending on the system design in a system-specific fashion.

ACPI control methods may be divided into global control methods, that are not associated with any particular devices, and device control methods, that have to be defined separately for each device supposed to be handled with the help of the platform. This means, in particular, that ACPI device control methods can only be used to handle devices that the BIOS writer knew about in advance. The ACPI methods used for device power management fall into that category.

The ACPI specification assumes that devices can be in one of four power states labeled as D0, D1, D2, and D3 that roughly correspond to the native PCI PM D0-D3 states (although the difference between D3hot and D3cold is not taken into account by ACPI). Moreover, for each power state of a device there is a set of power resources that have to be enabled for the device to be put into that state. These power resources are controlled (i.e. enabled or disabled) with the help of their own control methods, _ON and _OFF, that have to be defined individually for each of them.

To put a device into the ACPI power state Dx (where x is a number between 0 and 3 inclusive) the kernel is supposed to (1) enable the power resources required by the device in this state using their _ON control methods and (2) execute the _PSx control method defined for the device. In addition to that, if the device is going to be put into a low-power state (D1-D3) and is supposed to generate wakeup signals from that state, the _DSW (or _PSW, replaced with _DSW by ACPI 3.0) control method defined for it has to be executed before _PSx. Power resources that are not required by the device in the target power state and are not required any more by any other device should be disabled (by executing their _OFF control methods). If the current power state of the device is D3, it can only be put into D0 this way.

However, quite often the power states of devices are changed during a system-wide transition into a sleep state or back into the working state. ACPI defines four system sleep states, S1, S2, S3, and S4, and denotes the system working state as S0. In general, the target system sleep (or working) state determines the highest power (lowest number) state the device can be put into and the kernel is supposed to obtain this information by executing the device's _SxD control method (where x is a number between 0 and 4 inclusive). If the device is required to wake up the system from the target sleep state, the lowest power (highest number) state it can be put into is also determined by the target state of the system. The kernel is then supposed to use the device's _SxW control method to obtain the number of that state. It also is supposed to use the device's _PRW control method to learn which power resources need to be enabled for the device to be able to generate wakeup signals.

1.4. Wakeup Signaling

Wakeup signals generated by PCI devices, either as native PCI PMEs, or as a result of the execution of the _DSW (or _PSW) ACPI control method before putting the device into a low-power state, have to be caught and handled as appropriate. If they are sent while the system is in the working state (ACPI S0), they should be translated into interrupts so that the kernel can put the devices generating them into the full-power state and take care of the events that triggered them. In turn, if they are sent while the system is sleeping, they should cause the system's core logic to trigger wakeup.

On ACPI-based systems wakeup signals sent by conventional PCI devices are converted into ACPI General-Purpose Events (GPEs) which are hardware signals from the system core logic generated in response to various events that need to be acted upon. Every GPE is associated with one or more sources of potentially interesting events. In particular, a GPE may be associated with a PCI device capable of signaling wakeup. The information on the connections between GPEs and event sources is recorded in the system's ACPI BIOS from where it can be read by the kernel.

If a PCI device known to the system's ACPI BIOS signals wakeup, the GPE associated with it (if there is one) is triggered. The GPEs associated with PCI bridges may also be triggered in response to a wakeup signal from one of the devices below the bridge (this also is the case for root bridges) and, for example, native PCI PMEs from devices unknown to the system's ACPI BIOS may be handled this way.

A GPE may be triggered when the system is sleeping (i.e. when it is in one of the ACPI S1-S4 states), in which case system wakeup is started by its core logic (the device that was the source of the signal causing the system wakeup to occur may be identified later). The GPEs used in such situations are referred to as wakeup GPEs.

Usually, however, GPEs are also triggered when the system is in the working state (ACPI S0) and in that case the system's core logic generates a System Control Interrupt (SCI) to notify the kernel of the event. Then, the SCI handler identifies the GPE that caused the interrupt to be generated which, in turn, allows the kernel to identify the source of the event (that may be a PCI device signaling wakeup). The GPEs used for notifying the kernel of events occurring while the system is in the working state are referred to as runtime GPEs.

Unfortunately, there is no standard way of handling wakeup signals sent by conventional PCI devices on systems that are not ACPI-based, but there is one for PCI Express devices. Namely, the PCI Express Base Specification introduced a native mechanism for converting native PCI PMEs into interrupts generated by root ports. For conventional PCI devices native PMEs are out-of-band, so they are routed separately and they need not pass through bridges (in principle they may be routed directly to the system's core logic), but for PCI Express devices they are in-band messages that have to pass through the PCI Express hierarchy, including the root port on the path from the device to the Root Complex. Thus it was possible to introduce a mechanism by which a root port generates an interrupt whenever it receives a PME message from one of the devices below it. The PCI Express Requester ID of the device that sent the PME message is then recorded in one of the root port's configuration registers from where it may be read by the interrupt handler allowing the device to be identified. [PME messages sent by PCI Express endpoints integrated with the Root Complex don't pass through root ports, but instead they cause a Root Complex Event Collector (if there is one) to generate interrupts.]

In principle the native PCI Express PME signaling may also be used on ACPI-based systems along with the GPEs, but to use it the kernel has to ask the system's ACPI BIOS to release control of root port configuration registers. The ACPI BIOS, however, is not required to allow the kernel to control these registers and if it doesn't do that, the kernel must not modify their contents. Of course the native PCI Express PME signaling cannot be used by the kernel in that case.

2. PCI Subsystem and Device Power Management

2.1. Device Power Management Callbacks

The PCI Subsystem participates in the power management of PCI devices in a number of ways. First of all, it provides an intermediate code layer between the device power management core (PM core) and PCI device drivers. Specifically, the pm field of the PCI subsystem's struct bus_type object, pci_bus_type, points to a struct dev_pm_ops object, pci_dev_pm_ops, containing pointers to several device power management callbacks:

```
const struct dev_pm_ops pci_dev_pm_ops = {
      .prepare = pci_pm_prepare,
      .complete = pci_pm_complete,
      .suspend = pci_pm_suspend,
      .resume = pci_pm_resume,
      .freeze = pci_pm_freeze,
      .thaw = pci_pm_thaw,
      .poweroff = pci_pm_poweroff,
      .restore = pci_pm_restore,
      .suspend_noirq = pci_pm_suspend_noirq,
      .resume_noirq = pci_pm_resume_noirq,
      .freeze_noirq = pci_pm_freeze_noirq,
      .thaw_noirq = pci_pm_thaw_noirq,
      .poweroff_noirq = pci_pm_poweroff_noirq,
      .restore_noirq = pci_pm_restore_noirq,
      .runtime_suspend = pci_pm_runtime_suspend,
      .runtime_resume = pci_pm_runtime_resume,
      .runtime_idle = pci_pm_runtime_idle,
};
```

These callbacks are executed by the PM core in various situations related to device power management and they, in turn, execute power management callbacks provided by PCI device drivers. They also perform power management operations involving some standard configuration registers of PCI devices that device drivers need not know or care about.

The structure representing a PCI device, struct pci_dev, contains several fields that these callbacks operate on:

```
struct pci_dev {
      ...
      pci_power_t     current_state;  /* Current operating state. */
      int             pm_cap;         /* PM capability offset in the
                                         configuration space */
      unsigned int    pme_support:5;  /* Bitmask of states from which PME#
```

```
                                           can be generated */
        unsigned int    pme_interrupt:1;/* Is native PCIe PME signaling used? */
        unsigned int    d1_support:1;   /* Low power state D1 is supported */
        unsigned int    d2_support:1;   /* Low power state D2 is supported */
        unsigned int    no_d1d2:1;      /* D1 and D2 are forbidden */
        unsigned int    wakeup_prepared:1;  /* Device prepared for wake up */
        unsigned int    d3hot_delay;    /* D3hot->D0 transition time in ms */
        ...
  };
```

They also indirectly use some fields of the struct device that is embedded in struct pci_dev.

2.2. Device Initialization

The PCI subsystem's first task related to device power management is to prepare the device for power management and initialize the fields of struct pci_dev used for this purpose. This happens in two functions defined in drivers/pci/pci.c, pci_pm_init() and platform_pci_wakeup_init().

The first of these functions checks if the device supports native PCI PM and if that's the case the offset of its power management capability structure in the configuration space is stored in the pm_cap field of the device's struct pci_dev object. Next, the function checks which PCI low-power states are supported by the device and from which low-power states the device can generate native PCI PMEs. The power management fields of the device's struct pci_dev and the struct device embedded in it are updated accordingly and the generation of PMEs by the device is disabled.

The second function checks if the device can be prepared to signal wakeup with the help of the platform firmware, such as the ACPI BIOS. If that is the case, the function updates the wakeup fields in struct device embedded in the device's struct pci_dev and uses the firmware-provided method to prevent the device from signaling wakeup.

At this point the device is ready for power management. For driverless devices, however, this functionality is limited to a few basic operations carried out during system-wide transitions to a sleep state and back to the working state.

2.3. Runtime Device Power Management

The PCI subsystem plays a vital role in the runtime power management of PCI devices. For this purpose it uses the general runtime power management (runtime PM) framework described in Documentation/power/runtime_pm.rst. Namely, it provides subsystem-level callbacks:

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

that are executed by the core runtime PM routines. It also implements the entire mechanics necessary for handling runtime wakeup signals from PCI devices in low-power states, which at the time of this writing works for both the native PCI Express PME signaling and the ACPI GPE-based wakeup signaling described in Section 1.

First, a PCI device is put into a low-power state, or suspended, with the help of pm_schedule_suspend() or pm_runtime_suspend() which for PCI devices call pci_pm_runtime_suspend() to do the actual job. For this to work, the device's driver has to provide a pm->runtime_suspend() callback (see below), which is run by pci_pm_runtime_suspend() as the first action. If the driver's callback returns successfully, the device's standard configuration registers are saved, the device is prepared to generate wakeup signals and, finally, it is put into the target low-power state.

The low-power state to put the device into is the lowest-power (highest number) state from which it can signal wakeup. The exact method of signaling wakeup is system-dependent and is determined by the PCI subsystem on the basis of the reported capabilities of the device and the platform firmware. To prepare the device for signaling wakeup and put it into the selected low-power state, the PCI subsystem can use the platform firmware as well as the device's native PCI PM capabilities, if supported.

It is expected that the device driver's pm->runtime_suspend() callback will not attempt to prepare the device for signaling wakeup or to put it into a low-power state. The driver ought to leave these tasks to the PCI subsystem that has all of the information necessary to perform them.

A suspended device is brought back into the "active" state, or resumed, with the help of pm_request_resume() or pm_runtime_resume() which both call pci_pm_runtime_resume() for PCI devices. Again, this only works if the device's driver provides a pm->runtime_resume() callback (see below). However, before the driver's callback is executed, pci_pm_runtime_resume() brings the device back into the full-power state, prevents it from signaling wakeup while in that state and restores its standard configuration registers. Thus the driver's callback need not worry about the PCI-specific aspects of the device resume.

Note that generally pci_pm_runtime_resume() may be called in two different situations. First, it may be called at the request of the device's driver, for example if there are some data for it to process. Second, it may be called as a result of a wakeup signal from the device itself (this sometimes is referred to as "remote wakeup"). Of course, for this purpose the wakeup signal is handled in one of the ways described in Section 1 and finally converted into a notification for the PCI subsystem after the source device has been identified.

The pci_pm_runtime_idle() function, called for PCI devices by pm_runtime_idle() and pm_request_idle(), executes the device driver's pm->runtime_idle() callback, if defined, and if that callback doesn't return error code (or is not present at all), suspends the device with the help of pm_runtime_suspend(). Sometimes pci_pm_runtime_idle() is called automatically by the PM core (for example, it is called right after the device has just been resumed), in which cases it is expected to suspend the device if that makes sense. Usually, however, the PCI subsystem doesn't really know if the device really can be suspended, so it lets the device's driver decide by running its pm->runtime_idle() callback.

2.4. System-Wide Power Transitions

There are a few different types of system-wide power transitions, described in Documentation/driver-api/pm/devices.rst. Each of them requires devices to be handled in a specific way and the PM core executes subsystem-level power management callbacks for this purpose. They are executed in phases such that each phase involves executing the same subsystem-level callback for every device belonging to the given subsystem before the next phase begins. These phases always run after tasks have been frozen.

2.4.1. System Suspend

When the system is going into a sleep state in which the contents of memory will be preserved, such as one of the ACPI sleep states S1-S3, the phases are:

> prepare, suspend, suspend_noirq.

The following PCI bus type's callbacks, respectively, are used in these phases:

```
pci_pm_prepare()
pci_pm_suspend()
pci_pm_suspend_noirq()
```

The pci_pm_prepare() routine first puts the device into the "fully functional" state with the help of pm_runtime_resume(). Then, it executes the device driver's pm->prepare() callback if defined (i.e. if the driver's struct dev_pm_ops object is present and the prepare pointer in that object is valid).

The pci_pm_suspend() routine first checks if the device's driver implements legacy PCI suspend routines (see Section 3), in which case the driver's legacy suspend callback is executed, if present, and its result is returned. Next, if the device's driver doesn't provide a struct dev_pm_ops object (containing pointers to the driver's callbacks), pci_pm_default_suspend() is called, which simply turns off the device's bus master capability and runs pcibios_disable_device() to disable it, unless the device is a bridge (PCI bridges are ignored by this routine). Next, the device driver's pm->suspend() callback is executed, if defined, and its result is returned if it fails. Finally, pci_fixup_device() is called to apply hardware suspend quirks related to the device if necessary.

Note that the suspend phase is carried out asynchronously for PCI devices, so the pci_pm_suspend() callback may be executed in parallel for any pair of PCI devices that don't depend on each other in a known way (i.e. none of the paths in the device tree from the root bridge to a leaf device contains both of them).

The pci_pm_suspend_noirq() routine is executed after suspend_device_irqs() has been called, which means that the device driver's interrupt handler won't be invoked while this routine is running. It first checks if the device's driver implements legacy PCI suspends routines (Section 3), in which case the legacy late suspend routine is called and its result is returned (the standard configuration registers of the device are saved if the driver's callback hasn't done that). Second, if the device driver's struct dev_pm_ops object is not present, the device's standard configuration registers are saved and the routine returns success. Otherwise the device driver's pm->suspend_noirq() callback is executed, if present, and its result is returned if it fails. Next, if the device's standard configuration registers haven't been saved yet (one of the device driver's callbacks executed before might do that), pci_pm_suspend_noirq() saves them, prepares the device to signal wakeup (if necessary) and puts it into a low-power state.

The low-power state to put the device into is the lowest-power (highest number) state from which it can signal wakeup while the system is in the target sleep state. Just like in the runtime PM case described above, the mechanism of signaling wakeup is system-dependent and determined by the PCI subsystem, which is also responsible for preparing the device to signal wakeup from the system's target sleep state as appropriate.

PCI device drivers (that don't implement legacy power management callbacks) are generally not expected to prepare devices for signaling wakeup or to put them into low-power states. However, if one of the driver's suspend callbacks (pm->suspend() or pm->suspend_noirq()) saves the device's standard configuration registers, pci_pm_suspend_noirq() will assume that the device has been prepared to signal wakeup and put into a low-power state by the driver (the driver is then assumed to have used the helper functions provided by the PCI subsystem for this purpose). PCI device drivers are not encouraged to do that, but in some rare cases doing that in the driver may be the optimum approach.

2.4.2. System Resume

When the system is undergoing a transition from a sleep state in which the contents of memory have been preserved, such as one of the ACPI sleep states S1-S3, into the working state (ACPI S0), the phases are:

> resume_noirq, resume, complete.

The following PCI bus type's callbacks, respectively, are executed in these phases:

```
pci_pm_resume_noirq()
pci_pm_resume()
pci_pm_complete()
```

The pci_pm_resume_noirq() routine first puts the device into the full-power state, restores its standard configuration registers and applies early resume hardware quirks related to the device, if necessary. This is done unconditionally, regardless of whether or not the device's driver implements legacy PCI power management callbacks (this way all PCI devices are in the full-power state and their standard configuration registers have been restored when their interrupt handlers are invoked for the first time during resume, which allows the kernel to avoid problems with the handling of shared interrupts by drivers whose devices are still suspended). If legacy PCI power management callbacks (see Section 3) are implemented by the device's driver, the legacy early resume callback is executed and its result is returned. Otherwise, the device driver's pm->resume_noirq() callback is executed, if defined, and its result is returned.

The pci_pm_resume() routine first checks if the device's standard configuration registers have been restored and restores them if that's not the case (this only is necessary in the error path during a failing suspend). Next, resume hardware quirks related to the device are applied, if necessary, and if the device's driver implements legacy PCI power management callbacks (see Section 3), the driver's legacy resume callback is executed and its result is returned. Otherwise, the device's wakeup signaling mechanisms are blocked and its driver's pm->resume() callback is executed, if defined (the callback's result is then returned).

The resume phase is carried out asynchronously for PCI devices, like the suspend phase described above, which means that if two PCI devices don't depend on each other in a known way, the pci_pm_resume() routine may be executed for the both of them in parallel.

The pci_pm_complete() routine only executes the device driver's pm->complete() callback, if defined.

2.4.3. System Hibernation

System hibernation is more complicated than system suspend, because it requires a system image to be created and written into a persistent storage medium. The image is created atomically and all devices are quiesced, or frozen, before that happens.

The freezing of devices is carried out after enough memory has been freed (at the time of this writing the image creation requires at least 50% of system RAM to be free) in the following three phases:

> prepare, freeze, freeze_noirq

that correspond to the PCI bus type's callbacks:

```
pci_pm_prepare()
pci_pm_freeze()
pci_pm_freeze_noirq()
```

This means that the prepare phase is exactly the same as for system suspend. The other two phases, however, are different.

The pci_pm_freeze() routine is quite similar to pci_pm_suspend(), but it runs the device driver's pm->freeze() callback, if defined, instead of pm->suspend(), and it doesn't apply the suspend-related hardware quirks. It is executed asynchronously for different PCI devices that don't depend on each other in a known way.

The pci_pm_freeze_noirq() routine, in turn, is similar to pci_pm_suspend_noirq(), but it calls the device driver's pm->freeze_noirq() routine instead of pm->suspend_noirq(). It also doesn't attempt to prepare the device for signaling wakeup and put it into a low-power state. Still, it saves the device's standard configuration registers if they haven't been saved by one of the driver's callbacks.

Once the image has been created, it has to be saved. However, at this point all devices are frozen and they cannot handle I/O, while their ability to handle I/O is obviously necessary for the image saving. Thus they have to be brought back to the fully functional state and this is done in the following phases:

> thaw_noirq, thaw, complete

using the following PCI bus type's callbacks:

```
pci_pm_thaw_noirq()
pci_pm_thaw()
pci_pm_complete()
```

respectively.

The first of them, pci_pm_thaw_noirq(), is analogous to pci_pm_resume_noirq(). It puts the device into the full power state and restores its standard configuration registers. It also executes the device driver's pm->thaw_noirq() callback, if defined, instead of pm->resume_noirq().

The pci_pm_thaw() routine is similar to pci_pm_resume(), but it runs the device driver's pm->thaw() callback instead of pm->resume(). It is executed asynchronously for different PCI devices that don't depend on each other in a known way.

The complete phase is the same as for system resume.

After saving the image, devices need to be powered down before the system can enter the target sleep state (ACPI S4 for ACPI-based systems). This is done in three phases:

> prepare, poweroff, poweroff_noirq

where the prepare phase is exactly the same as for system suspend. The other two phases are analogous to the suspend and suspend_noirq phases, respectively. The PCI subsystem-level callbacks they correspond to:

```
pci_pm_poweroff()
pci_pm_poweroff_noirq()
```

work in analogy with pci_pm_suspend() and pci_pm_poweroff_noirq(), respectively, although they don't attempt to save the device's standard configuration registers.

2.4.4. System Restore

System restore requires a hibernation image to be loaded into memory and the pre-hibernation memory contents to be restored before the pre-hibernation system activity can be resumed.

As described in Documentation/driver-api/pm/devices.rst, the hibernation image is loaded into memory by a fresh instance of the kernel, called the boot kernel, which in turn is loaded and run by a boot loader in the usual way. After the boot kernel has loaded the image, it needs to replace its own code and data with the code and data of the "hibernated" kernel stored within the image, called the image kernel. For this purpose all devices are frozen just like before creating the image during hibernation, in the

> prepare, freeze, freeze_noirq

phases described above. However, the devices affected by these phases are only those having drivers in the boot kernel; other devices will still be in whatever state the boot loader left them.

Should the restoration of the pre-hibernation memory contents fail, the boot kernel would go through the "thawing" procedure described above, using the thaw_noirq, thaw, and complete phases (that will only affect the devices having drivers in the boot kernel), and then continue running normally.

If the pre-hibernation memory contents are restored successfully, which is the usual situation, control is passed to the image kernel, which then becomes responsible for bringing the system back to the working state. To achieve this, it must restore the devices' pre-hibernation functionality, which is done much like waking up from the memory sleep state, although it involves different phases:

> restore_noirq, restore, complete

The first two of these are analogous to the resume_noirq and resume phases described above, respectively, and correspond to the following PCI subsystem callbacks:

```
pci_pm_restore_noirq()
pci_pm_restore()
```

These callbacks work in analogy with pci_pm_resume_noirq() and pci_pm_resume(), respectively, but they execute the device driver's pm->restore_noirq() and pm->restore() callbacks, if available.

The complete phase is carried out in exactly the same way as during system resume.

3. PCI Device Drivers and Power Management

3.1. Power Management Callbacks

PCI device drivers participate in power management by providing callbacks to be executed by the PCI subsystem's power management routines described above and by controlling the runtime power management of their devices.

At the time of this writing there are two ways to define power management callbacks for a PCI device driver, the recommended one, based on using a dev_pm_ops structure described in Documentation/driver-api/pm/devices.rst, and the "legacy" one, in which the .suspend() and .resume() callbacks from struct pci_driver are used. The legacy approach, however, doesn't allow one to define runtime power management callbacks and is not really suitable for any new drivers. Therefore it is not covered by this document (refer to the source code to learn more about it).

It is recommended that all PCI device drivers define a struct dev_pm_ops object containing pointers to power management (PM) callbacks that will be executed by the PCI subsystem's PM routines in various circumstances. A pointer to the driver's struct dev_pm_ops object has to be assigned to the driver.pm field in its struct pci_driver object. Once that has happened, the "legacy" PM callbacks in struct pci_driver are ignored (even if they are not NULL).

The PM callbacks in struct dev_pm_ops are not mandatory and if they are not defined (i.e. the respective fields of struct dev_pm_ops are unset) the PCI subsystem will handle the device in a simplified default manner. If they are defined, though, they are expected to behave as described in the following subsections.

3.1.1. prepare()

The prepare() callback is executed during system suspend, during hibernation (when a hibernation image is about to be created), during power-off after saving a hibernation image and during system restore, when a hibernation image has just been loaded into memory.

This callback is only necessary if the driver's device has children that in general may be registered at any time. In that case the role of the prepare() callback is to prevent new children of the device from being registered until one of the resume_noirq(), thaw_noirq(), or restore_noirq() callbacks is run.

In addition to that the prepare() callback may carry out some operations preparing the device to be suspended, although it should not allocate memory (if additional memory is required to suspend the device, it has to be preallocated earlier, for example in a suspend/hibernate notifier as described in Documentation/driver-api/pm/notifiers.rst).

3.1.2. suspend()

The suspend() callback is only executed during system suspend, after prepare() callbacks have been executed for all devices in the system.

This callback is expected to quiesce the device and prepare it to be put into a low-power state by the PCI subsystem. It is not required (in fact it even is not recommended) that a PCI driver's suspend() callback save the standard configuration registers of the device, prepare it for waking up the system, or put it into a low-power state. All of these operations can very well be taken care of by the PCI subsystem, without the driver's participation.

However, in some rare case it is convenient to carry out these operations in a PCI driver. Then, pci_save_state(), pci_prepare_to_sleep(), and pci_set_power_state() should be used to save the device's standard configuration registers, to prepare it for system wakeup (if necessary), and to put it into a low-power state, respectively. Moreover, if the driver calls pci_save_state(), the PCI subsystem will not execute either pci_prepare_to_sleep(), or pci_set_power_state() for its device, so the driver is then responsible for handling the device as appropriate.

While the suspend() callback is being executed, the driver's interrupt handler can be invoked to handle an interrupt from the device, so all suspend-related operations relying on the driver's ability to handle interrupts should be carried out in this callback.

3.1.3. suspend_noirq()

The suspend_noirq() callback is only executed during system suspend, after suspend() callbacks have been executed for all devices in the system and after device interrupts have been disabled by the PM core.

The difference between suspend_noirq() and suspend() is that the driver's interrupt handler will not be invoked while suspend_noirq() is running. Thus suspend_noirq() can carry out operations that would cause race conditions to arise if they were performed in suspend().

3.1.4. freeze()

The freeze() callback is hibernation-specific and is executed in two situations, during hibernation, after prepare() callbacks have been executed for all devices in preparation for the creation of a system image, and during restore, after a system image has been loaded into memory from persistent storage and the prepare() callbacks have been executed for all devices.

The role of this callback is analogous to the role of the suspend() callback described above. In fact, they only need to be different in the rare cases when the driver takes the responsibility for putting the device into a low-power state.

In that cases the freeze() callback should not prepare the device system wakeup or put it into a low-power state. Still, either it or freeze_noirq() should save the device's standard configuration registers using pci_save_state().

3.1.5. freeze_noirq()

The freeze_noirq() callback is hibernation-specific. It is executed during hibernation, after prepare() and freeze() callbacks have been executed for all devices in preparation for the creation of a system image, and during restore, after a system image has been loaded into memory and after prepare() and freeze() callbacks have been executed for all devices. It is always executed after device interrupts have been disabled by the PM core.

The role of this callback is analogous to the role of the suspend_noirq() callback described above and it very rarely is necessary to define freeze_noirq().

The difference between freeze_noirq() and freeze() is analogous to the difference between suspend_noirq() and suspend().

3.1.6. poweroff()

The poweroff() callback is hibernation-specific. It is executed when the system is about to be powered off after saving a hibernation image to a persistent storage. prepare() callbacks are executed for all devices before poweroff() is called.

The role of this callback is analogous to the role of the suspend() and freeze() callbacks described above, although it does not need to save the contents of the device's registers. In particular, if the driver wants to put the device into a low-power state itself instead of allowing the PCI subsystem to do that, the poweroff() callback should use pci_prepare_to_sleep() and pci_set_power_state() to prepare the device for system wakeup and to put it into a low-power state, respectively, but it need not save the device's standard configuration registers.

3.1.7. poweroff_noirq()

The poweroff_noirq() callback is hibernation-specific. It is executed after poweroff() callbacks have been executed for all devices in the system.

The role of this callback is analogous to the role of the suspend_noirq() and freeze_noirq() callbacks described above, but it does not need to save the contents of the device's registers.

The difference between poweroff_noirq() and poweroff() is analogous to the difference between suspend_noirq() and suspend().

3.1.8. resume_noirq()

The resume_noirq() callback is only executed during system resume, after the PM core has enabled the non-boot CPUs. The driver's interrupt handler will not be invoked while resume_noirq() is running, so this callback can carry out operations that might race with the interrupt handler.

Since the PCI subsystem unconditionally puts all devices into the full power state in the resume_noirq phase of system resume and restores their standard configuration registers, resume_noirq() is usually not necessary. In general it should only be used for performing operations that would lead to race conditions if carried out by resume().

3.1.9. resume()

The resume() callback is only executed during system resume, after resume_noirq() callbacks have been executed for all devices in the system and device interrupts have been enabled by the PM core.

This callback is responsible for restoring the pre-suspend configuration of the device and bringing it back to the fully functional state. The device should be able to process I/O in a usual way after resume() has returned.

3.1.10. thaw_noirq()

The thaw_noirq() callback is hibernation-specific. It is executed after a system image has been created and the non-boot CPUs have been enabled by the PM core, in the thaw_noirq phase of hibernation. It also may be executed if the loading of a hibernation image fails during system restore (it is then executed after enabling the non-boot CPUs). The driver's interrupt handler will not be invoked while thaw_noirq() is running.

The role of this callback is analogous to the role of resume_noirq(). The difference between these two callbacks is that thaw_noirq() is executed after freeze() and freeze_noirq(), so in general it does not need to modify the contents of the device's registers.

3.1.11. thaw()

The thaw() callback is hibernation-specific. It is executed after thaw_noirq() callbacks have been executed for all devices in the system and after device interrupts have been enabled by the PM core.

This callback is responsible for restoring the pre-freeze configuration of the device, so that it will work in a usual way after thaw() has returned.

3.1.12. restore_noirq()

The restore_noirq() callback is hibernation-specific. It is executed in the restore_noirq phase of hibernation, when the boot kernel has passed control to the image kernel and the non-boot CPUs have been enabled by the image kernel's PM core.

This callback is analogous to resume_noirq() with the exception that it cannot make any assumption on the previous state of the device, even if the BIOS (or generally the platform firmware) is known to preserve that state over a suspend-resume cycle.

For the vast majority of PCI device drivers there is no difference between resume_noirq() and restore_noirq().

3.1.13. restore()

The restore() callback is hibernation-specific. It is executed after restore_noirq() callbacks have been executed for all devices in the system and after the PM core has enabled device drivers' interrupt handlers to be invoked.

This callback is analogous to resume(), just like restore_noirq() is analogous to resume_noirq(). Consequently, the difference between restore_noirq() and restore() is analogous to the difference between resume_noirq() and resume().

For the vast majority of PCI device drivers there is no difference between resume() and restore().

3.1.14. complete()

The complete() callback is executed in the following situations:

- during system resume, after resume() callbacks have been executed for all devices,
- during hibernation, before saving the system image, after thaw() callbacks have been executed for all devices,
- during system restore, when the system is going back to its pre-hibernation state, after restore() callbacks have been executed for all devices.

It also may be executed if the loading of a hibernation image into memory fails (in that case it is run after thaw() callbacks have been executed for all devices that have drivers in the boot kernel).

This callback is entirely optional, although it may be necessary if the prepare() callback performs operations that need to be reversed.

3.1.15. runtime_suspend()

The runtime_suspend() callback is specific to device runtime power management (runtime PM). It is executed by the PM core's runtime PM framework when the device is about to be suspended (i.e. quiesced and put into a low-power state) at run time.

This callback is responsible for freezing the device and preparing it to be put into a low-power state, but it must allow the PCI subsystem to perform all of the PCI-specific actions necessary for suspending the device.

3.1.16. runtime_resume()

The runtime_resume() callback is specific to device runtime PM. It is executed by the PM core's runtime PM framework when the device is about to be resumed (i.e. put into the full-power state and programmed to process I/O normally) at run time.

This callback is responsible for restoring the normal functionality of the device after it has been put into the full-power state by the PCI subsystem. The device is expected to be able to process I/O in the usual way after runtime_resume() has returned.

3.1.17. runtime_idle()

The runtime_idle() callback is specific to device runtime PM. It is executed by the PM core's runtime PM framework whenever it may be desirable to suspend the device according to the PM core's information. In particular, it is automatically executed right after runtime_resume() has returned in case the resume of the device has happened as a result of a spurious event.

This callback is optional, but if it is not implemented or if it returns 0, the PCI subsystem will call pm_runtime_suspend() for the device, which in turn will cause the driver's runtime_suspend() callback to be executed.

3.1.18. Pointing Multiple Callback Pointers to One Routine

Although in principle each of the callbacks described in the previous subsections can be defined as a separate function, it often is convenient to point two or more members of struct dev_pm_ops to the same routine. There are a few convenience macros that can be used for this purpose.

The SIMPLE_DEV_PM_OPS macro declares a struct dev_pm_ops object with one suspend routine pointed to by the .suspend(), .freeze(), and .poweroff() members and one resume routine pointed to by the .resume(), .thaw(), and .restore() members. The other function pointers in this struct dev_pm_ops are unset.

The UNIVERSAL_DEV_PM_OPS macro is similar to SIMPLE_DEV_PM_OPS, but it additionally sets the .runtime_resume() pointer to the same value as .resume() (and .thaw(), and .restore()) and the .runtime_suspend() pointer to the same value as .suspend() (and .freeze() and .poweroff()).

The SET_SYSTEM_SLEEP_PM_OPS can be used inside of a declaration of struct dev_pm_ops to indicate that one suspend routine is to be pointed to by the .suspend(), .freeze(), and .poweroff() members and one resume routine is to be pointed to by the .resume(), .thaw(), and .restore() members.

3.1.19. Driver Flags for Power Management

The PM core allows device drivers to set flags that influence the handling of power management for the devices by the core itself and by middle layer code including the PCI bus type. The flags should be set once at the driver probe time with the help of the dev_pm_set_driver_flags() function and they should not be updated directly afterwards.

The DPM_FLAG_NO_DIRECT_COMPLETE flag prevents the PM core from using the direct-complete mechanism allowing device suspend/resume callbacks to be skipped if the device is in runtime suspend when the system suspend starts. That also affects all of the ancestors of the device, so this flag should only be used if absolutely necessary.

The DPM_FLAG_SMART_PREPARE flag causes the PCI bus type to return a positive value from pci_pm_prepare() only if the ->prepare callback provided by the driver of the device returns a positive value. That allows the driver to opt out from using the direct-complete mechanism dynamically (whereas setting DPM_FLAG_NO_DIRECT_COMPLETE means permanent opt-out).

The DPM_FLAG_SMART_SUSPEND flag tells the PCI bus type that from the driver's perspective the device can be safely left in runtime suspend during system suspend. That causes pci_pm_suspend(), pci_pm_freeze() and pci_pm_poweroff() to avoid resuming the device from runtime suspend unless there are PCI-specific reasons for doing that. Also, it causes pci_pm_suspend_late/noirq() and pci_pm_poweroff_late/noirq() to return early if the device remains in runtime suspend during the "late" phase of the system-wide transition under way. Moreover, if the device is in runtime suspend in pci_pm_resume_noirq() or pci_pm_restore_noirq(), its runtime PM status will be changed to "active" (as it is going to be put into D0 going forward).

Setting the DPM_FLAG_MAY_SKIP_RESUME flag means that the driver allows its "noirq" and "early" resume callbacks to be skipped if the device can be left in suspend after a system-wide transition into the working state. This flag is taken into consideration by the PM core along with the power.may_skip_resume status bit of the device which is set by pci_pm_suspend_noirq() in certain situations. If the PM core determines that the driver's "noirq" and "early" resume callbacks should be skipped, the dev_pm_skip_resume() helper function will return "true" and that will cause pci_pm_resume_noirq() and pci_pm_resume_early() to return upfront without touching the device and executing the driver callbacks.

3.2. Device Runtime Power Management

In addition to providing device power management callbacks PCI device drivers are responsible for controlling the runtime power management (runtime PM) of their devices.

The PCI device runtime PM is optional, but it is recommended that PCI device drivers implement it at least in the cases where there is a reliable way of verifying that the device is not used (like when the network cable is detached from an Ethernet adapter or there are no devices attached to a USB controller).

To support the PCI runtime PM the driver first needs to implement the runtime_suspend() and runtime_resume() callbacks. It also may need to implement the runtime_idle() callback to prevent the device from being suspended again every time right after the runtime_resume() callback has returned (alternatively, the runtime_suspend() callback will have to check if the device should really be suspended and return -EAGAIN if that is not the case).

The runtime PM of PCI devices is enabled by default by the PCI core. PCI device drivers do not need to enable it and should not attempt to do so. However, it is blocked by pci_pm_init() that runs the pm_runtime_forbid() helper function. In addition to that, the runtime PM usage counter of each PCI device is incremented by local_pci_probe() before executing the probe callback provided by the device's driver.

If a PCI driver implements the runtime PM callbacks and intends to use the runtime PM framework provided by the PM core and the PCI subsystem, it needs to decrement the device's runtime PM usage counter in its probe callback function. If it doesn't do that, the counter will always be different from zero for the device and it will never be runtime-suspended. The simplest way to do that is by calling pm_runtime_put_noidle(), but if the driver wants to schedule an autosuspend right away, for example, it may call pm_runtime_put_autosuspend() instead for this purpose. Generally, it just needs to call a function that decrements the devices usage counter from its probe routine to make runtime PM work for the device.

It is important to remember that the driver's runtime_suspend() callback may be executed right after the usage counter has been decremented, because user space may already have caused the pm_runtime_allow() helper function unblocking the runtime PM of the device to run via sysfs, so the driver must be prepared to cope with that.

The driver itself should not call pm_runtime_allow(), though. Instead, it should let user space or some platform-specific code do that (user space can do it via sysfs as stated above), but it must be prepared to handle the runtime PM of the device correctly as soon as pm_runtime_allow() is called (which may happen at any time, even before the driver is loaded).

When the driver's remove callback runs, it has to balance the decrementation of the device's runtime PM usage counter at the probe time. For this reason, if it has decremented the counter in its probe callback, it must run pm_runtime_get_noresume() in its remove callback. [Since the core carries out a runtime resume of the device and bumps up the device's usage counter before running the driver's remove callback, the runtime PM of the device is effectively disabled for the duration of the remove execution and all runtime PM helper functions incrementing the device's usage counter are then effectively equivalent to pm_runtime_get_noresume().]

The runtime PM framework works by processing requests to suspend or resume devices, or to check if they are idle (in which cases it is reasonable to subsequently request that they be suspended). These requests are represented by work items put into the power management workqueue, pm_wq. Although there are a few situations in which power management requests are automatically queued by the PM core (for example, after processing a request to resume a device the PM core automatically queues a request to check if the device is idle), device drivers are generally responsible for queuing power management requests for their devices. For this purpose they should use the runtime PM helper functions provided by the PM core, discussed in Documentation/power/runtime_pm.rst.

Devices can also be suspended and resumed synchronously, without placing a request into pm_wq. In the majority of cases this also is done by their drivers that use helper functions provided by the PM core for this purpose.

For more information on the runtime PM of devices refer to Documentation/power/runtime_pm.rst.

4. Resources

PCI Local Bus Specification, Rev. 3.0

PCI Bus Power Management Interface Specification, Rev. 1.2

Advanced Configuration and Power Interface (ACPI) Specification, Rev. 3.0b

PCI Express Base Specification, Rev. 2.0

Documentation/driver-api/pm/devices.rst

Documentation/power/runtime_pm.rst