Device Power Management Basics

Author: Rafael J. Wysocki <rafael.j.wysocki@intel.com>

Most of the code in Linux is device drivers, so most of the Linux power management (PM) code is also driver-specific. Most drivers will do very little; others, especially for platforms with small batteries (like cell phones), will do a lot.

This writeup gives an overview of how drivers interact with system-wide power management goals, emphasizing the models and interfaces that are shared by everything that hooks up to the driver model core. Read it as background for the domain-specific work you'd do with any specific driver.

Two Models for Device Power Management

Drivers will use one or both of these models to put devices into low-power states:

System Sleep model:

> Drivers can enter low-power states as part of entering system-wide low-power states like "suspend" (also known as "suspend-to-RAM"), or (mostly for systems with disks) "hibernation" (also known as "suspend-to-disk").
>
> This is something that device, bus, and class drivers collaborate on by implementing various role-specific suspend and resume methods to cleanly power down hardware and software subsystems, then reactivate them without loss of data.
>
> Some drivers can manage hardware wakeup events, which make the system leave the low-power state. This feature may be enabled or disabled using the relevant :file:`/sys/devices/.../power/wakeup` file (for Ethernet drivers the ioctl interface used by ethtool may also be used for this purpose); enabling it may cost some power usage, but let the whole system enter low-power states more often.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 46); *backlink*

Unknown interpreted text role "file".

Runtime Power Management model:

> Devices may also be put into low-power states while the system is running, independently of other power management activity in principle. However, devices are not generally independent of each other (for example, a parent device cannot be suspended unless all of its child devices have been suspended). Moreover, depending on the bus type the device is on, it may be necessary to carry out some bus-specific operations on the device for this purpose. Devices put into low power states at run time may require special handling during system-wide power transitions (suspend or hibernation).
>
> For these reasons not only the device driver itself, but also the appropriate subsystem (bus type, device type or device class) driver and the PM core are involved in runtime power management. As in the system sleep power management case, they need to collaborate by implementing various role-specific suspend and resume methods, so that the hardware is cleanly powered down and reactivated without data or service loss.

There's not a lot to be said about those low-power states except that they are very system-specific, and often device-specific. Also, that if enough devices have been put into low-power states (at runtime), the effect may be very similar to entering some system-wide low-power state (system sleep) ... and that synergies exist, so that several drivers using runtime PM might put the system into a state where even deeper power saving options are available.

Most suspended devices will have quiesced all I/O: no more DMA or IRQs (except for wakeup events), no more data read or written, and requests from upstream drivers are no longer accepted. A given bus or platform may have different requirements though.

Examples of hardware wakeup events include an alarm from a real time clock, network wake-on-LAN packets, keyboard or mouse activity, and media insertion or removal (for PCMCIA, MMC/SD, USB, and so on).

Interfaces for Entering System Sleep States

There are programming interfaces provided for subsystems (bus type, device type, device class) and device drivers to allow them to participate in the power management of devices they are concerned with. These interfaces cover both system sleep and runtime power management.

Device Power Management Operations

Device power management operations, at the subsystem level as well as at the device driver level, are implemented by defining and populating objects of type struct dev_pm_ops defined in :file:`include/linux/pm.h`. The roles of the methods included in it will be explained in what follows. For now, it should be sufficient to remember that the last three methods are specific to runtime power management while the remaining ones are used during system-wide power transitions.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, **line 100**); *backlink*
>
> Unknown interpreted text role "file".

There also is a deprecated "old" or "legacy" interface for power management operations available at least for some subsystems. This approach does not use struct dev_pm_ops objects and it is suitable only for implementing system sleep power management methods in a limited way. Therefore it is not described in this document, so please refer directly to the source code for more information about it.

Subsystem-Level Methods

The core methods to suspend and resume devices reside in struct dev_pm_ops pointed to by the :c:member:`ops` member of struct dev_pm_domain, or by the :c:member:`pm` member of struct bus_type, struct device_type and struct class. They are mostly of interest to the people writing infrastructure for platforms and buses, like PCI or USB, or device type and device class drivers. They also are relevant to the writers of device drivers whose subsystems (PM domains, device types, device classes and bus types) don't provide all power management methods.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, **line 119**); *backlink*
>
> Unknown interpreted text role "c:member".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, **line 119**); *backlink*
>
> Unknown interpreted text role "c:member".

Bus drivers implement these methods as appropriate for the hardware and the drivers using it; PCI works differently from USB, and so on. Not many people write subsystem-level drivers; most driver code is a "device driver" that builds on top of bus-specific framework code.

For more information on these driver calls, see the description later; they are called in phases for every device, respecting the parent-child sequencing in the driver model tree.

:file:`/sys/devices/.../power/wakeup` files

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, **line 138**); *backlink*
>
> Unknown interpreted text role "file".

All device objects in the driver model contain fields that control the handling of system wakeup events (hardware signals that can force the system out of a sleep state). These fields are initialized by bus or device driver code using :c:func:`device_set_wakeup_capable()` and :c:func:`device_set_wakeup_enable()`, defined in :file:`include/linux/pm_wakeup.h`.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, **line 141**); *backlink*
>
> Unknown interpreted text role "c:func".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-

master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 141); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 141); *backlink*

Unknown interpreted text role "file".

The :c:member:`power.can_wakeup` flag just records whether the device (and its driver) can physically support wakeup events. The :c:func:`device_set_wakeup_capable()` routine affects this flag. The :c:member:`power.wakeup` field is a pointer to an object of type struct wakeup_source used for controlling whether or not the device should use its system wakeup mechanism and for notifying the PM core of system wakeup events signaled by the device. This object is only present for wakeup-capable devices (i.e. devices whose :c:member:`can_wakeup` flags are set) and is created (or removed) by :c:func:`device_set_wakeup_capable()`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 147); *backlink*

Unknown interpreted text role "c:member".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 147); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 147); *backlink*

Unknown interpreted text role "c:member".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 147); *backlink*

Unknown interpreted text role "c:member".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 147); *backlink*

Unknown interpreted text role "c:func".

Whether or not a device is capable of issuing wakeup events is a hardware matter, and the kernel is responsible for keeping track of it. By contrast, whether or not a wakeup-capable device should issue wakeup events is a policy decision, and it is managed by user space through a sysfs attribute: the :file:`power/wakeup` file. User space can write the "enabled" or "disabled" strings to it to indicate whether or not, respectively, the device is supposed to signal system wakeup. This file is only present if the :c:member:`power.wakeup` object exists for the given device and is created (or removed) along with that object, by :c:func:`device_set_wakeup_capable()`. Reads from the file will return the corresponding string.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 157); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 157); *backlink*

Unknown interpreted text role "c:member".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 157); *backlink*

Unknown interpreted text role "c:func".

The initial value in the :file:`power/wakeup` file is "disabled" for the majority of devices; the major exceptions are power buttons, keyboards, and Ethernet adapters whose WoL (wake-on-LAN) feature has been set up with ethtool. It should also default to "enabled" for devices that don't generate wakeup requests on their own but merely forward wakeup requests from one bus to another (like PCI Express ports).

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 168); *backlink*

Unknown interpreted text role "file".

The :c:func:`device_may_wakeup()` routine returns true only if the :c:member:`power.wakeup` object exists and the corresponding :file:`power/wakeup` file contains the "enabled" string. This information is used by subsystems, like the PCI bus type code, to see whether or not to enable the devices' wakeup mechanisms. If device wakeup mechanisms are enabled or disabled directly by drivers, they also should use :c:func:`device_may_wakeup()` to decide what to do during a system sleep transition. Device drivers, however, are not expected to call :c:func:`device_set_wakeup_enable()` directly in any case.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 175); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 175); *backlink*

Unknown interpreted text role "c:member".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 175); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 175); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 175); *backlink*

Unknown interpreted text role "c:func".

It ought to be noted that system wakeup is conceptually different from "remote wakeup" used by runtime power management, although it may be supported by the same physical mechanism. Remote wakeup is a feature allowing devices in low-power states to trigger specific interrupts to signal conditions in which they should be put into the full-power state. Those interrupts may or may not be used to signal system wakeup events, depending on the hardware design. On some systems it is impossible to trigger them from system sleep states. In any case, remote wakeup should always be enabled for runtime power management for all devices and drivers that support it.

:file:`/sys/devices/.../power/control` files

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api]`

[pm]devices.rst, line 195); *backlink*

Unknown interpreted text role "file".

Each device in the driver model has a flag to control whether it is subject to runtime power management. This flag, :c:member:`runtime_auto`, is initialized by the bus type (or generally subsystem) code using :c:func:`pm_runtime_allow()` or :c:func:`pm_runtime_forbid()`; the default is to allow runtime power management.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 198); *backlink*

Unknown interpreted text role "c:member".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 198); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 198); *backlink*

Unknown interpreted text role "c:func".

The setting can be adjusted by user space by writing either "on" or "auto" to the device's :file:`power/control` sysfs file. Writing "auto" calls :c:func:`pm_runtime_allow()`, setting the flag and allowing the device to be runtime power-managed by its driver. Writing "on" calls :c:func:`pm_runtime_forbid()`, clearing the flag, returning the device to full power if it was in a low-power state, and preventing the device from being runtime power-managed. User space can check the current value of the :c:member:`runtime_auto` flag by reading that file.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 204); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 204); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 204); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 204); *backlink*

Unknown interpreted text role "c:member".

The device's :c:member:`runtime_auto` flag has no effect on the handling of system-wide power transitions. In particular, the device can (and in the majority of cases should and will) be put into a low-power state during a system-wide transition to a sleep state even though its :c:member:`runtime_auto` flag is clear.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst, line 213); *backlink*

Unknown interpreted text role "c:member".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master] [Documentation] [driver-api] [pm]devices.rst`, line 213); *backlink*

Unknown interpreted text role "c:member".

For more information about the runtime power management framework, refer to Documentation/power/runtime_pm.rst.

Calling Drivers to Enter and Leave System Sleep States

When the system goes into a sleep state, each device's driver is asked to suspend the device by putting it into a state compatible with the target system state. That's usually some version of "off", but the details are system-specific. Also, wakeup-enabled devices will usually stay partly functional in order to wake the system.

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

For simple drivers, suspend might quiesce the device using class code and then turn its hardware as "off" as possible during suspend_noirq. The matching resume calls would then completely reinitialize the hardware before reactivating its class I/O queues.

More power-aware drivers might prepare the devices for triggering system wakeup events.

Call Sequence Guarantees

To ensure that bridges and similar links needing to talk to a device are available when the device is suspended or resumed, the device hierarchy is walked in a bottom-up order to suspend devices. A top-down order is used to resume those devices.

The ordering of the device hierarchy is defined by the order in which devices get registered: a child can never be registered, probed or resumed before its parent; and can't be removed or suspended after that parent.

The policy is that the device hierarchy should match hardware bus topology. [Or at least the control bus, for devices which use multiple busses.] In particular, this means that a device registration may fail if the parent of the device is suspending (i.e. has been chosen by the PM core as the next device to suspend) or has already suspended, as well as after all of the other devices have been suspended. Device drivers must be prepared to cope with such situations.

System Power Management Phases

Suspending or resuming the system is done in several phases. Different phases are used for suspend-to-idle, shallow (standby), and deep ("suspend-to-RAM") sleep states and the hibernation state ("suspend-to-disk"). Each phase involves executing callbacks for every device before the next phase begins. Not all buses or classes support all these callbacks and not all drivers use all the callbacks. The various phases always run after tasks have been frozen and before they are unfrozen. Furthermore, the `*_noirq` phases run at a time when IRQ handlers have been disabled (except for those marked with the IRQF_NO_SUSPEND flag).

All phases use PM domain, bus, type, class or driver callbacks (that is, methods defined in `dev->pm_domain->ops`, `dev->bus->pm`, `dev->type->pm`, `dev->class->pm` or `dev->driver->pm`). These callbacks are regarded by the PM core as mutually exclusive. Moreover, PM domain callbacks always take precedence over all of the other callbacks and, for example, type callbacks take precedence over bus, class and driver callbacks. To be precise, the following rules are used to determine which callback to execute in the given phase:

1. If `dev->pm_domain` is present, the PM core will choose the callback provided by `dev->pm_domain->ops` for execution.
2. Otherwise, if both `dev->type` and `dev->type->pm` are present, the callback provided by `dev->type->pm` will be chosen for execution.
3. Otherwise, if both `dev->class` and `dev->class->pm` are present, the callback provided by `dev->class->pm` will be chosen for execution.
4. Otherwise, if both `dev->bus` and `dev->bus->pm` are present, the callback provided by `dev->bus->pm` will be chosen for execution.

This allows PM domains and device types to override callbacks provided by bus types or device classes if necessary.

The PM domain, type, class and bus callbacks may in turn invoke device- or driver-specific methods stored in `dev->driver->pm`, but they don't have to do that.

If the subsystem callback chosen for execution is not present, the PM core will execute the corresponding method from the `dev->driver->pm` set instead if there is one.

Entering System Suspend

When the system goes into the freeze, standby or memory sleep state, the phases are: `prepare`, `suspend`, `suspend_late`, `suspend_noirq`.

1. The `prepare` phase is meant to prevent races by preventing new devices from being registered; the PM core

would never know that all the children of a device had been suspended if new children could be registered at will. [By contrast, from the PM core's perspective, devices may be unregistered at any time.] Unlike the other suspend-related phases, during the `prepare` phase the device hierarchy is traversed top-down.

After the `->prepare` callback method returns, no new children may be registered below the device. The method may also prepare the device or driver in some way for the upcoming system power transition, but it should not put the device into a low-power state. Moreover, if the device supports runtime power management, the `->prepare` callback method must not update its state in case it is necessary to resume it from runtime suspend later on.

For devices supporting runtime power management, the return value of the prepare callback can be used to indicate to the PM core that it may safely leave the device in runtime suspend (if runtime-suspended already), provided that all of the device's descendants are also left in runtime suspend. Namely, if the prepare callback returns a positive number and that happens for all of the descendants of the device too, and all of them (including the device itself) are runtime-suspended, the PM core will skip the `suspend`, `suspend_late` and `suspend_noirq` phases as well as all of the corresponding phases of the subsequent device resume for all of these devices. In that case, the `->complete` callback will be the next one invoked after the `->prepare` callback and is entirely responsible for putting the device into a consistent state as appropriate.

Note that this direct-complete procedure applies even if the device is disabled for runtime PM; only the runtime-PM status matters. It follows that if a device has system-sleep callbacks but does not support runtime PM, then its prepare callback must never return a positive value. This is because all such devices are initially set to runtime-suspended with runtime PM disabled.

This feature also can be controlled by device drivers by using the `DPM_FLAG_NO_DIRECT_COMPLETE` and `DPM_FLAG_SMART_PREPARE` driver power management flags. [Typically, they are set at the time the driver is probed against the device in question by passing them to the :c:func:`dev_pm_set_driver_flags` helper function.] If the first of these flags is set, the PM core will not apply the direct-complete procedure described above to the given device and, consequenty, to any of its ancestors. The second flag, when set, informs the middle layer code (bus types, device types, PM domains, classes) that it should take the return value of the `->prepare` callback provided by the driver into account and it may only return a positive value from its own `->prepare` callback if the driver's one also has returned a positive value.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 355);** ***backlink***
>
> Unknown interpreted text role "c:func".

2. The `->suspend` methods should quiesce the device to stop it from performing I/O. They also may save the device registers and put it into the appropriate low-power state, depending on the bus type the device is on, and they may enable wakeup events.

   However, for devices supporting runtime power management, the `->suspend` methods provided by subsystems (bus types and PM domains in particular) must follow an additional rule regarding what can be done to the devices before their drivers' `->suspend` methods are called. Namely, they may resume the devices from runtime suspend by calling :c:func:`pm_runtime_resume` for them, if that is necessary, but they must not update the state of the devices in any other way at that time (in case the drivers need to resume the devices from runtime suspend in their `->suspend` methods). In fact, the PM core prevents subsystems or drivers from putting devices into runtime suspend at these times by calling :c:func:`pm_runtime_get_noresume` before issuing the `->prepare` callback (and calling :c:func:`pm_runtime_put` after issuing the `->complete` callback).

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 374);** ***backlink***
>
> Unknown interpreted text role "c:func".

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 374);** ***backlink***
>
> Unknown interpreted text role "c:func".

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 374);** ***backlink***

Unknown interpreted text role "c:func".

3. For a number of devices it is convenient to split suspend into the "quiesce device" and "save device state" phases, in which cases `suspend_late` is meant to do the latter. It is always executed after runtime power management has been disabled for the device in question.
4. The `suspend_noirq` phase occurs after IRQ handlers have been disabled, which means that the driver's interrupt handler will not be called while the callback method is running. The `->suspend_noirq` methods should save the values of the device's registers that weren't saved previously and finally put the device into the appropriate low-power state.

   The majority of subsystems and device drivers need not implement this callback. However, bus types allowing devices to share interrupt vectors, like PCI, generally need it; otherwise a driver might encounter an error during the suspend phase by fielding a shared interrupt generated by some other device after its own device had been set to low power.

At the end of these phases, drivers should have stopped all I/O transactions (DMA, IRQs), saved enough state that they can re-initialize or restore previous state (as needed by the hardware), and placed the device into a low-power state. On many platforms they will gate off one or more clock sources; sometimes they will also switch off power supplies or reduce voltages. [Drivers supporting runtime PM may already have performed some or all of these steps.]

If :c:func:`device_may_wakeup()` returns `true`, the device should be prepared for generating hardware wakeup signals to trigger a system wakeup event when the system is in the sleep state. For example, :c:func:`enable_irq_wake()` might identify GPIO signals hooked up to a switch or other external hardware, and :c:func:`pci_enable_wake()` does something similar for the PCI PME signal.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 413); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 413); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 413); *backlink*

Unknown interpreted text role "c:func".

If any of these callbacks returns an error, the system won't enter the desired low-power state. Instead, the PM core will unwind its actions by resuming all the devices that were suspended.

Leaving System Suspend

When resuming from freeze, standby or memory sleep, the phases are: `resume_noirq`, `resume_early`, `resume`, `complete`.

1. The `->resume_noirq` callback methods should perform any actions needed before the driver's interrupt handlers are invoked. This generally means undoing the actions of the `suspend_noirq` phase. If the bus type permits devices to share interrupt vectors, like PCI, the method should bring the device and its driver into a state in which the driver can recognize if the device is the source of incoming interrupts, if any, and handle them correctly.

   For example, the PCI bus type's `->pm.resume_noirq()` puts the device into the full-power state (D0 in the PCI terminology) and restores the standard configuration registers of the device. Then it calls the device driver's `->pm.resume_noirq()` method to perform device-specific actions.
2. The `->resume_early` methods should prepare devices for the execution of the resume methods. This generally involves undoing the actions of the preceding `suspend_late` phase.
3. The `->resume` methods should bring the device back to its operating state, so that it can perform normal I/O. This generally involves undoing the actions of the `suspend` phase.
4. The `complete` phase should undo the actions of the `prepare` phase. For this reason, unlike the other resume-related phases, during the `complete` phase the device hierarchy is traversed bottom-up.

   Note, however, that new children may be registered below the device as soon as the `->resume` callbacks occur;

it's not necessary to wait until the `complete` phase runs.

Moreover, if the preceding `->prepare` callback returned a positive number, the device may have been left in runtime suspend throughout the whole system suspend and resume (its `->suspend`, `->suspend_late`, `->suspend_noirq`, `->resume_noirq`, `->resume_early`, and `->resume` callbacks may have been skipped). In that case, the `->complete` callback is entirely responsible for putting the device into a consistent state after system suspend if necessary. [For example, it may need to queue up a runtime resume request for the device for this purpose.] To check if that is the case, the `->complete` callback can consult the device's `power.direct_complete` flag. If that flag is set when the `->complete` callback is being run then the direct-complete mechanism was used, and special actions may be required to make the device work correctly afterward.

At the end of these phases, drivers should be as functional as they were before suspending: I/O can be performed using DMA and IRQs, and the relevant clocks are gated on.

However, the details here may again be platform-specific. For example, some systems support multiple "run" states, and the mode in effect at the end of resume might not be the one which preceded suspension. That means availability of certain clocks or power supplies changed, which could easily affect how a driver works.

Drivers need to be able to handle hardware which has been reset since all of the suspend methods were called, for example by complete reinitialization. This may be the hardest part, and the one most protected by NDA'd documents and chip errata. It's simplest if the hardware state hasn't changed since the suspend was carried out, but that can only be guaranteed if the target system sleep entered was suspend-to-idle. For the other system sleep states that may not be the case (and usually isn't for ACPI-defined system sleep states, like S3).

Drivers must also be prepared to notice that the device has been removed while the system was powered down, whenever that's physically possible. PCMCIA, MMC, USB, Firewire, SCSI, and even IDE are common examples of busses where common Linux platforms will see such removal. Details of how drivers will notice and handle such removals are currently bus-specific, and often involve a separate thread.

These callbacks may return an error value, but the PM core will ignore such errors since there's nothing it can do about them other than printing them in the system log.

Entering Hibernation

Hibernating the system is more complicated than putting it into sleep states, because it involves creating and saving a system image. Therefore there are more phases for hibernation, with a different set of callbacks. These phases always run after tasks have been frozen and enough memory has been freed.

The general procedure for hibernation is to quiesce all devices ("freeze"), create an image of the system memory while everything is stable, reactivate all devices ("thaw"), write the image to permanent storage, and finally shut down the system ("power off"). The phases used to accomplish this are: `prepare`, `freeze`, `freeze_late`, `freeze_noirq`, `thaw_noirq`, `thaw_early`, `thaw`, `complete`, `prepare`, `poweroff`, `poweroff_late`, `poweroff_noirq`.

1. The `prepare` phase is discussed in the "Entering System Suspend" section above.
2. The `->freeze` methods should quiesce the device so that it doesn't generate IRQs or DMA, and they may need to save the values of device registers. However the device does not have to be put in a low-power state, and to save time it's best not to do so. Also, the device should not be prepared to generate wakeup events.
3. The `freeze_late` phase is analogous to the `suspend_late` phase described earlier, except that the device should not be put into a low-power state and should not be allowed to generate wakeup events.
4. The `freeze_noirq` phase is analogous to the `suspend_noirq` phase discussed earlier, except again that the device should not be put into a low-power state and should not be allowed to generate wakeup events.

At this point the system image is created. All devices should be inactive and the contents of memory should remain undisturbed while this happens, so that the image forms an atomic snapshot of the system state.

5. The `thaw_noirq` phase is analogous to the `resume_noirq` phase discussed earlier. The main difference is that its methods can assume the device is in the same state as at the end of the `freeze_noirq` phase.
6. The `thaw_early` phase is analogous to the `resume_early` phase described above. Its methods should undo the actions of the preceding `freeze_late`, if necessary.
7. The `thaw` phase is analogous to the `resume` phase discussed earlier. Its methods should bring the device back to an operating state, so that it can be used for saving the image if necessary.
8. The `complete` phase is discussed in the "Leaving System Suspend" section above.

At this point the system image is saved, and the devices then need to be prepared for the upcoming system shutdown. This is much like suspending them before putting the system into the suspend-to-idle, shallow or deep sleep state, and the phases are similar.

9. The `prepare` phase is discussed above.
10. The `poweroff` phase is analogous to the `suspend` phase.
11. The `poweroff_late` phase is analogous to the `suspend_late` phase.

12. The `poweroff_noirq` phase is analogous to the `suspend_noirq` phase.

The `->poweroff`, `->poweroff_late` and `->poweroff_noirq` callbacks should do essentially the same things as the `->suspend`, `->suspend_late` and `->suspend_noirq` callbacks, respectively. A notable difference is that they need not store the device register values, because the registers should already have been stored during the `freeze`, `freeze_late` or `freeze_noirq` phases. Also, on many machines the firmware will power-down the entire system, so it is not necessary for the callback to put the device in a low-power state.

Leaving Hibernation

Resuming from hibernation is, again, more complicated than resuming from a sleep state in which the contents of main memory are preserved, because it requires a system image to be loaded into memory and the pre-hibernation memory contents to be restored before control can be passed back to the image kernel.

Although in principle the image might be loaded into memory and the pre-hibernation memory contents restored by the boot loader, in practice this can't be done because boot loaders aren't smart enough and there is no established protocol for passing the necessary information. So instead, the boot loader loads a fresh instance of the kernel, called "the restore kernel", into memory and passes control to it in the usual way. Then the restore kernel reads the system image, restores the pre-hibernation memory contents, and passes control to the image kernel. Thus two different kernel instances are involved in resuming from hibernation. In fact, the restore kernel may be completely different from the image kernel: a different configuration and even a different version. This has important consequences for device drivers and their subsystems.

To be able to load the system image into memory, the restore kernel needs to include at least a subset of device drivers allowing it to access the storage medium containing the image, although it doesn't need to include all of the drivers present in the image kernel. After the image has been loaded, the devices managed by the boot kernel need to be prepared for passing control back to the image kernel. This is very similar to the initial steps involved in creating a system image, and it is accomplished in the same way, using `prepare`, `freeze`, and `freeze_noirq` phases. However, the devices affected by these phases are only those having drivers in the restore 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 restore kernel would go through the "thawing" procedure described above, using the `thaw_noirq`, `thaw_early`, `thaw`, and `complete` phases, and then continue running normally. This happens only rarely. Most often the pre-hibernation memory contents are restored successfully and control is passed to the image kernel, which then becomes responsible for bringing the system back to the working state.

To achieve this, the image kernel must restore the devices' pre-hibernation functionality. The operation is much like waking up from a sleep state (with the memory contents preserved), although it involves different phases: `restore_noirq`, `restore_early`, `restore`, `complete`.

1. The `restore_noirq` phase is analogous to the `resume_noirq` phase.
2. The `restore_early` phase is analogous to the `resume_early` phase.
3. The `restore` phase is analogous to the `resume` phase.
4. The `complete` phase is discussed above.

The main difference from `resume[_early|_noirq]` is that `restore[_early|_noirq]` must assume the device has been accessed and reconfigured by the boot loader or the restore kernel. Consequently, the state of the device may be different from the state remembered from the `freeze`, `freeze_late` and `freeze_noirq` phases. The device may even need to be reset and completely re-initialized. In many cases this difference doesn't matter, so the `->resume[_early|_noirq]` and `->restore[_early|_norq]` method pointers can be set to the same routines. Nevertheless, different callback pointers are used in case there is a situation where it actually does matter.

Power Management Notifiers

There are some operations that cannot be carried out by the power management callbacks discussed above, because the callbacks occur too late or too early. To handle these cases, subsystems and device drivers may register power management notifiers that are called before tasks are frozen and after they have been thawed. Generally speaking, the PM notifiers are suitable for performing actions that either require user space to be available, or at least won't interfere with user space.

For details refer to Documentation/driver-api/pm/notifiers.rst.

Device Low-Power (suspend) States

Device low-power states aren't standard. One device might only handle "on" and "off", while another might support a dozen different versions of "on" (how many engines are active?), plus a state that gets back to "on" faster than from a full "off".

Some buses define rules about what different suspend states mean. PCI gives one example: after the suspend sequence completes, a non-legacy PCI device may not perform DMA or issue IRQs, and any wakeup events it issues would be issued through the PME# bus signal. Plus, there are several PCI-standard device states, some of which are optional.

In contrast, integrated system-on-chip processors often use IRQs as the wakeup event sources (so drivers would call :c:func:`enable_irq_wake`) and might be able to treat DMA completion as a wakeup event (sometimes DMA can stay active too, it'd only be the CPU and some peripherals that sleep).

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 675); *backlink*

Unknown interpreted text role "c:func".

Some details here may be platform-specific. Systems may have devices that can be fully active in certain sleep states, such as an LCD display that's refreshed using DMA while most of the system is sleeping lightly ... and its frame buffer might even be updated by a DSP or other non-Linux CPU while the Linux control processor stays idle.

Moreover, the specific actions taken may depend on the target system state. One target system state might allow a given device to be very operational; another might require a hard shut down with re-initialization on resume. And two different target systems might use the same device in different ways; the aforementioned LCD might be active in one product's "standby", but a different product using the same SOC might work differently.

Device Power Management Domains

Sometimes devices share reference clocks or other power resources. In those cases it generally is not possible to put devices into low-power states individually. Instead, a set of devices sharing a power resource can be put into a low-power state together at the same time by turning off the shared power resource. Of course, they also need to be put into the full-power state together, by turning the shared power resource on. A set of devices with this property is often referred to as a power domain. A power domain may also be nested inside another power domain. The nested domain is referred to as the sub-domain of the parent domain.

Support for power domains is provided through the :c:member:`pm_domain` field of struct device. This field is a pointer to an object of type struct dev_pm_domain, defined in :file:`include/linux/pm.h`, providing a set of power management callbacks analogous to the subsystem-level and device driver callbacks that are executed for the given device during all power transitions, instead of the respective subsystem-level callbacks. Specifically, if a device's :c:member:`pm_domain` pointer is not NULL, the `->suspend()` callback from the object pointed to by it will be executed instead of its subsystem's (e.g. bus type's) `->suspend()` callback and analogously for all of the remaining callbacks. In other words, power management domain callbacks, if defined for the given device, always take precedence over the callbacks provided by the device's subsystem (e.g. bus type).

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 707); *backlink*

Unknown interpreted text role "c:member".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 707); *backlink*

Unknown interpreted text role "file".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 707); *backlink*

Unknown interpreted text role "c:member".

The support for device power management domains is only relevant to platforms needing to use the same device driver power management callbacks in many different power domain configurations and wanting to avoid incorporating the support for power domains into subsystem-level callbacks, for example by modifying the platform bus type. Other platforms need not implement it or take it into account in any way.

Devices may be defined as IRQ-safe which indicates to the PM core that their runtime PM callbacks may be invoked with disabled interrupts (see Documentation/power/runtime_pm.rst for more information). If an IRQ-safe device belongs to a PM domain, the runtime PM of the domain will be disallowed, unless the domain itself is defined as IRQ-safe. However, it makes sense to define a PM domain as IRQ-safe only if all the devices in it are IRQ-safe. Moreover, if an IRQ-safe domain has a parent domain, the runtime PM of the parent is only allowed if the parent itself is IRQ-safe too with the additional restriction that all child domains of an IRQ-safe parent must also be IRQ-safe.

Runtime Power Management

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

A system-wide power transition can be started while some devices are in low power states due to runtime power management. The system sleep PM callbacks should recognize such situations and react to them appropriately, but the necessary actions are subsystem-specific.

In some cases the decision may be made at the subsystem level while in other cases the device driver may be left to decide. In some cases it may be desirable to leave a suspended device in that state during a system-wide power transition, but in other cases the device must be put back into the full-power state temporarily, for example so that its system wakeup capability can be disabled. This all depends on the hardware and the design of the subsystem and device driver in question.

If it is necessary to resume a device from runtime suspend during a system-wide transition into a sleep state, that can be done by calling :c:func:`pm_runtime_resume` from the `->suspend` callback (or the `->freeze` or `->poweroff` callback for transitions related to hibernation) of either the device's driver or its subsystem (for example, a bus type or a PM domain). However, subsystems must not otherwise change the runtime status of devices from their `->prepare` and `->suspend` callbacks (or equivalent) *before* invoking device drivers' `->suspend` callbacks (or equivalent).

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 763); *backlink*

Unknown interpreted text role "c:func".

The `DPM_FLAG_SMART_SUSPEND` Driver Flag

Some bus types and PM domains have a policy to resume all devices from runtime suspend upfront in their `->suspend` callbacks, but that may not be really necessary if the device's driver can cope with runtime-suspended devices. The driver can indicate this by setting `DPM_FLAG_SMART_SUSPEND` in :c:member:`power.driver_flags` at probe time, with the assistance of the :c:func:`dev_pm_set_driver_flags` helper routine.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 777); *backlink*

Unknown interpreted text role "c:member".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 777); *backlink*

Unknown interpreted text role "c:func".

Setting that flag causes the PM core and middle-layer code (bus types, PM domains etc.) to skip the `->suspend_late` and `->suspend_noirq` callbacks provided by the driver if the device remains in runtime suspend throughout those phases of the system-wide suspend (and similarly for the "freeze" and "poweroff" parts of system hibernation). [Otherwise the same driver callback might be executed twice in a row for the same device, which would not be valid in general.] If the middle-layer system-wide PM callbacks are present for the device then they are responsible for skipping these driver callbacks; if not then the PM core skips them. The subsystem callback routines can determine whether they need to skip the driver callbacks by testing the return value from the :c:func:`dev_pm_skip_suspend` helper function.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 784); *backlink*

Unknown interpreted text role "c:func".

In addition, with `DPM_FLAG_SMART_SUSPEND` set, the driver's `->thaw_noirq` and `->thaw_early` callbacks are skipped in hibernation if the device remained in runtime suspend throughout the preceding "freeze" transition. Again, if the middle-layer callbacks are present for the device, they are responsible for doing this, otherwise the PM core takes care of it.

The `DPM_FLAG_MAY_SKIP_RESUME` Driver Flag

During system-wide resume from a sleep state it's easiest to put devices into the full-power state, as explained in Documentation/power/runtime_pm.rst. [Refer to that document for more information regarding this particular issue as well as for information on the device runtime power management framework in general.] However, it often is desirable to leave devices in suspend after system transitions to the working state, especially if those devices had been in runtime suspend before the preceding system-wide suspend (or analogous) transition.

To that end, device drivers can use the `DPM_FLAG_MAY_SKIP_RESUME` flag to indicate to the PM core and middle-layer code that they allow their "noirq" and "early" resume callbacks to be skipped if the device can be left in suspend after system-wide PM transitions to the working state. Whether or not that is the case generally depends on the state of the device before the given system suspend-resume cycle and on the type of the system transition under way. In particular, the "thaw" and "restore" transitions related to hibernation are not affected by `DPM_FLAG_MAY_SKIP_RESUME` at all. [All callbacks are issued during the "restore" transition regardless of the flag settings, and whether or not any driver callbacks are skipped during the "thaw" transition depends whether or not the `DPM_FLAG_SMART_SUSPEND` flag is set (see above). In addition, a device is not allowed to remain in runtime suspend if any of its children will be returned to full power.]

The `DPM_FLAG_MAY_SKIP_RESUME` flag is taken into account in combination with the :c:member:`power.may_skip_resume` status bit set by the PM core during the "suspend" phase of suspend-type transitions. If the driver or the middle layer has a reason to prevent the driver's "noirq" and "early" resume callbacks from being skipped during the subsequent system resume transition, it should clear :c:member:`power.may_skip_resume` in its `->suspend`, `->suspend_late` or `->suspend_noirq` callback. [Note that the drivers setting `DPM_FLAG_SMART_SUSPEND` need to clear :c:member:`power.may_skip_resume` in their `->suspend` callback in case the other two are skipped.]

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 831); *backlink*

Unknown interpreted text role "c:member".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 831); *backlink*

Unknown interpreted text role "c:member".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 831); *backlink*

Unknown interpreted text role "c:member".

Setting the :c:member:`power.may_skip_resume` status bit along with the `DPM_FLAG_MAY_SKIP_RESUME` flag is necessary, but generally not sufficient, for the driver's "noirq" and "early" resume callbacks to be skipped. Whether or not they should be skipped can be determined by evaluating the :c:func:`dev_pm_skip_resume` helper function.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 841); *backlink*

Unknown interpreted text role "c:member".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\pm\[linux-master][Documentation][driver-api][pm]devices.rst`, line 841); *backlink*

Unknown interpreted text role "c:func".

If that function returns `true`, the driver's "noirq" and "early" resume callbacks should be skipped and the device's runtime PM status will be set to "suspended" by the PM core. Otherwise, if the device was runtime-suspended during the preceding system-wide suspend transition and its `DPM_FLAG_SMART_SUSPEND` is set, its runtime PM status will be set to "active" by the PM core. [Hence, the drivers that do not set `DPM_FLAG_SMART_SUSPEND` should not expect the runtime PM status of their devices to be changed from "suspended" to "active" by the PM core during system-wide resume-type transitions.]

If the `DPM_FLAG_MAY_SKIP_RESUME` flag is not set for a device, but `DPM_FLAG_SMART_SUSPEND` is set and the driver's "late" and "noirq" suspend callbacks are skipped, its system-wide "noirq" and "early" resume callbacks, if present, are invoked as usual and the device's runtime PM status is set to "active" by the PM core before enabling runtime PM for it. In that case, the driver must be prepared to cope with the invocation of its system-wide resume callbacks back-to-back with its `->runtime_suspend` one (without the intervening `->runtime_resume` and system-wide suspend callbacks) and the final state of the device must reflect the "active" runtime PM status in that case. [Note that this is not a problem at all if the driver's `->suspend_late` callback pointer points to the same function as its `->runtime_suspend` one and its `->resume_early` callback pointer points to the same function as the `->runtime_resume` one, while none of the other system-wide suspend-resume callbacks of the driver are present, for example.]

Likewise, if `DPM_FLAG_MAY_SKIP_RESUME` is set for a device, its driver's system-wide "noirq" and "early" resume callbacks may be skipped while its "late" and "noirq" suspend callbacks may have been executed (in principle, regardless of whether or not

`DPM_FLAG_SMART_SUSPEND` is set). In that case, the driver needs to be able to cope with the invocation of its `->runtime_resume` callback back-to-back with its "late" and "noirq" suspend ones. [For instance, that is not a concern if the driver sets both `DPM_FLAG_SMART_SUSPEND` and `DPM_FLAG_MAY_SKIP_RESUME` and uses the same pair of suspend/resume callback functions for runtime PM and system-wide suspend/resume.]