Device Power Management

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 /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.

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:

```
struct dev_pm_ops {
        int (*prepare)(struct device *dev);
        void (*complete)(struct device *dev);
        int (*suspend)(struct device *dev);
        int (*resume)(struct device *dev);
        int (*freeze)(struct device *dev);
        int (*thaw)(struct device *dev);
        int (*poweroff)(struct device *dev);
        int (*restore)(struct device *dev);
        int (*suspend_noirq)(struct device *dev);
        int (*resume_noirq)(struct device *dev);
        int (*freeze_noirq)(struct device *dev);
        int (*thaw_noirq)(struct device *dev);
        int (*poweroff_noirq)(struct device *dev);
        int (*restore_noirq)(struct device *dev);
        int (*runtime_suspend)(struct device *dev);
        int (*runtime_resume)(struct device *dev);
        int (*runtime_idle)(struct device *dev);
};
```

This structure is defined in include/linux/pm.h and the methods included in it are also described in that file. Their roles 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.

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. 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 pm member of struct bus_type, struct device_type and struct class. They are mostly of interest to the people writing infrastructure for buses, like PCI or USB, or device type and device class drivers.

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.

/sys/devices/.../power/wakeup files

All devices in the driver model have two flags to control handling of wakeup events (hardware signals that can force the device and/or system out of a low power state). These flags are initialized by bus or device driver code using device_set_wakeup_capable() and device_set_wakeup_enable(), defined in include/linux/pm_wakeup.h.

The "can_wakeup" flag just records whether the device (and its driver) can physically support wakeup events. The device_set_wakeup_capable() routine affects this flag. The "should_wakeup" flag controls whether the device should try to use its wakeup mechanism. device_set_wakeup_enable() affects this flag; for the most part drivers should not change its value. The initial value of should_wakeup is supposed to be false 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.

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 power/wakeup file. User space can write the strings "enabled" or "disabled" to set or clear the should_wakeup flag, respectively. Reads from the file will return the corresponding string if can_wakeup is true, but if can_wakeup is false then reads will return an empty string, to indicate that the device doesn't support wakeup events. (But even though the file appears empty, writes will still affect the should_wakeup flag.)

The device_may_wakeup() routine returns true only if both flags are set.

Drivers should check this routine when putting devices in a low-power state during a system sleep transition, to see whether or not to enable the devices' wakeup mechanisms. However for runtime power management, wakeup events should be enabled whenever the device and driver both support them, regardless of the should_wakeup flag.

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

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

The setting can be adjusted by user space by writing either "on" or "auto" to the device's power/control sysfs file. Writing "auto" calls pm_runtime_allow(), setting the flag and allowing the device to be runtime power-managed by its driver. Writing "on" calls 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 runtime_auto flag by reading the file.

The device's 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 runtime_auto flag is clear.

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

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 tree 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 tree 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 tree 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 standby or memory sleep states ("suspend-to-RAM") and the hibernation state ("suspend-to-disk"). Each phase involves executing callbacks for every device before the next phase begins. Not all busses 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 IRQ_WAKEUP flag).

Most phases use bus, type, and class callbacks (that is, methods defined in dev->bus->pm, dev->type->pm, and dev->class->pm). The prepare and complete phases are exceptions; they use only bus callbacks. When multiple callbacks are used in a phase, they are invoked in the order: <class, type, bus> during power-down transitions and in the opposite order during power-up transitions. For example, during the suspend phase the PM core invokes

```
dev->class->pm.suspend(dev);
dev->type->pm.suspend(dev);
dev->bus->pm.suspend(dev);
```

before moving on to the next device, whereas during the resume phase the core invokes

```
dev->bus->pm.resume(dev);
dev->type->pm.resume(dev);
dev->class->pm.resume(dev);
```

These callbacks may in turn invoke device- or driver-specific methods stored in dev->driver->pm, but they don't have to.

Entering System Suspend

When the system goes into the standby or memory sleep state, the phases are:

prepare, suspend, 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, devices may be unregistered at any time.) Unlike the other suspend-related phases, during the prepare phase the device tree is traversed top-down.

The prepare phase uses only a bus callback. After the 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.

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.

3. 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 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 device_may_wakeup(dev) 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, enable_irq_wake() might identify GPIO signals hooked up to a switch or other external hardware, and pci_enable_wake() does something similar for the PCI PME signal.

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 standby or memory sleep, the phases are:

resume_noirq, 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 methods should bring the 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.

3. The complete phase uses only a bus callback. The method should undo the actions of the prepare phase. 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.

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. Even if the device was in a low-power state before the system sleep because of runtime power management, afterwards it should be back in its full-power state. There are multiple reasons why it's best to do this; they are discussed in more detail in Documentation/power/runtime_pm.txt.

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 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't be guaranteed (in fact, it ususally is not the case).

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 the standby or

memory sleep state, 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 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 (poweroff). The phases used to accomplish this are:

prepare, freeze, freeze_noirq, thaw_noirq, thaw, complete,
prepare, poweroff, 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_noirq phase is analogous to the suspend_noirq phase discussed above, except again that the device should not be put in 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.

4. The thaw_noirq phase is analogous to the resume_noirq phase discussed above. 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.

5. The thaw phase is analogous to the resume phase discussed above. Its methods should bring the device back to an operating state, so that it can be used for saving the image if necessary.

6. 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 standby or memory sleep state, and the phases are similar.

7. The prepare phase is discussed above.

8. The poweroff phase is analogous to the suspend phase.

9. The poweroff_noirq phase is analogous to the suspend_noirq phase.

The poweroff and poweroff_noirq callbacks should do essentially the same things as the suspend and suspend_noirq callbacks. The only notable difference is that they need not store the device register values, because the registers should already have been stored during the freeze or freeze_noirq phases.

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 boot kernel, into memory and passes control to it in the usual way. Then the boot kernel reads the system image, restores the pre-hibernation memory contents, and passes control to the image kernel. Thus two different kernels are involved in resuming from hibernation. In fact, the boot 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 boot 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 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, 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 the memory sleep state, although it involves different phases:

    restore_noirq, restore, complete

1. The restore_noirq phase is analogous to the resume_noirq phase.

2. The restore phase is analogous to the resume phase.

3. The complete phase is discussed above.

The main difference from resume[_noirq] is that restore[_noirq] must assume the device has been accessed and reconfigured by the boot loader or the boot kernel. Consequently the state of the device may be different from the state remembered from the freeze 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[_noirq] and restore[_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 matters.

System Devices

System devices (sysdevs) follow a slightly different API, which can be found in

include/linux/sysdev.h
drivers/base/sys.c

System devices will be suspended with interrupts disabled, and after all other devices have been suspended. On resume, they will be resumed before any other devices, and also with interrupts disabled. These things occur in special "sysdev_driver" phases, which affect only system devices.

Thus, after the suspend_noirq (or freeze_noirq or poweroff_noirq) phase, when the non-boot CPUs are all offline and IRQs are disabled on the remaining online CPU, then a sysdev_driver.suspend phase is carried out, and the system enters a sleep state (or a system image is created). During resume (or after the image has been created or loaded) a sysdev_driver.resume phase is carried out, IRQs are enabled on the only online CPU, the non-boot CPUs are enabled, and the resume_noirq (or thaw_noirq or restore_noirq) phase begins.

Code to actually enter and exit the system-wide low power state sometimes involves hardware details that are only known to the boot firmware, and may leave a CPU running software (from SRAM or flash memory) that monitors the system and manages its wakeup sequence.

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 busses 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 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).

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.

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/power/notifiers.txt.

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.

During system-wide resume from a sleep state it's best to put devices into the full-power state, as explained in Documentation/power/runtime_pm.txt. 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.