Runtime Power Management Framework for I/O Devices

C. 2009-2011 Rafael J. Wysocki <rjw@sisk.pl>, Novell Inc.

C. 2010 Alan Stern <stern@rowland.harvard.edu>

C. 2014 Intel Corp., Rafael J. Wysocki <rafael.j.wysocki@intel.com>

1. Introduction

Support for runtime power management (runtime PM) of I/O devices is provided at the power management core (PM core) level by means of:

- The power management workqueue pm_wq in which bus types and device drivers can put their PM-related work items. It is strongly recommended that pm_wq be used for queuing all work items related to runtime PM, because this allows them to be synchronized with system-wide power transitions (suspend to RAM, hibernation and resume from system sleep states). pm_wq is declared in include/linux/pm_runtime.h and defined in kernel/power/main.c.
- A number of runtime PM fields in the 'power' member of 'struct device' (which is of the type 'struct dev_pm_info', defined in include/linux/pm.h) that can be used for synchronizing runtime PM operations with one another.
- Three device runtime PM callbacks in 'struct dev_pm_ops' (defined in include/linux/pm.h).
- A set of helper functions defined in drivers/base/power/runtime.c that can be used for carrying out runtime PM operations in such a way that the synchronization between them is taken care of by the PM core. Bus types and device drivers are encouraged to use these functions.

The runtime PM callbacks present in 'struct dev_pm_ops', the device runtime PM fields of 'struct dev_pm_info' and the core helper functions provided for runtime PM are described below.

2. Device Runtime PM Callbacks

There are three device runtime PM callbacks defined in 'struct dev_pm_ops':

```
struct dev_pm_ops {
        ...
        int (*runtime_suspend)(struct device *dev);
        int (*runtime_resume)(struct device *dev);
        int (*runtime_idle)(struct device *dev);
        ...
};
```

The ->runtime_suspend(), ->runtime_resume() and ->runtime_idle() callbacks are executed by the PM core for the device's subsystem that may be either of the following:

1. PM domain of the device, if the device's PM domain object, dev->pm_domain, is present.
2. Device type of the device, if both dev->type and dev->type->pm are present.
3. Device class of the device, if both dev->class and dev->class->pm are present.
4. Bus type of the device, if both dev->bus and dev->bus->pm are present.

If the subsystem chosen by applying the above rules doesn't provide the relevant callback, the PM core will invoke the corresponding driver callback stored in dev->driver->pm directly (if present).

The PM core always checks which callback to use in the order given above, so the priority order of callbacks from high to low is: PM domain, device type, class and bus type. Moreover, the high-priority one will always take precedence over a low-priority one. The PM domain, bus type, device type and class callbacks are referred to as subsystem-level callbacks in what follows.

By default, the callbacks are always invoked in process context with interrupts enabled. However, the pm_runtime_irq_safe() helper function can be used to tell the PM core that it is safe to run the ->runtime_suspend(), ->runtime_resume() and ->runtime_idle() callbacks for the given device in atomic context with interrupts disabled. This implies that the callback routines in question must not block or sleep, but it also means that the synchronous helper functions listed at the end of Section 4 may be used for that device within an interrupt handler or generally in an atomic context.

The subsystem-level suspend callback, if present, is _entirely_ _responsible_ for handling the suspend of the device as appropriate, which may, but need not include executing the device driver's own ->runtime_suspend() callback (from the PM core's point of view it is not necessary to implement a ->runtime_suspend() callback in a device driver as long as the subsystem-level suspend callback knows what to do to handle the device).

- Once the subsystem-level suspend callback (or the driver suspend callback, if invoked directly) has completed successfully for the given device, the PM core regards the device as suspended, which need not mean that it has been put into a low power state. It is supposed to mean, however, that the device will not process data and will not communicate with the CPU(s) and RAM until the appropriate resume callback is executed for it. The runtime PM status of a device after successful execution of the suspend callback is 'suspended'.

- If the suspend callback returns -EBUSY or -EAGAIN, the device's runtime PM status remains 'active', which means that the device _must_ be fully operational afterwards.
- If the suspend callback returns an error code different from -EBUSY and -EAGAIN, the PM core regards this as a fatal error and will refuse to run the helper functions described in Section 4 for the device until its status is directly set to either 'active', or 'suspended' (the PM core provides special helper functions for this purpose).

In particular, if the driver requires remote wakeup capability (i.e. hardware mechanism allowing the device to request a change of its power state, such as PCI PME) for proper functioning and device_can_wakeup() returns 'false' for the device, then ->runtime_suspend() should return -EBUSY. On the other hand, if device_can_wakeup() returns 'true' for the device and the device is put into a low-power state during the execution of the suspend callback, it is expected that remote wakeup will be enabled for the device. Generally, remote wakeup should be enabled for all input devices put into low-power states at run time.

The subsystem-level resume callback, if present, is **entirely responsible** for handling the resume of the device as appropriate, which may, but need not include executing the device driver's own ->runtime_resume() callback (from the PM core's point of view it is not necessary to implement a ->runtime_resume() callback in a device driver as long as the subsystem-level resume callback knows what to do to handle the device).

- Once the subsystem-level resume callback (or the driver resume callback, if invoked directly) has completed successfully, the PM core regards the device as fully operational, which means that the device _must_ be able to complete I/O operations as needed. The runtime PM status of the device is then 'active'.
- If the resume callback returns an error code, the PM core regards this as a fatal error and will refuse to run the helper functions described in Section 4 for the device, until its status is directly set to either 'active', or 'suspended' (by means of special helper functions provided by the PM core for this purpose).

The idle callback (a subsystem-level one, if present, or the driver one) is executed by the PM core whenever the device appears to be idle, which is indicated to the PM core by two counters, the device's usage counter and the counter of 'active' children of the device.

- If any of these counters is decreased using a helper function provided by the PM core and it turns out to be equal to zero, the other counter is checked. If that counter also is equal to zero, the PM core executes the idle callback with the device as its argument.

The action performed by the idle callback is totally dependent on the subsystem (or driver) in question, but the expected and recommended action is to check if the device can be suspended (i.e. if all of the conditions necessary for suspending the device are satisfied) and to queue up a suspend request for the device in that case. If there is no idle callback, or if the callback returns 0, then the PM core will attempt to carry out a runtime suspend of the device, also respecting devices configured for autosuspend. In essence this means a call to pm_runtime_autosuspend() (do note that drivers needs to update the device last busy mark, pm_runtime_mark_last_busy(), to control the delay under this circumstance). To prevent this (for example, if the callback routine has started a delayed suspend), the routine must return a non-zero value. Negative error return codes are ignored by the PM core.

The helper functions provided by the PM core, described in Section 4, guarantee that the following constraints are met with respect to runtime PM callbacks for one device:

1. The callbacks are mutually exclusive (e.g. it is forbidden to execute ->runtime_suspend() in parallel with ->runtime_resume() or with another instance of ->runtime_suspend() for the same device) with the exception that ->runtime_suspend() or ->runtime_resume() can be executed in parallel with ->runtime_idle() (although ->runtime_idle() will not be started while any of the other callbacks is being executed for the same device).
2. ->runtime_idle() and ->runtime_suspend() can only be executed for 'active' devices (i.e. the PM core will only execute ->runtime_idle() or ->runtime_suspend() for the devices the runtime PM status of which is 'active').
3. ->runtime_idle() and ->runtime_suspend() can only be executed for a device the usage counter of which is equal to zero _and_ either the counter of 'active' children of which is equal to zero, or the 'power.ignore_children' flag of which is set.
4. ->runtime_resume() can only be executed for 'suspended' devices (i.e. the PM core will only execute ->runtime_resume() for the devices the runtime PM status of which is 'suspended').

Additionally, the helper functions provided by the PM core obey the following rules:

- If ->runtime_suspend() is about to be executed or there's a pending request to execute it, ->runtime_idle() will not be executed for the same device.
- A request to execute or to schedule the execution of ->runtime_suspend() will cancel any pending requests to execute ->runtime_idle() for the same device.
- If ->runtime_resume() is about to be executed or there's a pending request to execute it, the other callbacks will not be executed for the same device.
- A request to execute ->runtime_resume() will cancel any pending or scheduled requests to execute the other callbacks for the same device, except for scheduled autosuspends.

3. Runtime PM Device Fields

The following device runtime PM fields are present in 'struct dev_pm_info', as defined in include/linux/pm.h:

struct timer_list suspend_timer;
- timer used for scheduling (delayed) suspend and autosuspend requests

unsigned long timer_expires;
- timer expiration time, in jiffies (if this is different from zero, the timer is running and will expire at that time, otherwise the timer is not running)

struct work_struct work;
- work structure used for queuing up requests (i.e. work items in pm_wq)

wait_queue_head_t wait_queue;
- wait queue used if any of the helper functions needs to wait for another one to complete

spinlock_t lock;
- lock used for synchronization

atomic_t usage_count;
- the usage counter of the device

atomic_t child_count;
- the count of 'active' children of the device

unsigned int ignore_children;
- if set, the value of child_count is ignored (but still updated)

unsigned int disable_depth;
- used for disabling the helper functions (they work normally if this is equal to zero); the initial value of it is 1 (i.e. runtime PM is initially disabled for all devices)

int runtime_error;
- if set, there was a fatal error (one of the callbacks returned error code as described in Section 2), so the helper functions will not work until this flag is cleared; this is the error code returned by the failing callback

unsigned int idle_notification;
- if set, ->runtime_idle() is being executed

unsigned int request_pending;
- if set, there's a pending request (i.e. a work item queued up into pm_wq)

enum rpm_request request;
- type of request that's pending (valid if request_pending is set)

unsigned int deferred_resume;
- set if ->runtime_resume() is about to be run while ->runtime_suspend() is being executed for that device and it is not practical to wait for the suspend to complete; means "start a resume as soon as you've suspended"

enum rpm_status runtime_status;
- the runtime PM status of the device; this field's initial value is RPM_SUSPENDED, which means that each device is initially regarded by the PM core as 'suspended', regardless of its real hardware status

enum rpm_status last_status;
- the last runtime PM status of the device captured before disabling runtime PM for it (invalid initially and when disable_depth is 0)

unsigned int runtime_auto;
- if set, indicates that the user space has allowed the device driver to power manage the device at run time via the /sys/devices/.../power/control *interface;* it may only be modified with the help of the pm_runtime_allow() and pm_runtime_forbid() helper functions

unsigned int no_callbacks;
- indicates that the device does not use the runtime PM callbacks (see Section 8); it may be modified only by the pm_runtime_no_callbacks() helper function

unsigned int irq_safe;
- indicates that the ->runtime_suspend() and ->runtime_resume() callbacks will be invoked with the spinlock held and interrupts disabled

unsigned int use_autosuspend;
- indicates that the device's driver supports delayed autosuspend (see Section 9); it may be modified only by the pm_runtime{_dont}_use_autosuspend() helper functions

unsigned int timer_autosuspends;
- indicates that the PM core should attempt to carry out an autosuspend when the timer expires rather than a normal suspend

int autosuspend_delay;
- the delay time (in milliseconds) to be used for autosuspend

unsigned long last_busy;
- the time (in jiffies) when the pm_runtime_mark_last_busy() helper function was last called for this device; used in calculating inactivity periods for autosuspend

All of the above fields are members of the 'power' member of 'struct device'.

4. Runtime PM Device Helper Functions

The following runtime PM helper functions are defined in drivers/base/power/runtime.c and include/linux/pm_runtime.h:

*void pm_runtime_init(struct device *dev);*

- initialize the device runtime PM fields in 'struct dev_pm_info'

*void pm_runtime_remove(struct device *dev);*

- make sure that the runtime PM of the device will be disabled after removing the device from device hierarchy

*int pm_runtime_idle(struct device *dev);*

- execute the subsystem-level idle callback for the device; returns an error code on failure, where -EINPROGRESS means that ->runtime_idle() is already being executed; if there is no callback or the callback returns 0 then run pm_runtime_autosuspend(dev) and return its result

*int pm_runtime_suspend(struct device *dev);*

- execute the subsystem-level suspend callback for the device; returns 0 on success, 1 if the device's runtime PM status was already 'suspended', or error code on failure, where -EAGAIN or -EBUSY means it is safe to attempt to suspend the device again in future and -EACCES means that 'power.disable_depth' is different from 0

*int pm_runtime_autosuspend(struct device *dev);*

- same as pm_runtime_suspend() except that the autosuspend delay is taken *into account;* if pm_runtime_autosuspend_expiration() says the delay has not yet expired then an autosuspend is scheduled for the appropriate time and 0 is returned

*int pm_runtime_resume(struct device *dev);*

- execute the subsystem-level resume callback for the device; returns 0 on success, 1 if the device's runtime PM status is already 'active' (also if 'power.disable_depth' is nonzero, but the status was 'active' when it was changing from 0 to 1) or error code on failure, where -EAGAIN means it may be safe to attempt to resume the device again in future, but 'power.runtime_error' should be checked additionally, and -EACCES means that the callback could not be run, because 'power.disable_depth' was different from 0

*int pm_runtime_resume_and_get(struct device *dev);*

- run pm_runtime_resume(dev) and if successful, increment the device's usage counter; return the result of pm_runtime_resume

*int pm_request_idle(struct device *dev);*

- submit a request to execute the subsystem-level idle callback for the device (the request is represented by a work item in pm_wq); returns 0 on success or error code if the request has not been queued up

*int pm_request_autosuspend(struct device *dev);*

- schedule the execution of the subsystem-level suspend callback for the device when the autosuspend delay has expired; if the delay has already expired then the work item is queued up immediately

*int pm_schedule_suspend(struct device *dev, unsigned int delay);*

- schedule the execution of the subsystem-level suspend callback for the device in future, where 'delay' is the time to wait before queuing up a suspend work item in pm_wq, in milliseconds (if 'delay' is zero, the work item is queued up immediately); returns 0 on success, 1 if the device's PM runtime status was already 'suspended', or error code if the request hasn't been scheduled (or queued up if 'delay' is 0); if the execution of ->runtime_suspend() is already scheduled and not yet expired, the new value of 'delay' will be used as the time to wait

*int pm_request_resume(struct device *dev);*

- submit a request to execute the subsystem-level resume callback for the device (the request is represented by a work item in pm_wq); returns 0 on success, 1 if the device's runtime PM status was already 'active', or error code if the request hasn't been queued up

*void pm_runtime_get_noresume(struct device *dev);*

- increment the device's usage counter

*int pm_runtime_get(struct device *dev);*

- increment the device's usage counter, run pm_request_resume(dev) and return its result

*int pm_runtime_get_sync(struct device *dev);*

- increment the device's usage counter, run pm_runtime_resume(dev) and return its result; note that it does not drop the device's usage counter on errors, so consider using pm_runtime_resume_and_get() instead of it, especially if its return value is checked by the caller, as this is likely to result in cleaner code.

*int pm_runtime_get_if_in_use(struct device *dev);*

- return -EINVAL if 'power.disable_depth' is nonzero; otherwise, if the runtime PM status is RPM_ACTIVE and the runtime PM usage counter is nonzero, increment the counter and return 1; otherwise return 0 without changing the counter

*int pm_runtime_get_if_active(struct device *dev, bool ign_usage_count);*

- return -EINVAL if 'power.disable_depth' is nonzero; otherwise, if the runtime PM status is RPM_ACTIVE, and either ign_usage_count is true or the device's usage_count is non-zero, increment the counter and return 1; otherwise return 0 without changing the counter

*void pm_runtime_put_noidle(struct device *dev);*

- decrement the device's usage counter

*int pm_runtime_put(struct device *dev);*

- decrement the device's usage counter; if the result is 0 then run pm_request_idle(dev) and return its result

*int pm_runtime_put_autosuspend(struct device *dev);*

- decrement the device's usage counter; if the result is 0 then run pm_request_autosuspend(dev) and return its

result

*int pm_runtime_put_sync(struct device *dev);*

- decrement the device's usage counter; if the result is 0 then run pm_runtime_idle(dev) and return its result

*int pm_runtime_put_sync_suspend(struct device *dev);*

- decrement the device's usage counter; if the result is 0 then run pm_runtime_suspend(dev) and return its result

*int pm_runtime_put_sync_autosuspend(struct device *dev);*

- decrement the device's usage counter; if the result is 0 then run pm_runtime_autosuspend(dev) and return its result

*void pm_runtime_enable(struct device *dev);*

- decrement the device's 'power.disable_depth' field; if that field is equal to zero, the runtime PM helper functions can execute subsystem-level callbacks described in Section 2 for the device

*int pm_runtime_disable(struct device *dev);*

- increment the device's 'power.disable_depth' field (if the value of that field was previously zero, this prevents subsystem-level runtime PM callbacks from being run for the device), make sure that all of the pending runtime PM operations on the device are either completed or canceled; returns 1 if there was a resume request pending and it was necessary to execute the subsystem-level resume callback for the device to satisfy that request, otherwise 0 is returned

*int pm_runtime_barrier(struct device *dev);*

- check if there's a resume request pending for the device and resume it (synchronously) in that case, cancel any other pending runtime PM requests regarding it and wait for all runtime PM operations on it in progress to complete; returns 1 if there was a resume request pending and it was necessary to execute the subsystem-level resume callback for the device to satisfy that request, otherwise 0 is returned

*void pm_suspend_ignore_children(struct device *dev, bool enable);*

- set/unset the power.ignore_children flag of the device

*int pm_runtime_set_active(struct device *dev);*

- clear the device's 'power.runtime_error' flag, set the device's runtime PM status to 'active' and update its parent's counter of 'active' children as appropriate (it is only valid to use this function if 'power.runtime_error' is set or 'power.disable_depth' is greater than zero); it will fail and return error code if the device has a parent which is not active and the 'power.ignore_children' flag of which is unset

*void pm_runtime_set_suspended(struct device *dev);*

- clear the device's 'power.runtime_error' flag, set the device's runtime PM status to 'suspended' and update its parent's counter of 'active' children as appropriate (it is only valid to use this function if 'power.runtime_error' is set or 'power.disable_depth' is greater than zero)

*bool pm_runtime_active(struct device *dev);*

- return true if the device's runtime PM status is 'active' or its 'power.disable_depth' field is not equal to zero, or false otherwise

*bool pm_runtime_suspended(struct device *dev);*

- return true if the device's runtime PM status is 'suspended' and its 'power.disable_depth' field is equal to zero, or false otherwise

*bool pm_runtime_status_suspended(struct device *dev);*

- return true if the device's runtime PM status is 'suspended'

*void pm_runtime_allow(struct device *dev);*

- set the power.runtime_auto flag for the device and decrease its usage counter (used by the /sys/devices/.../power/control interface to effectively allow the device to be power managed at run time)

*void pm_runtime_forbid(struct device *dev);*

- unset the power.runtime_auto flag for the device and increase its usage counter (used by the /sys/devices/.../power/control interface to effectively prevent the device from being power managed at run time)

*void pm_runtime_no_callbacks(struct device *dev);*

- set the power.no_callbacks flag for the device and remove the runtime PM attributes from /sys/devices/.../power (or prevent them from being added when the device is registered)

*void pm_runtime_irq_safe(struct device *dev);*

- set the power.irq_safe flag for the device, causing the runtime-PM callbacks to be invoked with interrupts off

*bool pm_runtime_is_irq_safe(struct device *dev);*

- return true if power.irq_safe flag was set for the device, causing the runtime-PM callbacks to be invoked with interrupts off

*void pm_runtime_mark_last_busy(struct device *dev);*

- set the power.last_busy field to the current time

*void pm_runtime_use_autosuspend(struct device *dev);*

- set the power.use_autosuspend flag, enabling autosuspend delays; call pm_runtime_get_sync if the flag was previously cleared and power.autosuspend_delay is negative

*void pm_runtime_dont_use_autosuspend(struct device *dev);*

- clear the power.use_autosuspend flag, disabling autosuspend delays; decrement the device's usage counter if the flag was previously set and power.autosuspend_delay is negative; call pm_runtime_idle

*void pm_runtime_set_autosuspend_delay(struct device *dev, int delay);*

- set the power.autosuspend_delay value to 'delay' (expressed in milliseconds); if 'delay' is negative then runtime suspends are prevented; if power.use_autosuspend is set, pm_runtime_get_sync may be called or the device's usage counter may be decremented and pm_runtime_idle called depending on if power.autosuspend_delay is changed to or from a negative value; if power.use_autosuspend is clear, pm_runtime_idle is called

*unsigned long pm_runtime_autosuspend_expiration(struct device *dev);*

- calculate the time when the current autosuspend delay period will expire, based on power.last_busy and power.autosuspend_delay; if the delay time is 1000 ms or larger then the expiration time is rounded up to the nearest second; returns 0 if the delay period has already expired or power.use_autosuspend isn't set, otherwise returns the expiration time in jiffies

It is safe to execute the following helper functions from interrupt context:

- pm_request_idle()
- pm_request_autosuspend()
- pm_schedule_suspend()
- pm_request_resume()
- pm_runtime_get_noresume()
- pm_runtime_get()
- pm_runtime_put_noidle()
- pm_runtime_put()
- pm_runtime_put_autosuspend()
- pm_runtime_enable()
- pm_suspend_ignore_children()
- pm_runtime_set_active()
- pm_runtime_set_suspended()
- pm_runtime_suspended()
- pm_runtime_mark_last_busy()
- pm_runtime_autosuspend_expiration()

If pm_runtime_irq_safe() has been called for a device then the following helper functions may also be used in interrupt context:

- pm_runtime_idle()
- pm_runtime_suspend()
- pm_runtime_autosuspend()
- pm_runtime_resume()
- pm_runtime_get_sync()
- pm_runtime_put_sync()
- pm_runtime_put_sync_suspend()
- pm_runtime_put_sync_autosuspend()

5. Runtime PM Initialization, Device Probing and Removal

Initially, the runtime PM is disabled for all devices, which means that the majority of the runtime PM helper functions described in Section 4 will return -EAGAIN until pm_runtime_enable() is called for the device.

In addition to that, the initial runtime PM status of all devices is 'suspended', but it need not reflect the actual physical state of the device. Thus, if the device is initially active (i.e. it is able to process I/O), its runtime PM status must be changed to 'active', with the help of pm_runtime_set_active(), before pm_runtime_enable() is called for the device.

However, if the device has a parent and the parent's runtime PM is enabled, calling pm_runtime_set_active() for the device will affect the parent, unless the parent's 'power.ignore_children' flag is set. Namely, in that case the parent won't be able to suspend at run time, using the PM core's helper functions, as long as the child's status is 'active', even if the child's runtime PM is still disabled (i.e. pm_runtime_enable() hasn't been called for the child yet or pm_runtime_disable() has been called for it). For this reason, once pm_runtime_set_active() has been called for the device, pm_runtime_enable() should be called for it too as soon as reasonably possible or its runtime PM status should be changed back to 'suspended' with the help of pm_runtime_set_suspended().

If the default initial runtime PM status of the device (i.e. 'suspended') reflects the actual state of the device, its bus type's or its driver's ->probe() callback will likely need to wake it up using one of the PM core's helper functions described in Section 4. In that case, pm_runtime_resume() should be used. Of course, for this purpose the device's runtime PM has to be enabled earlier by calling pm_runtime_enable().

Note, if the device may execute pm_runtime calls during the probe (such as if it is registered with a subsystem that may call back in) then the pm_runtime_get_sync() call paired with a pm_runtime_put() call will be appropriate to ensure that the device is not put back to sleep during the probe. This can happen with systems such as the network device layer.

It may be desirable to suspend the device once ->probe() has finished. Therefore the driver core uses the asynchronous pm_request_idle() to submit a request to execute the subsystem-level idle callback for the device at that time. A driver that makes use of the runtime autosuspend feature may want to update the last busy mark before returning from ->probe().

Moreover, the driver core prevents runtime PM callbacks from racing with the bus notifier callback in __device_release_driver(),

which is necessary because the notifier is used by some subsystems to carry out operations affecting the runtime PM functionality. It does so by calling pm_runtime_get_sync() before driver_sysfs_remove() and the BUS_NOTIFY_UNBIND_DRIVER notifications. This resumes the device if it's in the suspended state and prevents it from being suspended again while those routines are being executed.

To allow bus types and drivers to put devices into the suspended state by calling pm_runtime_suspend() from their ->remove() routines, the driver core executes pm_runtime_put_sync() after running the BUS_NOTIFY_UNBIND_DRIVER notifications in __device_release_driver(). This requires bus types and drivers to make their ->remove() callbacks avoid races with runtime PM directly, but it also allows more flexibility in the handling of devices during the removal of their drivers.

Drivers in ->remove() callback should undo the runtime PM changes done in ->probe(). Usually this means calling pm_runtime_disable(), pm_runtime_dont_use_autosuspend() etc.

The user space can effectively disallow the driver of the device to power manage it at run time by changing the value of its /sys/devices/.../power/control attribute to "on", which causes pm_runtime_forbid() to be called. In principle, this mechanism may also be used by the driver to effectively turn off the runtime power management of the device until the user space turns it on. Namely, during the initialization the driver can make sure that the runtime PM status of the device is 'active' and call pm_runtime_forbid(). It should be noted, however, that if the user space has already intentionally changed the value of /sys/devices/.../power/control to "auto" to allow the driver to power manage the device at run time, the driver may confuse it by using pm_runtime_forbid() this way.

6. Runtime PM and System Sleep

Runtime PM and system sleep (i.e., system suspend and hibernation, also known as suspend-to-RAM and suspend-to-disk) interact with each other in a couple of ways. If a device is active when a system sleep starts, everything is straightforward. But what should happen if the device is already suspended?

The device may have different wake-up settings for runtime PM and system sleep. For example, remote wake-up may be enabled for runtime suspend but disallowed for system sleep (device_may_wakeup(dev) returns 'false'). When this happens, the subsystem-level system suspend callback is responsible for changing the device's wake-up setting (it may leave that to the device driver's system suspend routine). It may be necessary to resume the device and suspend it again in order to do so. The same is true if the driver uses different power levels or other settings for runtime suspend and system sleep.

During system resume, the simplest approach is to bring all devices back to full power, even if they had been suspended before the system suspend began. There are several reasons for this, including:

- The device might need to switch power levels, wake-up settings, etc.
- Remote wake-up events might have been lost by the firmware.
- The device's children may need the device to be at full power in order to resume themselves.
- The driver's idea of the device state may not agree with the device's physical state. This can happen during resume from hibernation.
- The device might need to be reset.
- Even though the device was suspended, if its usage counter was > 0 then most likely it would need a runtime resume in the near future anyway.

If the device had been suspended before the system suspend began and it's brought back to full power during resume, then its runtime PM status will have to be updated to reflect the actual post-system sleep status. The way to do this is:

- pm_runtime_disable(dev);
- pm_runtime_set_active(dev);
- pm_runtime_enable(dev);

The PM core always increments the runtime usage counter before calling the ->suspend() callback and decrements it after calling the ->resume() callback. Hence disabling runtime PM temporarily like this will not cause any runtime suspend attempts to be permanently lost. If the usage count goes to zero following the return of the ->resume() callback, the ->runtime_idle() callback will be invoked as usual.

On some systems, however, system sleep is not entered through a global firmware or hardware operation. Instead, all hardware components are put into low-power states directly by the kernel in a coordinated way. Then, the system sleep state effectively follows from the states the hardware components end up in and the system is woken up from that state by a hardware interrupt or a similar mechanism entirely under the kernel's control. As a result, the kernel never gives control away and the states of all devices during resume are precisely known to it. If that is the case and none of the situations listed above takes place (in particular, if the system is not waking up from hibernation), it may be more efficient to leave the devices that had been suspended before the system suspend began in the suspended state.

To this end, the PM core provides a mechanism allowing some coordination between different levels of device hierarchy. Namely, if a system suspend .prepare() callback returns a positive number for a device, that indicates to the PM core that the device appears to be runtime-suspended and its state is fine, so it may be left in runtime suspend provided that all of its descendants are also left in runtime suspend. If that happens, the PM core will not execute any system suspend and resume callbacks for all of those devices, except for the .complete() callback, which is then entirely responsible for handling the device as appropriate. This only applies to system suspend transitions that are not related to hibernation (see Documentation/driver-api/pm/devices.rst for more information).

The PM core does its best to reduce the probability of race conditions between the runtime PM and system suspend/resume (and hibernation) callbacks by carrying out the following operations:

- During system suspend pm_runtime_get_noresume() is called for every device right before executing the subsystem-level .prepare() callback for it and pm_runtime_barrier() is called for every device right before executing the subsystem-level .suspend() callback for it. In addition to that the PM core calls __pm_runtime_disable() with 'false' as the second argument for every device right before executing the subsystem-level .suspend_late() callback for it.
- During system resume pm_runtime_enable() and pm_runtime_put() are called for every device right after executing the subsystem-level .resume_early() callback and right after executing the subsystem-level .complete() callback for it, respectively.

7. Generic subsystem callbacks

Subsystems may wish to conserve code space by using the set of generic power management callbacks provided by the PM core, defined in driver/base/power/generic_ops.c:

*int pm_generic_runtime_suspend(struct device *dev);*
- invoke the ->runtime_suspend() callback provided by the driver of this device and return its result, or return 0 if not defined

*int pm_generic_runtime_resume(struct device *dev);*
- invoke the ->runtime_resume() callback provided by the driver of this device and return its result, or return 0 if not defined

*int pm_generic_suspend(struct device *dev);*
- if the device has not been suspended at run time, invoke the ->suspend() callback provided by its driver and return its result, or return 0 if not defined

*int pm_generic_suspend_noirq(struct device *dev);*
- if pm_runtime_suspended(dev) returns "false", invoke the ->suspend_noirq() callback provided by the device's driver and return its result, or return 0 if not defined

*int pm_generic_resume(struct device *dev);*
- invoke the ->resume() callback provided by the driver of this device and, if successful, change the device's runtime PM status to 'active'

*int pm_generic_resume_noirq(struct device *dev);*
- invoke the ->resume_noirq() callback provided by the driver of this device

*int pm_generic_freeze(struct device *dev);*
- if the device has not been suspended at run time, invoke the ->freeze() callback provided by its driver and return its result, or return 0 if not defined

*int pm_generic_freeze_noirq(struct device *dev);*
- if pm_runtime_suspended(dev) returns "false", invoke the ->freeze_noirq() callback provided by the device's driver and return its result, or return 0 if not defined

*int pm_generic_thaw(struct device *dev);*
- if the device has not been suspended at run time, invoke the ->thaw() callback provided by its driver and return its result, or return 0 if not defined

*int pm_generic_thaw_noirq(struct device *dev);*
- if pm_runtime_suspended(dev) returns "false", invoke the ->thaw_noirq() callback provided by the device's driver and return its result, or return 0 if not defined

*int pm_generic_poweroff(struct device *dev);*
- if the device has not been suspended at run time, invoke the ->poweroff() callback provided by its driver and return its result, or return 0 if not defined

*int pm_generic_poweroff_noirq(struct device *dev);*
- if pm_runtime_suspended(dev) returns "false", run the ->poweroff_noirq() callback provided by the device's driver and return its result, or return 0 if not defined

*int pm_generic_restore(struct device *dev);*
- invoke the ->restore() callback provided by the driver of this device and, if successful, change the device's runtime PM status to 'active'

*int pm_generic_restore_noirq(struct device *dev);*
- invoke the ->restore_noirq() callback provided by the device's driver

These functions are the defaults used by the PM core if a subsystem doesn't provide its own callbacks for ->runtime_idle(), ->runtime_suspend(), ->runtime_resume(), ->suspend(), ->suspend_noirq(), ->resume(), ->resume_noirq(), ->freeze(), ->freeze_noirq(), ->thaw(), ->thaw_noirq(), ->poweroff(), ->poweroff_noirq(), ->restore(), ->restore_noirq() in the subsystem-level dev_pm_ops structure.

Device drivers that wish to use the same function as a system suspend, freeze, poweroff and runtime suspend callback, and similarly for system resume, thaw, restore, and runtime resume, can achieve this with the help of the UNIVERSAL_DEV_PM_OPS macro defined in include/linux/pm.h (possibly setting its last argument to NULL).

8. "No-Callback" Devices

Some "devices" are only logical sub-devices of their parent and cannot be power-managed on their own. (The prototype example is a USB interface. Entire USB devices can go into low-power mode or send wake-up requests, but neither is possible for individual interfaces.) The drivers for these devices have no need of runtime PM callbacks; if the callbacks did exist, ->runtime_suspend() and ->runtime_resume() would always return 0 without doing anything else and ->runtime_idle() would always call pm_runtime_suspend().

Subsystems can tell the PM core about these devices by calling pm_runtime_no_callbacks(). This should be done after the device structure is initialized and before it is registered (although after device registration is also okay). The routine will set the device's power.no_callbacks flag and prevent the non-debugging runtime PM sysfs attributes from being created.

When power.no_callbacks is set, the PM core will not invoke the ->runtime_idle(), ->runtime_suspend(), or ->runtime_resume() callbacks. Instead it will assume that suspends and resumes always succeed and that idle devices should be suspended.

As a consequence, the PM core will never directly inform the device's subsystem or driver about runtime power changes. Instead, the driver for the device's parent must take responsibility for telling the device's driver when the parent's power state changes.

Note that, in some cases it may not be desirable for subsystems/drivers to call pm_runtime_no_callbacks() for their devices. This could be because a subset of the runtime PM callbacks needs to be implemented, a platform dependent PM domain could get attached to the device or that the device is power managed through a supplier device link. For these reasons and to avoid boilerplate code in subsystems/drivers, the PM core allows runtime PM callbacks to be unassigned. More precisely, if a callback pointer is NULL, the PM core will act as though there was a callback and it returned 0.

9. Autosuspend, or automatically-delayed suspends

Changing a device's power state isn't free; it requires both time and energy. A device should be put in a low-power state only when there's some reason to think it will remain in that state for a substantial time. A common heuristic says that a device which hasn't been used for a while is liable to remain unused; following this advice, drivers should not allow devices to be suspended at runtime until they have been inactive for some minimum period. Even when the heuristic ends up being non-optimal, it will still prevent devices from "bouncing" too rapidly between low-power and full-power states.

The term "autosuspend" is an historical remnant. It doesn't mean that the device is automatically suspended (the subsystem or driver still has to call the appropriate PM routines); rather it means that runtime suspends will automatically be delayed until the desired period of inactivity has elapsed.

Inactivity is determined based on the power.last_busy field. Drivers should call pm_runtime_mark_last_busy() to update this field after carrying out I/O, typically just before calling pm_runtime_put_autosuspend(). The desired length of the inactivity period is a matter of policy. Subsystems can set this length initially by calling pm_runtime_set_autosuspend_delay(), but after device registration the length should be controlled by user space, using the /sys/devices/.../power/autosuspend_delay_ms attribute.

In order to use autosuspend, subsystems or drivers must call pm_runtime_use_autosuspend() (preferably before registering the device), and thereafter they should use the various *_autosuspend()* helper functions instead of the non-autosuspend counterparts:

```
Instead of: pm_runtime_suspend    use: pm_runtime_autosuspend;
Instead of: pm_schedule_suspend   use: pm_request_autosuspend;
Instead of: pm_runtime_put        use: pm_runtime_put_autosuspend;
Instead of: pm_runtime_put_sync   use: pm_runtime_put_sync_autosuspend.
```

Drivers may also continue to use the non-autosuspend helper functions; they will behave normally, which means sometimes taking the autosuspend delay into account (see pm_runtime_idle).

Under some circumstances a driver or subsystem may want to prevent a device from autosuspending immediately, even though the usage counter is zero and the autosuspend delay time has expired. If the ->runtime_suspend() callback returns -EAGAIN or -EBUSY, and if the next autosuspend delay expiration time is in the future (as it normally would be if the callback invoked pm_runtime_mark_last_busy()), the PM core will automatically reschedule the autosuspend. The ->runtime_suspend() callback can't do this rescheduling itself because no suspend requests of any kind are accepted while the device is suspending (i.e., while the callback is running).

The implementation is well suited for asynchronous use in interrupt contexts. However such use inevitably involves races, because the PM core can't synchronize ->runtime_suspend() callbacks with the arrival of I/O requests. This synchronization must be handled by the driver, using its private lock. Here is a schematic pseudo-code example:

```
foo_read_or_write(struct foo_priv *foo, void *data)
{
        lock(&foo->private_lock);
        add_request_to_io_queue(foo, data);
        if (foo->num_pending_requests++ == 0)
                pm_runtime_get(&foo->dev);
        if (!foo->is_suspended)
                foo_process_next_request(foo);
        unlock(&foo->private_lock);
}

foo_io_completion(struct foo_priv *foo, void *req)
{
        lock(&foo->private_lock);
```

```
        if (--foo->num_pending_requests == 0) {
                pm_runtime_mark_last_busy(&foo->dev);
                pm_runtime_put_autosuspend(&foo->dev);
        } else {
                foo_process_next_request(foo);
        }
        unlock(&foo->private_lock);
        /* Send req result back to the user ... */
}

int foo_runtime_suspend(struct device *dev)
{
        struct foo_priv foo = container_of(dev, ...);
        int ret = 0;

        lock(&foo->private_lock);
        if (foo->num_pending_requests > 0) {
                ret = -EBUSY;
        } else {
                /* ... suspend the device ... */
                foo->is_suspended = 1;
        }
        unlock(&foo->private_lock);
        return ret;
}

int foo_runtime_resume(struct device *dev)
{
        struct foo_priv foo = container_of(dev, ...);

        lock(&foo->private_lock);
        /* ... resume the device ... */
        foo->is_suspended = 0;
        pm_runtime_mark_last_busy(&foo->dev);
        if (foo->num_pending_requests > 0)
                foo_process_next_request(foo);
        unlock(&foo->private_lock);
        return 0;
}
```

The important point is that after foo_io_completion() asks for an autosuspend, the foo_runtime_suspend() callback may race with foo_read_or_write(). Therefore foo_runtime_suspend() has to check whether there are any pending I/O requests (while holding the private lock) before allowing the suspend to proceed.

In addition, the power.autosuspend_delay field can be changed by user space at any time. If a driver cares about this, it can call pm_runtime_autosuspend_expiration() from within the ->runtime_suspend() callback while holding its private lock. If the function returns a nonzero value then the delay has not yet expired and the callback should return -EAGAIN.