Run-time Power Management Framework for I/O Devices

1. Introduction

Support for run-time power management (run-time 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 run-time 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 run-time 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 run-time PM operations with one another.

* Three device run-time 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 run-time 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 run-time PM callbacks present in 'struct dev_pm_ops', the device run-time PM fields of 'struct dev_pm_info' and the core helper functions provided for run-time PM are described below.

2. Device Run-time PM Callbacks

There are three device run-time 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 either the bus type, or device type (if the bus type's callback is not defined), or device class (if the bus type's and device type's callbacks are not defined) of given device. The bus type, device type and device class callbacks are referred to as subsystem-level callbacks in what follows.

The subsystem-level suspend callback 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 has completed successfully
    for given device, the PM core regards the device as suspended, which need
    not mean that the device 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 subsystem-level resume
    callback is executed for it. The run-time PM status of a device after
    successful execution of the subsystem-level suspend callback is 'suspended'.

  * If the subsystem-level suspend callback returns -EBUSY or -EAGAIN,
    the device's run-time PM status is 'active', which means that the device
    must be fully operational afterwards.

  * If the subsystem-level suspend callback returns an error code different
    from -EBUSY or -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 the status of it is directly set either to 'active', or to 'suspended'
    (the PM core provides special helper functions for this purpose).

In particular, if the driver requires remote wake-up 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_run_wake() returns 'false' for the
device, then ->runtime_suspend() should return -EBUSY. On the other hand, if
device_run_wake() returns 'true' for the device and the device is put into a low
power state during the execution of the subsystem-level suspend callback, it is
expected that remote wake-up will be enabled for the device. Generally, remote
wake-up should be enabled for all input devices put into a low power state at
run time.

The subsystem-level resume callback 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 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 run-time PM status
    of the device is then 'active'.

  * If the subsystem-level 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
    either to 'active' or to 'suspended' (the PM core provides special helper
    functions for this purpose).

The subsystem-level idle callback 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
    subsystem-level idle callback with the device as an argument.

The action performed by a subsystem-level idle callback is totally dependent on
the subsystem 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. The value returned by this callback is 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 the bus type's run-time
PM callbacks:

(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 run-time 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 run-time
    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.

3. Run-time PM Device Fields

The following device run-time 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 request
```

```
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 synchronisation

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 funcions (they work normally if this is
    equal to zero); the initial value of it is 1 (i.e. run-time PM is
    initially disabled for all devices)

unsigned 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 funtions 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"

unsigned int run_wake;
  - set if the device is capable of generating run-time wake-up events

enum rpm_status runtime_status;
  - the run-time 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

  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
```

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

4. Run-time PM Device Helper Functions

The following run-time 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 run-time PM fields in 'struct dev_pm_info'

  void pm_runtime_remove(struct device *dev);
    - make sure that the run-time 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 0 on
      success or error code on failure, where -EINPROGRESS means that
      ->runtime_idle() is already being executed

  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 run-time 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

  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 run-time PM status was already 'active' 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

  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_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
      run-time 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 run-time 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

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, run pm_request_idle(dev) and return
    its result

int pm_runtime_put_sync(struct device *dev);
  - decrement the device's usage counter, run pm_runtime_idle(dev) and return
    its result

void pm_runtime_enable(struct device *dev);
  - enable the run-time PM helper functions to run the device bus type's
    run-time PM callbacks described in Section 2

int pm_runtime_disable(struct device *dev);
  - prevent the run-time PM helper functions from running subsystem-level
    run-time PM callbacks for the device, make sure that all of the pending
    run-time 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

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 run-time
    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 run-time
    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_suspended(struct device *dev);
  - return true if the device's runtime PM status is 'suspended', or false
    otherwise

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)
```

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

```
pm_request_idle()
pm_schedule_suspend()
pm_request_resume()
pm_runtime_get_noresume()
pm_runtime_get()
pm_runtime_put_noidle()
pm_runtime_put()
pm_suspend_ignore_children()
pm_runtime_set_active()
pm_runtime_set_suspended()
pm_runtime_enable()
```

5. Run-time PM Initialization, Device Probing and Removal

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

In addition to that, the initial run-time 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 run-time 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 run-time 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 run-time 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 run-time PM status should be changed back to 'suspended' with the help of pm_runtime_set_suspended().

If the default initial run-time 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 run-time PM has to be enabled earlier by calling pm_runtime_enable().

If the device bus type's or driver's ->probe() or ->remove() callback runs pm_runtime_suspend() or pm_runtime_idle() or their asynchronous counterparts, they will fail returning -EAGAIN, because the device's usage counter is incremented by the core before executing ->probe() and ->remove(). Still, it may be desirable to suspend the device as soon as ->probe() or ->remove() has finished, so the PM core uses pm_runtime_idle_sync() to invoke the subsystem-level idle callback for the device at that time.

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 run-time power management of the device until the user space turns it on. Namely, during the initialization the driver can make sure that the run-time 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. Run-time PM and System Sleep

Run-time 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 run-time PM and system sleep. For example, remote wake-up may be enabled for run-time 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 run-time suspend and system sleep.

During system resume, devices generally should be brought back to full power, even if they were suspended before the system sleep 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 run-time resume in the near future anyway.

  * Always going back to full power is simplest.

If the device was suspended before the sleep began, then its run-time 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 run-time usage counter before calling the
->prepare() callback and decrements it after calling the ->complete() callback.
Hence disabling run-time PM temporarily like this will not cause any run-time
suspend callbacks to be lost.

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_idle(struct device *dev);
    - invoke the ->runtime_idle() callback provided by the driver of this
      device, if defined, and call pm_runtime_suspend() for this device if the
      return value is 0 or the callback is not defined

  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 -EINVAL 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 -EINVAL 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_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_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_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_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_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'
```

These functions can be assigned to the ->runtime_idle(), ->runtime_suspend(), ->runtime_resume(), ->suspend(), ->resume(), ->freeze(), ->thaw(), ->poweroff(), or ->restore() callback pointers in the subsystem-level dev_pm_ops structures.

If a subsystem wishes to use all of them at the same time, it can simply assign the GENERIC_SUBSYS_PM_OPS macro, defined in include/linux/pm.h, to its dev_pm_ops structure pointer.

Device drivers that wish to use the same function as a system suspend, freeze, poweroff and run-time suspend callback, and similarly for system resume, thaw, restore, and run-time 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).