Linux Power Documentation

Release 6.8.0

The kernel development community

Jan 16, 2026

CONTENTS

APM OR ACPI?

If you have a relatively recent x86 mobile, desktop, or server system, odds are it supports either Advanced Power Management (APM) or Advanced Configuration and Power Interface (ACPI). ACPI is the newer of the two technologies and puts power management in the hands of the operating system, allowing for more intelligent power management than is possible with BIOS controlled APM.

The best way to determine which, if either, your system supports is to build a kernel with both ACPI and APM enabled (as of 2.3.x ACPI is enabled by default). If a working ACPI implementation is found, the ACPI driver will override and disable APM, otherwise the APM driver will be used.

No, sorry, you cannot have both ACPI and APM enabled and running at once. Some people with broken ACPI or broken APM implementations would like to use both to get a full set of working features, but you simply cannot mix and match the two. Only one power management interface can be in control of the machine at once. Think about it..

1.1 User-space Daemons

Both APM and ACPI rely on user-space daemons, apmd and acpid respectively, to be completely functional. Obtain both of these daemons from your Linux distribution or from the Internet (see below) and be sure that they are started sometime in the system boot process. Go ahead and start both. If ACPI or APM is not available on your system the associated daemon will exit gracefully.

apmd	http://ftp.debian.org/pool/main/a/apmd/
acpid	http://acpid.sf.net/

DEBUGGING HIBERNATION AND SUSPEND

(C) 2007 Rafael J. Wysocki <rjw@sisk.pl>, GPL

2.1 1. Testing hibernation (aka suspend to disk or STD)

To check if hibernation works, you can try to hibernate in the "reboot" mode:

```
# echo reboot > /sys/power/disk
# echo disk > /sys/power/state
```

and the system should create a hibernation image, reboot, resume and get back to the command prompt where you have started the transition. If that happens, hibernation is most likely to work correctly. Still, you need to repeat the test at least a couple of times in a row for confidence. [This is necessary, because some problems only show up on a second attempt at suspending and resuming the system.] Moreover, hibernating in the "reboot" and "shutdown" modes causes the PM core to skip some platform-related callbacks which on ACPI systems might be necessary to make hibernation work. Thus, if your machine fails to hibernate or resume in the "reboot" mode, you should try the "platform" mode:

```
# echo platform > /sys/power/disk
# echo disk > /sys/power/state
```

which is the default and recommended mode of hibernation.

Unfortunately, the "platform" mode of hibernation does not work on some systems with broken BIOSes. In such cases the "shutdown" mode of hibernation might work:

```
# echo shutdown > /sys/power/disk
# echo disk > /sys/power/state
```

(it is similar to the "reboot" mode, but it requires you to press the power button to make the system resume).

If neither "platform" nor "shutdown" hibernation mode works, you will need to identify what goes wrong.

2.1.1 a) Test modes of hibernation

To find out why hibernation fails on your system, you can use a special testing facility available if the kernel is compiled with CONFIG_PM_DEBUG set. Then, there is the file /sys/power/pm_test that can be used to make the hibernation core run in a test mode. There are 5 test modes available:

freezer

> • test the freezing of processes

devices

> • test the freezing of processes and suspending of devices

platform

> • test the freezing of processes, suspending of devices and platform global control methods[1]

processors

> • test the freezing of processes, suspending of devices, platform global control methods[1] and the disabling of nonboot CPUs

core

> • test the freezing of processes, suspending of devices, platform global control methods[1], the disabling of nonboot CPUs and suspending of platform/system devices

To use one of them it is necessary to write the corresponding string to /sys/power/pm_test (eg. "devices" to test the freezing of processes and suspending devices) and issue the standard hibernation commands. For example, to use the "devices" test mode along with the "platform" mode of hibernation, you should do the following:

```
# echo devices > /sys/power/pm_test
# echo platform > /sys/power/disk
# echo disk > /sys/power/state
```

Then, the kernel will try to freeze processes, suspend devices, wait a few seconds (5 by default, but configurable by the suspend.pm_test_delay module parameter), resume devices and thaw processes. If "platform" is written to /sys/power/pm_test , then after suspending devices the kernel will additionally invoke the global control methods (eg. ACPI global control methods) used to prepare the platform firmware for hibernation. Next, it will wait a configurable number of seconds and invoke the platform (eg. ACPI) global methods used to cancel hibernation etc.

Writing "none" to /sys/power/pm_test causes the kernel to switch to the normal hibernation/suspend operations. Also, when open for reading, /sys/power/pm_test contains a space-separated list of all available tests (including "none" that represents the normal functionality) in which the current test level is indicated by square brackets.

Generally, as you can see, each test level is more "invasive" than the previous one and the "core" level tests the hardware and drivers as deeply as possible without creating a hibernation image. Obviously, if the "devices" test fails, the "platform" test will fail as well and so on. Thus, as a rule of thumb, you should try the test modes starting from "freezer", through "devices", "platform"

---

[1] the platform global control methods are only available on ACPI systems and are only tested if the hibernation mode is set to "platform"

and "processors" up to "core" (repeat the test on each level a couple of times to make sure that any random factors are avoided).

If the "freezer" test fails, there is a task that cannot be frozen (in that case it usually is possible to identify the offending task by analysing the output of dmesg obtained after the failing test). Failure at this level usually means that there is a problem with the tasks freezer subsystem that should be reported.

If the "devices" test fails, most likely there is a driver that cannot suspend or resume its device (in the latter case the system may hang or become unstable after the test, so please take that into consideration). To find this driver, you can carry out a binary search according to the rules:

- if the test fails, unload a half of the drivers currently loaded and repeat (that would probably involve rebooting the system, so always note what drivers have been loaded before the test),

- if the test succeeds, load a half of the drivers you have unloaded most recently and repeat.

Once you have found the failing driver (there can be more than just one of them), you have to unload it every time before hibernation. In that case please make sure to report the problem with the driver.

It is also possible that the "devices" test will still fail after you have unloaded all modules. In that case, you may want to look in your kernel configuration for the drivers that can be compiled as modules (and test again with these drivers compiled as modules). You may also try to use some special kernel command line options such as "noapic", "noacpi" or even "acpi=off".

If the "platform" test fails, there is a problem with the handling of the platform (eg. ACPI) firmware on your system. In that case the "platform" mode of hibernation is not likely to work. You can try the "shutdown" mode, but that is rather a poor man's workaround.

If the "processors" test fails, the disabling/enabling of nonboot CPUs does not work (of course, this only may be an issue on SMP systems) and the problem should be reported. In that case you can also try to switch the nonboot CPUs off and on using the /sys/devices/system/cpu/cpu*/online sysfs attributes and see if that works.

If the "core" test fails, which means that suspending of the system/platform devices has failed (these devices are suspended on one CPU with interrupts off), the problem is most probably hardware-related and serious, so it should be reported.

A failure of any of the "platform", "processors" or "core" tests may cause your system to hang or become unstable, so please beware. Such a failure usually indicates a serious problem that very well may be related to the hardware, but please report it anyway.

2.1.2 b) Testing minimal configuration

If all of the hibernation test modes work, you can boot the system with the "init=/bin/bash" command line parameter and attempt to hibernate in the "reboot", "shutdown" and "platform" modes. If that does not work, there probably is a problem with a driver statically compiled into the kernel and you can try to compile more drivers as modules, so that they can be tested individually. Otherwise, there is a problem with a modular driver and you can find it by loading a half of the modules you normally use and binary searching in accordance with the algorithm: - if there are n modules loaded and the attempt to suspend and resume fails, unload n/2 of the modules and try again (that would probably involve rebooting the system), - if there are n modules loaded and the attempt to suspend and resume succeeds, load n/2 modules more and try again.

Again, if you find the offending module(s), it(they) must be unloaded every time before hibernation, and please report the problem with it(them).

2.1.3 c) Using the "test_resume" hibernation option

/sys/power/disk generally tells the kernel what to do after creating a hibernation image. One of the available options is "test_resume" which causes the just created image to be used for immediate restoration. Namely, after doing:

```
# echo test_resume > /sys/power/disk
# echo disk > /sys/power/state
```

a hibernation image will be created and a resume from it will be triggered immediately without involving the platform firmware in any way.

That test can be used to check if failures to resume from hibernation are related to bad interactions with the platform firmware. That is, if the above works every time, but resume from actual hibernation does not work or is unreliable, the platform firmware may be responsible for the failures.

On architectures and platforms that support using different kernels to restore hibernation images (that is, the kernel used to read the image from storage and load it into memory is different from the one included in the image) or support kernel address space randomization, it also can be used to check if failures to resume may be related to the differences between the restore and image kernels.

2.1.4 d) Advanced debugging

In case that hibernation does not work on your system even in the minimal configuration and compiling more drivers as modules is not practical or some modules cannot be unloaded, you can use one of the more advanced debugging techniques to find the problem. First, if there is a serial port in your box, you can boot the kernel with the 'no_console_suspend' parameter and try to log kernel messages using the serial console. This may provide you with some information about the reasons of the suspend (resume) failure. Alternatively, it may be possible to use a FireWire port for debugging with firescope (http://v3.sk/~lkundrak/firescope/). On x86 it is also possible to use the PM_TRACE mechanism documented in *How to get s2ram working* .

2.2 2. Testing suspend to RAM (STR)

To verify that the STR works, it is generally more convenient to use the s2ram tool available from http://suspend.sf.net and documented at http://en.opensuse.org/SDB:Suspend_to_RAM (S2RAM_LINK).

Namely, after writing "freezer", "devices", "platform", "processors", or "core" into /sys/power/pm_test (available if the kernel is compiled with CONFIG_PM_DEBUG set) the suspend code will work in the test mode corresponding to given string. The STR test modes are defined in the same way as for hibernation, so please refer to Section 1 for more information about them. In particular, the "core" test allows you to test everything except for the actual invocation of the platform firmware in order to put the system into the sleep state.

Among other things, the testing with the help of /sys/power/pm_test may allow you to identify drivers that fail to suspend or resume their devices. They should be unloaded every time before an STR transition.

Next, you can follow the instructions at S2RAM_LINK to test the system, but if it does not work "out of the box", you may need to boot it with "init=/bin/bash" and test s2ram in the minimal configuration. In that case, you may be able to search for failing drivers by following the procedure analogous to the one described in section 1. If you find some failing drivers, you will have to unload them every time before an STR transition (ie. before you run s2ram), and please report the problems with them.

There is a debugfs entry which shows the suspend to RAM statistics. Here is an example of its output:

```
# mount -t debugfs none /sys/kernel/debug
# cat /sys/kernel/debug/suspend_stats
success: 20
fail: 5
failed_freeze: 0
failed_prepare: 0
failed_suspend: 5
failed_suspend_noirq: 0
failed_resume: 0
failed_resume_noirq: 0
failures:
  last_failed_dev:      alarm
                        adc
  last_failed_errno:    -16
                        -16
  last_failed_step:     suspend
                        suspend
```

Field success means the success number of suspend to RAM, and field fail means the failure number. Others are the failure number of different steps of suspend to RAM. suspend_stats just lists the last 2 failed devices, error number and failed step of suspend.

CHARGER MANAGER

(C) 2011 MyungJoo Ham <myungjoo.ham@samsung.com>, GPL

Charger Manager provides in-kernel battery charger management that requires temperature monitoring during suspend-to-RAM state and where each battery may have multiple chargers attached and the userland wants to look at the aggregated information of the multiple chargers.

Charger Manager is a platform_driver with power-supply-class entries. An instance of Charger Manager (a platform-device created with Charger-Manager) represents an independent battery with chargers. If there are multiple batteries with their own chargers acting independently in a system, the system may need multiple instances of Charger Manager.

3.1 1. Introduction

Charger Manager supports the following:

- **Support for multiple chargers (e.g., a device with USB, AC, and solar panels)**
  A system may have multiple chargers (or power sources) and some of they may be activated at the same time. Each charger may have its own power-supply-class and each power-supply-class can provide different information about the battery status. This framework aggregates charger-related information from multiple sources and shows combined information as a single power-supply-class.

- **Support for in suspend-to-RAM polling (with suspend_again callback)**
  While the battery is being charged and the system is in suspend-to-RAM, we may need to monitor the battery health by looking at the ambient or battery temperature. We can accomplish this by waking up the system periodically. However, such a method wakes up devices unnecessarily for monitoring the battery health and tasks, and user processes that are supposed to be kept suspended. That, in turn, incurs unnecessary power consumption and slow down charging process. Or even, such peak power consumption can stop chargers in the middle of charging (external power input < device power consumption), which not only affects the charging time, but the lifespan of the battery.

  Charger Manager provides a function "cm_suspend_again" that can be used as suspend_again callback of platform_suspend_ops. If the platform requires tasks other than cm_suspend_again, it may implement its own suspend_again callback that calls cm_suspend_again in the middle. Normally, the platform will need to resume and suspend some devices that are used by Charger Manager.

- **Support for premature full-battery event handling**
  If the battery voltage drops by "fullbatt_vchkdrop_uV" after "fullbatt_vchkdrop_ms"

from the full-battery event, the framework restarts charging. This check is also performed while suspended by setting wakeup time accordingly and using suspend_again.

- **Support for uevent-notify**

  With the charger-related events, the device sends notification to users with UEVENT.

3.2 2. Global Charger-Manager Data related with suspend_again

In order to setup Charger Manager with suspend-again feature (in-suspend monitoring), the user should provide charger_global_desc with setup_charger_manager(*struct charger_global_desc* *). This charger_global_desc data for in-suspend monitoring is global as the name suggests. Thus, the user needs to provide only once even if there are multiple batteries. If there are multiple batteries, the multiple instances of Charger Manager share the same charger_global_desc and it will manage in-suspend monitoring for all instances of Charger Manager.

The user needs to provide all the three entries to *struct charger_global_desc* properly in order to activate in-suspend monitoring:

char *rtc_name;
>   The name of rtc (e.g., "rtc0") used to wakeup the system from suspend for Charger Manager. The alarm interrupt (AIE) of the rtc should be able to wake up the system from suspend. Charger Manager saves and restores the alarm value and use the previously-defined alarm if it is going to go off earlier than Charger Manager so that Charger Manager does not interfere with previously-defined alarms.

bool (*rtc_only_wakeup)(void);
>   This callback should let CM know whether the wakeup-from-suspend is caused only by the alarm of "rtc" in the same struct. If there is any other wakeup source triggered the wakeup, it should return false. If the "rtc" is the only wakeup reason, it should return true.

bool assume_timer_stops_in_suspend;
>   if true, Charger Manager assumes that the timer (CM uses jiffies as timer) stops during suspend. Then, CM assumes that the suspend-duration is same as the alarm length.

3.3 3. How to setup suspend_again

Charger Manager provides a function "extern bool cm_suspend_again(void)". When cm_suspend_again is called, it monitors every battery. The suspend_ops callback of the system's platform_suspend_ops can call cm_suspend_again function to know whether Charger Manager wants to suspend again or not. If there are no other devices or tasks that want to use suspend_again feature, the platform_suspend_ops may directly refer to cm_suspend_again for its suspend_again callback.

The cm_suspend_again() returns true (meaning "I want to suspend again") if the system was woken up by Charger Manager and the polling (in-suspend monitoring) results in "normal".

3.4 4. Charger-Manager Data (struct charger_desc)

For each battery charged independently from other batteries (if a series of batteries are charged by a single charger, they are counted as one independent battery), an instance of Charger Manager is attached to it. The following

struct charger_desc elements:

char *psy_name;
> The power-supply-class name of the battery. Default is "battery" if psy_name is NULL. Users can access the psy entries at "/sys/class/power_supply/[psy_name]/".

enum polling_modes polling_mode;

> **CM_POLL_DISABLE:**
>> do not poll this battery.

> **CM_POLL_ALWAYS:**
>> always poll this battery.

> **CM_POLL_EXTERNAL_POWER_ONLY:**
>> poll this battery if and only if an external power source is attached.

> **CM_POLL_CHARGING_ONLY:**
>> poll this battery if and only if the battery is being charged.

unsigned int fullbatt_vchkdrop_ms; / unsigned int fullbatt_vchkdrop_uV;
> If both have non-zero values, Charger Manager will check the battery voltage drop fullbatt_vchkdrop_ms after the battery is fully charged. If the voltage drop is over fullbatt_vchkdrop_uV, Charger Manager will try to recharge the battery by disabling and enabling chargers. Recharge with voltage drop condition only (without delay condition) is needed to be implemented with hardware interrupts from fuel gauges or charger devices/chips.

unsigned int fullbatt_uV;
> If specified with a non-zero value, Charger Manager assumes that the battery is full (capacity = 100) if the battery is not being charged and the battery voltage is equal to or greater than fullbatt_uV.

unsigned int polling_interval_ms;
> Required polling interval in ms. Charger Manager will poll this battery every polling_interval_ms or more frequently.

enum data_source battery_present;

> **CM_BATTERY_PRESENT:**
>> assume that the battery exists.

> **CM_NO_BATTERY:**
>> assume that the battery does not exists.

> **CM_FUEL_GAUGE:**
>> get battery presence information from fuel gauge.

> **CM_CHARGER_STAT:**
>> get battery presence from chargers.

char **psy_charger_stat;
> An array ending with NULL that has power-supply-class names of chargers. Each power-supply-class should provide ”PRESENT” (if battery_present is ”CM_CHARGER_STAT”), ”ONLINE” (shows whether an external power source is attached or not), and ”STATUS” (shows whether the battery is {”FULL” or not FULL} or {”FULL”, ”Charging”, ”Discharging”, ”NotCharging”}).

int num_charger_regulators; / struct regulator_bulk_data *charger_regulators;
> Regulators representing the chargers in the form for regulator framework's bulk functions.

char *psy_fuel_gauge;
> Power-supply-class name of the fuel gauge.

int (*temperature_out_of_range)(int *mC); / bool measure_battery_temp;
> This callback returns 0 if the temperature is safe for charging, a positive number if it is too hot to charge, and a negative number if it is too cold to charge. With the variable mC, the callback returns the temperature in 1/1000 of centigrade. The source of temperature can be battery or ambient one according to the value of measure_battery_temp.

3.5 5.   Notify   Charger-Manager   of   charger   events: cm_notify_event()

If there is an charger event is required to notify Charger Manager, a charger device driver that triggers the event can call cm_notify_event(psy, type, msg) to notify the corresponding Charger Manager. In the function, psy is the charger driver's power_supply pointer, which is associated with Charger-Manager. The parameter ”type” is the same as irq's type (enum cm_event_types). The event message ”msg” is optional and is effective only if the event type is ”UNDESCRIBED” or ”OTHERS”.

3.6 6. Other Considerations

At the charger/battery-related events such as battery-pulled-out, charger-pulled-out, charger-inserted, DCIN-over/under-voltage, charger-stopped, and others critical to chargers, the system should be configured to wake up. At least the following should wake up the system from a suspend: a) charger-on/off b) external-power-in/out c) battery-in/out (while charging)

It is usually accomplished by configuring the PMIC as a wakeup source.

TESTING SUSPEND AND RESUME SUPPORT IN DEVICE DRIVERS

(C) 2007 Rafael J. Wysocki <rjw@sisk.pl>, GPL

4.1 1. Preparing the test system

Unfortunately, to effectively test the support for the system-wide suspend and resume transitions in a driver, it is necessary to suspend and resume a fully functional system with this driver loaded. Moreover, that should be done several times, preferably several times in a row, and separately for hibernation (aka suspend to disk or STD) and suspend to RAM (STR), because each of these cases involves slightly different operations and different interactions with the machine's BIOS.

Of course, for this purpose the test system has to be known to suspend and resume without the driver being tested. Thus, if possible, you should first resolve all suspend/resume-related problems in the test system before you start testing the new driver. Please see *Debugging hibernation and suspend* for more information about the debugging of suspend/resume functionality.

4.2 2. Testing the driver

Once you have resolved the suspend/resume-related problems with your test system without the new driver, you are ready to test it:

  a) Build the driver as a module, load it and try the test modes of hibernation (see: *Debugging hibernation and suspend*, 1).

  b) Load the driver and attempt to hibernate in the "reboot", "shutdown" and "platform" modes (see: *Debugging hibernation and suspend*, 1).

  c) Compile the driver directly into the kernel and try the test modes of hibernation.

  d) Attempt to hibernate with the driver compiled directly into the kernel in the "reboot", "shutdown" and "platform" modes.

  e) Try the test modes of suspend (see: *Debugging hibernation and suspend*, 2). [As far as the STR tests are concerned, it should not matter whether or not the driver is built as a module.]

  f) Attempt to suspend to RAM using the s2ram tool with the driver loaded (see: *Debugging hibernation and suspend*, 2).

Each of the above tests should be repeated several times and the STD tests should be mixed with the STR tests. If any of them fails, the driver cannot be regarded as suspend/resume-safe.

ENERGY MODEL OF DEVICES
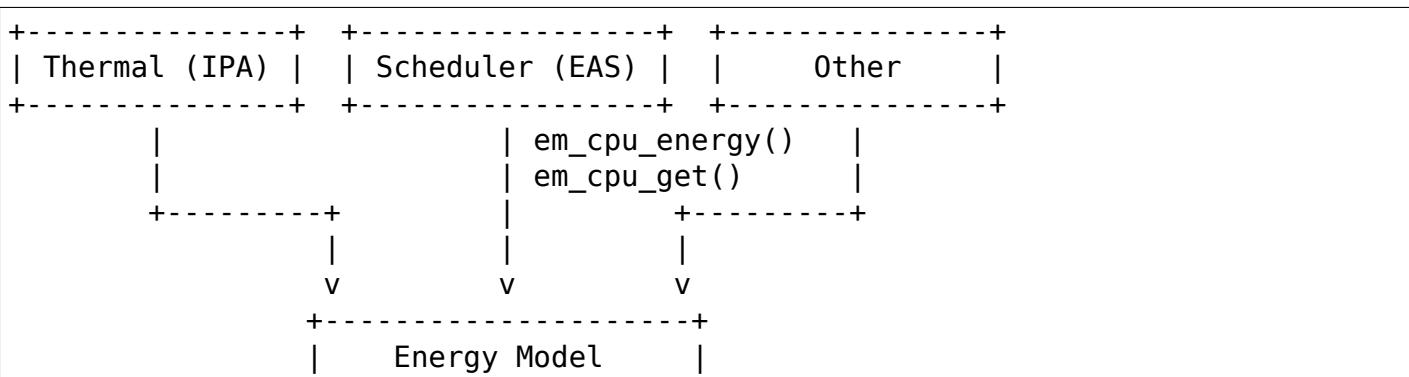
5.1 1. Overview
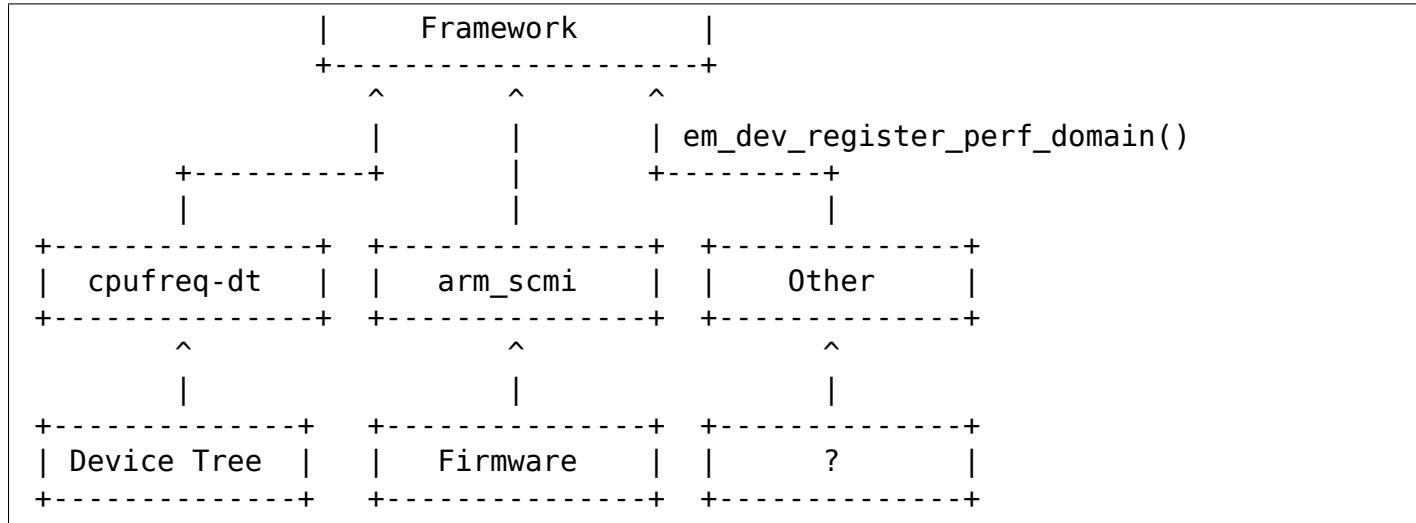
The Energy Model (EM) framework serves as an interface between drivers knowing the power consumed by devices at various performance levels, and the kernel subsystems willing to use that information to make energy-aware decisions.

The source of the information about the power consumed by devices can vary greatly from one platform to another. These power costs can be estimated using devicetree data in some cases. In others, the firmware will know better. Alternatively, userspace might be best positioned. And so on. In order to avoid each and every client subsystem to re-implement support for each and every possible source of information on its own, the EM framework intervenes as an abstraction layer which standardizes the format of power cost tables in the kernel, hence enabling to avoid redundant work.

The power values might be expressed in micro-Watts or in an 'abstract scale'. Multiple subsystems might use the EM and it is up to the system integrator to check that the requirements for the power value scale types are met. An example can be found in the Energy-Aware Scheduler documentation Documentation/scheduler/sched-energy.rst. For some subsystems like thermal or powercap power values expressed in an 'abstract scale' might cause issues. These subsystems are more interested in estimation of power used in the past, thus the real micro-Watts might be needed. An example of these requirements can be found in the Intelligent Power Allocation in Documentation/driver-api/thermal/power_allocator.rst. Kernel subsystems might implement automatic detection to check whether EM registered devices have inconsistent scale (based on EM internal flag). Important thing to keep in mind is that when the power values are expressed in an 'abstract scale' deriving real energy in micro-Joules would not be possible.

The figure below depicts an example of drivers (Arm-specific here, but the approach is applicable to any architecture) providing power costs to the EM framework, and interested clients reading the data from it:

```
+---------------+  +-----------------+  +---------------+
| Thermal (IPA) |  | Scheduler (EAS) |  |     Other     |
+---------------+  +-----------------+  +---------------+
        |                  | em_cpu_energy()   |
        |                  | em_cpu_get()      |
    +---------+            |          +---------+
        |            |     |          |
        v            v     v          v
      +----------------------+
      |     Energy Model     |
```

```
                  |      Framework      |
              +---------------------+
                   ^         ^         ^
                   |         |         | em_dev_register_perf_domain()
         +---------+         |     +---------+
         |                   |         |
 +---------------+ +---------------+ +---------------+
 |  cpufreq-dt   | |   arm_scmi    | |     Other     |
 +---------------+ +---------------+ +---------------+
         ^                 ^                 ^
         |                 |                 |
 +---------------+ +---------------+ +---------------+
 | Device Tree   | |   Firmware    | |       ?       |
 +---------------+ +---------------+ +---------------+
```

In case of CPU devices the EM framework manages power cost tables per 'performance domain' in the system. A performance domain is a group of CPUs whose performance is scaled together. Performance domains generally have a 1-to-1 mapping with CPUFreq policies. All CPUs in a performance domain are required to have the same micro-architecture. CPUs in different performance domains can have different micro-architectures.

5.2 2. Core APIs

5.2.1 2.1 Config options

CONFIG_ENERGY_MODEL must be enabled to use the EM framework.

5.2.2 2.2 Registration of performance domains

Registration of 'advanced' EM

The 'advanced' EM gets its name due to the fact that the driver is allowed to provide more precised power model. It's not limited to some implemented math formula in the framework (like it is in 'simple' EM case). It can better reflect the real power measurements performed for each performance state. Thus, this registration method should be preferred in case considering EM static power (leakage) is important.

Drivers are expected to register performance domains into the EM framework by calling the following API:

```
int em_dev_register_perf_domain(struct device *dev, unsigned int nr_states,
            struct em_data_callback *cb, cpumask_t *cpus, bool microwatts);
```

Drivers must provide a callback function returning <frequency, power> tuples for each performance state. The callback function provided by the driver is free to fetch data from any relevant location (DT, firmware, ...), and by any mean deemed necessary. Only for CPU devices, drivers must specify the CPUs of the performance domains using cpumask. For other devices than CPUs the last argument must be set to NULL. The last argument 'microwatts' is important to set with correct value. Kernel subsystems which use EM might rely on this flag

to check if all EM devices use the same scale. If there are different scales, these subsystems might decide to return warning/error, stop working or panic. See Section 3. for an example of driver implementing this callback, or Section 2.4 for further documentation on this API

Registration of EM using DT

The EM can also be registered using OPP framework and information in DT "operating-points-v2". Each OPP entry in DT can be extended with a property "opp-microwatt" containing micro-Watts power value. This OPP DT property allows a platform to register EM power values which are reflecting total power (static + dynamic). These power values might be coming directly from experiments and measurements.

Registration of 'artificial' EM

There is an option to provide a custom callback for drivers missing detailed knowledge about power value for each performance state. The callback .get_cost() is optional and provides the 'cost' values used by the EAS. This is useful for platforms that only provide information on relative efficiency between CPU types, where one could use the information to create an abstract power model. But even an abstract power model can sometimes be hard to fit in, given the input power value size restrictions. The .get_cost() allows to provide the 'cost' values which reflect the efficiency of the CPUs. This would allow to provide EAS information which has different relation than what would be forced by the EM internal formulas calculating 'cost' values. To register an EM for such platform, the driver must set the flag 'microwatts' to 0, provide .get_power() callback and provide .get_cost() callback. The EM framework would handle such platform properly during registration. A flag EM_PERF_DOMAIN_ARTIFICIAL is set for such platform. Special care should be taken by other frameworks which are using EM to test and treat this flag properly.

Registration of 'simple' EM

The 'simple' EM is registered using the framework helper function cpufreq_register_em_with_opp(). It implements a power model which is tight to math formula:

```
Power = C * V^2 * f
```

The EM which is registered using this method might not reflect correctly the physics of a real device, e.g. when static power (leakage) is important.

5.2.3 2.3 Accessing performance domains

There are two API functions which provide the access to the energy model: em_cpu_get() which takes CPU id as an argument and em_pd_get() with device pointer as an argument. It depends on the subsystem which interface it is going to use, but in case of CPU devices both functions return the same performance domain.

Subsystems interested in the energy model of a CPU can retrieve it using the em_cpu_get() API. The energy model tables are allocated once upon creation of the performance domains, and kept in memory untouched.

The energy consumed by a performance domain can be estimated using the em_cpu_energy() API. The estimation is performed assuming that the schedutil CPUfreq governor is in use in case of CPU device. Currently this calculation is not provided for other type of devices.

More details about the above APIs can be found in `<linux/energy_model.h>` or in Section 2.4

5.2.4 2.4 Description details of this API

struct **em_perf_state**

> Performance state of a performance domain

Definition:

```
struct em_perf_state {
    unsigned long frequency;
    unsigned long power;
    unsigned long cost;
    unsigned long flags;
};
```

Members

frequency

> The frequency in KHz, for consistency with CPUFreq

power

> The power consumed at this level (by 1 CPU or by a registered device). It can be a total power: static and dynamic.

cost

> The cost coefficient associated with this level, used during energy calculation. Equal to: power * max_frequency / frequency

flags

> see "em_perf_state flags" description below.

struct **em_perf_domain**

> Performance domain

Definition:

```
struct em_perf_domain {
    struct em_perf_state *table;
    int nr_perf_states;
    unsigned long flags;
    unsigned long cpus[];
};
```

Members

table

> List of performance states, in ascending order

nr_perf_states

> Number of performance states

flags
    See "em_perf_domain flags"

cpus
    Cpumask covering the CPUs of the domain. It's here for performance reasons to avoid potential cache misses during energy calculations in the scheduler and simplifies allocating/freeing that memory region.

Description

In case of CPU device, a "performance domain" represents a group of CPUs whose performance is scaled together. All CPUs of a performance domain must have the same micro-architecture. Performance domains often have a 1-to-1 mapping with CPUFreq policies. In case of other devices the **cpus** field is unused.

struct *em_perf_state* ***em_pd_get_efficient_state**(struct *em_perf_domain* *pd, unsigned
                                                                    long freq)
    Get an efficient performance state from the EM

Parameters

struct em_perf_domain *pd
    Performance domain for which we want an efficient frequency

unsigned long freq
    Frequency to map with the EM

Description

It is called from the scheduler code quite frequently and as a consequence doesn't implement any check.

Return

An efficient performance state, high enough to meet **freq** requirement.

unsigned long **em_cpu_energy**(struct *em_perf_domain* *pd, unsigned long max_util, unsigned
                                      long sum_util, unsigned long allowed_cpu_cap)
    Estimates the energy consumed by the CPUs of a performance domain

Parameters

struct em_perf_domain *pd
    performance domain for which energy has to be estimated

unsigned long max_util
    highest utilization among CPUs of the domain

unsigned long sum_util
    sum of the utilization of all CPUs in the domain

unsigned long allowed_cpu_cap
    maximum allowed CPU capacity for the **pd**, which might reflect reduced frequency (due to thermal)

Description

This function must be used only for CPU devices. There is no validation, i.e. if the EM is a CPU type and has cpumask allocated. It is called from the scheduler code quite frequently and that is why there is not checks.

Return

the sum of the energy consumed by the CPUs of the domain assuming a capacity state satisfying the max utilization of the domain.

int **em_pd_nr_perf_states**(struct *em_perf_domain* *pd)

> Get the number of performance states of a perf. domain

Parameters

struct em_perf_domain *pd

> performance domain for which this must be done

Return

the number of performance states in the performance domain table

struct *em_perf_domain* ***em_pd_get**(struct device *dev)

> Return the performance domain for a device

Parameters

struct device *dev

> Device to find the performance domain for

Description

Returns the performance domain to which **dev** belongs, or NULL if it doesn't exist.

struct *em_perf_domain* ***em_cpu_get**(int cpu)

> Return the performance domain for a CPU

Parameters

int cpu

> CPU to find the performance domain for

Description

Returns the performance domain to which **cpu** belongs, or NULL if it doesn't exist.

int **em_dev_register_perf_domain**(struct device *dev, unsigned int nr_states, struct em_data_callback *cb, cpumask_t *cpus, bool microwatts)

> Register the Energy Model (EM) for a device

Parameters

struct device *dev

> Device for which the EM is to register

unsigned int nr_states

> Number of performance states to register

struct em_data_callback *cb

> Callback functions providing the data of the Energy Model

cpumask_t *cpus

> Pointer to cpumask_t, which in case of a CPU device is obligatory. It can be taken from i.e. 'policy->cpus'. For other type of devices this should be set to NULL.

bool microwatts
>   Flag indicating that the power values are in micro-Watts or in some other scale. It must be set properly.

Description

Create Energy Model tables for a performance domain using the callbacks defined in cb.

The **microwatts** is important to set with correct value. Some kernel sub-systems might rely on this flag and check if all devices in the EM are using the same scale.

If multiple clients register the same performance domain, all but the first registration will be ignored.

Return 0 on success

void **em_dev_unregister_perf_domain**(struct device *dev)
>   Unregister Energy Model (EM) for a device

Parameters

struct device *dev
>   Device for which the EM is registered

Description

Unregister the EM for the specified **dev** (but not a CPU device).

5.3 3. Example driver

The CPUFreq framework supports dedicated callback for registering the EM for a given CPU(s) 'policy' object: cpufreq_driver::register_em(). That callback has to be implemented properly for a given driver, because the framework would call it at the right time during setup. This section provides a simple example of a CPUFreq driver registering a performance domain in the Energy Model framework using the (fake) 'foo' protocol. The driver implements an est_power() function to be provided to the EM framework:

```
-> drivers/cpufreq/foo_cpufreq.c

01    static int est_power(struct device *dev, unsigned long *mW,
02                    unsigned long *KHz)
03    {
04            long freq, power;
05
06            /* Use the 'foo' protocol to ceil the frequency */
07            freq = foo_get_freq_ceil(dev, *KHz);
08            if (freq < 0);
09                    return freq;
10
11            /* Estimate the power cost for the dev at the relevant freq. */
12            power = foo_estimate_power(dev, freq);
13            if (power < 0);
14                    return power;
15
16            /* Return the values to the EM framework */
```

```
17                 *mW = power;
18                 *KHz = freq;
19
20                 return 0;
21      }
22
23      static void foo_cpufreq_register_em(struct cpufreq_policy *policy)
24      {
25                 struct em_data_callback em_cb = EM_DATA_CB(est_power);
26                 struct device *cpu_dev;
27                 int nr_opp;
28
29                 cpu_dev = get_cpu_device(cpumask_first(policy->cpus));
30
31                 /* Find the number of OPPs for this policy */
32                 nr_opp = foo_get_nr_opp(policy);
33
34                 /* And register the new performance domain */
35                 em_dev_register_perf_domain(cpu_dev, nr_opp, &em_cb, policy->
↪cpus,
36                                             true);
37      }
38
39      static struct cpufreq_driver foo_cpufreq_driver = {
40                 .register_em = foo_cpufreq_register_em,
41      };
```

FREEZING OF TASKS

(C) 2007 Rafael J. Wysocki <rjw@sisk.pl>, GPL

6.1 I. What is the freezing of tasks?

The freezing of tasks is a mechanism by which user space processes and some kernel threads are controlled during hibernation or system-wide suspend (on some architectures).

6.2 II. How does it work?

There is one per-task flag (PF_NOFREEZE) and three per-task states (TASK_FROZEN, TASK_FREEZABLE and __TASK_FREEZABLE_UNSAFE) used for that. The tasks that have PF_NOFREEZE unset (all user space tasks and some kernel threads) are regarded as 'freezable' and treated in a special way before the system enters a sleep state as well as before a hibernation image is created (hibernation is directly covered by what follows, but the description applies to system-wide suspend too).

Namely, as the first step of the hibernation procedure the function freeze_processes() (defined in kernel/power/process.c) is called. A system-wide static key freezer_active (as opposed to a per-task flag or state) is used to indicate whether the system is to undergo a freezing operation. And freeze_processes() sets this static key. After this, it executes try_to_freeze_tasks() that sends a fake signal to all user space processes, and wakes up all the kernel threads. All freezable tasks must react to that by calling try_to_freeze(), which results in a call to __refrigerator() (defined in kernel/freezer.c), which changes the task's state to TASK_FROZEN, and makes it loop until it is woken by an explicit TASK_FROZEN wakeup. Then, that task is regarded as 'frozen' and so the set of functions handling this mechanism is referred to as 'the freezer' (these functions are defined in kernel/power/process.c, kernel/freezer.c & include/linux/freezer.h). User space tasks are generally frozen before kernel threads.

__refrigerator() must not be called directly. Instead, use the try_to_freeze() function (defined in include/linux/freezer.h), that checks if the task is to be frozen and makes the task enter __refrigerator().

For user space processes try_to_freeze() is called automatically from the signal-handling code, but the freezable kernel threads need to call it explicitly in suitable places or use the wait_event_freezable() or wait_event_freezable_timeout() macros (defined in include/linux/wait.h) that put the task to sleep (TASK_INTERRUPTIBLE) or freeze it (TASK_FROZEN) if freezer_active is set. The main loop of a freezable kernel thread may look like the following one:

```
set_freezable();

while (true) {
        struct task_struct *tsk = NULL;

        wait_event_freezable(oom_reaper_wait, oom_reaper_list != NULL);
        spin_lock_irq(&oom_reaper_lock);
        if (oom_reaper_list != NULL) {
                tsk = oom_reaper_list;
                oom_reaper_list = tsk->oom_reaper_list;
        }
        spin_unlock_irq(&oom_reaper_lock);

        if (tsk)
                oom_reap_task(tsk);
}
```

(from mm/oom_kill.c::oom_reaper()).

If a freezable kernel thread is not put to the frozen state after the freezer has initiated a freezing operation, the freezing of tasks will fail and the entire system-wide transition will be cancelled. For this reason, freezable kernel threads must call try_to_freeze() somewhere or use one of the wait_event_freezable() and wait_event_freezable_timeout() macros.

After the system memory state has been restored from a hibernation image and devices have been reinitialized, the function thaw_processes() is called in order to wake up each frozen task. Then, the tasks that have been frozen leave __refrigerator() and continue running.

6.2.1 Rationale behind the functions dealing with freezing and thawing of tasks

freeze_processes():

> • freezes only userspace tasks

freeze_kernel_threads():

> • freezes all tasks (including kernel threads) because we can't freeze kernel threads without freezing userspace tasks

thaw_kernel_threads():

> • thaws only kernel threads; this is particularly useful if we need to do anything special in between thawing of kernel threads and thawing of userspace tasks, or if we want to postpone the thawing of userspace tasks

thaw_processes():

> • thaws all tasks (including kernel threads) because we can't thaw userspace tasks without thawing kernel threads

6.3 III. Which kernel threads are freezable?

Kernel threads are not freezable by default. However, a kernel thread may clear PF_NOFREEZE for itself by calling set_freezable() (the resetting of PF_NOFREEZE directly is not allowed). From this point it is regarded as freezable and must call try_to_freeze() or variants of wait_event_freezable() in a suitable place.

6.4 IV. Why do we do that?

Generally speaking, there is a couple of reasons to use the freezing of tasks:

1. The principal reason is to prevent filesystems from being damaged after hibernation. At the moment we have no simple means of checkpointing filesystems, so if there are any modifications made to filesystem data and/or metadata on disks, we cannot bring them back to the state from before the modifications. At the same time each hibernation image contains some filesystem-related information that must be consistent with the state of the on-disk data and metadata after the system memory state has been restored from the image (otherwise the filesystems will be damaged in a nasty way, usually making them almost impossible to repair). We therefore freeze tasks that might cause the on-disk filesystems' data and metadata to be modified after the hibernation image has been created and before the system is finally powered off. The majority of these are user space processes, but if any of the kernel threads may cause something like this to happen, they have to be freezable.

2. Next, to create the hibernation image we need to free a sufficient amount of memory (approximately 50% of available RAM) and we need to do that before devices are deactivated, because we generally need them for swapping out. Then, after the memory for the image has been freed, we don't want tasks to allocate additional memory and we prevent them from doing that by freezing them earlier. [Of course, this also means that device drivers should not allocate substantial amounts of memory from their .suspend() callbacks before hibernation, but this is a separate issue.]

3. The third reason is to prevent user space processes and some kernel threads from interfering with the suspending and resuming of devices. A user space process running on a second CPU while we are suspending devices may, for example, be troublesome and without the freezing of tasks we would need some safeguards against race conditions that might occur in such a case.

Although Linus Torvalds doesn't like the freezing of tasks, he said this in one of the discussions on LKML (https://lore.kernel.org/r/alpine.LFD.0.98.0704271801020.9964@woody.linux-foundation.org):

"RJW:> Why we freeze tasks at all or why we freeze kernel threads?

Linus: In many ways, 'at all'.

I **do** realize the IO request queue issues, and that we cannot actually do s2ram with some devices in the middle of a DMA. So we want to be able to avoid *that*, there's no question about that. And I suspect that stopping user threads and then waiting for a sync is practically one of the easier ways to do so.

So in practice, the 'at all' may become a 'why freeze kernel threads?' and freezing user threads I don't find really objectionable."

Still, there are kernel threads that may want to be freezable. For example, if a kernel thread that belongs to a device driver accesses the device directly, it in principle needs to know when the device is suspended, so that it doesn't try to access it at that time. However, if the kernel thread is freezable, it will be frozen before the driver's .suspend() callback is executed and it will be thawed after the driver's .resume() callback has run, so it won't be accessing the device while it's suspended.

4. Another reason for freezing tasks is to prevent user space processes from realizing that hibernation (or suspend) operation takes place. Ideally, user space processes should not notice that such a system-wide operation has occurred and should continue running without any problems after the restore (or resume from suspend). Unfortunately, in the most general case this is quite difficult to achieve without the freezing of tasks. Consider, for example, a process that depends on all CPUs being online while it's running. Since we need to disable nonboot CPUs during the hibernation, if this process is not frozen, it may notice that the number of CPUs has changed and may start to work incorrectly because of that.

6.5 V. Are there any problems related to the freezing of tasks?

Yes, there are.

First of all, the freezing of kernel threads may be tricky if they depend one on another. For example, if kernel thread A waits for a completion (in the TASK_UNINTERRUPTIBLE state) that needs to be done by freezable kernel thread B and B is frozen in the meantime, then A will be blocked until B is thawed, which may be undesirable. That's why kernel threads are not freezable by default.

Second, there are the following two problems related to the freezing of user space processes:

1. Putting processes into an uninterruptible sleep distorts the load average.

2. Now that we have FUSE, plus the framework for doing device drivers in userspace, it gets even more complicated because some userspace processes are now doing the sorts of things that kernel threads do (https://lists.linux-foundation.org/pipermail/linux-pm/2007-May/012309.html).

The problem 1. seems to be fixable, although it hasn't been fixed so far. The other one is more serious, but it seems that we can work around it by using hibernation (and suspend) notifiers (in that case, though, we won't be able to avoid the realization by the user space processes that the hibernation is taking place).

There are also problems that the freezing of tasks tends to expose, although they are not directly related to it. For example, if request_firmware() is called from a device driver's .resume() routine, it will timeout and eventually fail, because the user land process that should respond to the request is frozen at this point. So, seemingly, the failure is due to the freezing of tasks. Suppose, however, that the firmware file is located on a filesystem accessible only through another device that hasn't been resumed yet. In that case, request_firmware() will fail regardless of whether or not the freezing of tasks is used. Consequently, the problem is not really related to the freezing of tasks, since it generally exists anyway.

A driver must have all firmwares it may need in RAM before suspend() is called. If keeping them is not practical, for example due to their size, they must be requested early enough using the suspend notifier API described in Documentation/driver-api/pm/notifiers.rst.

6.6 VI. Are there any precautions to be taken to prevent freezing failures?

Yes, there are.

First of all, grabbing the 'system_transition_mutex' lock to mutually exclude a piece of code from system-wide sleep such as suspend/hibernation is not encouraged. If possible, that piece of code must instead hook onto the suspend/hibernation notifiers to achieve mutual exclusion. Look at the CPU-Hotplug code (kernel/cpu.c) for an example.

However, if that is not feasible, and grabbing 'system_transition_mutex' is deemed necessary, it is strongly discouraged to directly call mutex_[un]lock(&system_transition_mutex) since that could lead to freezing failures, because if the suspend/hibernate code successfully acquired the 'system_transition_mutex' lock, and hence that other entity failed to acquire the lock, then that task would get blocked in TASK_UNINTERRUPTIBLE state. As a consequence, the freezer would not be able to freeze that task, leading to freezing failure.

However, the [un]lock_system_sleep() APIs are safe to use in this scenario, since they ask the freezer to skip freezing this task, since it is anyway "frozen enough" as it is blocked on 'system_transition_mutex', which will be released only after the entire suspend/hibernation sequence is complete. So, to summarize, use [un]lock_system_sleep() instead of directly using mutex_[un]lock(&system_transition_mutex). That would prevent freezing failures.

6.7 V. Miscellaneous

/sys/power/pm_freeze_timeout controls how long it will cost at most to freeze all user space processes or all freezable kernel threads, in unit of millisecond. The default value is 20000, with range of unsigned integer.

OPERATING PERFORMANCE POINTS (OPP) LIBRARY

(C) 2009-2010 Nishanth Menon <nm@ti.com>, Texas Instruments Incorporated

7.1 1. Introduction

7.1.1 1.1 What is an Operating Performance Point (OPP)?

Complex SoCs of today consists of a multiple sub-modules working in conjunction. In an operational system executing varied use cases, not all modules in the SoC need to function at their highest performing frequency all the time. To facilitate this, sub-modules in a SoC are grouped into domains, allowing some domains to run at lower voltage and frequency while other domains run at voltage/frequency pairs that are higher.

The set of discrete tuples consisting of frequency and voltage pairs that the device will support per domain are called Operating Performance Points or OPPs.

As an example:

Let us consider an MPU device which supports the following: {300MHz at minimum voltage of 1V}, {800MHz at minimum voltage of 1.2V}, {1GHz at minimum voltage of 1.3V}

We can represent these as three OPPs as the following {Hz, uV} tuples:

- {300000000, 1000000}
- {800000000, 1200000}
- {1000000000, 1300000}

7.1.2 1.2 Operating Performance Points Library

OPP library provides a set of helper functions to organize and query the OPP information. The library is located in drivers/opp/ directory and the header is located in include/linux/pm_opp.h. OPP library can be enabled by enabling CONFIG_PM_OPP from power management menuconfig menu. Certain SoCs such as Texas Instrument's OMAP framework allows to optionally boot at a certain OPP without needing cpufreq.

Typical usage of the OPP library is as follows:

```
(users)          -> registers a set of default OPPs          -> (library)
SoC framework  -> modifies on required cases certain OPPs   -> OPP layer
               -> queries to search/retrieve information     ->
```

OPP layer expects each domain to be represented by a unique device pointer. SoC framework registers a set of initial OPPs per device with the OPP layer. This list is expected to be an optimally small number typically around 5 per device. This initial list contains a set of OPPs that the framework expects to be safely enabled by default in the system.

Note on OPP Availability

As the system proceeds to operate, SoC framework may choose to make certain OPPs available or not available on each device based on various external factors. Example usage: Thermal management or other exceptional situations where SoC framework might choose to disable a higher frequency OPP to safely continue operations until that OPP could be re-enabled if possible.

OPP library facilitates this concept in its implementation. The following operational functions operate only on available opps: dev_pm_opp_find_freq_{ceil, floor}, dev_pm_opp_get_voltage, dev_pm_opp_get_freq, dev_pm_opp_get_opp_count.

dev_pm_opp_find_freq_exact is meant to be used to find the opp pointer which can then be used for dev_pm_opp_enable/disable functions to make an opp available as required.

WARNING: Users of OPP library should refresh their availability count using get_opp_count if dev_pm_opp_enable/disable functions are invoked for a device, the exact mechanism to trigger these or the notification mechanism to other dependent subsystems such as cpufreq are left to the discretion of the SoC specific framework which uses the OPP library. Similar care needs to be taken care to refresh the cpufreq table in cases of these operations.

7.2 2. Initial OPP List Registration

The SoC implementation calls dev_pm_opp_add function iteratively to add OPPs per device. It is expected that the SoC framework will register the OPP entries optimally- typical numbers range to be less than 5. The list generated by registering the OPPs is maintained by OPP library throughout the device operation. The SoC framework can subsequently control the availability of the OPPs dynamically using the dev_pm_opp_enable / disable functions.

dev_pm_opp_add
> Add a new OPP for a specific domain represented by the device pointer. The OPP is defined using the frequency and voltage. Once added, the OPP is assumed to be available and control of its availability can be done with the dev_pm_opp_enable/disable functions. OPP library internally stores and manages this information in the dev_pm_opp struct. This function may be used by SoC framework to define a optimal list as per the demands of SoC usage environment.
>
> **WARNING:**
> > Do not use this function in interrupt context.
>
> Example:

```
soc_pm_init()
{
        /* Do things */
        r = dev_pm_opp_add(mpu_dev, 1000000, 900000);
        if (!r) {
```

```
                    pr_err("%s: unable to register mpu opp(%d)\n", r);
                    goto no_cpufreq;
        }
        /* Do cpufreq things */
no_cpufreq:
        /* Do remaining things */
}
```

7.3 3. OPP Search Functions

High level framework such as cpufreq operates on frequencies. To map the frequency back to the corresponding OPP, OPP library provides handy functions to search the OPP list that OPP library internally manages. These search functions return the matching pointer representing the opp if a match is found, else returns error. These errors are expected to be handled by standard error checks such as IS_ERR() and appropriate actions taken by the caller.

Callers of these functions shall call dev_pm_opp_put() after they have used the OPP. Otherwise the memory for the OPP will never get freed and result in memleak.

dev_pm_opp_find_freq_exact
Search for an OPP based on an *exact* frequency and availability. This function is especially useful to enable an OPP which is not available by default. Example: In a case when SoC framework detects a situation where a higher frequency could be made available, it can use this function to find the OPP prior to call the dev_pm_opp_enable to actually make it available:

```
opp = dev_pm_opp_find_freq_exact(dev, 1000000000, false);
dev_pm_opp_put(opp);
/* dont operate on the pointer.. just do a sanity check.. */
if (IS_ERR(opp)) {
        pr_err("frequency not disabled!\n");
        /* trigger appropriate actions.. */
} else {
        dev_pm_opp_enable(dev,1000000000);
}
```

NOTE:
This is the only search function that operates on OPPs which are not available.

dev_pm_opp_find_freq_floor
Search for an available OPP which is *at most* the provided frequency. This function is useful while searching for a lesser match OR operating on OPP information in the order of decreasing frequency. Example: To find the highest opp for a device:

```
freq = ULONG_MAX;
opp = dev_pm_opp_find_freq_floor(dev, &freq);
dev_pm_opp_put(opp);
```

dev_pm_opp_find_freq_ceil
Search for an available OPP which is *at least* the provided frequency. This function is

useful while searching for a higher match OR operating on OPP information in the order of increasing frequency. Example 1: To find the lowest opp for a device:

```
freq = 0;
opp = dev_pm_opp_find_freq_ceil(dev, &freq);
dev_pm_opp_put(opp);
```

Example 2: A simplified implementation of a SoC cpufreq_driver->target:

```
soc_cpufreq_target(..)
{
        /* Do stuff like policy checks etc. */
        /* Find the best frequency match for the req */
        opp = dev_pm_opp_find_freq_ceil(dev, &freq);
        dev_pm_opp_put(opp);
        if (!IS_ERR(opp))
                soc_switch_to_freq_voltage(freq);
        else
                /* do something when we can't satisfy the req */
        /* do other stuff */
}
```

7.4 4. OPP Availability Control Functions

A default OPP list registered with the OPP library may not cater to all possible situation. The OPP library provides a set of functions to modify the availability of a OPP within the OPP list. This allows SoC frameworks to have fine grained dynamic control of which sets of OPPs are operationally available. These functions are intended to *temporarily* remove an OPP in conditions such as thermal considerations (e.g. don't use OPPx until the temperature drops).

WARNING:
Do not use these functions in interrupt context.

dev_pm_opp_enable
Make a OPP available for operation. Example: Lets say that 1GHz OPP is to be made available only if the SoC temperature is lower than a certain threshold. The SoC framework implementation might choose to do something as follows:

```
if (cur_temp < temp_low_thresh) {
        /* Enable 1GHz if it was disabled */
        opp = dev_pm_opp_find_freq_exact(dev, 1000000000, false);
        dev_pm_opp_put(opp);
        /* just error check */
        if (!IS_ERR(opp))
                ret = dev_pm_opp_enable(dev, 1000000000);
        else
                goto try_something_else;
}
```

dev_pm_opp_disable
Make an OPP to be not available for operation Example: Lets say that 1GHz OPP is to be

disabled if the temperature exceeds a threshold value. The SoC framework implementation might choose to do something as follows:

```
if (cur_temp > temp_high_thresh) {
        /* Disable 1GHz if it was enabled */
        opp = dev_pm_opp_find_freq_exact(dev, 1000000000, true);
        dev_pm_opp_put(opp);
        /* just error check */
        if (!IS_ERR(opp))
                ret = dev_pm_opp_disable(dev, 1000000000);
        else
                goto try_something_else;
}
```

7.5 5. OPP Data Retrieval Functions

Since OPP library abstracts away the OPP information, a set of functions to pull information from the dev_pm_opp structure is necessary. Once an OPP pointer is retrieved using the search functions, the following functions can be used by SoC framework to retrieve the information represented inside the OPP layer.

dev_pm_opp_get_voltage
    Retrieve the voltage represented by the opp pointer. Example: At a cpufreq transition to a different frequency, SoC framework requires to set the voltage represented by the OPP using the regulator framework to the Power Management chip providing the voltage:

```
soc_switch_to_freq_voltage(freq)
{
        /* do things */
        opp = dev_pm_opp_find_freq_ceil(dev, &freq);
        v = dev_pm_opp_get_voltage(opp);
        dev_pm_opp_put(opp);
        if (v)
                regulator_set_voltage(.., v);
        /* do other things */
}
```

dev_pm_opp_get_freq
    Retrieve the freq represented by the opp pointer. Example: Lets say the SoC framework uses a couple of helper functions we could pass opp pointers instead of doing additional parameters to handle quiet a bit of data parameters:

```
soc_cpufreq_target(..)
{
        /* do things.. */
         max_freq = ULONG_MAX;
         max_opp = dev_pm_opp_find_freq_floor(dev,&max_freq);
         requested_opp = dev_pm_opp_find_freq_ceil(dev,&freq);
         if (!IS_ERR(max_opp) && !IS_ERR(requested_opp))
                r = soc_test_validity(max_opp, requested_opp);
```

```
        dev_pm_opp_put(max_opp);
        dev_pm_opp_put(requested_opp);
        /* do other things */
}
soc_test_validity(..)
{
        if(dev_pm_opp_get_voltage(max_opp) < dev_pm_opp_get_
 →voltage(requested_opp))
                return -EINVAL;
        if(dev_pm_opp_get_freq(max_opp) < dev_pm_opp_get_freq(requested_
 →opp))
                return -EINVAL;
        /* do things.. */
}
```

dev_pm_opp_get_opp_count
 Retrieve the number of available opps for a device Example: Lets say a co-processor in
 the SoC needs to know the available frequencies in a table, the main processor can notify
 as following:

```
soc_notify_coproc_available_frequencies()
{
        /* Do things */
        num_available = dev_pm_opp_get_opp_count(dev);
        speeds = kzalloc(sizeof(u32) * num_available, GFP_KERNEL);
        /* populate the table in increasing order */
        freq = 0;
        while (!IS_ERR(opp = dev_pm_opp_find_freq_ceil(dev, &freq))) {
                speeds[i] = freq;
                freq++;
                i++;
                dev_pm_opp_put(opp);
        }

        soc_notify_coproc(AVAILABLE_FREQs, speeds, num_available);
        /* Do other things */
}
```

7.6 6. Data Structures

Typically an SoC contains multiple voltage domains which are variable. Each domain is repre-
sented by a device pointer. The relationship to OPP can be represented as follows:

```
SoC
 |- device 1
 |     |- opp 1 (availability, freq, voltage)
 |     |- opp 2 ..
 ...   ...
 |     `- opp n ..
```

```
 |- device 2
 ...
 `- device m
```

OPP library maintains a internal list that the SoC framework populates and accessed by various functions as described above. However, the structures representing the actual OPPs and domains are internal to the OPP library itself to allow for suitable abstraction reusable across systems.

struct dev_pm_opp
> The internal data structure of OPP library which is used to represent an OPP. In addition to the freq, voltage, availability information, it also contains internal book keeping information required for the OPP library to operate on. Pointer to this structure is provided back to the users such as SoC framework to be used as a identifier for OPP in the interactions with OPP layer.
>
> > **WARNING:**
> > The struct dev_pm_opp pointer should not be parsed or modified by the users. The defaults of for an instance is populated by dev_pm_opp_add, but the availability of the OPP can be modified by dev_pm_opp_enable/disable functions.

struct device
> This is used to identify a domain to the OPP layer. The nature of the device and its implementation is left to the user of OPP library such as the SoC framework.

Overall, in a simplistic view, the data structure operations is represented as following:

```
Initialization / modification:
            +-----+         /- dev_pm_opp_enable
dev_pm_opp_add --> | opp | <-------
   |        +-----+         \- dev_pm_opp_disable
  \-------> domain_info(device)

Search functions:
            /-- dev_pm_opp_find_freq_ceil  ---\   +-----+
domain_info<---- dev_pm_opp_find_freq_exact ----->| opp |
            \-- dev_pm_opp_find_freq_floor ---/   +-----+

Retrieval functions:
+-----+      /- dev_pm_opp_get_voltage
| opp | <---
+-----+      \- dev_pm_opp_get_freq

domain_info <- dev_pm_opp_get_opp_count
```

PCI POWER MANAGEMENT

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

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

This document only covers the aspects of power management specific to PCI devices. For general description of the kernel's interfaces related to device power management refer to Documentation/driver-api/pm/devices.rst and *Runtime Power Management Framework for I/O Devices*.

8.1 1. Hardware and Platform Support for PCI Power Management

8.1.1 1.1. Native and Platform-Based Power Management

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

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

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

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

though a PCI device is set up to generate PMEs, it also may be necessary to prepare the platform firmware for notifying the CPU of the PMEs coming from the device (e.g. by generating interrupts).

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

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

8.1.2 1.2. Native PCI Power Management

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

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

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

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

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

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

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

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

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

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

8.1.3 1.3. ACPI Device Power Management

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

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

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

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

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

required by the device in the target power state and are not required any more by any other device should be disabled (by executing their _OFF control methods). If the current power state of the device is D3, it can only be put into D0 this way.

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

8.1.4 1.4. Wakeup Signaling

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

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

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

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

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

Unfortunately, there is no standard way of handling wakeup signals sent by conventional PCI devices on systems that are not ACPI-based, but there is one for PCI Express devices. Namely,

the PCI Express Base Specification introduced a native mechanism for converting native PCI PMEs into interrupts generated by root ports. For conventional PCI devices native PMEs are out-of-band, so they are routed separately and they need not pass through bridges (in principle they may be routed directly to the system's core logic), but for PCI Express devices they are in-band messages that have to pass through the PCI Express hierarchy, including the root port on the path from the device to the Root Complex. Thus it was possible to introduce a mechanism by which a root port generates an interrupt whenever it receives a PME message from one of the devices below it. The PCI Express Requester ID of the device that sent the PME message is then recorded in one of the root port's configuration registers from where it may be read by the interrupt handler allowing the device to be identified. [PME messages sent by PCI Express endpoints integrated with the Root Complex don't pass through root ports, but instead they cause a Root Complex Event Collector (if there is one) to generate interrupts.]

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

8.2 2. PCI Subsystem and Device Power Management

8.2.1 2.1. Device Power Management Callbacks

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

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

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

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

```
struct pci_dev {
     ...
     pci_power_t     current_state;  /* Current operating state. */
     int             pm_cap;         /* PM capability offset in the
                                        configuration space */
     unsigned int    pme_support:5;  /* Bitmask of states from which PME#
                                        can be generated */
     unsigned int    pme_poll:1;     /* Poll device's PME status bit */
     unsigned int    d1_support:1;   /* Low power state D1 is supported */
     unsigned int    d2_support:1;   /* Low power state D2 is supported */
     unsigned int    no_d1d2:1;      /* D1 and D2 are forbidden */
     unsigned int    wakeup_prepared:1;  /* Device prepared for wake up */
     unsigned int    d3hot_delay;    /* D3hot->D0 transition time in ms */
     ...
};
```

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

8.2.2 2.2. Device Initialization

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

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

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

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

8.2.3 2.3. Runtime Device Power Management

The PCI subsystem plays a vital role in the runtime power management of PCI devices. For this purpose it uses the general runtime power management (runtime PM) framework described in *Runtime Power Management Framework for I/O Devices*. Namely, it provides subsystem-level callbacks:

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

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

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

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

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

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

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

The pci_pm_runtime_idle() function, called for PCI devices by pm_runtime_idle() and pm_request_idle(), executes the device driver's pm->runtime_idle() callback, if defined, and if that callback doesn't return error code (or is not present at all), suspends the device with the

help of pm_runtime_suspend(). Sometimes pci_pm_runtime_idle() is called automatically by the PM core (for example, it is called right after the device has just been resumed), in which cases it is expected to suspend the device if that makes sense. Usually, however, the PCI subsystem doesn't really know if the device really can be suspended, so it lets the device's driver decide by running its pm->runtime_idle() callback.

8.2.4 2.4. System-Wide Power Transitions

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

2.4.1. System Suspend

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

    prepare, suspend, suspend_noirq.

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

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

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

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

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

The pci_pm_suspend_noirq() routine is executed after suspend_device_irqs() has been called, which means that the device driver's interrupt handler won't be invoked while this routine is running. It first checks if the device's driver implements legacy PCI suspends routines (Section 3), in which case the legacy late suspend routine is called and its result is returned (the standard configuration registers of the device are saved if the driver's callback hasn't done

that). Second, if the device driver's struct dev_pm_ops object is not present, the device's standard configuration registers are saved and the routine returns success. Otherwise the device driver's pm->suspend_noirq() callback is executed, if present, and its result is returned if it fails. Next, if the device's standard configuration registers haven't been saved yet (one of the device driver's callbacks executed before might do that), pci_pm_suspend_noirq() saves them, prepares the device to signal wakeup (if necessary) and puts it into a low-power state.

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

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

2.4.2. System Resume

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

> resume_noirq, resume, complete.

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

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

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

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

the device's wakeup signaling mechanisms are blocked and its driver's pm->resume() callback is executed, if defined (the callback's result is then returned).

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

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

2.4.3. System Hibernation

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

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

> prepare, freeze, freeze_noirq

that correspond to the PCI bus type's callbacks:

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

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

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

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

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

> thaw_noirq, thaw, complete

using the following PCI bus type's callbacks:

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

respectively.

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

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

The complete phase is the same as for system resume.

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

> prepare, poweroff, poweroff_noirq

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

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

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

2.4.4. System Restore

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

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

> prepare, freeze, freeze_noirq

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

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

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

> restore_noirq, restore, complete

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

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

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

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

8.3 3. PCI Device Drivers and Power Management

8.3.1 3.1. Power Management Callbacks

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

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

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

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

3.1.1. prepare()

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

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

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

3.1.2. suspend()

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

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

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

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

3.1.3. suspend_noirq()

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

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

3.1.4. freeze()

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

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

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

3.1.5. freeze_noirq()

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

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

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

3.1.6. poweroff()

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

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

3.1.7. poweroff_noirq()

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

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

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

3.1.8. resume_noirq()

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

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

3.1.9. resume()

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

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

3.1.10. thaw_noirq()

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

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

3.1.11. thaw()

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

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

3.1.12. restore_noirq()

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

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

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

3.1.13. restore()

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

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

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

3.1.14. complete()

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

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

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

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

3.1.15. runtime_suspend()

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

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

3.1.16. runtime_resume()

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

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

3.1.17. runtime_idle()

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

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

3.1.18. Pointing Multiple Callback Pointers to One Routine

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

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

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

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

3.1.19. Driver Flags for Power Management

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

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

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

The DPM_FLAG_SMART_SUSPEND flag tells the PCI bus type that from the driver's perspective the device can be safely left in runtime suspend during system suspend. That causes pci_pm_suspend(), pci_pm_freeze() and pci_pm_poweroff() to avoid resuming the device from runtime suspend unless there are PCI-specific reasons for doing that. Also, it

causes pci_pm_suspend_late/noirq() and pci_pm_poweroff_late/noirq() to return early if the device remains in runtime suspend during the "late" phase of the system-wide transition under way. Moreover, if the device is in runtime suspend in pci_pm_resume_noirq() or pci_pm_restore_noirq(), its runtime PM status will be changed to "active" (as it is going to be put into D0 going forward).

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

8.3.2 3.2. Device Runtime Power Management

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

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

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

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

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

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

The driver itself should not call pm_runtime_allow(), though. Instead, it should let user space or some platform-specific code do that (user space can do it via sysfs as stated above), but it must

be prepared to handle the runtime PM of the device correctly as soon as pm_runtime_allow() is called (which may happen at any time, even before the driver is loaded).

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

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

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

For more information on the runtime PM of devices refer to *Runtime Power Management Framework for I/O Devices*.

8.4 4. Resources

PCI Local Bus Specification, Rev. 3.0

PCI Bus Power Management Interface Specification, Rev. 1.2

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

PCI Express Base Specification, Rev. 2.0

Documentation/driver-api/pm/devices.rst

Runtime Power Management Framework for I/O Devices

PM QUALITY OF SERVICE INTERFACE

This interface provides a kernel and user mode interface for registering performance expectations by drivers, subsystems and user space applications on one of the parameters.

Two different PM QoS frameworks are available:

- CPU latency QoS.

- The per-device PM QoS framework provides the API to manage the per-device latency constraints and PM QoS flags.

The latency unit used in the PM QoS framework is the microsecond (usec).

9.1 1. PM QoS framework

A global list of CPU latency QoS requests is maintained along with an aggregated (effective) target value. The aggregated target value is updated with changes to the request list or elements of the list. For CPU latency QoS, the aggregated target value is simply the min of the request values held in the list elements.

Note: the aggregated target value is implemented as an atomic variable so that reading the aggregated value does not require any locking mechanism.

From kernel space the use of this interface is simple:

void cpu_latency_qos_add_request(handle, target_value):
Will insert an element into the CPU latency QoS list with the target value. Upon change to this list the new target is recomputed and any registered notifiers are called only if the target value is now different. Clients of PM QoS need to save the returned handle for future use in other PM QoS API functions.

void cpu_latency_qos_update_request(handle, new_target_value):
Will update the list element pointed to by the handle with the new target value and recompute the new aggregated target, calling the notification tree if the target is changed.

void cpu_latency_qos_remove_request(handle):
Will remove the element. After removal it will update the aggregate target and call the notification tree if the target was changed as a result of removing the request.

int cpu_latency_qos_limit():
Returns the aggregated value for the CPU latency QoS.

int cpu_latency_qos_request_active(handle):
> Returns if the request is still active, i.e. it has not been removed from the CPU latency QoS list.

int cpu_latency_qos_add_notifier(notifier):
> Adds a notification callback function to the CPU latency QoS. The callback is called when the aggregated value for the CPU latency QoS is changed.

int cpu_latency_qos_remove_notifier(notifier):
> Removes the notification callback function from the CPU latency QoS.

From user space:

The infrastructure exposes one device node, /dev/cpu_dma_latency, for the CPU latency QoS.

Only processes can register a PM QoS request. To provide for automatic cleanup of a process, the interface requires the process to register its parameter requests as follows.

To register the default PM QoS target for the CPU latency QoS, the process must open /dev/cpu_dma_latency.

As long as the device node is held open that process has a registered request on the parameter.

To change the requested target value, the process needs to write an s32 value to the open device node. Alternatively, it can write a hex string for the value using the 10 char long format e.g. "0x12345678". This translates to a cpu_latency_qos_update_request() call.

To remove the user mode request for a target value simply close the device node.

9.2 2. PM QoS per-device latency and flags framework

For each device, there are three lists of PM QoS requests. Two of them are maintained along with the aggregated targets of resume latency and active state latency tolerance (in microseconds) and the third one is for PM QoS flags. Values are updated in response to changes of the request list.

The target values of resume latency and active state latency tolerance are simply the minimum of the request values held in the parameter list elements. The PM QoS flags aggregate value is a gather (bitwise OR) of all list elements' values. One device PM QoS flag is defined currently: PM_QOS_FLAG_NO_POWER_OFF.

Note: The aggregated target values are implemented in such a way that reading the aggregated value does not require any locking mechanism.

From kernel mode the use of this interface is the following:

int dev_pm_qos_add_request(device, handle, type, value):
> Will insert an element into the list for that identified device with the target value. Upon change to this list the new target is recomputed and any registered notifiers are called only if the target value is now different. Clients of dev_pm_qos need to save the handle for future use in other dev_pm_qos API functions.

int dev_pm_qos_update_request(handle, new_value):
> Will update the list element pointed to by the handle with the new target value and recompute the new aggregated target, calling the notification trees if the target is changed.

int dev_pm_qos_remove_request(handle):
Will remove the element. After removal it will update the aggregate target and call the notification trees if the target was changed as a result of removing the request.

s32 dev_pm_qos_read_value(device, type):
Returns the aggregated value for a given device's constraints list.

enum pm_qos_flags_status dev_pm_qos_flags(device, mask)
Check PM QoS flags of the given device against the given mask of flags. The meaning of the return values is as follows:

> **PM_QOS_FLAGS_ALL:**
> All flags from the mask are set
>
> **PM_QOS_FLAGS_SOME:**
> Some flags from the mask are set
>
> **PM_QOS_FLAGS_NONE:**
> No flags from the mask are set
>
> **PM_QOS_FLAGS_UNDEFINED:**
> The device's PM QoS structure has not been initialized or the list of requests is empty.

int dev_pm_qos_add_ancestor_request(dev, handle, type, value)
Add a PM QoS request for the first direct ancestor of the given device whose power.ignore_children flag is unset (for DEV_PM_QOS_RESUME_LATENCY requests) or whose power.set_latency_tolerance callback pointer is not NULL (for DEV_PM_QOS_LATENCY_TOLERANCE requests).

int dev_pm_qos_expose_latency_limit(device, value)
Add a request to the device's PM QoS list of resume latency constraints and create a sysfs attribute pm_qos_resume_latency_us under the device's power directory allowing user space to manipulate that request.

void dev_pm_qos_hide_latency_limit(device)
Drop the request added by dev_pm_qos_expose_latency_limit() from the device's PM QoS list of resume latency constraints and remove sysfs attribute pm_qos_resume_latency_us from the device's power directory.

int dev_pm_qos_expose_flags(device, value)
Add a request to the device's PM QoS list of flags and create sysfs attribute pm_qos_no_power_off under the device's power directory allowing user space to change the value of the PM_QOS_FLAG_NO_POWER_OFF flag.

void dev_pm_qos_hide_flags(device)
Drop the request added by dev_pm_qos_expose_flags() from the device's PM QoS list of flags and remove sysfs attribute pm_qos_no_power_off from the device's power directory.

Notification mechanisms:

The per-device PM QoS framework has a per-device notification tree.

int dev_pm_qos_add_notifier(device, notifier, type):
Adds a notification callback function for the device for a particular request type.

The callback is called when the aggregated value of the device constraints list is changed.

int dev_pm_qos_remove_notifier(device, notifier, type):
> Removes the notification callback function for the device.

9.2.1 Active state latency tolerance

This device PM QoS type is used to support systems in which hardware may switch to energy-saving operation modes on the fly. In those systems, if the operation mode chosen by the hardware attempts to save energy in an overly aggressive way, it may cause excess latencies to be visible to software, causing it to miss certain protocol requirements or target frame or sample rates etc.

If there is a latency tolerance control mechanism for a given device available to software, the .set_latency_tolerance callback in that device's dev_pm_info structure should be populated. The routine pointed to by it is should implement whatever is necessary to transfer the effective requirement value to the hardware.

Whenever the effective latency tolerance changes for the device, its .set_latency_tolerance() callback will be executed and the effective value will be passed to it. If that value is negative, which means that the list of latency tolerance requirements for the device is empty, the callback is expected to switch the underlying hardware latency tolerance control mechanism to an autonomous mode if available. If that value is PM_QOS_LATENCY_ANY, in turn, and the hardware supports a special "no requirement" setting, the callback is expected to use it. That allows software to prevent the hardware from automatically updating the device's latency tolerance in response to its power state changes (e.g. during transitions from D3cold to D0), which generally may be done in the autonomous latency tolerance control mode.

If .set_latency_tolerance() is present for the device, sysfs attribute pm_qos_latency_tolerance_us will be present in the devivce's power directory. Then, user space can use that attribute to specify its latency tolerance requirement for the device, if any. Writing "any" to it means "no requirement, but do not let the hardware control latency tolerance" and writing "auto" to it allows the hardware to be switched to the autonomous mode if there are no other requirements from the kernel side in the device's list.

Kernel code can use the functions described above along with the DEV_PM_QOS_LATENCY_TOLERANCE device PM QoS type to add, remove and update latency tolerance requirements for devices.

LINUX POWER SUPPLY CLASS

10.1 Synopsis

Power supply class used to represent battery, UPS, AC or DC power supply properties to user-space.

It defines core set of attributes, which should be applicable to (almost) every power supply out there. Attributes are available via sysfs and uevent interfaces.

Each attribute has well defined meaning, up to unit of measure used. While the attributes provided are believed to be universally applicable to any power supply, specific monitoring hardware may not be able to provide them all, so any of them may be skipped.

Power supply class is extensible, and allows to define drivers own attributes. The core attribute set is subject to the standard Linux evolution (i.e. if it will be found that some attribute is applicable to many power supply types or their drivers, it can be added to the core set).

It also integrates with LED framework, for the purpose of providing typically expected feedback of battery charging/fully charged status and AC/USB power supply online status. (Note that specific details of the indication (including whether to use it at all) are fully controllable by user and/or specific machine defaults, per design principles of LED framework).

10.2 Attributes/properties

Power supply class has predefined set of attributes, this eliminates code duplication across drivers. Power supply class insist on reusing its predefined attributes *and* their units.

So, userspace gets predictable set of attributes and their units for any kind of power supply, and can process/present them to a user in consistent manner. Results for different power supplies and machines are also directly comparable.

See drivers/power/supply/ds2760_battery.c for the example how to declare and handle attributes.

10.3 Units

Quoting include/linux/power_supply.h:

> All voltages, currents, charges, energies, time and temperatures in µV, µA, µAh, µWh, seconds and tenths of degree Celsius unless otherwise stated. It's driver's job to convert its raw values to units in which this class operates.

10.4 Attributes/properties detailed

Charge/Energy/Capacity - how to not confuse

Because both "charge" (µAh) and "energy" (µWh) represents "capacity" of battery, this class distinguish these terms. Don't mix them!
  • ***CHARGE_****
      attributes represents capacity in µAh only.
  • ***ENERGY_****
      attributes represents capacity in µWh only.
  • ***CAPACITY***
      attribute represents capacity in *percents*, from 0 to 100.

Postfixes:

_AVG
> *hardware* averaged value, use it if your hardware is really able to report averaged values.

_NOW
> momentary/instantaneous values.

STATUS
> this attribute represents operating status (charging, full, discharging (i.e. powering a load), etc.). This corresponds to *BATTERY_STATUS_** values, as defined in battery.h.

CHARGE_TYPE
> batteries can typically charge at different rates. This defines trickle and fast charges. For batteries that are already charged or discharging, 'n/a' can be displayed (or 'unknown', if the status is not known).

AUTHENTIC
> indicates the power supply (battery or charger) connected to the platform is authentic(1) or non authentic(0).

HEALTH
> represents health of the battery, values corresponds to POWER_SUPPLY_HEALTH_*, defined in battery.h.

VOLTAGE_OCV
> open circuit voltage of the battery.

VOLTAGE_MAX_DESIGN, VOLTAGE_MIN_DESIGN
> design values for maximal and minimal power supply voltages. Maximal/minimal means values of voltages when battery considered "full"/"empty" at normal conditions. Yes, there is no direct relation between voltage and battery capacity, but some dumb batteries use

voltage for very approximated calculation of capacity. Battery driver also can use this attribute just to inform userspace about maximal and minimal voltage thresholds of a given battery.

VOLTAGE_MAX, VOLTAGE_MIN
same as _DESIGN voltage values except that these ones should be used if hardware could only guess (measure and retain) the thresholds of a given power supply.

VOLTAGE_BOOT
Reports the voltage measured during boot

CURRENT_BOOT
Reports the current measured during boot

CHARGE_FULL_DESIGN, CHARGE_EMPTY_DESIGN
design charge values, when battery considered full/empty.

ENERGY_FULL_DESIGN, ENERGY_EMPTY_DESIGN
same as above but for energy.

CHARGE_FULL, CHARGE_EMPTY
These attributes means "last remembered value of charge when battery became full/empty". It also could mean "value of charge when battery considered full/empty at given conditions (temperature, age)". I.e. these attributes represents real thresholds, not design values.

ENERGY_FULL, ENERGY_EMPTY
same as above but for energy.

CHARGE_COUNTER
the current charge counter (in µAh). This could easily be negative; there is no empty or full value. It is only useful for relative, time-based measurements.

PRECHARGE_CURRENT
the maximum charge current during precharge phase of charge cycle (typically 20% of battery capacity).

CHARGE_TERM_CURRENT
Charge termination current. The charge cycle terminates when battery voltage is above recharge threshold, and charge current is below this setting (typically 10% of battery capacity).

CONSTANT_CHARGE_CURRENT
constant charge current programmed by charger.

CONSTANT_CHARGE_CURRENT_MAX
maximum charge current supported by the power supply object.

CONSTANT_CHARGE_VOLTAGE
constant charge voltage programmed by charger.

CONSTANT_CHARGE_VOLTAGE_MAX
maximum charge voltage supported by the power supply object.

INPUT_CURRENT_LIMIT
input current limit programmed by charger. Indicates the current drawn from a charging source.

INPUT_VOLTAGE_LIMIT
    input voltage limit programmed by charger. Indicates the voltage limit from a charging source.

INPUT_POWER_LIMIT
    input power limit programmed by charger. Indicates the power limit from a charging source.

CHARGE_CONTROL_LIMIT
    current charge control limit setting

CHARGE_CONTROL_LIMIT_MAX
    maximum charge control limit setting

CALIBRATE
    battery or coulomb counter calibration status

CAPACITY
    capacity in percents.

CAPACITY_ALERT_MIN
    minimum capacity alert value in percents.

CAPACITY_ALERT_MAX
    maximum capacity alert value in percents.

CAPACITY_LEVEL
    capacity level. This corresponds to POWER_SUPPLY_CAPACITY_LEVEL_*.

TEMP
    temperature of the power supply.

TEMP_ALERT_MIN
    minimum battery temperature alert.

TEMP_ALERT_MAX
    maximum battery temperature alert.

TEMP_AMBIENT
    ambient temperature.

TEMP_AMBIENT_ALERT_MIN
    minimum ambient temperature alert.

TEMP_AMBIENT_ALERT_MAX
    maximum ambient temperature alert.

TEMP_MIN
    minimum operatable temperature

TEMP_MAX
    maximum operatable temperature

TIME_TO_EMPTY
    seconds left for battery to be considered empty (i.e. while battery powers a load)

TIME_TO_FULL
    seconds left for battery to be considered full (i.e. while battery is charging)

10.5 Battery <-> external power supply interaction

Often power supplies are acting as supplies and supplicants at the same time. Batteries are good example. So, batteries usually care if they're externally powered or not.

For that case, power supply class implements notification mechanism for batteries.

External power supply (AC) lists supplicants (batteries) names in "supplied_to" struct member, and each power_supply_changed() call issued by external power supply will notify supplicants via external_power_changed callback.

10.6 Devicetree battery characteristics

Drivers should call power_supply_get_battery_info() to obtain battery characteristics from a devicetree battery node, defined in Documentation/devicetree/bindings/power/supply/battery.yaml. This is implemented in drivers/power/supply/bq27xxx_battery.c.

Properties in struct power_supply_battery_info and their counterparts in the battery node have names corresponding to elements in enum power_supply_property, for naming consistency between sysfs attributes and battery node properties.

10.7 QA

Q:

> Where is POWER_SUPPLY_PROP_XYZ attribute?

A:

> If you cannot find attribute suitable for your driver needs, feel free to add it and send patch along with your driver.
>
> The attributes available currently are the ones currently provided by the drivers written.
>
> Good candidates to add in future: model/part#, cycle_time, manufacturer, etc.

Q:

> I have some very specific attribute (e.g. battery color), should I add this attribute to standard ones?

A:

> Most likely, no. Such attribute can be placed in the driver itself, if it is useful. Of course, if the attribute in question applicable to large set of batteries, provided by many drivers, and/or comes from some general battery specification/standard, it may be a candidate to be added to the core attribute set.

Q:

> Suppose, my battery monitoring chip/firmware does not provides capacity in percents, but provides charge_{now,full,empty}. Should I calculate percentage capacity manually, inside the driver, and register CAPACITY attribute? The same question about time_to_empty/time_to_full.

A:

Most likely, no. This class is designed to export properties which are directly measurable by the specific hardware available.

Inferring not available properties using some heuristics or mathematical model is not subject of work for a battery driver. Such functionality should be factored out, and in fact, apm_power, the driver to serve legacy APM API on top of power supply class, uses a simple heuristic of approximating remaining battery capacity based on its charge, current, voltage and so on. But full-fledged battery model is likely not subject for kernel at all, as it would require floating point calculation to deal with things like differential equations and Kalman filters. This is better be handled by batteryd/libbattery, yet to be written.

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>

11.1 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.

11.2 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.

11.3 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;

11.3. 3. Runtime PM Device Fields

- 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 auto-suspend

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

11.4 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.

11.4. 4. Runtime PM Device Helper Functions 73

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()

11.5 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.

11.6 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).

11.7 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.

11.8 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.

HOW TO GET S2RAM WORKING

2006 Linus Torvalds 2006 Pavel Machek

1) Check suspend.sf.net, program s2ram there has long whitelist of "known ok" machines, along with tricks to use on each one.

2) If that does not help, try reading *swsusp/S3 tricks* and *Video issues with S3 resume*. Perhaps problem is as simple as broken module, and simple module unload can fix it.

3) You can use Linus' TRACE_RESUME infrastructure, described below.

12.1 Using TRACE_RESUME

I've been working at making the machines I have able to STR, and almost always it's a driver that is buggy. Thank God for the suspend/resume debugging - the thing that Chuck tried to disable. That's often the _only_ way to debug these things, and it's actually pretty powerful (but time-consuming - having to insert TRACE_RESUME() markers into the device driver that doesn't resume and recompile and reboot).

Anyway, the way to debug this for people who are interested (have a machine that doesn't boot) is:

- enable PM_DEBUG, and PM_TRACE

- use a script like this:

```
#!/bin/sh
sync
echo 1 > /sys/power/pm_trace
echo mem > /sys/power/state
```

   to suspend

- if it doesn't come back up (which is usually the problem), reboot by holding the power button down, and look at the dmesg output for things like:

```
Magic number: 4:156:725
hash matches drivers/base/power/resume.c:28
hash matches device 0000:01:00.0
```

   which means that the last trace event was just before trying to resume device 0000:01:00.0. Then figure out what driver is controlling that device (lspci and /sys/devices/pci* is your friend), and see if you can fix it, disable it, or trace into its resume function.

If no device matches the hash (or any matches appear to be false positives), the culprit may be a device from a loadable kernel module that is not loaded until after the hash is checked. You can check the hash against the current devices again after more modules are loaded using sysfs:

```
cat /sys/power/pm_trace_dev_match
```

For example, the above happens to be the VGA device on my EVO, which I used to run with "radeonfb" (it's an ATI Radeon mobility). It turns out that "radeonfb" simply cannot resume that device - it tries to set the PLL's, and it just _hangs_. Using the regular VGA console and letting X resume it instead works fine.

12.1.1 NOTE

pm_trace uses the system's Real Time Clock (RTC) to save the magic number. Reason for this is that the RTC is the only reliably available piece of hardware during resume operations where a value can be set that will survive a reboot.

pm_trace is not compatible with asynchronous suspend, so it turns asynchronous suspend off (which may work around timing or ordering-sensitive bugs).

Consequence is that after a resume (even if it is successful) your system clock will have a value corresponding to the magic number instead of the correct date/time! It is therefore advisable to use a program like ntp-date or rdate to reset the correct date/time from an external time source when using this trace option.

As the clock keeps ticking it is also essential that the reboot is done quickly after the resume failure. The trace option does not use the seconds or the low order bits of the minutes of the RTC, but a too long delay will corrupt the magic value.

INTERACTION OF SUSPEND CODE (S3) WITH THE CPU HOTPLUG INFRASTRUCTURE

(C) 2011 - 2014 Srivatsa S. Bhat <srivatsa.bhat@linux.vnet.ibm.com>

13.1 I. Differences between CPU hotplug and Suspend-to-RAM

How does the regular CPU hotplug code differ from how the Suspend-to-RAM infrastructure uses it internally? And where do they share common code?
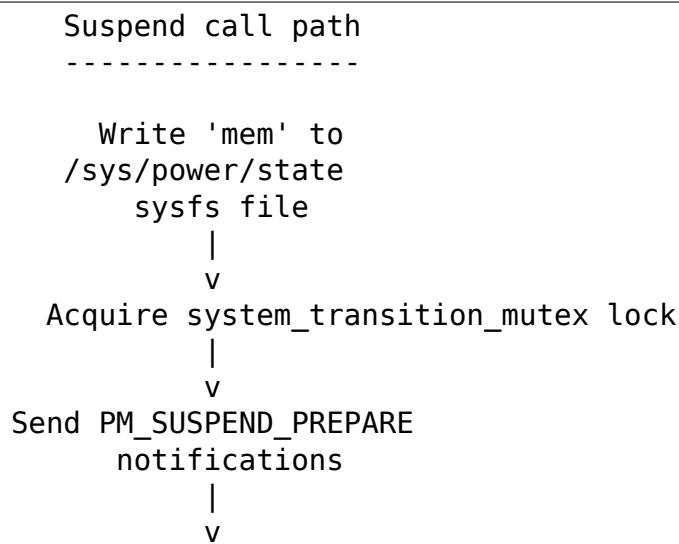
Well, a picture is worth a thousand words... So ASCII art follows :-)
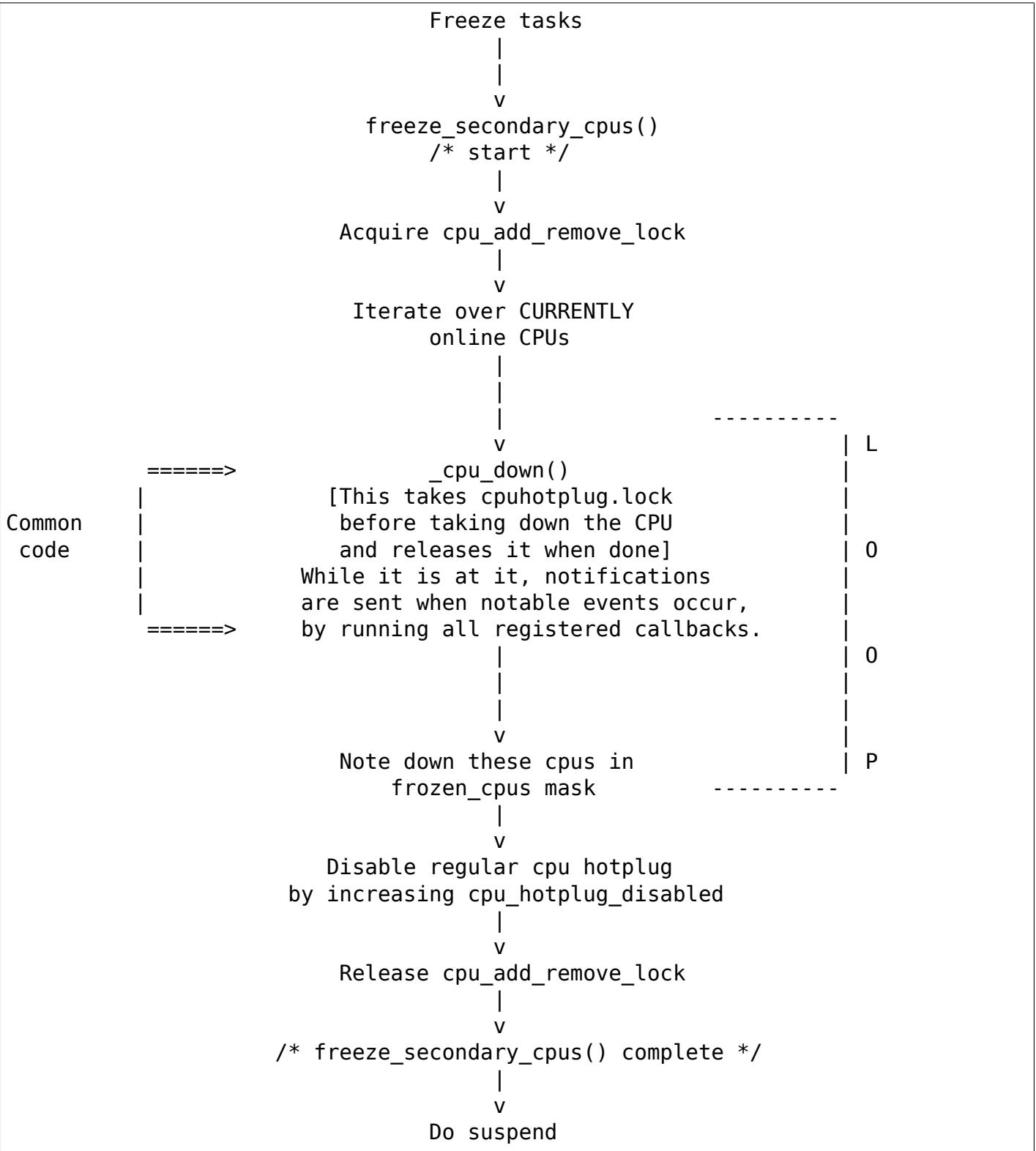
[This depicts the current design in the kernel, and focusses only on the interactions involving the freezer and CPU hotplug and also tries to explain the locking involved. It outlines the notifications involved as well. But please note that here, only the call paths are illustrated, with the aim of describing where they take different paths and where they share code. What happens when regular CPU hotplug and Suspend-to-RAM race with each other is not depicted here.]

On a high level, the suspend-resume cycle goes like this:

```
|Freeze|  -> |Disable nonboot|  -> |Do suspend|  -> |Enable nonboot|  -> |Thaw |
|tasks |     |     cpus       |     |          |     |     cpus      |     |tasks|
```

More details follow:

```
                    Suspend call path
                    -----------------

                      Write 'mem' to
                     /sys/power/state
                        sysfs file
                            |
                            v
                 Acquire system_transition_mutex lock
                            |
                            v
                 Send PM_SUSPEND_PREPARE
                       notifications
                            |
                            v
```

```
                              Freeze tasks
                                   |
                                   |
                                   v
                         freeze_secondary_cpus()
                             /* start */
                                   |
                                   v
                         Acquire cpu_add_remove_lock
                                   |
                                   v
                         Iterate over CURRENTLY
                             online CPUs
                                   |
                                   |
                                   |                   ----------
                                   v                            | L
            ======>              _cpu_down()                    |
            |               [This takes cpuhotplug.lock         |
   Common   |                 before taking down the CPU        |
    code    |                 and releases it when done]        | O
            |            While it is at it, notifications       |
            |            are sent when notable events occur,    |
            ======>       by running all registered callbacks.  |
                                   |                            | O
                                   |                            |
                                   |                            |
                                   v                            |
                         Note down these cpus in                | P
                             frozen_cpus mask          ----------
                                   |
                                   v
                         Disable regular cpu hotplug
                        by increasing cpu_hotplug_disabled
                                   |
                                   v
                         Release cpu_add_remove_lock
                                   |
                                   v
                   /* freeze_secondary_cpus() complete */
                                   |
                                   v
                              Do suspend
```

Resuming back is likewise, with the counterparts being (in the order of execution during resume):

- thaw_secondary_cpus() which involves:

```
|   Acquire cpu_add_remove_lock
|   Decrease cpu_hotplug_disabled, thereby enabling regular cpu hotplug
```
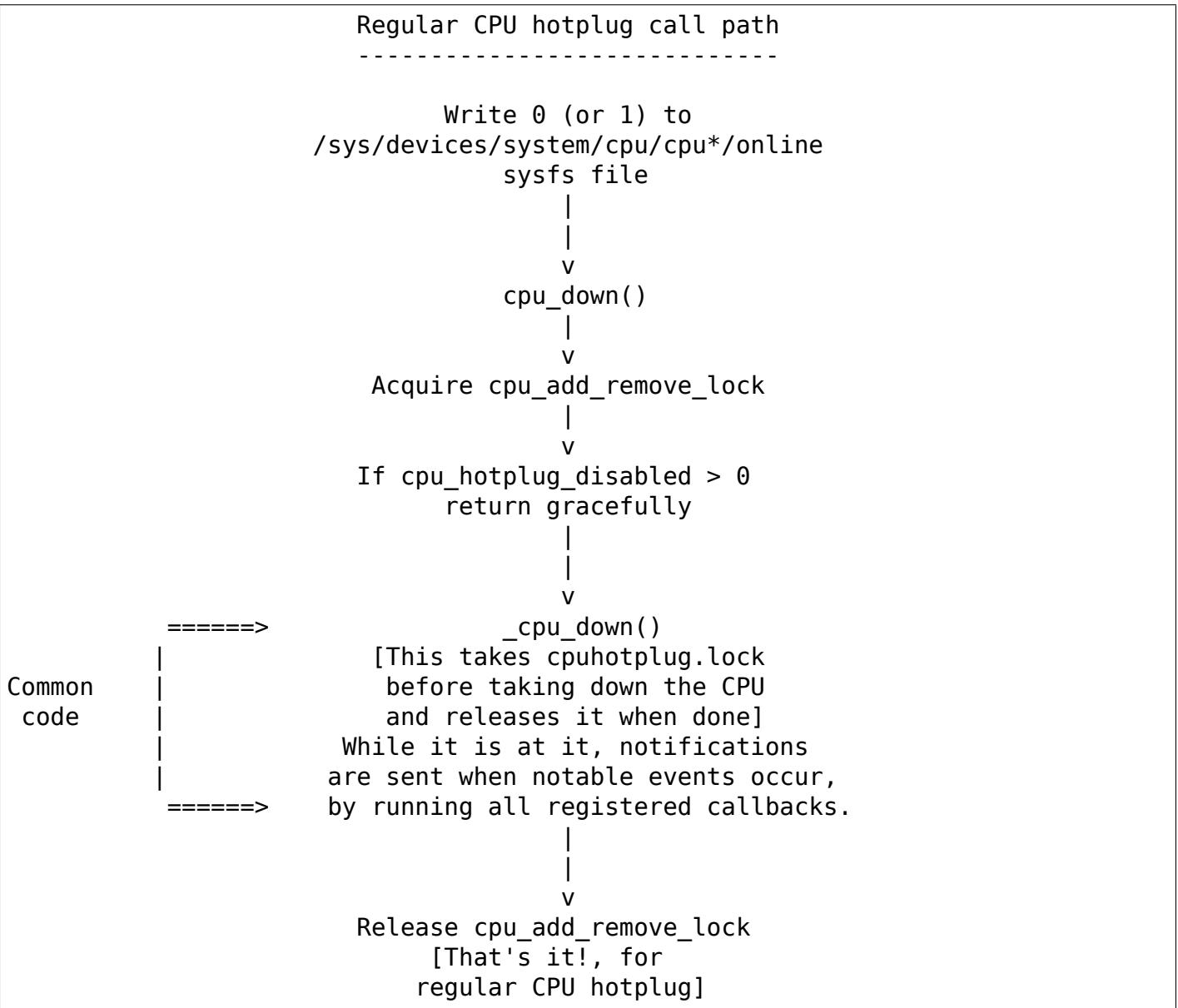
```
|   Call _cpu_up() [for all those cpus in the frozen_cpus mask, in a loop]
|   Release cpu_add_remove_lock
v
```

- thaw tasks

- send PM_POST_SUSPEND notifications

- Release system_transition_mutex lock.

It is to be noted here that the system_transition_mutex lock is acquired at the very beginning, when we are just starting out to suspend, and then released only after the entire cycle is complete (i.e., suspend + resume).

```
                    Regular CPU hotplug call path
                    ----------------------------

                        Write 0 (or 1) to
                /sys/devices/system/cpu/cpu*/online
                            sysfs file
                                |
                                |
                                v
                           cpu_down()
                                |
                                v
                    Acquire cpu_add_remove_lock
                                |
                                v
                    If cpu_hotplug_disabled > 0
                           return gracefully
                                |
                                |
                                v
         ======>                 _cpu_down()
            |              [This takes cpuhotplug.lock
Common      |               before taking down the CPU
 code       |               and releases it when done]
            |             While it is at it, notifications
            |            are sent when notable events occur,
         ======>         by running all registered callbacks.
                                |
                                |
                                v
                    Release cpu_add_remove_lock
                          [That's it!, for
                          regular CPU hotplug]
```

So, as can be seen from the two diagrams (the parts marked as "Common code"), regular CPU hotplug and the suspend code path converge at the _cpu_down() and _cpu_up() functions. They differ in the arguments passed to these functions, in that during regular CPU hotplug, 0 is passed for the 'tasks_frozen' argument. But during suspend, since the tasks are already frozen

by the time the non-boot CPUs are offlined or onlined, the _cpu_*() functions are called with the 'tasks_frozen' argument set to 1. [See below for some known issues regarding this.]

13.1.1 Important files and functions/entry points:

- kernel/power/process.c : freeze_processes(), thaw_processes()
- kernel/power/suspend.c : suspend_prepare(), suspend_enter(), suspend_finish()
- kernel/cpu.c: cpu_[up|down](), _cpu_[up|down](), [disable|enable]_nonboot_cpus()

13.1.2 II. What are the issues involved in CPU hotplug?

There are some interesting situations involving CPU hotplug and microcode update on the CPUs, as discussed below:

[Please bear in mind that the kernel requests the microcode images from userspace, using the request_firmware() function defined in drivers/base/firmware_loader/main.c]

a. When all the CPUs are identical:

This is the most common situation and it is quite straightforward: we want to apply the same microcode revision to each of the CPUs. To give an example of x86, the collect_cpu_info() function defined in arch/x86/kernel/microcode_core.c helps in discovering the type of the CPU and thereby in applying the correct microcode revision to it. But note that the kernel does not maintain a common microcode image for the all CPUs, in order to handle case 'b' described below.

b. When some of the CPUs are different than the rest:

In this case since we probably need to apply different microcode revisions to different CPUs, the kernel maintains a copy of the correct microcode image for each CPU (after appropriate CPU type/model discovery using functions such as collect_cpu_info()).

c. When a CPU is physically hot-unplugged and a new (and possibly different type of) CPU is hot-plugged into the system:

In the current design of the kernel, whenever a CPU is taken offline during a regular CPU hotplug operation, upon receiving the CPU_DEAD notification (which is sent by the CPU hotplug code), the microcode update driver's callback for that event reacts by freeing the kernel's copy of the microcode image for that CPU.

Hence, when a new CPU is brought online, since the kernel finds that it doesn't have the microcode image, it does the CPU type/model discovery afresh and then requests the userspace for the appropriate microcode image for that CPU, which is subsequently applied.

For example, in x86, the mc_cpu_callback() function (which is the microcode update driver's callback registered for CPU hotplug events) calls microcode_update_cpu() which would call microcode_init_cpu() in this case, instead of microcode_resume_cpu() when it finds that the kernel doesn't have a valid microcode image. This ensures that the CPU type/model discovery is performed and the right microcode is applied to the CPU after getting it from userspace.

d. Handling microcode update during suspend/hibernate:

Strictly speaking, during a CPU hotplug operation which does not involve physically removing or inserting CPUs, the CPUs are not actually powered off during a CPU offline. They are just put to the lowest C-states possible. Hence, in such a case, it is not really necessary to re-apply microcode when the CPUs are brought back online, since they wouldn't have lost the image during the CPU offline operation.

This is the usual scenario encountered during a resume after a suspend. However, in the case of hibernation, since all the CPUs are completely powered off, during restore it becomes necessary to apply the microcode images to all the CPUs.

[Note that we don't expect someone to physically pull out nodes and insert nodes with a different type of CPUs in-between a suspend-resume or a hibernate/restore cycle.]

In the current design of the kernel however, during a CPU offline operation as part of the suspend/hibernate cycle (cpuhp_tasks_frozen is set), the existing copy of microcode image in the kernel is not freed up. And during the CPU online operations (during resume/restore), since the kernel finds that it already has copies of the microcode images for all the CPUs, it just applies them to the CPUs, avoiding any re-discovery of CPU type/model and the need for validating whether the microcode revisions are right for the CPUs or not (due to the above assumption that physical CPU hotplug will not be done in-between suspend/resume or hibernate/restore cycles).

13.2 III. Known problems

Are there any known problems when regular CPU hotplug and suspend race with each other?

Yes, they are listed below:

1. When invoking regular CPU hotplug, the 'tasks_frozen' argument passed to the _cpu_down() and _cpu_up() functions is *always* 0. This might not reflect the true current state of the system, since the tasks could have been frozen by an out-of-band event such as a suspend operation in progress. Hence, the cpuhp_tasks_frozen variable will not reflect the frozen state and the CPU hotplug callbacks which evaluate that variable might execute the wrong code path.

2. If a regular CPU hotplug stress test happens to race with the freezer due to a suspend operation in progress at the same time, then we could hit the situation described below:

   - A regular cpu online operation continues its journey from userspace into the kernel, since the freezing has not yet begun.

   - Then freezer gets to work and freezes userspace.

   - If cpu online has not yet completed the microcode update stuff by now, it will now start waiting on the frozen userspace in the TASK_UNINTERRUPTIBLE state, in order to get the microcode image.

   - Now the freezer continues and tries to freeze the remaining tasks. But due to this wait mentioned above, the freezer won't be able to freeze the cpu online hotplug task and hence freezing of tasks fails.

As a result of this task freezing failure, the suspend operation gets aborted.

SYSTEM SUSPEND AND DEVICE INTERRUPTS

Copyright (C) 2014 Intel Corp. Author: Rafael J. Wysocki <rafael.j.wysocki@intel.com>

14.1 Suspending and Resuming Device IRQs

Device interrupt request lines (IRQs) are generally disabled during system suspend after the "late" phase of suspending devices (that is, after all of the ->prepare, ->suspend and ->suspend_late callbacks have been executed for all devices). That is done by suspend_device_irqs().

The rationale for doing so is that after the "late" phase of device suspend there is no legitimate reason why any interrupts from suspended devices should trigger and if any devices have not been suspended properly yet, it is better to block interrupts from them anyway. Also, in the past we had problems with interrupt handlers for shared IRQs that device drivers implementing them were not prepared for interrupts triggering after their devices had been suspended. In some cases they would attempt to access, for example, memory address spaces of suspended devices and cause unpredictable behavior to ensue as a result. Unfortunately, such problems are very difficult to debug and the introduction of suspend_device_irqs(), along with the "noirq" phase of device suspend and resume, was the only practical way to mitigate them.

Device IRQs are re-enabled during system resume, right before the "early" phase of resuming devices (that is, before starting to execute ->resume_early callbacks for devices). The function doing that is resume_device_irqs().

14.2 The IRQF_NO_SUSPEND Flag

There are interrupts that can legitimately trigger during the entire system suspend-resume cycle, including the "noirq" phases of suspending and resuming devices as well as during the time when nonboot CPUs are taken offline and brought back online. That applies to timer interrupts in the first place, but also to IPIs and to some other special-purpose interrupts.

The IRQF_NO_SUSPEND flag is used to indicate that to the IRQ subsystem when requesting a special-purpose interrupt. It causes suspend_device_irqs() to leave the corresponding IRQ enabled so as to allow the interrupt to work as expected during the suspend-resume cycle, but does not guarantee that the interrupt will wake the system from a suspended state -- for such cases it is necessary to use enable_irq_wake().

Note that the IRQF_NO_SUSPEND flag affects the entire IRQ and not just one user of it. Thus, if the IRQ is shared, all of the interrupt handlers installed for it will be executed as usual after

suspend_device_irqs(), even if the IRQF_NO_SUSPEND flag was not passed to request_irq() (or equivalent) by some of the IRQ's users. For this reason, using IRQF_NO_SUSPEND and IRQF_SHARED at the same time should be avoided.

14.3 System Wakeup Interrupts, enable_irq_wake() and disable_irq_wake()

System wakeup interrupts generally need to be configured to wake up the system from sleep states, especially if they are used for different purposes (e.g. as I/O interrupts) in the working state.

That may involve turning on a special signal handling logic within the platform (such as an SoC) so that signals from a given line are routed in a different way during system sleep so as to trigger a system wakeup when needed. For example, the platform may include a dedicated interrupt controller used specifically for handling system wakeup events. Then, if a given interrupt line is supposed to wake up the system from sleep states, the corresponding input of that interrupt controller needs to be enabled to receive signals from the line in question. After wakeup, it generally is better to disable that input to prevent the dedicated controller from triggering interrupts unnecessarily.

The IRQ subsystem provides two helper functions to be used by device drivers for those purposes. Namely, enable_irq_wake() turns on the platform's logic for handling the given IRQ as a system wakeup interrupt line and disable_irq_wake() turns that logic off.

Calling enable_irq_wake() causes suspend_device_irqs() to treat the given IRQ in a special way. Namely, the IRQ remains enabled, by on the first interrupt it will be disabled, marked as pending and "suspended" so that it will be re-enabled by resume_device_irqs() during the subsequent system resume. Also the PM core is notified about the event which causes the system suspend in progress to be aborted (that doesn't have to happen immediately, but at one of the points where the suspend thread looks for pending wakeup events).

This way every interrupt from a wakeup interrupt source will either cause the system suspend currently in progress to be aborted or wake up the system if already suspended. However, after suspend_device_irqs() interrupt handlers are not executed for system wakeup IRQs. They are only executed for IRQF_NO_SUSPEND IRQs at that time, but those IRQs should not be configured for system wakeup using enable_irq_wake().

14.4 Interrupts and Suspend-to-Idle

Suspend-to-idle (also known as the "freeze" sleep state) is a relatively new system sleep state that works by idling all of the processors and waiting for interrupts right after the "noirq" phase of suspending devices.

Of course, this means that all of the interrupts with the IRQF_NO_SUSPEND flag set will bring CPUs out of idle while in that state, but they will not cause the IRQ subsystem to trigger a system wakeup.

System wakeup interrupts, in turn, will trigger wakeup from suspend-to-idle in analogy with what they do in the full system suspend case. The only difference is that the wakeup from suspend-to-idle is signaled using the usual working state interrupt delivery mechanisms and doesn't require the platform to use any special interrupt handling logic for it to work.

14.5 IRQF_NO_SUSPEND and enable_irq_wake()

There are very few valid reasons to use both enable_irq_wake() and the IRQF_NO_SUSPEND flag on the same IRQ, and it is never valid to use both for the same device.

First of all, if the IRQ is not shared, the rules for handling IRQF_NO_SUSPEND interrupts (interrupt handlers are invoked after suspend_device_irqs()) are directly at odds with the rules for handling system wakeup interrupts (interrupt handlers are not invoked after suspend_device_irqs()).

Second, both enable_irq_wake() and IRQF_NO_SUSPEND apply to entire IRQs and not to individual interrupt handlers, so sharing an IRQ between a system wakeup interrupt source and an IRQF_NO_SUSPEND interrupt source does not generally make sense.

In rare cases an IRQ can be shared between a wakeup device driver and an IRQF_NO_SUSPEND user. In order for this to be safe, the wakeup device driver must be able to discern spurious IRQs from genuine wakeup events (signalling the latter to the core with pm_system_wakeup()), must use enable_irq_wake() to ensure that the IRQ will function as a wakeup source, and must request the IRQ with IRQF_COND_SUSPEND to tell the core that it meets these requirements. If these requirements are not met, it is not valid to use IRQF_COND_SUSPEND.

USING SWAP FILES WITH SOFTWARE SUSPEND (SWSUSP)

(C) 2006 Rafael J. Wysocki <rjw@sisk.pl>

The Linux kernel handles swap files almost in the same way as it handles swap partitions and there are only two differences between these two types of swap areas: (1) swap files need not be contiguous, (2) the header of a swap file is not in the first block of the partition that holds it. From the swsusp's point of view (1) is not a problem, because it is already taken care of by the swap-handling code, but (2) has to be taken into consideration.

In principle the location of a swap file's header may be determined with the help of appropriate filesystem driver. Unfortunately, however, it requires the filesystem holding the swap file to be mounted, and if this filesystem is journaled, it cannot be mounted during resume from disk. For this reason to identify a swap file swsusp uses the name of the partition that holds the file and the offset from the beginning of the partition at which the swap file's header is located. For convenience, this offset is expressed in <PAGE_SIZE> units.

In order to use a swap file with swsusp, you need to:

1) Create the swap file and make it active, eg.:

```
# dd if=/dev/zero of=<swap_file_path> bs=1024 count=<swap_file_size_in_k>
# mkswap <swap_file_path>
# swapon <swap_file_path>
```

2) Use an application that will bmap the swap file with the help of the FIBMAP ioctl and determine the location of the file's swap header, as the offset, in <PAGE_SIZE> units, from the beginning of the partition which holds the swap file.

3) Add the following parameters to the kernel command line:

```
resume=<swap_file_partition> resume_offset=<swap_file_offset>
```

where <swap_file_partition> is the partition on which the swap file is located and <swap_file_offset> is the offset of the swap header determined by the application in 2) (of course, this step may be carried out automatically by the same application that determines the swap file's header offset using the FIBMAP ioctl)

OR

Use a userland suspend application that will set the partition and offset with the help of the SNAPSHOT_SET_SWAP_AREA ioctl described in *Documentation for userland software suspend interface* (this is the only method to suspend to a swap file allowing the resume to be initiated from an initrd or initramfs image).

Now, swsusp will use the swap file in the same way in which it would use a swap partition. In particular, the swap file has to be active (ie. be present in /proc/swaps) so that it can be used for suspending.

Note that if the swap file used for suspending is deleted and recreated, the location of its header need not be the same as before. Thus every time this happens the value of the "resume_offset=" kernel command line parameter has to be updated.

HOW TO USE DM-CRYPT AND SWSUSP TOGETHER

Author: Andreas Steinmetz <ast@domdv.de>

Some prerequisites: You know how dm-crypt works. If not, visit the following web page: http://www.saout.de/misc/dm-crypt/ You have read *Swap suspend* and understand it. You did read Documentation/admin-guide/initrd.rst and know how an initrd works. You know how to create or how to modify an initrd.

Now your system is properly set up, your disk is encrypted except for the swap device(s) and the boot partition which may contain a mini system for crypto setup and/or rescue purposes. You may even have an initrd that does your current crypto setup already.

At this point you want to encrypt your swap, too. Still you want to be able to suspend using swsusp. This, however, means that you have to be able to either enter a passphrase or that you read the key(s) from an external device like a pcmcia flash disk or an usb stick prior to resume. So you need an initrd, that sets up dm-crypt and then asks swsusp to resume from the encrypted swap device.

The most important thing is that you set up dm-crypt in such a way that the swap device you suspend to/resume from has always the same major/minor within the initrd as well as within your running system. The easiest way to achieve this is to always set up this swap device first with dmsetup, so that it will always look like the following:

```
brw-------  1 root root 254, 0 Jul 28 13:37 /dev/mapper/swap0
```

Now set up your kernel to use /dev/mapper/swap0 as the default resume partition, so your kernel .config contains:

```
CONFIG_PM_STD_PARTITION="/dev/mapper/swap0"
```

Prepare your boot loader to use the initrd you will create or modify. For lilo the simplest setup looks like the following lines:

```
image=/boot/vmlinuz
initrd=/boot/initrd.gz
label=linux
append="root=/dev/ram0 init=/linuxrc rw"
```

Finally you need to create or modify your initrd. Lets assume you create an initrd that reads the required dm-crypt setup from a pcmcia flash disk card. The card is formatted with an ext2 fs which resides on /dev/hde1 when the card is inserted. The card contains at least the encrypted swap setup in a file named "swapkey". /etc/fstab of your initrd contains something like the following:

```
/dev/hda1    /mnt    ext3    ro                              0 0
none         /proc   proc    defaults,noatime,nodiratime     0 0
none         /sys    sysfs   defaults,noatime,nodiratime     0 0
```

/dev/hda1 contains an unencrypted mini system that sets up all of your crypto devices, again by reading the setup from the pcmcia flash disk. What follows now is a /linuxrc for your initrd that allows you to resume from encrypted swap and that continues boot with your mini system on /dev/hda1 if resume does not happen:

```
#!/bin/sh
PATH=/sbin:/bin:/usr/sbin:/usr/bin
mount /proc
mount /sys
mapped=0
noresume=`grep -c noresume /proc/cmdline`
if [ "$*" != "" ]
then
  noresume=1
fi
dmesg -n 1
/sbin/cardmgr -q
for i in 1 2 3 4 5 6 7 8 9 0
do
  if [ -f /proc/ide/hde/media ]
  then
    usleep 500000
    mount -t ext2 -o ro /dev/hde1 /mnt
    if [ -f /mnt/swapkey ]
    then
      dmsetup create swap0 /mnt/swapkey > /dev/null 2>&1 && mapped=1
    fi
    umount /mnt
    break
  fi
  usleep 500000
done
killproc /sbin/cardmgr
dmesg -n 6
if [ $mapped = 1 ]
then
  if [ $noresume != 0 ]
  then
    mkswap /dev/mapper/swap0 > /dev/null 2>&1
  fi
  echo 254:0 > /sys/power/resume
  dmsetup remove swap0
fi
umount /sys
mount /mnt
umount /proc
cd /mnt
```

```
pivot_root . mnt
mount /proc
umount -l /mnt
umount /proc
exec chroot . /sbin/init $* < dev/console > dev/console 2>&1
```

Please don't mind the weird loop above, busybox's msh doesn't know the let statement. Now, what is happening in the script? First we have to decide if we want to try to resume, or not. We will not resume if booting with "noresume" or any parameters for init like "single" or "emergency" as boot parameters.

Then we need to set up dmcrypt with the setup data from the pcmcia flash disk. If this succeeds we need to reset the swap device if we don't want to resume. The line "echo 254:0 > /sys/power/resume" then attempts to resume from the first device mapper device. Note that it is important to set the device in /sys/power/resume, regardless if resuming or not, otherwise later suspend will fail. If resume starts, script execution terminates here.

Otherwise we just remove the encrypted swap device and leave it to the mini system on /dev/hda1 to set the whole crypto up (it is up to you to modify this to your taste).

What then follows is the well known process to change the root file system and continue booting from there. I prefer to unmount the initrd prior to continue booting but it is up to you to modify this.

SWAP SUSPEND

Some warnings, first.

> **Warning: BIG FAT WARNING**
>
> **If you touch anything on disk between suspend and resume...**
>   ...kiss your data goodbye.
>
> **If you do resume from initrd after your filesystems are mounted...**
>
>   ...bye bye root partition.
>
>   [this is actually same case as above]
>
> If you have unsupported ( ) devices using DMA, you may have some problems. If your disk driver does not support suspend... (IDE does), it may cause some problems, too. If you change kernel command line between suspend and resume, it may do something wrong. If you change your hardware while system is suspended... well, it was not good idea; but it will probably only crash.
>
> ( ) suspend/resume support is needed to make it safe.
>
> If you have any filesystems on USB devices mounted before software suspend, they won't be accessible after resume and you may lose data, as though you have unplugged the USB devices with mounted filesystems on them; see the FAQ below for details. (This is not true for more traditional power states like "standby", which normally don't turn USB off.)

Swap partition:
    You need to append resume=/dev/your_swap_partition to kernel command line or specify it using /sys/power/resume.

Swap file:
    If using a swapfile you can also specify a resume offset using resume_offset=<number> on the kernel command line or specify it in /sys/power/resume_offset.

After preparing then you suspend by:

```
echo shutdown > /sys/power/disk; echo disk > /sys/power/state
```

- If you feel ACPI works pretty well on your system, you might try:

```
echo platform > /sys/power/disk; echo disk > /sys/power/state
```

- If you would like to write hibernation image to swap and then suspend to RAM (provided your platform supports it), you can try:

```
echo suspend > /sys/power/disk; echo disk > /sys/power/state
```

- If you have SATA disks, you'll need recent kernels with SATA suspend support. For suspend and resume to work, make sure your disk drivers are built into kernel -- not modules. [There's way to make suspend/resume with modular disk drivers, see FAQ, but you probably should not do that.]

If you want to limit the suspend image size to N bytes, do:

```
echo N > /sys/power/image_size
```

before suspend (it is limited to around 2/5 of available RAM by default).

- The resume process checks for the presence of the resume device, if found, it then checks the contents for the hibernation image signature. If both are found, it resumes the hibernation image.

- The resume process may be triggered in two ways:

  1) During lateinit: If resume=/dev/your_swap_partition is specified on the kernel command line, lateinit runs the resume process. If the resume device has not been probed yet, the resume process fails and bootup continues.

  2) Manually from an initrd or initramfs: May be run from the init script by using the /sys/power/resume file. It is vital that this be done prior to remounting any filesystems (even as read-only) otherwise data may be corrupted.

17.1 Article about goals and implementation of Software Suspend for Linux

Author: Gábor Kuti Last revised: 2003-10-20 by Pavel Machek

17.1.1 Idea and goals to achieve

Nowadays it is common in several laptops that they have a suspend button. It saves the state of the machine to a filesystem or to a partition and switches to standby mode. Later resuming the machine the saved state is loaded back to ram and the machine can continue its work. It has two real benefits. First we save ourselves the time machine goes down and later boots up, energy costs are real high when running from batteries. The other gain is that we don't have to interrupt our programs so processes that are calculating something for a long time shouldn't need to be written interruptible.

swsusp saves the state of the machine into active swaps and then reboots or powerdowns. You must explicitly specify the swap partition to resume from with *resume*= kernel option. If signature is found it loads and restores saved state. If the option *noresume* is specified as a boot parameter, it skips the resuming. If the option *hibernate=nocompress* is specified as a boot parameter, it saves hibernation image without compression.

In the meantime while the system is suspended you should not add/remove any of the hardware, write to the filesystems, etc.

17.2 Sleep states summary

There are three different interfaces you can use, /proc/acpi should work like this:

In a really perfect world:

```
echo 1 > /proc/acpi/sleep        # for standby
echo 2 > /proc/acpi/sleep        # for suspend to ram
echo 3 > /proc/acpi/sleep        # for suspend to ram, but with more power
                                 # conservative
echo 4 > /proc/acpi/sleep        # for suspend to disk
echo 5 > /proc/acpi/sleep        # for shutdown unfriendly the system
```

and perhaps:

```
echo 4b > /proc/acpi/sleep       # for suspend to disk via s4bios
```

17.3 Frequently Asked Questions

Q:

> well, suspending a server is IMHO a really stupid thing, but... (Diego Zuccato):

A:

> You bought new UPS for your server. How do you install it without bringing machine down? Suspend to disk, rearrange power cables, resume.
>
> You have your server on UPS. Power died, and UPS is indicating 30 seconds to failure. What do you do? Suspend to disk.

Q:

> Maybe I'm missing something, but why don't the regular I/O paths work?

A:

> We do use the regular I/O paths. However we cannot restore the data to its original location as we load it. That would create an inconsistent kernel state which would certainly result in an oops. Instead, we load the image into unused memory and then atomically copy it back to it original location. This implies, of course, a maximum image size of half the amount of memory.
>
> There are two solutions to this:
>
> - require half of memory to be free during suspend. That way you can read "new" data onto free spots, then cli and copy
>
> - assume we had special "polling" ide driver that only uses memory between 0-640KB. That way, I'd have to make sure that 0-640KB is free during suspending, but otherwise it would work...
>
> suspend2 shares this fundamental limitation, but does not include user data and disk caches into "used memory" by saving them in advance. That means that the limitation goes away in practice.

Q:

> Does linux support ACPI S4?

A:

Yes. That's what echo platform > /sys/power/disk does.

Q:

What is 'suspend2'?

A:

suspend2 is 'Software Suspend 2', a forked implementation of suspend-to-disk which is available as separate patches for 2.4 and 2.6 kernels from swsusp.sourceforge.net. It includes support for SMP, 4GB highmem and preemption. It also has a extensible architecture that allows for arbitrary transformations on the image (compression, encryption) and arbitrary backends for writing the image (eg to swap or an NFS share[Work In Progress]). Questions regarding suspend2 should be sent to the mailing list available through the suspend2 website, and not to the Linux Kernel Mailing List. We are working toward merging suspend2 into the mainline kernel.

Q:

What is the freezing of tasks and why are we using it?

A:

The freezing of tasks is a mechanism by which user space processes and some kernel threads are controlled during hibernation or system-wide suspend (on some architectures). See *Freezing of tasks* for details.

Q:

What is the difference between "platform" and "shutdown"?

A:

shutdown:
    save state in linux, then tell bios to powerdown

platform:
    save state in linux, then tell bios to powerdown and blink "suspended led"

"platform" is actually right thing to do where supported, but "shutdown" is most reliable (except on ACPI systems).

Q:

I do not understand why you have such strong objections to idea of selective suspend.

A:

Do selective suspend during runtime power management, that's okay. But it's useless for suspend-to-disk. (And I do not see how you could use it for suspend-to-ram, I hope you do not want that).

Lets see, so you suggest to

- SUSPEND all but swap device and parents

- Snapshot

- Write image to disk

- SUSPEND swap device and parents

- Powerdown

Oh no, that does not work, if swap device or its parents uses DMA, you've corrupted data. You'd have to do

- SUSPEND all but swap device and parents

- FREEZE swap device and parents

- Snapshot

- UNFREEZE swap device and parents

- Write

- SUSPEND swap device and parents

Which means that you still need that FREEZE state, and you get more complicated code. (And I have not yet introduce details like system devices).

Q:

There don't seem to be any generally useful behavioral distinctions between SUSPEND and FREEZE.

A:

Doing SUSPEND when you are asked to do FREEZE is always correct, but it may be unnecessarily slow. If you want your driver to stay simple, slowness may not matter to you. It can always be fixed later.

For devices like disk it does matter, you do not want to spindown for FREEZE.

Q:

After resuming, system is paging heavily, leading to very bad interactivity.

A:

Try running:

```
cat /proc/[0-9]*/maps | grep / | sed 's:.* /:/:' | sort -u | while read␣
 ↪file
do
  test -f "$file" && cat "$file" > /dev/null
done
```

after resume. swapoff -a; swapon -a may also be useful.

Q:

What happens to devices during swsusp? They seem to be resumed during system suspend?

A:

That's correct. We need to resume them if we want to write image to disk. Whole sequence goes like

Suspend part

running system, user asks for suspend-to-disk

user processes are stopped

suspend(PMSG_FREEZE): devices are frozen so that they don't interfere with state snapshot

state snapshot: copy of whole used memory is taken with interrupts disabled

resume(): devices are woken up so that we can write image to swap

write image to swap

suspend(PMSG_SUSPEND): suspend devices so that we can power off

turn the power off

Resume part

(is actually pretty similar)

running system, user asks for suspend-to-disk

user processes are stopped (in common case there are none, but with resume-from-initrd, no one knows)

read image from disk

suspend(PMSG_FREEZE): devices are frozen so that they don't interfere with image restoration

image restoration: rewrite memory with image

resume(): devices are woken up so that system can continue

thaw all user processes

Q:

What is this 'Encrypt suspend image' for?

A:

First of all: it is not a replacement for dm-crypt encrypted swap. It cannot protect your computer while it is suspended. Instead it does protect from leaking sensitive data after resume from suspend.

Think of the following: you suspend while an application is running that keeps sensitive data in memory. The application itself prevents the data from being swapped out. Suspend, however, must write these data to swap to be able to resume later on. Without suspend encryption your sensitive data are then stored in plaintext on disk. This means that after resume your sensitive data are accessible to all applications having direct access to the swap device which was used for suspend. If you don't need swap after resume these data can remain on disk virtually forever. Thus it can happen that your system gets broken in weeks later and sensitive data which you thought were encrypted and protected are retrieved and stolen from the swap device. To prevent this situation you should use 'Encrypt suspend image'.

During suspend a temporary key is created and this key is used to encrypt the data written to disk. When, during resume, the data was read back into memory the temporary key is destroyed which simply means that all data written to disk during suspend are then inaccessible so they can't be stolen later on. The only thing that you must then take care of is that you call 'mkswap' for the swap partition used for suspend as early as possible during regular boot. This asserts that any temporary key from an oopsed suspend or from a failed or aborted resume is erased from the swap device.

As a rule of thumb use encrypted swap to protect your data while your system is shut down or suspended. Additionally use the encrypted suspend image to prevent sensitive data from being stolen after resume.

Q:

Can I suspend to a swap file?

A:

Generally, yes, you can. However, it requires you to use the "resume=" and "resume_offset=" kernel command line parameters, so the resume from a swap file cannot be initiated from an initrd or initramfs image. See *Using swap files with software suspend (swsusp)* for details.

Q:

Is there a maximum system RAM size that is supported by swsusp?

A:

It should work okay with highmem.

Q:

Does swsusp (to disk) use only one swap partition or can it use multiple swap partitions (aggregate them into one logical space)?

A:

Only one swap partition, sorry.

Q:

If my application(s) causes lots of memory & swap space to be used (over half of the total system RAM), is it correct that it is likely to be useless to try to suspend to disk while that app is running?

A:

No, it should work okay, as long as your app does not mlock() it. Just prepare big enough swap partition.

Q:

What information is useful for debugging suspend-to-disk problems?

A:

Well, last messages on the screen are always useful. If something is broken, it is usually some kernel driver, therefore trying with as little as possible modules loaded helps a lot. I also prefer people to suspend from console, preferably without X running. Booting with init=/bin/bash, then swapon and starting suspend sequence manually usually does the trick. Then it is good idea to try with latest vanilla kernel.

Q:

How can distributions ship a swsusp-supporting kernel with modular disk drivers (especially SATA)?

A:

Well, it can be done, load the drivers, then do echo into /sys/power/resume file from initrd. Be sure not to mount anything, not even read-only mount, or you are going to lose your data.

Q:

How do I make suspend more verbose?

A:

If you want to see any non-error kernel messages on the virtual terminal the kernel switches to during suspend, you have to set the kernel console loglevel to at least 4 (KERN_WARNING), for example by doing:

```
# save the old loglevel
read LOGLEVEL DUMMY < /proc/sys/kernel/printk
```

```
# set the loglevel so we see the progress bar.
# if the level is higher than needed, we leave it alone.
if [ $LOGLEVEL -lt 5 ]; then
        echo 5 > /proc/sys/kernel/printk
        fi

IMG_SZ=0
read IMG_SZ < /sys/power/image_size
echo -n disk > /sys/power/state
RET=$?
#
# the logic here is:
# if image_size > 0 (without kernel support, IMG_SZ will be zero),
# then try again with image_size set to zero.
if [ $RET -ne 0 -a $IMG_SZ -ne 0 ]; then # try again with minimal image␣
 ↪size
        echo 0 > /sys/power/image_size
        echo -n disk > /sys/power/state
        RET=$?
fi

# restore previous loglevel
echo $LOGLEVEL > /proc/sys/kernel/printk
exit $RET
```

Q:

Is this true that if I have a mounted filesystem on a USB device and I suspend to disk, I can lose data unless the filesystem has been mounted with "sync"?

A:

That's right ... if you disconnect that device, you may lose data. In fact, even with "-o sync" you can lose data if your programs have information in buffers they haven't written out to a disk you disconnect, or if you disconnect before the device finished saving data you wrote.

Software suspend normally powers down USB controllers, which is equivalent to disconnecting all USB devices attached to your system.

Your system might well support low-power modes for its USB controllers while the system is asleep, maintaining the connection, using true sleep modes like "suspend-to-RAM" or "standby". (Don't write "disk" to the /sys/power/state file; write "standby" or "mem".) We've not seen any hardware that can use these modes through software suspend, although in theory some systems might support "platform" modes that won't break the USB connections.

Remember that it's always a bad idea to unplug a disk drive containing a mounted filesystem. That's true even when your system is asleep! The safest thing is to unmount all filesystems on removable media (such USB, Firewire, CompactFlash, MMC, external SATA, or even IDE hotplug bays) before suspending; then remount them after resuming.

There is a work-around for this problem. For more information, see Documentation/driver-api/usb/persist.rst.

Q:

Can I suspend-to-disk using a swap partition under LVM?

A:

Yes and No. You can suspend successfully, but the kernel will not be able to resume on its own. You need an initramfs that can recognize the resume situation, activate the logical volume containing the swap volume (but not touch any filesystems!), and eventually call:

```
echo -n "$major:$minor" > /sys/power/resume
```

where $major and $minor are the respective major and minor device numbers of the swap volume.

uswsusp works with LVM, too. See http://suspend.sourceforge.net/

Q:

I upgraded the kernel from 2.6.15 to 2.6.16. Both kernels were compiled with the similar configuration files. Anyway I found that suspend to disk (and resume) is much slower on 2.6.16 compared to 2.6.15. Any idea for why that might happen or how can I speed it up?

A:

This is because the size of the suspend image is now greater than for 2.6.15 (by saving more data we can get more responsive system after resume).

There's the /sys/power/image_size knob that controls the size of the image. If you set it to 0 (eg. by echo 0 > /sys/power/image_size as root), the 2.6.15 behavior should be restored. If it is still too slow, take a look at suspend.sf.net -- userland suspend is faster and supports LZF compression to speed it up further.

VIDEO ISSUES WITH S3 RESUME

2003-2006, Pavel Machek

During S3 resume, hardware needs to be reinitialized. For most devices, this is easy, and kernel driver knows how to do it. Unfortunately there's one exception: video card. Those are usually initialized by BIOS, and kernel does not have enough information to boot video card. (Kernel usually does not even contain video card driver -- vesafb and vgacon are widely used).

This is not problem for swsusp, because during swsusp resume, BIOS is run normally so video card is normally initialized. It should not be problem for S1 standby, because hardware should retain its state over that.

We either have to run video BIOS during early resume, or interpret it using vbetool later, or maybe nothing is necessary on particular system because video state is preserved. Unfortunately different methods work on different systems, and no known method suits all of them.

Userland application called s2ram has been developed; it contains long whitelist of systems, and automatically selects working method for a given system. It can be downloaded from CVS at www.sf.net/projects/suspend . If you get a system that is not in the whitelist, please try to find a working solution, and submit whitelist entry so that work does not need to be repeated.

Currently, VBE_SAVE method (6 below) works on most systems. Unfortunately, vbetool only runs after userland is resumed, so it makes debugging of early resume problems hard/impossible. Methods that do not rely on userland are preferable.

18.1 Details

There are a few types of systems where video works after S3 resume:

(1) systems where video state is preserved over S3.

(2) systems where it is possible to call the video BIOS during S3 resume. Unfortunately, it is not correct to call the video BIOS at that point, but it happens to work on some machines. Use acpi_sleep=s3_bios.

(3) systems that initialize video card into vga text mode and where the BIOS works well enough to be able to set video mode. Use acpi_sleep=s3_mode on these.

(4) on some systems s3_bios kicks video into text mode, and acpi_sleep=s3_bios,s3_mode is needed.

(5) radeon systems, where X can soft-boot your video card. You'll need a new enough X, and a plain text console (no vesafb or radeonfb). See http://www.doesi.gmxhome.de/linux/tm800s3/s3.html for more information. Alternatively, you should use vbetool (6) instead.

(6) other radeon systems, where vbetool is enough to bring system back to life. It needs text console to be working. Do vbetool vbestate save > /tmp/delme; echo 3 > /proc/acpi/sleep; vbetool post; vbetool vbestate restore < /tmp/delme; setfont <whatever>, and your video should work.

(7) on some systems, it is possible to boot most of kernel, and then POSTing bios works. Ole Rohne has patch to do just that at http://dev.gentoo.org/~marineam/patch-radeonfb-2.6.11-rc2-mm2.

(8) on some systems, you can use the video_post utility and or do echo 3 > /sys/power/state && /usr/sbin/video_post - which will initialize the display in console mode. If you are in X, you can switch to a virtual terminal and back to X using CTRL+ALT+F1 - CTRL+ALT+F7 to get the display working in graphical mode again.

Now, if you pass acpi_sleep=something, and it does not work with your bios, you'll get a hard crash during resume. Be careful. Also it is safest to do your experiments with plain old VGA console. The vesafb and radeonfb (etc) drivers have a tendency to crash the machine during resume.

You may have a system where none of above works. At that point you either invent another ugly hack that works, or write proper driver for your video card (good luck getting docs :-(). Maybe suspending from X (proper X, knowing your hardware, not XF68_FBcon) might have better chance of working.

Table of known working notebooks:

Model	hack (or "how to do it")
Acer Aspire 1406LC	ole's late BIOS init (7), turn off DRI
Acer TM 230	s3_bios (2)
Acer TM 242FX	vbetool (6)
Acer TM C110	video_post (8)
Acer TM C300	vga=normal (only suspend on console, not in X), vbetool (6) or video_post (8)
Acer TM 4052LCi	s3_bios (2)
Acer TM 636Lci	s3_bios,s3_mode (4)
Acer TM 650 (Radeon M7)	vga=normal plus boot-radeon (5) gets text console back
Acer TM 660	???[1]
Acer TM 800	vga=normal, X patches, see webpage (5) or vbetool (6)
Acer TM 803	vga=normal, X patches, see webpage (5) or vbetool (6)
Acer TM 803LCi	vga=normal, vbetool (6)
Arima W730a	vbetool needed (6)
Asus L2400D	s3_mode (3)[2] (S1 also works OK)
Asus L3350M (SiS 740)	(6)
Asus L3800C (Radeon M7)	s3_bios (2) (S1 also works OK)
Asus M6887Ne	vga=normal, s3_bios (2), use radeon driver instead of fglrx in x.org
Athlon64 desktop prototype	s3_bios (2)

Table 1 – continued from previous page

Model	hack (or "how to do it")
Compal CL-50	???[Page 116, 1]
Compaq Armada E500 - P3-700	none (1) (S1 also works OK)
Compaq Evo N620c	vga=normal, s3_bios (2)
Dell 600m, ATI R250 Lf	none (1), but needs xorg-x11-6.8.1.902-1
Dell D600, ATI RV250	vga=normal and X, or try vbestate (6)
Dell D610	vga=normal and X (possibly vbestate (6) too, but not tested)
Dell Inspiron 4000	???[Page 116, 1]
Dell Inspiron 500m	???[Page 116, 1]
Dell Inspiron 510m	???
Dell Inspiron 5150	vbetool needed (6)
Dell Inspiron 600m	???[Page 116, 1]
Dell Inspiron 8200	???[Page 116, 1]
Dell Inspiron 8500	???[Page 116, 1]
Dell Inspiron 8600	???[Page 116, 1]
eMachines athlon64 machines	vbetool needed (6) (someone please get me model #s)
HP NC6000	s3_bios, may not use radeonfb (2); or vbetool (6)
HP NX7000	???[Page 116, 1]
HP Pavilion ZD7000	vbetool post needed, need open-source nv driver for X
HP Omnibook XE3 athlon version	none (1)
HP Omnibook XE3GC	none (1), video is S3 Savage/IX-MV
HP Omnibook XE3L-GF	vbetool (6)
HP Omnibook 5150	none (1), (S1 also works OK)
IBM TP T20, model 2647-44G	none (1), video is S3 Inc. 86C270-294 Savage/IX-MV, vesafb gets "interesting" but X work.
IBM TP A31 / Type 2652-M5G	s3_mode (3) [works ok with BIOS 1.04 2002-08-23, but not at all with BIOS 1.11 2004-11-05 :-(]
IBM TP R32 / Type 2658-MMG	none (1)
IBM TP R40 2722B3G	???[Page 116, 1]
IBM TP R50p / Type 1832-22U	s3_bios (2)
IBM TP R51	none (1)
IBM TP T30 236681A	???[Page 116, 1]
IBM TP T40 / Type 2373-MU4	none (1)
IBM TP T40p	none (1)
IBM TP R40p	s3_bios (2)
IBM TP T41p	s3_bios (2), switch to X after resume
IBM TP T42	s3_bios (2)
IBM ThinkPad T42p (2373-GTG)	s3_bios (2)
IBM TP X20	???[Page 116, 1]
IBM TP X30	s3_bios, s3_mode (4)

Table 1 – continued from previous page

Model	hack (or "how to do it")
IBM TP X31 / Type 2672-XXH	none (1), use radeontool (http://fdd.com/software/radeon/) to turn off backlight.
IBM TP X32	none (1), but backlight is on and video is trashed after long suspend. s3_bios, s3_mode (4) works too. Perhaps that gets better results?
IBM Thinkpad X40 Type 2371-7JG	s3_bios,s3_mode (4)
IBM TP 600e	none(1), but a switch to console and back to X is needed
Medion MD4220	???[Page 116, 1]
Samsung P35	vbetool needed (6)
Sharp PC-AR10 (ATI rage)	none (1), backlight does not switch off
Sony Vaio PCG-C1VRX/K	s3_bios (2)
Sony Vaio PCG-F403	???[1]
Sony Vaio PCG-GRT995MP	none (1), works with 'nv' X driver
Sony Vaio PCG-GR7/K	none (1), but needs radeonfb, use radeontool (http://fdd.com/software/radeon/) to turn off backlight.
Sony Vaio PCG-N505SN	???[1]
Sony Vaio vgn-s260	X or boot-radeon can init it (5)
Sony Vaio vgn-S580BH	vga=normal, but suspend from X. Console will be blank unless you return to X.
Sony Vaio vgn-FS115B	s3_bios (2),s3_mode (4)
Toshiba Libretto L5	none (1)
Toshiba Libretto 100CT/110CT	vbetool (6)
Toshiba Portege 3020CT	s3_mode (3)
Toshiba Satellite 4030CDT	s3_mode (3) (S1 also works OK)
Toshiba Satellite 4080XCDT	s3_mode (3) (S1 also works OK)
Toshiba Satellite 4090XCDT	???[1]
Toshiba Satellite P10-554	s3_bios,s3_mode (4)[#f3]_
Toshiba M30	(2) xor X with nvidia driver using internal AGP
Uniwill 244IIO	???[1]

---

[1] from https://wiki.ubuntu.com/HoaryPMResults, not sure which options to use. If you know, please tell me.

[2] To be tested with a newer kernel.

18.2 Known working desktop systems

Mainboard	Graphics card	hack (or "how to do it")
Asus A7V8X	nVidia RIVA TNT2 model 64	s3_bios,s3_mode (4)

SWSUSP/S3 TRICKS

Pavel Machek <pavel@ucw.cz>

If you want to trick swsusp/S3 into working, you might want to try:

- go with minimal config, turn off drivers like USB, AGP you don't really need

- turn off APIC and preempt

- use ext2. At least it has working fsck. [If something seems to go wrong, force fsck when you have a chance]

- turn off modules

- use vga text console, shut down X. [If you really want X, you might want to try vesafb later]

- try running as few processes as possible, preferably go to single user mode.

- due to video issues, swsusp should be easier to get working than S3. Try that first.

When you make it work, try to find out what exactly was it that broke suspend, and preferably fix that.

DOCUMENTATION FOR USERLAND SOFTWARE SUSPEND INTERFACE

(C) 2006 Rafael J. Wysocki <rjw@sisk.pl>

First, the warnings at the beginning of *Swap suspend* still apply.

Second, you should read the FAQ in *Swap suspend* _now_ if you have not done it already.

Now, to use the userland interface for software suspend you need special utilities that will read/write the system memory snapshot from/to the kernel. Such utilities are available, for example, from <http://suspend.sourceforge.net>. You may want to have a look at them if you are going to develop your own suspend/resume utilities.

The interface consists of a character device providing the open(), release(), read(), and write() operations as well as several ioctl() commands defined in include/linux/suspend_ioctls.h . The major and minor numbers of the device are, respectively, 10 and 231, and they can be read from /sys/class/misc/snapshot/dev.

The device can be open either for reading or for writing. If open for reading, it is considered to be in the suspend mode. Otherwise it is assumed to be in the resume mode. The device cannot be open for simultaneous reading and writing. It is also impossible to have the device open more than once at a time.

Even opening the device has side effects. Data structures are allocated, and PM_HIBERNATION_PREPARE / PM_RESTORE_PREPARE chains are called.

The ioctl() commands recognized by the device are:

SNAPSHOT_FREEZE
> freeze user space processes (the current process is not frozen); this is required for SNAPSHOT_CREATE_IMAGE and SNAPSHOT_ATOMIC_RESTORE to succeed

SNAPSHOT_UNFREEZE
> thaw user space processes frozen by SNAPSHOT_FREEZE

SNAPSHOT_CREATE_IMAGE
> create a snapshot of the system memory; the last argument of ioctl() should be a pointer to an int variable, the value of which will indicate whether the call returned after creating the snapshot (1) or after restoring the system memory state from it (0) (after resume the system finds itself finishing the SNAPSHOT_CREATE_IMAGE ioctl() again); after the snapshot has been created the read() operation can be used to transfer it out of the kernel

SNAPSHOT_ATOMIC_RESTORE
> restore the system memory state from the uploaded snapshot image; before calling it you

should transfer the system memory snapshot back to the kernel using the write() operation; this call will not succeed if the snapshot image is not available to the kernel

SNAPSHOT_FREE
free memory allocated for the snapshot image

SNAPSHOT_PREF_IMAGE_SIZE
set the preferred maximum size of the image (the kernel will do its best to ensure the image size will not exceed this number, but if it turns out to be impossible, the kernel will create the smallest image possible)

SNAPSHOT_GET_IMAGE_SIZE
return the actual size of the hibernation image (the last argument should be a pointer to a loff_t variable that will contain the result if the call is successful)

SNAPSHOT_AVAIL_SWAP_SIZE
return the amount of available swap in bytes (the last argument should be a pointer to a loff_t variable that will contain the result if the call is successful)

SNAPSHOT_ALLOC_SWAP_PAGE
allocate a swap page from the resume partition (the last argument should be a pointer to a loff_t variable that will contain the swap page offset if the call is successful)

SNAPSHOT_FREE_SWAP_PAGES
free all swap pages allocated by SNAPSHOT_ALLOC_SWAP_PAGE

SNAPSHOT_SET_SWAP_AREA
set the resume partition and the offset (in <PAGE_SIZE> units) from the beginning of the partition at which the swap header is located (the last ioctl() argument should point to a struct resume_swap_area, as defined in kernel/power/suspend_ioctls.h, containing the resume device specification and the offset); for swap partitions the offset is always 0, but it is different from zero for swap files (see *Using swap files with software suspend (swsusp)* for details).

SNAPSHOT_PLATFORM_SUPPORT
enable/disable the hibernation platform support, depending on the argument value (enable, if the argument is nonzero)

SNAPSHOT_POWER_OFF
make the kernel transition the system to the hibernation state (eg. ACPI S4) using the platform (eg. ACPI) driver

SNAPSHOT_S2RAM
suspend to RAM; using this call causes the kernel to immediately enter the suspend-to-RAM state, so this call must always be preceded by the SNAPSHOT_FREEZE call and it is also necessary to use the SNAPSHOT_UNFREEZE call after the system wakes up. This call is needed to implement the suspend-to-both mechanism in which the suspend image is first created, as though the system had been suspended to disk, and then the system is suspended to RAM (this makes it possible to resume the system from RAM if there's enough battery power or restore its state on the basis of the saved suspend image otherwise)

The device's read() operation can be used to transfer the snapshot image from the kernel. It has the following limitations:

- you cannot read() more than one virtual memory page at a time

- read()s across page boundaries are impossible (ie. if you read() 1/2 of a page in the previous call, you will only be able to read() **at most** 1/2 of the page in the next call)

The device's write() operation is used for uploading the system memory snapshot into the kernel. It has the same limitations as the read() operation.

The release() operation frees all memory allocated for the snapshot image and all swap pages allocated with SNAPSHOT_ALLOC_SWAP_PAGE (if any). Thus it is not necessary to use either SNAPSHOT_FREE or SNAPSHOT_FREE_SWAP_PAGES before closing the device (in fact it will also unfreeze user space processes frozen by SNAPSHOT_UNFREEZE if they are still frozen when the device is being closed).

Currently it is assumed that the userland utilities reading/writing the snapshot image from/to the kernel will use a swap partition, called the resume partition, or a swap file as storage space (if a swap file is used, the resume partition is the partition that holds this file). However, this is not really required, as they can use, for example, a special (blank) suspend partition or a file on a partition that is unmounted before SNAPSHOT_CREATE_IMAGE and mounted afterwards.

These utilities MUST NOT make any assumptions regarding the ordering of data within the snapshot image. The contents of the image are entirely owned by the kernel and its structure may be changed in future kernel releases.

The snapshot image MUST be written to the kernel unaltered (ie. all of the image data, metadata and header MUST be written in _exactly_ the same amount, form and order in which they have been read). Otherwise, the behavior of the resumed system may be totally unpredictable.

While executing SNAPSHOT_ATOMIC_RESTORE the kernel checks if the structure of the snapshot image is consistent with the information stored in the image header. If any inconsistencies are detected, SNAPSHOT_ATOMIC_RESTORE will not succeed. Still, this is not a fool-proof mechanism and the userland utilities using the interface SHOULD use additional means, such as checksums, to ensure the integrity of the snapshot image.

The suspending and resuming utilities MUST lock themselves in memory, preferably using mlockall(), before calling SNAPSHOT_FREEZE.

The suspending utility MUST check the value stored by SNAPSHOT_CREATE_IMAGE in the memory location pointed to by the last argument of ioctl() and proceed in accordance with it:

1. If the value is 1 (ie. the system memory snapshot has just been created and the system is ready for saving it):

   (a) The suspending utility MUST NOT close the snapshot device _unless_ the whole suspend procedure is to be cancelled, in which case, if the snapshot image has already been saved, the suspending utility SHOULD destroy it, preferably by zapping its header. If the suspend is not to be cancelled, the system MUST be powered off or rebooted after the snapshot image has been saved.

   (b) The suspending utility SHOULD NOT attempt to perform any file system operations (including reads) on the file systems that were mounted before SNAPSHOT_CREATE_IMAGE has been called. However, it MAY mount a file system that was not mounted at that time and perform some operations on it (eg. use it for saving the image).

2. If the value is 0 (ie. the system state has just been restored from the snapshot image), the suspending utility MUST close the snapshot device. Afterwards it will be treated as a regular userland process, so it need not exit.

The resuming utility SHOULD NOT attempt to mount any file systems that could be mounted before suspend and SHOULD NOT attempt to perform any operations involving such file systems.

For details, please refer to the source code.

POWER CAPPING FRAMEWORK

The power capping framework provides a consistent interface between the kernel and the user space that allows power capping drivers to expose the settings to user space in a uniform way.

21.1 Terminology

The framework exposes power capping devices to user space via sysfs in the form of a tree of objects. The objects at the root level of the tree represent 'control types', which correspond to different methods of power capping. For example, the intel-rapl control type represents the Intel "Running Average Power Limit" (RAPL) technology, whereas the 'idle-injection' control type corresponds to the use of idle injection for controlling power.

Power zones represent different parts of the system, which can be controlled and monitored using the power capping method determined by the control type the given zone belongs to. They each contain attributes for monitoring power, as well as controls represented in the form of power constraints. If the parts of the system represented by different power zones are hierarchical (that is, one bigger part consists of multiple smaller parts that each have their own power controls), those power zones may also be organized in a hierarchy with one parent power zone containing multiple subzones and so on to reflect the power control topology of the system. In that case, it is possible to apply power capping to a set of devices together using the parent power zone and if more fine grained control is required, it can be applied through the subzones.

Example sysfs interface tree:

```
/sys/devices/virtual/powercap
└──intel-rapl
    ├──intel-rapl:0
    │   ├──constraint_0_name
    │   ├──constraint_0_power_limit_uw
    │   ├──constraint_0_time_window_us
    │   ├──constraint_1_name
    │   ├──constraint_1_power_limit_uw
    │   ├──constraint_1_time_window_us
    │   ├──device -> ../../intel-rapl
    │   ├──energy_uj
    │   ├──intel-rapl:0:0
    │   │   ├──constraint_0_name
    │   │   ├──constraint_0_power_limit_uw
    │   │   ├──constraint_0_time_window_us
```

```
│   │   ├──constraint_1_name
│   │   ├──constraint_1_power_limit_uw
│   │   ├──constraint_1_time_window_us
│   │   ├──device -> ../../intel-rapl:0
│   │   ├──energy_uj
│   │   ├──max_energy_range_uj
│   │   ├──name
│   │   ├──enabled
│   │   ├──power
│   │   │   ├──async
│   │   │   []
│   │   ├──subsystem -> ../../../../../../class/power_cap
│   │   └──uevent
│   ├──intel-rapl:0:1
│   │   ├──constraint_0_name
│   │   ├──constraint_0_power_limit_uw
│   │   ├──constraint_0_time_window_us
│   │   ├──constraint_1_name
│   │   ├──constraint_1_power_limit_uw
│   │   ├──constraint_1_time_window_us
│   │   ├──device -> ../../intel-rapl:0
│   │   ├──energy_uj
│   │   ├──max_energy_range_uj
│   │   ├──name
│   │   ├──enabled
│   │   ├──power
│   │   │   ├──async
│   │   │   []
│   │   ├──subsystem -> ../../../../../../class/power_cap
│   │   └──uevent
│   ├──max_energy_range_uj
│   ├──max_power_range_uw
│   ├──name
│   ├──enabled
│   ├──power
│   │   ├──async
│   │   []
│   ├──subsystem -> ../../../../../class/power_cap
│   ├──enabled
│   ├──uevent
├──intel-rapl:1
│   ├──constraint_0_name
│   ├──constraint_0_power_limit_uw
│   ├──constraint_0_time_window_us
│   ├──constraint_1_name
│   ├──constraint_1_power_limit_uw
│   ├──constraint_1_time_window_us
│   ├──device -> ../../intel-rapl
```

```
|       ├──energy_uj
|       ├──intel-rapl:1:0
|       │   ├──constraint_0_name
|       │   ├──constraint_0_power_limit_uw
|       │   ├──constraint_0_time_window_us
|       │   ├──constraint_1_name
|       │   ├──constraint_1_power_limit_uw
|       │   ├──constraint_1_time_window_us
|       │   ├──device -> ../../intel-rapl:1
|       │   ├──energy_uj
|       │   ├──max_energy_range_uj
|       │   ├──name
|       │   ├──enabled
|       │   ├──power
|       │   │   ├──async
|       │   │   []
|       │   ├──subsystem -> ../../../../../../class/power_cap
|       │   └──uevent
|       ├──intel-rapl:1:1
|       │   ├──constraint_0_name
|       │   ├──constraint_0_power_limit_uw
|       │   ├──constraint_0_time_window_us
|       │   ├──constraint_1_name
|       │   ├──constraint_1_power_limit_uw
|       │   ├──constraint_1_time_window_us
|       │   ├──device -> ../../intel-rapl:1
|       │   ├──energy_uj
|       │   ├──max_energy_range_uj
|       │   ├──name
|       │   ├──enabled
|       │   ├──power
|       │   │   ├──async
|       │   │   []
|       │   ├──subsystem -> ../../../../../../class/power_cap
|       │   └──uevent
|       ├──max_energy_range_uj
|       ├──max_power_range_uw
|       ├──name
|       ├──enabled
|       ├──power
|       │   ├──async
|       │   []
|       ├──subsystem -> ../../../../../class/power_cap
|       ├──uevent
├──power
|   ├──async
|   []
├──subsystem -> ../../../../class/power_cap
```

```
├──enabled
└──uevent
```

The above example illustrates a case in which the Intel RAPL technology, available in Intel® IA-64 and IA-32 Processor Architectures, is used. There is one control type called intel-rapl which contains two power zones, intel-rapl:0 and intel-rapl:1, representing CPU packages. Each of these power zones contains two subzones, intel-rapl:j:0 and intel-rapl:j:1 (j = 0, 1), representing the "core" and the "uncore" parts of the given CPU package, respectively. All of the zones and subzones contain energy monitoring attributes (energy_uj, max_energy_range_uj) and constraint attributes (constraint_*) allowing controls to be applied (the constraints in the 'package' power zones apply to the whole CPU packages and the subzone constraints only apply to the respective parts of the given package individually). Since Intel RAPL doesn't provide instantaneous power value, there is no power_uw attribute.

In addition to that, each power zone contains a name attribute, allowing the part of the system represented by that zone to be identified. For example:

```
cat /sys/class/power_cap/intel-rapl/intel-rapl:0/name
```

21.1.1 package-0

Depending on different power zones, the Intel RAPL technology allows one or multiple constraints like short term, long term and peak power, with different time windows to be applied to each power zone. All the zones contain attributes representing the constraint names, power limits and the sizes of the time windows. Note that time window is not applicable to peak power. Here, constraint_j_* attributes correspond to the jth constraint (j = 0,1,2).

For example:

```
constraint_0_name
constraint_0_power_limit_uw
constraint_0_time_window_us
constraint_1_name
constraint_1_power_limit_uw
constraint_1_time_window_us
constraint_2_name
constraint_2_power_limit_uw
constraint_2_time_window_us
```

21.2 Power Zone Attributes

21.2.1 Monitoring attributes

energy_uj (rw)
    Current energy counter in micro joules. Write "0" to reset. If the counter can not be reset, then this attribute is read only.

max_energy_range_uj (ro)
    Range of the above energy counter in micro-joules.

power_uw (ro)
> Current power in micro watts.

max_power_range_uw (ro)
> Range of the above power value in micro-watts.

name (ro)
> Name of this power zone.

It is possible that some domains have both power ranges and energy counter ranges; however, only one is mandatory.

21.2.2 Constraints

constraint_X_power_limit_uw (rw)
> Power limit in micro watts, which should be applicable for the time window specified by "constraint_X_time_window_us".

constraint_X_time_window_us (rw)
> Time window in micro seconds.

constraint_X_name (ro)
> An optional name of the constraint

constraint_X_max_power_uw(ro)
> Maximum allowed power in micro watts.

constraint_X_min_power_uw(ro)
> Minimum allowed power in micro watts.

constraint_X_max_time_window_us(ro)
> Maximum allowed time window in micro seconds.

constraint_X_min_time_window_us(ro)
> Minimum allowed time window in micro seconds.

Except power_limit_uw and time_window_us other fields are optional.

21.2.3 Common zone and control type attributes

enabled (rw): Enable/Disable controls at zone level or for all zones using a control type.

21.3 Power Cap Client Driver Interface

The API summary:

Call powercap_register_control_type() to register control type object. Call powercap_register_zone() to register a power zone (under a given control type), either as a top-level power zone or as a subzone of another power zone registered earlier. The number of constraints in a power zone and the corresponding callbacks have to be defined prior to calling powercap_register_zone() to register that zone.

To Free a power zone call powercap_unregister_zone(). To free a control type object call powercap_unregister_control_type(). Detailed API can be generated using kernel-doc on include/linux/powercap.h.

DYNAMIC THERMAL POWER MANAGEMENT FRAMEWORK

On the embedded world, the complexity of the SoC leads to an increasing number of hotspots which need to be monitored and mitigated as a whole in order to prevent the temperature to go above the normative and legally stated 'skin temperature'.

Another aspect is to sustain the performance for a given power budget, for example virtual reality where the user can feel dizziness if the performance is capped while a big CPU is processing something else. Or reduce the battery charging because the dissipated power is too high compared with the power consumed by other devices.

The user space is the most adequate place to dynamically act on the different devices by limiting their power given an application profile: it has the knowledge of the platform.

The Dynamic Thermal Power Management (DTPM) is a technique acting on the device power by limiting and/or balancing a power budget among different devices.

The DTPM framework provides an unified interface to act on the device power.

22.1 Overview

The DTPM framework relies on the powercap framework to create the powercap entries in the sysfs directory and implement the backend driver to do the connection with the power manageable device.

The DTPM is a tree representation describing the power constraints shared between devices, not their physical positions.

The nodes of the tree are a virtual description aggregating the power characteristics of the children nodes and their power limitations.

The leaves of the tree are the real power manageable devices.

For instance:

```
SoC
 |
 `-- pkg
      |
      |-- pd0 (cpu0-3)
      |
      `-- pd1 (cpu4-5)
```

The pkg power will be the sum of pd0 and pd1 power numbers:

```
SoC (400mW - 3100mW)
 |
 `-- pkg (400mW - 3100mW)
        |
        |-- pd0 (100mW - 700mW)
        |
        `-- pd1 (300mW - 2400mW)
```

When the nodes are inserted in the tree, their power characteristics are propagated to the parents:

```
SoC (600mW - 5900mW)
 |
 |-- pkg (400mW - 3100mW)
 |      |
 |      |-- pd0 (100mW - 700mW)
 |      |
 |      `-- pd1 (300mW - 2400mW)
 |
 `-- pd2 (200mW - 2800mW)
```

Each node have a weight on a 2^{10} basis reflecting the percentage of power consumption along the siblings:

```
SoC (w=1024)
 |
 |-- pkg (w=538)
 |      |
 |      |-- pd0 (w=231)
 |      |
 |      `-- pd1 (w=794)
 |
 `-- pd2 (w=486)

 Note the sum of weights at the same level are equal to 1024.
```

When a power limitation is applied to a node, then it is distributed along the children given their weights. For example, if we set a power limitation of 3200mW at the 'SoC' root node, the resulting tree will be:

```
SoC (w=1024) <--- power_limit = 3200mW
 |
 |-- pkg (w=538) --> power_limit = 1681mW
 |      |
 |      |-- pd0 (w=231) --> power_limit = 378mW
 |      |
 |      `-- pd1 (w=794) --> power_limit = 1303mW
 |
 `-- pd2 (w=486) --> power_limit = 1519mW
```

22.1.1 Flat description

A root node is created and it is the parent of all the nodes. This description is the simplest one and it is supposed to give to user space a flat representation of all the devices supporting the power limitation without any power limitation distribution.

22.1.2 Hierarchical description

The different devices supporting the power limitation are represented hierarchically. There is one root node, all intermediate nodes are grouping the child nodes which can be intermediate nodes also or real devices.

The intermediate nodes aggregate the power information and allows to set the power limit given the weight of the nodes.

22.2 User space API

As stated in the overview, the DTPM framework is built on top of the powercap framework. Thus the sysfs interface is the same, please refer to the powercap documentation for further details.

- power_uw: Instantaneous power consumption. If the node is an intermediate node, then the power consumption will be the sum of all children power consumption.
- max_power_range_uw: The power range resulting of the maximum power minus the minimum power.
- name: The name of the node. This is implementation dependent. Even if it is not recommended for the user space, several nodes can have the same name.
- constraint_X_name: The name of the constraint.
- constraint_X_max_power_uw: The maximum power limit to be applicable to the node.
- constraint_X_power_limit_uw: The power limit to be applied to the node. If the value contained in constraint_X_max_power_uw is set, the constraint will be removed.
- constraint_X_time_window_us: The meaning of this file will depend on the constraint number.

22.2.1 Constraints

- Constraint 0: The power limitation is immediately applied, without limitation in time.

22.3 Kernel API

22.3.1 Overview

The DTPM framework has no power limiting backend support. It is generic and provides a set of API to let the different drivers to implement the backend part for the power limitation and create the power constraints tree.

It is up to the platform to provide the initialization function to allocate and link the different nodes of the tree.

A special macro has the role of declaring a node and the corresponding initialization function via a description structure. This one contains an optional parent field allowing to hook different devices to an already existing tree at boot time.

For instance:

```
struct dtpm_descr my_descr = {
        .name = "my_name",
        .init = my_init_func,
};

DTPM_DECLARE(my_descr);
```

The nodes of the DTPM tree are described with dtpm structure. The steps to add a new power limitable device is done in three steps:

- Allocate the dtpm node
- Set the power number of the dtpm node
- Register the dtpm node

The registration of the dtpm node is done with the powercap ops. Basically, it must implements the callbacks to get and set the power and the limit.

Alternatively, if the node to be inserted is an intermediate one, then a simple function to insert it as a future parent is available.

If a device has its power characteristics changing, then the tree must be updated with the new power numbers and weights.

22.3.2 Nomenclature

- dtpm_alloc() : Allocate and initialize a dtpm structure
- dtpm_register() : Add the dtpm node to the tree
- dtpm_unregister() : Remove the dtpm node from the tree
- dtpm_update_power() : Update the power characteristics of the dtpm node

REGULATOR CONSUMER DRIVER INTERFACE

This text describes the regulator interface for consumer device drivers. Please see overview.txt for a description of the terms used in this text.

23.1 1. Consumer Regulator Access (static & dynamic drivers)

A consumer driver can get access to its supply regulator by calling

```
regulator = regulator_get(dev, "Vcc");
```

The consumer passes in its struct device pointer and power supply ID. The core then finds the correct regulator by consulting a machine specific lookup table. If the lookup is successful then this call will return a pointer to the struct regulator that supplies this consumer.

To release the regulator the consumer driver should call

```
regulator_put(regulator);
```

Consumers can be supplied by more than one regulator e.g. codec consumer with analog and digital supplies

```
digital = regulator_get(dev, "Vcc");   /* digital core */
analog = regulator_get(dev, "Avdd");   /* analog */
```

The regulator access functions regulator_get() and regulator_put() will usually be called in your device drivers probe() and remove() respectively.

23.2 2. Regulator Output Enable & Disable (static & dynamic drivers)

A consumer can enable its power supply by calling:

```
int regulator_enable(regulator);
```

NOTE:
> The supply may already be enabled before regulator_enable() is called. This may happen if the consumer shares the regulator or the regulator has been previously enabled by bootloader or kernel board initialization code.

A consumer can determine if a regulator is enabled by calling:

```
int regulator_is_enabled(regulator);
```

This will return > zero when the regulator is enabled.

A consumer can disable its supply when no longer needed by calling:

```
int regulator_disable(regulator);
```

NOTE:
> This may not disable the supply if it's shared with other consumers. The regulator will only be disabled when the enabled reference count is zero.

Finally, a regulator can be forcefully disabled in the case of an emergency:

```
int regulator_force_disable(regulator);
```

NOTE:
> this will immediately and forcefully shutdown the regulator output. All consumers will be powered off.

23.3 3. Regulator Voltage Control & Status (dynamic drivers)

Some consumer drivers need to be able to dynamically change their supply voltage to match system operating points. e.g. CPUfreq drivers can scale voltage along with frequency to save power, SD drivers may need to select the correct card voltage, etc.

Consumers can control their supply voltage by calling:

```
int regulator_set_voltage(regulator, min_uV, max_uV);
```

Where min_uV and max_uV are the minimum and maximum acceptable voltages in microvolts.

NOTE: this can be called when the regulator is enabled or disabled. If called when enabled, then the voltage changes instantly, otherwise the voltage configuration changes and the voltage is physically set when the regulator is next enabled.

The regulators configured voltage output can be found by calling:

```
int regulator_get_voltage(regulator);
```

NOTE:
> get_voltage() will return the configured output voltage whether the regulator is enabled or disabled and should NOT be used to determine regulator output state. However this can be used in conjunction with is_enabled() to determine the regulator physical output voltage.

23.4 4.   Regulator Current Limit Control & Status (dynamic drivers)

Some consumer drivers need to be able to dynamically change their supply current limit to match system operating points. e.g. LCD backlight driver can change the current limit to vary the backlight brightness, USB drivers may want to set the limit to 500mA when supplying power.

Consumers can control their supply current limit by calling:

```
int regulator_set_current_limit(regulator, min_uA, max_uA);
```

Where min_uA and max_uA are the minimum and maximum acceptable current limit in microamps.

NOTE:
> this can be called when the regulator is enabled or disabled. If called when enabled, then the current limit changes instantly, otherwise the current limit configuration changes and the current limit is physically set when the regulator is next enabled.

A regulators current limit can be found by calling:

```
int regulator_get_current_limit(regulator);
```

NOTE:
> get_current_limit() will return the current limit whether the regulator is enabled or disabled and should not be used to determine regulator current load.

23.5 5.   Regulator Operating Mode Control & Status (dynamic drivers)

Some consumers can further save system power by changing the operating mode of their supply regulator to be more efficient when the consumers operating state changes. e.g. consumer driver is idle and subsequently draws less current

Regulator operating mode can be changed indirectly or directly.

23.5.1 Indirect operating mode control.

Consumer drivers can request a change in their supply regulator operating mode by calling:

```
int regulator_set_load(struct regulator *regulator, int load_uA);
```

This will cause the core to recalculate the total load on the regulator (based on all its consumers) and change operating mode (if necessary and permitted) to best match the current operating load.

The load_uA value can be determined from the consumer's datasheet. e.g. most datasheets have tables showing the maximum current consumed in certain situations.

Most consumers will use indirect operating mode control since they have no knowledge of the regulator or whether the regulator is shared with other consumers.

23.5.2 Direct operating mode control.

Bespoke or tightly coupled drivers may want to directly control regulator operating mode depending on their operating point. This can be achieved by calling:

```
int regulator_set_mode(struct regulator *regulator, unsigned int mode);
unsigned int regulator_get_mode(struct regulator *regulator);
```

Direct mode will only be used by consumers that *know* about the regulator and are not sharing the regulator with other consumers.

23.6 6. Regulator Events

Regulators can notify consumers of external events. Events could be received by consumers under regulator stress or failure conditions.

Consumers can register interest in regulator events by calling:

```
int regulator_register_notifier(struct regulator *regulator,
                                struct notifier_block *nb);
```

Consumers can unregister interest by calling:

```
int regulator_unregister_notifier(struct regulator *regulator,
                                  struct notifier_block *nb);
```

Regulators use the kernel notifier framework to send event to their interested consumers.

23.7 7. Regulator Direct Register Access

Some kinds of power management hardware or firmware are designed such that they need to do low-level hardware access to regulators, with no involvement from the kernel. Examples of such devices are:

- clocksource with a voltage-controlled oscillator and control logic to change the supply voltage over I2C to achieve a desired output clock rate

- thermal management firmware that can issue an arbitrary I2C transaction to perform system poweroff during overtemperature conditions

To set up such a device/firmware, various parameters like I2C address of the regulator, addresses of various regulator registers etc. need to be configured to it. The regulator framework provides the following helpers for querying these details.

Bus-specific details, like I2C addresses or transfer rates are handled by the regmap framework. To get the regulator's regmap (if supported), use:

```
struct regmap *regulator_get_regmap(struct regulator *regulator);
```

To obtain the hardware register offset and bitmask for the regulator's voltage selector register, use:

```
int regulator_get_hardware_vsel_register(struct regulator *regulator,
                                         unsigned *vsel_reg,
                                         unsigned *vsel_mask);
```

To convert a regulator framework voltage selector code (used by regulator_list_voltage) to a hardware-specific voltage selector that can be directly written to the voltage selector register, use:

```
int regulator_list_hardware_vsel(struct regulator *regulator,
                                 unsigned selector);
```

TWENTYFOUR

REGULATOR API DESIGN NOTES

This document provides a brief, partially structured, overview of some of the design considerations which impact the regulator API design.

24.1 Safety

- Errors in regulator configuration can have very serious consequences for the system, potentially including lasting hardware damage.

- It is not possible to automatically determine the power configuration of the system - software-equivalent variants of the same chip may have different power requirements, and not all components with power requirements are visible to software.

Note: The API should make no changes to the hardware state unless it has specific knowledge that these changes are safe to perform on this particular system.

24.2 Consumer use cases
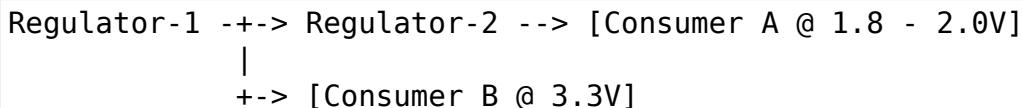
- The overwhelming majority of devices in a system will have no requirement to do any runtime configuration of their power beyond being able to turn it on or off.

- Many of the power supplies in the system will be shared between many different consumers.

Note: The consumer API should be structured so that these use cases are very easy to handle and so that consumers will work with shared supplies without any additional effort.

REGULATOR MACHINE DRIVER INTERFACE

The regulator machine driver interface is intended for board/machine specific initialisation code to configure the regulator subsystem.

Consider the following machine:

```
Regulator-1 -+-> Regulator-2 --> [Consumer A @ 1.8 - 2.0V]
             |
             +-> [Consumer B @ 3.3V]
```

The drivers for consumers A & B must be mapped to the correct regulator in order to control their power supplies. This mapping can be achieved in machine initialisation code by creating a struct regulator_consumer_supply for each regulator:

```
struct regulator_consumer_supply {
     const char *dev_name;   /* consumer dev_name() */
     const char *supply;     /* consumer supply - e.g. "vcc" */
};
```

e.g. for the machine above:

```
static struct regulator_consumer_supply regulator1_consumers[] = {
     REGULATOR_SUPPLY("Vcc", "consumer B"),
};

static struct regulator_consumer_supply regulator2_consumers[] = {
     REGULATOR_SUPPLY("Vcc", "consumer A"),
};
```

This maps Regulator-1 to the 'Vcc' supply for Consumer B and maps Regulator-2 to the 'Vcc' supply for Consumer A.

Constraints can now be registered by defining a struct regulator_init_data for each regulator power domain. This structure also maps the consumers to their supply regulators:

```
static struct regulator_init_data regulator1_data = {
     .constraints = {
             .name = "Regulator-1",
             .min_uV = 3300000,
             .max_uV = 3300000,
             .valid_modes_mask = REGULATOR_MODE_NORMAL,
     },
```

```
        .num_consumer_supplies = ARRAY_SIZE(regulator1_consumers),
        .consumer_supplies = regulator1_consumers,
};
```

The name field should be set to something that is usefully descriptive for the board for configuration of supplies for other regulators and for use in logging and other diagnostic output. Normally the name used for the supply rail in the schematic is a good choice. If no name is provided then the subsystem will choose one.

Regulator-1 supplies power to Regulator-2. This relationship must be registered with the core so that Regulator-1 is also enabled when Consumer A enables its supply (Regulator-2). The supply regulator is set by the supply_regulator field below and co:

```
static struct regulator_init_data regulator2_data = {
        .supply_regulator = "Regulator-1",
        .constraints = {
                .min_uV = 1800000,
                .max_uV = 2000000,
                .valid_ops_mask = REGULATOR_CHANGE_VOLTAGE,
                .valid_modes_mask = REGULATOR_MODE_NORMAL,
        },
        .num_consumer_supplies = ARRAY_SIZE(regulator2_consumers),
        .consumer_supplies = regulator2_consumers,
};
```

Finally the regulator devices must be registered in the usual manner:

```
static struct platform_device regulator_devices[] = {
        {
                .name = "regulator",
                .id = DCDC_1,
                .dev = {
                        .platform_data = &regulator1_data,
                },
        },
        {
                .name = "regulator",
                .id = DCDC_2,
                .dev = {
                        .platform_data = &regulator2_data,
                },
        },
};
/* register regulator 1 device */
platform_device_register(&regulator_devices[0]);

/* register regulator 2 device */
platform_device_register(&regulator_devices[1]);
```

LINUX VOLTAGE AND CURRENT REGULATOR FRAMEWORK

26.1 About

This framework is designed to provide a standard kernel interface to control voltage and current regulators.

The intention is to allow systems to dynamically control regulator power output in order to save power and prolong battery life. This applies to both voltage regulators (where voltage output is controllable) and current sinks (where current limit is controllable).

 (C) 2008 Wolfson Microelectronics PLC.

Author: Liam Girdwood <lrg@slimlogic.co.uk>

26.2 Nomenclature

Some terms used in this document:

- **Regulator**

    - Electronic device that supplies power to other devices. Most regulators can enable and disable their output while some can control their output voltage and or current.

        Input Voltage -> Regulator -> Output Voltage

- **PMIC**

    - Power Management IC. An IC that contains numerous regulators and often contains other subsystems.

- **Consumer**

    - Electronic device that is supplied power by a regulator. Consumers can be classified into two types:-
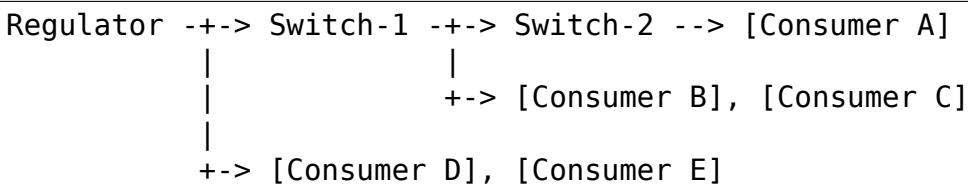
        Static: consumer does not change its supply voltage or current limit. It only needs to enable or disable its power supply. Its supply voltage is set by the hardware, bootloader, firmware or kernel board initialisation code.

        Dynamic: consumer needs to change its supply voltage or current limit to meet operation demands.

- **Power Domain**

- Electronic circuit that is supplied its input power by the output power of a regulator, switch or by another power domain.

   The supply regulator may be behind a switch(s). i.e.:

```
Regulator -+-> Switch-1 -+-> Switch-2 --> [Consumer A]
           |             |
           |             +-> [Consumer B], [Consumer C]
           |
           +-> [Consumer D], [Consumer E]
```
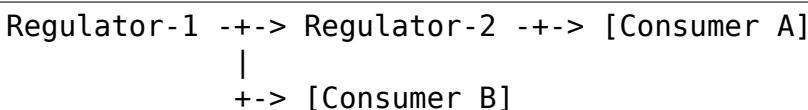
   That is one regulator and three power domains:

   * Domain 1: Switch-1, Consumers D & E.

   * Domain 2: Switch-2, Consumers B & C.

   * Domain 3: Consumer A.

   and this represents a "supplies" relationship:

   Domain-1 --> Domain-2 --> Domain-3.

   A power domain may have regulators that are supplied power by other regulators. i.e.:

```
Regulator-1 -+-> Regulator-2 -+-> [Consumer A]
             |
             +-> [Consumer B]
```

   This gives us two regulators and two power domains:

   * Domain 1: Regulator-2, Consumer B.

   * Domain 2: Consumer A.

   and a "supplies" relationship:

   Domain-1 --> Domain-2

- **Constraints**

   - Constraints are used to define power levels for performance and hardware protection. Constraints exist at three levels:

   Regulator Level: This is defined by the regulator hardware operating parameters and is specified in the regulator datasheet. i.e.

   * voltage output is in the range 800mV -> 3500mV.

   * regulator current output limit is 20mA @ 5V but is 10mA @ 10V.

   Power Domain Level: This is defined in software by kernel level board initialisation code. It is used to constrain a power domain to a particular power range. i.e.

   * Domain-1 voltage is 3300mV

   * Domain-2 voltage is 1400mV -> 1600mV

   * Domain-3 current limit is 0mA -> 20mA.

Consumer Level: This is defined by consumer drivers dynamically setting voltage or current limit levels.

e.g. a consumer backlight driver asks for a current increase from 5mA to 10mA to increase LCD illumination. This passes to through the levels as follows :-

Consumer: need to increase LCD brightness. Lookup and request next current mA value in brightness table (the consumer driver could be used on several different personalities based upon the same reference device).

Power Domain: is the new current limit within the domain operating limits for this domain and system state (e.g. battery power, USB power)

Regulator Domains: is the new current limit within the regulator operating parameters for input/output voltage.

If the regulator request passes all the constraint tests then the new regulator value is applied.

26.3 Design

The framework is designed and targeted at SoC based devices but may also be relevant to non SoC devices and is split into the following four interfaces:-

1. Consumer driver interface.

   This uses a similar API to the kernel clock interface in that consumer drivers can get and put a regulator (like they can with clocks atm) and get/set voltage, current limit, mode, enable and disable. This should allow consumers complete control over their supply voltage and current limit. This also compiles out if not in use so drivers can be reused in systems with no regulator based power control.

   > See *Regulator Consumer Driver Interface*

2. Regulator driver interface.

   This allows regulator drivers to register their regulators and provide operations to the core. It also has a notifier call chain for propagating regulator events to clients.

   > See *Regulator Driver Interface*

3. Machine interface.

   This interface is for machine specific code and allows the creation of voltage/current domains (with constraints) for each regulator. It can provide regulator constraints that will prevent device damage through overvoltage or overcurrent caused by buggy client drivers. It also allows the creation of a regulator tree whereby some regulators are supplied by others (similar to a clock tree).

   > See *Regulator Machine Driver Interface*

4. Userspace ABI.

   The framework also exports a lot of useful voltage/current/opmode data to userspace via sysfs. This could be used to help monitor device power consumption and status.

   > See Documentation/ABI/testing/sysfs-class-regulator

REGULATOR DRIVER INTERFACE

The regulator driver interface is relatively simple and designed to allow regulator drivers to register their services with the core framework.

27.1 Registration

Drivers can register a regulator by calling:

```
struct regulator_dev *regulator_register(struct regulator_desc *regulator_desc,
                                          const struct regulator_config
↪*config);
```

This will register the regulator's capabilities and operations to the regulator core.

Regulators can be unregistered by calling:

```
void regulator_unregister(struct regulator_dev *rdev);
```

27.2 Regulator Events

Regulators can send events (e.g. overtemperature, undervoltage, etc) to consumer drivers by calling:

```
int regulator_notifier_call_chain(struct regulator_dev *rdev,
                                  unsigned long event, void *data);
```

E