Fixing PCI Suspend and Resume

Rafael J. Wysocki
University of Warsaw, Faculty of Physics, Hoża 69, 00-681 Warsaw
rjw@sisk.pl

Abstract

Interrupt handlers implemented by some PCI device drivers can misbehave if the device they are supposed to handle is not in the state they expect it to be in. If this happens, interrupt storm may occur, potentially leading to a system lockup. Unfortunately, if the device in question uses shared interrupts, this can easily happen during suspend to RAM, after the device has been put into a low power state. It is even more likely that this will happen at resume time, before the device is brought back to the state it was in before the suspend. On some machines this leads to intermittent resume failures that are very difficult to diagnose.

In Linux kernels prior to 2.6.29-rc3 the power management core did not do anything to prevent such failures from happening, but during the 2.6.29 development cycle we started to address the issue. Still, the solution finally implemented in the 2.6.29 kernel is partial, because it only covers the devices that support the native PCI power management and it only affects the resume part of the code. The complete solution, which has been included into 2.6.30 and which is described in the present paper, required us to make some radical changes, including a rearrangement of the major steps performed by the kernel during suspend and resume. However, not only should it make suspend and resume much more reliable on a number of systems, but it should also allow the writers of PCI device drivers to simplify their code, because some standard PCI device power management operations will now be carried out by the core.

1 Introduction

From a computer user's point of view suspend to RAM appears to be a relatively simple operation. There is an event that triggers it, for example a button is pressed or a laptop lid is closed, and in a few seconds the machine is put into a state in which power is only provided to its memory chips in order to preserve their contents. This state will further be referred to as the *memory sleep* state.

Analogously, during resume, after a specific event which may be pressing of a button, opening a laptop lid or receiving a magic packet from a network, the system is brought from the memory sleep state back to the working state in a few seconds. The whole operation does not seem to be very complicated, but at the kernel level it involves the execution of large amount of code that is not run in any other circumstances. Moreover, if one part of this code fails, it usually means that the system will not reach the working state again or, even if it does, its functionality will be adversely affected.

It usually is hard to diagnose such failures, especially if they happen sufficiently late during suspend or sufficiently early during resume, since in that cases there is no practical way to get any useful debugging information out of the failing machine. Quite often the only chance to get some insight into the problem is when it appears as a regression and we are able to identify the exact change that caused it to appear.

Something like this happened during the 2.6.28 development cycle, when resume started to break on one of the author's test machines in a reproducible way and it seemed to be a result of a change of the PCI core code responsible for allocating PCI resources. It was confusing, because the change in question should not have affected suspend and resume in any way, but in the process of debugging it Linus Torvalds suggested that it might be a result of mishandling shared PCI interrupts during resume. Namely, due to the way in which the majority of PCI device drivers handled suspend and resume, there was a time window in which an interrupt could arrive before the devices were put into the states they had been in before the suspend. Then, if one of the involved interrupt handlers could not cope with this situation, the system would crash [LINUS1].

Following this suggestion, we created a patch that made the PCI power management core code put all devices supporting the native PCI power management into the full power state (*D*0) and restore their standard configuration registers to the pre-suspend state before enabling the CPU to receive hardware interrupts [PATCH1]. That indeed made the problem go away on the affected system and it was confirmed to fix resume on Linus' own machine, so it has been included into the 2.6.29 kernel. Still, the devices that do not support the native PCI power management might also be affected by the issue with shared interrupts during resume, and the patch did not cover the suspend part of the code at all. Thus, we had to do more to fix the problem completely, but that turned out to be quite difficult [LINUS2].

The main obstacle was the need to use platform hooks for changing the power state of PCI devices that did not support the native power management, because these hooks could not be executed with interrupts disabled on the CPU. To overcome it we had to rework the core kernel's suspend and resume code so that it prevented hardware interrupts from reaching device drivers when PCI devices might not be in the right states without disabling interrupts on the CPU. Moreover, the ACPI specification[1] wants us to put devices into low power states before calling the platform firmware to prepare itself for the system power transition and we had to take this into account as well [ACPI-SPEC]. Consequently, to meet all of the constraints, we rearranged the major steps carried out by the kernel during suspend and resume, which allowed us to develop the complete solution that is present in 2.6.30.

In what follows we describe the problem in greater detail and show a scenario in which it could manifest itself. For this purpose, among other things, we examine the structure of the kernel's suspend and resume code, and explain how it made the observed failures possible. Next, we present both the partial solution used in 2.6.29 and the final one recently merged into the main kernel tree. Finally, we discuss the consequences of this for the authors of PCI device drivers.

2 Description of the problem

To understand the problem and the approach used to fix it, one needs to know how the kernel's suspend and resume code works, but that has been presented elsewhere

[1]ACPI 2.0 or later.

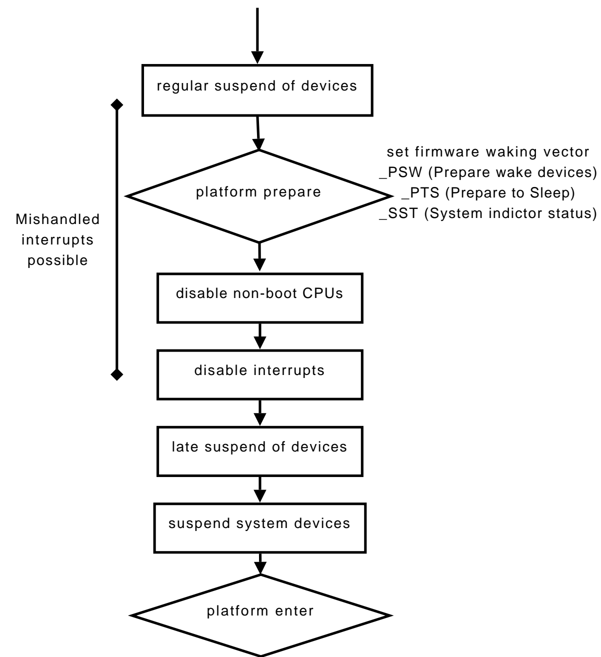

Figure 1: Suspend code structure (2.6.29).

and we are not going to repeat this entire discussion [S2RAM]. Still, we need to recall the parts of it the problem description below is based on.

In the 2.6.29 and earlier kernels there are two phases of suspending devices. The first of them, that further will be referred to as the *regular suspend of devices*, is carried out right before invoking the platform `.prepare()` callback which executes the `_PTS` global control method on ACPI systems. After `.prepare()` has returned, the non-boot CPUs are put off line[2] and interrupts are disabled on the only remaining active CPU. Then, the second phase of suspending devices, that we will refer to as the *late suspend of devices*, is performed. Next, the special system devices called *sysdevs*, such as the local APIC of the boot processor and I/O-APICs, are suspended and the platform is called to put the system into the memory sleep state [S2RAM]. The structure of the suspend code in the 2.6.29 and earlier kernels is schematically shown in Figure 1.

The resume part of the code is organized analogously. First, *sysdevs* are resumed right after the platform firmware has returned control to the kernel. Next, the kernel carries out the first phase of resuming devices

[2]This is accomplished with the help of the CPU hotplug infrastructure.

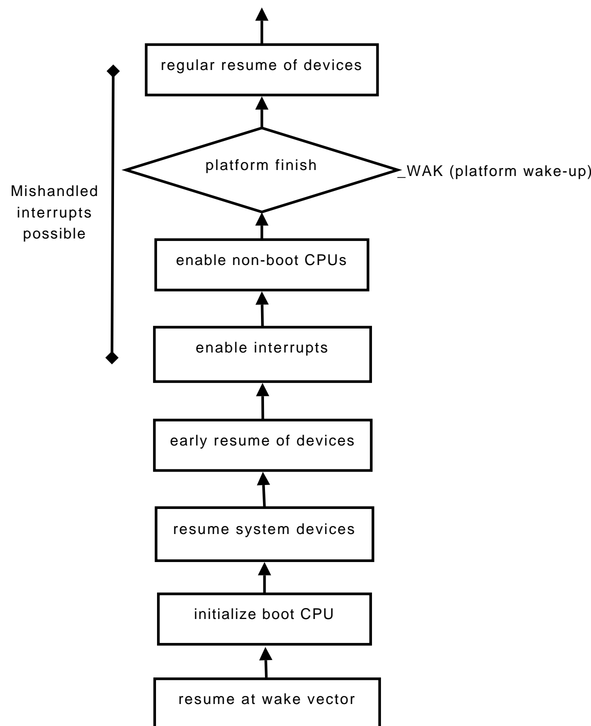

Figure 2: Resume code structure (2.6.29).

that we will refer to as the *early resume of devices*. After that, interrupts are enabled on the only active CPU and the other CPUs are brought back on line. Subsequently, the platform `.finish()` callback is run which causes the `_WAK` global control method to be executed on ACPI systems. Then, the kernel starts the second phase of resuming devices, that further will be referred to as the *regular resume of devices* [S2RAM]. The structure of the resume code in the 2.6.29 and earlier kernels is schematically illustrated in Figure 2.

Thus, in principle, every PCI device driver can implement two suspend callbacks, a *regular* one, to be called before the platform `.prepare()` routine, and a *late* one, to be executed with interrupts disabled. It can also define two analogous resume callbacks, an *early* one, to be executed with interrupts disabled, and a *regular* one, to be executed after the platform `.finish()` callback. However, according to the ACPI specification devices should be put into low power state before the `_PTS` global control method is run [ACPI-SPEC], so the vast majority of drivers implement the regular suspend and resume callbacks only. The late suspend and early resume callbacks are only provided by a few drivers for special purposes. Accordingly, during suspend and analogously during resume there is a time interval in which devices may not be operational or even accessible to their drivers, but the processors can receive interrupts. Then, interrupt handlers may be invoked, even if their devices do not generate any interrupts and if they are not prepared to cope with that situation, the consequences may be dire [LINUS3]. The time intervals in which this is possible during suspend and resume are marked in Figures 1 and 2, respectively, with a vertical line on the left-hand side.

Of course, a PCI device in a low-power state cannot generate interrupts. Yet, if that device uses shared interrupts, then its driver's interrupt handler may be invoked as a result of an interrupt generated by one of the other devices sharing an interrupt vector with it. Moreover, this actually is quite probable, because the suspend and resume callbacks provided by device drivers are executed sequentially [S2RAM], so it is *guaranteed* that one of the devices sharing the interrupt vector will be suspended earlier and resumed later than the other ones[3]. Therefore, if one of these devices is handled by a driver with an interrupt handler that is not designed to work correctly even if the device is not in the right state, things are likely to go wrong. Namely, if the device that has not been suspended yet or that has already been resumed generates an interrupt, the other devices' interrupt handlers will be invoked, and if they fail, the system is going to crash.

This generally is more likely to happen during resume. Specifically, while during suspend the devices are either fully operational or in low power states, so they behave more or less in accordance with the drivers' expectations, during resume they are usually in $D0$ (full power state), but they need not have been initialized yet. Then, before they eventually get initialized, they may respond to the drivers' attempts to access them in a confusing way. Furthermore, during resume interrupts may be generated as a result of a chipset glitch or something similar and that also may confuse interrupt handlers that are not prepared to cope with devices in unexpected states.

Certainly, if every interrupt handler had been able to cope with an already suspended or not yet resumed device, the problem would not have manifested itself. Unfortunately, this evidently is not the case and it would not have been practical to try to fix all of the affected

[3]The order of resuming devices is reverse with respect to the order of suspending them.

drivers [LINUS2]. For this reason the power management core (PM core) code had to be modified to prevent the problem from happening.

3 Preliminary fix

To fix the problem described in Section 2, we used the observation that it would not appear during resume if the standard configuration registers of PCI devices were restored to the pre-suspend state before enabling the CPUs to receive hardware interrupts. Analogously, during suspend the problem could not happen if devices were put into low power states after disabling interrupts on the boot CPU, but this turned out to be more difficult to implement, so we first focused on the resume fix.

To implement it, we had to make the PM core restore the standard configuration registers of PCI devices in the early phase of resume, before executing the early resume callbacks provided by PCI device drivers. Fortunately, that was not too difficult to achieve, thanks to the organization of the device resume code. Namely, the lowest-level resume code[4] does not execute the drivers' resume callbacks directly [S2RAM]. Instead, it invokes the resume callbacks defined by the `pci_bus_type` bus type driver[5], which in turn are responsible for executing the drivers' callbacks. More precisely, the `pci_bus_type` bus type implements a `dev_pm_ops` object, called `pci_dev_pm_ops`, that points to a set of power management callbacks executed by the lowest-level power management code in various phases of suspend and resume[6]. In particular, the `pci_pm_resume_noirq()` routine is called during the early resume of devices and it is responsible for executing the early resume callback provided by each PCI device driver. Thus, it was sufficient to make `pci_pm_resume_noirq()` restore the standard configuration registers of each PCI device before executing the early resume callback provided by its driver, if there was one.

For this purpose, however, we had to ensure the accessibility of the device's standard configuration registers before attempting to restore their pre-suspend values. Of course, in theory, the configuration space of a PCI device should be accessible in any power state, except for $D3_{cold}$ [PCI-PM]; but in practice it is better to put all devices into $D0$ before restoring their configuration registers. Still, we could not use `pci_set_power_state()` to do that, because this function might sleep and therefore it would not be valid to call it with interrupts disabled.

There are two reasons why `pci_set_power_state()` may sleep. First, when changing the power state of a device from $D3$ to $D0$ there is the mandatory 10 ms delay, necessary to allow the device to actually enter the full power state [PCI-PM], and it is implemented in `pci_set_power_state()` with the help of `msleep()`. Second, `pci_set_power_state()` executes a platform callback that in principle may be necessary to set up some power resources (e.g. power planes, reference clock) required to power up the device and this callback may sleep. Thus, during the early resume of devices we could only use the native PCI power management mechanism, implemented in `pci_raw_set_power_state()`, for putting devices into $D0$. Moreover, to be able to prevent `pci_raw_set_power_state()` from calling `msleep()` with interrupts disabled we had to add a new parameter to it. Accordingly, the function called `pci_restore_standard_config()`, shown in Figure 3, was introduced and `pci_pm_resume_noirq()` was modified to call it before executing the device driver's resume callback [PATCH1].

Since the configuration spaces of PCI devices were going to be restored during the early resume, it was necessary to make sure that they would be saved during suspend. For this reason `pci_pm_suspend()` was modified to execute `pci_save_state()` before returning to the caller. However, it could not do that unconditionally, because many device drivers saved the PCI configuration spaces of their devices by themselves before putting the devices into low power states (usually $D3_{hot}$). Of course, in that case, the contents of PCI configuration registers saved by the driver before the device was put into the low power state should not be replaced by their contents saved by `pci_pm_suspend()` when the device *was* already in that state. Therefore, the `state_saved` flag was added to the `pci_dev` structure and `pci_save_state()` was changed to set this flag on every execution, so that `pci_pm_suspend()` could use it do decide whether or not to save the PCI configuration registers of given device. This flag was also

[4] Located in `drivers/base/power/main.c`

[5] Defined in `drivers/pci/pci-driver.c`

[6] It also points to several hibernation-specific callbacks, but they are out of the scope of the present discussion.

```
int pci_restore_standard_config(struct pci_dev *dev)
{
  pci_power_t prev_state;
  int error;

  pci_update_current_state(dev, PCI_D0);

  prev_state = dev->current_state;
  if (prev_state == PCI_D0)
    goto Restore;

  error = pci_raw_set_power_state(dev, PCI_D0, false);
  if (error)
    return error;

  /*
   * This assumes that we won't get a bus in B2 or B3
   * from the BIOS, but we've made this assumption
   * forever and it appears to be universally satisfied.
   */
  switch(prev_state) {
  case PCI_D3cold:
  case PCI_D3hot:
    mdelay(pci_pm_d3_delay);
    break;
  case PCI_D2:
    udelay(PCI_PM_D2_DELAY);
    break;
  }

  pci_update_current_state(dev, PCI_D0);

 Restore:
  return dev->state_saved ?
    pci_restore_state(dev) : 0;
}
```

Figure 3: Function called during early resume to restore configuration space of a PCI device (2.6.29).

used by `pci_restore_standard_config()` to decide whether or not the device's configuration space ought to be restored [PATCH1].

The changes described above caused the standard configuration registers of PCI devices supporting the native PCI power management to be saved during suspend and restored during early resume with interrupts disabled on the CPU. That turned out to fix the problem described in Section 2 on a number of test machines, but it obviously did not cover devices requiring additional power resources controlled by the platform to be set up for putting them into *D*0. It also did not cover the suspend-specific part of the problem in which interrupt handlers might be invoked although their devices had already been put into low power states. Still, both of these shortcomings were related to the fact that the platform hooks necessary to change the power state of some devices could not be executed with interrupts disabled and we were not able to use `pci_set_power_state()` during the late suspend and early resume of devices. Hence, to remove the limitations, it was necessary to either modify the platform hooks so that they could be executed with interrupts disabled, or change the suspend and resume code so that interrupts were enabled on the CPU during the late and early phases of device suspend and resume, respectively [LINUS4].

4 Complete solution

By using the approach presented in Section 3 we were able to partially fix the problem described in Section 2 for devices that supported the PCI native power management. Still, more in-depth changes were necessary to fix it completely and for all devices. Namely, for this purpose we had to make it possible to execute platform callbacks used for changing the power states of devices during the late phase of suspend and the early phase of resume.

There were two possible ways to achieve this goal. First, the platform callbacks, most importantly the ACPI ones, could be modified so that executing them with interrupts disabled was valid, which would have been the case if they had not taken any mutexes and generally if they had avoided using functions that might sleep. This, however, turned out to be impractical due to the complexity of the ACPI code and to the fact that substantial part of it, known as ACPICA[7], was shared with some other operating systems, like BSD [ACPICA]. Second, the core suspend and resume code could be modified so that the CPUs were able to receive hardware interrupts while executing the late suspend and early resume callbacks provided by device drivers. Specifically, as suggested by Linus Torvalds, instead of disabling interrupts on the CPU before the late suspend of devices, we could prevent device drivers from receiving interrupts by running `irq_disable()` for all interrupt vectors except for the timer ones [LINUS5]. Then, even though the CPUs would be able to receive and acknowledge hardware interrupts, the interrupts would be disabled from the drivers' point of view. Moreover, since timer interrupts would still be handled normally, it would be valid to call the potentially sleeping functions from the device drivers' late suspend routines and if we did the analogous change in the resume part

[7] ACPI Common Architecture.

of the code, we would be able to invoke these functions from the drivers' early resume callbacks and from `pci_restore_standard_config()` as well.

This approach was used in the 2.6.30-rc3 and later kernels. It allowed us to put PCI devices into $D0$ from within `pci_restore_standard_config()` with the help of `pci_set_power_state()` which caused the platform callback changing the device's power state to be called as appropriate. Analogously, we could use `pci_set_power_state()` to put PCI devices into low power states during the late suspend of devices. Still, at the same time we wanted to preserve the suspend and resume code ordering following from the ACPI specification[8] stating that devices ought to be put into low power states before the execution of the `_PTS` global control method which, in turn, ought to happen before disabling the non-boot CPUs [ACPI-SPEC]. Similarly, during resume we were supposed to enable the non-boot CPUs before executing the `_WAK` ACPI global control method which, in turn, ought to happen before putting devices into $D0$. Thus, we proposed to make device drivers' late suspend callbacks be executed before the platform `.prepare()` callback and, analogously, their early resume callbacks be executed after the platform `.finish()` callback. In principle, this change should not matter to device drivers and it would allow us to avoid some otherwise inevitable complications related to reordering ACPI callbacks with respect to one another [RJW1].

Unfortunately, it turned out that some platforms, like for example ARM PXA, used the `.prepare()` and `.finish()` callbacks to communicate with power control devices over an I2C bus the controller of which is effectively turned off and on during the late suspend and early resume of devices, respectively [RMK]. As a result, suspend and resume would not have worked on these platforms if the `.prepare()` and `.finish()` callbacks had been executed, respectively, after the late suspend and before the early resume of devices. For this reason we decided to introduce two additional platform callbacks for suspend and resume, `.prepare_late()` and `.wake()`, to be executed in these two places and we used them for running the code that was previously run in `.prepare()` and `.finish()`, respectively, on ACPI systems [RJW2].

Consequently, in the 2.6.30-rc3 and later kernels the ordering of the suspend and resume code is different from its ordering in the 2.6.29 and earlier kernels. Now, after the first phase of suspending devices (i.e. regular suspend) and the execution of the `.prepare()` platform callback, device drivers are prevented from receiving hardware interrupts which is immediately followed by the second phase of suspending devices (i.e. late suspend). Then, the platform `.prepare_late()` callback is executed, the non-boot CPUs are disabled and interrupts are disabled on the only on-line CPU. Finally, *sysdevs* are suspended and the platform is called to put the system into the memory sleep state. Analogously, during resume, after the platform firmware has transferred control to the kernel, *sysdevs* are resumed, interrupts are enabled on the only on-line CPU and the other CPUs are brought back on line. Next, the platform `.wake()` callback is executed and device drivers' early resume callbacks are run. Finally, device drivers are allowed to receive hardware interrupts, the platform `.finish()` callback is executed and the regular resume of devices is carried out. The structure of the suspend and resume code in the 2.6.30-rc3 and later kernels is illustrated in Figure 4.

To make suspend and resume work as described in the previous paragraph, we needed a reversible mechanism for preventing device drivers from receiving interrupts after the regular suspend of devices. For this purpose we had to make some changes to the generic interrupt management code.[9] Roughly, we wanted `irq_disable()` to be called for every interrupt vector, except for the timer ones, right after the completion of the regular suspend of devices, but we also needed to ensure that `irq_enable()` would be called for these interrupt vectors during resume. Thus the `IRQ_SUSPENDED` interrupt status flags was introduced and used to mark the interrupt vectors disabled during suspend, so that they could be re-enabled during resume. Moreover, we had to take the locking and reference counting done by the interrupt management code into account, so we introduced the helper function `__disable_irq()` complementary to `__enable_irq()` and we made both of these functions take an additional Boolean parameter specifying whether or not they were called by the kernel's core suspend and resume code. [It calls them via `suspend_device_irqs()` and `resume_device_irqs()` shown in Figures 5 and 6, respectively.]

In addition, the fact that some platforms used wake-up

[8] ACPI 2.0 or later.

[9] Located in `kernel/irq/manage.c`.

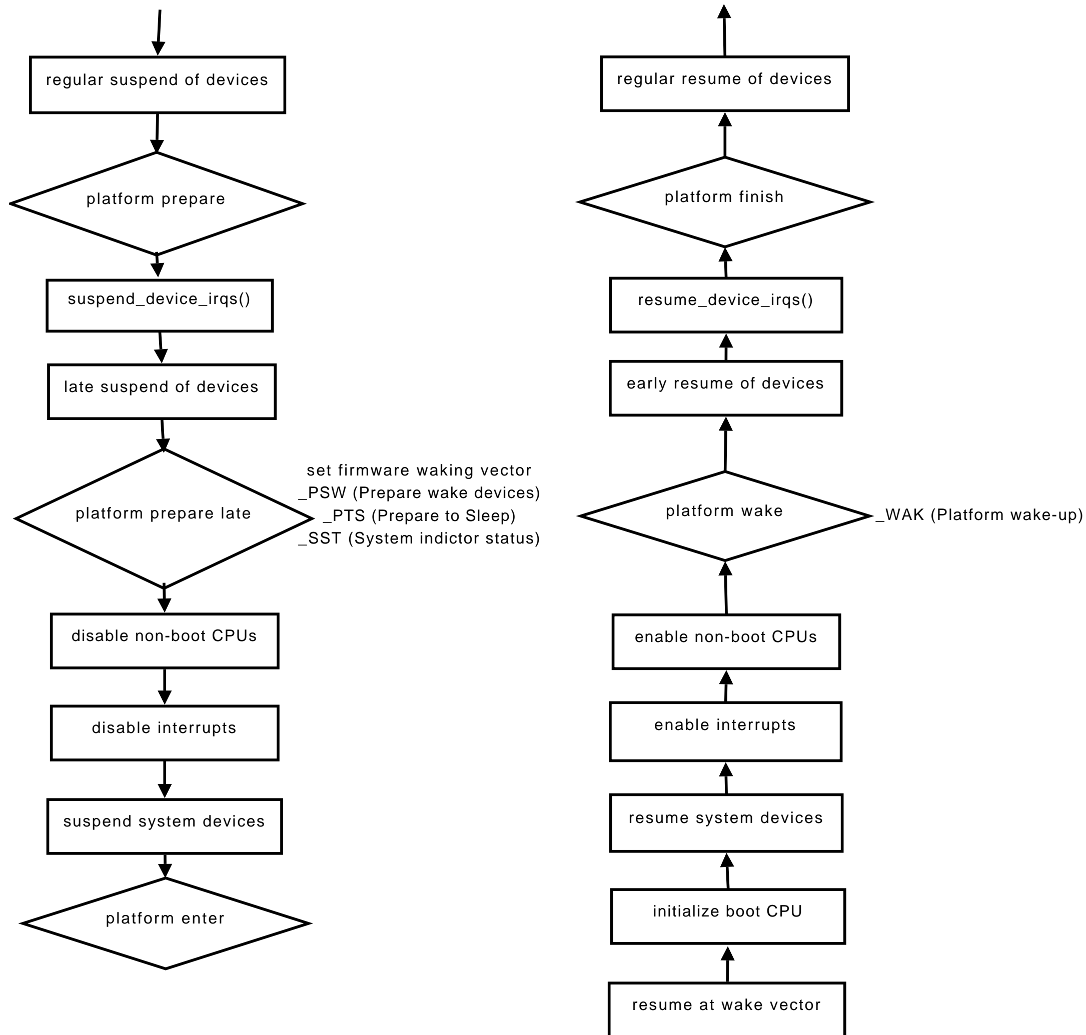

Figure 4: Suspend (left) and resume (right) code structure (2.6.30-rc3 and later).

interrupts to abort suspend, if necessary, had to be taken into consideration. Specifically, in the 2.6.29 and earlier kernels it was possible to mark an interrupt vector with the `IRQ_WAKEUP` status flag and make the platform abort suspend if that interrupt was pending after *sysdevs* had been suspended[10]. However, that might not work any more after introducing `suspend_device_irqs()` and `resume_device_irqs()` and rearranging the core suspend and resume code as described above, because the wake-up interrupts generated after the late suspend of devices would be acknowledged by one of the CPUs. In that case they would not appear to the platform code as pending, since they had been acknowledged by the CPU and the suspend would not be aborted. That would have been a significant change of behavior relative to the 2.6.29 kernel and we had to avoid it. For this purpose we used the observation that the `IRQ_PENDING` flag was set for interrupts acknowledged by the CPUs after `suspend_device_irqs()` had run, so it was sufficient to check this flag along with `IRQ_WAKEUP` after putting the non-boot CPUs off line and disabling interrupts on the remaining active one. Therefore, we introduced the function `check_wakeup_irqs()` shown in Figure 7 and made `sysdev_suspend()` call it and fail if the error code was returned [RJW3].

With all of the above modifications in place we could change `pci_restore_standard_config()` so that it used `pci_set_power_state()` to put devices into $D0$ before attempting to restore their standard PCI configuration registers with the help of `pci_`

[10] The *x86* architecture has never used wake-up interrupts.

```
void suspend_device_irqs(void) {
  struct irq_desc *desc;
  int irq;

  for_each_irq_desc(irq, desc) {
    unsigned long flags;

    spin_lock_irqsave(&desc->lock, flags);
    __disable_irq(desc, irq, true);
    spin_unlock_irqrestore(&desc->lock, flags);
  }

  for_each_irq_desc(irq, desc)
    if (desc->status & IRQ_SUSPENDED)
      synchronize_irq(irq);
}
```

Figure 5: Function called during suspend to prevent device drivers from receiving interrupts (2.6.30-rc3 and later)

```
void resume_device_irqs(void) {
  struct irq_desc *desc;
  int irq;

  for_each_irq_desc(irq, desc) {
    unsigned long flags;

    if (!(desc->status & IRQ_SUSPENDED))
      continue;

    spin_lock_irqsave(&desc->lock, flags);
    __enable_irq(desc, irq, true);
    spin_unlock_irqrestore(&desc->lock, flags);
  }
}
```

Figure 6: Function called during resume to allow device drivers to receive interrupts (2.6.30-rc3 and later)

`restore_state()` [RJW4]. As a result, it has been substantially simplified[11] as shown in Figure 8. We also changed the suspend callbacks of the `pci_bus_type` driver so that the late suspend callback saved the configuration spaces of the devices for which they were not saved by the drivers [RJW5]. This made it possible to develop a working PCI device driver supporting power management that would not touch the standard PCI configuration registers of the device in its suspend and resume callbacks allowing the PCI PM core to handle them as appropriate [RJW6].

[11]It also has been moved to `drivers/pci/pci-driver.c`.

```
int check_wakeup_irqs(void) {
  struct irq_desc *desc;
  int irq;

  for_each_irq_desc(irq, desc)
    if ((desc->status & IRQ_WAKEUP)
        && (desc->status & IRQ_PENDING))
      return -EBUSY;

  return 0;
}
```

Figure 7: Function called to check for wake-up interrupts right before suspending sysdevs (2.6.30-rc3 and later)

```
static int pci_restore_standard_config(struct pci_dev
                                       *pci_dev) {
  pci_update_current_state(pci_dev, PCI_UNKNOWN);

  if (pci_dev->current_state != PCI_D0) {
    int error = pci_set_power_state(pci_dev, PCI_D0);
    if (error)
      return error;
  }

  return pci_dev->state_saved ?
    pci_restore_state(pci_dev) : 0;
}
```

Figure 8: Function called during early resume to restore configuration space of a PCI device (2.6.30-rc3 and later)

5 Consequences

Using the approach presented in Section 4 has profound consequences for the writers of PCI device drivers wanting to support suspend and resume.[12] Namely, it allows them to let the PCI power management (PM) core take care of the "ugly" details related to PCI power management, such as the saving and restoration of the standard configuration registers, putting the device into a low power state during suspend and into $D0$ during resume, and preparing it to wake up the system from the memory sleep state, if desired. Since PCI device drivers have traditionally had problems with getting these things right, allowing them to leave it all to the core appears to be a big improvement. Of course, the drivers can still power manage the devices by themselves, which may even be

[12]In our not so humble opinion, every new PCI device driver ought to support suspend and resume.

necessary in some more complicated cases, but generally it is better if their authors avoid doing that, unless they know *very well* what they are doing.

In general, PCI device drivers can implement suspend and resume callbacks in two different ways. First, they can implement the `.suspend()`, `.suspend_late()`, `.resume()`, and `.resume_early()` callbacks available in the `pci_driver` structure, as shown in Figure 9. In that case, as indicated by the names of the callbacks, `.suspend()` will be executed in the first phase of suspending devices (regular suspend), `.suspend_late()` will be executed in the second phase of suspending devices (late suspend) and analogously for the resume callbacks. Drivers do not have to implement all of these callbacks, but if at least one of them is implemented, the PCI PM core will regard the driver as a "legacy" one and will apply special rules to the device handled by it. In particular, such a device will not be power managed by the PCI PM core during suspend and it will not be prepared by the core to wake up the system. Apart from this, the suspend and resume callbacks defined in `pci_driver` are used for hibernation as well as for suspend to RAM. Accordingly, the suspend callbacks take an additional argument of type `pm_message_t` specifying the context in which they are called (suspend to RAM or hibernation), but the resume callbacks do not take any additional arguments, so it is the driver's responsibility to preserve the context information over the suspend-resume (or hibernation-resume) cycle if needed.

```
struct pci_driver {
  ...
  int (*suspend) (struct pci_dev *dev, pm_message_t state);
  int (*suspend_late) (struct pci_dev *dev, pm_message_t state);
  int (*resume_early) (struct pci_dev *dev);
  int (*resume) (struct pci_dev *dev);
  ...
};
```

Figure 9: Members of `struct pci_driver` used during suspend to RAM and resume

The second way to implement suspend and resume callbacks in a PCI device driver is to use a `dev_pm_ops` object pointed to by the `driver.pm` member of the `pci_driver` structure.[13] This object, if present, contains pointers to several power management callbacks that can be implemented by a device driver. The majority of them are hibernation-specific and we are not going to discuss them here, but the ones shown in Figure 10 are used during suspend to RAM and resume. Still, the first two of them, `.prepare()` and `.complete()`, are only of interest to the authors of complicated drivers involving the management of children devices that may be registered and unregistered at any time, so we will not discuss them either. The remaining four callbacks are direct counterparts of the "legacy" ones discussed in the previous paragraph, where the `_noirq` suffix in the name means that the callback is a "late" or "early" one. In other words, `.suspend_noirq()` plays the role of `.suspend_late()` discussed previously and analogously for `.resume_noirq()`.

```
struct dev_pm_ops {
  int (*prepare)(struct device *dev);
  void (*complete)(struct device *dev);
  int (*suspend)(struct device *dev);
  int (*resume)(struct device *dev);
  ...
  int (*suspend_noirq)(struct device *dev);
  int (*resume_noirq)(struct device *dev);
  ...
};
```

Figure 10: Members of `struct dev_pm_ops` used during suspend to RAM and resume

As already stated, a PCI device driver implementing the "legacy" suspend and resume callbacks *has to* take care of putting the device into a low power state and, if necessary, preparing it to wake up the system during suspend, although it need not put the device into *D*0 during resume, since the PCI PM core is going to do that anyway via `pci_restore_standard_config()`. Moreover, during suspend the device should be put into the low power state by `.suspend_late()`, since otherwise the problem described in Section 2 may appear. In turn, a PCI device driver supporting suspend and resume through a `dev_pm_ops` object need not power manage the device during suspend and resume at all. However, if its author decides to put device into a low power state[14] and prepare it for waking up the system during suspend, these operations should be carried out in `.suspend_noirq()` to avoid the problem described in Section 2. Thus, it is possible to consider the "late" and "early" callbacks as the ones in which the actual power management of the device takes place, while

[13] The `struct dev_pm_ops` structure is defined in `include/linux/pm.h`.

[14] The standard PCI configuration registers of the device must be saved before that happens, since otherwise the restoration of their contents during resume may lead to undefined behavior.

the remaining "regular" callbacks can be regarded as the ones causing the driver to stop—`.suspend()`—or start—`.resume()`—using the device without changing the power state and related properties of it. Of course, in the kernels preceding 2.6.30-rc1 it was impossible to classify the suspend and resume callbacks provided by device drivers this way.

As follows from the above discussion, it is recommended and in the majority of cases more convenient to support suspend and resume by using a `dev_pm_ops` object rather then by implementing the "legacy" suspend and resume callbacks. Thus, it seems reasonable to give an example showing how to replace the "legacy" callbacks by a `dev_pm_ops` object in an existing driver and clearly illustrating the benefit of doing so. For this purpose consider the suspend and resume callbacks implemented by the `r8169` network driver shown in Figures 11 and 12, respectively, and observe that the PCI-specific operations carried out by `rtl8169_suspend()` should in fact be moved to a "late" suspend callback. Still, that will not be necessary if the driver provides the support for suspend and resume through a `dev_pm_ops` object.

```
static int rtl8169_suspend(struct pci_dev *pdev, pm_message_t state)
{
  struct net_device *dev = pci_get_drvdata(pdev);
  struct rtl8169_private *tp = netdev_priv(dev);
  void __iomem *ioaddr = tp->mmio_addr;

  if (!netif_running(dev))
    goto out_pci_suspend;

  netif_device_detach(dev);
  netif_stop_queue(dev);

  spin_lock_irq(&tp->lock);
  rtl8169_asic_down(ioaddr);
  rtl8169_rx_missed(dev, ioaddr);
  spin_unlock_irq(&tp->lock);

out_pci_suspend:
  pci_save_state(pdev);
  pci_enable_wake(pdev, pci_choose_state(pdev, state),
    (tp->features & RTL_FEATURE_WOL) ? 1 : 0);
  pci_set_power_state(pdev, pci_choose_state(pdev, state));

  return 0;
}
```

Figure 11: Suspend callback of the `r8169` driver (2.6.29)

To replace the "legacy" callbacks provided by the `r8169` driver with an implementation based on a `dev_pm_ops` object one can move the non-PCI part of `rtl8169_suspend()` to a separate function, like the one shown in Figure 13, and drop all of the PCI-specific

```
static int rtl8169_resume(struct pci_dev *pdev) {
  struct net_device *dev = pci_get_drvdata(pdev);

  pci_set_power_state(pdev, PCI_D0);
  pci_restore_state(pdev);
  pci_enable_wake(pdev, PCI_D0, 0);

  if (!netif_running(dev))
    goto out;

  netif_device_attach(dev);

  rtl8169_schedule_work(dev, rtl8169_reset_task);
out:
  return 0;
}
```

Figure 12: Resume callback of the `r8169` driver (2.6.29)

operations from both the suspend and resume routines. Of course, the `dev_pm_ops` object has to be defined too, but this is really straightforward as illustrated by the code in Figure 14 showing a possible implementation of it for the `r8169` driver[15]. Finally, the `driver.pm` member of the driver's `pci_driver` object has to be made point to the `dev_pm_ops` object defined by the driver, like the `rtl8169_pm_ops` object shown in Figure 14 in this particular case.

```
static void rtl8169_net_suspend(struct net_device *dev) {
  struct rtl8169_private *tp = netdev_priv(dev);
  void __iomem *ioaddr = tp->mmio_addr;

  if (!netif_running(dev))
    return;

  netif_device_detach(dev);
  netif_stop_queue(dev);

  spin_lock_irq(&tp->lock);
  rtl8169_asic_down(ioaddr);
  rtl8169_rx_missed(dev, ioaddr);
  spin_unlock_irq(&tp->lock);
}
```

Figure 13: Non-PCI part of the `r8169` driver's suspend callback.

The benefit of implementing suspend and resume support in the new way, as illustrated in Figures 13 and 14, to the `r8169` driver is that it need not worry about the PCI-specific handling of the device during suspend

[15] This particular definition means that the suspend and resume callbacks are going to be used for hibernation as well as for suspend to RAM and the PCI PM core is supposed to take care of the PCI-specific handling of the device in both cases.

```
static int rtl8169_suspend(struct device *device) {
  struct pci_dev *pdev = to_pci_dev(device);
  struct net_device *dev = pci_get_drvdata(pdev);

  rtl8169_net_suspend(dev);

  return 0;
}

static int rtl8169_resume(struct device *device) {
  struct pci_dev *pdev = to_pci_dev(device);
  struct net_device *dev = pci_get_drvdata(pdev);

  if (!netif_running(dev))
    goto out;

  netif_device_attach(dev);

  rtl8169_schedule_work(dev, rtl8169_reset_task);
out:
  return 0;
}

static struct dev_pm_ops rtl8169_pm_ops = {
  .suspend = rtl8169_suspend,
  .resume = rtl8169_resume,
  .freeze = rtl8169_suspend,
  .thaw = rtl8169_resume,
  .poweroff = rtl8169_suspend,
  .restore = rtl8169_resume,
};
```

Figure 14: Simplified suspend and resume support for the `r8169` driver

and resume and it need not implement any "late" and "early" callbacks. Moreover, it can use the same simplified "regular" callbacks for both suspend to RAM and hibernation allowing the PCI PM core to take care of the PCI-specific operations that ought to be carried out in any of these cases. In fact, if that implementation of suspend and resume support is used, the driver only needs to stop using the device during the regular suspend of devices and start using it during the regular resume of devices without doing anything else. This turns out out be the case for the majority of PCI device drivers currently in the kernel tree.

6 Conclusion

By using the approach presented in Sections 3 and 4 to solve the problem described in Section 2 we made it possible to significantly simplify suspend and resume callbacks provided by PCI device drivers. In particular, as shown in Section 5, a PCI device driver's suspend and resume callbacks can be implemented in such a way that all of the PCI-specific operations related to the power management of the device will be carried out by the PCI PM core.

Since the PCI PM core is now going to put every PCI device into *D*0 and restore its standard configuration registers during the early resume of devices, PCI device drivers need not do that any more. Therefore, it would be reasonable to remove these operations from all of the existing resume callbacks provided by PCI device drivers. Moreover, the drivers that put devices into low power states in their regular suspend callbacks should be modified to do it in their late suspend callbacks. Still, it may be even more beneficial to use a `dev_pm_ops` object for implementing suspend and resume support, as illustrated in Section 5, in which case the driver can leave the PCI-specific power management of the device to the PCI PM core.

The changes discussed in Sections 3 and 4 also made it possible to look at the device drivers' suspend and resume callbacks from a new perspective. Namely, the "regular" suspend callback, executed in the first phase of suspend, can be regarded as the one that should make the driver stop using the device, while the "late" suspend callback can be treated as the one preparing the device to wake up the system, if necessary, and putting it into a low power state. Analogously, the "early" resume callback, executed during the early resume of devices, can be regarded as the one that should put the device into the full power state and restore its pre-suspend configuration, while the "regular" resume callback can be treated as the one preparing the driver to use the device again. Of course, for PCI devices some or even all of the tasks of the "late" suspend and "early" resume callbacks can be completed by the PCI PM core, as described above.

7 Acknowledgements

The author thanks Linus Torvalds for his help and support of the work presented in this paper, Ingo Molnar and Thomas Gleixner for their help with the modifications of the core interrupt management code and Jesse Barnes for his help with the PCI part. He also thanks everyone who commented and tested patches or contributed to the work presented in this paper in any other way.

References

[LINUS1] L. Torvalds, *Re: Regression from 2.6.26: Hibernation (possibly suspend) broken* (`http:`

//marc.info/?l=linux-kernel&m=122852860315385&w=4).

[PATCH1] R. J. Wysocki, *PCI PM: Restore standard config registers of all devices early* (http://marc.info/?l=linux-pci&m=123213931514157&w=2).

[LINUS2] L. Torvalds, *Re: PCI PM: Restore standard config registers of all devices early* (http://marc.info/?l=linux-kernel&m=123360797907858&w=2).

[ACPI-SPEC] *Advanced Configuration and Power Interface Specification* (http://www.acpi.info).

[S2RAM] L. Brown, R. J. Wysocki, *Suspend to RAM in Linux* (Proceedings of the Linux Symposium, Ottawa 2008 http://ols.fedoraproject.org/OLS/Reprints-2008/brown-reprint.pdf).

[LINUS3] L. Torvalds, *Re: What should PCI core do during suspend-resume?* (http://marc.info/?l=linux-netdev&m=123343922809244&w=2).

[PCI-PM] *PCI Bus Power Management Interface Specification* (http://www.pcisig.com/specifications/conventional/).

[LINUS4] L. Torvalds, *Re: PCI PM: Restore standard config registers of all devices early* (http://marc.info/?l=linux-kernel&m=123361303317554&w=2).

[ACPICA] ACPICA project home page (http://acpica.org).

[LINUS5] L. Torvalds, *Re: PCI PM: Restore standard config registers of all devices early* (http://marc.info/?l=linux-kernel&m=123361270416948&w=2).

[RJW1] R. J. Wysocki, *Re: PCI PM: Restore standard config registers of all devices early* (http://marc.info/?l=linux-kernel&m=123365338201811&w=2).

[RMK] R. King, *900af0d breaks some embedded suspend/resume* (http://marc.info/?l=linux-kernel&m=124026014415587&w=2).

[RJW2] R. J. Wysocki, *PM/Suspend: Introduce two new platform callbacks to avoid breakage* (http://marc.info/?l=linux-kernel&m=124022459914679&w=2).

[RJW3] R. J. Wysocki, *PM: Introduce functions for suspending and resuming device interrupts* (http://marc.info/?l=linux-kernel&m=123703066215140&w=2).

[RJW4] R. J. Wysocki, *PCI PM: Use `pci_set_power_state` during early resume* (http://marc.info/?l=linux-kernel&m=123703114515664&w=2).

[RJW5] R. J. Wysocki, *PCI PM: Put devices into low power states during late suspend (rev. 2)* (http://marc.info/?l=linux-kernel&m=123703114515667&w=2).

[RJW6] R. J. Wysocki, *NET/r8169: Rework suspend and resume* (http://marc.info/?l=linux-kernel&m=123895686321519&w=4).

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.