System Suspend and Device Interrupts

 Author: Rafael J. Wysocki <rafael.j.wysocki@intel.com>

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().

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.

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 sates, 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().

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.

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.