Linux 系统休眠和设备中断处理

一、设备 IRQ 的 suspend 和 resume

主要解决这样一个问题：在系统休眠过程中，如何 suspend 设备中断（IRQ）？在从休眠中唤醒的过程中，如何 resume 设备 IRQ？

一般而言，在系统 suspend 过程的后期，各个设备的 IRQ (interrupt request line)会被 disable 掉。具体的时间点是在各个设备的 late suspend 阶段之后。代码如下（删除了部分无关代码）：

```
static int suspend_enter(suspend_state_t state, bool *wakeup)

{……

error = dpm_suspend_late(PMSG_SUSPEND);－－－－－late suspend 阶段

error = platform_suspend_prepare_late(state);

下面的代码中会 disable 各个设备的 irq

error = dpm_suspend_noirq(PMSG_SUSPEND);－－－－进入 noirq 的阶段

error = platform_suspend_prepare_noirq(state);

……

}
```

在 dpm_suspend_noirq 函数中，会针对系统中的每一个 device，依次调用 device_suspend_noirq 来执行该设备 noirq 情况下的 suspend callback 函数，当然，在此之前会调用 suspend_device_irqs 函数来 disable 所有设备的 irq。

之所以这么做，其思路是这样的：在各个设备驱动完成了 late suspend 之后，按理说这些已经被 suspend 的设备不应该再触发中断了。如果还有一些设备没有被正确的 suspend，那么我们最好的策略是 mask 该设备的 irq，从而阻止中断的递交。此外，在过去的代码中（指 interrupt handler），我们对设备共享 IRQ 的情况处理的不是很好，存在这样的问题：在共享 IRQ 的设备们完成 suspend 之后，如果有中断触发，这时候设备驱动的 interrupt handler 并没有准备好。在有些场景下，interrupt handler 会访问已经 suspend 设备的 IO 地址空间，从而导致不可预知的 issue。这些 issue 很难 debug，因此，我们引入了 suspend_device_irqs()以及设备 noirq 阶段的 callback 函数。

系统 resume 过程中，在各个设备的 early resume 过程之前，各个设备的 IRQ 会被重新打开，具体代码如下（删除了部分无关代码）：

```
static int suspend_enter(suspend_state_t state, bool *wakeup)

{……
```

```
platform_resume_noirq(state);－－－－首先执行 noirq 阶段的 resume

dpm_resume_noirq(PMSG_RESUME);－－－－－－在这里会恢复 irq，然后进入 early resume 阶段

platform_resume_early(state);

dpm_resume_early(PMSG_RESUME);

……}
```

在 dpm_resume_noirq 函数中，会调用各个设备驱动的 noirq callback，在此之后，调用 resume_device_irqs 函数，完成各个设备 irq 的 enable。

二、关于 **IRQF_NO_SUSPEND Flag**

当然，有些中断需要在整个系统的 suspend-resume 过程中（包括在 noirq 阶段，包括将 nonboot CPU 推送到 offline 状态以及系统 resume 后，将其重新设置为 online 的阶段）保持能够触发的状态。一个简单的例子就是 timer 中断，此外 IPI 以及一些特殊目的设备中断也需要如此。

在中断申请的时候，IRQF_NO_SUSPEND flag 可以用来告知 IRQ subsystem，这个中断就是上一段文字中描述的那种中断：需要在系统的 suspend-resume 过程中保持 enable 状态。有了这个 flag，suspend_device_irqs 并不会 disable 该 IRQ，从而让该中断在随后的 suspend 和 resume 过程中，保持中断开启。当然，这并不能保证该中断可以将系统唤醒。如果想要达到唤醒的目的，请调用 enable_irq_wake。

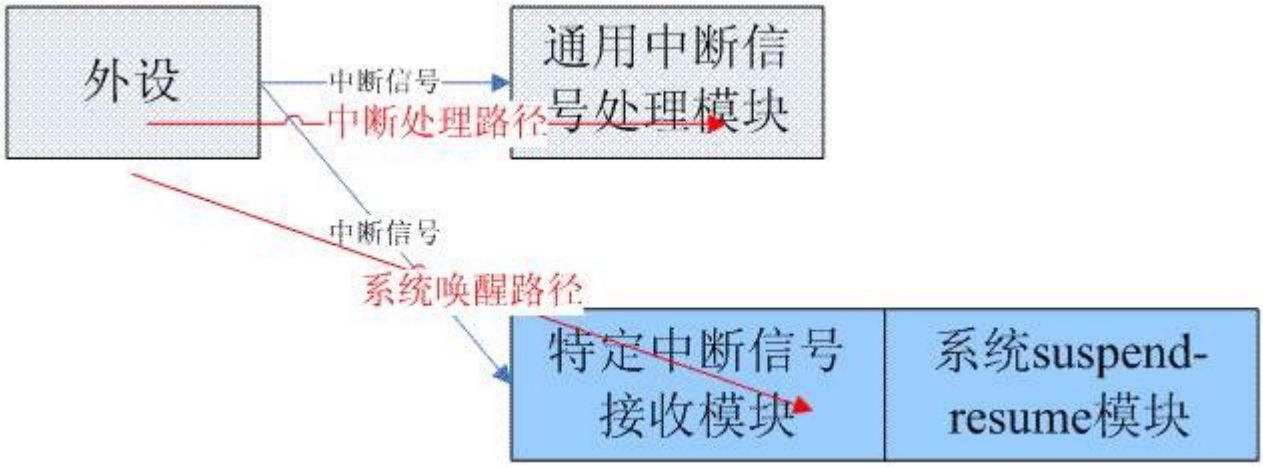
需要注意的是：IRQF_NO_SUSPEND flag 影响使用该 IRQ 的所有外设（一个 IRQ 可以被多个外设共享，不过 ARM 中不会这么用）。如果一个 IRQ 被多个外设共享，并且各个外设都注册了对应的 interrupt handler，如果其一在申请中断的时候使用了 IRQF_NO_SUSPEND flag，那么在系统 suspend 的时候（指 suspend_device_irqs 之后，按理说各个 IRQ 已经被 disable 了），所有该 IRQ 上的各个设备的 interrupt handler 都可以被正常的被触发执行，即便是有些设备在调用 request_irq(或者其他中断注册函数)的时候没有设定 IRQF_NO_SUSPEND flag。正因为如此，我们应该尽可能的避免同时使用 IRQF_NO_SUSPEND 和 IRQF_SHARED 这两个 flag。

三、系统中断唤醒接口：**enable_irq_wake()** 和 **disable_irq_wake()**

有些中断可以将系统从睡眠状态中唤醒，我们称之“可以唤醒系统的中断”，当然，“可以唤醒系统的中断”需要配置才能启动唤醒系统这样的功能。这样的中断一般在工作状态的时候就是作为普通 I/O interrupt 出现，只要在准备使能唤醒系统功能的时候，才会发起一些特别的配置和设定。

这样的配置和设定有可能是和硬件系统（例如 SOC）上的信号处理逻辑相关的，我们可以考虑下面的 HW block 图：

外设的中断信号被送到“通用的中断信号处理模块”和“特定中断信号接收模块”。正常工作的时候，我们会 turn on“通用的中断信号处理模块”的处理逻辑，而 turn off“特定中断信号接收模块” 的处理逻辑。但是，在系统进入睡眠状态的时候，有可能“通用的中断信号处理模块”已经 off 了，这时候，我们需要启动“特定中断信号接收模块”来接收中断信号，从而让系统 suspend-resume 模块（它往往是 suspend 状态时候唯一能够工作的 HW block 了）可以正常的被该中断信号唤醒。一旦唤醒，我们最好是 turn off“特定中断信号接收模块”，让外设的中断处理回到正常的工作模式，同时，也避免了系统 suspend-resume 模块收到不必要的干扰。



IRQ 子系统提供了两个接口函数来完成这个功能：enable_irq_wake()函数用来打开该外设中断线通往系统电源管理模块（也就是上面的 suspend-resume 模块）之路，另外一个接口是 disable_irq_wake()，用来关闭该外设中断线通往系统电源管理模块路径上的各种 HW block。

调用了 enable_irq_wake 会影响系统 suspend 过程中的 suspend_device_irqs 处理，代码如下：

```
static bool suspend_device_irq(struct irq_desc *desc)

{

……

if (irqd_is_wakeup_set(&desc->irq_data)) {

irqd_set(&desc->irq_data, IRQD_WAKEUP_ARMED);

return true;

}

省略 Disable 中断的代码

}
```

也就是说，一旦调用 enable_irq_wake 设定了该设备的中断作为系统 suspend 的唤醒源，那么在该外设的中断不会被 disable，只是被标记一个 IRQD_WAKEUP_ARMED 的标记。对于那些不是 wakeup source 的中断，在 suspend_device_irq 函数中会标记 IRQS_SUSPENDED 并 disable 该设备的 irq。在系统唤醒过程中（resume_device_irqs），被 diable 的中断会重新 enable。

当然，如果在 suspend 的过程中发生了某些事件（例如 wakeup source 产生了有效信号），从而导致本次 suspend abort，那么这个 abort 事件也会通知到 PM core 模块。事件并不需要被立刻通知到 PM core 模块，一般而言，suspend thread 会在某些点上去检查 pending 的 wakeup event。

在系统 suspend 的过程中，每一个来自 wakeup source 的中断都会终止 suspend 过程或者将系统

唤醒（如果系统已经进入 suspend 状态）。但是，在执行了 suspend_device_irqs 之后，普通的中断被屏蔽了，这时候，即便 HW 触发了中断信号也无法执行其 interrupt handler。作为 wakeup source 的 IRQ 会怎样呢？虽然它的中断没有被 mask 掉，但是其 interrupt handler 也不会执行（这时候的 HW Signal 只是用来唤醒系统）。唯一有机会执行的 interrupt handler 是那些标记 IRQF_NO_SUSPEND flag 的 IRQ，因为它们的中断始终是 enable 的。当然，这些中断不应该调用 enable_irq_wake 进行唤醒源的设定。

四、 **Interrupts and Suspend-to-Idle**

Suspend-to-idle (也被称为"freeze" 状态)是一个相对比较新的系统电源管理状态，相关代码如下：

```
static int suspend_enter(suspend_state_t state, bool *wakeup)

{

……

各个设备的 late suspend 阶段

各个设备的 noirq suspend 阶段

if (state == PM_SUSPEND_FREEZE) {

freeze_enter();

goto Platform_wake;

}

……

}
```

Freeze 和 suspend 的前面的操作基本是一样的：首先冻结系统中的进程，然后是 suspend 系统中的形形色色的 device，不一样的地方在 noirq suspend 完成之后，freeze 不会 disable 那些 non-BSP 的处理器和 syscore suspend 阶段，而是调用 freeze_enter 函数，把所有的处理器推送到 idle 状态。这时候，任何的 enable 的中断都可以将系统唤醒。而这也就意味着那些标记 IRQF_NO_SUSPEND（其 IRQ 没有在 suspend_device_irqs 过程中被 mask 掉）是有能力将处理器从 idle 状态中唤醒（不过，需要注意的是：这种信号并不会触发一个系统唤醒信号），而普通中断由于其 IRQ 被 disable 了，因此无法唤醒 idle 状态中的处理器。

那些能够唤醒系统的 wakeup interrupt 呢？由于其中断没有被 mask 掉，因此也可以将系统从 suspend-to-idle 状态中唤醒。整个过程和将系统从 suspend 状态中唤醒一样，唯一不同的是：将系统从 freeze 状态唤醒走的中断处理路径，而将系统从 suspend 状态唤醒走的唤醒处理路径，需要电源管理 HW BLOCK 中特别的中断处理逻辑的参与。

五、 **IRQF_NO_SUSPEND** 标志和 **enable_irq_wake** 函数不能同时使用

针对一个设备，在申请中断的时候使用 IRQF_NO_SUSPEND flag，又同时调用 enable_irq_wake 设定唤醒源是不合理的，主要原因如下：

1、如果 IRQ 没有共享，使用 IRQF_NO_SUSPEND flag 说明你想要在整个系统的 suspend-resume 过程中（包括 suspend_device_irqs 之后的阶段）保持中断打开以便正常的调用其 interrupt handler。而调用 enable_irq_wake 函数则说明你想要将该设备的 irq 信号设定为中断源，因此并不期望调用其 interrupt handler。而这两个需求明显是互斥的。

2、IRQF_NO_SUSPEND 标志和 enable_irq_wake 函数都不是针对一个 interrupt handler 的，而是针对该 IRQ 上的所有注册的 handler 的。在一个 IRQ 上共享唤醒源以及 no suspend 中断源是比较荒谬的。

不过，在非常特殊的场合下，一个 IRQ 可以被设定为 wakeup source，同时也设定 IRQF_NO_SUSPEND 标志。为了代码逻辑正确，该设备的驱动代码需要满足一些特别的需求。