### 18.3 Linux中断系统中的重要数据结构

本节内容，可以从request\_irq(include/linux/interrupt.h)函数一路分析得到。

能弄清楚下面这个图，对Linux中断系统的掌握也基本到位了。



最核心的结构体是irq\_desc，之前为了易于理解，我们说在Linux内核中有一个中断数组，对于每一个硬件中断，都有一个数组项，这个数组就是irq\_desc数组。

**注意**：如果内核配置了CONFIG\_SPARSE\_IRQ，那么它就会用基数树(radix tree)来代替irq\_desc数组。SPARSE的意思是“稀疏”，假设大小为1000的数组中只用到2个数组项，那不是浪费嘛？所以在中断比较“稀疏”的情况下可以用基数树来代替数组。

#### 18.3.1 irq\_desc数组

irq\_desc结构体在include/linux/irqdesc.h中定义，主要内容如下图：



每一个irq\_desc数组项中都有一个函数：handle\_irq，还有一个action链表。要理解它们，需要先看中断结构图：



外部设备1、外部设备n共享一个GPIO中断B，多个GPIO中断汇聚到GIC(通用中断控制器)的A号中断，GIC再去中断CPU。那么软件处理时就是反过来，先读取GIC获得中断号A，再细分出GPIO中断B，最后判断是哪一个外部芯片发生了中断。

所以，中断的处理函数来源有三：

① GIC的处理函数：

假设irq\_desc[A].handle\_irq是XXX\_gpio\_irq\_handler(XXX指厂家)，这个函数需要读取芯片的GPIO控制器，细分发生的是哪一个GPIO中断(假设是B)，再去调用irq\_desc[B]. handle\_irq。

**注意**：irq\_desc[A].handle\_irq细分出中断后B，调用对应的irq\_desc[B].handle\_irq。

显然中断A是CPU感受到的顶层的中断，GIC中断CPU时，CPU读取GIC状态得到中断A。

② 模块的中断处理函数：

比如对于GPIO模块向GIC发出的中断B，它的处理函数是irq\_desc[B].handle\_irq。

BSP开发人员会设置对应的处理函数，一般是handle\_level\_irq或handle\_edge\_irq，从名字上看是用来处理电平触发的中断、边沿触发的中断。

**注意**：导致GPIO中断B发生的原因很多，可能是外部设备1，可能是外部设备n，可能只是某一个设备，也可能是多个设备。所以irq\_desc[B].handle\_irq会调用某个链表里的函数，这些函数由外部设备提供。这些函数自行判断该中断是否自己产生，若是则处理。

③ 外部设备提供的处理函数：

这里说的“外部设备”可能是芯片，也可能总是简单的按键。它们的处理函数由自己驱动程序提供，这是最熟悉这个设备的“人”：它知道如何判断设备是否发生了中断，如何处理中断。

对于共享中断，比如GPIO中断B，它的中断来源可能有多个，每个中断源对应一个中断处理函数。所以irq\_desc[B]中应该有一个链表，存放着多个中断源的处理函数。

一旦程序确定发生了GPIO中断B，那么就会从链表里把那些函数取出来，一一执行。

这个链表就是action链表。

对于我们举的这个例子来说，irq\_desc数组如下：



#### 18.3.2 irqaction结构体

irqaction结构体在include/linux/interrupt.h中定义，主要内容如下图：



当调用request\_irq、request\_threaded\_irq注册中断处理函数时，内核就会构造一个irqaction结构体。在里面保存name、dev\_id等，最重要的是handler、thread\_fn、thread。

handler是中断处理的上半部函数，用来处理紧急的事情。

thread\_fn对应一个内核线程thread，当handler执行完毕，Linux内核会唤醒对应的内核线程。在内核线程里，会调用thread\_fn函数。

可以提供handler而不提供thread\_fn，就退化为一般的request\_irq函数。

可以不提供handler只提供thread\_fn，完全由内核线程来处理中断。

也可以既提供handler也提供thread\_fn，这就是中断上半部、下半部。

里面还有一个名为sedondary的irqaction结构体，它的作用以后再分析。

在reqeust\_irq时可以传入dev\_id，为何需要dev\_id？作用有2：

① 中断处理函数执行时，可以使用dev\_id

② 卸载中断时要传入dev\_id，这样才能在action链表中根据dev\_id找到对应项

所以在共享中断中必须提供dev\_id，非共享中断可以不提供。

#### 18.3.3 irq\_data结构体

irq\_data结构体在include/linux/irq.h中定义，主要内容如下图：



它就是个中转站，里面有irq\_chip指针 irq\_domain指针，都是指向别的结构体。

比较有意思的是irq、hwirq，irq是软件中断号，hwirq是硬件中断号。比如上面我们举的例子，在GPIO中断B是软件中断号，可以找到irq\_desc[B]这个数组项；GPIO里的第x号中断，这就是hwirq。

谁来建立irq、hwirq之间的联系呢？由irq\_domain来建立。irq\_domain会把本地的hwirq映射为全局的irq，什么意思？比如GPIO控制器里有第1号中断，UART模块里也有第1号中断，这两个“第1号中断”是不一样的，它们属于不同的“域”──irq\_domain。

#### 18.3.4 irq\_domain结构体

irq\_domain结构体在include/linux/irqdomain.h中定义，主要内容如下图：



当我们后面从设备树讲起，如何在设备树中指定中断，设备树的中断如何被转换为irq时，irq\_domain将会起到极大的作为。

这里基于入门的解度简单讲讲，在设备树中你会看到这样的属性：

interrupt-parent = <&gpio1>;

interrupts = <5 IRQ\_TYPE\_EDGE\_RISING>;

它表示要使用gpio1里的第5号中断，hwirq就是5。

但是我们在驱动中会使用request\_irq(irq, handler)这样的函数来注册中断，irq是什么？它是软件中断号，它应该从“gpio1的第5号中断”转换得来。

谁把hwirq转换为irq？由gpio1的相关数据结构，就是gpio1对应的irq\_domain结构体。

irq\_domain结构体中有一个irq\_domain\_ops结构体，里面有各种操作函数，主要是：

① xlate

用来解析设备树的中断属性，提取出hwirq、type等信息。

② map

把hwirq转换为irq。

#### 18.3.5 irq\_chip结构体

irq\_chip结构体在include/linux/irq.h中定义，主要内容如下图：



这个结构体跟“chip”即芯片相关，里面各成员的作用在头文件中也列得很清楚，摘录部分如下：

\* @irq\_startup: start up the interrupt (defaults to ->enable if NULL)

\* @irq\_shutdown: shut down the interrupt (defaults to ->disable if NULL)

\* @irq\_enable: enable the interrupt (defaults to chip->unmask if NULL)

\* @irq\_disable: disable the interrupt

\* @irq\_ack: start of a new interrupt

\* @irq\_mask: mask an interrupt source

\* @irq\_mask\_ack: ack and mask an interrupt source

\* @irq\_unmask: unmask an interrupt source

\* @irq\_eoi: end of interrupt

我们在request\_irq后，并不需要手工去使能中断，原因就是系统调用对应的irq\_chip里的函数帮我们使能了中断。

我们提供的中断处理函数中，也不需要执行主芯片相关的清中断操作，也是系统帮我们调用irq\_chip中的相关函数。

但是对于外部设备相关的清中断操作，还是需要我们自己做的。

就像上面图里的“外部设备1“、“外部设备n”，外设备千变万化，内核里可没有对应的清除中断操作。