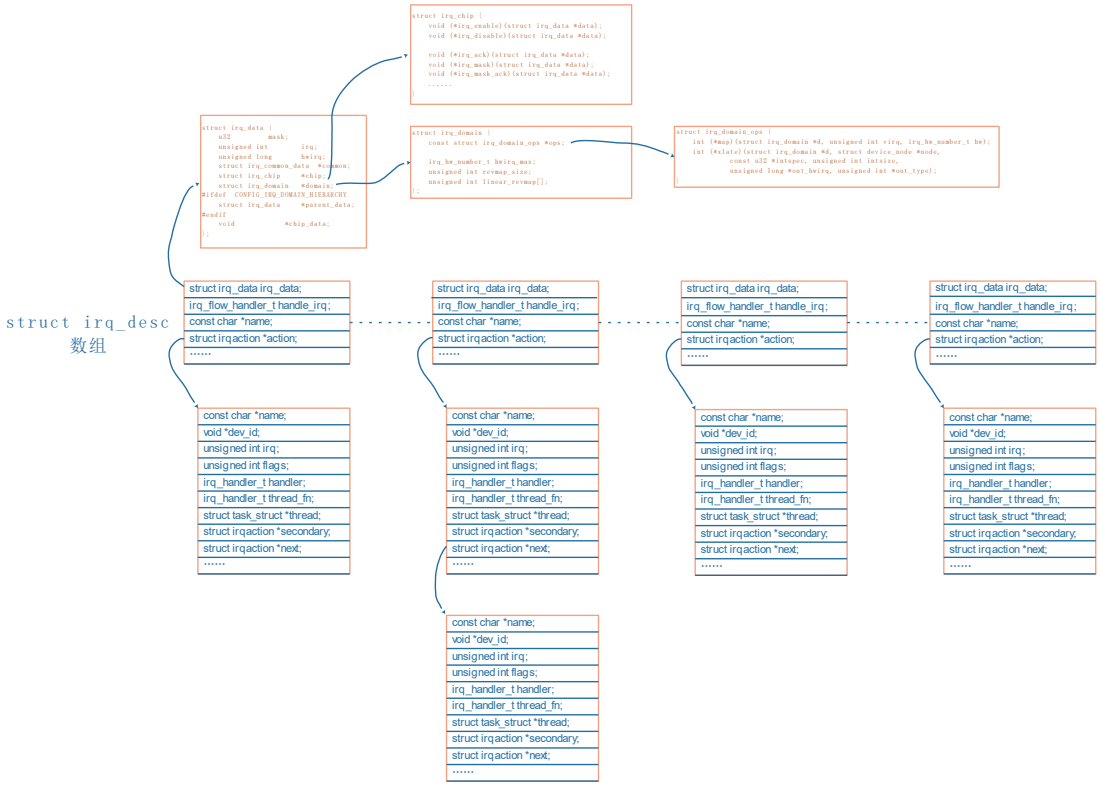
18.3 Linux 中断系统中的重要数据结构

能弄清楚下面这个图，对 Linux 中断系统的掌握也基本到位了。

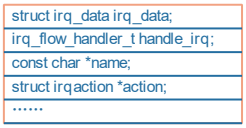


最核心的结构体是 irq_desc，之前为了易于理解，我们说在 Linux 内核中有一个中断数组，对于每一个硬件中断，都有一个数组项，这个数组就是 irq_desc 数组。

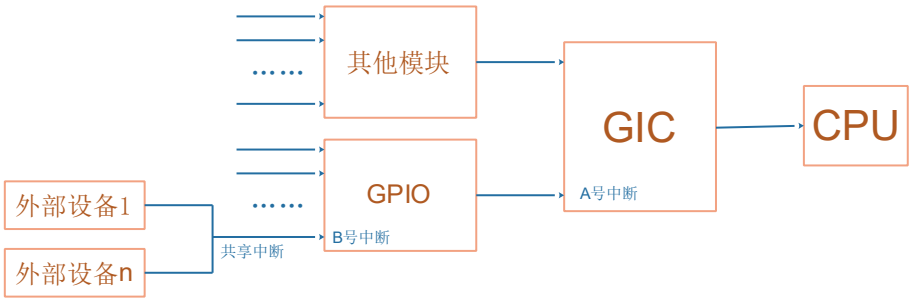
注意：如果内核配置了 CONFIG_SPARSE_IRQ，那么它就会用基数树(radix tree)来代替 irq_desc 数组。SPARSE 的意思是“稀疏”，假设大小为 1000 的数组中只用到 2 个数组项，那不是浪费嘛？所以在中断比较“稀疏”的情况下可以用基数树来代替数组。

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：

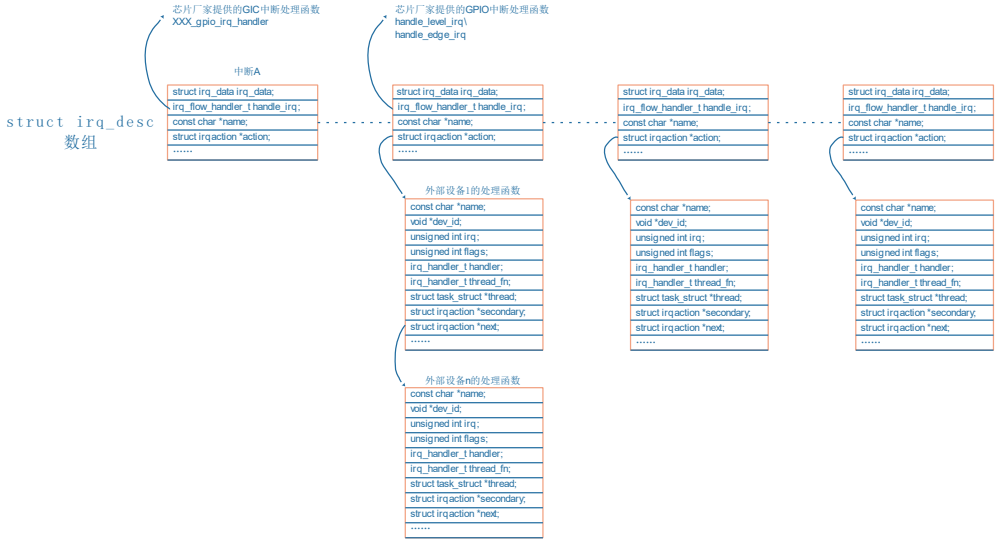
这里说的“外部设备”可能是芯片，也可能总是简单的按键。它们的处理函数由自己驱动程序提供，这是最熟悉这个设备的“人”：它知道如何判断设备是否发生了中断，如何处理中断。

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。

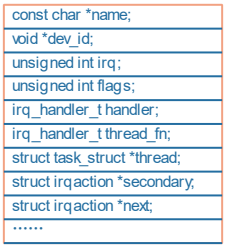
这个链表就是 action 链表。

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：

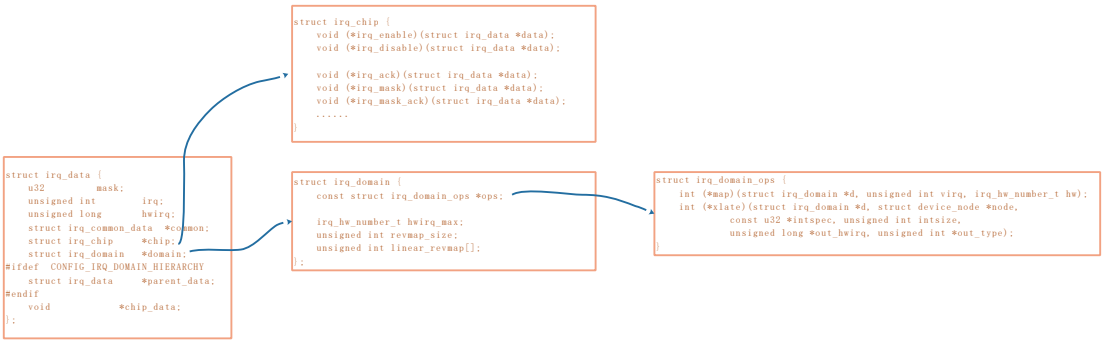
① 中断处理函数执行时，可以使用 dev_id

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：



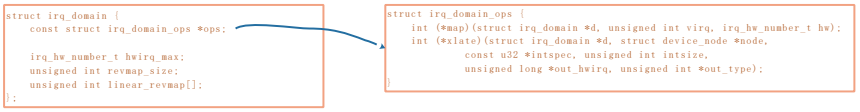
它就是个中转站，里面有 irq_chip 指针 irq_domain 指针，都是指向别的结构体。

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

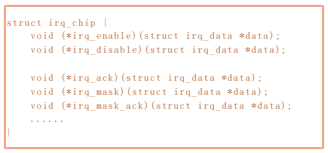
用来解析设备树的中断属性，提取出 hwirq、type 等信息。

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。