第 4 章 打下基础

我们现在已经与编写设备驱动之间的距 离已经非常逼近。但是，在此之前，让我们先装备一些驱动的概念。本章首先开始于对本书的问题陈述的理念，接下来分析 PC 兼 容的系统和嵌入式计算机中典型的设备和 I/O 接口。中断处理在大多数驱动中的都存在，因此，本章讨 论了编写中断服务程序的方法问题。之后，我们将注意力转移到了 2.6 内核中新引入的设备模型，该新 模型建立于 sysfs、kobject、设 备类、udev 等抽象事物上，它们是从设备驱动中提炼出来的有共性的东西。新的设备模型也需要内核 空间之外的策略，这些策略被推到用户空间，这导致了/dev 结点管理、热插拔、冷插拔、模块自动加 载、固件下载等功能的改变。

设 备和驱动介绍

由于对硬件的操作要求拥有执行特殊指令和处理中断等处理器特权，所以用户应用程序一般不能直接 和硬件通信。设备驱动则承担了硬件交互的工作，它也向应用程序和内核中其他的部分引出接口这些接口。应用程序通过/dev 目 录中的设备结点可对设备进行操作，通过/sys 目录中的结点可以收集设备信息[1]。

[1]以后你将学习到，网络应用程序通过不同的机制将请求发给底层驱动。

图 4.1 是一个典型的 PC 兼容的系统的硬件块结构图。从图中可以看出,系统支 持各种各样的设备和接口,如内存、视频、音频、USB、PCI、WiFi、I2C、IDE、以太网、串口、键盘、鼠标、软驱、并口 和红外等。内核控制器和图形控制器在 PC 体系结构中位于北桥芯片组中，然后外设总线则源自南桥芯片组。

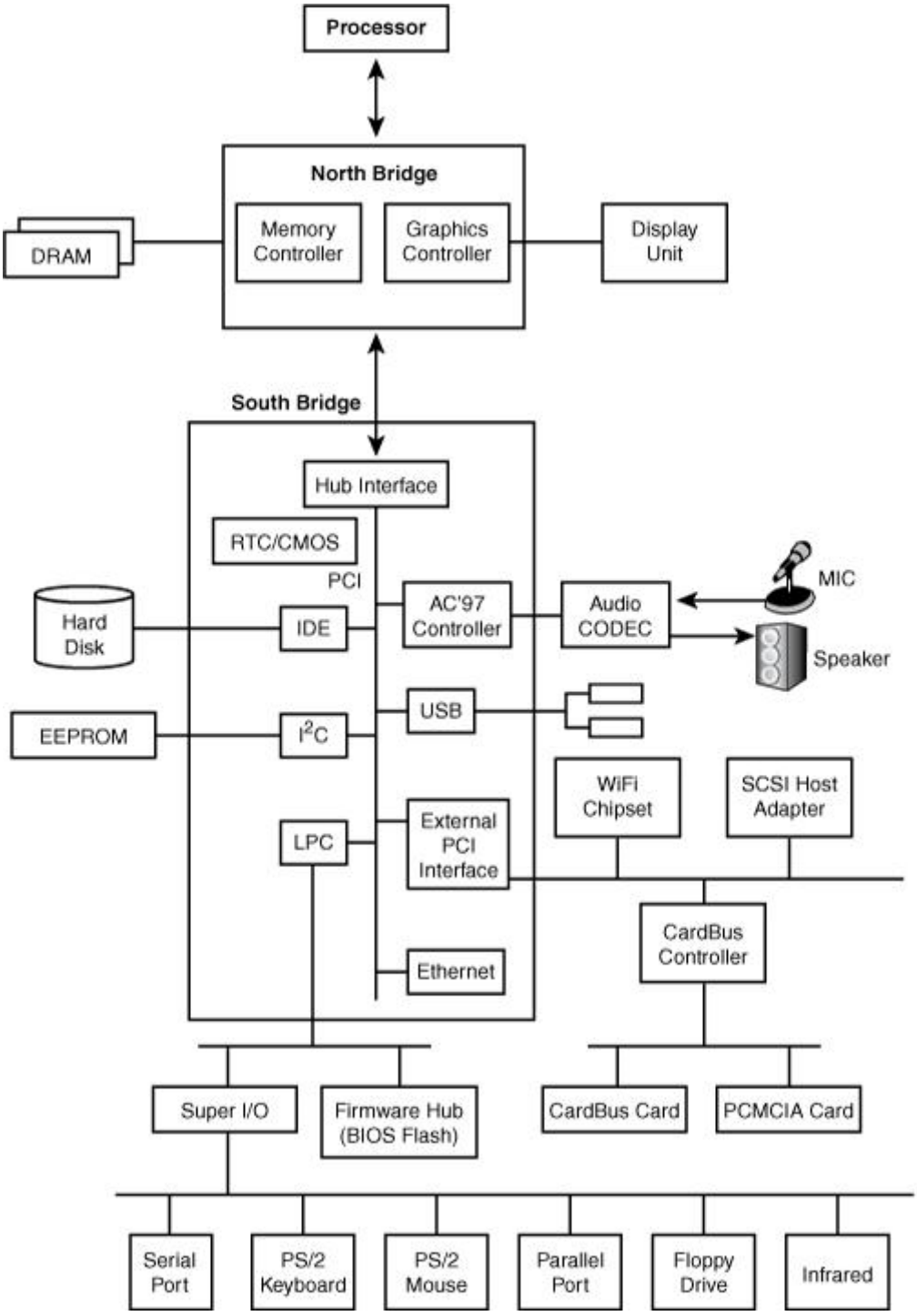
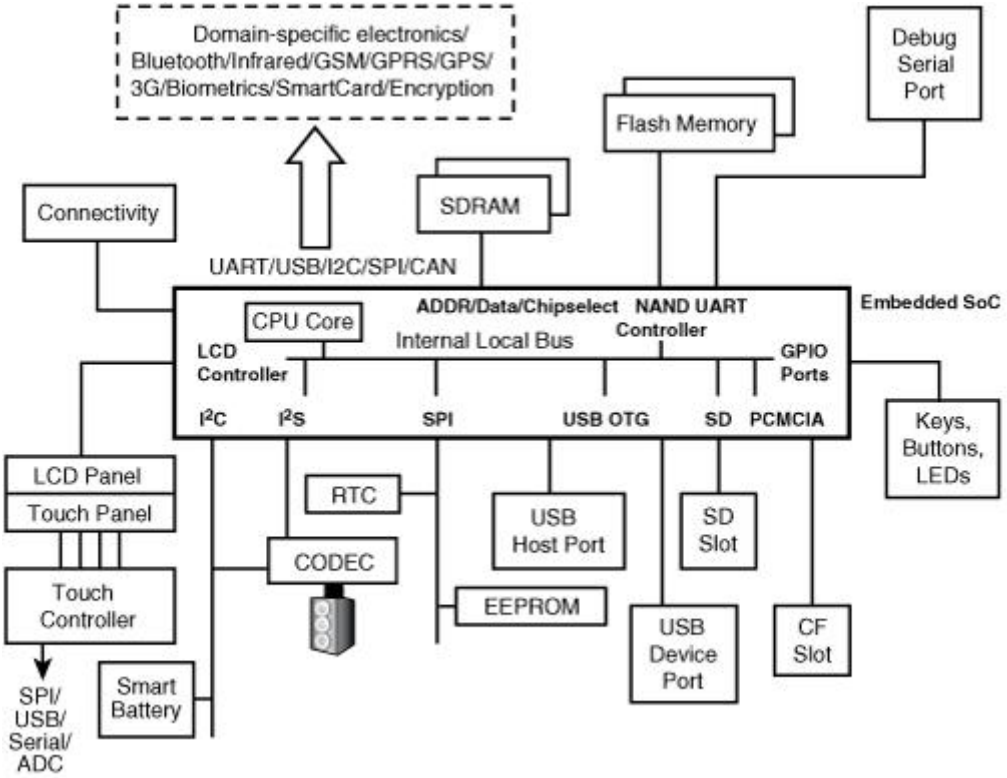


图 4.2 给出了一个假想的嵌入式设备的 类似于图 4.1 的块图。该图中包含了数个 PC 中 通常不存在的接口，如闪存、LCD、触摸屏和无线调制解调器。



显然，访问外设的能力是系统整体机能的重要组成部分。设备驱动提供了达到此目的的引擎。本书中 剩余的章节将聚焦于设备结构，并将会读者怎样实现相应的设备驱动。

中 断处理

由于 I/O 操作的不确定因素，以及处理 器和 I/O 设备之间速度的不匹配，设备往往通过某种硬件信号异步地唤起处理器的注意。这些硬件信号 就是所谓的中断。每个中断设备都被分配给一个相关的标识符，被称为中断请求（IRQ）号。当处理器 检测到某一 IRQ 号对应的中断产生时，它将停止它现在的工作，并引用该 IRQ 所对应的中断服务例程（ISR）。中断处理函数 ISR 在中断上下文执行。

中 断上下文

ISR 是与硬件交互的非常重要的代码片段。它们被给予了立即执行的特权，以便最大化系统的性能。不 过，如果 ISR 执行过慢、负载太重的化，就违背了自身的设计哲学。贵宾都被给予了优惠待遇，但是， 尽量减少由此造成的对公众的不便也是他们的义务。为了对粗暴打断当前执行线程的行为进行补偿，ISR 不 得不礼貌地执行于受限制的环境下，即所谓的中断上下文（或原子上下文）。

下面给出了中断上下文可为和不可为事项的列表：

1. 如果你的中断上下文进入睡眠，它是一项应该被处以监禁的罪行。中断 处理函数不能通过调用 schedule_timeout()等睡眠函数放弃处理器，在中断处理函数 中调用一个内核 API 之前，应该仔细分析它以确保其内部不会触发阻塞等待。例如，input_register_device()表面上看起来没有问题，但是它内部以 GFP_KERNEL 为参数调用了 kmalloc()。 从第 2 章《内核一瞥》可以看出，用这种方式调用 kmalloc()的 话，如果系统的空闲内存低于某门限，kmalloc()将睡眠等待 swapper 释放内存。

2. 为了在中断处理函数中保护临界区，你不能使用互斥体，因为它们也许导致睡眠。应该使用自旋锁代替 互斥体，但是一定要记住的是只有真正需要的时候才采用它。

3. 中断处理函数不能与用户空间直接交互数据，因为它们经由进程上下文与用户空间建立连接。这也是为 什么中断处理函数不能睡眠的第 2 个理由：调度器工作于进程之间，如果中断处理函数睡眠并被调度出 去，它们怎么返回到运行队列呢？

4. 中断处理函数一方面需要快速地出来，另一方面又需要完成它的工作。为了规避这种冲突，中断处理函 数通常被分成 2 个部分。瘦小的顶半部标志一个响应以宣称它已经服务了该中断，而重大的工作负载都被 丢给了肥胖的底半部。底半部的执行被延后，在其执行环境中，所有的中断都是使能的。在讨论 softirq 和 tasklet 的时候，你将学习到真也难怪开发底半部。

5. 中断处理函数不必是可重用的。当某中断被执行的时候，在它返回之前，相应的 IRQ 都被禁止了。因此，与进程上下文代码不同的是，同一中断处理函数的不同实例不可能同时运行在多个处理器 上。

6. 中断处理函数可以被更高优先级 IRQ 的中 断处理函数打断。如果你请求内核将你的中断处理函数作为快中断处理的话，此类中断嵌套将被禁止。快中断服务函数运行的时候，本处理器上的所有中断都会被禁 止。在禁止中断或将你的中断标识为快中断之前，请意识到中断屏蔽对系统性能的坏处。中断屏蔽的时间越长，中断延迟就会更长，或者说已经被产生的中断得到服 务的延迟就会越久。中断延迟与系统真实的响应时间成反比。

函数中可以检查 in_interrupt()的 返回值以查看自身是否位于中断上下文。

与外部硬件产生的异步中断不一样，也存在同步到达的中断。同步中断意味着它们不会不期而遇，它 们由处理器本身执行某指令而产生。外部中断和同步中断在内核中使用相同的机制处理。

同步中断的例子包括：

（1）异常，被用于报告严重的运行时错 误；

（2）软中断，如 int 0x80 指令，被用户实现 x86 体系结构上的系 统调用。

分配IRQ号

设备驱动必须将它们的 IRQ 号与一个中 断处理函数连接。因此，它们需要知道它们正在驱动的设备的 IRQ 号。IRQ 的分配可以很直接，也可能需要复杂的探测过程。在 PC 体系结构中，例如，定时器中断被分配了 IRQ 0，RTC 中 断也是 IRQ 8。现代的中断技术（如 PCI） 足够强大，它能够响应对 IRQ 的查询（系统启动过程中由 BIOS 分 配），PCI 驱动能够访问设备配置空间的相应区域并获得 IRQ。 对于较老的设备，如基于工业标准体系结构（ISA）的卡而言，驱动也许不得不利用特定硬件的知识以 探测和解析 IRQ。

通过/proc/interrupts 可 以查看系统中活动的 IRQ 的列表。

设备实例：辊轮

现在你已经学习了中断处理的基本知识，现在我们来实现一个辊轮设备实例的中断处理。在一些手机 和 PDA 上能找到辊轮，它支持 3 种动作（顺 时针旋转，逆时针旋转和按

键），可便利菜单导航。本例辊轮中的任何运行都会向处理器产生 IRQ 7。 通用目的 I/O（GPIO）端口 D 的低 3 位与辊轮设备连接。这些引脚上产生的波形与图 4.3 中不同的辊轮运动一致。中断处理函数的工作是通过查看端口 D 的 GPIO 数据寄存器解析出辊轮的运动。

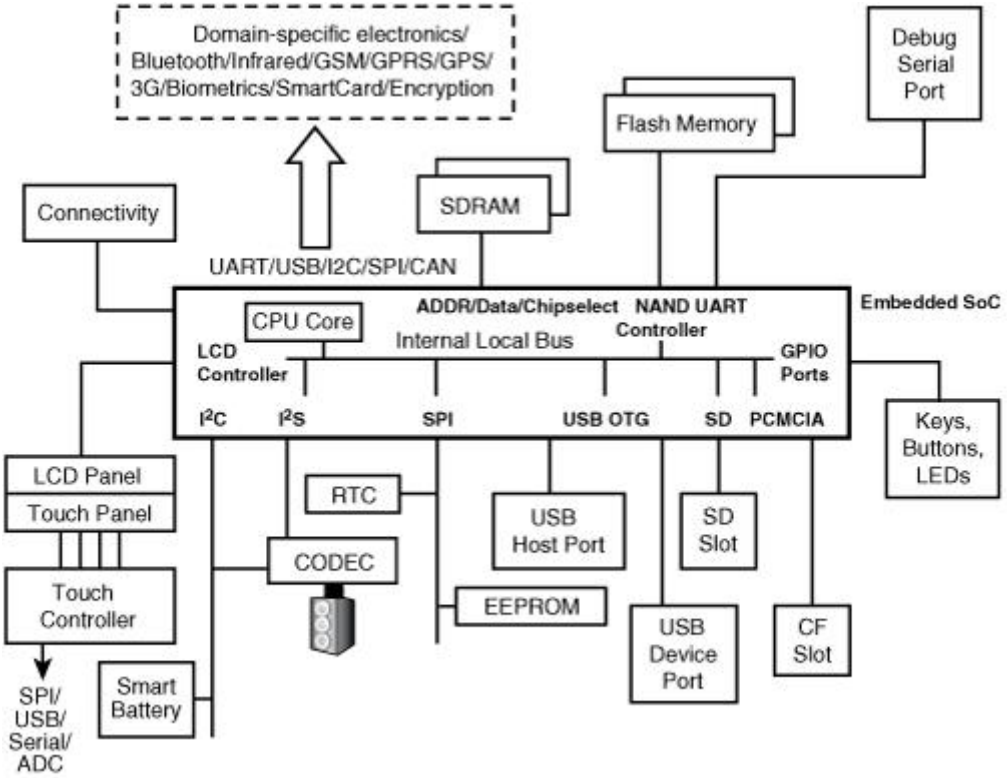


图 4.3 辊轮运动产生的波形

驱动必须首先请求 IRQ 并将一个中断处 理函数与其绑定：

```
#define ROLLER_IRQ  7

static irqreturn_t roller_interrupt(int irq, void *dev_id);




if (request_irq(ROLLER_IRQ, roller_interrupt, IRQF_DISABLED |

            IRQF_TRIGGER_RISING, "roll", NULL)) {

  printk(KERN_ERR "Roll: Can't register IRQ %d\n", ROLLER_IRQ);

  return -EIO;

}
```

我们看一下传递给 request_irq()的 参数，本例中没有查询或探测 IRQ 号，而是直接硬编码为 ROLLER_IRQ。 第 2 个参数 roller_interrupt()是 中断处理函数。中断处理函数的原型的返回值类型为 irqreturn_t，如果中断处理成功，则返 回 IRQ_HANDLED，否则，返回 IRQ_NONE。 对于 PCI 等 I/O 而言，该返回值的意义更 重要，因为多个设备可能共享同一 IRQ。

IRQF_DISABLED 标志意味着这个中断处理为快中断，因此，在调用该处理函数的时候，内核将禁止所有的中断。IRQF_TRIGGER_RISING 暗示辊轮将在中断线上产生一个上升沿以发出中断。换句话说，辊轮是一 个边沿触发的设备。有一些设备是电平触发的，在 CPU 服务其中断之前，它一直将中断线保持在一个电 平上。使用 IRQF_TRIGGER_HIGH 或 IRQF_TRIGGER_LOW 可 以标识一个中断为高/低电平触发。该参数其他的可能值包括 IRQF_SAMPLE_RANDOM（第 5 章《字符设备驱动》的《伪字符设备驱动》一节会用到）、IRQF_SHARED（定 义这个 IRQ 被多个设备共享）。

下一个参数"roll"，用于标识这个 设备，在/proc/interrupts 等文件中也会利用它产生数据。最后一个参数（本例中为 NULL），仅在共享中断的时候有用，用于区分共享同一 IRQ 线 的每个设备。

从 2.6.19 内核开始，中断处理接口发 生了一些变化。以前的中断处理函数的第 3 个参数为 struct pt_regs *，它指向存放 CPU 寄存器的地址，在 2.6.19 中已经移除。另外，IRQF_xxx 型中断 标志取代了 SA_xxx 型中断标志。例如，在较早的内核中，你应该使用 SA_INTERRUPT 而不是 IRQF_DISABLED 来 将中断处理标识为快中断处理。

驱动初始化的时候申请 IRQ 并不是太 好，因为这样会导致甚至设备未被使用的时候，有价值的资源也被占用。因此，设备驱动通常在设备被应用打开的时候申请 IRQ。 类似地，IRQ 也在应用关闭设备的时候释放 IRQ， 而不是在退出驱动模块的时候进行。使用下面的方法可以释放一个 IRQ：

```
free_irq(int irq, void *dev_id);
```

清单 4.1 给出了辊轮中断处理的实现。

roller_interrupt()有 2 个参数，IRQ 和设备标识符（传递给 request_irq()的 最后一个参数）。请对照图 4.3 查看清单 4.1。

清单 4.1 辊轮中断处理

```
spinlock_t roller_lock = SPIN_LOCK_UNLOCKED;

static DECLARE_WAIT_QUEUE_HEAD(roller_poll);



static irqreturn_t

roller_interrupt(int irq, void *dev_id)

{
```

```
int i, PA_t, PA_delta_t, movement = 0;

/* Get the waveforms from bits 0, 1 and 2
   of Port D as shown in Figure 4.3 */
PA_t = PORTD & 0x07;

/* Wait until the state of the pins change.
   (Add some timeout to the loop) */
for (i=0; (PA_t==PA_delta_t); i++){
  PA_delta_t = PORTD & 0x07;
}

movement = determine_movement(PA_t, PA_delta_t); /* See below */

spin_lock(&roller_lock);

/* Store the wheel movement in a buffer for
   later access by the read()/poll() entry points */
store_movements(movement);

spin_unlock(&roller_lock);

/* Wake up the poll entry point that might have
   gone to sleep, waiting for a wheel movement */
```

```
 wake_up_interruptible(&roller_poll);



 return IRQ_HANDLED;

}

int

determine_movement(int PA_t, int PA_delta_t)

{

 switch (PA_t){

  case 0:

   switch (PA_delta_t){

   case 1:

    movement = ANTICLOCKWISE;

    break;

   case 2:

    movement = CLOCKWISE;

    break;

   case 4:

    movement = KEYPRESSED;

    break;

   }

   break;

  case 1:

   switch (PA_delta_t){

   case 3:
```

```
        movement = ANTICLOCKWISE;
        break;
      case 0:
        movement = CLOCKWISE;
        break;
      }
      break;
    case 2:
      switch (PA_delta_t){
      case 0:
        movement = ANTICLOCKWISE;
        break;
      case 3:
        movement = CLOCKWISE;
        break;
      }
      break;
    case 3:
      switch (PA_delta_t){
      case 2:
        movement = ANTICLOCKWISE;
        break;
      case 1:
        movement = CLOCKWISE;
```

```
        break;
      }
    case 4:
      movement = KEYPRESSED;
      break;
  }
}
```

驱动入口点(如 read()和 poll())尾随 roller_interrupt()进行操作。例如，当中断处理函数解析完一个辊轮运动后，它唤醒正在等待的 poll()线程（可能已经因为 X Windows 等 应用发起的 select()系统调用而睡眠）。请在学习完第 5 章 字符设备驱动的知识后，重新查看清单 4.1 并实现辊轮设备的完整驱动。

第 7 章《输入设备驱动》的清单 7.3 利用了内核的输入接口，将辊轮转化为辊鼠标。

在本节结束前，我们介绍一下使能和禁止特定 IRQ 的 函数。enable_irq(ROLLER_IRQ)用于使能辊轮运动的中断发生，disable_irq(ROLLER_IRQ)则进行相反的工作。disable_irq_nosync(ROLLER_IRQ)禁 止辊轮中断，并且不等待任何正在执行的 roller_interrupt()实例的返回。disable_irq()的非同步变体执行地更快，但是可能导致潜在的竞态。只有在你确认没有竞争的尽快 下，才可以这样使用。drivers/ide/ide-io.c 由一个使用 disable_irq_nosync()的例子，在初始化过程中，它阻止了一些中断，因为一些系统中可能在此 方面存在问题。

软中断（Softirq）和Tasklet

正如以前讨论的那样，中断处理有 2 个矛 盾的要求：它们需要完成大量的设备数据处理，但是又不得不尽可能快地退出。为了摆脱这一困境，中断处理过程被分成 2 部 分：一个急切的且抢占的与硬件交互的顶半部，和一个在所有中断都使能情况下并非十分急切的处理大量工作的底半部。如顶半部不一样，底半部是同步的，因为内 核决定了它什么时候会执行它们。如下机制都可用于内核中延后一个工作到底半部执行：softirq、tasklet 和工作队列（work queue）。

Softirq 是一种基本的底半部机制，有较强的加锁需求。仅仅在一些对性能敏感的子系统（如网络层、SCSI 层和内核定时器）中才会使用 softirq。Tasklet 建立在 softirq 之上，使用起来更 简单。除非有严格的可扩展性和速度要求，都建议使用 Tasklet。Softirq 和 Tasklet 的主要不同是前者是可重 用的而后者则不需要。Softirq 的不同实例可运行在不同的处理器上，而 tasklet 则不允许。

为了论证 Softirq 和 Tasklet 的用法，假定前例中的辊轮由存在由于运动部件导致的潜在问题（如旋轮偶尔被卡住）从而导致不同于 spec 的波形。一个被卡住的旋轮会不停地产生假的中断，并可能使系统冻结。为了解决这个问题，可以捕获波 形，进行一些分析，并在发现卡住的情况下动态地从中断模式切换到轮询模式，如果旋轮恢复正常，软件也恢复到正常模式。我们在中断处理函数中捕获波形，并在 底半部分析它。清单 4.2 和 4.3 分别用 Softirq 和 Tasklet 对此进行了实现。

它们都是清单 4.1 的简化的变体，它们 将中断处理简化为 2 个函数：从 GPIO 端口 D 捕获波形的 roller_capture()和对波 形进行算术分析并按需切换到轮询模式的 roller_analyze()。

清单 4.2 使用Softirq 分担中断处理的负载

```
void __init
roller_init()
{
  /* ... */


  /* Open the softirq. Add an entry for ROLLER_SOFT_IRQ in
     the enum list in include/linux/interrupt.h */
  open_softirq(ROLLER_SOFT_IRQ, roller_analyze, NULL);
}




/* The bottom half */
void
roller_analyze()
{
  /* Analyze the waveforms and switch to polled mode if required */
}
/* The interrupt handler */
static irqreturn_t
roller_interrupt(int irq, void *dev_id)
{
```

```
  /* Capture the wave stream */

  roller_capture();



  /* Mark softirq as pending */

  raise_softirq(ROLLER_SOFT_IRQ);



  return IRQ_HANDLED;

}
```

为了定义一个 softirq，你必须在 include/linux/interrupt.h 中 静态地添加一个入口。你不能动态地定义 softirq。raise_softirq()用 于宣布相应的 softirq 需要被执行。内核会在下一个可获得的机会里执行它。可能发生在退出硬中 断处理函数的时候，也可能在 ksoftirqd 内核线程中。

清单 4.3 使用tasklet分担中断处理的负载

```
struct roller_device_struct { /* Device-specific structure */

  /* ... */

  struct tasklet_struct tsklt;

  /* ... */

}



void __init roller_init()

{

  struct roller_device_struct *dev_struct;

  /* ... */



  /* Initialize tasklet */

  tasklet_init(&dev_struct->tsklt, roller_analyze, dev);
```

```
}



/* The bottom half */
void
roller_analyze()
{
/* Analyze the waveforms and switch to
   polled mode if required */
}
/* The interrupt handler */
static irqreturn_t
roller_interrupt(int irq, void *dev_id)
{
  struct roller_device_struct *dev_struct;

  /* Capture the wave stream */
  roller_capture();

  /* Mark tasklet as pending */
  tasklet_schedule(&dev_struct->tsklt);

  return IRQ_HANDLED;
}
```

tasklet_init()用于动态地初始化一个 tasklet，该函数不会为 tasklet_struct 分配内存，相反地，你必须将已经分配好的地址传递给它。tasklet_schedule()用于宣布相应的 tasklet 需 要被执行。和中断类似，内核提供了一系列用于控制在多处理器系统中 tasklet 执行状态的函数：

（1）tasklet_enable()使能 tasklet；

（2）tasklet_disable()禁止 tasklet， 并等待正在执行的 tasklet 退出；

（3）tasklet_disable_nosync()的语义和 disable_irq_nosync()相 似，它并不等待正在执行的 tasklet 退出。

你已经看到了中断处理函数和底半部的不同，但是，也有几个相似点。中断处理函数和 tasklet 都不需要可重用。而且，二者都不能睡眠。另外，中断处理函数、tasklet 和 softirq 都不能被抢占。

工作队列是中断处理延后执行的第 3 种方 式。它们在进程上下文执行，允许睡眠，因此可以使用 mutex 这类可能导致睡眠的函数。在前一章分 析内核辅助接口的时候，我们已经讨论了工作队列。表 4.1 对 softirq、tasklet 和工作队列进行了对比分析。

表 4.1 softirq、tasklet和工作队列对比

	Softirq	Tasklet	Work Queue
执行上下文	延 后的工作，运行于中断上下文	延 后的工作，运行于中断上下文	延 后的工作，运行于进程上下文
可重用	可 以在不同的 CPU 上同时运行	不 能在不同的 CPU 上同时运行，但是不同的 CPU 可以运行不同的 tasklet	可 以在不同的 CPU 上同时运行
睡眠	不 能睡眠	不 能睡眠	可 以睡眠
抢占	不 能抢占/调度	不 能抢占/调度	可 以抢占/调度
易用性	不 容易使用	容 易使用	容 易使用
何时使用	如 果延后的工作不会睡眠，而且有严格的可扩展性或速度要求	如果延后的工作不会睡眠	如果延后的工作会睡眠

在 LKML 正在进行一项去除 tasklet 的可行性的讨 论。Tasklet 比进程上下文的代码优先级更高，因此它们可能存在延迟问题。另外，你已经学习 到，它们不允许睡眠，且只能在同一 CPU 上执行。因此，有人提议将现存的 tasklet 基于其场景随机应变地转换为 softirq 或 工作队列。

第 2 章讨论的 -rt 补丁集将中断处理移到了内核线程执行，以实现更广泛的抢占支持。

Linux设备 模型

新的 Linux 设备模型引入了类似于 C++的抽象机制，它总结出设备驱动的共性，并提取出了总线和核心层。接下来，我们分析一下设备模型中的 udev、sysfs、kobject 和设备类（device class）等 组件，以及这些组件对/dev 结点管理、热插拔、固件下载和模块自动加载等关键内核子系统的影响。Udev 是分析设备模型优点的最佳入口点，我们先从它开始讲解。

Udev

几年前，Linux 操作系统还很年轻， 管理设备节点的工作一点都不好玩。所有需要的结点（达到数千个）都不得不在/dev 目录下静态创 建。该问题实际起源于原始的 UNIX 系统。在 2.4 内 核中，引入了 devfs，它支持设备结点的动态创建。Devfs 提 供在了位于内存的文件系统中创建设备结点的能力，而命名结点的负担还是落在了设备驱动头上。但是，设备命名策略是可管理的，不应与内核混在一起。策略可位 于头文件、模块参数或用户空间中。而 Udev 将成功地设备管理的任务彻底推向了用户空间。

Udev 的工作依赖于：

1. 内核中的 sysfs 支持，sysfs 是 Linux 设备模型的一个重要组成部分。Sysfs 位于内存中，在启动时被挂载在了/sys 目 录（见/etc/fstab）。下一节我们会分析 sysfs， 你可以认为访问 sysfs 是理所当然的。

2. 一套用户空间守护程序和实用工具，如 udevd 和 udevinfo。

3. 用户自定义的规则，位于/etc/udev/rules.d/目录。你可以根据对应设备的特点设置规则。

为了理解 udev 的用法，我们先看一个例子。假定你有一个 USB DVD 驱 动器或一个 USB CD-RW 驱动器。根据你热插拔设备顺序的不同，一个被命名为/dev/sr0，另一个被命名为/dev/sr1。在 没有 udev 的情况下，你必须执行区分这些名字对应的设备。但是，有了 udev 以后，不管你以什么顺序热插拔它们，你都能分辨出二者,如 DVD 命名为/dev/usbdvd，CD-RW 命名为/dev/usbcdrw。

首先，从 sysfs 相应的文件中提取产 品信息。假定 Targus DVD 驱动器被分配的设备结点为/dev/sr0，Addonics CD-RW 驱动器为/dev/sr1， 使用 udevinfo 可收集设备信息：

```
bash> udevinfo -a -p /sys/block/sr0

...

looking at the device chain at

'/sys/devices/pci0000:00/0000:00:1d.7/usb1/1-4':

 BUS=»usb»

 ID=»1-4»

 SYSFS{bConfigurationValue}=»1»
```

```
...
SYSFS{idProduct}=»0701»
SYSFS{idVendor}=»05e3»
SYSFS{manufacturer}=»Genesyslogic»
SYSFS{maxchild}=»0»
SYSFS{product}=»USB Mass Storage Device»
...

bash> udevinfo -a -p /sys/block/sr1
...
looking at the device chain at
'/sys/devices/pci0000:00/0000:00:1d.7/usb1/1-3':
BUS=»usb»
ID=»1-3»
SYSFS{bConfigurationValue}=»2»
 ...
SYSFS{idProduct}=»0302»
SYSFS{idVendor}=»0dbf»
SYSFS{manufacturer}=»Addonics»
SYSFS{maxchild}=»0»
SYSFS{product}=»USB to IDE Cable»
...
```

接下来，使用搜集到的产品信息标识设备并 且添加 udev 命名规则。创建 /etc/udev/rules.d/40-cdvd.rules 文 件并添加如下规则信息：

```
BUS="usb", SYSFS{idProduct}="0701", SYSFS{idVendor}="05e3",
```

```
KERNEL="sr[0-9]*", NAME="%k", SYMLINK="usbdvd"



BUS="usb", SYSFS{idProduct}="0302", SYSFS{idVendor}="0dbf",

KERNEL="sr[0-9]*", NAME="%k", SYMLINK="usbcdrw"
```

第 1 条规则告诉 udev，一旦它发现一个 USB 设备的产品 ID 为 0x0701，厂商 ID 为 0x05e3，就增加一个以 sr 开始的名称，udev 将在/dev 创建一个同名的结点并为之创建一个名为 usbdvd 的 符号链接。类似地，第 2 条规则为 CD-RW 驱 动器创建一个名为 usbcdrw 的符号链接。

为了测试新创建规则的语法错误，可以对/sys/block/sr*运 行 udevtest。为了打开/var/log/messages 中 的相关提示信息，可以将/etc/udev/udev.conf 文件中的 udev_log 设置为“yes”。为了在运行过程中对/dev 目录应用新增规则，可以运行 udevstart 重 启 udev。此后，你的 DVD 驱动器在系统 中将始终为/dev/usbdvd，而你的 CD-RW 驱 动器将总是为/dev/usbcdrw。

你可以自主地通过在 shell 的脚本中 通过如下命令来挂载设备：

```
mount /dev/usbdvd /mnt/dvd
```

为设备结点（以及网络接口）进行恒定的命名并非 udev 的 唯一功能。实际上，udev 已经演变为 Linux 热 插拔管理器。Udev 也承担了按需自动加载模块和为设备下载微码的任务。在挖掘这些能力之前，让我 们首先对设备模型的内在机理有一个基本的认识。

Sysfs、Kobject和 设备类（Device Class）

Sysfs、Kobjects 和设备类（Device Classes）是设备模型的组成模块，但是它们羞于在公众面前露面，一直深藏于幕后。它们主要在总线和核心层的实现中使用，隐藏在为设备驱动提供服务的 API 中。

Sysfs 是内核中结构化的设备模型在用户空间的印证。它与 procfs 类型，二者都是位于内 存的文件系统，而且包含内核数据结构的信息。但是，procfs 是查看内核内部的一个通用视窗，而 sysfs 则特定地对应于设备模型。因而，Sysfs 并 非 procfs 的替代品。进程描述符、sysctl 参数等信息属于 procfs 而非 sysfs。很快，我们会发现 udev 的大多数功能都依赖于 sysfs。

Kobject 封装了一些公用的对象属性，如引用计数。它们通常被嵌在更大的数据结构中。 Kobject 的主要字段如下（定义于 include/linux/kobject.h 文件）：

1. kref 对象，用于引用计数管理。kref_init()接口用于初始化 kref 接口，kref_get()用户增加引用计数，而 kref_put()则 用于减少引用计数，当没有剩下的引用后，对象会被释放。例如，URB 结构体（见第 11 章《通用串行总线》）就包含一个 kref，用于跟踪对它的引用的数量 [2]。

[2]usb_alloc_urb() 接口调用 kref_init()，usb_submit_urb()调用 kref_get()，而 usb_free_urb()则调用 kref_put()。

2. kset 的指针，表征 kobject 归属的对象集（object set）。

3. kobj_type，用于描述对象的类型。

Kobject 与 sysfs 紧密关联。内核中的每个对象实例对有一个 sysfs 的代表。

设备类概念的引入是设备模型的另一个特点，在驱动中，我们更有可能用到此接口。类接口抽象了这 一理念，即每个设备都属于某一个总括性的分类。例如，USB 鼠标、PS/2 键盘和操纵杆对属于输入设备类，对在/sys/class/input/拥有入 口。

图 4.4 显示了一个连接了外部 USB 鼠标的笔记本电脑的 sysfs 结构。顶层的 bus、class 和 device 目录被展开，以便显示 sysfs 是怎样基于设备类型和物理连接提供 USB 鼠标的视图的。鼠标是一个输入类设备，但是在物理上是一 个 USB 设备，包含 2 个端点：1 个控制端点 ep00 和一个中断断点 ep81。上述 B 端口位于 BUS 2 上的 USB 主机控制器，而 USB 主机控制器自身则通过 PCI 总线被桥接给 CPU。如果到目前为止，你还是理不清的话，不必担心。在读完讲解输入设备驱动 的第 7 章、PCI 驱动的第 10 章和 USB 驱动的第 11 章后，请回过头来阅读本节。

图 4.4 USB鼠标的组织

```
[/sys]
    +[block]
    -[bus]—[usb]—[devices]—[usb2]—[2-2]—[2-2:1.0]-[usbendpoint:usbdev2.2-ep81]
    -[class]-[input]—[mouse2]—[device]—[bus]—[usbendpoint:usbdev2.2-ep81]
         -[usb_device]—[usbdev2.2]—[device]—[bus]
         -[usb_endpoint]—[usbdev2.2-ep00]—[device]
                    —[usbdev2.2-ep81]—[device]
    -[devices]—[pci0000:00]—[0000:00:1d:1]—[usb2]—[2-2]—[2-2:1.0]
    +[firmware]
    +[fs]
    +[kernel]
    +[module]
    +[power]
```

读者可以浏览/sys 并查看与其他设备关联 的入口（如你的网卡）以便对 sysfs 的层次结构组织有更好的理解。第 10 章的《Addressing and Identification》一节论证了 sysfs 怎样为笔记本电脑上 CardBus 以太网 Modem 卡的物理连接建立镜像的例子。

class 编程接口则建立在 kobject 和 sysfszhishang，因此它是深入理解设备模型中各种组件进行端到端交换的很好的着入点。我们以 RTC 驱动为例，RTC 驱动（drivers/char/rtc.c）是一个混杂设备（misc）驱动。在第 5 章分析字符设备驱动的时候，我们会对 misc 驱动进行讨论。

加载 RTC 驱动模块，查看/sys 和/dev 下载的结点：

```
bash> modprobe rtc

bash> ls -lR /sys/class/misc

drwr-xr-x 2 root root 0 Jan 15 01:23 rtc

/sys/class/misc/rtc:

total 0

-r--r--r-- 1 root root 4096 Jan 15 01:23 dev

--w------- 1 root root 4096 Jan 15 01:23 uevent

bash> ls -l /dev/rtc

crw-r--r-- 1 root root 10, 135 Jan 15 01:23 /dev/rtc
```

/sys/class/misc/rtc/dev 包含了分配给该设备的主次设备号（下一章讨 论），/sys/class/misc/rtc/uevent 用于冷插拔（下一节讨论），/dev/rtc 是应用程序访问 RTC 驱动的入口。

下面分析一下设备模型的代码流程。在初始化过程中，Misc 驱动利用了 misc_register()服务，抽取一些代码片段：

```
/* ... */

dev = MKDEV(MISC_MAJOR, misc->minor);



misc->class = class_device_create(misc_class, NULL, dev,

                          misc->dev,

                          "%s", misc->name);

if (IS_ERR(misc->class)) {

  err = PTR_ERR(misc->class);
```

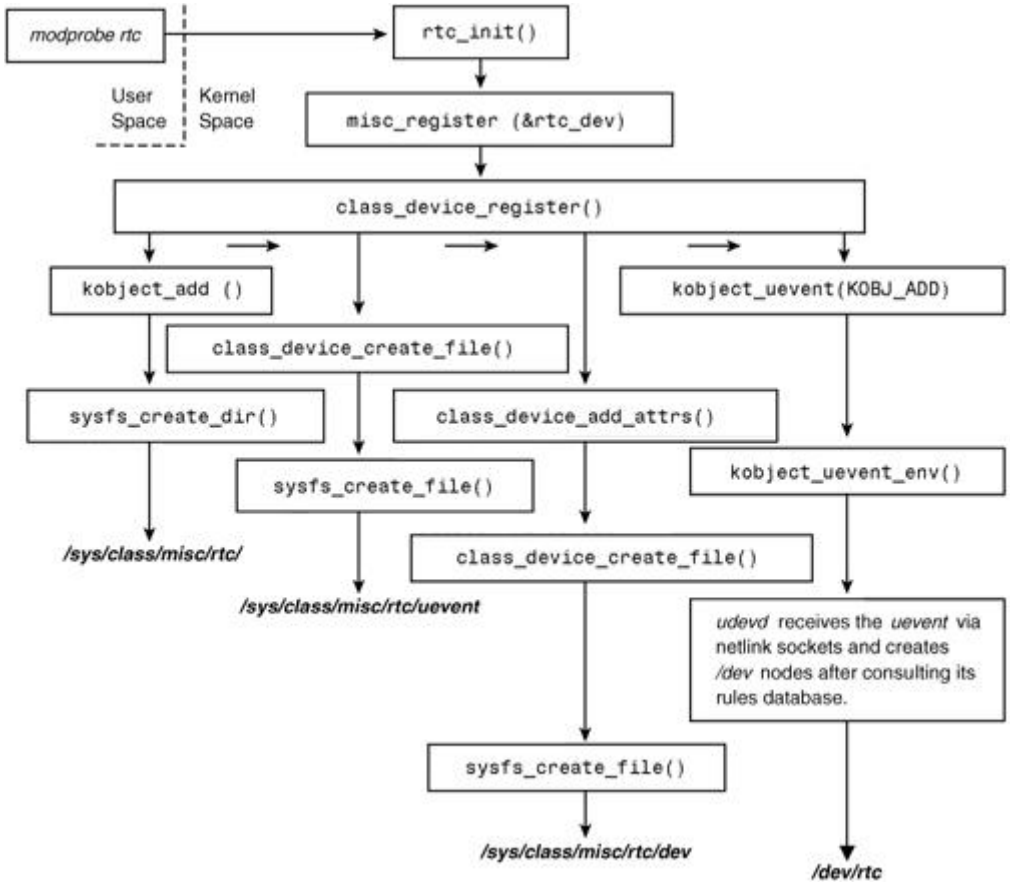
```
    goto out;

}

/* ... */
```

图 4.5 继续剥离了更多层以便深入到设备迷信的底层。它论证了 class、kobject、sysfs 以及 udev 之间的交互，这些交互会在/sys 和/dev 中产生相应的文件。

并口 LED 驱动（第 5 章《Talking to the Parallel Port》一节）和虚拟鼠标输入设备驱动（第 7 章《Device Example: Virtual Mouse》一节）可以作为创建 sysfs 中设备控制文件的例子。

图 4.5 设备模型相关组件的联系

<!--[if !vml]-->



<!--[endif]-->

总线—设备—驱动编程结构也是设备模型抽象的另一个部分。内核设备支持被清晰地结构化为总线、设备和驱动。这使得单独的驱动更容易实现 也更通用。例如，总线能用于寻找操作某一设备的驱动。

以内核的I2C子系统（第 8 章《The Inter-Integrated Circuit Protocol》）为例。I^2C层包含一个核心设施、总线适配器的设备驱动以及client设备驱动。I^2C 核心层使用bus_register()注册每个被侦测到的I^2C总线适配器。当一个I^2C client设备（例如，一个电可擦除的可编程只读存储EEPROM芯片）被探测到后，device_register()将表征它的存

在。最后，I2C EEPROM client驱动通过driver_register()注册自身。实际上，这些注册是由I²C核心提供的API 间接执行的。

bus_register()会在/sys/bus/增加一个相应的入口，而 device_register()会在/sys/devices/增 加相应的入口。bus_type、device、 device_driver 这 3 个结构体分别是与总线、设备和驱动对应的主要数据结构。include/linux/device.h 中包含了它们的定义。

热插拔和冷插拔

在运行过程中往系统中插入设备称为热插拔，而在系统启动前就连接设备则称为冷插拔。以前，内核 通过调用由/proc 文件系统注册的辅助程序来向用户空间通知热插拔 事件。而在当前的内核里，侦测到热插拔事件后，它们会通过 netlink 套接字向用户空间派生 uevent。Netlink 一种在内核空间和用户空间透过 套接字 API 进行通信的足够强大的机制。用户空间的 udevd（管理设备结点创建和移除的守护程序）会接收 uevent 并管理热插拔。

为了查看热插拔处理机制最新的进展，查看一下 2.6 内 核各个不同版本情况下 udev 的逐步变迁：

1. udev-039 和 2.6.9 内核，当内核侦测到一个热插 拔事件后，它激活/proc/sys/kernel/hotplug 中注册的用户空间辅助程序。用户空间辅助程序默认为/sbin/hotplug ，它接收热插拔设备的属性信息，在执行完/etc/hotplug/目录中的其它脚本之后，/sbin/hotplug 查看热 插拔配置目录（通常为/etc/hotplug.d/default/）并且运行相应的程序（如 /etc/hotplug.d/default/10-udev.hotplug）。

```
bash> ls -l /etc/hotplug.d/default/

...

lrwcrwxrwx 1 root root 14 May 11 2005 10-udev.hotplug -> /sbin/udevsend

...
```

当/sbin/udevsend 执行 后，它将热插拔设备的信息传递给 udevd。

2. udev-058 和 2.6.11 内核，情况发生了一些变 化。Udevsend 程序代替了/sbin/hotplug:

```
bash> cat /proc/sys/kernel/hotplug

/sbin/udevsend
```

3. 对于最新的 udev 和内核而言，udevd 承担了管理热插拔的全部 责任，不再依赖于 udevsend 。它现在通过 netlink 套 接字直接从内核提取热插拔事件。/proc/sys/kernel/hotplug 什么都不包含：

```
bash> cat /proc/sys/kernel/hotplug
```

```
bash>
```

Udev 也处理冷插拔。由于 udev 是用户空间的 一部分，仅仅在内核启动后才开始运行，需要一种特殊的机制针对冷插拔设备模拟热插拔事件。启动时，内核为所有设备在 sysfs 下 创建了一个名为 uevent 的文件，并将冷插拔事件记录于这些文件。当 udev 开始运行后，它读取/sys 下所有的 uevent 文件，并为每个冷插拔设备产生热插拔 uevent。

微 码下载

一些设备在开始运行前，必须下载微码。微码会在片上的微控制器中执行，过去，设备驱动通常将微 码存放于头文件的静态数组中。但是，这种途径并不安全，这是因为微码通常是设备制造商的具有产权的映像文件，它们不宜进入 GPL 的内核。另一个原因导致不宜将二者混在一起的原因是内核和固件发布的时间线不同。显而易见，更好的解决方 法是在用户空间维护微码，并在内核需要的时候将其下载进去。Sysfs 和 udev 为此提供了支持。

以一些笔记本电脑上的Intel PRO/Wireless 2100 无线mini PCI卡 为例，该卡建立在一个需要执行外部提供微码的微控制器之上。下面分析一下Linux驱动下载微码到 该卡的步骤。假定用户已经从http://ipw2100.sourceforge.net/firmware.php拿 到了微码映像（ipw2100-1.3.fw），并已将其存放到系统的/lib/firmware/目录，并且加载了驱动模块 ipw2100.ko：

1.初始化过程中，驱动调用如下语句：

```
request_firmware(..,"ipw2100-1.3.fw",..);
```

2.步骤 1 将向用户空间分发一个热插拔 uevent，并提供所请求微码映像的标识信息。

3.udevd 接收该 uevent，并调用/sbin/firmware_helper 进行响应。为此，它使用了 /etc/udev/rules.d/目 录下某一规则文件中类似的规则进行处理：

```
ACTION=="add", SUBSYSTEM=="firmware", RUN="/sbin/firmware_helper"
```

4./sbin/firmware_helper 在/lib/firmware/目录中 找到对应的微码映像 ipw2100-1.3.fw，并将该映像存储到/sys/class/0000:02:02.0/data。0000:02:02 是 本例中无线网卡的“总线:设备:功能”标识。

5.驱动接收微码并将其下载到设备中。下载完成后，它调用 release_firmware()释放相应的数据结构。

6.驱动完成剩余的初始化工作，无线网卡进入工作状态。

模 块自动加载

按需自动加载内核模块是 Linux 支持 的一种实用功能。为了理解内核怎样发起“模块缺失”事件以及 udev 怎么处理它，下面以向笔记本电 脑的 PC 卡插槽插入 Xircom CardBus 以 太网适配器为例进行说明：

1. 在编译的时候，驱动包含了一个它所支持的设备的标识列表。查看 Xircom CardBus 网卡的驱动（drivers/net/tulip/xircom_cb.c）， 可以发现如下的代码片段：

```
static struct pci_device_id xircom_pci_table[] = {

   {0x115D, 0x0003, PCI_ANY_ID, PCI_ANY_ID,},

   {0,},

};



/* Mark the device table */

MODULE_DEVICE_TABLE(pci, xircom_pci_table);
```

这表明该驱动支持 PCI 制造商 ID 为 0x115D 和设备 ID（详见第 10 章）为 0x0003 的任何卡。当安装该驱动模块的时候，depmod 将 分析模块映像并提取其中的设备表以获得 ID，并将如下的入口添加到/lib/modules/kernel-version/modules.alias：

```
alias pci:v0000115Dd00000003sv*sd*bc*sc*i* xircom_cb
```

其中，v 代表制造商 ID，d 代表设备 ID，sv 代表子制造商 ID，*为通配符。

2. 当用户将该 Xircom 卡热插入 CardBus 插槽后，内核将产生一个 uevent， 在其中包含新插入设备的 ID。可以使用 udevmonitor 查 看产生的 uevent：

```
bash> udevmonitor --env

  …

  MODALIAS=pci:v0000115Dd00000003sv0000115Dsd00001181bc02sc00i00

  …
```

3. Udevd 通过 netlink 套接字接收 uevent，并用内核传给它的上述别名调用 modprobe：

```
modprobe pci:v0000115Dd00000003sv0000115Dsd00001181bc02sc00i00
```

4. modprobe在第 1 步创建的/lib/modules/kernel-version/modules.alias文件中寻找匹配的入口，接着插入xircom_cb：

```
bash> lsmod

Module     Size   Used by

xircom_cb   10433  0

…
```

该卡现在就可以用于网上冲浪了。

在读完第 10 章以后，读者可以回过头来 阅读本节。

嵌 入式设备上的udev

有一种流派反对在嵌入式设备上使用 udev，而支持静态创建设备结点，原因如下：

1. udev 在每次重启时创建/dev 结点，而静态结 点只是在软件安装的时候才创建。如果嵌入式设备使用 Flash 存储器，存放/dev 结点的 Flash 页面在每次重启的时候都需要进行擦除——写操作，这会减少 Flash 的寿命。(第 17 章《存储技术设备》将对 FlashFlash 存储 器进行讨论。)当然，为了避免此问题，/dev 可 以挂载在基于 RAM 的文件系统中。

2. udev 增加了启动时间。

3. udev 提供了动态创建/dev 结点和自动加载模 块的能力，这对某些设备造成了一定程度的不确定性，一些特定目的的嵌入式设备（尤其是不通过可热插拔的总线与外界交互信息的设备）需要避免此一不确定性。 基于此，静态创建结点并在运行时加载任何需要的模块意味着可以对系统进行更多的控制，而且也更易于测试。

内 存屏障

为了优化执行速度，许多处理器和编译器都会重新排序指令。重新排序后，新的指令流和原始指令流在语义上等同。但是，如果正在写的是 I/O 设备上被映射的寄存器，指令重排序会产生无法预料的副作用。为了阻止处理器重新排序指令，可以在代码中添 加一个屏障。wmb()函数可以阻止写操作的移动，rmb()则 禁止了读操作的移动，mb()会设置读和写的屏障。

除了前文提到的 CPU 与硬件的交互以 外，内存屏障也关系到 SMP 系统中 CPU 与 CPU 之间的交互。如果 CPU 的数据高速缓冲区正在使 用写回模式（在该模式下，只有当数据确实需要被写进内存的时候，它才会被从高速缓冲区拷贝到内存），某些情况下，程序可能需要停止指令流以等待高速缓冲区 写回内存的操作全部结束。这一点关系重大，到遇到需要获取和释放锁的指令序列时，使用屏障可以在保证 CPU 获 得一致的视图。

在讨论完第 10 章的 PCI 驱动和第 17 章的 Flash 映射驱动后，读者可以重新回顾内存屏障的知识。与此同时，Documentation/memory-barriers.txt 包 含了各种内存屏障的解释。

电 源管理

电源管理对于使用电池的设备（如笔记本电脑、手持设备）而言非常关键。Linux 驱动需要意识到电源状态，并对待机、睡眠和电池电压低等事件作出反应，并在不同的状态间进行转换。在 切换到低功耗模式的时候，驱动能够利用底层硬件支持的节能功能。例如，存储器驱动减速运行，视频驱动显示空白。

设备驱动中编写具“电 源意识”的代码只是整个电源管理框架的一小部分。电源管理功能也牵涉到用户空间守护进程、实用工具、配置文件和启动固件。两种流行的电源管理机制是：APM（在附录 B《Linux 和 BIOS》的《保护模式调用》一节中讨论） 和高级配置和电源接口（ACPI）。APM 开 始过时了，ACPI 已经成为 Linux 系统 中事实上的电源管理策略。第 20 章《更多的设备和驱动》将对其进行讨论。

查看源代码

核心的中断处理代码是通用的，位于 kernel/irq/目 录，而体系结构相关的部分在位于 arch/your-arch/kernel/irq.c。该文件 中定义的 do_IRQ()函数是开始分析内核中断处理机制的良好起点。

内核软中断和 tasklet 的实现位于 kernel/softirq.c，该文件也包含了提供更细粒度对软中断和 tasklet 进行控制的函数。查看 include/linux/interrupt.h 可 以获得软中断向量的枚举以及实现自己的中断处理函数的原型。drivers/net/lib8390.c 可 以作为编写中断处理函数和底半部的实例，读者可以追索其从中断处理到网络协议栈的过程。

Kobject的实现和相关的编程接口位于lib/kobject.c和include/linux/kobject.h文件。drivers/base/sys.c包 含了sysfs的实现。drivers/base/class.c包 含了设备类API。lib/kobject_uevent.c文 件提供了通过netlink套接字分发热插拔uevent的 代码。从 如 下 网 站 可 以 下 载 udev 的 源 代 码 和 相 关 文 档 ：www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html。

为了更好地理解 APM 在基于 x86 体系结构 Linux 上的实现，可以查看内核中的 arch/x86/kernel/apm_32.c 、 include/linux/apm_bios.h 和 include/asm-x86/mach-default/apm.h 文件。如果读者对无 BIOS 体系结构（如 ARM）的 APM 实现感兴趣的话，可以查看 include/linux/apm-emulation.h 和 它的使用。内核的 ACPI 实现位于 drivers/acpi/。

表 4.2 给出了本章所涉及的主要数据结 构以及其在源代码树中位置的总结。表 4.3 列出了本 章中主要的内核编程接口以及其定义的位置。

表 4.2 数据结构总结

数据结构	位置	描述
tasklet_struct	include/linux/interrupt.h	管理 tasklet，实现底半部的一种方法
kobject	include/linux/kobject.h	封装一个内核对象的公共属性
kset	include/linux/kobject.h	kobject 所属于 的对象集
kobj_type	include/linux/kobject.h	描述一个 kobject 的对象类型
class	include/linux/device.h	对驱动属于某泛类的理念 的抽象
bus	include/linux/device.h	构建 Linux 设备模型支柱的结构体
device		
device_driver		

表 4.3 内核编程接口总结

内核接口	位置	描述
request_irq()	kernel/irq/manage.c	请求一个 IRQ， 并为其分配一个中断处理函数
free_irq()	kernel/irq/manage.c	释放一个 IRQ

内核接口	位置	描述
disable_irq()	kernel/irq/manage.c	禁 止与某 IRQ 关联的中断
disable_irq_nosync()	kernel/irq/manage.c	禁止与某 IRQ 关 联的中断，并且不等待目前的中断处理实例返回
enable_irq()	kernel/irq/manage.c	重新使能已经被 disable_irq()或 disable_irq_nosync()禁止的中断
open_softirq()	kernel/softirq.c	打 开一个软中断
raise_softirq()	kernel/softirq.c	标识软中断需要被执行
tasklet_init()	kernel/softirq.c	动态地初始化一个 tasklet
tasklet_schedule()	include/linux/interrupt.hkernel/softirq.c	标 识 tasklet 需要被执行
tasklet_enable()	include/linux/interrupt.h	使 能一个 tasklet
tasklet_disable()	include/linux/interrupt.h	禁 止一个 tasklet
tasklet_disable_nosync()	include/linux/interrupt.h	禁止一个 tasklet，并且不等待其运行的实例完成
class_device_register()	drivers/base/class.c	Linux 设备模型中的一系列函数：创建/破坏类、设备类以及关联的 kobject 和 sysfs 文件
kobject_add()	lib/kobject.c	
sysfs_create_dir()	lib/kobject_uevent.c	
class_device_create()	fs/sysfs/dir.c	
class_device_destroy()	fs/sysfs/file.c	
class_create()		
class_destroy()		
class_device_create_file()		
sysfs_create_file()		
class_device_add_attrs()		
kobject_uevent()		

至此，设备驱动的一些重要概念就介绍完成了。在开发具体设备驱动的时候，读者可以返回本章进行 深入挖掘。