LINUX

业界权威机构和专家强力推荐

多年培训、研发经验的总结

设备驱动开发详解

华清远见嵌入式培训中心 宋宝华 编著

Linux

附送代码光盘

人民邮电出版社
POSTS & TELECOM PRESS

第 1 章　设备驱动概述

本章将带您走进 Linux 设备驱动的精彩世界。

1.1 节讲解了设备驱动的概念和作用。

1.2 节和 1.3 节分别讲述无操作系统和有操作系统情况下设备驱动的设计方法，通过分析讲解设备驱动与硬件和操作系统的关系。

1.4 节对 Linux 操作系统的设备驱动进行了概要性的介绍，讲解设备驱动与系统软硬件的关系，分析了 Linux 设备驱动的重点难点和学习方法。

本章的最后给出了一个设备驱动的“Hello World”实例，即最简单的 LED 驱动在无操作系统情况下和 Linux 操作系统下的实现。

华清远见

1.1 设备驱动的作用

任何一个计算机系统的运行都是系统中软硬件协作的结果，没有硬件的软件是空中楼阁，而没有软件的硬件则只是一堆废铁。硬件是底层基础，是所有软件得以运行的平台，代码最终会落实为硬件上的组合逻辑与时序逻辑；软件则实现了具体应用，它按照各种不同的业务需求而设计，满足了用户的需求。硬件较固定，软件则很灵活，可以适应各种复杂多变的应用。可以说，计算机系统的软硬件互相成就了对方。

但是，软硬件之间同样存在着悖论，那就是软件和硬件不应该互相渗透到对方的领地。为了尽可能快速地完成设计，应用软件工程师不想也不必关心硬件，而硬件工程师也难有足够的闲暇和能力来顾及软件。例如，应用软件工程师在调用套接字发送和接收数据包的时候，他不必关心网卡上的中断、寄存器、存储空间、I/O 端口、片选以及其他任何硬件词汇；在使用 printf()函数输出信息的时候，他不用知道底层究竟是怎样把相应的信息输出到屏幕或串口。

也就是说，应用软件工程师需要看到一个没有硬件的纯粹的软件世界，硬件必须被透明地呈现给他们。谁来实现硬件对应用软件工程师的隐形？这个艰巨的任务就落在了驱动工程师的头上。

对设备驱动最通俗的解释就是“驱使硬件设备行动”。设备驱动与底层硬件直接打交道，按照硬件设备的具体工作方式读写设备寄存器，完成设备的轮询、中断处理、DMA 通信，进行物理内存向虚拟内存的映射，最终使通信设备能够收发数据，使显示设备能够显示文字和画面，使存储设备能够记录文件和数据。

由此可见，设备驱动充当了硬件和应用软件之间的纽带，它使得应用软件只需要调用系统软件的应用编程接口（API）就可让硬件去完成要求的工作。在系统中没有操作系统的情况下，工程师可以根据硬件设备的特点自行定义接口，如对串口定义 SerialSend()、SerialRecv()；对 LED 定义 LightOn()、LightOff()；以及对 Flash 定义 FlashWrite()、FlashRead()等。而在有操作系统的情况下，设备驱动的架构则由相应的操作系统定义，驱动工程师必须按照相应的架构设计设备驱动，这样，设备驱动才能良好地整合到操作系统的内核中。

驱动程序沟通着硬件和应用软件，而驱动工程师则沟通着硬件工程师和应用软件工程师。随着通信、电子行业的迅速发展，全世界每天都会有大量的新芯片被生产，大量的新电路板被设计，因此，也会有大量设备驱动需要开发。这些设备驱动，或运行在简单的单任务环境中，或运行在 VxWorks、Linux、Windows 等多任务操作系统环境中，发挥着不可替代的作用。

1.2 无操作系统时的设备驱动

并不是任何一个计算机系统都一定要运行操作系统，在许多情况下操作系统是不必要的。对于功能比较单一、控制并不复杂的系统，如公交车刷卡机、电冰箱、微波炉、简单的手机和小灵通等，并不需要多任务调度、文件系统、内存管理等复杂功能，用单任务架构完全可以很好地支持它们的工作。一个无限循环中夹杂对设备中断的检测或者对设备的轮询是这种系统中软件的典型架构，如代码清单 1.1 所示。

代码清单 1.1　单任务软件典型架构

```
int main(int argc, char* argv[])
{
  while (1)
  {
    if (serialInt == 1)
    /*有串口中断*/
    {
      ProcessSerialInt(); /*处理串口中断*/
      serialInt = 0;          /*中断标志变量清零*/
    }
    if (keyInt == 1)
    /*有按键中断*/
    {
      ProcessKeyInt();        /*处理按键中断*/
      keyInt = 0;         /*中断标志变量清零*/
    }
    status = CheckXXX();
    switch (status)
    {
      ...
    }
    ...
  }
}
```

在这样的系统中，虽然不存在操作系统，但是设备驱动是必须存在的。一般情况下，对每一种设备驱动都会定义为一个软件模块，包含.h 文件和.c 文件，前者定义该设备驱动的数据结构并声明外部函数，后者进行设备驱动的具体实现。代码清单 1.2 定义了一个串口的驱动。

代码清单 1.2　无操作系统情况下串口的驱动

```
/**********************
 *serial.h 文件
 **********************/
extern void SerialInit(void);
extern void SerialSend(const char buf*,int count);
```

```
extern void SerialRecv(char buf*,int count);

/**********************
 *serial.c 文件
 **********************/
/*初始化串口*/
void SerialInit(void)
{
 ...
}
/*串口发送*/
void SerialSend(const char buf*,int count)
{
 ...
}
/*串口接收*/
void SerialRecv(char buf*,int count)
{
 ...
}
/*串口中断处理函数*/
void SerialIsr(void)
{
 ...
 serialInt = 1;
}
```

其他模块需要使用这个设备的时候，只需要包含设备驱动的头文件 serial.h，然后调用其中的外部接口函数即可。如我们要从串口上发送字符串“Hello World”，使用函数 SerialSend(" Hello World ",11)即可。

由此可见，在没有操作系统的情况下，设备驱动的接口被直接提交给了应用软件工程师，应用软件没有跨越任何层次就直接访问了设备驱动的接口。设备驱动包含的接口函数也与硬件的功能直接吻合，没有任何附加功能。图 1.1 所示为无操作系统情况下硬件、设备驱动与应用软件的关系。

图 1.1　无操作系统时硬件、设备驱和应用软件的关系

有的工程师把单任务系统设计成了如图1.2所示的结构，即设备驱动和具体的应用软件模块处于同一层次，这显然是不合理的，不符合软件设计中高内聚低耦合的要求。

另一种不合理的设计是直接在应用中操作硬件的寄存器，而不单独设计驱动模块，如图 1.3 所示。这种设计意味着系统中不存在或未能充分利用可被重用的驱动代码。

图 1.2　设备驱动与应用高耦合的不合理设计

图 1.3　应用直接访问硬件的不合理设计

1.3 有操作系统时的设备驱动

1.2 节中的设备驱动直接运行在硬件之上，不与任何操作系统关联。当系统中包含操作系统后，设备驱动会变得怎样？

首先，无操作系统时设备驱动的硬件操作工作仍然是必不可少的，没有这一部分，设备驱动不可能与硬件打交道。

其次，我们还需要将设备驱动融入内核。为了实现这种融合，必须在所有的设备驱动中设计面向操作系统内核的接口，这样的接口由操作系统规定，对一类设备而言结构一致，独立于具体的设备。

由此可见，当系统中存在操作系统的时候，设备驱动变成了连接硬件和内核的桥梁。如图 1.4 所示，操作系统的存在势必要求设备驱动附加更多的代码和功能，把单一的“驱使硬件设备行动”变成了操作系统内与硬件交互的模块，它对外呈现为操作系统的 API，不再给应用软件工程师直接提供接口。

图 1.4　硬件、设备驱动、操作系统和应用程序的关系

有了操作系统之后，设备驱动反而变得复杂，那要操作系统干什么？

首先，一个复杂的软件系统需要处理多个并发的任务，没有操作系统，想完成多任务并发是很困难的。

其次，操作系统给我们提供内存管理机制。一个典型的例子是，对于多数含 MMU 的处理器而言，Windows、Linux 等操作系统可以让每个进程都独立地访问 4GB 的内存空间。

上述优点似乎并没有体现在设备驱动身上，操作系统的存在给设备驱动究竟带来了什么好处呢？

简而言之，操作系统通过给设备驱动制造麻烦来达到给上层应用提供便利的目的。如果设备驱动都按照操作系统给出的独立于设备的接口而设计，应用程序将可使用统一的系统调用接口来访问各种设备。对于类 UNIX 的 VxWorks、Linux 等操作系统而言，应用程序通过 write()、read()等函数读写文件就可以访问各种字符设备和块设备，而不用管设备的具体类型和工作方式，是非常方便的。

1.4 Linux 设备驱动

1.4.1 设备的分类及特点

计算机系统的硬件主要由 CPU、存储器和外设组成。随着 IC 制造工艺的发展，目前，芯片的集成度越来越高，往往在 CPU 内部就集成了存储器和外设适配器。ARM、PowerPC、MIPS 等处理器都集成了 UART、I^2C 控制器、USB 控制器、SDRAM 控制器等，有的处理器还集成了片内 RAM 和 Flash。

驱动针对的对象是存储器和外设（包括 CPU 内部集成的存储器和外设），而不是针对 CPU 核。Linux 将存储器和外设分为 3 个基础大类：

- 字符设备；
- 块设备；
- 网络设备。

字符设备指那些必须以串行顺序依次进行访问的设备，如触摸屏、磁带驱动器、鼠标等。块设备可以用任意顺序进行访问，以块为单位进行操作，如硬盘、软驱等。字符设备不经过系统的快速缓冲，而块设备经过系统的快速缓冲。但是，字符设备和块设备并没有明显的界限，如 Flash 设备符合块设备的特点，但是我们仍然可以把它作为一个字符设备来访问。

字符设备和块设备的驱动设计呈现出很大的差异，但是对于用户而言，他们都使用文件系统的操作接口 open()、close()、read()、write()等函数进行访问。

在 Linux 系统中，网络设备面向数据包的接收和发送而设计，它并不对应于文件系统的节点。内核与网络设备的通信和内核与字符设备、块设备的通信方式完全不同。

另外，TTY 驱动、I^2C 驱动、USB 驱动、PCI 驱动、LCD 驱动等本身大体可归纳入 3 个基础大类，但是对于这些复杂的设备，Linux 系统还定义了独特的驱动体系结构。

1.4.2 Linux 设备驱动与整个软硬件系统的关系

如图 1.5 所示，除网络设备外，字符设备与块设备都被映射到 Linux 文件系统的文件和目录，通过文件系统的系统调用接口 open()、write()、read()、close()等函数即可访问字符设备和块设备。所有的字符设备和块设备都被统一地呈现给用户。块设备比字符设备复杂，在它上面会首先建立一个磁盘/Flash 文件系统，如 FAT、Ext3、YAFFS、JFFS 等。FAT、Ext3、YAFFS、JFFS 规范了文件和目录在存储介质上的组织。

华清远见

应用程序可以使用 Linux 的系统调用接口编程，也可以使用 C 库函数，出于代码可移植性的考虑，后者更值得推荐。C 库函数本身也通过系统调用接口而实现，如 C 库函数中的 fopen()、fwrite()、fread()、fclose()分别会调用操作系统 API 的 open()、write()、read()、close()函数。

图 1.5　Linux 设备驱动与整个软硬件系统的关系

1.4.3　编写 Linux 设备驱动的技术基础

Linux 设备驱动的学习是一项浩大的工程，读者需要首先掌握以下基础。

- 编写 Linux 设备驱动要求工程师具有良好的硬件基础，懂得 SRAM、Flash、SDRAM、磁盘的读写方式，UART、I^2C、USB 等设备的接口，轮询、中断、DMA 的原理，PCI 总线的工作方式以及 CPU 的内存管理单元（MMU）等。
- 编写 Linux 设备驱动要求工程师具有良好的 C 语言基础，能灵活地运用 C 语言的结构体、指针、函数指针及内存动态申请和释放等。
- 编写 Linux 设备驱动要求工程师具有一定的 Linux 内核基础，虽然并不要求工程师对内核各个部分有深入的研究，但至少要了解设备驱动与内核的接口，尤其是对于块设备、网络设备、Flash 设备、串口设备等复杂设备。
- 编写 Linux 设备驱动要求工程师具有良好的多任务并发控制和同步的基础，因为在设备驱动中会大量使用自旋锁、互斥、信号量、等待队列等并发与同步机制。

本书对以上内容都进行了详细的讲解，以使读者快速掌握编写 Linux 设备驱动的基础。

1.4.4 Linux 设备驱动的学习方法

动手实践永远是学习任何软件开发的最好方法，学习 Linux 设备驱动也不例外。因此，您最好有一块可以实际练手的电路板来构造嵌入式开发环境。如果您暂时没有，则可以用 VmWare 搭建两台虚拟机，两台虚拟机上都运行 Linux 操作系统，一台作为开发主机，另一台作为目标机。

目前的 PC 上往往只有 1 个串口，但是调试要求主机和目标机之间使用串口通信，这要求 2 个串口。在虚拟机中我们可以用管道虚拟串口。在主机端设置“终端是客户机”，并选择“其他终端是一个虚拟机”。在目标机端设置“终端是服务器”，同样选择“其他终端是一个虚拟机”，但是要启用轮询。主机和目标机的串口设置分别如图 1.6 和图 1.7 所示。

图 1.6　VmWare 中主机串口设置

图 1.7　VmWare 中目标机串口设置

另外，为了使 VmWare 中安装的多个 Linux 系统之间以及 Linux 与 Windows 系统之间的网络互通，可以设置 VmWare 中的 Linux 系统使用 VMnet1（仅主机），如图 1.8 所示。打开 VMnet1 虚拟网卡的连接共享，将 Linux 系统的网关设置为 VMnet1 的 IP 地址“192.168.0.1”，Linux 系统将通过这个地址连接外网。

图 1.8　VmWare 中虚拟机网卡的设置

为了使 VmWare 中的 Linux 系统具有较高的显示分辨率，并能与主机上的 Windows 之间通过 hgfs 文件系统共享文件，最好在 Linux 中安装 VmWare tools。

源代码是学习 Linux 驱动的最权威资料，阅读 Linux 源代码的最佳工具是 Source Insight，在其中建立一个工程，并将 Linux 内核的所有源代码加入该工程，同步这个工程之后，我们将可以非常方便地在代码之间进行关联阅读，如图 1.9 所示。

图 1.9　在 Source Insight 中阅读 Linux 源代码

网站 http://lxr.linux.no/ident 提供了 1.0.9 、1.2.13 、2.0.40、2.2.26、2.4.18 、2.4.20、2.4.28、2.6.10、2.6.11、2.6.17.13 、2.6.18、2.6.20.1 版本内核和 i386、Alpha、ARM、IA-64、m68k、MIPS、MIPS64、PowerPC、IBMS/390、SH、SPARC、SPARC64、x86-64 体系结构下 Linux 源代码的“Cross-Referencing”，在其中输入 Linux 内核中的函数、数据结构或变量的名称就可以直接得到以超链接形式给出的定义和引用它的所有位置。如搜索 2.6.20.1 内核、i386 体系结构下的 sys_write()函数，得到如下信息：

```
sys_write
Defined as a function in:

    * fs/read_write.c, line 374

Defined as a function prototype in:

    * include/linux/syscalls.h, line 380

Referenced (in 11 files total) in:

    * arch/mips/kernel/irixioctl.c, line 236
    * arch/mips/kernel/sysirix.c:
        o line 1046
        o line 1531
    * arch/s390/kernel/compat_linux.c, line 955
    * arch/sparc/kernel/sys_sunos.c, line 1093
    * arch/sparc64/kernel/sys_sunos32.c, line 1239
    * fs/read_write.c, line 374
    * include/asm-x86_64/unistd.h, line 18
```

```
    * include/asm-xtensa/unistd.h, line 48
    * include/linux/syscalls.h, line 380
    * init/do_mounts_rd.c:
        o line 239
        o line 373
    * init/initramfs.c:
        o line 305
        o line 311
        o line 552
```

除此之外，阅读经典书籍和参与 Linux 社区的讨论也是非常好的学习方法。Linux 内核源代码中包含了一个 Documentation 目录，其中包含了一批内核设计的文档，全部是文本文件。很遗憾，这些文档的组织不太好，内容也不够细致。

学习 Linux 设备驱动的一个注意事项是要避免管中窥豹、只见树木不见森林，因为各类 Linux 设备驱动都从属于一个 Linux 设备驱动的架构，单纯而片面地学习几个函数、几个数据结构是不可能理清设备驱动中各组成部分之间的关系的。因此，Linux 驱动的分析方法是点面结合，将对函数和数据结构的理解放在整体架构的背景之中，而这正是本书各章节讲解驱动的方法。

1.5 设备驱动的 Hello World：LED 驱动

1.5.1 无操作系统时的 LED 驱动

在嵌入式系统的设计中，LED 一般直接由 CPU 的 GPIO（通用可编程 I/O 口）控制。GPIO 一般由两组寄存器控制，即一组控制寄存器和一组数据寄存器。控制寄存器可设置 GPIO 口的工作方式为输入或输出。当引脚被设置为输出时，向数据寄存器的对应位写入 1 和 0 会分别在引脚上产生高电平和低电平；当引脚设置为输入时，读取数据寄存器的对应位可获得引脚上相应的电平信号。

在本例子中，我们屏蔽具体 CPU 的差异，假设在 GPIO_REG_CTRL 物理地址处的控制寄存器处的第 n 位写入 1 可设置 GPIO 为输出，在 GPIO_REG_DATA 物理地址处的数据寄存器的第 n 位写入 1 或 0 可在引脚上产生高或低电平，则在无操作系统的情况下，设备驱动代码如清单 1.3 所示。

代码清单 1.3　无操作系统时的 LED 驱动

```
1  #define reg_gpio_ctrl *(volatile int *)(ToVirtual(GPIO_REG_CTRL))
2  #define reg_gpio_data *(volatile int *)(ToVirtual(GPIO_REG_DATA))
3  /*初始化 LED*/
4  void LightInit(void)
5  {
6    reg_gpio_ctrl |= (1 << n); /*设置 GPIO 为输出*/
```

```
7  }
8
9  /*点亮 LED*/
10 void LightOn(void)
11 {
12   reg_gpio_data |= (1 << n); /*在 GPIO 上输出高电平*/
13 }
14
15 /*熄灭 LED*/
16 void LightOff(void)
17 {
18   reg_gpio_data &= ~(1 << n); /*在 GPIO 上输出低电平*/
19 }
```

上述程序中的 LightInit()、LightOn()、LightOff()等函数都将作为 LED 驱动提供给应用程序的外部接口函数。程序中 ToVirtual()等函数的作用是当系统启动了硬件 MMU 之后，根据物理地址和虚拟地址的映射关系，将寄存器的物理地址转化为虚拟地址。

1.5.2 Linux 系统下的 LED 驱动

在 Linux 操作系统下编写 LED 设备的驱动时，操作硬件的 LightInit()、LightOn()、LightOff()这些函数仍然需要，但是，需要遵循 Linux 编程的命名习惯，重新将其命名为 light_init()、light_on()、light_off()。这些函数将被 LED 驱动中独立于设备的针对内核的接口进行调用，代码清单 1.4 给出了 Linux 系统下的 LED 驱动，现在读者并不需要能读懂这些代码。

代码清单 1.4　Linux 系统下的 LED 驱动

```
1   #include .../*包含内核中的多个头文件*/
2
3   /*设备结构体*/
4   struct light_dev
5   {
6     struct cdev cdev; /*字符设备 cdev 结构体*/
7     unsigned char value; /*LED 亮时为 1，熄灭时为 0，用户可读写此值*/
8   };
9
10  struct light_dev *light_devp;
11  int light_major = LIGHT_MAJOR;
12
13  MODULE_AUTHOR("Song Baohua");
14  MODULE_LICENSE("Dual BSD/GPL");
15
16  /*打开和关闭函数*/
```

```
int light_open(struct inode *inode, struct file *filp)
{
  struct light_dev *dev;
  /* 获得设备结构体指针 */
  dev = container_of(inode->i_cdev, struct light_dev, cdev);
  /* 让设备结构体作为设备的私有信息 */
  filp->private_data = dev;
  return 0;
}
int light_release(struct inode *inode, struct file *filp)
{
  return 0;
}

/*写设备:可以不需要 */
ssize_t light_read(struct file *filp, char _ _user *buf, size_t count,
  loff_t*f_pos)
{
  struct light_dev *dev = filp->private_data; /*获得设备结构体 */

  if (copy_to_user(buf, &(dev->value), 1))
  {
    return  - EFAULT;
  }
  return 1;
}

ssize_t light_write(struct file *filp, const char _ _user *buf, size_t count,
  loff_t *f_pos)
{
  struct light_dev *dev = filp->private_data;

  if (copy_from_user(&(dev->value), buf, 1))
  {
    return  - EFAULT;
  }
  /*根据写入的值点亮和熄灭 LED*/
  if (dev->value == 1)
```

```
    light_on();
  else
    light_off();

  return 1;
}

/* ioctl 函数 */
int light_ioctl(struct inode *inode, struct file *filp, unsigned int cmd,
  unsigned long arg)
{
  struct light_dev *dev = filp->private_data;

  switch (cmd)
  {
    case LIGHT_ON:
      dev->value = 1;
      light_on();
      break;
    case LIGHT_OFF:
      dev->value = 0;
      light_off();
      break;
    default:
      /* 不能支持的命令 */
      return  - ENOTTY;
  }

  return 0;
}

struct file_operations light_fops =
{
  .owner = THIS_MODULE,
  .read = light_read,
  .write = light_write,
  .ioctl = light_ioctl,
  .open = light_open,
  .release = light_release,
```

```
};

/*设置字符设备 cdev 结构体*/
static void light_setup_cdev(struct light_dev *dev, int index)
{
  int err, devno = MKDEV(light_major, index);

  cdev_init(&dev->cdev, &light_fops);
  dev->cdev.owner = THIS_MODULE;
  dev->cdev.ops = &light_fops;
  err = cdev_add(&dev->cdev, devno, 1);
  if (err)
    printk(KERN_NOTICE "Error %d adding LED%d", err, index);
}

/*模块加载函数*/
int light_init(void)
{
  int result;
  dev_t dev = MKDEV(light_major, 0);

  /* 申请字符设备号*/
  if (light_major)
    result = register_chrdev_region(dev, 1, "LED");
  else
  {
    result = alloc_chrdev_region(&dev, 0, 1, "LED");
    light_major = MAJOR(dev);
  }
  if (result < 0)
    return result;

  /* 分配设备结构体的内存 */
  light_devp = kmalloc(sizeof(struct light_dev), GFP_KERNEL);
  if (!light_devp)   /*分配失败*/
  {
    result =  - ENOMEM;
    goto fail_malloc;
  }
  memset(light_devp, 0, sizeof(struct light_dev));
```

```
  light_setup_cdev(light_devp, 0);
  light_init();
  return 0;

  fail_malloc: unregister_chrdev_region(dev, light_devp);
  return result;
}

/*模块卸载函数*/
void light_cleanup(void)
{
  cdev_del(&light_devp->cdev); /*删除字符设备结构体*/
  kfree(light_devp); /*释放在 light_init 中分配的内存*/
  unregister_chrdev_region(MKDEV(light_major, 0), 1); /*删除字符设备*/
}

module_init(light_init);
module_exit(light_cleanup);
```

代码清单 1.4 的行数与代码清单 1.3 相比多了很多，除了代码清单 1.3 中的硬件操作函数仍然需要外，代码清单 1.4 中还包含了大量读者陌生的元素，如结构体 file_operations、cdev，Linux 内核模块声明用的 MODULE_AUTHOR、MODULE_LICENSE、module_init、module_exit，以及用于字符设备注册、分配和注销用的函数 register_chrdev_region()、alloc_chrdev_region()、unregister_chrdev_region() 等。此外设驱动中也增加了 light_init ()、light_cleanup ()、light_read()、light_write() 等这样的函数。

此时，我们只需要有一个感性认识，那就是，上述元素都是 Linux 驱动与内核的接口。Linux 对各类设备的驱动都定义了类似的数据结构和函数。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm

嵌入式学院 华清远见旗下品牌

第 2 章　驱动设计的硬件基础

本章讲解底层驱动工程师必备的硬件基础，给出了嵌入式系统硬件原理及分析方法的全景视图。

2.1 节讲解微控制器、微处理器、数字信号处理器以及应用于特定领域的处理器各自的特点。

2.2 节对嵌入式系统中所使用的各类存储器与 CPU 的接口、应用领域及特点进行了详细讲解。

2.3 节讲解常见的外设接口与总线的工作方式，包括串口、I^2C、USB、以太网接口、ISA、PCI 和 cPCI 等。

嵌入式系统硬件电路中经常会使用 CPLD 和 FPGA，2.4 节讲解了 CPLD 和 FPGA 在电路中的作用。

2.5～2.7 节讲解实际项目开发过程中硬件分析的方法，包括如何进行原理图分析、时序分析以及如何快速地从芯片手册获取有效信息。

2.8 节讲解了调试过程中常用仪器、仪表的使用方法，涉及万用表、示波器和逻辑分析仪。

华清远见

2.1 处理器

2.1.1 通用处理器

通用处理器（GPP）并不针对特定的应用领域进行体系结构和指令集的优化，它们具有一般化的通用体系结构和指令集，以支持复杂的运算并易于新开发功能的添加。一般而言，在嵌入式微控制器（MCU）和微处理器（MPU）中会包含一个通用处理器核。

MPU 通常代表一个 CPU（中央处理器），而 MCU 则强调把中央处理器、存储器和外围电路集成在一个芯片中。早期，微控制器被称为单片机，意思是把计算机集成在一个芯片内。嵌入式微控制器也常被称作片上系统（SoC），含义是在一个芯片上设计了整个系统。芯片厂商在推出 MCU 时，往往会有明确的市场定位，如定位于 PDA、MP3、ADSL 等。定位不同的产品可能包含共同的 CPU 核，但是集成的扩展电路则不一样。图 2.1 所示为一个典型的集成了外围电路的 MCU 的结构。

图 2.1　典型的 MCU 内部结构

举个例子，Intel 的 80386 属于微处理器，而内部集成了 80386 处理器、片选单元、中断控制、定时器、看门狗、定时器、串行 I/O、DMA、总线仲裁、DRAM 控制器等的 386EX 则是 80386 微处理器的微控制器版本。

嵌入式微控制器一般由一个 CPU 核和多个外围电路集成，目前主流的 CPU 核如下。

1．Advanced RISC Machines 公司的 ARM

ARM 内核的设计技术被授权给数百家半导体厂商，做成不同的 SoC 芯片。ARM 的功耗很低，在当今最活跃的无线局域网、3G、手机终端、手持设备、有线网络通信设备等中应用非常广泛。

2．Silicon Graphics 公司的 MIPS

两个最重要的 MIPS 芯片厂商为 PMC 和 IDT。PMC-Sierra 公司的 MIPS 处理器被 CISCO 公司大量采用在高端路由器上。IDT 公司在 MIPS 核上集成 PCI 接口，广泛用于以太网交换，另外也尝试增加了 HDLC、Ethernet、串口、SDRAM 控制器、片选、DMA 控制器等外设接口，以用于低端通信产品。

3．IBM 和 Motorola 的 PowerPC

PowerPC 处理器是通信和工控领域应用最广泛的处理器，国内包括华为、中兴在内的通信公司都大量使用 PowerPC，MPC860 和 MPC8260 是其最经典的两款。

4．Motorola 公司独有的内核 68K/COLDFIRE

68K 内核是最早在嵌入式领域广泛应用的内核，其最著名的代表芯片是 68360。Coldfire 则继承了 68K 的特点并对其保持了兼容。Coldfire 内核被用于 DSP 模块、CAN 总线模块以及一般嵌入式处理器所集成的外设模块，在工业控制、机器人研究、家电控制等领域被广泛采用。

Motorola 的半导体部已经独立为飞思卡尔半导体公司(Freescale Semiconductor Inc.)，因为历史原因，上文仍然使用 Motorola。

中央处理器的体系架构可以分为两类，一类为冯•诺伊曼结构，一类为哈佛结构。

冯•诺伊曼结构也称普林斯顿结构，是一种将程序指令存储器和数据存储器合并在一起的存储器结构。程序指令存储地址和数据存储地址指向同一个存储器的不同物理位置，因此程序指令和数据的宽度相同。而哈佛结构将程序指令和数据分开存储，指令和数据可以有不同的数据宽度。此外，哈佛结构还采用了独立的程序总线和数据总线，分别作为 CPU 与每个存储器之间的专用通信路径，具有较高的执行效率。图 2.2 显示了冯•诺伊曼结构和哈佛结构的区别。

图 2.2　冯•诺伊曼结构与哈佛结构

从指令集的角度来讲，中央处理器也可以分为两类，即 RISC（精简指令集计算机）和 CISC（复杂指令集计算机）。CSIC 强调增强指令的能力、减少目标代码的数

量，但是指令复杂，指令周期长；而 RISC 强调尽可能减少指令集、指令单周期执行，但是目标代码会更大。ARM、MIPS、PowerPC 等 CPU 内核都采用了 RISC 指令集。目前，RISC 和 CSIC 的融合非常明显。

2.1.2 数字信号处理器

数字信号处理器（DSP）针对通信、图像、语音和视频处理等领域的算法而设计。它包含独立的硬件乘法器。DSP 的乘法指令一般在单周期内完成，且优化了卷积、数字滤波、FFT（快速傅立叶变换）、相关、矩阵运算等算法中的大量重复乘法。

DSP 一般采用如图 2.3 所示的改进的哈佛架构，它具有独立的地址总线和数据总线，两条总线由程序存储器和数据存储器分时共用。

图 2.3 改进的哈佛结构

DSP 分为两类，一类是定点 DSP，一类是浮点 DSP。浮点 DSP 的浮点运算用硬件来实现，可以在单周期内完成，因而其浮点运算处理速度高于定点 DSP。而定点 DSP 只能用定点运算模拟浮点运算。

德州仪器（TI）、美国模拟器件公司（ADI）是全球 DSP 的两大主要厂商。

TI 的 TMS320™DSP 平台包含了功能不同的多个系列：2000 系列、3000 系列、4000 系列、5000 系列、6000 系列，工程师也习惯称其为 2X、3X、4X、5X、6X。除了 2000 与 5000 系列是定点 DSP 外，其余的均为浮点 DSP。

ADI 主要有 16 位定点的 21xx 系列、32 位浮点的 SHARC 系列、从 SHARC 系列发展而来的 TigerSHARC 系列及高性能 16 位 DSP 信号处理能力与通用微控制器方便性相结合的 blackfin 系列等。

通用处理器和数字信号处理器也有相互融合以取长补短的趋势，如数字信号控制器（DSC）即为 MCU+DSP，blackfin 系列属于 DSC。目前，芯片厂商推出了许多 ARM+DSP 的双核处理器。

除了上述的通用微控制器和数字信号处理器外，还有一些针对特定领域而设计的专用处理器（ASP），它们都是针对一些特定应用而设计的，如用于 HDTV、ADSL、Cable Modem 等的专用处理器。

网络处理器是一种可编程器件，它应用于电信领域的各种任务，如包处理、协议分析、路由查找、声音/数据的汇聚、防火墙、QoS 等。网络处理器器件内部通常由若干

微码处理器和若干硬件协处理器组成，多个微码处理器在网络处理器内部并行处理，通过预先编制的微码来控制处理流程。而对于一些复杂的标准操作（如内存操作、路由表查找算法、QoS 的拥塞控制算法、流量调度算法等）则采用硬件协处理器来进一步提高处理性能，从而实现了业务灵活性和高性能的有机结合。

对于某些应用场合，使用 ASIC（专用集成电路）往往是低成本且高性能的方案。ASIC 专门针对特定应用而设计，不具备也不需要灵活的编程能力。使用 ASIC 完成同样地功能往往比直接使用 CPU 资源或 CPLD（复杂可编程逻辑器件）/FPGA（现场可编程门阵列）更廉价、高效。

在实际项目的硬件方案中，往往会根据应用的需求选择通用处理器、数字信号处理器、特定领域处理器、CPLD/FPGA 或 ASIC 之一的解决方案，在复杂的系统中，这些芯片可能会同时存在，协同合作，各自发挥自己的长处。如在一款智能手机中，可使用 MCU 处理图形用户界面和用户的按键输入并运行多任务操作系统，使用 DSP 进行音视频编解码，而在射频方面则采用 ASIC。

综合 2.1 节的内容，我们可得出如图 2.4 所示的处理器分类。

图 2.4　处理器分类

2.2 存储器

存储器可分为只读储存器（ROM）、闪存（Flash）、随机存取存储器（RAM）、光介质存储器和磁介质存储器。

ROM 还可再细分为不可编程 ROM、可编程 ROM（PROM）、可擦除可编程 ROM（EPROM）和电可擦除可编程 ROM（E^2PROM），E^2PROM 完全可以用软件来擦写，已经非常方便了。

目前 ROM 有被 Flash 替代的趋势，NOR（或非）和 NAND（与非）是市场上两种主要的 Flash 闪存技术。Intel 公司于 1988 年首先开发出 NOR Flash，彻底改变了原先由 EPROM 和 E^2PROM 一统天下的局面。紧接着，1989 年，东芝公司开发出 NAND Flash，每比特的成本被大大降低。

NOR Flash 和 CPU 的接口属于典型的类 SRAM 接口（如图 2.5 所示），不需要增加额外的控制电路。NOR Flash 的特点是可芯片内执行(XIP，eXecute In Place)，程序可以直接在 NOR 内运行。而 NAND Flash 和 CPU 的接口必须由相应的控制电路进行转换，当然也可以通过地址线或 GPIO 产生 NAND Flash 接口的信号。NAND Flash 以块方式进行访问，不支持芯片内执行。

公共闪存接口（Common Flash Interface，简称 CFI）是一个公开的、标准的从 NOR Flash 器件中读取数据的接口。它可以使系统软件查询已安装的 Flash 器件的各种参数，包括器件阵列结构参数、电气和时间参数以及器件支持的功能等。利用 CFI，在不修改系统软件的情况下，就可以用新型的和改进的产品代替旧版本的产品。

图 2.5　典型的类 SRAM 接口

一个 NAND Flash 的接口主要包含如下信号。

- I/O 总线：地址、指令和数据通过这组总线传输，一般为 8 位或 16 位。
- 芯片启动（Chip Enable，CE#）：如果没有检测到 CE#信号，那么，NAND 器件就保持待机模式，不对任何控制信号做出响应。
- 写使能（Write Enable，WE#）： WE#负责将数据、地址或指令写入 NAND 之中。
- 读使能（Read Enable，RE#）： RE#允许数据输出。
- 指令锁存使能（Command Latch Enable，CLE）： 当 CLE 为高时，在 WE#信号的上升沿，指令将被锁存到 NAND 指令寄存器中。
- 地址锁存使能（Address Latch Enable，ALE）：当 ALE 为高时，在 WE#信号的上升沿，地址将被锁存到 NAND 地址寄存器中。

- 就绪/忙（Ready/Busy，R/B#）：如果 NAND 器件忙，R/B#信号将变低。该信号是漏极开路，需要采用上拉电阻。

NAND Flash 与 NOR Flash 相比容量大、价格低；NAND Flash 中每个块的最大擦写次数是 100 万次，而 NOR 的擦写次数是 10 万次；NAND Flash 的擦除、编程速度远超过 NOR Flash。

由于 Flash 固有的电器特性，在读写数据过程中偶然会产生 1 位或几位数据错误，即位反转，NAND Flash 发生位反转的几率要远大于 NOR Flash。位反转无法避免，因此，使用 NAND Flash 的同时，应采用错误探测/错误更正（EDC/ECC）算法。

Flash 的编程原理都是只能将 1 写为 0，而不能将 0 写为 1。所以在 Flash 编程之前，必须将对应的块擦除，而擦除的过程就是把所有位都写为 1 的过程，块内的所有字节变为 0xFF。

许多嵌入式系统都提供了 IDE（Integrated Drive Electronics）接口，以供连接硬盘控制器或光驱，IDE 接口的信号与 SRAM 类似。人们通常也把 IDE 接口称为 ATA（Advanced Technology Attachment）接口，技术角度而言并不准确。其实，ATA 接口发展至今，已经经历了 ATA-1（IDE）、ATA-2（EIDE Enhanced IDE/Fast ATA）、ATA-3（FastATA-2）、Ultra ATA、Ultra ATA/33、Ultra ATA/66、Ultra ATA/100 及 Serial ATA 的发展过程。

以上所述的各种 ROM、Flash 和磁介质存储器都属于非易失性存储器（NVM），掉电信息不会丢失，而 RAM 则与此相反。

RAM 也可再分为静态 RAM（SRAM）和动态 RAM（DRAM）。DRAM 以电荷形式进行存储，数据存储在电容器中。由于电容器会由于漏电而导致电荷丢失，因而 DRAM 器件需要定期被刷新。SRAM 是静态的，只要供电它就会保持一个值，SRAM 没有刷新周期。每个 SRAM 存储单元由 6 个晶体管组成，而 DRAM 存储单元由 1 个晶体管和 1 个电容器组成。

通常所说的 SDRAM、DDR SDRAM 皆属于 DRAM 的范畴，它们采用与 CPU 外存控制器同步的时钟工作（注意，不是 CPU 的工作频率）。与 SDRAM 相比，DDR SDRAM 同时利用了时钟脉冲的上升沿和下降沿传输数据，因此在时钟频率不变的情况下，数据传输频率加倍。此外，还存在使用 RSL（Rambus 发信电平）技术的 RDRAM（Rambus DRAM）和 Direct RDRAM。

针对许多特定场合的应用，嵌入式系统中往往还使用了一些特定类型的 RAM。

1．NVRAM（非易失性 RAM）

既然是 RAM，就是易失性的，为什么会有一类非易失性的 RAM 呢？

实际上，NVRAM 借助带有备用电源的 SRAM 或借助 NVM（如 E^2PROM）存储 SRAM 的信息并恢复来实现非易失性。NVRAM 的特点是完全像 SRAM 一样读写，而且写入的信息掉电不丢失，不需要 E^2PROM 和 Flash 的特定擦除和编程操作。NVRAM 多用于存放系统中的参数信息。

2．DPRAM（双端口 RAM）

DPRAM 的特点是可以通过两个端口同时访问，具有两套完全独立的数据总线、

地址总线线和读写控制线，通常用于两个处理器之间交互数据，如图 2.6 所示。当一端被写入数据后，另一端可以通过轮询或中断获知，并读取其写入的数据。由于双 CPU 同时访问 DPRAM 时的仲裁逻辑电路集成在 DPRAM 内部，因而硬件工程师设计电路的原理比较简单。

DPRAM 的优点是通信速度快、实时性强、接口简单，而且两边 CPU 都可主动进行数据传输。除了双端口 RAM 以外，目前 IDT 等芯片厂商还推出了多端口 RAM，可以供 3 个以上的 CPU 互通数据。

图 2.6　双端口 RAM

3．CAM（内容寻址 RAM）

CAM 是按内容进行寻址的存储器，是一种特殊的存储阵列 RAM，它的主要工作机制就是将一个输入数据项与存储在 CAM 中的所有数据项自动同时进行比较，判别该输入数据项与 CAM 中存储的数据项是否相匹配，并输出该数据项对应的匹配信息。

如图 2.7 所示，在 CAM 中，输入的是所要查询的数据，输出的是数据地址和匹配标志。若匹配（即搜寻到数据），则输出数据地址。CAM 用于数据检索的优势是软件无法比拟的，可以极大地提高系统性能。

图 2.7　CAM 的输入与输出

4．FIFO（先进先出队列）

FIFO 存储器的特点是先进先出，进出有序，FIFO 多用于数据缓冲。FIFO 和 DPRAM 类似，具有两个访问端口，但是 FIFO 两边的端口并不对等，某一时刻只能被设置为一边作为输入，一边作为输出。

如果 FIFO 的区域共为 *n* 个字节，我们只能通过循环 *n* 次读取同一个地址才能将该片区域读出，不能指定偏移地址。对于有 *n* 个数据的 FIFO，当循环读取 *m* 次，下一次读会自动读取到第 *m*+1 个数据，这是由 FIFO 本身的特性决定的。

总结 2.2 节的内容，我们可得出如图 2.8 所示的存储器分类。

图 2.8　存储器分类

2.3 接口与总线

2.3.1　串口

RS-232、RS-422 与 RS-485 都是串行数据接口标准，最初都是由电子工业协会（EIA）制定并发布的。

RS-232 在 1962 年发布，命名为 EIA-232-E。之后发布的 RS-422 定义了一种平衡通信接口，它是一种单机发送、多机接收的单向、平衡传输规范，被命名为 TIA/EIA-422-A 标准。RS-422 改进了 RS-232 通信距离短、速率低的缺点。为进一步扩展应用范围，EIA 又于 1983 年在 RS-422 的基础上制定了 RS-485 标准，增加了多点、双向通信能力，即允许多个发送器连接到同一条总线上，同时增加了发送器的驱动能力和冲突保护特性，并扩展了总线共模范围，被命名为 TIA/EIA-485-A 标准。

1969 年发布的 RS-232 修改版 RS-232C 是嵌入式系统应用最广泛的串行接口，它为连接 DTE（数据终端设备）与 DCE（数据通信设备）而制定。RS-232C 规标准接口有 25 条线（4 条数据线、11 条控制线、3 条定时线、7 条备用和未定义线），常用的只有 9 根，它们是 RTS/CTS（请求发送/清除发送流控制）、RxD/TxD（数据收发）、DSR/DTR（数据终端就绪/数据设置就绪流控制）、DCD（数据载波检测，也称 RLSD）、Ringing-RI（振铃指示）、SG（信号地）信号。RTS/CTS、TxD/RxD、DRS/DTR 等信号的定义如下。

- RTS：用来表示 DTE 请求 DCE 发送数据，当终端要发送数据时，使该信号有效。
- CTS：用来表示 DCE 准备好接收 DTE 发来的数据，是对 RTS 的响应信号。
- TxD：DTE 通过 TxD 将串行数据发送到 DCE。
- RxD：DTE 通过 RxD 接收从 DCE 发来的串行数据。
- DSR：有效（ON 状态）则表明 DCE 可以使用。
- DTR：有效（ON 状态）则表明 DTE 可以使用。
- DCD：当本地 DCE 设备收到对方 DCE 设备送来的载波信号时，使 DCD 有效，通知 DTE 准备接收，并且由 DCE 将接收到的载波信号解调为数字信号，经 RXD 线送给 DTE。
- Ringing-RI：当调制解调器收到交换台送来的振铃呼叫信号时，使该信号有效（ON 状态），通知终端，已被呼叫。

在嵌入式系统中，并不太注重 DTE 和 DCE 的概念，而 RS-232C 也很少用来连接调制解调器，多使用 RS-232C 进行对等通信，如 Windows 超级终端、Linux minicom 用来连接电路板控制台等。最简单的 RS-232C 串口只需要连接 RxD、TxD、SG 这 3 个信号，使用 XON/XOFF 软件流控。

组成一个 RS-232C 串口的硬件原理如图 2.9 所示，从 CPU 到连接器依次为 CPU、UART（通用异步接收器发送器，作用是完成并/串转换）、CMOS/TTL 电平与 RS-232C 电平转换器、DB9/DB25 或自定义的连接器。

图 2.9　RS-232C 串口电路原理

2.3.2　I²C 总线

I²C（内置集成电路）总线是由 Philips 公司开发的两线式串行总线，产生于 20 世纪 80 年代，用于连接微控制器及其外围设备。I²C 总线简单而有效，占用很少的 PCB（印

刷电路板）空间，芯片管脚数量少，设计成本低。I^2C 总线支持多主控（multi-mastering）模式，任何能够进行发送和接收的设备都可以成为主设备。主控能够控制数据的传输和时钟频率，在任意时刻只能有一个主控。

组成 I^2C 总线的两个信号为数据线 SDA 和时钟线 SCL。为了避免总线信号的混乱，要求各设备连接到总线的输出端必须是开漏输出或集电极开路输出的结构。总线空闲时，上拉电阻使 SDA 和 SCL 线都保持高电平。根据开漏输出或集电极开路输出信号的“线与”逻辑，I^2C 总线上任意器件输出低电平都会使相应总线上的信号线变低。

“线与”逻辑指的是两个或两个以上的输出直接互连就可以实现“AND”的逻辑功能，只有输出端是开漏（对于 CMOS 器件）输出或集电极开路（对于 TTL 器件）输出时才满足此条件。工程师一般以“OC 门”简称开漏或集电极开路。

I^2C 设备上的串行数据线 SDA 接口电路是双向的，输出电路用于向总线发送数据，输入电路用于接收总线上的数据。同样地，串行时钟线 SCL 也是双向的，作为控制总线数据传送的主机要通过 SCL 输出电路发送时钟信号，并检测总线上 SCL 上的电平以决定什么时候发下一个时钟脉冲电平；作为接收主机命令的从设备需按总线上 SCL 的信号发送或接收 SDA 上的信号，它也可以向 SCL 线发出低电平信号以延长总线时钟信号周期。

当 SCL 稳定在高电平时，SDA 由高到低的变化将产生一个开始位，而由低到高的变化则产生一个停止位，如图 2.10 所示。

图 2.10　I^2C 总线开始位和停止位

开始位和停止位都由 I^2C 主设备产生。在选择从设备时，如果从设备采用 7 位地址，则主设备在发起传输过程前，需先发送 1 字节的地址信息，前 7 位为设备地址，最后 1 位为读写标志。之后，每次传输的数据也是 1 个字节，从 MSB 位开始传输。每个字节传完后，在 SCL 的第 9 个上升沿到来之前，接收方应该发出 1 个 ACK 位。SCL 上的时钟脉冲由 I^2C 主控方发出，在第 8 个时钟周期之后，主控方应该释放 SDA，I^2C 总线的时序如图 2.11 所示。

图 2.11　I^2C 总线时序

2.3.3　USB

USB（通用串行总线）是 Intel、Microsoft 等厂商为解决计算机外设种类的日益增加与有限的主板插槽和端口之间的矛盾而于 1995 年提出的，它具有数据传输率高、易扩展、支持即插即用和热插拔的优点，目前已得到广泛的应用。

USB1.1 包含全速和低速两种模式，低速方式的速率为 1.5Mbit/s，支持一些不需要很大数据吞吐量和很高实时性的设备，如鼠标等。全速模式为 12Mbit/s，可以外接速率更高的外设。在 USB 2.0 中，增加了一种高速方式，数据传输率达到 480Mbit/s，可以满足更高速外设的需要。

USB 的物理拓扑结构如图 2.12 所示，在 USB 2.0 中，高速方式下 Hub 使全速和低速方式的信令环境独立出来，图 2.13 所示为高速方式下 Hub 的作用。

在嵌入式系统中，电路板若需要挂接 USB 设备（device），则需提供 USB 主机（host）控制器和连接器；若电路板需要作为 USB 设备，则需提供 USB 设备适配器和连接器。有的 MCU 集成了 USB 主机控制器和设备适配器。

图 2.12　USB 的物理拓扑

图 2.13　USB 2.0 连接高速、全/低速设备

USB 总线的机械连接非常简单，采用 4 芯的屏蔽线，一对差分线（D+，D−）传

送信号，另一对（VBUS，电源地）传送+5V 的直流电。一个 USB 主控制器端口最多可连接 127 个器件，各器件之间的距离不超过 5 米。

USB 提供了 4 种传输方式以适应各种设备的需要，说明如下。

（1）控制（Control）传输方式。

控制传输是双向传输，数据量通常较小，主要用来进行查询、配置和给 USB 设备发送通用的命令。

（2）同步（Synchronization）传输方式。

同步传输提供了确定的带宽和间隔时间，它被用于时间严格并具有较强容错性的流数据传输，或者用于要求恒定的数据传送率的即时应用。例如进行语音业务传输时，使用同步传输方式是很好的选择。

（3）中断（Interrupt）传输方式。

中断方式传送是单向的，对于 USB 主机而言，只有输入。中断传输方式主要用于定时查询设备是否有中断数据要传送，该传输方式应用在少量的、分散的、不可预测的数据传输场合，键盘、游戏杆和鼠标属于这一类型。

（4）批量（Bulk）传输方式。

批量传输主要应用在没有带宽和间隔时间要求的大量数据的传送和接收，它进行有保证的传输。打印机和扫描仪均采用这种传输类型。

2.3.4 以太网接口

以太网接口由 MAC（以太网媒体接入控制器）和 PHY（物理接口收发器）组成。以太网 MAC 由 IEEE-802.3 以太网标准定义，实现了数据链路层。常用的 MAC 支持 10Mbit/s 或 100Mbit/s 两种速率。PHY 则实现物理层功能，IEEE-802.3 标准定义了以太网 PHY，它符合 IEEE-802.3k 中用于 10BaseT（第 14 条）和 100BaseTX（第 24 条和第 25 条）的规范。10BaseT 和 100BaseTX PHY 两种实现的帧格式是一样的，但信令机制不同，而且 10BaseT 采用曼彻斯特编码，100BaseTX 采用 4B/5B 编码。

MAC 和 PHY 之间采用 MII（媒体独立接口）连接，它是 IEEE-802.3 定义的以太网行业标准，包括 1 个数据接口和 1 个 MAC 和 PHY 之间的管理接口。数据接口包括分别用于发送和接收的两条独立信道，每条信道都有自己的数据、时钟和控制信号，MII 数据接口总共需要 16 个信号。MII 管理接口包含两个信号，一个是时钟信号，另一个是数据信号。通过管理接口，上层能监视和控制 PHY。

以太网接口电路原理如图 2.14 所示，从 CPU 到最终接口依次为 CPU、MAC、PHY、以太网隔离变压器、RJ45 插座。以太网隔离变压器是以太网收发芯片与连接器之间的磁性组件，在其两者之间起着信号传输、阻抗匹配、波形修复、信号杂波抑制和高电压隔离作用。

许多处理器内部集成了 MAC 或同时集成了 MAC 和 PHY，另有许多以太网控制芯片也集成了 MAC 和 PHY。

华清远见

图 2.14　以太网接口电路原理

2.3.5　ISA 总线

ISA（工业标准结构总线）总线起源于 1981 年 IBM 公司生产的以 Intel 8088 为 CPU 的 IBM-PC 微计算机，开始时总线宽度为 8 位。1984 年推出的 IBM-PC/AT 系统将 ISA 总线扩充为 16 位数据总线宽度，同时地址总线宽度也由 20 位扩充到了 24 位。其后推出的 EISA（扩展的 ISA）采用 32 位地址线，数据总线也扩展为 32 位，但仍保持了与 ISA 的兼容。

图 2.15 所示为 ISA 总线的信号，这些信号可分为 3 组。

- 总线基本信号：ISA 总线工作所需要的最基本信号，含复位、时钟、电源、地等。
- 总线访问信号：用于访问 ISA 总线设备的地址线、数据线以及相应的应答信号。
- 总线控制信号：中断和 DMA 请求。

图 2.15 中各信号的详细定义如下。

- RESET、BCLK：复位及总线基本时钟，BLCK 为 8MHz。
- SA19～SA0：存储器及 I/O 空间 20 位地址，带锁存。
- LA23～LA17：存储器及 I/O 空间 20 位地址，不带锁存。
- BALE：总线地址锁存，外部锁存器的选通。
- AEN：地址允许，表明 CPU 让出总线，DMA 开始。
- SMEMR#、SMEMW#：8 位 ISA 存储器读写控制。
- MEMR#、MEMW#：16 位 ISA 存储器读写控制。
- SD15～SD0：数据总线，访问 8 位 ISA 卡时高 8 位自动传送到 SD7～SD0。
- SBHE#：高字节允许，打开 SD15～SD8 数据通路。
- MEMCS16#、IOCS16#：ISA 卡发出此信号确认可以进行 16 位传送。
- I/OCHRDY：ISA 卡准备信号，可控制插入等待周期。
- NOWS#：有效则暗示不用插入等待周期。
- I/OCHCK#：ISA 卡奇偶校验错。
- IRQ15、IRQ14、IRQ12～IRQ9、IRQ7～IRQ3：中断请求。
- DRQ7～DRQ5、DRQ3～DRQ0：ISA 卡 DMA 请求信号。
- DACK7#～DACK5#、DACK3#～DACK0#：DMA 请求响应信号。
- MASTER#：ISA 主模块确立信号，ISA 发出此信号，与主机内 DMAC（DMA

华清远见

控制器）配合使 ISA 卡成为主模块。

图 2.15　ISA 总线信号

2.3.6　PCI 和 cPCI

PCI（外围部件互连）是 Intel 公司于 1991 年推出的一种局部总线，作为一种通用的总线接口标准，它在目前的计算机系统中得到了非常广泛的应用。PCI 提供了一组完整的总线接口规范，其目的是描述如何将计算机系统中的外围设备以一种结构化和可控化的方式连接在一起，给出了外围设备在连接时的电气特性和行为规约，并且详细定义了计算机系统中的各个不同部件之间应该如何正确地进行交互。PCI 总线具有如下特点。

- 数据总线 32 位，可扩充到 64 位。
- 可进行突发（burst）模式传输。

> 突发方式传输是指取得总线控制权后连续进行多个数据的传输。突发传输时，只需要给出目的地的首地址，访问第 1 个数据后，第 2 ~ *n* 个数据会在首地址基础上按一定规则自动被寻址和传输。与突发方式对应的是单周期方式，它在 1 个总线周期只传送 1 个数据。

- 总线时钟频率为 33MHz 或 66MHz，最高传输率可达 528MB/s。
- 采用中央集中式总线仲裁。
- 支持全自动配置、资源分配，PCI 卡内有设备信息寄存器组为系统提供卡的信息，可实现即插即用。

- PCI 总线规范独立于微处理器，通用性好。
- PCI 设备可以完全作为主控设备控制总线。

图 2.16 所示为一个典型的基于 PCI 总线的计算机系统逻辑示意图，系统的各个部分通过 PCI 总线和 PCI-PCI 桥连接在一起。CPU 和 RAM 通过 PCI 桥连接到 PCI 总线 0（即主 PCI 总线），而具有 PCI 接口的显卡则可以直接连接到主 PCI 总线上。PCI-PCI 桥是一个特殊的 PCI 设备，它负责将 PCI 总线 0 和 PCI 总线 1（即从 PCI 主线）连接在一起，通常 PCI 总线 1 称为 PCI-PCI 桥的下游（downstream），而 PCI 总线 0 则称为 PCI-PCI 桥的上游（upstream）。为了兼容旧的 ISA 总线标准，PCI 总线还可以通过 PCI-ISA 桥来连接 ISA 总线，从而支持以前的 ISA 设备。

图 2.16　基于 PCI 总线的计算机系统

当 PCI 卡刚加电时，卡上只有配置空间是可被访问的，因而 PCI 卡开始不能由驱动或用户程序访问，这与 ISA 卡有本质的区别（CPU 可直接读取 ISA 卡在存储空间或 I/O 空间映射的地址）。PCI 配置空间保存着该卡工作所需的所有信息，如厂家、卡功能、资源要求、处理能力、功能模块数量、主控卡能力等。通过对这个空间信息的读取与编程，可完成对 PCI 卡的配置。图 2.17 中 PCI 配置空间共为 256 字节，主要包括如下信息。

- 制造商标识（Vendor ID）：由 PCI 组织分配给厂家。
- 设备标识（Device ID）：按产品分类给本卡的编号。
- 分类码（Class Code）：本卡功能的分类码，如图卡、显示卡、解压卡等。
- 申请存储器空间：PCI 卡内有存储器或以存储器编址的寄存器和 I/O 空间，为使驱动程序和应用程序能访问它们，需申请 CPU 的一段存储区域以进行定位。配置空间的基地址寄存器用于此目的。
- 申请 I/O 空间：配置空间的基地址寄存器也用来进行系统 I/O 空间的申请。
- 中断资源申请：配置空间中的中断引脚和中断线用来向系统申请中断资源。中断资源的申请通过中断引脚（interrupt pin）和中断线（interrupt line）来完成的。偏移 3DH 处为中断引脚寄存器，其值表明 PCI 设备使用了哪一个中断引脚，对应关系为 1-INTA#、2-INTB#、3-INTC#、4-INTD#。

PCI 总线上的信号大体可分为如下几组：

- 系统接口信号；

- 地址与数据接口信号；
- 接口控制信号；
- 仲裁信号；
- 错误报告信号；
- 中断接口信号；
- 其他接口信号。

<table>
<tr><td colspan="2">31 … 16</td><td colspan="2">15 … 0</td><td></td></tr>
<tr><td colspan="2">设备标识</td><td colspan="2">制造商标识</td><td>00H</td></tr>
<tr><td colspan="2">状态</td><td colspan="2">命令</td><td>04H</td></tr>
<tr><td colspan="3">分类码</td><td>修正标志</td><td>08H</td></tr>
<tr><td>BIST</td><td>头类型</td><td>延迟定时器</td><td>行大小</td><td>0CH</td></tr>
<tr><td colspan="4" rowspan="6">基地址寄存器</td><td>10H</td></tr>
<tr><td>14H</td></tr>
<tr><td>18H</td></tr>
<tr><td>1CH</td></tr>
<tr><td>20H</td></tr>
<tr><td>24H</td></tr>
<tr><td colspan="4">卡总线 CIS 指针</td><td>28H</td></tr>
<tr><td colspan="2">子系统标识</td><td colspan="2">子系统制造商标识</td><td>2CH</td></tr>
<tr><td colspan="4">扩展 ROM 基地址</td><td>30H</td></tr>
<tr><td colspan="3">保留</td><td>容量指针</td><td>34H</td></tr>
<tr><td colspan="4">保留</td><td>38H</td></tr>
<tr><td>Max_Lat</td><td>Max_Gnt</td><td>中断引脚</td><td>中断线</td><td>3CH</td></tr>
</table>

图 2.17　PCI 配置空间

如图 2.18 所示，这些信号的详细定义如下。

- CLK：系统时钟。
- AD31～AD0：地址和数据复用信号线信号。
- C/BE3～C/BE0：总线命令和地址使能信号。
- PAR：奇偶校验信号。
- FRAM E# ：帧周期信号，指示总线操作起始和终止。
- IRDY# ：主设备准备好信号。
- TRDY# ：目标设备准备好信号。
- STOP# ：目标设备要求终止当前数据传输信号。
- DEVSEL #：目标设备选中信号。
- IDSEL：配置空间读写时的片选信号。
- LOCK#：总线锁定信号。

- RST#：复位信号。
- INTA#、INTB#、INTC#和 INTD#： 中断请求。
- REQ#、GNT#：PCI 总线请求与仲裁后的授权。
- AD63～AD32、C/BE7～C/BE4 等：作用于扩展的 64 位 PCI 总线。

图 2.18　PCI 总线信号

cPCI（Compact PCI，紧凑型 PCI）是以 PCI 电气规范为标准的高性能工业用总线，结合了 VME（Visa Module Eurocard，维萨信用卡模块欧洲卡）的高性能、可扩展性和可靠性与 PCI 标准的经济有效和灵活性。cPCI 的 CPU 及外设与标准 PCI 是相同的，使用与传统 PCI 相同的芯片和软件，操作系统、驱动和应用程序都感觉不到两者的区别。图 2.19 所示为与 cPCI 总线相关的板卡、背板和机箱，应用于工业控制和大型通信设备。

图 2.19　cPCI 板卡、背板与机箱

2.4 CPLD 和 FPGA

CPLD（复杂可编程逻辑器件）由完全可编程的与或门阵列以及宏单元构成。

CPLD 中基本逻辑单元是宏单元，宏单元由一些“与或”阵列加上触发器构成，其中“与或”阵列完成组合逻辑功能，触发器完成时序逻辑。宏单元中与阵列的输出称为乘积项，其数量标志了 CPLD 的容量。乘积项阵列实际上就是一个“与或”阵列，每一个交叉点都是一个可编程熔丝，如果导通就是实现“与”逻辑。在“与”阵列后一般还有一个“或”阵列，用以完成最小逻辑表达式中的“或”关系。图 2.20 给出了非常典型的 CPLD 的单个宏单元的结构。

图 2.20　典型的 CPLD 宏单元

图 2.21 给出了一个典型 CPLD 的整体结构。这个 CPLD 由 LAB（逻辑阵列模块，由多个宏单元组成）通过 PIA （可编程互连阵列）互连组成，而 CPLD 与外部的接口则由 I/O 控制模块提供。

图中宏单元的输出会经 I/O 控制块送至 I/O 引脚，I/O 控制块控制每一个 I/O 引脚的工作模式，决定其为输入、输出还是双向引脚，并决定其三态输出的使能端控制。

与 CPLD 不同，FPGA（现场可编程门阵列）基于 LUT（查找表）工艺。查找表本质上是一片 RAM，当用户通过原理图或 HDL（硬件描述语言）语言描述了一个逻辑电路以后，FPGA 开发软件会自动计算逻辑电路的所有可能的结果，并把结果事先

写入 RAM。这样，输入一组信号进行逻辑运算就等于同输入一个地址进行查表，并输出对应地址的内容。

图 2.22 所示为一个典型的 FPGA 的内部结构。这个 FPGA 由 IOC（输入/输出控制模块）、EAB（嵌入式阵列块）、LAB 和 FAST TRACK（快速通道互连）构成。

IOC 是内部信号到 I/O 引脚的接口，它位于 FAST TRACK 的行和列的末端，每个 IOC 包含一个双向 I/O 缓冲器和一个既可做输入寄存器，也可做输出寄存器的触发器。

图 2.21　典型 CPLD 的结构

EAB（嵌入式存储块）是一种输入输出端带有寄存器的非常灵活的 RAM。EAB 不仅可以用作存储器，也可以事先被写入查表值来以构成如乘法器、纠错逻辑等电路。当用于 RAM 时，EAB 可配制成 8 位、4 位、2 位和 1 位长度的数据格式。

LAB 主要用于逻辑电路设计，一个 LAB 包括多个 LE（逻辑单元），每个 LE 包括组合逻辑及一个可编程触发器。一系列 LAB 构成的逻辑阵列用来实现普通逻辑功能，如计数器、加法器、状态机等。

器件内部信号的互连和器件引出端之间的信号互连由 FAST TRACK 连线提供，FAST TRACK 遍布于整个 FPGA 器件，是一系列水平和垂直走向的连续式布线通道。

图 2.22　典型 FPGA 的结构

表 2.1 所示为一个 4 输入 LUT 的实际逻辑电路与 LUT 实现方式的对应关系。

表 2.1　　实际逻辑电路与查找表的实现

实际逻辑电路		LUT 的实现方式	
A、b、c、d 输入	逻辑输出	地址	RAM 中存储的内容
0000	0	0000	0
0001	0	0001	0
...	0	...	0
1111	1	1111	1

CPLD 和 FPGA 的主要厂商有 Altera、Xilinx 和 Lattice 等，采用专门的开发流程，在设计阶段使用 HDL 语言（如 VHDL、Verilog HDL）编程。它们可以实现许多复杂的功能，如实现 UART、I^2C 等 I/O 控制芯片、通信算法、音视频编解码算法等，甚至还可以直接集成 ARM 等 CPU 核和外围电路。

对于驱动工程师而言，我们只需要这样看待 CPLD 和 FPGA：如果它完成的是特定的接口和控制功能，我们就直接把它当成由很多逻辑门（与、非、或、D 触发器）组成的完成一系列时序逻辑和组合逻辑的 ASIC；如果它完成的是 CPU 的功能，我们就直接把它当成 CPU。驱动工程师眼里的硬件比 IC 设计师要宏观。

2.5 原理图分析

2.5.1　原理图分析的内容

原理图分析的含义是指通过阅读电路板的原理图获得各种存储器、外设所使用的硬件资源，主要包括存储器和外设控制芯片所使用的片选、中断和 DMA 资源。通过分析片选得出芯片的内存、I/O 基地址，通过分析中断、DMA 信号获得芯片使用的中断号和 DMA 通道，并归纳出如表 2.2 所示的表格。

表 2.2　　存储器、外设控制器资源占用表

芯　片	片　选	基 地 址	字长	大　小	中 断 号	DMA 通
Flash	CS0	0x0	16	0x00100000	—	—
SDRAM	CS1	0x00100000	32	0x00800000	—	—
UART	CS2	0x00900000	16	0x00000400	3	0

以太网控制器	CS3	0x00A00000	16	0x00000400	9	1
…	…	…	…	…	…	…

上述表格对驱动开发的意义重大，在大多数情况下，硬件工程师已经给驱动工程师提供了这个表格。

华清远见

2.5.2 原理图的分析方法

原理图的分析方法是以 CPU 为中心向存储器和外设辐射，步骤如下。

（1）阅读 CPU 部分，获知 CPU 的哪些片选、中断和集成的外设控制器被使用，列出这些元素 a、b、c…。

CPU 引脚比较多的时候，芯片可能会被分成几个模块单独被画在原理图的不同页上，这时候应该把相应的部分都分析到位。

（2）对第一步中列出的元素从原理图中对应的外设和存储器电路中分析出实际的使用情况。

硬件原理图中包含如下元素。

- 符号（symbol）。

symbol 描述芯片的外围引脚以及引脚的信号，复杂的芯片可能被分割为几个 symbol。在 symbol 中，一般把属于同一个信号群的引脚排列在一起。图 2.23 给出了 NOR Flash AM29LV800B 和 E28F128J3A 的 symbol。

图 2.23 原理图中的 symbol

- 网络（net）。

描述芯片、接插件和分离元器件引脚之间的互连关系，每个网络需要根据信号的定义赋予一个合适的名字，如果没有给网络取名字，EDA 软件会自动添加一个默认的网络名。添加网络后的 E28F128J3A 如图 2.24 所示。

- 描述。

原理图中会添加一些文字来辅助描述原理图（类似于源代码中的注释），如每页页脚会有该页的功能描述，对重要的信号，在原理图的相应 symbol 和 net 也会附带文字说明。图 2.25 所示为原理图中描述的例子。

图 2.24　原理图中的 net

图 2.25　原理图中的描述

2.6 硬件时序分析

2.6.1　时序分析的概念

对驱动工程师或硬件工程师而言，时序分析的意思是让芯片之间的访问满足芯片手册中时序图信号有效的先后顺序、采样建立时间（setup time）和保持时间（hold time）的要求，在电路板工作不正常的时候，准确地定位时序方面的问题。

图 2.26　建立时间和保持时间

建立时间是指在触发器的时钟信号边沿到来以前，数据已经保持稳定不变的时间，如果建立时间不够，数据将不能在这个时钟边沿被正确采样；保持时间是指在触发器的时钟信号边沿到来以后，数据还需稳定不变的时间，如果保持时间不够，数据同样不能被正确采样。如图 2.26 所示，数据稳定传输必须满足建立和保持时间的要求，当然在一些情况下，建立时间和保持时间的值可以为 0。

在工具方面，SynaptiCAD 公司的 Timing Diagrammer Pro 是一种非常好的数字/模拟时序图编辑器及分析引擎。

2.6.2　典型硬件时序

最典型的硬件时序是 SRAM 的读写时序，在读/写过程中涉及的信号包括地址、数据、片选、读/写、字节使能和就绪/忙。对于一个 16 位、32 位（甚至 64 位）的 SRAM，字节使能表明哪些字节被读写。

图 2.27 给出了 SRAM 的一个读时序，写时序与此相似。首先是地址总线上输出要读写的地址，然后 SRAM 片选被发出，接着读/写信号被输出，之后读写信号要经历数个等待周期。当 SRAM 读写速度比较慢时，等待周期可以由 MCU 的相应寄存器设置，也可以通过设备就绪/忙（如图 2.27 中的 nWait）向 CPU 报告，这样，读写过程中会自动添加等待周期。

图 2.27　SRAM 读时序图

NOR Flash 和许多外设控制芯片都使用了类似 SRAM 的访问时序，因此，牢固掌握这个时序意义重大。一般，在芯片数据手册给出的时序图中，会给出图中各段时间的含义和要求，真实的电路板必须满足芯片手册上描述的建立时间和保持时间的最小要求。

2.7 仪器仪表使用

2.7.1 万用表

在电路板调试过程中我们主要使用万用表的以下两个功能：

- 测量电平；
- 使用二极管挡测量电路板上网络的连通性，当示波器被设置在二极管挡时，测量连通的网络会发出“滴滴”的鸣叫，否则，没有连通。

2.7.2 示波器

示波器是利用电子示波管的特性，将人眼无法直接观测的交变电信号转换成图像，显示在荧光屏上以便测量的电子仪器。它是观察数字电路实验现象、分析实验中的问题、测量实验结果必不可少的重要仪器。

使用示波器主要应注意调节垂直偏转因数选择（VOLTS / DIV）和微调、时基选择（TIME / DIV）和微调以及触发方式。

如果 VOLTS / DIV 设置不合理，则可能出现电压幅度超出整个屏幕或在屏幕上变动太过微小无法观测的现象。图 2.28 所示同一个波形在 VOLTS / DIV 设置由大到小变化过程中的示意图。

如果 TIME/DIV 设置不合适，则可能造成波形混迭。混迭意味着屏幕上显示的波形频率低于信号的实际频率。这时候，可以通过慢慢改变 TIME/DIV 到较快的时基挡，如果波形的频率参数急剧改变或者晃动的波形在某个较快的时基挡稳定下来，说明之前发生了波形混迭。根据奈奎斯特定理，采样速率至少高于信号高频成分的两倍才不会发生混迭，如一个 500MHz 的信号，至少需要 1GS/s 的采样速率。图 2.29 所示同一个波形在 TIME/DIV 设置由小到大变化过程中的示意图。

（a）50V/DIV

图 2.28　示波器的 VOLTS / DIV 设置与波形

（b）10V/DIV

（c）1V/DIV

图 2.28　示波器的 VOLTS / DIV 设置与波形

奈奎斯特定理即为采样定理，指当采样频率 fs_{max} 大于信号中最高频率 f_{max} 的两倍时，即 $fs_{max}>=2f_{max}$ 时，采样之后的数字信号可完整地保留原始信息。

示波器的触发能使信号在正确的位置点同步水平扫描，使信号特性清晰。触发控制按钮可以稳定重复的波形并捕获单次波形。大多数用示波器的用户只采用边沿触发方式，如果拥有其他触发能力在某些应用上是非常有用的，特别是对新设计产品的故障查寻，先进的触发方式可将所关心的事件分离出来，找出用户所关心的非正常问题，从而最有效地利用采样速率和存储深度。触发能力的提高可以较大提高测试过程的灵活性。

（a）500ns/DIV

（b）50ns/DIV

图 2.29　示波器的 TIME/DIV 设置与波形

2.7.3　逻辑分析仪

逻辑分析仪是利用时钟从测试设备上采集数字信号并进行显示的仪器，其最主要的作用是用于时序的判定。与示波器不同，逻辑分析仪并不具备许多电压等级，通常只显示两个电压（逻辑 1 和 0）。在设定了参考电压之后，逻辑分析仪对待测试信号通过比较器来进行判定，高于参考电压者为 High，低于参考电压者为 Low。

例如，如果以 *n* MHz 采样率测量一个信号，逻辑分析仪会以 1000/*n* ns 为周期采样信号，当参考电压设定为 1.5V 时，超过 1.5V 则判定为 1，低于 1.5V 则为 0，将逻辑 1 和 0 连接成连续的波形，工程师依据此连续波形可寻找时序问题。

高端的逻辑分析仪会安装有 Windows XP 操作系统并提供非常友善的逻辑分析应用软件，在其中可方便地编辑探针、信号并查看波形，如图 2.30 所示。

逻辑分析仪的波形可以显示地址、数据、控制信号及任意外部探钩信号的变化轨迹，在使用之前应先编辑每个探钩的信号名。

图 2.30　逻辑分析仪及配套软件

逻辑分析仪具有超强的逻辑跟踪分析功能，它可以捕获并记录嵌入式处理器的总线周期，也可以捕获如实时跟踪用的 ETM 接口的程序执行信息，并对这些记录进行分析、译码并还原出应用程序的执行过程。因此，可使用逻辑分析仪通过触发接口与 ICD（在线调试器）协调工作以补充 ICD 在跟踪功能方面的不足。逻辑分析仪与 ICD 协作可为工程师提供断点、触发和跟踪调试手段，如图 2.31 所示。

ICD 是一个容易与 ICE（在线仿真器）混淆的概念，ICE 本身需要完全仿真 CPU 的行为，可以从物理上完全替代 CPU，而 ICD 则只是与芯片内部提供的嵌入式 ICE 单元通过 JTAG 等接口互通。因此，对 ICD 的硬件性能要求远低于 ICE。

图 2.31　逻辑分析仪与 ICD 协作

2.8 总结

本章简单地讲解了驱动工程师必备的硬件基础知识，包括处理器、存储器的分类、

原理与用途，并分析了常见的外围设备接口与总线的工作方式。

此外，本章还讲解了原理图、硬件时序分析方法以及万用表、示波器和逻辑分析仪的使用方法。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm
- 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程： 华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：
  http://www.farsight.com.cn/courses/TS-LinuxApp.htm
- 嵌入式 Linux 系统开发班：
  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm
- 嵌入式 Linux 驱动开发班：
  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm

华清远见

第 3 章　Linux 内核及内核编程

由于 Linux 驱动编程本质属于 Linux 内核编程，因此有必要掌握 Linux 内核及内核编程的基础知识。

3.1～3.2 节讲解 Linux 内核的演变及 Linux 2.6 内核的特点。

3.3 节分析了 Linux 内核源代码目录结构和 Linux 内核的组成部分及其关系，并对 Linux 的用户空间和内核空间进行了讲解。

3.4 节讲解了 Linux 2.6 内核的编译及内核引导过程。另外，还描述了在 Linux 内核中新增程序的方法，驱动工程师编写的设备驱动也应该以此方式被添加。

3.5 节讲解了 Linux 下 C 编程的命名习惯以及 Linux 所使用的 GNU C 针对标准 C 的扩展语法。

华清远见

3.1 Linux 内核的发展与演变

Linux 操作系统诞生于 1991 年 10 月 5 日（第一次正式向外公布的时间）。Linux 操作系统的诞生、发展和成长过程依赖于 UNIX 操作系统、MINIX 操作系统、GNU 计划、POSIX 标准和 Internet。

1．UNIX 操作系统

UNIX 操作系统是美国贝尔实验室的 Ken. Thompson 和 Dennis Ritchie 于 1969 年在 DEC PDP-7 小型计算机上开发的一个分时操作系统。Linux 操作系统可看作 UNIX 操作系统的一个克隆版本。

2．Minix 操作系统

Minix 操作系统也是 UNIX 操作系统的一种克隆系统，它于 1987 年由著名计算机教授 Andrew S. Tanenbaum 开发完成。开放源代码 Minix 操作系统的出现在全世界的大学中刮起了学习 UNIX 操作系统的旋风。Linux 操作系统刚开始就是参照 Minix 操作系统开发进行。

3．GNU 计划

GNU 计划和自由软件基金会（FSF）是由 Richard M. Stallman 于 1984 年创办的，GNU 是“GNU’s Not UNIX”的缩写。到 20 世纪 90 年代初，GNU 项目已经开发出许多高质量的免费软件，其中包括 Emacs 编辑系统、Bash Shell 程序、GCC 系列编译程序、GDB 调试程序等。这些软件为 Linux 操作系统的开发创造了一个合适的环境，是 Linux 操作系统诞生的基础之一。没有 GNU 软件环境，Linux 操作系统将寸步难行。因此，严格而言，Linux 应该被称为 GNU/Linux 操作系统。

4．POSIX 标准

POSIX（Portable Operating System Interface for Computing Systems，可移植的操作系统接口）是由 IEEE 和 ISO/IEC 开发的一组标准。该标准基于现有的 UNIX 实践和经验完成，描述了操作系统的调用服务接口，用于保证编制的应用程序可以在源代码一级上在多种操作系统上移植。该标准在推动 Linux 操作系统朝着正规化方向发展起着重要的作用，是 Linux 前进的灯塔。

5．Internet

如果没有 Internet，没有遍布全世界的 Linux 爱好者的无私奉献，那么 Linux 操作系统就不可能发展到现在的水平。

Linux 操作系统重要版本的变迁历史及各版本的主要特点如表 3.1 所示。

表 3.1　　Linux 操作系统版本历史

华清远见

版　本	时　间	特　点
0.1	1991.10	最初的原型
1.0	1994.3	包含了对 386 的官方支持，仅支持单 CPU 系统
1.2	1995.3	第一个包含多平台（Alpha、Sparc、MIPS 等）支持的官方版本
2.0	1996.6	包含很多新的平台支持，是第一个支持 SMP（对称多处理器）体系的内核版本
2.2	1999.1	极大提升 SMP 系统上 Linux 的性能，并支持更多的硬件
2.4	2001.1	进一步地提升了 SMP 系统的扩展性，同时也集成了很多用于支持桌面系统的特性，如 USB、PC 卡（PCMCIA）的支持和内置的即插即用等
2.6	2003.12	无论是对于企业服务器还是对于嵌入式系统，Linux 2.6 都是一个巨大的进步

从表 3.1 可以看出，Linux 操作系统一直朝着支持更多的 CPU、硬件体系结构和外部设备，支持更广泛领域的应用，提供更好的性能方向发展。

除了 Linux 内核本身可提供免费下载以外，一些厂商封装了 Linux 内核和大量有用的软件包，制定了相应的 Linux 发行版，如 Red Hat Linux、TurboLinux、Debian、SuSe，国内的 RedFlag 和 Xteam 等。

针对嵌入式系统的应用，一些改进内核的 Linux 操作系统被开发出来，如改进实时性的 Hard Hat Linux 和 RTLinux，支持不含 MMU CPU 的 McLinux，面向数字相机和 MP3 等微型嵌入式设备的 ThinLinux，以及颇有商业背景的 MontaVista 等。

3.2 Linux 2.6 内核的特点

尽管 Linux 2.4 目前仍然被广泛应用，但是本书是基于 Linux 2.6 内核的。从 2003 年 12 月 Linux 2.6.0 发布至今，一直还处于开发之中，并还将稳定较长一段时间。Linux 2.6 相对于 Linux 2.4 有相当大的改进，主要体现在如下几个方面。

1．新的调度器

2.6 版本的 Linux 内核使用了新的进程调度算法，它在高负载的情况下执行得极其出色，并且当有很多处理器时也可以很好地扩展。

2．内核抢占

在 2.6 版本的 Linux 内核中，内核任务可以被抢占，从而提高系统的实时性。这样做最主要的优势在于可以极大地增强系统的用户交互性。

3．改进的线程模型

2.6 版本的 Linux 内核中线程操作速度得以提高，可以处理任意数目的线程，PID

最大可以到 2000000000。

4．虚拟内存的变化

从虚拟内存的角度来看，新内核融合了 r-map（反向映射）技术，显著改善虚拟内存在一定程度负载下的性能。

5．文件系统

2.6 版本的 Linux 内核增加了对日志文件系统功能的支持，解决了 2.4 版本的 Linux 内核在这方面的不足。2.6 版本的 Linux 内核在文件系统上的关键变化还包括对扩展属性及 POSIX 标准访问控制的支持。Ext2/Ext3 作为大多数 Linux 系统缺省安装的文件系统，在 2.6 版本的 Linux 内核中增加了对扩展属性的支持，可以给指定的文件在文件系统中嵌入元数据。

6．音频

新的 Linux 音频体系结构 ALSA（Advanced Linux Sound Architecture）取代了缺陷很多的旧的 OSS（Open Sound System）。新的声音体系结构支持 USB 音频和 MIDI 设备，并支持全双工重放等功能。

7．总线

SCSI/IDE 子系统经过大幅度的重写，解决和改善了以前的一些问题。比如 2.6 版本的 Linux 内核可以直接通过 IDE 驱动程序来支持 IDE CD/RW 设备，而不必像以前那样使用一个特别的 SCSI 模拟驱动程序。

8．电源管理

支持 ACPI（高级电源配置管理界面，Advanced Configuration and Power Interface），用于调整 CPU 在不同的负载下工作于不同的时钟频率以降低功耗。

9．联网和 IPSec

2.6 版本的 Linux 内核中加入了对 IPSec 的支持，删除了原来内核内置的 HTTP 服务器 kttpd，加入了对新的 NFSv4（网络文件系统）客户机/服务器的支持，并改进了对 IPv6 的支持。

10．用户界面层

2.6 版本的 Linux 内核重写了帧缓冲/控制台层，人机界面层还加入了对大多数接口设备的支持（从触摸屏到盲人用的设备和各种各样的鼠标）。

在设备驱动程序的方面，Linux 2.6 相对于 Linux 2.4 也有较大的改动，这主要表现在内核 API 中增加了不少新功能（例如内存池）、sysfs 文件系统、内核模块从.o 变为.ko、驱动模块编译方式、模块使用计数、模块加载和卸载函数的定义等方面。

华清远见

3.3 Linux 内核的组成

3.3.1 Linux 内核源代码目录结构

本书范例程序所基于的 Linux 2.6.15.5 内核源代码压缩包“linux-2.6.15.5.tar.bz2”共 38.912KB，包含如下目录。

- arch：包含和硬件体系结构相关的代码，每种平台占一个相应的目录，如 i386、ARM、PowerPC、MIPS 等。
- block：块设备驱动程序 I/O 调度。
- crypto：常用加密和散列算法（如 AES、SHA 等），还有一些压缩和 CRC 校验算法。
- Documentation：内核各部分的通用解释和注释。
- drivers：设备驱动程序，每个不同的驱动占用一个子目录，如 char、block、net、mtd、i2c 等。
- fs：支持的各种文件系统，如 EXT、FAT、NTFS、JFFS2 等。
- include：头文件，与系统相关的头文件被放置在 include/linux 子目录下。
- init：内核初始化代码。
- ipc：进程间通信的代码。
- kernel：内核的最核心部分，包括进程调度、定时器等，而和平台相关的一部分代码放在 arch/*/kernel 目录下。
- lib：库文件代码。
- mm：内存管理代码，和平台相关的一部分代码放在 arch/*/mm 目录下。
- net：网络相关代码，实现了各种常见的网络协议。
- scripts：包含用于配置内核的脚本文件。
- security：主要包含 SELinux 模块。
- sound：ALSA、OSS 音频设备的驱动核心代码和常用设备驱动。
- usr：实现了用于打包和压缩的 cpio 等。

3.3.2 Linux 内核的组成部分

如图 3.1 所示，Linux 内核主要由进程调度（SCHED）、内存管理（MM）、虚拟文件系统（VFS）、网络接口（NET）和进程间通信（IPC）等 5 个子系统组成。

1．进程调度

精度调度控制系统中的多个进程对 CPU 的访问使得多个进程能在 CPU 中微观串行，宏观并行地执行。进程调度处于系统的中心位置，内核中其他的子系统都依赖它，因为每个子系统都需要挂起或恢复进程。

如图 3.2 所示，Linux 的进程在几个状态

间进行切换。在设备驱动编程中，当请求的资源不能得到满足时，驱动一般会调度其他进程执行，并使驱动对应的进程进入睡眠状态，直到它请求的资源被释放，才会被唤醒而进入就绪状态。睡眠分为可被打断的睡眠和不可被打断的睡眠，两者的区别在于可被打断的睡眠在收到信号的时候会醒来。

图 3.2　Linux 进程状态转换

在设备驱动编程中，当请求的资源不能得到满足时，驱动一般会调度其他进程执行，其对应的进程进入睡眠状态，直到它请求的资源被释放，才会被唤醒而进入就绪状态。

设备驱动中，如果需要几个并发执行的任务，可以启动内核线程，启动内核线程的函数为：

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags);
```

2．内存管理

内存管理的主要作用是控制多个进程安全地共享主内存区域。当 CPU 提供内存管理单元（MMU）时，Linux 内存管理完成为每个进程进行虚拟内存到物理内存的转换。Linux 2.6 引入了对无 MMU CPU 的支持。

如图 3.3 所示，一般而言，Linux 的每个进程享有 4GB 的内存空间，0～3GB 属于用户空间，3～4GB 属于内核空间，内核空间对常规内存、I/O 设备内存以及高端内存存在不同的处理方式。

图 3.3　Linux 进程地址空间

3．虚拟文件系统

如图 3.4 所示，Linux 虚拟文件系统（VFS）隐藏各种了硬件的具体细节，为所有的设备提供了统一的接口。而且，它独立于各个具体的文件系统，是对各种文件系统的一个抽象，它使用超级块 super block 存放文件系统相关信息，使用索引节点 inode 存放文件的物理信息，使用目录项 dentry 存放文件的逻辑信息。

图 3.4　Linux 文件系统

4．网络接口

网络接口提供了对各种网络的标准的存取和各种网络硬件的支持。网络接口可分为网络协议和网络驱动程序，网络协议部分负责实现每一种可能的网络传输协议，网络设备驱动程序负责与硬件设备进行通信，每一种可能的硬件设备都有相应的设备驱动程序。

5．进程通信

Linux 支持进程间的多种通信机制，包含信号量、共享内存、管道等，这些机制可协助多个进程、多资源的互斥访问、进程间的同步和消息传递。

Linux 内核的 5 个组成部分之间的依赖关系如下。

- 进程调度与内存管理之间的关系：这两个子系统互相依赖。在多道程序环境下，必须为程序创建进程，而创建进程的第一件事情就是将程序和数据装入内存。
- 进程间通信与内存管理的关系：进程间通信子系统要依赖内存管理支持共享内存通信机制，这种机制允许两个进程除了拥有自己的私有空间，还可以存

取共同的内存区域。

- 虚拟文件系统与网络接口之间的关系：虚拟文件系统利用网络接口支持网络文件系统（NFS），也利用内存管理支持 RAMDISK 设备。

图 3.5　Linux 网络体系结构

- 内存管理与虚拟文件系统之间的关系：内存管理利用虚拟文件系统支持交换，交换进程（swapd）定期由调度程序调度，这也是内存管理依赖于进程调度的惟一原因。当一个进程存取的内存映射被换出时，内存管理向文件系统发出请求，同时，挂起当前正在运行的进程。

除了这些依赖关系外，内核中的所有子系统还要依赖于一些共同的资源。这些资源包括所有子系统都用到的例程，如分配和释放内存空间的函数、打印警告或错误信息的函数及系统提供的调试例程等。

3.3.3　Linux 内核空间与用户空间

现代 CPU 内部往往实现了不同的操作模式（级别）。

例如，ARM 处理器有以下 7 种工作模式。

- 用户模式（usr）：大多数的应用程序运行在用户模式下，当处理器运行在用户模式下时，某些被保护的系统资源是不能被访问的。
- 快速中断模式（fiq）：用于高速数据传输或通道处理。
- 外部中断模式（irq）：用于通用的中断处理。
- 管理模式（svc）：操作系统使用的保护模式。
- 数据访问终止模式（abt）：当数据或指令预取终止时进入该模式，可用于虚拟存储及存储保护。
- 系统模式（sys）：运行具有特权的操作系统任务。
- 未定义指令中止模式（und）：当未定义的指令执行时进入该模式，可用于支持硬件协处理器的软件仿真。

又如，X86 处理器包含 4 个不同的特权级，称为 Ring0～Ring3。Ring0 下，可以执行特权级指令，对任何 I/O 设备都有访问权等，而 Ring3 则有很多操作限制。

Linux 系统充分利用 CPU 的这一硬件特性，但它只使用了两级。在 Linux 系统中，内核可进行任何操作，而应用程序则被禁止对硬件的直接访问和对内存的未授权访问。例如，若使用 X86 处理器，则用户代码运行在特权级 3，而系统内核代码则运行在特权级 0。

内核空间和用户空间这两个名词被用来区分程序执行的这两种不同状态，它们使用不同的地址空间。Linux 系统只能通过系统调用和硬件中断完成从用户空间到内核空间的控制转移。

3.4 Linux 内核的编译及加载

3.4.1 Linux 内核的编译

Linux 驱动工程师需要牢固地掌握 Linux 内核的编译方法以为嵌入式系统构建可运行的 Linux 操作系统映像。下面我们介绍在 VmWare 虚拟机上安装的含 Linux 2.4.20-8 内核的 Red Hat 中编译 Linux 2.6.15.5 内核并启动这个内核的过程，其他情况下的 Linux 2.6 内核编译与此类似。

（1）下载 Linux 2.6.15.5 内核源代码 linux-2.6.15.5.tar.bz2，放入/usr/src 目录。

（2）由于 Linux 2.6 内核的内核模块处理过程有所改变，因此 Linux 2.4 内核下的 modutils 工具包已经不再适合 Linux 2.6 内核，需要下载较新版本的 module-init-tools 和 modutils，笔者下载的是 module-init-tools-3.2.2.tar.bz2 和 modutils-2.4.5-1.src.rpm。

module-init-tools 下载地址如下：

http://www.kernel.org/pub/linux/utils/kernel/module-init-tools/

modutils 下载地址为：

http://www.kernel.org/pub/linux/utils/kernel/modutils/v2.4/

（3）安装 modutils，运行如下命令：

```
#rpm -e --nodeps modutils （强行卸载原有的 modutils RPM 包）
#rpm -ivh modutils-2.4.5-1.src.rpm （把源代码包安装到了/usr/src/redhat 目录下）
#cd /usr/src/redhat/SPECS （进入规范文件目录下）
#rpmbuild --bb modutils.spec （生成二进制的 RPM 包）
#cd ../RPMS/i386 （转入刚生成的二进制的 RPM 包所在位置）
#rpm -ivh modutils*.rpm （安装生成的 modutils-2.4.5-1.i386.rpm 和 modutils- debuginfo-2.4.5-1.i386.rpm 二进制 RPM 包）
```

（4）安装 module-init-tools，它会替代 depmod [/sbin/depmod]和其他工具，运行如下命令：

```
tar -jxvf module-init-tools-3.2.2.tar.bz2
cd module-init-tools-3.2.2
./configure --prefix=/sbin
make
make install
```

```
./generate-modprobe.conf /etc/modprobe.conf
```

（5）解压缩内核源代码，把下载的源代码包放到目录/usr/src 下，然后运行如下命令：

```
cd /usr/src
tar –jxvf linux-2.6.15.5.tar.bz2
ln –s linux-2.6.15.5 linux （将 Linux 符号连接映射到 linux-2.6.15.5）
cd linux
```

（6）执行 make mrproper 命令，该命令确保源代码目录下没有不正确的.o 文件以及文件的互相依赖。对于刚下载的完整的源程序包，本步可以省略。但是，如果这些源程序被多次使用过，那么在编译之前应运行此命令。

（7）配置内核，可以使用下面命令中的一个：

```
#make config（基于文本的最为传统的配置界面，不推荐使用）
#make menuconfig（基于文本菜单的配置界面）
#make xconfig（要求 QT 被安装）
#make gconfig（要求 GTK+被安装）
```

在配置 Linux 2.6 内核所使用的 make config、make menuconfig、make xconfig 和 make gconfig 这 4 种方式中，最值得推荐的是 make menuconfig，它不依赖于 QT 或 GTK+，且非常直观，对 Linux 2.6.15.5 内核运行 make menuconfig 后的界面如图 3.6 所示。

图 3.6 Linux 内核编译配置

内核配置包含的项目相当之多，下面介绍几个常用的配置选项。

- Code maturity level options（代码成熟等级）。

此处只有一项“prompt for development and/or incomplete code/drivers”，如果要使用目前仍处于实验阶段的功能，选择该项为“Y”，否则选择为“N”。

- Loadable module support（内核模块支持）。

Linux 2.6.15.5 的该项设置包括 5 个子项目，“Enable loadable module support”、“Module versioning support”、“Automatic Kernel module loading”3 项被默认选中。

- General setup （大众化的设置）。

包含“System V IPC”、“Sysctl support”、“BSD Process Accounting”等通用设置。相对 Linux 2.4 的这个项目而言，包含的子项目大大缩小了，PCI、PCMCIS、ISA、EISA 和 MCA 进入了单独的“Bus options”项目，“Networking support”进入“Networking”项目。

- File system（文件系统）。

包含对 Ext2、Ext3、FAT、NFS 等多种文件系统的设置选项。

- Processor type and features（处理器类型和特征）。

设置目标平台的处理器类型及其特征。

- Device drivers（设备驱动）。

按照 char、MTD、ATA、SCSI、I^2C、SPI、USB、MMC/SD、sound、IEEE 1394（FireWire）、IrDA 等设备设立了多个子项目，从“Device drivers”这个项目的展开情况可以看出，Linux 2.6 支持目前大多数的设备类型。

在 VmWare 中使用了模拟的 SCSI 硬盘和 AMD PCnet32 网卡，因此在内核中应该增加对“SCSI disk support”、“BusLogic SCSI support”和“AMD PCnet32 PCI support”的支持，最好直接编译入内核而不是编译为内核模块。

（8）编译内核，并将映像复制到相应的目录：

```
make bzImage （生成使用 gzip 压缩的内核，生成的文件位于/usr/src/linux/arch/i386/boot 目录）
make modules modules_install （编译内核模块并安装到/lib/modules/2.6.15.5 目录）
cd /usr/src/linux
mv arch/i386/boot/bzImage /boot/bzImage2.6.15.5
mv System.map /boot/System.map-2.6.15.5
cd /boot
mv System.map oldSystem.map
ln -s System.map-2.6.15.5 System.map （创建相应的 System.map 符号链接）
```

System.map-2.6.15.5 是编译生成的 Linux 2.6.15.5 内核的符号表，而 System.map 是内核符号表的链接。虽然内核本身并不真正使用 System.map，但其他程序比如 klogd 和 ps 等软件需要一个正确的 System.map。如果使用错误的 System.map 或没有 System.map，klogd 的输出将是不可靠的，这对排除程序故障带来困难。一个典型的 System.map 记录的符号信息如下：

```
00100000 A phys_startup_32
c0100000 T startup_32
c0100000 A _text
c01000c6 t checkCPUtype
c0100147 t is486
c010014e t is386
c0100199 t L6
c010019b t check_x87
c01001c2 t setup_idt
```

```
c01001df t rp_sidt
c01001ec t ignore_int
c0100220 t rest_init
c0100220 T stext
c0100220 T _stext
c010023e t do_pre_smp_initcalls
c0100248 t run_init_process
c0100274 t init
c0100394 t try_name
c0100522 T name_to_dev_t
c01007a0 t calibrate_delay_direct
…
```

（9）针对 2.6 内核和 2.4 内核的差异修改脚本和配置文件。

Linux 2.6 内核增加了 sysfs 文件系统，我们首先要创建/sys 目录，使用“#mkdir /sys”完成。然后按照下列步骤修改启动脚本文件/etc/rc.sysinit。

① 找到行“mount -f /proc”，在其下面增加如下内容：

```
if [ "$KERNELVER" = "2.6" ]; then
mount -f /sys
fi
```

② 找到行“action $"Mounting proc filesystem: " mount -n -t proc /proc /proc”， 在其下面增加如下内容：

```
#
# Mount /sys for kernel-2.6.x
#
if [ "$KERNELVER" = "2.6" ]; then
action $"Mounting sysfs filesystem: " mount -n -t sysfs /sys /sys
fi
```

③ 修改/etc/fstab 文件系统列表，在其中增加如下内容：

```
none /sys sysfs defaults 0 0
```

④ 修改/etc/init.d/halt 脚本中的 halt_get_remaining 函数，将

```
awk '$2 ~ /^\/$|^\/proc|^\/dev/{next}
```

改为

```
awk '$2 ~ /^\/$|^\/proc|^\/sys|^\/dev/{next}
```

由于内核符号文件/proc/ksyms 被改名为/proc/kallsyms，所以必需修改启动脚本/etc/rc.sysinit，为了兼容原有的 Linux 2.4 系统，可以加上下面几行：

```
#
# ----- KERNEL 2.6.x support ------
# This is for compatibility between kernel-2.4.x and kernel-2.6.x
#
UNAME='uname -r'
KERNELVER=${UNAME:0:3}
if [ "$KERNELVER" = "2.6" ]; then
```

```
    #
    # This is kernel-2.6.x
    #
    KSYMS=/proc/kallsyms
    else
    #
    # This is kernel-2.4.x
    #
    KSYMS=/proc/ksyms
    fi
```

然后，把 rc.sysinit 文件中所有出现/proc/ksyms 的地方都换成“$KSYMS”变量引用。

Linux 2.6 的 USB 驱动模块被改名了，在主机控制器方面，usb-ohci 被改为 ochi-hcd，usb-uhci 被改为 uhci-hcd。因此，需在/etc/modprobe.conf 配置文件中进行响应修改。

此外，支持 USB 键盘的模块名称也从 keybdev 改为 usbkbd，支持 USB 鼠标的模块名称也从 mousedev 改为 usbmouse，因此，必需修改启动脚本文件/etc/rc.sysinit。为了兼容原有的 Linux 2.4 系统，在脚本的一开始定义两个变量 KEYBDEV_NAME 和 MOUSEDEV_NAME，如下所示：

```
    #
    # ----- KERNEL 2.6.x support ------
    # This is for compatibility between kernel-2.4.x and kernel-2.6.x
    #
    UNAME='uname -r'
    KERNELVER=${UNAME:0:3}
    if [ "$KERNELVER" = "2.6" ]; then
    #
    # This is kernel-2.6.x
    #
    KSYMS=/proc/kallsyms
    KEYBDEV_NAME=usbkbd
    MOUSEDEV_NAME=usbmouse
    else
    #
    # This is kernel-2.4.x
    #
    KSYMS=/proc/ksyms
    KEYBDEV_NAME=keybdev
    MOUSEDEV_NAME=mousedev
    fi
```

然后，把/etc/rc.sysinit 脚本文件中出现 keybdev 和 mousedev 的地方都改为$KEYBDEV_NAME 和$MOUSEDEV_NAME，接着，把脚本文件/etc/rc.sysinit 中的以下内容进行修改：

```
    needusbstorage=
    if [ $usb = "1" ]; then
    needusbstorage='LC_ALL=C grep -e "^I.*Cls=08" /proc/bus/usb/devices
2>/dev/null'
    LC_ALL=C grep 'hid' /proc/bus/usb/drivers || action $"Initializing USB
HID interface: " modprobe hid 2> /dev/null

    action $"Initializing USB keyboard: " modprobe $KEYBDEV_NAME 2>
/dev/null
    action $"Initializing USB mouse: " modprobe $MOUSEDEV_NAME 2> /dev/null
    fi
```

改为：

```
    needusbstorage=
    if [ $usb = "1" ]; then
    if [ "$KERNELVER" = "2.6" ]; then
    needusbstorage='LC_ALL=C grep -e "^I.*Cls=08" /sys/bus/usb/devices
2>/dev/null'
    LC_ALL=C grep 'hid' /sys/bus/usb/drivers || action $"Initializing USB
HID interface: " modprobe hid 2> /dev/null
    else
    needusbstorage='LC_ALL=C grep -e "^I.*Cls=08" /proc/bus/usb/devices
2>/dev/null'
    LC_ALL=C grep 'hid' /proc/bus/usb/drivers || action $"Initializing USB
HID interface: " modprobe hid 2> /dev/null
    fi
    action $"Initializing USB keyboard: " modprobe $KEYBDEV_NAME 2>
/dev/null
    action $"Initializing USB mouse: " modprobe $MOUSEDEV_NAME 2> /dev/null
    fi
```

（10）修改启动选项，以支持从 Linux 2.4.20-8 和 Linux 2.6.15.5 选择菜单。

修改/boot/grub/grub.conf 文件，进行如下设置：

```
    #boot=/dev/sda
    default=1
    timeout=10
    splashimage=(hd0,0)/boot/grub/splash.xpm.gz
    title Red Hat Linux (2.4.20-8)
          root (hd0,0)
          kernel /boot/vmlinuz-2.4.20-8 ro root=LABEL=/ hdc=ide-scsi
          initrd /boot/initrd-2.4.20-8.img
    title Red Hat Linux (2.6.15.5)
          root (hd0,0)
          kernel /boot/bzImage2.6.15.5 ro root=/dev/sda1 hdc=ide-scsi
```

经过此项设置后，Linux 启动时会出现选择菜单，如图 3.7 所示。

图 3.7　Linux 启动菜单

在上面的 Linux 2.6.15.5 启动中，没有设置 initrd，这是因为我们直接把 SCSI、EXT3 等编译进了内核。如果将它们编译为模块，initrd 则是必须的，否则启动过程中会提示“VFS: unable to mounting root fs”。为使用 initrd，则需先下载较新版的 mkinitrd 并升级安装。由于程

序包 mkinitrd 依赖于程序包 device-mapper，而程序包 device-mapper 又依赖于程序包 lvm2，因此还要同时下载程序包 device-mapper 和程序包 lvm2。

initrd（Bootloader initialized RAM disk）是指由 Bootloader 初始化的内存盘。在 Linux 内核启动前，Bootloader 会将存储介质中的 initrd 文件加载到内存，内核启动时会在访问真正的根文件系统前先访问该内存中的 initrd 文件系统。为使得启动时加载 initrd，只需修改 grub.conf 引导配置文件，在最后添加“initrd /boot/initrd-2.6.15.5.img”。

在 Bootloader 配置了 initrd 的情况下，内核启动被分成了两个阶段，第一阶段先执行 initrd 文件系统中的文件，完成加载驱动模块等任务，第二阶段才会执行真正的根文件系统中的/sbin/init 进程。第一阶段启动的目的是为第二阶段的启动扫清障碍，最主要的是加载根文件系统存储介质的驱动模块。

由于 Linux 2.6.15.5 直接提供了对 S3C2410 评估板 SMDK2410 的支持，因此为它编译 Linux 2.6.15.5 非常快捷。首先运行以下命令：

```
make ARCH=arm CROSS_COMPILE=arm-linux-menuconfig   （指定体系结构为 ARM，交叉编译器为 arm-linux-gcc）
```

在弹出的菜单中直接选择“Load an Alternate Configuration File”，把默认配置文件路径输入即可，如图 3.8 所示。

以上两步也可以合并为一步，即运行：

```
make ARCH=arm CROSS_COMPILE=arm-linux- smdk2410_defconfig（指定体系结构为 ARM，交叉编译器为 arm-linux-gcc，采用 smdk_2410defconfig 配置）
```

图 3.8　使用默认配置文件为 SMDK2410 编译 Linux 2.6 内核

配置完 SMDK2410 电路板后，运行如下命令可编译内核：

```
make ARCH=arm CROSS_COMPILE=arm-linux- zImage
```

运行如下命令编译模块：

```
make ARCH=arm CROSS_COMPILE=arm-linux- modules
```

当然，如果在源代码根目录的 Makefile 中将 ARCH 和 CROSS_COMPILE 直接指定为 arm 和 arm-linux-后，就没有必要每次编译的时候都指定体系结构和交叉编译器了。

Linux 2.6 内核的配置系统由以下 3 个部分组成。

- Makefile：分布在 Linux 内核源代码中的 Makefile，定义 Linux 内核的编译规则。
- 配置文件（Kconfig）：给用户提供配置选择的功能。
- 配置工具：包括配置命令解释器（对配置脚本中使用的配置命令进行解释）和配置用户界面（提供字符界面和图形界面）。这些配置工具都是使用脚本语言编写的，如 Tcl/TK、Perl 等。

使用 make config、make menuconfig 等命令后，会生成一个 config 配置文件（隐含在顶层 Makefile 中），记录哪些部分被编译入内核、哪些部分被编译为内核模块。

运行 make menuconfig 时，配置工具首先分析与体系结构对应的/arch/xxx/Kconfig 文件（xxx 即为传入的 ARCH 参数），/arch/xxx/Kconfig 文件中除本身包含一些与体系结构相关的配置项和配置菜单以外，还通过 source 语句引入了一系列 Kconfig 文件，而这些 Kconfig 文件又可能再次通过 source 引入下一层的 Kconfig，配置工具依据这些 Kconfig 包含的菜单和项目即可描绘出一个如图 3.6 所示的分层结构。例如，/arch/arm/Kconfig 文件的结构如下：

```
    mainmenu "Linux Kernel Configuration"

    config ARM
         bool
         default y
         help
           The ARM series is a line of low-power-consumption RISC chip
designs
           licensed by ARM Ltd and targeted at embedded applications and
           handhelds such as the Compaq IPAQ.  ARM-based PCs are no longer
           manufactured, but legacy ARM-based PC hardware remains popular
in
           Europe.  There is an ARM Linux project with a web page at
           <http://www.arm.linux.org.uk/>.

    config MMU
         bool
         default y

    ...
    config ARCH_S3C2410
         bool "Samsung S3C2410"
         help
           Samsung S3C2410X CPU based systems, such as the Simtec Electronics
           BAST (<http://www.simtec.co.uk/products/EB110ITX/>), the IPAQ
1940 or
           the Samsung SMDK2410 development board (and derivatives).
    ...

    source "mm/Kconfig"
    ...
    source "net/Kconfig"
    ...
    source "arch/arm/mach-s3c2410/Kconfig"
    ...
```

用户运行 make zImage、make bzImage 等生成映像的命令时，先检索顶层 Makefile（在 arch/xxx 目录下的 Makefile 为顶层 Makefile 补充体系结构相关的信息），顶层 Makefile 完成两个主要的任务：产生内核映像文件和内核模块。为了达到此目的，顶层 Makefile 会递归地进入内核的各个子目录中，分别调用位于这些子目录中 Makefile（子目录中的 Makefile 属于 kbuild 类型的 Makefile，记录编译目标）。进入哪些子目录则取决于内核的配置，这也就是 make config 和 make menuconfig 等生成的.config 文件发挥作用的地方。

3.4.2 Kconfig 和 Makefile

在 Linux 内核中增加程序需要完成以下 3 项工作。

- 将编写的源代码复制到 Linux 内核源代码的相应目录。
- 在目录的 Kconfig 文件中增加新源代码对应项目的编译配置选项。
- 在目录的 Makefile 文件中增加对新源代码的编译条目。

1. 实例引导：S3C2410 处理器的 RTC 与 LED 驱动配置

在讲解 Kconfig 和 Makefile 的语法之前，我们先利用两个简单的实例引导读者建立初步的认识。

首先，在 linux-2.6.15.5\drivers\char 目录中包含了 S3C2410 处理器的 RTC 设备驱动源代码 s3c2410-rtc.c。

而在该目录的 Kconfig 文件中包含 S3C2410_RTC 的配置项目：

```
config S3C2410_RTC
    bool "S3C2410 RTC Driver"
    depends on ARCH_S3C2410
    help
        RTC (Realtime Clock) driver for the clock inbuilt into the
        Samsung S3C2410. This can provide periodic interrupt rates
        from 1Hz to 64Hz for user programs, and wakeup from Alarm.
```

上述 Kconfig 文件的这段脚本意味着只有在 ARCH_S3C2410 项目被配置的情况下，才会出现 S3C2410_RTC 配置项目，这个配置项目为布尔型（要么编译入内核，要么不编译，选项为“Y”或“N”），菜单上显示的字符串为“S3C2410 RTC Driver”，“help”后面的内容为帮助信息。图 3.9 所示为 S3C2410_RTC 菜单以及其 help 在运行 make menuconfig 时的情况。

```
Linux Kernel v2.6.15.5 Configuration
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
 lqqqqqqqqqqqqqqqqqqqqqqqqqqq Character devices qqqqqqqqqqqqqqqqqqqqqqqqqqqqqk
 x  Arrow keys navigate the menu.  <Enter> selects submenus --->.          x
 x  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, x
 x  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> x
 x  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >       x
 x lqqqqqqq^(-)qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqk x
 x x           Serial drivers  --->                                       x x
 x x       [*] Legacy (BSD) PTY support                                   x x
 x x       (256) Maximum number of legacy PTY in use                      x x
 x x           IPMI  --->                                                 x x
 x x           Watchdog Cards  --->                                       x x
 x x       [ ] /dev/nvram support                                         x x
 x x       [ ] Enhanced Real Time Clock Support                           x x
 x x       [*] S3C2410 RTC Driver                                         x x
 x x       [ ] Double Talk PC internal speech card support                x x
 x x       [ ] Siemens R3964 line discipline                              x x
 x mqqqqqqqv(+)qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj x
 tqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqu
 x                    <Select>    < Exit >    < Help >                    x
 mqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj
```

```
Linux Kernel v2.6.15.5 Configuration
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
 lqqqqqqqqqqqqqqqqqqqqqqqqqqq S3C2410 RTC Driver qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqk
 x CONFIG_S3C2410_RTC:                                                          x
 x                                                                              x
 x RTC (Realtime Clock) driver for the clock inbuilt into the                   x
 x Samsung S3C2410. This can provide periodic interrupt rates                   x
 x from 1Hz to 64Hz for user programs, and wakeup from Alarm.                   x
 x                                                                              x
 x Symbol: S3C2410_RTC [=y]                                                     x
 x Prompt: S3C2410 RTC Driver                                                   x
 x   Defined at drivers/char/Kconfig:774                                        x
 x   Depends on: ARCH_S3C2410                                                   x
 x   Location:                                                                  x
 x     -> Device Drivers                                                        x
 x       -> Character devices                                                   x
 x                                                                              x
 x                                                                              x
 x                                                                              x
 tqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqu
 x                              < Exit >                                        x
 mqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj
```

图 3.9　Kconfig 菜单项目与帮助信息

除了布尔型的配置项目外，还存在一种三态型（tristate）配置选项，它意味着要么编译入内核，要么编译为内核模块，要么不编译，选项为“Y”、“M”或“N”。

在目录的 Makefile 中关于 S3C2410_RTC 的编译脚本为：

```
obj-$(CONFIG_S3C2410_RTC) += s3c2410-rtc.o
```

上述脚本意味着如果 S3C2410_RTC 配置选项被选择为“Y”或“M”，即 obj-$(CONFIG_S3C2410_RTC)等同于 obj-y 或 obj-m 时，则编译 s3c2410-rtc.c，选“Y”的情况直接会将生成的目标代码直接连接到内核，为“M”的情况则会生成模块 s3c2410-rtc.ko（由于 S3C2410_RTC 为布尔型，实际不会为“M”）；如果 S3C2410_RTC 配置选项被选择为“N”，即 obj-$(CONFIG_S3C2410_RTC)等同于 obj-n 时，则不编译 s3c2410-rtc.c。

一般而言，驱动工程师在内核源代码的 drivers 目录的相应子目录中增加新设备驱动的源代码，并增加或修改 Kconfig 配置脚本和 Makefile 脚本，完全仿照上述过程执行即可。

再如，我们为 S3C2410 的 LED 编写了驱动，源代码为 s3c2410-led.c，为使内核能支持对该模块的编译配置，我们应进行如下 3 项处理。

- 将编写的 s3c2410-led.c 源代码复制到 linux-2.6.15.5\drivers\char 目录。
- 在目录的 Kconfig 文件中增加 LED 的编译配置选项，如下所示：

```
config S3C2410_LED
    bool "S3C2410 LED Driver"
    depends on ARCH_S3C2410
    help
        LED driver for the Samsung S3C2410.
```

- 在目录的 Makefile 文件中增加对 s3c2410-led.c 源代码的编译，如下所示：

```
obj-$(CONFIG_S3C2410_LED) += s3c2410-led.o
```

2. Makefile

下面我们对内核源代码各级子目录中的 kbuild Makefile 进行介绍，这部分是内核模块或设备驱动的开发者最常接触到的。

kbuild Makefile 的语法包括如下几个方面。

（1）目标定义。

目标定义用来定义哪些内容要作为模块编译，哪些要编译并连接进内核。

例如：

```
obj-y += foo.o
```

表示要由 foo.c 或者 foo.s 文件编译得到 foo.o 并连接进内核，而 obj-m 则表示该文件要作为模块编译。除了 y、m 以外的 obj-x 形式的目标都不会被编译。

而更常见的做法是根据.config 文件的 CONFIG_变量来决定文件的编译方式，如下所示：

```
obj-$(CONFIG_ISDN) += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

除了 obj-形式的目标以外，还有 lib-y library 库、hostprogs-y 主机程序等目标，但是基本都应用在特定的目录和场合下。

（2）多文件模块的定义。

如果一个模块由多个文件组成，这时候应采用模块名加-objs 后缀或者-y 后缀的形式来定义模块的组成文件。如下面的例子所示：

```
obj-$(CONFIG_EXT2_FS) += ext2.o
ext2-y := balloc.o bitmap.o
ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o
```

模块的名字为 ext2，由 balloc.o 和 bitmap.o 两个目标文件最终连接生成 ext2.o 直至 ext2.ko 文件，是否包括 xattr.o 取决于内核配置文件的配置情况。如果 CONFIG_EXT2_FS 的值是 y 也没有关系，在此过程中生成的 ext2.o 将被连接进 built-in.o 最终连接进内核。这里需要注意的一点是，该 kbuild Makefile 所在的目录中不能再包含和模块名相同的源文件如 ext2.c/ext2.s。

或者写成如-objs 的形式：

```
obj-$(CONFIG_ISDN) += isdn.o
isdn-objs := isdn_net_lib.o isdn_v110.o isdn_common.o
```

（3）目录层次的迭代。

示例：

```
obj-$(CONFIG_EXT2_FS) += ext2/
```

当 CONFIG_EXT2_FS 的值为 y 或 m 时，kbuild 将会把 ext2 目录列入向下迭代的目标中，具体 ext2 目录下的文件是要作为模块编译还是链入内核由 ext2 目录下的 Makefile 文件的内容决定。

3. Kconfig

内核配置脚本文件的语法也比较简单，主要包括如下几个方面。

（1）菜单入口。

大多数的内核配置选项都对应 Kconfig 中的一个菜单入口，如下所示：

```
config MODVERSIONS
    bool "Set version information on all module symbols"
    depends on MODULES
    help
      Usually, modules have to be recompiled whenever you switch to a
```

```
new
          kernel.  ...
```

“config”关键字定义新的配置选项，之后的几行定义了该配置选项的属性。配置选项的属性包括类型、数据范围、输入提示、依赖关系（及反向依赖关系）、帮助信息和默认值等。

每个配置选项都必须指定类型，类型包括 bool、tristate、string、hex 和 int，其中 tristate 和 string 是两种基本的类型，其他类型都基于这两种基本类型。类型定义后可以紧跟输入提示，下面的两段脚本是等价的。

脚本 1：

```
bool "Networking support"
```

脚本 2：

```
bool
prompt "Networking support"
```

输入提示的一般格式如下所示：

```
prompt <prompt> [if <expr>]
```

其中可选的 if 用来表示该提示的依赖关系。

默认值的格式如下所示：

```
default <expr> [if <expr>]
```

一个配置选项可以存在任意多个默认值，这种情况下，只有第一个被定义的值是可用的。如果用户不设置对应的选项，配置选项的值就是默认值。

依赖关系的格式如下所示：

```
depends on（或者 requires） <expr>
```

如果定义了多个依赖关系，它们之间用“&&”间隔。依赖关系也可以应用到该菜单中所有的其他选项中（这些选项同样可接受 if 表达式），下面的两段脚本是等价的。

脚本 1：

```
bool "foo" if BAR
default y if BAR
```

脚本 2：

```
depends on BAR
bool "foo"
default y
```

反向依赖关系的格式如下所示：

```
select <symbol> [if <expr>]
```

depends 能限定一个 symbol 的上限，即如果 A 依赖于 B，则在 B 被配置为“Y”的情况下，A 可以为“Y”、“M”和“N”；在 B 被配置为“M”的情况下，A 可以被配置为“M”或“N”；B 在被配置为“N”的情况下，A 只能为“N”。

select 能限定一个 symbol 的下限，若 A 反向依赖于 B，则 A 的配置值会高于或等于 B（正好与 depends 相反）。如果 symbol 反向依赖于多个对象，则它的下限是这些对象的最大值。

kbuild Makefile 中的 expr（表达式）定义如下所示：

```
<expr> ::= <symbol>
                <symbol> '=' <symbol>
                <symbol> '!=' <symbol>
               '(' <expr> ')'
               '!' <expr>
               <expr> '&&' <expr>
               <expr> '||' <expr>
```

也就是说 expr 是由 symbol、两个 symbol 相等、两个 symbol 不等以及 expr 的赋值、非、与或运算构成。而 symbol 分为两类，一类是由菜单入口定义配置选项定义的非常数 symbol，另一类是作为 expr 组成部分的常数 symbol。

数据范围的格式如下：

```
range <symbol> <symbol> [if <expr>]
```

为 int 和 hex 类型的选项设置可以接受的输入值范围，用户只能输入大于等于第一个 symbol，小于等于第二个 symbol 的值。

帮助信息的格式如下：

```
help（或---help---）
  开始
  …
  结束
```

帮助信息完全靠文本缩进识别结束。“---help---”和“help”在作用上没有区别，设计“---help---”的初衷在于将文件中的配置逻辑与给开发人员的提示分开。

menuconfig 关键字的作用与 config 类似，但它在 config 的基础上要求所有的子选项作为独立的行显示。

（2）菜单结构。

菜单入口在菜单树结构中的位置可由两种方法决定。第一种方式如下所示：

```
menu "Network device support"
    depends on NET
config NETDEVICES
    …
endmenu
```

所有处于“menu”和“endmenu”之间的菜单入口都会成为“Network device support”的子菜单。而且，所有子菜单选项都会继承父菜单的依赖关系，比如，“Network device support”对“NET”的依赖被加到了配置选项 NETDEVICES 的依赖列表中。

另一种方式是通过分析依赖关系生成菜单结构。如果菜单选项在一定程度上依赖于前面的选项，它就能成为该选项的子菜单。如果父选项为“N”，则子选项不可见；如果父选项为“Y”或“M”，则子选项可见。例如：

```
config MODULES
    bool "Enable loadable module support"

config MODVERSIONS
    bool "Set version information on all module symbols"
    depends on MODULES

comment "module support disabled"
    depends on !MODULES
```

MODVERSIONS 直接依赖 MODULES，如果 MODULES 不为“N”，该选项才可见。

除此之外，Kconfig 中还可能使用“choices … endchoice”、“comment”、“if…endif”这样的语法结构。其中“choices … endchoice”的结构如下所示：

```
choice
<choice options>
<choice block>
endchoice
```

它定义一个选择群，其接受的选项（choice options）可以是前面描述的任何属性。在一个硬件有多个驱动的情况下使用，使用选择可以实现最终只有一个驱动被编译进内核或模块。选择群还可以接受的另一个选项是“optional”，这样菜单入口就被设置为“N”，没有被选中。

4．应用实例：在内核中新增驱动代码目录和子目录

下面讲解一个综合实例，假设我们要在内核源代码 drivers 目录下为 ARM 体系结构新增如下用于 test driver 的树型目录：

```
|--test
    |-- cpu
        | -- cpu.c
    |-- test.c
    |-- test_client.c
    |-- test_ioctl.c
    |-- test_proc.c
    |-- test_queue.c
```

在内核中增加目录和子目录，我们需为相应的新增目录创建 Kconfig 和 Makefile 文件，而新增目录的父目录中的 Kconfig 和 Makefile 文件也需要修改，以便新增的 Kconfig 和 Makefile 文件能被引用。

在新增的 test 目录下，应该包含如下 Kconfig 文件：

```
#
# TEST driver configuration
#
menu "TEST Driver "
comment " TEST Driver"


config CONFIG_TEST
    bool "TEST support "


config CONFIG_TEST_USER
    tristate "TEST user-space interface"
    depends on CONFIG_TEST

endmenu
```

由于 TEST driver 对于内核来说是新的功能，所以首先需要创建一个菜单 TEST Driver；然后显示“TEST support”，等待用户选择；接下来判断用户是否选择了 TEST Driver，如果是（CONFIG_TEST=y），则进一步显示子功能：用户接口与 CPU 功能支持；由于用户接口功能可以被编译成内核模块，所以这里的询问语句使用了 tristate。

为了使这个 Kconfig 文件能起作用，需要修改 arch/arm/Kconfig 文件，增加以下内

容：

```
source "drivers/test/Kconfig"
```

脚本中的 source 意味着引用新的 Kconfig 文件。

在新增的 test 目录下，应该包含如下 Makefile 文件：

```
# drivers/test/Makefile
#
# Makefile for the TEST.
#
obj-$(CONFIG_TEST) += test.o test_queue.o test_client.o
obj-$(CONFIG_TEST_USER) += test_ioctl.o
obj-$(CONFIG_PROC_FS) += test_proc.o

obj-$(CONFIG_TEST_CPU) += cpu/
```

该脚本根据配置变量的取值构建 obj-*列表。由于 test 目录中包含一个子目录 cpu，当 CONFIG_ TEST_CPU=y 时，需要将 cpu 目录加入列表。

test 目录中的 cpu 子目录也需包含如下的 Makefile 文件：

```
# drivers/test/test/Makefile
#
# Makefile for the TEST CPU
#
obj-$(CONFIG_TEST_CPU) += cpu.o
```

为了使得整个 test 目录能够被编译命令作用到，test 目录父目录中的 Makefile 文件也需新增如下脚本：

```
obj-$(CONFIG_TEST) += test/
```

在 drivers/Makefile 中加入 obj-$(CONFIG_TEST) += test/，使得用户在进行内核编译时能够进入 test 目录。

增加了 Kconfig 和 Makefile 文件之后的新的 test 树型目录如下所示：

```
|--test
    |-- cpu
        | -- cpu.c
    | -- Makefile
|-- test.c
|-- test_client.c
|-- test_ioctl.c
|-- test_proc.c
|-- test_queue.c
|-- Makefile
|-- Kconfig
```

3.4.3 Linux 内核的引导

1. 引导过程概述

引导 Linux 内核的过程包括很多阶段，这里将以引导 X86 PC 为例来进行讲解。引导 X86 PC 上的 Linux 内核的过程和引导嵌入式系统上的 Linux 内核的过程基本类似。不过在 X86 PC 上有一个从 BIOS（基本输入/输出系统）转移到 Bootloader 的过程，如图 3.10 所示，而嵌入式系统往往复位后就直接运行 Bootloader。

图 3.10　X86 PC 上的 Linux 内核引导流程

图 3.10 给出了在 X86 PC 上从上电/复位到运行 Linux 用户空间初始进程的流程。在进入与 Linux 相关代码之前，会经历如下阶段。

① 当系统上电或复位时，CPU 会将 PC 指针赋值为一个特定的地址 0xFFFF0，并执行该地址处的指令。在 PC 中，该地址位于 BIOS 中，它保存在主板上的 ROM 或 Flash 中。

② BIOS 运行时按照 CMOS 的设置定义的启动设备顺序来搜索处于活动状态，并且可以引导的设备。若从硬盘启动，BIOS 会将硬盘 MBR（主引导记录）中的内容加载到 RAM。MBR 是一个 512 字节大小的扇区，位于磁盘上的第一个扇区中（0 道 0 柱面 1 扇区）。当 MBR 被加载到 RAM 中之后，BIOS 就会将控制权交给 MBR。

③ 主引导加载程序查找并加载次引导加载程序。它在分区表中查找活动分区，当找到一个活动分区时，扫描分区表中的其他分区，以确保它们都不是活动的。当这个过程验证完成之后，就将活动分区的引导记录从这个设备中读入 RAM 中并执行它。

④ 次引导加载程序加载 Linux 内核和可选的初始 RAM 磁盘，将控制权交给 Linux 内核源代码。

⑤ 运行被加载的内核，并启动用户空间应用程序。

2．Bootloader

嵌入式系统中 Linux 内核的引导过程与之类似，但一般更加简洁。不论具体以怎样的方式实现，只要具备如下特征就可以称其为 Bootloader。

- 可以在系统上电或复位的时候以某种方式执行，这些方式包括被 BIOS 引导执行、直接在 NOR Flash 中执行、NAND Flash 中的代码被 MCU 自动复制到内部或外部 RAM 执行等。
- 能将 U 盘、磁盘、光盘、NOR/NAND Flash、ROM、SD 卡等存储介质，或将网口、串口中的操作系统加载到 RAM，并把控制权交给操作系统源代码执行。

完成上述功能的 Bootloader 的实现方式非常多样化，甚至本身也可以是一个简化版的操作系统。著名的 Linux Bootloader 包括应用于 PC 的 LILO 和 GRUB，应用于嵌入式系统的 U-Boot、RedBoot 等。

相比较于 LILO，GRUB 本身能理解 Ext2、Ext3 文件系统，因此可在文件系统中加载 Linux 系统，而 LILO 只能识别“裸扇区”。

U-Boot 的定位为“Universal Bootloader”，其功能比较强大，涵盖了包括 PowerPC、ARM、MIPS 和 X86 在内的绝大部分处理器构架，提供网卡、串口、Flash 等外设驱动，提供必要的网络协议（BOOTP、DHCP、TFTP），能识别多种文件系统（cramfs、fat、jffs2 和 registerfs 等），并附带了调试、脚本、引导等工具，应用十分广泛。

Redboot 是 Redhat 公司随 eCos 发布的 Bootloader 开源项目，除了包含 U-Boot 类似的强大功能外，它还包含 GDB stub（插桩），因此能通过串口或网口与 GDB 进行通信，调试 GCC 产生的任何程序（包括内核）。

3．详细分析

下面对上述流程的第 5 个阶段进行更详细的分析，它完成启动内核并运行用户空间的 init 进程的功能。

当内核映像被加载到 RAM 之后，Bootloader 的控制权被释放。内核映像并不是可直接执行的目标代码，而是一个压缩过的 zImage（小内核）或 bzImage（大内核，bzImage 中的“b”是“big”意思）。

但是，并非 zImage 和 bzImage 映像中的一切都被压缩了，映像中包含未被压缩的部分，这部分中包含解压缩程序，解压缩程序会解压缩映像中被压缩的部分。zImage 和 bzImage 都是用 gzip 压缩的，它们不仅是一个压缩文件，而且在这两个文件的开头部分内嵌有 gzip 解压缩代码。

如图 3.11 所示，当 bzImage（用于 i386 映像）被调用时，它从/arch/i386/boot/head.S 的 start 汇编例程开始执行。这个例程子进行一些基本的硬件设置，并调用/arch/i386/boot/compressed/head.S 中的 startup_32 例程。startup_32 例程设置一个基本的运行环境（如堆栈）后清除 BSS 段，调用/arch/i386/boot/compressed/misc.c 中的 decompress_kernel()解压缩内核。

图 3.11　X86 PC 上的 Linux 内核初始化

内核被解压缩到内存中之后会再调用/arch/i386/kernel/head.S 文件中的 startup_32

例程，这个新的 startup_32 例程（称为清除程序或进程 0）会初始化页表，并启用内存分页机制，接着为任何可选的浮点单元（FPU）检测 CPU 的类型，并将其存储起来供以后使用。

这些都做完之后，/init/main.c 中的 start_kernel()函数被调用，进入与体系结构无关的 Linux 内核部分。

start_kernel()会调用一系列初始化函数来设置中断，执行进一步的内存配置。之后，/arch/i386/kernel/process.c 中 kernel_thread()被调用以启动第一个核心线程，该线程执行 init() 函数，而原执行序列会调用 cpu_idle()，等待调度。

作为核心线程的 init()函数完成外设及其驱动程序的加载和初始化，挂接根文件系统。init()打开/dev/console 设备，重定向 stdin、stdout 和 stderr 到控制台。之后，它搜索文件系统中的 init 程序（也可以由“init=”命令行参数指定 init 程序），并使用 execve()系统调用执行 init 程序。搜索 init 程序的顺序为/sbin/init、/etc/init、/bin/init 和/bin/sh。在嵌入式系统中，多数情况下，可以给内核传入一个简单的 shell 脚本来启动必需的嵌入式应用程序。

至此，漫长的 Linux 内核引导和启动过程就结束了，而 init()对应的由 start_kernel()创建的第一个线程也进入用户模式。

3.5 Linux 下的 C 编程

3.5.1 Linux 程序命名习惯

Linux 程序的命名习惯和 Windows 程序的命名习惯及著名的匈牙利命名法有很大的不同。

在 Windows 程序中，习惯以如下方式命名宏、变量和函数：

```
#define PI 3.1415926            /*用大写字母代表宏*/
int minValue, maxValue;         /*变量：第一个单词全写，其后的单词第一个字母小写*/
void SendData(void);            /*函数：所有单词第一个字母都大写*/
```

这种命名方式在程序员中非常盛行，意思表达清晰且避免了匈牙利法的臃肿，单词之间通过首字母大写来区分。通过第一个单词的首字母是否大写可以区分名称属于变量还是属于函数，而看到整串的大写字母可以断定为宏。

但是 Linux 不以这种习惯命名，对应于上面的一段程序，在 Linux 程序的命名如下所示：

```
#define PI 3.1415926
int min_value, max_value;
void send_data(void);
```

3.5.2 GNU C 与 ANSI C

Linux 系统上可用的 C 编译器是 GNU C 编译器，它建立在自由软件基金会的编程许可证的基础上，因此可以自由发布。GNU C 对标准 C 进行一系列扩展，以增强标准 C 的功能。

1. 零长度数组

GNU C 允许使用零长度数组，在定义变长对象的头结构时，这个特性非常有用。例如：

```
struct var_data
{
   int len;
   char data[0];
};
```

char data[0]仅仅意味着程序中通过 var_data 结构体实例的 data[index]成员可以访问 len 之后的第 index 个地址，它并没有为 data[]数组分配内存，因此 sizeof(struct var_data)=sizeof(int)。

假设 struct var_data 的数据域保存在 struct var_data 紧接着的内存区域，通过如下代码可以遍历这些数据：

```
struct var_data s;
...
for (i = 0; i < s.len; i++)
{
  printf("%02x", s.data[i]);
}
```

2. case 范围

GNU C 支持 case x…y 这样的语法，区间[x,y]的数都会满足这个 case 的条件，请看下面的代码：

```
switch (ch)
{
   case '0'... '9': c -= '0';
   break;
   case 'a'... 'f': c -= 'a' - 10;
   break;
   case 'A'... 'F': c -= 'A' - 10;
   break;
}
```

代码中的 case '0'... '9'等价于标准 C 中的如下代码：

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
```

3．语句表达式

GNU C 把包含在括号中的复合语句看做是一个表达式，称为语句表达式，它可以出现在任何允许表达式的地方。我们可以在语句表达式中使用原本只能在复合语句中使用的循环变量、局部变量等，例如：

```
#define min_t(type,x,y) \
({ type _ _x = (x); type _ _y = (y); _ _x < _ _y ? _ _x: _ _y; })

int ia, ib, mini;
float fa, fb, minf;

mini = min_t(int, ia, ib);
minf = min_t(float, fa, fb);
```

因为重新定义了_ _xx 和_ _y 这两个局部变量，所以以上述方式定义的宏将不会有副作用。在标准 C 中，对应的如下宏则会产生副作用：

```
#define min(x,y) ((x) < (y) ? (x) : (y))
```

代码 min(++ia,++ib)会被展开为((++ia) < (++ib) ? (++ia): (++ib))，传入宏的参数被增加两次。

4．typeof 关键字

typeof(x)语句可以获得 x 的类型，因此，我们可以借助 typeof 重新定义 min 这个宏：

```
#define min(x,y) ({ \
 const typeof(x) _x = (x);      \
 const typeof(y) _y = (y);      \
 (void) (&_x == &_y);          \
 _x < _y ? _x : _y; })
```

我们不需要像 min_t(type,x,y)这个宏那样把 type 传入，因为通过 typeof(x)、typeof(y)可以获得 type。代码行(void) (&_x == &_y)的作用是检查_x 和_y 的类型是否一致。

5．可变参数的宏

标准 C 只支持可变参数的函数，意味着函数的参数是不固定的，例如 printf()函数的原型为：

```
int printf( const char *format [, argument]... );
```

而在 GNU C 中，宏也可以接受可变数目的参数，例如：

```
#define pr_debug(fmt,arg...) \
        printk(fmt,##arg)
```

这里 arg 表示其余的参数可以是零个或多个，这些参数以及参数之间的逗号构成 arg 的值，在宏扩展时替换 arg，例如下列代码：

```
pr_debug("%s:%d",filename,line)
```

会被扩展为：

```
printk("%s:%d", filename, line)
```

使用“##”的原因是处理 arg 不代表任何参数的情况，这时候，前面的逗号就变得多余了。使用“##”之后，GNU C 预处理器会丢弃前面的逗号，这样，代码：

```
pr_debug("success!\n")
```

会被正确地扩展为：

```
printk("success!\n")
```

而不是：

```
printk("success!\n",)
```

6．标号元素

标准 C 要求数组或结构体的初始化值必须以固定的顺序出现，在 GNU C 中，通过指定索引或结构体成员名，允许初始化值以任意顺序出现。

指定数组索引的方法是在初始化值前添加“[INDEX] =”，当然也可以用“[FIRST … LAST] =”的形式指定一个范围。例如下面的代码定义一个数组，并把其中的所有元素赋值为 0：

```
unsigned char data[MAX] = { [0 ... MAX-1] = 0 };
```

下面的代码借助结构体成员名初始化结构体：

```
struct file_operations ext2_file_operations =
{
  llseek: generic_file_llseek,
  read: generic_file_read,
  write: generic_file_write,
  ioctl: ext2_ioctl,
  mmap: generic_file_mmap,
  open: generic_file_open,
  release: ext2_release_file,
  fsync: ext2_sync_file,
};
```

但是，Linux 2.6 推荐类似的代码应该尽量采用标准 C 的方式，如下所示：

```
struct file_operations ext2_file_operations =
{
    .llseek         = generic_file_llseek,
    .read           = generic_file_read,
    .write          = generic_file_write,
    .aio_read       = generic_file_aio_read,
    .aio_write      = generic_file_aio_write,
    .ioctl          = ext2_ioctl,
    .mmap           = generic_file_mmap,
    .open           = generic_file_open,
    .release        = ext2_release_file,
    .fsync          = ext2_sync_file,
    .readv          = generic_file_readv,
    .writev         = generic_file_writev,
    .sendfile       = generic_file_sendfile,
};
```

7．当前函数名

GNU C 预定义了两个标志符保存当前函数的名字，__FUNCTION__保存函数在源码中的名字，__PRETTY_FUNCTION__保存带语言特色的名字。在 C 函数中，这两个名字是相同的。

```
void example()
{
  printf("This is function:%s", __FUNCTION__);
}
```

代码中的__FUNCTION__意味着字符串“example”。

8．特殊属性声明

GNU C 允许声明函数、变量和类型的特殊属性，以便进行手工的代码优化和定制代码检查的方法。指定一个声明的属性，只需要在声明后添加__attribute__((ATTRIBUTE))。其中 ATTRIBUTE 为属性说明，如果存在多个属性，则以逗号分隔。GNU C 支持 noreturn、format、section、aligned、packed 等十多个属性。

noreturn 属性作用于函数，表示该函数从不返回。这会让编译器优化代码，并消除不必要的警告信息。例如：

```
# define ATTRIB_NORET __attribute__((noreturn)) ....
asmlinkage NORET_TYPE void do_exit(long error_code) ATTRIB_NORET;
```

format 属性也用于函数，表示该函数使用 printf、scanf 或 strftime 风格的参数，指定 format 属性可以让编译器根据格式串检查参数类型。例如：

```
asmlinkage int printk(const char * fmt, ...) __attribute__ ((format
(printf, 1, 2)));
```

上述代码中的第一个参数是格式串，从第二个参数开始都会根据 printf()函数的格式串规则检查参数。

unused 属性作用于函数和变量，表示该函数或变量可能不会被用到，这个属性可以避免编译器产生警告信息。

aligned 属性用于变量、结构体或联合体，指定变量、结构体或联合体的对界方式，以字节为单位，例如：

```
struct example_struct
{
  char a;
  int b;
  long c;
} __attribute__((aligned(4)));
```

表示该结构类型的变量以 4 字节对界。

packed 属性作用于变量和类型，用于变量或结构体成员时表示使用最小可能的对界，用于枚举、结构体或联合体类型时表示该类型使用最小的内存。例如：

```
struct example_struct
{
  char a;
  int b;
  long c __attribute__((packed));
};
```

编译器对结构体成员及变量对界的目的是为了更快地访问结构体成员及变量占据的内存。例如，对于一个 32 位的整型变量，若以 4 字节方式存放（即低两位地址为 00），则 CPU 在一个总线周期内就可以读取 32 位；若不然，CPU 需要两次总线周期才能组合为一个 32 位整型。

9．内建函数

GNU C 提供了大量的内建函数，其中大部分是标准 C 库函数的 GNU C 编译器内建版本，例如 memcpy()等，它们与对应的标准 C 库函数功能相同。

不属于库函数的其他内建函数的命名通常以_ _builtin 开始，如下所示。

- 内建函数_ _builtin_return_address (LEVEL)返回当前函数或其调用者的返回地址，参数 LEVEL 指定调用栈的级数，如 0 表示当前函数的返回地址，1 表示当前函数的调用者的返回地址。
- 内建函数_ _builtin_constant_p(EXP)用于判断一个值是否为编译时常数，如果参数 EXP 的值是常数，函数返回 1，否则返回 0。
- 内建函数_ _builtin_expect(EXP, C)用于为编译器提供分支预测信息，其返回值是整数表达式 EXP 的值，C 的值必须是编译时常数。

例如，下面的代码检测第 1 个参数是否为编译时常数以确定采用参数版本还是非参数版本的代码：

```
#define test_bit(nr,addr) \
(_ _builtin_constant_p(nr) ? \
constant_test_bit((nr),(addr)) : \
variable_test_bit((nr),(addr)))
```

3.5.3 do { } while(0)

在 Linux 内核中，经常会看到 do{}while(0)这样的语句，许多人开始都会疑惑，认为 do{}while(0)毫无意义，因为它只会执行一次，加不加 do{}while(0)效果是完全一样的，其实 do{}while(0)主要用于宏定义中。

这里用一个简单点的宏来演示：

```
#define SAFE_FREE(p) do{ free(p); p = NULL;} while(0)
```

假设这里去掉 do…while(0)，即定义 SAFE_DELETE 为：

```
#define SAFE_FREE(p) free(p); p = NULL;
```

那么以下代码：

```
if(NULL != p)
  SAFE_DELETE(p)
else
  ...//do something
```

会被展开为：

```
if(NULL != p)
  free(p); p = NULL;
else
```

```
  ...//do something
```

展开的代码中存在两个问题：

（1）if 分支后有两个语句，导致 else 分支没有对应的 if，编译失败；

（2）假设没有 else 分支，则 SAFE_FREE 中的第二个语句无论 if 测试是否通过都会执行。

将 SAFE_FREE 的定义加上{}就可以解决上述问题了，即：

```
#define SAFE_FREE(p) { free(p); p = NULL;}
```

这样，代码

```
if(NULL != p)
  SAFE_DELETE(p)
else
  ...//do something
```

会被展开为：

```
if(NULL != p)
 { free(p); p = NULL; }
else
  ...//do something
```

但是，在 C 程序中，每个语句后面加分号是一种约定俗成的习惯，那么，如下代码：

```
if(NULL != p)
  SAFE_DELETE(p);
else
  ...//do something
```

将被扩展为：

```
if(NULL != p)
 { free(p); p = NULL; };
else
  ...//do something
```

这样，else 分支就又没有对应的 if 了，编译将无法通过。假设用了 do{}while(0)，情况就不一样了，同样的代码会被展开为：

```
if(NULL != p)
 do{ free(p); p = NULL;} while(0);
else
  ...//do something
```

不会再出现编译问题。do{}while(0)的使用完全是为了保证宏定义的使用者能无编译错误地使用宏，它不对其使用者做任何假设。

3.5.4 goto

用不用 goto 一直是一个著名的争议话题，Linux 内核源代码中对 goto 的应用非常广泛，但是一般只限于错误处理中，其结构如下：

```
if(register_a()!=0)
{
    goto err;
```

```
    }
    if(register_b()!=0)
    {
        goto err1;
    }
    if(register_c()!=0)
    {
        goto err2;
    }
    if(register_d()!=0)
    {
        goto err3;
    }

    ...

    err3:
      unregister_c();
    err2:
      unregister_b();
    err1:
      unregister_a();
    err:
      return ret;
```

用于错误处理的 goto 的用法简单而高效，只需保证在错误处理时注销、资源释放的顺序与正常的注册、释放申请的顺序相反。

3.6 总结

本章主要讲解了 Linux 内核和 Linux 内核编程的基础知识，为进行 Linux 驱动开发打下软件基础。

在 Linux 内核方面，本章主要介绍了 Linux 内核的发展史、组成、特点、源代码结构、内核编译方法及内核引导过程。

由于 Linux 驱动编程本质属于内核编程，因此掌握内核编程的基础知识显得尤为重要。本章在这方面主要讲解了在内核中新增程序及目录和编写 Kconfig 和 Makefile 的方法，并分析了 Linux 下 C 编程习惯以及 Linux 所使用的 GNU C 针对标准 C 的扩展语法。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm

嵌入式学院 华清远见旗下品牌

第 4 章　Linux 内核模块

Linux 设备驱动会以内核模块的形式出现，因此，学会编写 Linux 内核模块编程是学习 Linux 设备驱动的先决条件。

4.1～4.2 节讲解了 Linux 内核模块的概念和结构，4.3～4.8 节对 Linux 内核模块的各个组成部分进行详细讲解，4.1～4.2 节与 4.3～4.8 节是整体与部分的关系。

4.9 节讲解了独立存在的 Linux 内核模块的 Makefile 文件编写方法和模块的编译方法。

华清远见

4.1 Linux 内核模块简介

Linux 内核的整体结构非常庞大，其包含的组件也非常多。我们怎样把需要的部分都包含在内核中呢？

一种方法是把所有需要的功能都编译到 Linux 内核。这会导致两个问题，一是生成的内核会很大，二是如果我们要在现有的内核中新增或删除功能，将不得不重新编译内核。

有没有一种机制使得编译出的内核本身并不需要包含所有功能，而在这些功能需要被使用的时候，其对应的代码可被动态地加载到内核中呢？

Linux 提供了这样的一种机制，这种机制被称为模块（Module），可以实现以上效果。模块具有以下特点。

- 模块本身不被编译入内核映像，从而控制了内核的大小。
- 模块一旦被加载，它就和内核中的其他部分完全一样。

为了使读者对模块建立初步的感性认识，我们先来看一个最简单的内核模块“Hello World”，如代码清单 4.1 所示。

代码清单 4.1　一个最简单的 Linux 内核模块

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  MODULE_LICENSE("Dual BSD/GPL");
4  static int hello_init(void)
5  {
6  printk(KERN_ALERT " Hello World enter\n");
7  return 0;
8  }
9  static void hello_exit(void)
10 {
11 printk(KERN_ALERT " Hello World exit\n ");
12 }
13 module_init(hello_init);
14 module_exit(hello_exit);
15
16 MODULE_AUTHOR("Song Baohua");
17 MODULE_DESCRIPTION("A simple Hello World Module");
18 MODULE_ALIAS("a simplest module");
```

这个最简单的内核模块只包含内核模块加载函数、卸载函数和对 Dual BSD/GPL 许可权限的声明以及一些描述信息。编译它会产生 hello.ko 目标文件，通过

“insmod ./hello.ko”命令可以加载它，通过“rmmod hello”命令可以卸载它，加载时输出“Hello World enter”，卸载时输出“Hello World exit”。

内核模块中用于输出的函数是内核空间的 printk()而非用户空间的 printf()，printk()的用法和 printf()相似，但前者可定义输出级别。printk()可作为一种最基本的内核调试手段。

在 Linux 系统中，使用 lsmod 命令可以获得系统中加载了的所有模块以及模块间的依赖关系，例如：

```
[root@localhost driver_study]# lsmod
Module                  Size   Used by
hello                   1568    0
ohci1394                32716   0
ide_scsi            16708   0
ide_cd                  39392   0
cdrom                   36960   1 ide_cd
```

lsmod 命令实际上读取并分析/proc/modules 文件，与上述 lsmod 命令结果对应的/proc/modules 文件如下：

```
[root@localhost driver_study]# cat /proc/modules
hello 1568 0 - Live 0xc8859000
ohci1394 32716 0 - Live 0xc88c8000
ieee1394 94420 1 ohci1394, Live 0xc8840000
ide_scsi 16708 0 - Live 0xc883a000
ide_cd 39392 0 - Live 0xc882f000
cdrom 36960 1 ide_cd, Live 0xc8876000
```

内核中已加载模块的信息也存在于/sys/module 目录下，加载 hello.ko 后，内核中将包含/sys/module/hello 目录，该目录下又包含一个 refcnt 文件和一个 sections 目录，在/sys/module/hello 目录下运行“tree –a”得到如下目录树：

```
[root@localhost hello]# tree -a
.
|-- refcnt
'-- sections
     |-- .bss
     |-- .data
     |-- .gnu.linkonce.this_module
     |-- .rodata
     |-- .rodata.str1.1
     |-- .strtab
     |-- .symtab
     |-- .text
     '-- __versions
```

modprobe 命令比 insmod 命令要强大，它在加载某模块时会同时加载该模块所依赖的

其他模块。使用 modprobe 命令加载的模块若以“modprobe -r filename”的方式卸载将同时卸载其依赖的模块。

使用 modinfo <模块名>命令可以获得模块的信息，包括模块的作者、模块的说明、模块所支持的参数以及 vermagic，如下所示：

```
[root@localhost driver_study]# modinfo hello.ko
filename:       hello.ko
license:        Dual BSD/GPL
author:         Song Baohua
description:    A simple Hello World Module
alias:          a simplest module
vermagic:       2.6.15.5 686 gcc-3.2
depends:
```

4.2 Linux 内核模块的程序结构

一个 Linux 内核模块主要由以下几个部分组成。

- 模块加载函数（必须）。

当通过 insmod 或 modprobe 命令加载内核模块时，模块的加载函数会自动被内核执行，完成本模块的相关初始化工作。

- 模块卸载函数（必须）。

当通过 rmmod 命令卸载某模块时，模块的卸载函数会自动被内核执行，完成与模块加载函数相反的功能。

- 模块许可证声明（必须）。

模块许可证（LICENSE）声明描述内核模块的许可权限，如果不声明 LICENSE，模块被加载时，将收到内核被污染 （kernel tainted）的警告。

在 Linux 2.6 内核中，可接受的 LICENSE 包括“GPL”、“GPL v2”、“GPL and additional rights”、“Dual BSD/GPL”、“Dual MPL/GPL”和“Proprietary”。

大多数情况下，内核模块应遵循 GPL 兼容许可权。Linux 2.6 内核模块最常见的是以 MODULE_LICENSE("Dual BSD/GPL")语句声明模块采用 BSD/GPL 双 LICENSE。

- 模块参数（可选）。

模块参数是模块被加载的时候可以被传递给它的值，它本身对应模块内部的全局变量。

- 模块导出符号（可选）。

内核模块可以导出符号（symbol，对应于函数或变量），这样其他模块可以使用本模块中的变量或函数。

- 模块作者等信息声明（可选）。

4.3 模块加载函数

Linux 内核模块加载函数一般以__init 标识声明，典型的模块加载函数的形式如代码清单 4.2 所示。

代码清单 4.2　内核模块加载函数

```
static int __init initialization_function(void)
{
/* 初始化代码 */
}
module_init(initialization_function);
```

模块加载函数必须以“module_init(函数名)”的形式被指定。它返回整型值，若初始化成功，应返回 0。而在初始化失败时，应该返回错误编码。在 Linux 内核里，错误编码是一个负值，在<linux/errno.h>中定义，包含-ENODEV、-ENOMEM 之类的符号值。返回相应的错误编码是种非常好的习惯，因为只有这样，用户程序才可以利用 perror 等方法把它们转换成有意义的错误信息字符串。

在 Linux 2.6 内核中，可以使用 request_module(const char *fmt, …)函数加载内核模块，驱动开发人员可以通过调用

```
request_module(module_name);
```

或

```
request_module("char-major-%d-%d", MAJOR(dev), MINOR(dev));
```

来加载其他内核模块。

在 Linux 内核中，所有标识为__init 的函数在连接的时候都放在.init.text 这个区段内，此外，所有的__init 函数在区段.initcall.init 中还保存了一份函数指针，在初始化时内核会通过这些函数指针调用这些__init 函数，并在初始化完成后释放 init 区段（包括.init.text，.initcall.init 等）。

4.4 模块卸载函数

Linux 内核模块卸载函数一般以__exit 标识声明，典型的模块卸载函数的形式如代码清单 4.3 所示。

代码清单 4.3　内核模块卸载函数

```
static void __exit cleanup_function(void)
```

```
{
/* 释放代码 */
}
module_exit(cleanup_function);
```

模块卸载函数在模块卸载的时候执行，不返回任何值，必须以“module_exit(函数名)”的形式来指定。

通常来说，模块卸载函数要完成与模块加载函数相反的功能，如下所示。

- 若模块加载函数注册了 XXX，则模块卸载函数应该注销 XXX。
- 若模块加载函数动态申请了内存，则模块卸载函数应释放该内存。
- 若模块加载函数申请了硬件资源（中断、DMA 通道、I/O 端口和 I/O 内存等）的占用，则模块卸载函数应释放这些硬件资源。
- 若模块加载函数开启了硬件，则卸载函数中一般要关闭硬件。

和__init 一样，__exit 也可以使对应函数在运行完成后自动回收内存。实际上，__init 和__exit 都是宏，其定义分别为：

```
#define __init __attribute__ ((__section__ (".init.text")))
```

和

```
#ifdef MODULE
#define __exit __attribute__ ((__section__ (".exit.text")))
#else
#define __exit __attribute_used__ __attribute__ ((__section__(".exit.text")))
#endif
```

数据也可以被定义为_initdata 和_exitdata，这两个宏分别为：

```
#define __initdata __attribute__ ((__section__ (".init.data")))
```

和

```
#define __exitdata __attribute__ ((__section__ (".exit.data")))
```

4.5 模块参数

我们可以用“module_param(参数名,参数类型,参数读/写权限)”为模块定义一个参数，例如下列代码定义了一个整型参数和一个字符指针参数：

```
static char *book_name = "深入浅出 Linux 设备驱动";
static int num = 4000;
module_param(num, int, S_IRUGO);
module_param(book_name, charp, S_IRUGO);
```

在装载内核模块时，用户可以向模块传递参数，形式为“insmode（或 modprobe）模块名 参数名=参数值”，如果不传递，参数将使用模块内定义的默认值。

参数类型可以是 byte、short、ushort、int、uint、long、ulong、charp（字符指针）、bool 或 invbool（布尔的反），在模块被编译时会将 module_param 中声明的类型与变

量定义的类型进行比较，判断是否一致。

模块被加载后，在/sys/module/目录下将出现以此模块名命名的目录。当“参数读/写权限”为 0 时，表示此参数不存在 sysfs 文件系统下对应的文件节点，如果此模块存在“参数读/写权限”不为 0 的命令行参数，在此模块的目录下还将出现 parameters 目录，包含一系列以参数名命名的文件节点，这些文件的权限值就是传入 module_param()的“参数读/写权限”，而文件的内容为参数的值。

除此之外，模块也可以拥有参数数组，形式为“module_param_array(数组名,数组类型,数组长,参数读/写权限)”。从 2.6.0～2.6.10 版本，需将数组长变量名赋给“数组长”，从 2.6.10 版本开始，需将数组长变量的指针赋给“数组长”，当不需要保存实际输入的数组元素个数时，可以设置“数组长”为 NULL。

运行 insmod 或 modprobe 命令时，应使用逗号分隔输入的数组元素。

现在我们定义一个包含两个参数的模块（如代码清单 4.4 所示），并观察模块加载时被传递参数和不传递参数时的输出。

代码清单 4.4　带参数的内核模块

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  MODULE_LICENSE("Dual BSD/GPL");
4
5  static char *book_name = "dissecting Linux Device Driver";
6  static int num = 4000;
7
8  static int book_init(void)
9  {
10  printk(KERN_INFO " book name:%s\n",book_name);
11  printk(KERN_INFO " book num:%d\n",num);
12  return 0;
13 }
14 static void book_exit(void)
15 {
16  printk(KERN_ALERT " Book module exit\n ");
17 }
18 module_init(book_init);
19 module_exit(book_exit);
20 module_param(num, int, S_IRUGO);
21 module_param(book_name, charp, S_IRUGO);
22
23 MODULE_AUTHOR("Song Baohua, author@linuxdriver.cn");
24 MODULE_DESCRIPTION("A simple Module for testing module params");
25 MODULE_VERSION("V1.0");
```

对上述模块运行“insmod book.ko”命令加载，相应输出都为模块内的默认值，通过查看“/var/log/messages”日志文件可以看到内核的输出，如下所示：

```
[root@localhost driver_study]# tail -n 2 /var/log/messages
Jul  2 01:03:10 localhost kernel:  <6> book name:dissecting Linux Device
Driver
Jul  2 01:03:10 localhost kernel:  book num:4000
```

当用户运行“insmod book.ko book_name='GoodBook' num=5000”命令时，输出的

是用户传递的参数，如下所示：

```
[root@localhost driver_study]# tail -n 2 /var/log/messages
Jul  2 01:06:21 localhost kernel:  <6> book name:GoodBook
Jul  2 01:06:21 localhost kernel:  book num:5000
```

4.6 导出符号

Linux 2.6 的“/proc/kallsyms”文件对应着内核符号表，它记录了符号以及符号所在的内存地址。

模块可以使用如下宏导出符号到内核符号表：

```
EXPORT_SYMBOL(符号名);
EXPORT_SYMBOL_GPL(符号名);
```

导出的符号将可以被其他模块使用，使用前声明一下即可。EXPORT_SYMBOL_GPL()只适用于包含 GPL 许可权的模块。代码清单 4.5 给出了一个导出整数加、减运算函数符号的内核模块的例子（这些导出符号没有实际意义，只是为了演示）。

代码清单 4.5 内核模块中的符号导出

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  MODULE_LICENSE("Dual BSD/GPL");
4
5  int add_integar(int a,int b)
6  {
7   return a+b;
8  }
9
10 int sub_integar(int a,int b)
11 {
12  return a-b;
13 }
14
15 EXPORT_SYMBOL(add_integar);
16 EXPORT_SYMBOL(sub_integar);
```

从“/proc/kallsyms”文件中找出 add_integar、sub_integar 相关信息：

```
[root@localhost driver_study]# cat /proc/kallsyms | grep integar
c886f050 r _ _kcrctab_add_integar       [export]
c886f058 r _ _kstrtab_add_integar       [export]
c886f070 r _ _ksymtab_add_integar       [export]
c886f054 r _ _kcrctab_sub_integar       [export]
c886f064 r _ _kstrtab_sub_integar       [export]
c886f078 r _ _ksymtab_sub_integar       [export]
c886f000 T add_integar  [export]
c886f00b T sub_integar  [export]
13db98c9 a _ _crc_sub_integar    [export]
```

```
e1626dee a _ _crc_add_integar    [export]
```

4.7 模块声明与描述

在 Linux 内核模块中，我们可以用 MODULE_AUTHOR、MODULE_DESCRIPTION、MODULE_ VERSION、MODULE_DEVICE_TABLE、MODULE_ALIAS 分别声明模块的作者、描述、版本、设备表和别名，例如：

```
MODULE_AUTHOR(author);
MODULE_DESCRIPTION(description);
MODULE_VERSION(version_string);
MODULE_DEVICE_TABLE(table_info);
MODULE_ALIAS(alternate_name);
```

对于 USB、PCI 等设备驱动，通常会创建一个 MODULE_DEVICE_TABLE，如代码清单 4.6 所示。

代码清单 4.6　驱动所支持的设备列表

```
1 /* 对应此驱动的设备表 */
2 static struct usb_device_id skel_table [] = {
3 { USB_DEVICE(USB_SKEL_VENDOR_ID,
4   USB_SKEL_PRODUCT_ID) },
5  { } /* 表结束 */
6 };
7
8 MODULE_DEVICE_TABLE (usb, skel_table);
```

此时，并不需要读者理解 MODULE_DEVICE_TABLE 的作用，后续相关章节会有详细介绍。

4.8 模块的使用计数

Linux 2.4 内核中，模块自身通过 MOD_INC_USE_COUNT 、MOD_DEC_USE_COUNT 宏来管理自己被使用的计数。

Linux 2.6 内核提供了模块计数管理接口 try_module_get(&module)和 module_put (&module)，从而取代 Linux 2.4 内核中的模块使用计数管理宏。模块的使用计数一般不必由模块自身管理，而且模块计数管理还考虑了 SMP 与 PREEMPT 机制的影响。

```
int try_module_get(struct module *module);
```

该函数用于增加模块使用计数；若返回为 0，表示调用失败，希望使用的模块没

有被加载或正在被卸载中。

```
void module_put(struct module *module);
```

该函数用于减少模块使用计数。

try_module_get ()与 module_put()的引入与使用与 Linux 2.6 内核下的设备模型密切相关。Linux 2.6 内核为不同类型的设备定义了 struct module *owner 域，用来指向管理此设备的模块。当开始使用某个设备时，内核使用 try_module_get(dev->owner)去增加管理此设备的 owner 模块的使用计数；当不再使用此设备时，内核使用 module_put(dev->owner)减少对管理此设备的 owner 模块的使用计数。这样，当设备在使用时，管理此设备的模块将不能被卸载。只有当设备不再被使用时，模块才允许被卸载。

在 Linux 2.6 内核下，对于设备驱动工程师而言，很少需要亲自调用 try_module_get()与 module_put()，因为此时开发人员所写的驱动通常为支持某具体设备的 owner 模块，对此设备 owner 模块的计数管理由内核里更底层的代码（如总线驱动或是此类设备共用的核心模块）来实现，从而简化了设备驱动的开发。

4.9 模块的编译

我们可以为代码清单 4.1 的模板编写一个简单的 Makefile，如下所示：

```
obj-m := hello.o
```

并使用如下命令编译 Hello World 模块，如下所示：

```
make -C /usr/src/linux-2.6.15.5/ M=/driver_study/ modules
```

如果当前处于模块所在的目录，则以下命令与上述命令同等：

```
make –C /usr/src/linux-2.6.15.5 M=$(pwd) modules
```

其中-C 后指定的是 Linux 内核源代码的目录，而 M=后指定的是 hello.c 和 Makefile 所在的目录，编译结果如下所示：

```
[root@localhost driver_study]# make -C /usr/src/linux-2.6.15.5/ M=/driver_study/ modules
make: Entering directory '/usr/src/linux-2.6.15.5'
  CC [M]  /driver_study/hello.o
/driver_study/hello.c:18:35: warning: no newline at end of file
  Building modules, stage 2.
  MODPOST
  CC      /driver_study/hello.mod.o
  LD [M]  /driver_study/hello.ko
make: Leaving directory '/usr/src/linux-2.6.15.5'
```

从中可以看出，编译过程中经历了这样的步骤：先进入 Linux 内核所在的目录，并编译出 hello.o 文件，运行 MODPOST 会生成临时的 hello.mod.c 文件，而后根据此

文件编译出 hello.mod.o，之后连接 hello.o 和 hello.mod.o 文件得到模块目标文件 hello.ko，最后离开 Linux 内核所在的目录。

中间生成的 hello.mod.c 文件的源代码如代码清单 4.7 所示。

代码清单 4.7　模块编译时生成的.mod.c 文件

```
#include <linux/module.h>
#include <linux/vermagic.h>
#include <linux/compiler.h>

MODULE_INFO(vermagic, VERMAGIC_STRING);

struct module _ _this_module
_ _attribute_ _((section(".gnu.linkonce.this_module"))) = {
 .name = KBUILD_MODNAME,
 .init = init_module,
#ifdef CONFIG_MODULE_UNLOAD
 .exit = cleanup_module,
#endif
};

static const char _ _module_depends[]
_ _attribute_used_ _
_ _attribute_ _((section(".modinfo"))) =
"depends=";
```

hello.mod.o 产生了 ELF（Linux 所采用的可执行/可连接的文件格式）的两个节，即 modinfo 和.gun.linkonce.this_module。

如果一个模块包括多个.c 文件（如 file1.c、file2.c），则应该以如下方式编写 Makefile：

```
obj-m := modulename.o
module-objs := file1.o file2.o
```

4.10 模块与 GPL

对于自己编写的驱动等内核代码，如果不编译为模块则无法绕开 GPL，编译为模块后企业在产品中使用模块，则公司对外不再需要提供对应的源代码，为了使公司产品所使用的 Linux 操作系统支持模块，需要完成如下工作。

- 在内核编译时应该选上“Enable loadable module support”，嵌入式产品一般不需要动态卸载模块，所以“可以卸载模块”不用选，当然选了也没关系，如图 4.1 所示。

```
Linux Kernel v2.6.15.5 Configuration
qqqq
```

4.11 总结

本章主要讲解了 Linux 内核模块的概念和基本的编程方法。内核模块由加载/卸载函数、功能函数以及一系列声明组成，它可以被传入参数，也可以导出符号供其他模块使用。

由于 Linux 设备驱动以内核模块的形式而存在，因此，掌握这一章的内容是编写任何类型设备驱动所必须的。在具体的设备驱动开发中，将驱动编译为模块也有很强的工程意义，因为如果将正在开发中的驱动直接编译入内核，而开发过程中会不断修改驱动的代码，则需要不断地编译内核并重启 Linux，但是如果编译为模块，则只需要 rmmod 并 insmod 即可，开发效率大为提高。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm
- 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程： 华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：
  http://www.farsight.com.cn/courses/TS-LinuxApp.htm
- 嵌入式 Linux 系统开发班：
  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm
- 嵌入式 Linux 驱动开发班：
  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm

华清远见

第 5 章　Linux 文件系统与设备文件系统

由于字符设备和块设备都很好地体现了“一切都是文件”的设计思想，掌握 Linux 文件系统、设备文件系统的知识非常重要。

首先，设备驱动最终通过操作系统的文件系统调用或 C 库函数（本质也基于系统调用）被访问。

其次，驱动工程师在设备驱动中不可避免地会与设备文件系统打交道，如 Linux 2.4 内核的 devfs 文件系统和 Linux 2.6 内核的基于 sysfs 的 udev 文件系统。

5.1 节讲解了通过 Linux API 和 C 库函数在用户空间进行 Linux 文件操作的编程方法。

5.2 节分析了 Linux 文件系统的目录结构，简单介绍了 Linux 内核中文件系统的实现，并给出了文件系统与设备驱动的关系。

5.3 节和 5.4 节分别讲解 Linux 2.4 内核的 devfs 和 Linux 2.6 内核所采用的 udev 设备文件系统，并分析了两者的区别。

华清远见

5.1 Linux 文件操作

5.1.1 文件操作的相关系统调用

Linux 的文件操作系统调用（在 Windows 编程领域，习惯称操作系统提供的接口为 API）涉及创建、打开、读写和关闭文件。

1. 创建

```
int creat(const char *filename, mode_t mode);
```

参数 mode 指定新建文件的存取权限，它同 umask 一起决定文件的最终权限（mode&umask），其中 umask 代表了文件在创建时需要去掉的一些存取权限。umask 可通过系统调用 umask()来改 变，如下所示：

```
int umask(int newmask);
```

该调用将 umask 设置为 newmask，然后返回旧的 umask，它只影响读、写和执行权限。

2. 打开

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

open()函数有两个形式，其中 pathname 是我们要打开的文件名（包含路径名称，默认时认为在当前路径下面），flags 可以是如表 5.1 所示的一个值或者是几个值的组合。

表 5.1 文件打开标志

标　志	含　义
O_RDONLY	以只读的方式打开文件
O_WRONLY	以只写的方式打开文件
O_RDWR	以读写的方式打开文件
O_APPEND	以追加的方式打开文件
O_CREAT	创建一个文件
O_EXEC	如果使用了 O_CREAT 而且文件已经存在，就会发生一个错误
O_NOBLOCK	以非阻塞的方式打开一个文件
O_TRUNC	如果文件已经存在，则删除文件的内容

O_RDONLY、O_WRONLY、O_RDWR 这 3 个标志只能使用任意的一个。

如果使用了 O_CREATE 标志，则使用的函数是 int open（const char *pathname,int

flags,mode_t mode）；这个时候我们还要指定 mode 标志，用来表示文件的访问权限。mode 可以是如表 5.2 所示值的组合。

表 5.2　　文件访问权限

标　志	含　义
S_IRUSR	用户可以读
S_IWUSR	用户可以写
S_IXUSR	用户可以执行

续表

标　志	含　义
S_IRWXU	用户可以读、写、执行
S_IRGRP	组可以读
S_IWGRP	组可以写
S_IXGRP	组可以执行
S_IRWXG	组可以读、写、执行
S_IROTH	其他人可以读
S_IWOTH	其他人可以写
S_IXOTH	其他人可以执行
S_IRWXO	其他人可以读、写、执行
S_ISUID	设置用户的执行 ID
S_ISGID	设置组的执行 ID

华清远见

除了可以通过上述宏进行“或”逻辑产生标志以外，我们也可以自己用数字来表示，Linux 总共用 5 个数字来表示文件的各种权限：第一位表示设置用户 ID；第二位表示设置组 ID；第三位表示用户自己的权限位；第四位表示组的权限；第五位表示其他人的权限。每个数字可以取 1（执行权限）、2（写权限）、4（读权限）、0（无）或者是这些值的和。

例如，如果要创建一个用户可读、可写、可执行，但是组没有权限，其他人可以读、可以执行的文件，并设置用户 ID 位。那么，应该使用的模式是 1（设置用户 ID）、0（不设置组 ID）、7（1+2+4，读、写、执行）、0（没有权限）、5（1+4，读、执行）即 10705，如下所示：

```
open("test", O_CREAT, 10705);
```

上述语句等价于：

```
open("test", O_CREAT, S_IRWXU | S_IROTH | S_IXOTH | S_ISUID );
```

如果文件打开成功，open 函数会返回一个文件描述符，以后对该文件的所有操作就可以通过对这个文件描述符进行操作来实现。

3．读写

在文件打开以后，我们才可对文件进行读写，Linux 系统中提供文件读写的系统

调用是 read、write 函数，如下所示：

```
int read(int fd, const void *buf, size_t length);
int write(int fd, const void *buf, size_t length);
```

其中参数 buf 为指向缓冲区的指针，length 为缓冲区的大小（以字节为单位）。函数 read()实现从文件描述符 fd 所指定的文件中读取 length 个字节到 buf 所指向的缓冲区中，返回值为实际读取的字节数。函数 write 实现将把 length 个字节从 buf 指向的缓冲区中写到文件描述符 fd 所指向的文件中，返回值为实际写入的字节数。

以 O_CREAT 为标志的 open 函数实际上实现了文件创建的功能，因此，下面的函数等同 creat()函数：

```
int open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
```

4. 定位

对于随机文件，我们可以随机地指定位置读写，使用如下函数进行定位：

```
int lseek(int fd, offset_t offset, int whence);
```

lseek()将文件读写指针相对 whence 移动 offset 个字节。操作成功时，返回文件指针相对于文件头的位置。参数 whence 可以使用如下值。

SEEK_SET：相对文件开头。

SEEK_CUR：相对文件读写指针的当前位置。

SEEK_END：相对文件末尾。

offset 可取负值，例如下述调用可将文件指针相对当前位置向前移动 5 个字节。

```
lseek(fd, -5, SEEK_CUR);
```

由于 lseek 函数的返回值为文件指针相对于文件头的位置， 因此下列调用的返回值就是文件的长度：

```
lseek(fd, 0, SEEK_END);
```

5. 关闭

当操作完成以后，就要关闭文件了，只要调用 close 函数就可以了，其中 fd 是要关闭的文件描述符。

```
int close(int fd);
```

例程：编写一个程序，在当前目录下创建用户可读写文件“hello.txt”，在其中写入“Hello, software weekly”，关闭该文件。再次打开该文件，读取其中的内容并输出在屏幕上，如代码清单 5.1 所示。

代码清单 5.1　Linux 文件操作用户空间编程（使用系统调用）

```
1   #include <sys/types.h>
2   #include <sys/stat.h>
3   #include <fcntl.h>
4   #include <stdio.h>
5   #define LENGTH 100
6   main()
```

```
7   {
8     int fd, len;
9     char str[LENGTH];
10
11    fd = open("hello.txt", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR); /*
12    创建并打开文件 */
13    if (fd)
14    {
15      write(fd, "Hello World", strlen("Hello World")); /*写入字符串 */
16
17      close(fd);
18    }
19
20    fd = open("hello.txt", O_RDWR);
21    len = read(fd, str, LENGTH); /* 读取文件内容 */
22    str[len] = '\0';
23    printf("%s\n", str);
24    close(fd);
25  }
```

编译并运行，执行结果为输出“Hello World”。

5.1.2 C 库函数的文件操作

C 库函数的文件操作实际上是独立于具体的操作系统平台的，不管是在 DOS、Windows、Linux 还是在 VxWorks 中都是这些函数。

1. 创建和打开

```
FILE *fopen(const char *path, const char *mode);
```

fopen()实现打开指定文件 filename，其中的 mode 为打开模式，C 库函数中支持的打开模式如表 5.3 所示。

表 5.3　　C 库函数文件打开标志

标　志	含　义
r、rb	以只读方式打开
w、wb	以只写方式打开。如果文件不存在，则创建该文件，否则文件被截断
a、ab	以追加方式打开。如果文件不存在，则创建该文件
r+、r+b、rb+	以读写方式打开
w+、w+b、wh+	以读写方式打开。如果文件不存在，则创建新文件，否则文件被截断
a+、a+b、ab+	以读和追加方式打开。如果文件不存在，则创建新文件

其中 b 用于区分二进制文件和文本文件，这一点在 DOS、Windows 系统中是有区分的，但 Linux 系统不区分二进制文件和文本文件。

2．读写

C 库函数支持以字符、字符串等为单位，支持按照某种格式进行文件的读写，这一组函数为：

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
char *fgets(char *s, int n, FILE *stream);
int fputs(const char *s, FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
int fscanf (FILE *stream, const char *format, ...);
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
size_t fwrite (const void *ptr, size_t size, size_t n, FILE *stream);
```

fread()实现从 stream 中读取 *n* 个字段，每个字段为 size 个字节，并将读取的字段放入 ptr 所指的字符数组中，返回实际已读取的字段数。在读取的字段数小于 num 时，可能是在函数调用时出现错误，也可能是读到文件的结尾。所以要通过调用 feof()和 ferror()来判断。

write()实现从缓冲区 ptr 所指的数组中把 *n* 个字段写到 stream 中，每个字段长为 size 个字节，返回实际写入的字段数。

另外，C 库函数还提供了读写过程中的定位能力，这些函数包括：

```
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
int fseek(FILE *stream, long offset, int whence);
```

3．关闭

利用 C 库函数关闭文件依然是很简单的操作，如下所示：

```
int fclose (FILE *stream);
```

例程：将第 5.1.1 节中的例程用 C 库函数来实现，如代码清单 5.2 所示。

华清远见

代码清单 5.2　Linux 文件操作用户空间编程范例（使用 C 库函数）

```
1   #include <stdio.h>
2   #define LENGTH 100
3   main()
4   {
5     FILE *fd;
6     char str[LENGTH];
7
8     fd = fopen("hello.txt", "w+"); /* 创建并打开文件 */
9     if (fd)
10    {
11      fputs("Hello World", fd); /* 写入字符串 */
12      fclose(fd);
13    }
14
15    fd = fopen("hello.txt", "r");
16    fgets(str, LENGTH, fd); /* 读取文件内容 */
17    printf("%s\n", str);
18    fclose(fd);
19  }
```

5.2 Linux 文件系统

5.2.1　Linux 文件系统目录结构

进入 Linux 根目录（即“/”，Linux 文件系统的入口，也是处于最高一级的目录），运行“ls –l”命令，可以看到 Linux 系统包含以下目录。

1．/bin

包含基本命令，如 ls、cp、mkdir 等，这个目录中的文件都是可执行的。

2．/boot

Linux 系统的内核及引导系统程序所需要的文件，如 vmlinuz、initrd.img 文件都位于这个目录中。

3．/dev

设备文件存储目录，应用程序通过对这些文件的读写和控制就可以访问实际的设备。

4．/etc

系统配置文件的所在地，一些服务器的配置文件也在这里，如用户账号及密码配置文件。

5．/home

普通用户的家目录。

6．/lib

库文件存放目录。

7．/lost+found

在 Ext2 或 Ext3 文件系统中，当系统意外崩溃或机器意外关机时会产生一些文件碎片放在这里。

8．/mnt

/mnt 这个目录一般是用于存放挂载储存设备的挂载目录的，比如有 cdrom 等目录，可以参看/etc/fstab 的定义。有时我们可以把让系统开机自动挂载文件系统，把挂载点放在这里也是可以的。

9．/opt

opt 是“可选”的意思，有些软件包会被安装在这里，比如在 Fedora Core 5.0 中的 OpenOffice 就是安装在这里，用户自己编译的软件包也可以安装在这个目录中。

10．/proc

操作系统运行时，进程及内核信息（比如 CPU、硬盘分区、内存信息等）存放在这里。/proc 目录为伪文件系统 proc 的挂载目录，proc 并不是真正的文件系统，它存在于内存之中。

11．/root

Linux 超级权限用户 root 的家目录。

12．/sbin

存放可执行文件，大多是涉及系统管理的命令，是超级权限用户 root 的可执行命令存放地，普通用户无权限执行这个目录下的命令，这个目录和/usr/sbin;/usr/X11R6/sbin 或/usr/local/sbin 目录是相似的。

13．/tmp

有时用户运行程序的时候会产生临时文件，/tmp 用来存放临时文件。

14．/usr

这个是系统存放程序的目录，比如命令、帮助文件等，它包含很多文件和目录，Linux 发行版提供的软件包大多被安装在这里。

华清远见

15．/var

var 表示的是变化的意思，这个目录的内容经常变动，如/var 的/var/log 目录被用来存放系统日志。

16．/sys

Linux 2.6 内核所支持的 sysfs 文件系统被映射在此目录。Linux 设备驱动模型中的总线、驱动和设备都可以在 sysfs 文件系统中找到对应的节点。当内核检测到在系统中出现了新设备后，内核会在 sysfs 文件系统中为该新设备生成一项新的记录。

17．/initrd

若在启动过程中使用了 initrd 映像作为临时根文件系统，则在执行完其上的/linuxrc 挂接真正的根文件系统后，原来的初始 RAM 文件系统被映射到/initrd 目录。

5.2.2 Linux 文件系统与设备驱动

图 5.1 所示为 Linux 系统中虚拟文件系统、磁盘文件（存放于 RamDisk、Flash、ROM、SD 卡、U 盘等文件系统中的文件也属于磁盘文件）及一般的设备文件与设备驱动程序之间的关系。

应用程序和 VFS 之间的接口是系统调用，而 VFS 与磁盘文件系统以及普通设备之间的接口是 file_operations 结构体成员函数，这个结构体包含对文件进行打开、关闭、读写、控制的一系列成员函数。

由于字符设备的上层没有磁盘文件系统，所以字符设备的 file_operations 成员函数就直接由设备驱动提供了，file_operations 正是字符设备驱动的核心。

华清远见

图 5.1　文件系统与设备驱动

而对于块存储设备而言，ext2、fat、jffs2 等文件系统中会实现针对 VFS 的 file_operations 成员函数，设备驱动层将看不到 file_operations 的存在。磁盘文件系统和设备驱动会将对磁盘上文件的访问最终转换成对磁盘上柱面和扇区的访问。

在设备驱动程序的设计中，一般而言，会关心结构体 file 和 inode 这两个结构体。

1．file 结构体

文件结构体代表一个打开的文件（设备对应于设备文件），系统中每个打开的文件在内核空间都有一个关联的 struct file。它由内核在打开文件时创建，并传递给在文件上进行操作的任何函数。在文件的所有实例都关闭后，内核释放这个数据结构。在内核和驱动源代码中，struct file 的指针通常被命名为 file 或 filp（即 file pointer）。代码清单 5.3 给出了文件结构体的定义。

代码清单 5.3　文件结构体

```
struct file
{
  union
  {
    struct list_head fu_list;
    struct rcu_head fu_rcuhead;
  } f_u;
  struct dentry *f_dentry; /*与文件关联的目录入口(dentry)结构*/
  struct vfsmount *f_vfsmnt;
  struct file_operations *f_op; /* 和文件关联的操作*/
  atomic_t f_count;
  unsigned int f_flags; /*文件标志，如 O_RDONLY、O_NONBLOCK、O_SYNC*/
  mode_t f_mode; /*文件读/写模式，FMODE_READ 和 FMODE_WRITE*/
  loff_t f_pos; /* 当前读写位置*/
  struct fown_struct f_owner;
  unsigned int f_uid, f_gid;
  struct file_ra_state f_ra;
```

```
18
19  unsigned long f_version;
20  void *f_security;
21
22  /* tty 驱动需要，其他的驱动可能需要 */
23  void *private_data; /*文件私有数据*/
24
25  #ifdef CONFIG_EPOLL
26    /* 被 fs/eventpoll.c 使用以便连接所有这个文件的钩子(hooks) */
27    struct list_head f_ep_links;
28    spinlock_t f_ep_lock;
29  #endif /* #ifdef CONFIG_EPOLL */
30  struct address_space *f_mapping;
31 };
```

文件读/写模式 mode、标志 f_flags 都是设备驱动关心的内容，而私有数据指针 private_data 在设备驱动中被广泛应用，大多被指向设备驱动自定义用于描述设备的结构体。

驱动程序中经常会使用如下类似的代码来检测用户打开文件的读写方式。

```
if (file->f_mode & FMODE_WRITE) //用户要求可写
{
}
if (file->f_mode & FMODE_READ) //用户要求可读
{
}
```

下面的代码可用于判断以阻塞还是非阻塞方式打开设备文件。

```
if (file->f_flags & O_NONBLOCK)    //非阻塞
    pr_debug("open: non-blocking\n");
else                               //阻塞
    pr_debug("open: blocking\n");
```

2．inode 结构体

VFS inode 包含文件访问权限、属主、组、大小、生成时间、访问时间、最后修改时间等信息。它是 Linux 管理文件系统的最基本单位，也是文件系统连接任何子目录、文件的桥梁，inode 结构体的定义如代码清单 5.4 所示。

代码清单 5.4　inode 结构体

```
1  struct inode
2  {
3    ...
4    umode_t i_mode; /* inode 的权限 */
5    uid_t i_uid; /* inode 拥有者的 id */
```

```
    gid_t i_gid; /* inode 所属的群组 id */
    dev_t i_rdev; /* 若是设备文件，此字段将记录设备的设备号 */
    loff_t i_size; /* inode 所代表的文件大小 */

    struct timespec i_atime; /* inode 最近一次的存取时间 */
    struct timespec i_mtime; /* inode 最近一次的修改时间 */
    struct timespec i_ctime; /* inode 的产生时间 */

    unsigned long i_blksize; /* inode 在做 I/O 时的区块大小 */
    unsigned long i_blocks; /* inode 所使用的 block 数，一个 block 为 512 byte*/

    struct block_device *i_bdev;
            /*若是块设备，为其对应的 block_device 结构体指针*/
    struct cdev *i_cdev;   /*若是字符设备，为其对应的 cdev 结构体指针*/
    ...
};
```

对于表示设备文件的 inode 结构，i_rdev 字段包含设备编号。Linux 2.6 设备编号分为主设备编号和次设备编号，前者为 dev_t 的高 12 位，后者为 dev_t 的低 20 位。下列操作用于从一个 inode 中获得主设备号和次设备号：

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

查看/proc/devices 文件可以获知系统中注册的设备，第 1 列为主设备号，第 2 列为设备名，如下所示：

```
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 21 sg
 29 fb
128 ptm
136 pts
171 ieee1394
180 usb
189 usb_device

Block devices:
  1 ramdisk
  2 fd
```

```
  8 sd
  9 md
 22 ide1
...
```

查看/dev 目录可以获知系统中包含的设备文件，日期的前两列给出了对应设备的主设备号和次设备号，如下所示：

```
crw-rw----    1 root     uucp      4,  64 Jan 30  2003 /dev/ttyS0
brw-rw----    1 root     disk      8,   0 Jan 30  2003 /dev/sda
```

主设备号是与驱动对应的概念，同一类设备一般使用相同的主设备号，不同类的设备一般使用不同的主设备号（但是也不排除在同一主设备号下包含有一定差异的设备）。因为同一驱动可支持多个同类设备，因此用次设备号来描述使用该驱动的设备的序号，序号一般从 0 开始。

内核 Documents 目录下的 devices.txt 文件描述了 Linux 设备号的分配情况，它由 LANANA（The Linux Assigned Names And Numbers Authority，网址：http://www.lanana.org/）组织维护，Torben Mathiasen 是其中的主要维护者。需要注意的是，LANANA 给出的设备号标准并不是硬性规定，在具体的设备驱动程序中，尽管一般会遵循 LANANA，但是也可以有例外。

5.3 devfs 设备文件系统

devfs（设备文件系统）是由 Linux 2.4 内核引入的，引入时被许多工程师给予了高度评价，它的出现使得设备驱动程序能自主地管理它自己的设备文件。具体来说，devfs 具有如下优点。

- 可以通过程序在设备初始化时在/dev 目录下创建设备文件，卸载设备时将它删除。
- 设备驱动程序可以指定设备名、所有者和权限位，用户空间程序仍可以修改所有者和权限位。
- 不再需要为设备驱动程序分配主设备号以及处理次设备号，在程序中可以直接给 register_chrdev()传递 0 主设备号以动态获得可用的主设备号，并在 devfs_register()中指定次设备号。

驱动程序应调用下面这些函数来进行设备文件的创建和删除工作。

```
/*创建设备目录*/
devfs_handle_t devfs_mk_dir(devfs_handle_t dir, const char *name, void *info);
/*创建设备文件*/
devfs_handle_t devfs_register(devfs_handle_t dir, const char *name, unsigned
    int flags, unsigned int major, unsigned int minor, umode_t mode, void *ops,
```

```
    void *info);
  /*撤销设备文件*/
  void devfs_unregister(devfs_handle_t de);
```

在 Linux 2.4 的设备驱动编程中，分别在模块加载和卸载函数中创建和撤销设备文件是被普遍采用并值得大力推荐的好方法。代码清单 5.5 给出了一个使用 devfs 的例子。

代码清单 5.5　devfs 的使用范例

```
1   static devfs_handle_t devfs_handle;
2   static int _ _init xxx_init(void)
3   {
4     int ret;
5     int i;
6     /*在内核中注册设备*/
7     ret = register_chrdev(XXX_MAJOR, DEVICE_NAME, &xxx_fops);
8     if (ret < 0)
9     {
10      printk(DEVICE_NAME " can't register major number\n");
11      return ret;
12    }
13    /*创建设备文件*/
14    devfs_handle         =devfs_register(NULL,         DEVICE_NAME,
DEVFS_FL_DEFAULT,
15    XXX_MAJOR, 0, S_IFCHR | S_IRUSR | S_IWUSR, &xxx_fops, NULL);
16    ...
17    printk(DEVICE_NAME " initialized\n");
18    return 0;
19  }
20
21  static void _ _exit xxx_exit(void)
22  {
23    devfs_unregister(devfs_handle); /*撤销设备文件*/
24    unregister_chrdev(XXX_MAJOR, DEVICE_NAME); /*注销设备*/
25  }
26
27  module_init(xxx_init);
28  module_exit(xxx_exit);
```

代码中第 7 行和第 24 行分别用于注册和注销字符设备，使用的 register_chrdev()和 unregister_ chrdev()在 Linux2.6 内核中虽然仍然被支持，但是是过时的做法。第 14 行和第 23 行分别用于创建和删除 devfs 文件节点。

5.4 udev 设备文件系统

5.4.1 udev 与 devfs 的区别

尽管 devfs 有这样和那样的优点，但是，在 Linux 2.6 内核中，devfs 被认为是过时的方法，并最终被抛弃，udev 取代了它。Linux VFS 内核维护者 Al Viro 指出了 udev 取代 devfs 的几点原因：

- devfs 所做的工作被确信可以在用户态来完成。
- 一些 bug 相当长的时间内未被修复。
- devfs 的维护者和作者停止了对代码的维护工作。

udev 完全在用户态工作，利用设备加入或移除时内核所发送的热插拔事件（hotplug event）来工作。在热插拔时，设备的详细信息会由内核输出到位于/sys 的 sysfs 文件系统。udev 的设备命名策略、权限控制和事件处理都是在用户态下完成的，它利用 sysfs 中的信息来进行创建设备文件节点等工作。

由于 udev 根据系统中硬件设备的状态动态更新设备文件，进行设备文件的创建和删除等，因此，在使用 udev 后，/dev 目录下就会只包含系统中真正存在的设备了。

devfs 与 udev 的另一个显著区别在于：采用 devfs，当一个并不存在的/dev 节点被打开的时候，devfs 能自动加载对应的驱动，而 udev 则不能。这是因为 udev 的设计者认为 Linux 应该在设备被发现的时候加载驱动模块，而不是当它被访问的时候。udev 的设计者认为 devfs 所提供的打开/dev 节点时自动加载驱动的功能对于一个配置正确的计算机是多余的。系统中所有的设备都应该产生热插拔事件并加载恰当的驱动，而 udev 能注意到这点并且为它创建对应的设备节点。

5.4.2 sysfs 文件系统与 Linux 设备模型

1. sysfs 文件系统

Linux 2.6 内核引入了 sysfs 文件系统，sysfs 被看成是与 proc、devfs 和 devpty 同类别的文件系统，该文件系统是一个虚拟的文件系统，它可以产生一个包括所有系统硬件的层级视图，与提供进程和状态信息的 proc 文件系统十分类似。

sysfs 把连接在系统上的设备和总线组织成为一个分级的文件，它们可以由用户空间存取，向用户空间导出内核数据结构以及它们的属性。sysfs 的一个目的就是展示设备驱动模型中各组件的层次关系，其顶级目录包括 block、device、bus、drivers、class、power 和 firmware。

block 目录包含所有的块设备，devices 目录包含系统所有的设备并根据设备挂接的总线类型组织成层次结构，bus 目录包含系统中所有的总线类型，drivers 目录包括内核中所有已注册的设备驱动程序，class 目录包含系统中的设备类型（如网卡设备、声卡设备、输入设备等）。在/sys 目录运行 tree 会得到一个相当长的树型目录，下面摘取一部分：

```
    |-- block
    |   |-- fd0
    |   |-- md0
    |   |-- ram0
    |   |-- ram1
    |   |-- ...
    |-- bus
    |   |-- eisa
    |   |   |-- devices
    |   |   '-- drivers
    |   |-- ide
    |   |-- ieee1394
    |   |-- pci
    |   |   |-- devices
    |               |               |               |--      0000:00:00.0
-> ../../../devices/pci0000:00/0000:00:00.0
    |               |               |               |--      0000:00:01.0
-> ../../../devices/pci0000:00/0000:00:01.0
    |               |               |               |--      0000:00:07.0
-> ../../../devices/pci0000:00/0000:00:07.0
    |   |   '-- drivers
    |   |       |-- PCI_IDE
    |   |       |   |-- bind
    |   |       |   |-- new_id
    |   |       |   '-- unbind
    |   |       '-- pcnet32
    |           |                                   |--    0000:00:11.0
-> ../../../../../devices/pci0000:00/0000:00:11.0
    |   |           |-- bind
    |   |           |-- new_id
    |   |           '-- unbind
    |   |-- platform
    |   |-- pnp
    |   '-- usb
    |       |-- devices
    |       '-- drivers
    |           |-- hub
    |           |-- usb
    |           |-- usb-storage
    |           '-- usbfs
    |-- class
    |   |-- graphics
    |   |-- hwmon
    |   |-- ieee1394
    |   |-- ieee1394_host
    |   |-- ieee1394_node
    |   |-- ieee1394_protocol
    |   |-- input
    |   |-- mem
    |   |-- misc
    |   |-- net
    |   |-- pci_bus
    |   |   |-- 0000:00
    |   |   |   |-- bridge -> ../../../devices/pci0000:00
    |   |   |   |-- cpuaffinity
    |   |   |   '-- uevent
    |   |   '-- 0000:01
```

```
|   |      |-- bridge -> ../../../devices/pci0000:00/0000:00:01.0
|   |      |-- cpuaffinity
|   |      '-- uevent
|   |-- scsi_device
|   |-- scsi_generic
|   |-- scsi_host
|   |-- tty
|   |-- usb
|   |-- usb_device
|   |-- usb_host
|   '-- vc
|-- devices
|   |-- pci0000:00
|   |   |-- 0000:00:00.0
|   |   |-- 0000:00:07.0
|   |   |-- 0000:00:07.1
|   |   |-- 0000:00:07.2
|   |   |-- 0000:00:07.3
|   |-- platform
|   |   |-- floppy.0
|   |   |-- host0
|   |   |-- i8042
|   |-- pnp0
|   '-- system
|-- firmware
|-- kernel
|   '-- hotplug_seqnum
|-- module
|   |-- apm
|   |-- autofs
|   |-- cdrom
|   |-- eisa_bus
|   |-- i8042
|   |-- ide_cd
|   |-- ide_scsi
|   |-- ieee1394
|   |-- md_mod
|   |-- ohci1394
|   |-- parport
|   |-- parport_pc
|   |-- usb_storage
|   |-- usbcore
|   |   |-- parameters
|   |   |-- refcnt
|   |   '-- sections
|   |-- virtual_root
|   |   '-- parameters
|   |      '-- force_probe
|   '-- vmhgfs
|      |-- refcnt
|      '-- sections
|         '-- _ _versions
'-- power
   '-- state
```

在/sys/bus 的 pci 等子目录下，又会在分出 drivers 和 devices 目录，而 devices 目录中的文件是对/sys/devices 目录中文件的符号链接。同样地，/sys/class 目录下包含许多对

/sys/devices 下文件的链接。如图 5.2 所示，这与设备、驱动、总线和类的现实状况是直接对应的，也正符合 Linux 2.6 内核的设备模型。

图 5.2　Linux 设备模型

随着技术的不断进步，系统的拓扑结构越来越复杂，对智能电源管理、热插拔以及即插即用的支持要求也越来越高，Linux 2.4 内核已经难以满足这些需求。为适应这种形势的需要，Linux 2.6 内核开发了上述全新的设备、总线、类和驱动环环相扣的设备模型。图 5.3 也形象地表示了 Linux 驱动模型中设备、总线和类之间的关系。

图 5.3　Linux 驱动模型中设备、总线和类之间的关系

大多数情况下，Linux 2.6 内核中的设备模型代码会处理好这些关系，内核中的总

线级和其他内核子系统会完成与设备模型的交互，这使得驱动工程师几乎不需要关心设备模型。但是，理解 Linux 设备模型的实现机制对驱动工程师仍然是大有裨益的，具体而言，内核将借助下文将介绍的 kobject、kset、subsystem、bus_type、device、device_driver、class、class_device、class_interface 等重量级数据结构来完成设备模型的架构。

2．kobject 内核对象

kobject 是 Linux 2.6 引入的设备管理机制，在内核中由 kobject 结构体表示，这个数据结构使所有设备在底层都具有统一的接口。kobject 提供了基本的对象管理能力，是构成 Linux 2.6 设备模型的核心结构，每个在内核中注册的 kobject 对象都对应于 sysfs 文件系统中的一个目录。 kobject 结构体的定义如代码清单 5.6 所示。

代码清单 5.6 kobject 结构体

```
struct kobject
{
  char *k_name;
  char name[KOBJ_NAME_LEN]; //对象名称
  struct kref kref; //对象引用计数
  struct list_head entry; //用于挂接该 kobject 对象到 kset 链表
  struct kobject *parent; //指向父对象的指针
  struct kset *kset; //所属 kset 的指针
  struct kobj_type *ktype; //指向对象类型描述符的指针
  struct dentry *dentry; //sysfs 文件系统中与该对象对应的文件节点入口
};
```

内核通过 kobject 的 kref 成员实现对象引用计数管理，且提供两个函数 kobject_get()、kobject_put()分别用于增加和减少引用计数，当引用计数为 0 时，所有该对象使用的资源将被释放。

kobject 的 ktype 成员是一个指向 kobj_type 结构的指针，表示该对象的类型。如代码清单 5.7 所示，kobj_type 数据结构包含 3 个成员：用于释放 kobject 占用的资源的 release()函数、指向 sysfs 操作的 sysfs_ops 指针和 sysfs 文件系统默认属性列表。

代码清单 5.7 kobj_type 结构体

```
struct kobj_type
{
 void (*release)(struct kobject *);//release 函数
 struct sysfs_ops    * sysfs_ops;//属性操作
 struct attribute    ** default_attrs;//默认属性
};
```

kobj_type 结构体种的 sysfs_ops 包括 store()和 show()两个成员函数，用于实现属性的读写，代码清单 5.8 给出了 sysfs_ops 结构体的定义。当从用户空间读取属性时，show()函数将被调用，该函数将指定属性值存入 buffer 中返回给用户，而 store()函数用于存储用户通过 buffer 传入的属性值。和 kobject 不同的是，属性在 sysfs 中呈现为一个文件，而 kobject 则呈现为 sysfs 中的目录。

代码清单 5.8 sysfs_ops 结构体

```
struct sysfs_ops
{
 ssize_t (*show)(struct kobject *, struct attribute *,char *);
```

```
    4   ssize_t (*store)(struct kobject *,struct attribute *,const char *,
size_t);
    5 };
```

Linux 内核中提供一系列操作 kobject 的函数：

```
void kobject_init(struct kobject * kobj);
```

该函数用于初始化 kobject，它设置 kobject 引用计数为 1，entry 域指向自身，其所属 kset 引用计数加 1。

```
int kobject_set_name(struct kobject *kobj, const char *format, ...);
```

该函数用于设置指定 kobject 的名称。

```
void kobject_cleanup(struct kobject * kobj) 和 void
kobject_release(struct kref *kref);
```

该函数用于清除 kobject，当其引用计数为 0 时，释放对象占用的资源。

```
struct kobject *kobject_get(struct kobject *kobj);
```

该函数用于将 kobj 对象的引用计数加 1，同时返回该对象的指针。

```
void kobject_put(struct kobject * kobj);
```

该函数用于将 kobj 对象的引用计数减 1，如果引用计数降为 0，则调用 kobject_release()释放该 kobject 对象。

```
int kobject_add(struct kobject * kobj);
```

该函数用于将 kobject 对象加入 Linux 设备层次，它会挂接该 kobject 对象到 kset 的 list 链中，增加父目录各级 kobject 的引用计数，在其 parent 指向的目录下创建文件节点，并启动该类型内核对象的 hotplug 函数。

```
int kobject_register(struct kobject * kobj);
```

该函数用于注册 kobject，它会先调用 kobject_init()初始化 kobj，再调用 kobject_add()完成该内核对象的添加。

```
void kobject_del(struct kobject * kobj);
```

这个函数是 kobject_add()的反函数，它从 Linux 设备层次(hierarchy)中删除 kobject 对象。

```
void kobject_unregister(struct kobject * kobj);
```

这个函数是 kobject_register()的反函数，用于注销 kobject。与 kobject_register()相反，它首先调用 kobject_del()从设备层次中删除该对象，再调用 kobject_put()减少该对象的引用计数，如果引用计数降为 0，则释放该 kobject 对象。

3．kset 内核对象集合

kobject 通常通过 kset 组织成层次化的结构，kset 是具有相同类型的 kobject 的集合，在内核中用 kset 数据结构表示，其定义如代码清单 5.9 所示。

代码清单 5.9　kset 结构体

```
1 struct kset
2 {
3   struct subsystem    * subsys; //所在的subsystem的指针
4   struct kobj_type    * ktype; //指向该kset对象类型描述符的指针
```

```
    struct list_head    list; //用于连接该 kset 中所有 kobject 的链表头
    spinlock_t      list_lock;
    struct kobject      kobj; //嵌入的 kobject
    struct kset_uevent_ops  * uevent_ops;//事件操作集
};
```

包含在 kset 中的所有 kobject 被组织成一个双向循环链表，list 即是该链表的头。ktype 成员指向一个 kobj_type 结构，被该 kset 中的所有 kobject 共享，表示这些对象的类型。kset 数据结构还内嵌了一个 kobject 对象（kobj 成员表示），所有属于这个 kset 的 kobject 对象的 parent 均指向这个内嵌的对象。此外，kset 还依赖于 kobj 维护引用计数， kset 的引用计数实际上就是内嵌的 kobject 对象的引用计数。

与 kobject 相似，Linux 提供一系列函数操作 kset。kset_init()完成指定 kset 的初始化，kset_get()和 kset_put()分别增加和减少 kset 对象的引用计数，kset_add()和 kset_del()函数分别实现将指定 keset 对象加入设备层次和从其中删除，kset_register()函数完成 kset 的注册，kset_unregister()函数则完成 kset 的注销。

kobject 被创建或删除时会产生事件（event），kobject 所属的 kset 将有机会过滤事件或为用户空间添加信息。每个 kset 能支持一些特定的事件变量，在热插拔事件发生时，kset 的成员函数可以设置一些事件变量，这些变量将被导出到用户空间。kset 的 uevent_ops 成员是执行该 kset 事件操作集 kset_uevent_ops 的指针，kset_uevent_ops 的定义如代码清单 5.10 所示。

代码清单 5.10 kset_uevent_ops 结构体

```
struct kset_uevent_ops
{
  int (*filter)(struct kset *kset, struct kobject *kobj);//事件过滤
  const char *(*name)(struct kset *kset, struct kobject *kobj);
  int (*uevent)(struct kset *kset, struct kobject *kobj, char **envp,
          int num_envp, char *buffer, int buffer_size);//环境变量设置
};
```

filter()函数用于过滤掉不需要导出到用户空间的事件，uevent()函数用于导出一些环境变量给用户的热插拔处理程序，各类设备导出的环境变量如下。

- PCI 设备：ACTION（“add”/“remove”）、PCI_ CLASS（16 进制的 PCI 类、子类、接口，如 c0310）、PCI_ ID（Vendor:Device，如 0123:4567）、PCI_SUBSYS_ID（子系统 Vendor: 子系统 Device，如 89ab:cdef）、PCI_ SLOT_ NAME（Bus:Slot.Func，如 00:07.2）。
- USB 设备：ACTION（“add”/“remove”）、DEVPATH（/sys/DEVPATH）、PRODUCT（idVendor/ idProduct/bcdDevice，如 46d/c281/108）、TYPE（bDeviceClass/bDeviceSubClass/bDeviceProtocol，如 9/0/0）、INTERFACE（bInterfaceClass/bInterfaceSubClass/bInterfaceProtocol，如 3/1/1），如果内核配置了 usbfs 文件系统，还会导出 DEVFS（USB 驱动列表的位置，如/proc/bus/usb）和

DEVICE（USB 设备节点路径）。

- 网络设备：ACTION（“register”/“unregister”）、INTERFACE（接口名，如“eth0”）。
- 输入设备：ACTION（“add”/“remove”）、PRODUCT（idbus/idvendor/idproduct/idversion，如 1/46d/c281/108）、NAME（设备名，如“ALCOR STRONG MAN KBD HUB”）、PHYS（设备物理地址 ID，如 usb-00:07.2-2.3/input0）、EV（来自 evbit，如 120002）、KEY（来自 evbit，如 e080ffdf 1dfffff ffffffff fffffe）、LED（来自 ledbit，如 7）。
- IEEE1394 设备：ACTION（“add”/“remove”）、VENDOR_ID（24 位的 vendor id，如 123456）、GUID（64 位 ID）、SPECIFIER_ID（24 位协议标准拥有者 ID）、VERSION（协议标准版本 ID）。

用户空间的热插拔脚本根据传入给它的参数（如 PCI 的参数为热插拔程序路径、“pci”和 0）以及内核导出的环境变量采取相应的行动，如下面的脚本程序会在 PRODUCT 为“82d/100/0”的 USB 设备被插入时加载 visor 模块，在被拔出时卸载 visor 模块，如下所示。

```
if [ "$1"="usb" ]; then
    if [ "$PRODUCT"="82d/100/0" ]; then
        if[ "$ACTION" = "add" ]; then
            /sbin/modprobe visor
        else
            /sbin/rmmod visor
        fi
    fi
fi
```

4．subsystem 内核对象子系统

subsystem 是一系列 kset 的集合，它描述系统中某一类设备子系统，如 block_subsys 表示所有的块设备，对应于 sysfs 文件系统中的 block 目录。devices_subsys 对应于 sysfs 中的 devices 目录，描述系统中所有的设备。subsystem 由 struct subsystem 数据结构描述，其定义如代码清单 5.11 所示。

代码清单 5.11　subsystem 结构体

```
struct subsystem
{
  struct kset kset; //内嵌的 kset 对象
  struct rw_semaphore rwsem; //互斥访问信号量
};
```

每个 kset 必须属于某个 subsystem，通过设置 kset 结构中的 subsys 域指向指定的 subsystem 可以将一个 kset 加入到该 subsystem。所有挂接到同一 subsystem 的 kset 共享同一个 rwsem 信号量，用于同步访问 kset 中的链表。

内核也提供了一组类似于 kobect 和 kset 的操作 subsystem 的操作，如下所示：

```
void subsystem_init(struct subsystem *subsys);
int subsystem_register(struct subsystem *subsys);
void subsystem_unregister(struct subsystem *subsys);
struct subsystem *subsys_get(struct subsystem *subsys)
void subsys_put(struct subsystem *subsys);
```

5. Linux 设备模型组件

系统中的任一设备在设备模型中都由一个 device 对象描述，其对应的数据结构 struct device 定义如代码清单 5.12 所示。

代码清单 5.12　device 结构体

```
struct device
{
 struct klist        klist_children; // 设备列表中的孩子列表
 struct klist_node   knode_parent;// 兄弟节点
 struct klist_node   knode_driver; // 驱动结点
 struct klist_node   knode_bus; // 总线结点
 struct device   * parent; // 指向父设备

 struct kobject kobj;   // 内嵌一个 kobject 对象
 char    bus_id[BUS_ID_SIZE];// 总线上的位置
 struct device_attribute uevent_attr;

 struct semaphore    sem;

 struct bus_type * bus;  // 总线
 struct device_driver *driver;// 使用的驱动
 void        *driver_data;   // 驱动私有数据
 void        *platform_data; //平台特定的数据
 void        *firmware_data; //固件特定的数据(如 ACPI、BIOS 数据)
 struct dev_pm_info  power;

 u64     *dma_mask;  //dma 掩码
 u64     coherent_dma_mask;
 struct list_head    dma_pools;  // DMA 缓冲池
 struct dma_coherent_mem *dma_mem;

 void    (*release)(struct device * dev); // 释放设备方法
};
```

device 结构体用于描述设备相关的信息设备之间的层次关系，以及设备与总线、

驱动的关系。

内核提供了相应的函数用于操作 device 对象。其中 device_register()函数将一个新的 device 对象插入设备模型，并自动在/sys/devices 下创建一个对应的目录。device_unregister()完成相反的操作，注销设备对象。get_device()和 put_device()分别增加与减少设备对象的引用计数。通常 device 结构体不单独使用，而是包含在更大的结构体中，比如描述 PCI 设备的 struct pci_dev，其中的 dev 域就是一个 device 对象。

系统中的每个驱动程序由一个 device_driver 对象描述，对应的数据结构的定义如代码清单 5.13 所示。

代码清单 5.13　device_driver 结构体

```
struct device_driver
{
 const char      * name;   //设备驱动程序的名称
 struct bus_type     * bus; //总线

 struct completion   unloaded;
 struct kobject      kobj;//内嵌的 kobject 对象
 struct klist        klist_devices;
 struct klist_node   knode_bus;

 struct module       * owner;

 int (*probe)    (struct device * dev); //指向设备探测函数
 int (*remove)   (struct device * dev); //指向设备移除函数
 void    (*shutdown) (struct device * dev);
 int (*suspend)  (struct device * dev, pm_message_t state);
 int (*resume)   (struct device * dev);
};
```

与 device 结构体类似，device_driver 对象依靠内嵌的 kobject 对象实现引用计数管理和层次结构组织。内核提供类似的函数用于操作 device_driver 对象。如 get_driver()增加引用计数，driver_register()用于向设备模型插入新的 driver 对象，同时在 sysfs 文件系统中创建对应的目录。device_driver()结构体还包括几个函数，用于处理探测、移除和电源管理事件。

系统中总线由 struct bus_type 描述，其定义如代码清单 5.14 所示。

代码清单 5.14　bus_type 结构体

```
struct bus_type
{
 const char      * name;//总线类型的名称

 struct subsystem    subsys;//与该总线相关的 subsystem
```

```
struct kset     drivers;//所有与该总线相关的驱动程序集合
struct kset     devices;//所有挂接在该总线上的设备集合
struct klist        klist_devices;
struct klist        klist_drivers;

struct bus_attribute    * bus_attrs;//总线属性
struct device_attribute * dev_attrs;//设备属性
struct driver_attribute * drv_attrs; //驱动程序属性

int     (*match)(struct device * dev, struct device_driver * drv);
int     (*uevent)(struct device *dev, char **envp,
                int num_envp, char *buffer, int buffer_size);//事件
int     (*probe)(struct device * dev);
int     (*remove)(struct device * dev);
void        (*shutdown)(struct device * dev);
int     (*suspend)(struct device * dev, pm_message_t state);
int     (*resume)(struct device * dev);
};
```

每个 bus_type 对象都内嵌一个 subsystem 对象，bus_subsys 对象管理系统中所有总线类型的 subsystem 对象。每个 bus_type 对象都对应/sys/bus 目录下的一个子目录，如 PCI 总线类型对应于/sys/bus/pci。在每个这样的目录下都存在两个子目录：devices 和 drivers（分别对应于 bus_type 结构中的 devices 和 drivers 域）。其中 devices 子目录描述连接在该总线上的所有设备，而 drivers 目录则描述与该总线关联的所有驱动程序。与 device_driver 对象类似，bus_type 结构还包含几个函数处理热插拔、即插即拔和电源管理事件的函数。

系统中的设备类由 struct class 描述，表示某一类设备。所有的 class 对象都属于 class_subsys 子系统，对应于 sysfs 文件系统中的/sys/class 目录。代码清单 5.15 给出了 class 结构体的定义。

代码清单 5.15　class 结构体

```
struct class
{
  const char      * name;  //类名
  struct module       * owner;

  struct subsystem    subsys; //对应的subsystem
  struct list_head    children; //class_device链表
  struct list_head    interfaces; //class_interface链表
  struct semaphore    sem;    //children和interfaces链表锁

  struct class_attribute      * class_attrs;//类属性
```

```
struct class_device_attribute  * class_dev_attrs;//类设备属性

int (*uevent)(struct class_device *dev, char **envp,
            int num_envp, char *buffer, int buffer_size);//事件

void    (*release)(struct class_device *dev);
void    (*class_release)(struct class *class);
};
```

每个 class 对象包括一个 class_device 链表，每个 class_device 对象表示一个逻辑设备，并通过 struct class_device 中的 dev 成员（一个指向 struct device 的指针）关联一个物理设备。这样，一个逻辑设备总是对应于一个物理设备，但是一个物理设备却可能对应于多个逻辑设备，代码清单 5.16 给出了 class_device 的定义。此外，class 结构中还包括用于处理热插拔、即插即拔和电源管理事件的函数，这与 bus_type 对象相似。

代码清单 5.16　class_device 结构体

```
struct class_device
{
 struct list_head    node;

 struct kobject      kobj;//内嵌的 kobject
 struct class        * class;    //所属的类
 dev_t           devt;       // dev_t
 struct class_device_attribute *devt_attr;
 struct class_device_attribute uevent_attr;
 struct device       * dev;  //如果存在，创建到/sys/devices 相应入口的符号链接
 void            * class_data;   //私有数据
 struct class_device *parent;    //父设备

 void    (*release)(struct class_device *dev);
 int (*uevent)(struct class_device *dev, char **envp,
            int num_envp, char *buffer, int buffer_size);
 char    class_id[BUS_ID_SIZE]; //类标识
};
```

下面两个函数用于注册和注销 class。

```
int class_register(struct class * cls);
void class_unregister(struct class * cls);
```

下面两个函数用于注册和注销 class_device。

```
int class_device_register(struct class_device *class_dev);
void class_device_unregister(struct class_device *class_dev);
```

除了 class、class_device 结构体外，还存在一个 class_interface 结构体，当设备加入或离开类时，将引发 class_interface 中的成员函数被调用，class_interface 的定义如代码清单 5.17 所示。

代码清单 5.17　class_interface 结构体

```
struct class_interface
```

```
{
 struct list_head    node;
 struct class        *class;//对应的class
 int (*add)(struct class_device *, struct class_interface *);//设备加入时触发
 void (*remove)(struct class_device *, struct class_interface *);//设备移出时触发
};
```

下面两个函数用于注册和注销 class_interface。

```
int class_interface_register(struct class_interface *class_intf);
void class_interface_unregister(struct class_interface *class_intf);
```

6. 属性

在 bus、device、driver 和 class 层次上都分别定义了其属性结构体，包括 bus_attribute、driver_attribute、class_attribute、class_device_attribute，这几个结构体的定义在本质是完全相同的，如代码清单 5.18 所示。

华清远见

代码清单 5.18　bus_attribute/driver_attribute/class_attribute/class_device_attribute 结构体

```
/* 总线属性 */
struct bus_attribute
{
 struct attribute    attr;
 ssize_t (*show)(struct bus_type *, char * buf);
 ssize_t (*store)(struct bus_type *, const char * buf, size_t count);
};
/* 驱动属性 */
struct driver_attribute
{
 struct attribute    attr;
 ssize_t (*show)(struct device_driver *, char * buf);
 ssize_t (*store)(struct device_driver *, const char * buf, size_t count);
};
/* 类属性 */
struct class_attribute
{
 struct attribute    attr;
 ssize_t (*show)(struct class *, char * buf);
 ssize_t (*store)(struct class *, const char * buf, size_t count);
};
/* 类设备属性 */
struct class_device_attribute
{
 struct attribute    attr;
 ssize_t (*show)(struct class_device *, char * buf);
 ssize_t (*store)(struct class_device *, const char * buf, size_t count);
};
```

下面一组宏分别用于创建和初始化 bus_attribute、driver_attribute、class_attribute、class_device_ attribute。

```
BUS_ATTR(_name,_mode,_show,_store)
DRIVER_ATTR(_name,_mode,_show,_store)
CLASS_ATTR(_name,_mode,_show,_store)
CLASS_DEVICE_ATTR(_name,_mode,_show,_store)
```

下面一组函数分别用于添加和删除 bus、driver、class、class_device 属性。

```
int bus_create_file(struct bus_type * bus, struct bus_attribute * attr);
void bus_remove_file(struct bus_type * bus, struct bus_attribute * attr);

int device_create_file(struct device * dev, struct device_attribute * attr);
void device_remove_file(struct device * dev, struct device_attribute * attr);

int class_create_file(struct class * cls, const struct class_attribute * attr);
void class_remove_file(struct class * cls, const struct class_attribute * attr);

int class_device_create_file(struct class_device * class_dev,
```

```
                const struct class_device_attribute * attr);
void class_device_remove_file(struct class_device * class_dev,
                const struct class_device_attribute * attr);
```

xxx_create_file()函数中会调用 sysfs_create_file()，而 xxx_remove_file()函数中会调用 sysfs_ remove_file()函数，xxx_create_file()会创建对应的 sysfs 文件节点，而 xxx_remove_file()会删除对应的 xxx 文件节点。

华清远见

5.4.3 udev 的组成

udev 的主页位于 http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html，上面包含了 udev 的详细介绍，从 http://www.us.kernel.org/pub/linux/utils/kernel/hotplug/ 上可以下载最新的 udev 包。udev 的设计目标如下。

- 在用户空间中执行。
- 动态建立/删除设备文件。
- 允许每个人都不用关心主/次设备号。
- 提供 LSB 标准名称。
- 如果需要，可提供固定的名称。

为了提供这些功能，udev 以 3 个分割的子计划发展： namedev、libsysfs 和 udev。namedev 为设备命名子系统，libsysfs 提供访问 sysfs 文件系统从中获取信息的标准接口，udev 提供/dev 设备节点文件的动态创建和删除策略。udev 程序承担与 namedev 和 libsysfs 库交互的任务，当/sbin /hotplug 程序被内核调用时，udev 将被运行。udev 的工作过程如下。

（1） 当内核检测到在系统中出现了新设备后，内核会在 sysfs 文件系统中为该新设备生成新的记录并导出一些设备特定的信息及所发生的事件。

（2） udev 获取内核导出的信息，它调用 namedev 决定应该给该设备指定的名称，如果是新插入设备，udev 将调用 libsysfs 决定应该为该设备的设备文件指定的主/次设备号，并用分析获得的设备名称和主/次设备号创建/dev 中的设备文件；如果是设备移除，则之前已经被创建的/dev 文件将被删除。

在 namedev 中使用 5 个步骤来决定指定设备的命名。

（1）标签（label）/序号（serial）：这一步检查设备是否有惟一的识别记号，例如 USB 设备有惟一的 USB 序号，SCSI 有惟一的 UUID。如果 namedev 找到与这种惟一编号相对应的规则，它将使用该规则提供的名称。

（2）设备总线号：这一步会检查总线设备编号，对于不可热插拔的环境，这一步足以辨别设备。例如，PCI 总线编号在系统的使用期间内很少变更。如果 namedev 找到相对应的规则，规则中的名称就会被使用。

（3）总线上的拓扑：当设备在总线上的位置匹配用户指定的规则时，就会使用该规则指定的名称。

（4）替换名称：当内核提供的名称匹配指定的替代字符串时，就会使用替代字符串指定的名称。

（5）内核提供的名称：如果以前的几个步骤都没有被提供，缺省的内核将被指定给该设备。

代码清单 5.19 给出了一个 namedev 命名规则的例子，第 2、4 行定义的是符合第 1 步的规则，第 6、8 行定义的是符合第 2 步的规则，第 11、14 行定义的是符合第 3 步的规则，第 16 行定义的是符合第 4 步的规则。

代码清单 5.19　namedev 命名规则

```
1  # USB Epson printer to be called lp_epson
2  LABEL, BUS="usb", serial="HXOLL0012202323480", NAME="lp_epson"
```

```
3  # USB HP printer to be called lp_hp,
4  LABEL, BUS="usb", serial="W09090207101241330", NAME="lp_hp"
5  # sound card with PCI bus id 00:0b.0 to be the first sound card
6  NUMBER, BUS="pci", id="00:0b.0", NAME="dsp"
7  # sound card with PCI bus id 00:07.1 to be the second sound card
8  NUMBER, BUS="pci", id="00:07.1", NAME="dsp1"
9  # USB mouse plugged into the third port of the first hub to be
10 # called mouse0
11 TOPOLOGY, BUS="usb", place="1.3", NAME="mouse0"
12 # USB tablet plugged into the second port of the second hub to be
13 # called mouse1
14 TOPOLOGY, BUS="usb", place="2.2", NAME="mouse1"
15 # ttyUSB1 should always be called visor
16 REPLACE, KERNEL="ttyUSB1", NAME="visor"
```

5.4.4 udev 规则文件

udev 规则文件以行为单位，以“#”开头的行代表注释行。其余的每一行代表一个规则。每个规则分成一个或多个匹配和赋值部分。匹配部分用匹配专用的关键字来表示，相应的赋值部分用赋值专用的关键字来表示。

匹配关键字包括：ACTION（用于匹配行为）、KERNEL（用于匹配内核设备名）、BUS（用于匹配总线类型）、SYSFS（用于匹配从 sysfs 得到的信息，比如 label、vendor、USB 序列号）、SUBSYSTEM（匹配子系统名）等，赋值关键字包括：NAME（创建的设备文件名）、SYMLINK（符号创建链接名）、OWNER（设置设备的所有者）、GROUP（设置设备的组）、IMPORT（调用外部程序）等。

例如如下规则：

```
SUBSYSTEM=="net", ACTION=="add", SYSFS{address}=="00:0d:87:f6:59:f3",
IMPORT="/sbin/ rename_netiface %k eth0"
```

其中的匹配部分有 3 项，分别是 SUBSYSTEM、ACTION 和 SYSFS。而赋值部分有一项，是 IMPORT。这个规则的意思是：当系统中出现的新硬件属于 net 子系统范畴，系统对该硬件采取的动作是加入这个硬件，且这个硬件在 sysfs 文件系统中的“address”信息等于“00:0d:87:f6:59:f3”时，对这个硬件在 udev 层次施行的动作是调用外部程序/sbin/rename_netiface，并给该程序传递两个参数，一个是“%k”，代表内核对该新设备定义的名称，另一个是“eth0”。

通过一个简单的例子可以看出 udev 和 devfs 在命名方面的差异。如果系统中有两个 USB 打印机，一个被称为/dev/usb/lp0，则另外一个便是/dev/usb/lp1。但是哪个文件对应哪个打印机是无法确定的，lp0、lp1 和实际的设备没有一一对应的关系，映射关系会因为设备发现的顺序、打印机本身关闭等原因而不确定。因此，理想的方式是两个打印机应该采用基于它们的序列号或者其他标识信息的办法来进行确定的映射，devfs 无法做到这一点，udev 却可以做到。使用如下规则：

```
BUS="usb",        SYSFS{serial}="HXOLL0012202323480",        NAME="lp_epson",
SYMLINK="printers/ epson _stylus"
```

该规则中的匹配项目有 BUS 和 SYSFS，赋值项目为 NAME 和 SYMLINK，它意味着当一台 USB 打印机的序列号为“HXOLL0012202323480”时，创建/dev/lp_epson 文件，并同时创建一个符号链接/dev/printers/epson_styles。序列号为“HXOLL0012202323480”的 USB 打印机不管何时被插入，对应的设备名都是

/dev/lp_epson，而 devfs 显然无法实现设备的这种固定命名。

udev 规则的写法非常灵活，在匹配部分，可以通过“*”、“?”、[a-c]、[1-9]等 shell 通配符来灵活匹配多个项目。“*”类似于 shell 中的“*”通配符，代替任意长度的任意字符串，“？”代替一个字符，[x-y]是访问定义。此外，“%k”就是 KERNEL，“%n”则是设备的 KERNEL 序号（如存储设备的分区号）。

可以借助 udev 中的 udevinfo 工具查找规则文件可以利用的信息，如运行“udevinfo -a -p /sys/ block/sda”命令将得到以下信息：

```
    Udevinfo starts with the device specified by the devpath and then
    walks up the chain of parent devices. It prints for every device
    found, all possible attributes in the udev rules key format.
    A rule to match, can be composed by the attributes of the device
    and the attributes from one single parent device.

      looking at device '/block/sda':
        KERNEL=="sda"
        SUBSYSTEM=="block"
        DRIVER==""
        ATTR{stat}==""    1689     3169    85746    24000     2017     2095
32896    47292
        0    23188    71292"
        ATTR{size}=="6291456"
        ATTR{removable}=="0"
        ATTR{range}=="16"
        ATTR{dev}=="8:0"

      looking                at                parent                device
'/devices/platform/host0/target0:0:0/0:0:0:0':
        KERNELS=="0:0:0:0"
        SUBSYSTEMS=="scsi"
        DRIVERS=="sd"
        ATTRS{ioerr_cnt}=="0x5"
        ATTRS{iodone_cnt}=="0xe86"
        ATTRS{iorequest_cnt}=="0xe86"
        ATTRS{iocounterbits}=="32"
        ATTRS{timeout}=="30"
        ATTRS{state}=="running"
        ATTRS{rev}=="1.0 "
        ATTRS{model}=="VMware Virtual S"
        ATTRS{vendor}=="VMware, "
        ATTRS{scsi_level}=="3"
        ATTRS{type}=="0"
        ATTRS{queue_type}=="none"
        ATTRS{queue_depth}=="3"
        ATTRS{device_blocked}=="0"

      looking at parent device '/devices/platform/host0/target0:0:0':
        KERNELS=="target0:0:0"
        SUBSYSTEMS==""
        DRIVERS==""

      looking at parent device '/devices/platform/host0':
        KERNELS=="host0"
```

```
    SUBSYSTEMS==""
    DRIVERS==""

  looking at parent device '/devices/platform':
    KERNELS=="platform"
    SUBSYSTEMS==""
    DRIVERS==""
```

5.4.5 创建和配置 udev

udev 包括 udev、udevcontrol、udevd、udevsend、udevmonitor、udevsettle、udevstart、udevinfo、udevtest 和 udevtrigger 等处于用户空间的程序。在嵌入式系统中，只需要 udevd 和 udevstart 就能使 udev 工作。我们可以按照下面的步骤来生成和配置 udev。

（1）从 http://www.us.kernel.org/pub/linux/utils/kernel/hotplug/下载 udev 程序，如笔者下载的是 udev-114.tar.gz。

（2）运行“tar zxvf udev-114.tar.gz”命令解压缩 udev 程序包。

（3）运行 make 编译 udev，当前目录下会生成 test-udev、udevcontrol、udevd、udevinfo、udevmonitor、udevsettle、udevstart、udevtest 和 udevtrigger 共 9 个工具程序。

（4）把第（3）步生成的工具程序复制到/sbin 目录，同时把解压缩 udev-114.tar.gz 后获得的 etc 目录下的 udev 目录复制到系统的/etc 下，etc/udev 下包含 udev.conf 配置文件、rules.d 目录（该目录中的文件给出了设备规则）等。

（5）编写完成启动、停止、重新启动等工作的 udev 脚本，代码清单 5.20 给出了一个范例。

代码清单 5.20　udev 运行脚本范例

```
#!/bin/sh -e
# udev 初始脚本

# 检查包是否已经被安装
[ -x /sbin/udevd ] || exit 0

case "$1" in
   start)
  # 需要 2.6.15 中引入的 uevent 的支持，如果系统中没有，使用静态/dev
  if [ ! -f /sys/class/mem/null/uevent ]; then
     if mountpoint -q /dev; then
      umount -l /dev/.static/dev
      umount -l /dev
     fi
     exit 1
  fi

  if ! mountpoint -q /dev; then
     # initramfs 没有挂载/dev，因此，这里需要完成
     mount -n --bind /dev /etc/udev
     mount -n -t tmpfs -o mode=0755 udev /dev
     mkdir -m 0700 -p /dev/.static/dev
     mount -n --move /etc/udev /dev/.static/dev
  fi

  # 复制默认的设备树
```

```
cp -a -f /lib/udev/devices/* /dev

# 现在全部通过 netlink 进行，所以之前的热插拔程序不再使用
if [ -e /proc/sys/kernel/hotplug ]; then
    echo "" > /proc/sys/kernel/hotplug
fi

# 开始 udevd
log_begin_msg "Starting kernel event manager..."
if start-stop-daemon --start --quiet --exec /sbin/udevd -- --daemon; then
    log_end_msg 0
else
    log_end_msg $?
fi

# 打开 udevtrigger 的事务日志
/sbin/udevmonitor -e >/dev/.udev.log &
UDEV_MONITOR_PID=$!

# 弥补丢失的事件
log_begin_msg "Loading hardware drivers..."
/sbin/udevtrigger
if /sbin/udevsettle; then
    log_end_msg 0
else
    log_end_msg $?
fi

# 杀死 udevmonitor
kill $UDEV_MONITOR_PID
;;
  stop)
log_begin_msg "Stopping kernel event manager..."
if start-stop-daemon --stop --quiet --oknodo --exec /sbin/udevd; then
    log_end_msg 0
else
    log_end_msg $?
fi
```

```
    umount -l /dev/.static/dev
    umount -l /dev
    ;;
      restart)
    cp -au /lib/udev/devices/* /dev

    log_begin_msg "Loading additional hardware drivers..."
    /sbin/udevtrigger
    if /sbin/udevsettle; then
        log_end_msg 0
    else
        log_end_msg $?
    fi
    ;;
      reload|force-reload)
    log_begin_msg "Reloading kernel event manager..."
    if start-stop-daemon --stop --signal 1 --exec /sbin/udevd; then
        log_end_msg 0
    else
        log_end_msg $?
    fi
    ;;
  *)
    echo                    "Usage:                    /etc/init.d/udev
{start|stop|restart|reload|force-reload}"
    exit 1
    ;;
  esac

  exit 0
```

如果已经知道了设备对应的 sysfs 路径，则可以使用 udevtest 工具测试 udev 对该设备所采取的行动，这可以帮助进行 udev 规则的调试。如运行“udevtest /sys/class/tty/ttys0”命令的结果为：

```
    This program is for debugging only, it does not run any program,
    specified by a RUN key. It may show incorrect results, because
    some values may be different, or not available at a simulation run.

    parse_file: reading '/etc/udev/rules.d/05-udev-early.rules' as rules
file
    parse_file: reading '/etc/udev/rules.d/60-persistent-input.rules' as
```

```
rules file
    parse_file: reading '/etc/udev/rules.d/60-persistent-storage.rules'
as rules file
    parse_file: reading '/etc/udev/rules.d/95-udev-late.rules' as rules
file
    main: looking at device '/class/tty/ttys0' from subsystem 'tty'
    udev_rules_get_name: no node name set, will use kernel name ''
    udev_db_get_device: found a symlink as db file
    udev_device_event: device '/class/tty/ttys0' already in database,
cleanup
    udev_node_add: creating device node '/dev/ttys0', major=3, minor=48,
mode=0660, uid=0, gid=0
    main: run: 'socket:/org/kernel/udev/monitor'
```

从中可以看出 udev 将为与/sys/class/tty/ttys0 对应的设备使用内核中的名称通过 udev_node_ add 创建/dev/ttys0 设备文件，主、次设备号分别为 3 和 48。

5.5 总结

Linux 系统用户空间的文件编程有两种方法，即通过 Linux API 和通过 C 库函数访问文件。用户空间看不到设备驱动，能看到的只有设备对应的文件，因此文件编程即是用户空间的设备编程。

Linux 系统按照功能对文件系统的目录结构进行了良好的规划。/dev 是设备文件的存放目录，devfs 和 udev 分别是 Linux 2.4 设备和 Linux 2.6 设备生成设备文件节点的方法，前者运行于内核空间，后者运行于用户空间。

Linux 2.6 设备通过一系列数据结构定义了设备模型，设备模型与 sysfs 文件系统中的目录和文件存在一种对应关系，udev 可以利用 sysfs 中记录的信息定义规则并提取主次设备号动态创建/dev 设备文件节点。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm

第 6 章 字符设备驱动

在 Linux 设备驱动中，字符设备驱动较为基础。本章主要讲解 Linux 字符设备驱动程序的结构及其主要组成部分的编程方法。

6.1 节讲解了 Linux 字符设备驱动的关键数据结构 cdev 及 file_operations 结构体的操作方法，并分析了 Linux 字符设备的整体结构，给出了简单的设计模板。

6.2 节讲解了本章及后续各章所基于的 globalmem 虚拟字符设备，第 6～9 章都将基于该虚拟设备实例进行字符设备驱动及并发控制等知识的讲解。

6.3 节依据 6.1 节的知识讲解 globalmem 设备的驱动编写方法，对读写函数、seek() 函数和 I/O 控制函数等进行了重点分析。该节的最后改造 globalmem 的驱动程序以利用文件私有数据。

6.4 节给出了 6.3 节的 globalmem 设备驱动在用户空间的验证。

华清远见

6.1 Linux 字符设备驱动结构

6.1.1 cdev 结构体

在 Linux 2.6 内核中使用 cdev 结构体描述字符设备，cdev 结构体的定义如代码清单 6.1 所示。

代码清单 6.1 cdev 结构体

```
struct cdev
{
 struct kobject kobj; /* 内嵌的 kobject 对象 */
 struct module *owner;   /*所属模块*/
 struct file_operations *ops;  /*文件操作结构体*/
 struct list_head list;
 dev_t dev;          /*设备号*/
 unsigned int count;
};
```

cdev 结构体的 dev_t 成员定义了设备号，为 32 位，其中高 12 位为主设备号，低 20 位为次设备号。使用下列宏可以从 dev_t 获得主设备号和次设备号。

```
MAJOR(dev_t dev)
MINOR(dev_t dev)
```

而使用下列宏则可以通过主设备号和设备号生成 dev_t。

```
MKDEV(int major, int minor)
```

cdev 结构体的另一个重要成员 file_operations 定义了字符设备驱动提供给虚拟文件系统的接口函数。

Linux 2.6 内核提供了一组函数用于操作 cdev 结构体，如下所示：

```
void cdev_init(struct cdev *, struct file_operations *);
struct cdev *cdev_alloc(void);
void cdev_put(struct cdev *p);
int cdev_add(struct cdev *, dev_t, unsigned);
void cdev_del(struct cdev *);
```

cdev_init()函数用于初始化 cdev 的成员，并建立 cdev 和 file_operations 之间的连接，其源代码如代码清单 6.2 所示。

代码清单 6.2 cdev_init()函数

```
void cdev_init(struct cdev *cdev, struct file_operations *fops)
{
 memset(cdev, 0, sizeof *cdev);
 INIT_LIST_HEAD(&cdev->list);
 cdev->kobj.ktype = &ktype_cdev_default;
 kobject_init(&cdev->kobj);
 cdev->ops = fops;   /*将传入的文件操作结构体指针赋值给 cdev 的 ops*/
}
```

cdev_alloc()函数用于动态申请一个 cdev 内存，其源代码如代码清单 6.3 所示。

代码清单 6.3　cdev_alloc()函数

```
struct cdev *cdev_alloc(void)
{
  struct cdev *p=kmalloc(sizeof(struct cdev),GFP_KERNEL); /*分配 cdev 的内存*/
  if (p) {
      memset(p, 0, sizeof(struct cdev));
      p->kobj.ktype = &ktype_cdev_dynamic;
      INIT_LIST_HEAD(&p->list);
      kobject_init(&p->kobj);
  }
  return p;
}
```

cdev_add()函数和 cdev_del()函数分别向系统添加和删除一个 cdev，完成字符设备的注册和注销。对 cdev_add()的调用通常发生在字符设备驱动模块加载函数中，而对 cdev_del()函数的调用则通常发生在字符设备驱动模块卸载函数中。

6.1.2　分配和释放设备号

在调用 cdev_add()函数向系统注册字符设备之前，应首先调用 register_chrdev_region()或 alloc_chrdev_region()函数向系统申请设备号，这两个函数的原型如下：

```
int register_chrdev_region(dev_t from, unsigned count, const char *name);
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,
                const char *name);
```

register_chrdev_region()函数用于已知起始设备的设备号的情况；而 alloc_chrdev_region()用于设备号未知，向系统动态申请未被占用的设备号的情况。函数调用成功之后，会把得到的设备号放入第一个参数 dev 中。alloc_chrdev_region()与 register_chrdev_region()对比的优点在于它会自动避开设备号重复的冲突。

相反地，在调用 cdev_del()函数从系统注销字符设备之后，unregister_chrdev_region()应该被调用以释放原先申请的设备号，这个函数的原型如下：

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

6.1.3　file_operations 结构体

file_operations 结构体中的成员函数是字符设备驱动程序设计的主体内容，这些函数实际会在应用程序进行 Linux 的 open()、write()、read()、close()等系统调用时最终被调用。file_operations 结构体目前已经比较庞大，它的定义如代码清单 6.4 所示。

代码清单 6.4　file_operations 结构体

```
struct file_operations
{
  struct module *owner;
    // 拥有该结构的模块的指针，一般为 THIS_MODULES
  loff_t(*llseek)(struct file *, loff_t, int);
    // 用来修改文件当前的读写位置
  ssize_t(*read)(struct file *, char __user *, size_t, loff_t*);
    // 从设备中同步读取数据
  ssize_t(*aio_read)(struct kiocb *, char __user *, size_t, loff_t);
    // 初始化一个异步的读取操作
```

```
    ssize_t(*write)(struct file *, const char __user *, size_t, loff_t*);
    // 向设备发送数据
    ssize_t(*aio_write)(struct kiocb *, const char __user *, size_t, loff_t);
    // 初始化一个异步的写入操作
    int(*readdir)(struct file *, void *, filldir_t);
    // 仅用于读取目录，对于设备文件，该字段为 NULL
    unsigned int(*poll)(struct file *, struct poll_table_struct*);
    // 轮询函数，判断目前是否可以进行非阻塞的读取或写入
    int(*ioctl)(struct inode *, struct file *, unsigned int, unsigned long);
    // 执行设备 I/O 控制命令
    long(*unlocked_ioctl)(struct file *, unsigned int, unsigned long);
    // 不使用 BLK 文件系统，将使用此种函数指针代替 ioctl
    long(*compat_ioctl)(struct file *, unsigned int, unsigned long);
    // 在 64 位系统上，32 位的 ioctl 调用将使用此函数指针代替
    int(*mmap)(struct file *, struct vm_area_struct*);
    // 用于请求将设备内存映射到进程地址空间
    int(*open)(struct inode *, struct file*);
    // 打开
    int(*flush)(struct file*);
    int(*release)(struct inode *, struct file*);
    // 关闭
    int(*synch)(struct file *, struct dentry *, int datasync);
    // 刷新待处理的数据
    int(*aio_fsync)(struct kiocb *, int datasync);
    // 异步 fsync
    int(*fasync)(int, struct file *, int);
    // 通知设备 FASYNC 标志发生变化
    int(*lock)(struct file *, int, struct file_lock*);
    ssize_t(*readv)(struct file *, const struct iovec *, unsigned long, loff_t*);
    ssize_t(*writev)(struct file *, const struct iovec *, unsigned long, loff_t*);
    // readv 和 writev: 分散/聚集型的读写操作
    ssize_t(*sendfile)(struct file *, loff_t *, size_t, read_actor_t, void*);
    // 通常为 NULL
    ssize_t(*sendpage)(struct file *, struct page *, int, size_t, loff_t *, int);
    // 通常为 NULL
    unsigned long(*get_unmapped_area)(struct file *,unsigned long, unsigned long,
    unsigned long, unsigned long);
    // 在进程地址空间找到一个将底层设备中的内存段映射的位置
    int(*check_flags)(int);
    // 允许模块检查传递给 fcntl(F_SETEL...)调用的标志
    int(*dir_notify)(struct file *filp, unsigned long arg);
    // 仅对文件系统有效，驱动程序不必实现
    int(*flock)(struct file *, int, struct file_lock*);
};
```

下面对 file_operations 结构体中的主要成员进行讲解。

llseek()函数用来修改一个文件的当前读写位置，并将新位置返回，在出错时，这个函数返回一个负值。

read()函数用来从设备中读取数据，成功时函数返回读取的字节数，出错时返回一个负值。

write()函数向设备发送数据，成功时该函数返回写入的字节数。如果此函数未被实现，当用户进行 write()系统调用时，将得到-EINVAL 返回值。

readdir()函数仅用于目录，设备节点不需要实现它。

ioctl()提供设备相关控制命令的实现（既不是读操作也不是写操作），当调用成功时，返回给调用程序一个非负值。内核本身识别部分控制命令，而不必调用设备驱动中的 ioctl()。如果设备不提供 ioctl()函数，对于内核不能识别的命令，用户进行 ioctl()系统调用时将获得-EINVAL 返回值。

mmap()函数将设备内存映射到进程内存中，如果设备驱动未实现此函数，用户进行 mmap()系统调用时将获得-ENODEV 返回值。这个函数对于帧缓冲等设备特别有意义。

当用户空间调用 Linux API 函数 open()打开设备文件时，设备驱动的 open()函数最终被调用。驱动程序可以不实现这个函数，在这种情况下，设备的打开操作永远成功。与 open()函数对应的是 release()函数。

poll()函数一般用于询问设备是否可被非阻塞地立即读写。当询问的条件未触发时，用户空间进行 select()和 poll()系统调用将引起进程的阻塞。

aio_read()和 aio_write()函数分别对与文件描述符对应的设备进行异步读、写操作。设备实现这两个函数后，用户空间可以对该设备文件描述符调用 aio_read()、aio_write()等系统调用进行读写。

6.1.4 Linux 字符设备驱动的组成

在 Linux 系统中，字符设备驱动由如下几个部分组成。

1. 字符设备驱动模块加载与卸载函数

在字符设备驱动模块加载函数中应该实现设备号的申请和 cdev 的注册，而在卸载函数中应实现设备号的释放和 cdev 的注销。

工程师通常习惯将设备定义为一个设备相关的结构体，其包含该设备所涉及的 cdev、私有数据及信号量等信息。常见的设备结构体、模块加载和卸载函数形式如代码清单 6.5 所示。

代码清单 6.5　字符设备驱动模块加载与卸载函数模板

```
//设备结构体
struct xxx_dev_t
{
  struct cdev cdev;
  ...
} xxx_dev;
//设备驱动模块加载函数
static int __init xxx_init(void)
{
  ...
  cdev_init(&xxx_dev.cdev, &xxx_fops); //初始化 cdev
  xxx_dev.cdev.owner = THIS_MODULE;
  //获取字符设备号
  if (xxx_major)
  {
    register_chrdev_region(xxx_dev_no, 1, DEV_NAME);
```

```
17   }
18   else
19   {
20     alloc_chrdev_region(&xxx_dev_no, 0, 1, DEV_NAME);
21   }
22
23   ret = cdev_add(&xxx_dev.cdev, xxx_dev_no, 1); //注册设备
24   ...
25 }
26 /*设备驱动模块卸载函数*/
27 static void __exit xxx_exit(void)
28 {
29  unregister_chrdev_region(xxx_dev_no, 1); //释放占用的设备号
30  cdev_del(&xxx_dev.cdev); //注销设备
31  ...
32 }
```

2．字符设备驱动的 file_operations 结构体中成员函数

file_operations 结构体中成员函数是字符设备驱动与内核的接口，是用户空间对 Linux 进行系统调用最终的落实者。大多数字符设备驱动会实现 read()、write()和 ioctl()函数，常见的字符设备驱动的这 3 个函数的形式如代码清单 6.6 所示。

代码清单 6.6　字符设备驱动读、写、I/O 控制函数模板

```
1   /* 读设备*/
2   ssize_t xxx_read(struct file *filp, char __user *buf, size_t count,
3     loff_t*f_pos)
4   {
5     ...
6     copy_to_user(buf, ..., ...);
7     ...
8   }
9   /* 写设备*/
10  ssize_t xxx_write(struct file *filp, const char __user *buf, size_t
count,
11    loff_t *f_pos)
12  {
13    ...
14    copy_from_user(..., buf, ...);
15    ...
16  }
17  /* ioctl 函数 */
18  int xxx_ioctl(struct inode *inode, struct file *filp, unsigned int
cmd,
19    unsigned long arg)
20  {
21    ...
22    switch (cmd)
23    {
24      case XXX_CMD1:
25        ...
26        break;
27      case XXX_CMD2:
28        ...
29        break;
30      default:
```

```
        /* 不能支持的命令 */
        return  - ENOTTY;
    }
    return 0;
}
```

设备驱动的读函数中，filp 是文件结构体指针，buf 是用户空间内存的地址，该地址在内核空间不能直接读写，count 是要读的字节数，f_pos 是读的位置相对于文件开头的偏移。

设备驱动的写函数中，filp 是文件结构体指针，buf 是用户空间内存的地址，该地址在内核空间不能直接读写，count 是要写的字节数，f_pos 是写的位置相对于文件开头的偏移。

由于内核空间与用户空间的内存不能直接互访，因此借助函数 copy_from_user()完成用户空间到内核空间的复制，函数 copy_to_user()完成内核空间到用户空间的复制。

copy_from_user()和 copy_to_user()的原型如下所示：

```
unsigned long copy_from_user(void *to, const void __user *from, unsigned long count);
unsigned long copy_to_user(void __user *to, const void *from, unsigned long count);
```

上述函数均返回不能被复制的字节数，因此，如果完全复制成功，返回值为 0。

如果要复制的内存是简单类型，如 char、int、long 等，则可以使用简单的 put_user()和 get_user()，如下所示：

```
int val; //内核空间整型变量
...
get_user(val, (int *) arg); //用户空间到内核空间，arg 是用户空间的地址
...
put_user(val, (int *) arg); //内核空间到用户空间，arg 是用户空间的地址
```

读和写函数中的__user 是一个宏，表明其后的指针指向用户空间，这个宏定义如下：

```
#ifdef __CHECKER__
# define __user    __attribute__((noderef, address_space(1)))
#else
# define __user
#endif
```

I/O 控制函数的 cmd 参数为事先定义的 I/O 控制命令，而 arg 为对应于该命令的参数。例如对于串行设备，如果 SET_BAUDRATE 是一个设置波特率的命令，那后面的 arg 就应该是波特率值。

在字符设备驱动中，需要定义一个 file_operations 的实例，并将具体设备驱动的函数赋值给 file_operations 的成员，如代码清单 6.7 所示。

代码清单 6.7　字符设备驱动文件操作结构体模板

```
struct file_operations xxx_fops =
{
  .owner = THIS_MODULE,
  .read = xxx_read,
  .write = xxx_write,
  .ioctl = xxx_ioctl,
  ...
};
```

上述 xxx_fops 在代码清单 6.5 第 11 行的 cdev_init（&xxx_dev.cdev, &xxx_fops)的语句中被建立与 cdev 的连接。

图 6.1 所示为字符设备驱动的结构、字符设备驱动与字符设备以及字符设备驱动与用户空间访问该设备的程序之间的关系。

图 6.1 字符设备驱动的结构

6.2 globalmem 虚拟设备实例描述

从本章开始，后续的几章都将基于虚拟的 globalmem 设备进行字符设备驱动的讲解。globalmem 意味着“全局内存”，在 globalmem 字符设备驱动中会分配一片大小为 GLOBALMEM_ SIZE（4KB）的内存空间，并在驱动中提供针对该片内存的读写、控制和定位函数，以供用户空间的进程能通过 Linux 系统调用访问这片内存。

本章将给出 globalmem 设备驱动的雏形，而后续章节会在这个雏形的基础上添加并发与同步控制等复杂功能。

6.3 globalmem 设备驱动

6.3.1 头文件、宏及设备结构体

在 globalmem 字符设备驱动中，应包含它要使用的头文件，并定义 globalmem 设备结构体及相关宏。

代码清单 6.8 globalmem 设备结构体和宏

```
1  #include <linux/module.h>
```

```
2  #include <linux/types.h>
3  #include <linux/fs.h>
4  #include <linux/errno.h>
5  #include <linux/mm.h>
6  #include <linux/sched.h>
7  #include <linux/init.h>
8  #include <linux/cdev.h>
9  #include <asm/io.h>
10 #include <asm/system.h>
11 #include <asm/uaccess.h>
12
13 #define GLOBALMEM_SIZE  0x1000  /*全局内存大小: 4KB*/
14 #define MEM_CLEAR 0x1  /*清零全局内存*/
15 #define GLOBALMEM_MAJOR 254   /*预设的globalmem的主设备号*/
16
17 static globalmem_major = GLOBALMEM_MAJOR;
18 /*globalmem设备结构体*/
19 struct globalmem_dev
20 {
21   struct cdev cdev; /*cdev结构体*/
22   unsigned char mem[GLOBALMEM_SIZE]; /*全局内存*/
23 };
24
25 struct globalmem_dev dev; /*设备结构体实例*/
```

从第 19～23 行代码可以看出，定义的 globalmem_dev 设备结构体包含了对应于 globalmem 字符设备的 cdev、使用的内存 mem[GLOBALMEM_SIZE]。当然，程序中并不一定要把 mem[GLOBA- LMEM_SIZE]和 cdev 包含在一个设备结构体中，但这样定义的好处在于它借用了面向对象程序设计中“封装”的思想，体现了一种良好的编程习惯。

6.3.2 加载与卸载设备驱动

globalmem 设备驱动的模块加载和卸载函数遵循代码清单 6.5 的类似模板，其实现的功能与代码清单 6.5 完全一致，如代码清单 6.9 所示。

代码清单 6.9 globalmem 设备驱动模块加载与卸载函数

```
1  /*globalmem设备驱动模块加载函数*/
2  int globalmem_init(void)
3  {
4    int result;
5    dev_t devno = MKDEV(globalmem_major, 0);
6
7    /* 申请字符设备驱动区域*/
8    if (globalmem_major)
9      result = register_chrdev_region(devno, 1, "globalmem");
```

```
10  else
11   /* 动态获得主设备号 */
12  {
13    result = alloc_chrdev_region(&devno, 0, 1, "globalmem");
14    globalmem_major = MAJOR(devno);
15  }
16  if (result < 0)
17    return result;
18
19  globalmem_setup_cdev();
20  return 0;
21 }
22
23 /*globalmem 设备驱动模块卸载函数*/
24 void globalmem_exit(void)
25 {
26  cdev_del(&dev.cdev); /*删除 cdev 结构*/
27  unregister_chrdev_region(MKDEV(globalmem_major, 0), 1);/*注销设备区域*/
28 }
```

第 19 行调用的 globalmem_setup_cdev()函数完成 cdev 的初始化和添加，如代码清单 6.10 所示。

代码清单 6.10　初始化并添加 cdev 结构体

```
1  /*初始化并添加 cdev 结构体*/
2  static void globalmem_setup_cdev()
3  {
4    int err, devno = MKDEV(globalmem_major, 0);
5
6    cdev_init(&dev.cdev, &globalmem_fops);
7    dev.cdev.owner = THIS_MODULE;
8    dev.cdev.ops = &globalmem_fops;
9    err = cdev_add(&dev.cdev, devno, 1);
10   if (err)
11     printk(KERN_NOTICE "Error %d adding globalmem", err);
12 }
```

在 cdev_init()函数中，与 globalmem 的 cdev 关联的 file_operations 结构体如代码清单 6.11 所示。

代码清单 6.11　globalmem 设备驱动文件操作结构体

```
1  static const struct file_operations globalmem_fops =
2  {
3    .owner = THIS_MODULE,
4    .llseek = globalmem_llseek,
5    .read = globalmem_read,
6    .write = globalmem_write,
7    .ioctl = globalmem_ioctl,
8  };
```

6.3.3　读写函数

globalmem 设备驱动的读写函数主要是让设备结构体的 mem[]数组与用户空间

交互数据，并随着访问的字节数变更返回用户的文件读写偏移位置。读和写函数的实现分别如代码清单 6.12 和代码清单 6.13 所示。

代码清单 6.12　globalmem 设备驱动的读函数

```
static ssize_t globalmem_read(struct file *filp, char __user *buf, size_t count,
  loff_t *ppos)
{
  unsigned long p =  *ppos;
  int ret = 0;

  /*分析和获取有效的读长度*/
  if (p >= GLOBALMEM_SIZE) //要读的偏移位置越界
    return count ?  - ENXIO: 0;
  if (count > GLOBALMEM_SIZE - p)//要读的字节数太大
    count = GLOBALMEM_SIZE - p;

  /*内核空间→用户空间*/
  if (copy_to_user(buf, (void*)(dev.mem + p), count))
  {
    ret =  - EFAULT;
  }
  else
  {
    *ppos += count;
    ret = count;

    printk(KERN_INFO "read %d bytes(s) from %d\n", count, p);
  }

  return ret;
}
```

代码清单 6.13　globalmem 设备驱动的写函数

```
static ssize_t globalmem_write(struct file *filp, const char __user *buf,
  size_t count, loff_t *ppos)
{
  unsigned long p =  *ppos;
  int ret = 0;

```

```
  /*分析和获取有效的写长度*/
  if (p >= GLOBALMEM_SIZE)   //要写的偏移位置越界
    return count ?  - ENXIO: 0;
  if (count > GLOBALMEM_SIZE - p) //要写的字节数太多
    count = GLOBALMEM_SIZE - p;

  /*用户空间→内核空间*/
  if (copy_from_user(dev->mem + p, buf, count))
    ret =  - EFAULT;
  else
  {
    *ppos += count;
    ret = count;

    printk(KERN_INFO "written %d bytes(s) from %d\n", count, p);
  }

  return ret;
}
```

6.3.4 seek()函数

seek()函数对文件定位的起始地址可以是文件开头（SEEK_SET，0）、当前位置（SEEK_CUR，1）和文件尾（SEEK_END，2），globalmem 支持从文件开头和当前位置相对偏移。

在定位的时候，应该检查用户请求的合法性，若不合法，函数返回- EINVAL，合法时返回文件的当前位置，如代码清单 6.14 所示。

代码清单 6.14　globalmem 设备驱动的 seek()函数

```
static loff_t globalmem_llseek(struct file *filp, loff_t offset, int orig)
{
  loff_t ret;
  switch (orig)
  {
    case 0:   /*从文件开头开始偏移*/
      if (offset < 0)
      {
        ret =  - EINVAL;
        break;
      }
      if ((unsigned int)offset > GLOBALMEM_SIZE) //偏移越界
```

```
    {
      ret =  - EINVAL;
      break;
    }
    filp->f_pos = (unsigned int)offset;
    ret = filp->f_pos;
    break;
  case 1:   /*从当前位置开始偏移*/
    if ((filp->f_pos + offset) > GLOBALMEM_SIZE) //偏移越界
    {
      ret =  - EINVAL;
      break;
    }
    if ((filp->f_pos + offset) < 0)
    {
      ret =  - EINVAL;
      break;
    }
    filp->f_pos += offset;
    ret = filp->f_pos;
    break;
  default:
    ret =  - EINVAL;
  }
  return ret;
}
```

6.3.5 ioctl()函数

1. globalmem 设备驱动的 ioctl()函数

globalmem 设备驱动的 ioctl()函数接受 MEM_CLEAR 命令，这个命令会将全局内存的有效数据长度清零，对于设备不支持的命令，ioctl()函数应该返回-EINVAL，如代码清单 6.15 所示。

代码清单 6.15　globalmem 设备驱动的 ioctl()函数

```
static int globalmem_ioctl(struct inode *inodep, struct file *filp, unsigned
  int cmd, unsigned long arg)
{
  switch (cmd)
  {
    case MEM_CLEAR:
      //清除全局内存
      memset(dev->mem, 0, GLOBALMEM_SIZE);
```

```
    9       printk(KERN_INFO "globalmem is set to zero\n");
    10      break;
    11
    12    default:
    13      return  - EINVAL; //其他不支持的命令
    14   }
    15   return 0;
    16 }
```

在上述程序中，MEM_CLEAR 被宏定义为 0x01，实际上并不是一种值得推荐的方法，简单地对命令定义为 0x0、0x1、0x2 等类似值会导致不同的设备驱动拥有相同的命令号。如果设备 A、B 都支持 0x0、0x1、0x2 这样的命令，假设用户本身希望给 A 发 0x1 命令，可是不经意间发给了 B，这个时候 B 因为支持该命令，它就会执行该命令。因此，Linux 内核推荐采用一套统一的 ioctl()命令生成方式。

2．ioctl()命令

Linux 系统建议以如图 6.2 所示的方式定义 ioctl()的命令码。

设备类型	序列号	方向	数据尺寸
8bit	8bit	2bit	13/14bit

图 6.2　ioctl()命令码的组成

命令码的设备类型字段为一个“幻数”，可以是 0～0xff 之间的值，内核中的 ioctl-number.txt 给出了一些推荐的和已经被使用的“幻数”，新设备驱动定义“幻数”的时候要避免与其冲突。

命令码的序列号也是 8 位宽。

命令码的方向字段为 2 位，该字段表示数据传送的方向，可能的值是_IOC_NONE（无数据传输）、_IOC_READ（读）、_IOC_WRITE（写）和_IOC_READ|_IOC_WRITE（双向）。数据传送的方向是从应用程序的角度来看的。

命令码的数据长度字段表示涉及的用户数据的大小，这个成员的宽度依赖于体系结构，通常是 13 位或者 14 位。

内核还定义了_IO()、_IOR()、_IOW()和_IOWR()这 4 个宏来辅助生成命令，这 4 个宏的通用定义如代码清单 6.16 所示。

代码清单 6.16　_IO()、_IOR()、_IOW()和_IOWR()宏定义

```
1  #define _IO(type,nr)        _IOC(_IOC_NONE,(type),(nr),0)
2  #define _IOR(type,nr,size) _IOC(_IOC_READ,(type),(nr),\
3                                  (_IOC_TYPECHECK(size)))
4  #define _IOW(type,nr,size)   _IOC(_IOC_WRITE,(type),(nr),\
5                                  (_IOC_TYPECHECK(size)))
6                              #define              _IOWR(type,nr,size)
_IOC(_IOC_READ|_IOC_WRITE,(type),(nr),  \
7                                  (_IOC_TYPECHECK(size)))
8  /*_IO、_IOR 等使用的_IOC 宏*/
9  #define _IOC(dir,type,nr,size) \
10     (((dir)  << _IOC_DIRSHIFT) | \
11     ((type) << _IOC_TYPESHIFT) | \
12     ((nr)   << _IOC_NRSHIFT) | \
```

```
13      ((size) << _IOC_SIZESHIFT))
```

由此可见，这几个宏的作用是根据传入的 type（设备类型字段）、nr（序列号字段）和 size（数据长度字段）和宏名隐含的方向字段移位组合生成命令码。

由于 globalmem 的 MEM_CLEAR 命令不涉及数据传输，因此它可以定义如下：

```
#define GLOBALMEM_MAGIC …
#define MEM_CLEAR _IO(GLOBALMEM_MAGIC,0)
```

3．预定义命令

内核中预定义了一些 I/O 控制命令，如果某设备驱动中包含了与预定义命令一样的命令，这些命令会被当作预定义命令被内核处理而不是被设备驱动处理，预定义命令有如下 4 种。

- FIOCLEX：即 File IOctl Close on Exec，对文件设置专用标志，通知内核当 exec()系统调用发生时自动关闭打开的文件。
- FIONCLEX：即 File IOctl Not CLose on Exec，与 FIOCLEX 标志相反，清除由 FIOCLEX 命令设置的标志。
- FIOQSIZE：获得一个文件或者目录的大小，当用于设备文件时，返回一个 ENOTTY 错误。
- FIONBIO：即 File IOctl Non-Blocking I/O，这个调用修改在 filp->f_flags 中的 O_NONBLOCK 标志。

FIOCLEX、FIONCLEX、FIOQSIZE 和 FIONBIO 这些宏的定义如下：

```
#define FIONCLEX        0x5450
#define FIOCLEX     0x5451
#define FIOQSIZE        0x5460
#define FIONBIO     0x5421
```

由以上定义可以看出，FIOCLEX、FIONCLEX、FIOQSIZE 和 FIONBIO 的幻数为“T”。

6.3.6 使用文件私有数据

6.3.1～6.3.5 节给出的代码完整地实现了预期的 globalmem 雏形，在其代码中，为 globalmem 设备结构体 globalmem_dev 定义了全局实例 dev（见代码清单 6.8 第 25 行），而 globalmem 的驱动中 read()、write()、ioctl()、llseek()函数都针对 dev 进行操作。

实际上，大多数 Linux 驱动工程师都将文件的私有数据 private_data 指向设备结构体，read()、write()、ioctl()、llseek()等函数通过 private_data 访问设备结构体。

下面对各函数进行少量的修改，使读者了解字符设备驱动的全貌，代码清单 6.17 列出了完整的使用文件私有数据的 globalmem 的设备驱动。

代码清单 6.17　使用文件私有数据的 globalmem 的设备驱动

```
1   #include <linux/module.h>
2   #include <linux/types.h>
3   #include <linux/fs.h>
4   #include <linux/errno.h>
5   #include <linux/mm.h>
6   #include <linux/sched.h>
```

```
7   #include <linux/init.h>
8   #include <linux/cdev.h>
9   #include <asm/io.h>
10  #include <asm/system.h>
11  #include <asm/uaccess.h>
12
13  #define GLOBALMEM_SIZE  0x1000  /*全局内存最大 4KB*/
14  #define MEM_CLEAR 0x1  /*清零全局内存*/
15  #define GLOBALMEM_MAJOR 254   /*预设的 globalmem 的主设备号*/
16
17  static globalmem_major = GLOBALMEM_MAJOR;
18  /*globalmem 设备结构体*/
19  struct globalmem_dev
20  {
21    struct cdev cdev; /*cdev 结构体*/
22    unsigned char mem[GLOBALMEM_SIZE]; /*全局内存*/
23  };
24
25  struct globalmem_dev *globalmem_devp; /*设备结构体指针*/
26  /*文件打开函数*/
27  int globalmem_open(struct inode *inode, struct file *filp)
28  {
29    /*将设备结构体指针赋值给文件私有数据指针*/
30    filp->private_data = globalmem_devp;
31    return 0;
32  }
33  /*文件释放函数*/
34  int globalmem_release(struct inode *inode, struct file *filp)
35  {
36    return 0;
37  }
38
39  /* ioctl 设备控制函数 */
40  static int globalmem_ioctl(struct inode *inodep, struct file *filp,
unsigned
41    int cmd, unsigned long arg)
42  {
43    struct globalmem_dev *dev = filp->private_data;/*获得设备结构体指
针*/
44
```

```
  switch (cmd)
  {
    case MEM_CLEAR:
      memset(dev->mem, 0, GLOBALMEM_SIZE);
      printk(KERN_INFO "globalmem is set to zero\n");
      break;

    default:
      return  - EINVAL;
  }
  return 0;
}

/*读函数*/
static ssize_t globalmem_read(struct file *filp, char __user *buf, size_t size,
  loff_t *ppos)
{
  unsigned long p =  *ppos;
  unsigned int count = size;
  int ret = 0;
  struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指针*/

  /*分析和获取有效的写长度*/
  if (p >= GLOBALMEM_SIZE)
    return count ?  - ENXIO: 0;
  if (count > GLOBALMEM_SIZE - p)
    count = GLOBALMEM_SIZE - p;

  /*内核空间→用户空间*/
  if (copy_to_user(buf, (void*)(dev->mem + p), count))
  {
    ret =  - EFAULT;
  }
  else
  {
    *ppos += count;
    ret = count;

```

```
    83      printk(KERN_INFO "read %d bytes(s) from %d\n", count, p);
    84    }
    85
    86    return ret;
    87  }
    88
    89  /*写函数*/
    90  static ssize_t globalmem_write(struct file *filp, const char __user
*buf,
    91    size_t size, loff_t *ppos)
    92  {
    93    unsigned long p =  *ppos;
    94    unsigned int count = size;
    95    int ret = 0;
    96    struct globalmem_dev *dev = filp->private_data; /*获得设备结构体
指针*/
    97
    98    /*分析和获取有效的写长度*/
    99    if (p >= GLOBALMEM_SIZE)
    100     return count ?  - ENXIO: 0;
    101   if (count > GLOBALMEM_SIZE - p)
    102     count = GLOBALMEM_SIZE - p;
    103
    104   /*用户空间→内核空间*/
    105   if (copy_from_user(dev->mem + p, buf, count))
    106     ret =  - EFAULT;
    107   else
    108   {
    109     *ppos += count;
    110     ret = count;
    111
    112     printk(KERN_INFO "written %d bytes(s) from %d\n", count, p);
    113   }
    114
    115   return ret;
    116 }
    117
    118 /* seek 文件定位函数 */
    119 static loff_t globalmem_llseek(struct file *filp, loff_t offset,
int orig)
```

```
{
  loff_t ret = 0;
  switch (orig)
  {
    case 0:   /*相对文件开始位置偏移*/
      if (offset < 0)
      {
        ret =  - EINVAL;
        break;
      }
      if ((unsigned int)offset > GLOBALMEM_SIZE)
      {
        ret =  - EINVAL;
        break;
      }
      filp->f_pos = (unsigned int)offset;
      ret = filp->f_pos;
      break;
    case 1:   /*相对文件当前位置偏移*/
      if ((filp->f_pos + offset) > GLOBALMEM_SIZE)
      {
        ret =  - EINVAL;
        break;
      }
      if ((filp->f_pos + offset) < 0)
      {
        ret =  - EINVAL;
        break;
      }
      filp->f_pos += offset;
      ret = filp->f_pos;
      break;
    default:
      ret =  - EINVAL;
      break;
  }
  return ret;
}

/*文件操作结构体*/
```

```
static const struct file_operations globalmem_fops =
{
  .owner = THIS_MODULE,
  .llseek = globalmem_llseek,
  .read = globalmem_read,
  .write = globalmem_write,
  .ioctl = globalmem_ioctl,
  .open = globalmem_open,
  .release = globalmem_release,
};

/*初始化并注册cdev*/
static void globalmem_setup_cdev(struct globalmem_dev *dev, int index)
{
  int err, devno = MKDEV(globalmem_major, index);

  cdev_init(&dev->cdev, &globalmem_fops);
  dev->cdev.owner = THIS_MODULE;
  dev->cdev.ops = &globalmem_fops;
  err = cdev_add(&dev->cdev, devno, 1);
  if (err)
    printk(KERN_NOTICE "Error %d adding LED%d", err, index);
}

/*设备驱动模块加载函数*/
int globalmem_init(void)
{
  int result;
  dev_t devno = MKDEV(globalmem_major, 0);

  /* 申请设备号*/
  if (globalmem_major)
    result = register_chrdev_region(devno, 1, "globalmem");
  else  /* 动态申请设备号 */
  {
    result = alloc_chrdev_region(&devno, 0, 1, "globalmem");
    globalmem_major = MAJOR(devno);
  }
  if (result < 0)
```

```
199     return result;
200
201   /* 动态申请设备结构体的内存*/
202       globalmem_devp  =  kmalloc(sizeof(struct  globalmem_dev),
GFP_KERNEL);
203   if (!globalmem_devp)   /*申请失败*/
204   {
205     result =  - ENOMEM;
206     goto fail_malloc;
207   }
208   memset(globalmem_devp, 0, sizeof(struct globalmem_dev));
209
210   globalmem_setup_cdev(globalmem_devp, 0);
211   return 0;
212
213   fail_malloc: unregister_chrdev_region(devno, 1);
214   return result;
215 }
216
217 /*模块卸载函数*/
218 void globalmem_exit(void)
219 {
220   cdev_del(&globalmem_devp->cdev);   /*注销 cdev*/
221   kfree(globalmem_devp);    /*释放设备结构体内存*/
222   unregister_chrdev_region(MKDEV(globalmem_major, 0), 1); /*释放设
备号*/
223 }
224
225 MODULE_AUTHOR("Song Baohua");
226 MODULE_LICENSE("Dual BSD/GPL");
227
228 module_param(globalmem_major, int, S_IRUGO);
229
230 module_init(globalmem_init);
231 module_exit(globalmem_exit);
```

除了在 globalmem_open()函数中通过 filp->private_data = globalmem_devp 语句（见第 31 行）将设备结构体指针赋值给文件私有数据指针并在 globalmem_read()、globalmem_write()、globalmem_llseek() 和 globalmem_ioctl() 函数中通过 struct globalmem_dev *dev = filp->private_data 语句获得设备结构体指针并使用该指针操作设备结构体外，代码清单 6.17 与代码清单 6.8～代码清单 6.15 的程序基本相同。

现在翻回到本书的第 1 章，再次阅读代码清单 1.4，即 Linux 下 LED 的设备驱动，是否豁然开朗？

代码清单 6.17 仅仅作为使用 private_data 的范例，直接访问全局变量 globalmem_devp 会更加结构清晰。如果 globalmem 不只包括一个设备，而是同时包括两个或两个以上的设备，采用 private_data 的优势就会显现出来。

在不对代码清单 6.17 中的 globalmem_read()、globalmem_write()、globalmem_ioctl() 等重要函数及 globalmem_fops 结构体等数据结构进行任何修改的前提下，只是简单地

修改 globalmem_ init()、globalmem_exit()和 globalmem_open()，就可以轻松地让 globalmem 驱动中包含两个同样的设备（次设备号分别为 0 和 1），如代码清单 6.18 所示。

代码清单 6.18　支持两个 globalmem 设备的 globalmem 驱动

```
/*文件打开函数*/
int globalmem_open(struct inode *inode, struct file *filp)
{
  /*将设备结构体指针赋值给文件私有数据指针*/
  struct globalmem_dev *dev;

  dev = container_of(inode->i_cdev,struct globalmem_dev,cdev);
  filp->private_data = dev;
  return 0;
}

/*设备驱动模块加载函数*/
int globalmem_init(void)
{
  int result;
  dev_t devno = MKDEV(globalmem_major, 0);

  /* 申请设备号*/
  if (globalmem_major)
    result = register_chrdev_region(devno, 2, "globalmem");
  else  /* 动态申请设备号 */
  {
    result = alloc_chrdev_region(&devno, 0, 2, "globalmem");
    globalmem_major = MAJOR(devno);
  }
  if (result < 0)
    return result;

  /* 动态申请两个设备结构体的内存*/
    globalmem_devp  =  kmalloc(2*sizeof(struct  globalmem_dev), GFP_KERNEL);
  if (!globalmem_devp)   /*申请失败*/
  {
    result =  - ENOMEM;
    goto fail_malloc;
  }
  memset(globalmem_devp, 0, 2*sizeof(struct globalmem_dev));

  globalmem_setup_cdev(globalmem_devp[0], 0);
  globalmem_setup_cdev(&globalmem_devp[1], 1);
  return 0;

  fail_malloc: unregister_chrdev_region(devno, 1);
  return result;
}

/*模块卸载函数*/
void globalmem_exit(void)
{
  cdev_del(&(global mem_devp[0].cdev));
  cdev_del(&(global mem_devp[1].cdev)); /*注销 cdev*/
```

```
51  kfree(globalmem_devp);     /*释放设备结构体内存*/
52  unregister_chrdev_region(MKDEV(globalmem_major, 0), 2); /*释放设备号*/
53 }
/* 其他代码同代码清单 6.16 */
```

代码清单 6.18 第 7 行调用的 container_of()的作用是通过结构体成员的指针找到对应结构体的指针，这个技巧在 Linux 内核编程中十分常用。在 container_of（inode->i_cdev,struct globalmem_dev,cdev）语句中，传给 container_of()的第 1 个参数是结构体成员的指针，第 2 个参数为整个结构体的类型，第 3 个参数为传入的第 1 个参数即结构体成员的类型，container_of()返回值为整个结构体的指针。

下面是代码清单 6.18 针对 6.17 的 patch：

```
36c38,41
<   filp->private_data = globalmem_devp;
---
>   struct globalmem_dev *dev;
>
>   dev = container_of(inode->i_cdev,struct globalmem_dev,cdev);
>   filp->private_data = dev;
198c203
<    result = register_chrdev_region(devno, 1, "globalmem");
---
>    result = register_chrdev_region(devno, 2, "globalmem");
201c206
<    result = alloc_chrdev_region(&devno, 0, 1, "globalmem");
---
>    result = alloc_chrdev_region(&devno, 0, 2, "globalmem");
207,208c212,213
<   /* 动态申请设备结构体的内存*/
<       globalmem_devp  =  kmalloc(sizeof(struct  globalmem_dev), GFP_KERNEL);
---
>   /* 动态申请两个设备结构体的内存*/
>      globalmem_devp  =  kmalloc(2*sizeof(struct  globalmem_dev), GFP_KERNEL);
214c219
<   memset(globalmem_devp, 0, sizeof(struct globalmem_dev));
---
>   memset(globalmem_devp, 0, 2*sizeof(struct globalmem_dev));
216c221,222
<   globalmem_setup_cdev(globalmem_devp, 0);
---
>   globalmem_setup_cdev(&globalmem_devp[0], 0);
>   globalmem_setup_cdev(&globalmem_devp[1], 1);
226c232,233
<   cdev_del(&globalmem_devp->cdev);   /*注销 cdev*/
---
>   cdev_del(&(globalmem_devp[0].cdev));
>   cdev_del(&(globalmem_devp[1].cdev));   /*注销 cdev*/
228c235
<   unregister_chrdev_region(MKDEV(globalmem_major, 0), 1); /*释放设备号*/
---
>   unregister_chrdev_region(MKDEV(globalmem_major, 0), 2); /*释放设备号*/
```

6.4 globalmem 驱动在用户空间的验证

编译 globalmem 的驱动，得到 globalmem.ko 文件。运行“insmod globalmem.ko”命令加载模块，通过“lsmod”命令，发现 globalmem 模块已被加载。再通过“cat /proc/devices”命令查看，发现多出了主设备号为 254 的“globalmem”字符设备驱动，如下所示：

```
[root@localhost driver_study]# cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 21 sg
 29 fb
128 ptm
136 pts
171 ieee1394
180 usb
189 usb_device
254 globalmem
```

接下来，通过“mknod /dev/globalmem c 254 0”命令创建“/dev/globalmem”设备节点，并通过“echo 'hello world' > /dev/globalmem”命令和“cat /dev/globalmem”命令分别验证设备的写和读，结果证明“hello world”字符串被正确地写入 globalmem 字符设备：

```
[root@localhost driver_study]# echo 'hello world' > /dev/globalmem
[root@localhost driver_study]# cat /dev/globalmem
hello world
```

如果启用了 sysfs 文件系统，将发现多出了/sys/module/globalmem 目录，该目录下的树型结构如下所示：

```
|-- refcnt
'-- sections
      |-- .bss
      |-- .data
      |-- .gnu.linkonce.this_module
      |-- .rodata
      |-- .rodata.str1.1
      |-- .strtab
      |-- .symtab
```

```
        |-- .text
        '-- __versions
```

refcnt 记录了 globalmem 模块的引用计数，sections 下包含的多个文件则给出了 globalmem 所包含的 BSS、数据段和代码段等的地址及其他信息。

对于代码清单 6.18 给出的支持两个 globalmem 设备的驱动，在加载模块后需创建两个设备节点，/dev/globalmem0 对应主设备号 globalmem_major，次设备号 0，/dev/globalmem1 对应主设备号 globalmem_major，次设备号 1。分别读写 /dev/globalmem0 和/dev/globalmem1，发现都读写到了正确的对应设备。

6.5 总结

字符设备是 3 大类设备（字符设备、块设备和网络设备）中较简单的一类设备，其驱动程序中完成的主要工作是初始化、添加和删除 cdev 结构体，申请和释放设备号，以及填充 file_operations 结构体中的操作函数，实现 file_operations 结构体中的 read()、write()和 ioctl()等函数是驱动设计的主体工作。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm
- 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程： 华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：
  http://www.farsight.com.cn/courses/TS-LinuxApp.htm
- 嵌入式 Linux 系统开发班：
  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm
- 嵌入式 Linux 驱动开发班：

华清远见

第 7 章 Linux 设备驱动中的并发控制

Linux 设备驱动中必须解决的一个问题是多个进程对共享资源的并发访问，并发访问会导致竞态。

Linux 提供了多种解决竞态问题的方式，这些方式适合不同的应用场景。7.1 节描述了并发和竞态的概念及发生场合。7.2～7.5 节分别讲解了中断屏蔽、原子操作、自旋锁和信号量等并发控制机制。7.6 节讲解增加并发控制后的 globalmem 的设备驱动。

华清远见

7.1 并发与竞态

并发（concurrency）指的是多个执行单元同时、并行被执行，而并发的执行单元对共享资源（硬件资源和软件上的全局变量、静态变量等）的访问则很容易导致竞态（race conditions）。例如，对于 globalmem 设备，假设一个执行单元 A 对其写入 3000 个字符“a”，而另一个执行单元 B 对其写入 4000 个字符“b”，第三个执行单元 C 读取 globalmem 的所有字符。如果执行单元 A、B 的写操作如图 7.1 所示的顺序执行，执行单元 C 的读操作不会有问题。但是，如果执行单元 A、B 如图 7.2 所示的顺序执行，而执行单元 C 又“不合时宜”地读，则会读出 3000 个“b”。

执行单元 A	执行单元 B	执行单元 C
copy_from_user(dev->mem+p, buf, count)		
…		
dev->current_len=p+count		
		read_globalmem()
	copy_from_user(dev->mem+p, buf, count)	
	…	
	dev->current_len=p+count	
		read_globalmem()

图 7.1 并发执行单元的顺序执行

比图 7.2 更复杂、更混乱的并发大量地存在于设备驱动中，只要并发的多个执行单元存在对共享资源的访问，竞态就可能发生。在 Linux 内核中，主要的竞态发生于如下几种情况。

执行单元 A	执行单元 B	执行单元 C
copy_from_user(dev->men+p, buf, count)		
…	copy_from_user(dev->men+p, buf, count)	
dev->current_len=p+count		
	…	read_globalmem
	dev->current_len=p+count	

图 7.2 并发执行单元的交错执行

1. 对称多处理器（SMP）的多个 CPU

图 7.3 SMP 体系结构

SMP 是一种紧耦合、共享存储的系统模型，其体系结构如图 7.3 所示，它的特点是多个 CPU 使用共同的系统总线，因此可访问共同的外设和储存器。

2. 单 CPU 内进程与抢占它的进程

Linux 2.6 内核支持抢占调度，一个进程

嵌入式学院 华清远见旗下品牌

在内核执行的时候可能被另一高优先级进程打断，进程与抢占它的进程访问共享资源的情况类似于 SMP 的多 个 CPU。

3．中断（硬中断、软中断、Tasklet、底半部）与进程之间

中断可以打断正在执行的进程，如果中断处理程序访问进程正在访问的资源，则竞态也会发生。

此外，中断也有可能被新的更高优先级的中断打断，因此，多个中断之间本身也可能引起并发而导致竞态。

上述并发的发生情况除了 SMP 是真正的并行以外，其他的都是“宏观并行、微观串行”的，但其引发的实质问题和 SMP 相似。

解决竞态问题的途径是保证对共享资源的互斥访问，所谓互斥访问是指一个执行单元在访问共享资源的时候，其他的执行单元被禁止访问。

访问共享资源的代码区域称为临界区（critical sections），临界区需要以某种互斥机制加以保护。中断屏蔽、原子操作、自旋锁和信号量等是 Linux 设备驱动中可采用的互斥途径，7.2～7.5 节将进行一一讲解。

7.2 中断屏蔽

在单 CPU 范围内避免竞态的一种简单方法是在进入临界区之前屏蔽系统的中断。CPU 一般都具备屏蔽中断和打开中断的功能，这项功能可以保证正在执行的内核执行路径不被中断处理程序所抢占，防止某些竞态条件的发生。具体而言，中断屏蔽将使得中断与进程之间的并发不再发生，而且，由于 Linux 内核的进程调度等操作都依赖中断来实现，内核抢占进程之间的并发也就得以避免了。

中断屏蔽的使用方法为：

```
local_irq_disable() //屏蔽中断
...
critical section //临界区
...
local_irq_enable()  //开中断
```

由于 Linux 系统的异步 I/O、进程调度等很多重要操作都依赖于中断，中断对于内核的运行非常重要，在屏蔽中断期间所有的中断都无法得到处理，因此长时间屏蔽中断是很危险的，有可能造成数据丢失甚至系统崩溃。这就要求在屏蔽了中断之后，当前的内核执行路径应当尽快地执行完临界区的代码。

local_irq_disable()和 local_irq_enable()都只能禁止和使能本 CPU 内的中断，因此，并不能解决 SMP 多 CPU 引发的竞态。因此，单独使用中断屏蔽通常不是一种值得推荐的避免竞态的方法，它适宜与自旋锁联合使用。

与 local_irq_disable()不同的是，local_irq_save（flags）除了进行禁止中断的操作以外，还保存目前 CPU 的中断位信息，local_irq_restore（flags）进行的是与 local_irq_save（flags）相反的操作。

如果只是想禁止中断的底半部，应使用 local_bh_disable()，使能被 local_bh_disable() 禁止的底半部应该调用 local_bh_enable()。

7.3 原子操作

原子操作指的是在执行过程中不会被别的代码路径所中断的操作。

Linux 内核提供了一系列函数来实现内核中的原子操作，这些函数又分为两类，分别针对位和整型变量进行原子操作。它们的共同点是在任何情况下操作都是原子的，内核代码可以安全地调用它们而不被打断。位和整型变量原子操作都依赖底层 CPU 的原子操作来实现，因此所有这些函数都与 CPU 架构密切相关。

7.3.1 整型原子操作

1. 设置原子变量的值

```
void atomic_set(atomic_t *v, int i); //设置原子变量的值为 i
atomic_t v = ATOMIC_INIT(0); //定义原子变量 v 并初始化为 0
```

2. 获取原子变量的值

```
atomic_read(atomic_t *v);  //返回原子变量的值
```

3. 原子变量加/减

```
void atomic_add(int i, atomic_t *v); //原子变量增加 i
void atomic_sub(int i, atomic_t *v); //原子变量减少 i
```

4. 原子变量自增/自减

```
void atomic_inc(atomic_t *v);    //原子变量增加 1
void atomic_dec(atomic_t *v);    //原子变量减少 1
```

5. 操作并测试

```
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
```

上述操作对原子变量执行自增、自减和减操作后（注意没有加）测试其是否为 0，为 0 则返回 true，否则返回 false。

6. 操作并返回

```
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

上述操作对原子变量进行加/减和自增/自减操作，并返回新的值。

7.3.2 位原子操作

1. 设置位

```
void set_bit(nr, void *addr);
```

上述操作设置 addr 地址的第 nr 位，所谓设置位即将位写为 1。

2. 清除位

```
void clear_bit(nr, void *addr);
```

上述操作清除 addr 地址的第 nr 位，所谓清除位即将位写为 0。

3. 改变位

```
void change_bit(nr, void *addr);
```

上述操作对 addr 地址的第 nr 位进行反置。

4. 测试位

```
test_bit(nr, void *addr);
```

上述操作返回 addr 地址的第 nr 位。

5. 测试并操作位

```
int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
```

上述 test_and_xxx_bit（nr, void *addr）操作等同于执行 test_bit（nr, void *addr）后再执行 xxx_bit（nr, void *addr）。

代码清单 7.1 给出了原子变量的使用实例，它用于使设备最多只能被一个进程打开。

代码清单 7.1 使用原子变量使设备只能被一个进程打开

```
static atomic_t xxx_available = ATOMIC_INIT(1); /*定义原子变量*/

static int xxx_open(struct inode *inode, struct file *filp)
{
  ...
  if (!atomic_dec_and_test(&xxx_available))
  {
    atomic_inc(&xxx_available);
    return  - EBUSY; /*已经打开*/
  }
  ...
  return 0; /* 成功 */
}
```

```
14
15 static int xxx_release(struct inode *inode, struct file *filp)
16 {
17   atomic_inc(&xxx_available); /* 释放设备 */
18   return 0;
19 }
```

7.4 自旋锁

7.4.1 自旋锁的使用

自旋锁（spin lock）是一种对临界资源进行互斥手访问的典型手段，其名称来源于它的工作方式。为了获得一个自旋锁，在某 CPU 上运行的代码需先执行一个原子操作，该操作测试并设置（test-and-set）某个内存变量，由于它是原子操作，所以在该操作完成之前其他执行单元不可能访问这个内存变量。

如果测试结果表明锁已经空闲，则程序获得这个自旋锁并继续执行；如果测试结果表明锁仍被占用，程序将在一个小的循环内重复这个“测试并设置”操作，即进行所谓的“自旋”，通俗地说就是“在原地打转”。当自旋锁的持有者通过重置该变量释放这个自旋锁后，某个等待的“测试并设置”操作向其调用者报告锁已释放。

理解自旋锁最简单的方法是把它作为一个变量看待，该变量把一个临界区或者标记为“我当前在运行，请稍等一会”或者标记为“我当前不在运行，可以被使用”。如果 A 执行单元首先进入例程，它将持有自旋锁；当 B 执行单元试图进入同一个例程时，将获知自旋锁已被持有，需等到 A 执行单元释放后才能进入。

Linux 系统中与自旋锁相关的操作主要有如下 4 种。

1. 定义自旋锁

```
spinlock_t spin;
```

2. 初始化自旋锁

```
spin_lock_init(lock)
```

该宏用于动态初始化自旋锁 lock

3. 获得自旋锁

```
spin_lock(lock)
```

该宏用于获得自旋锁 lock，如果能够立即获得锁，它就马上返回，否则，它将自旋在那里，直到该自旋锁的保持者释放；

```
spin_trylock(lock)
```

该宏尝试获得自旋锁 lock，如果能立即获得锁，它获得锁并返回真，否则立即返

回假，实际上不再“在原地打转”；

4．释放自旋锁

```
spin_unlock(lock)
```

该宏释放自旋锁 lock，它与 spin_trylock 或 spin_lock 配对使用。

自旋锁一般这样被使用，如下所示：

```
//定义一个自旋锁
spinlock_t lock;
spin_lock_init(&lock);

spin_lock (&lock) ; //获取自旋锁，保护临界区
...//临界区
spin_unlock (&lock) ; //解锁
```

自旋锁主要针对 SMP 或单 CPU 但内核可抢占的情况，对于单 CPU 和内核不支持抢占的系统，自旋锁退化为空操作。在单 CPU 和内核可抢占的系统中，自旋锁持有期间内核的抢占将被禁止。由于内核可抢占的单 CPU 系统的行为实际很类似于 SMP 系统，因此，在这样的单 CPU 系统中使用自旋锁仍十分必要。

尽管用了自旋锁可以保证临界区不受别的 CPU 和本 CPU 内的抢占进程打扰，但是得到锁的代码路径在执行临界区的时候还可能受到中断和底半部（BH）的影响。为了防止这种影响，就需要用到自旋锁的衍生。spin_lock()/spin_unlock()是自旋锁机制的基础，它们和关中断 local_irq_ disable()/开中断 local_irq_enable()、关底半部 local_bh_disable()/开底半部 local_bh_enable()、关中断并保存状态字 local_irq_save()/开中断并恢复状态 local_irq_restore()结合就形成了整套自旋锁机制，关系如下所示：

```
spin_lock_irq() = spin_lock() + local_irq_disable()
spin_unlock_irq() = spin_unlock() + local_irq_enable()
spin_lock_irqsave() = spin_unlock() + local_irq_save()
spin_unlock_irqrestore() = spin_unlock() + local_irq_restore()
spin_lock_bh() = spin_lock() + local_bh_disable()
spin_unlock_bh() = spin_unlock() + local_bh_enable()
```

驱动工程师应谨慎使用自旋锁，而且在使用中还要特别注意如下几个问题。

- 自旋锁实际上是忙等锁，当锁不可用时，CPU 一直循环执行“测试并设置”该锁直到可用而取得该锁，CPU 在等待自旋锁时不做任何有用的工作，仅仅是等待。因此，只有在占用锁的时间极短的情况下，使用自旋锁才是合理的。当临界区很大或有共享设备的时候，需要较长时间占用锁，使用自旋锁会降低系统的性能。
- 自旋锁可能导致系统死锁。引发这个问题最常见的情况是递归使用一个自旋锁，即如果一个已经拥有某个自旋锁的 CPU 想第二次获得这个自旋锁，则该 CPU 将死锁。此外，如果进程获得自旋锁之后再阻塞，也有可能导致死锁的发生。copy_from_user()、copy_to_user()和 kmalloc()等函数都有可能引起阻塞，因此在自旋锁的占用期间不能调用这些函数。

代码清单 7.2 给出了自旋锁的使用实例，它被用于实现使得设备只能被最多一个进程打开。

代码清单 7.2 使用自旋锁使设备只能被一个进程打开

```
1  int xxx_count = 0;/*定义文件打开次数计数*/
2
3  static int xxx_open(struct inode *inode, struct file *filp)
4  {
5    ...
6    spinlock(&xxx_lock);
7    if (xxx_count)/*已经打开*/
8    {
9      spin_unlock(&xxx_lock);
10     return  - EBUSY;
11   }
12   xxx_count++;/*增加使用计数*/
13   spin_unlock(&xxx_lock);
14   ...
15   return 0; /* 成功 */
16 }
17
18 static int xxx_release(struct inode *inode, struct file *filp)
19 {
20   ...
21   spinlock(&xxx_lock);
22   xxx_count--; /*减少使用计数*/
23   spin_unlock(&xxx_lock);
24
25   return 0;
26 }
```

7.4.2 读写自旋锁

自旋锁不关心锁定的临界区究竟进行怎样的操作，不管是读还是写，它都一视同仁。即便多个执行单元同时读取临界资源也会被锁住。实际上，对共享资源并发访问时，多个执行单元同时读取它是不会有问题的，自旋锁的衍生锁读写自旋锁（rwlock）可允许读的并发。

读写自旋锁是一种比自旋锁粒度更小的锁机制，它保留了“自旋”的概念，但是在写操作方面，只能最多有一个写进程，在读操作方面，同时可以有多个读执行单元。当然，读和写也不能同时进行。

读写自旋锁涉及的操作如下所示。

1. 定义和初始化读写自旋锁

```
rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* 静态初始化 */
rwlock_t my_rwlock;
rwlock_init(&my_rwlock); /* 动态初始化 */
```

2. 读锁定

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);
```

3．读解锁

```
void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

在对共享资源进行读取之前，应该先调用读锁定函数，完成之后应调用读解锁函数。

read_lock_irqsave()、read_lock_irq()和 read_lock_bh()分别是 read_lock()分别与 local_irq_save()、local_irq_disable()和 local_bh_disable()的组合，读解锁函数 read_unlock_irqrestore()、read_unlock_ irq()、read_unlock_bh()的情况与此类似。

4．写锁定

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);
```

5．写解锁

```
void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

write_lock_irqsave()、write_lock_irq()、write_lock_bh()分别是 write_lock()与 local_irq_save()、local_irq_disable()和 local_bh_disable()的组合，写解锁函数 write_unlock_irqrestore()、write_unlock_irq()、write_unlock_bh()的情况与此类似。

在对共享资源进行读取之前，应该先调用写锁定函数，完成之后应调用写解锁函数。和 spin_trylock()一样，write_trylock()也只是尝试获取读写自旋锁，不管成功失败，都会立即返回。

读写自旋锁一般这样被使用，如下所示：

```
rwlock_t lock;   //定义 rwlock
rwlock_init(&lock); //初始化 rwlock

//读时获取锁
read_lock(&lock);
...  //临界资源
read_unlock(&lock);

//写时获取锁
write_lock_irqsave(&lock, flags);
... //临界资源
write_unlock_irqrestore(&lock, flags);
```

7.4.3 顺序锁

顺序锁（seqlock）是对读写锁的一种优化，若使用顺序锁，读执行单元绝不会被写执行单元阻塞，也就是说，读执行单元可以在写执行单元对被顺序锁保护的共享资源进行写操作时仍然可以继续读，而不必等待写执行单元完成写操作，写执行单元也不需要等待所有读执行单元完成读操作才去进行写操作。

但是，写执行单元与写执行单元之间仍然是互斥的，即如果有写执行单元在进行写操作，其他写执行单元必须自旋在那里，直到写执行单元释放了顺序锁。

如果读执行单元在读操作期间，写执行单元已经发生了写操作，那么，读执行单元必须重新读取数据，以便确保得到的数据是完整的。这种锁在读写同时进行的概率比较小时，性能是非常好的，而且它允许读写同时进行，因而更大地提高了并发性。

顺序锁有一个限制，它必须要求被保护的共享资源不含有指针，因为写执行单元可能使得指针失效，但读执行单元如果正要访问该指针，将导致 Oops。

在 Linux 内核中，写执行单元涉及如下顺序锁操作。

1．获得顺序锁

```
void write_seqlock(seqlock_t *sl);
int write_tryseqlock(seqlock_t *sl);
write_seqlock_irqsave(lock, flags)
write_seqlock_irq(lock)
write_seqlock_bh(lock)
```

其中：

```
write_seqlock_irqsave() = loal_irq_save() + write_seqlock()
write_seqlock_irq() = local_irq_disable() + write_seqlock()
write_seqlock_bh() = local_bh_disable() + write_seqlock()
```

2．释放顺序锁

```
void write_sequnlock(seqlock_t *sl);
write_sequnlock_irqrestore(lock, flags)
write_sequnlock_irq(lock)
write_sequnlock_bh(lock)
```

其中：

```
write_sequnlock_irqrestore() = write_sequnlock() + local_irq_restore()
write_sequnlock_irq() = write_sequnlock() + local_irq_enable()
write_sequnlock_bh() = write_sequnlock() + local_bh_enable()
```

写执行单元使用顺序锁的模式如下：

```
write_seqlock(&seqlock_a);
...//写操作代码块
write_sequnlock(&seqlock_a);
```

因此，对写执行单元而言，它的使用与 spinlock 相同。

读执行单元涉及如下顺序锁操作。

1．读开始

```
unsigned read_seqbegin(const seqlock_t *sl);
read_seqbegin_irqsave(lock, flags)
```

读执行单元在对被顺序锁 s1 保护的共享资源进行访问前需要调用该函数，该函数仅返回顺序锁 s1 的当前顺序号。其中：

```
read_seqbegin_irqsave() = local_irq_save() + read_seqbegin()
```

2．重读

```
int read_seqretry(const seqlock_t *sl, unsigned iv);
read_seqretry_irqrestore(lock, iv, flags)
```

读执行单元在访问完被顺序锁 s1 保护的共享资源后需要调用该函数来检查，在读访问期间是否有写操作。如果有写操作，读执行单元就需要重新进行读操作。其中：

```
read_seqretry_irqrestore() = read_seqretry() + local_irq_restore()
```

读执行单元使用顺序锁的模式如下：

```
do {
   seqnum = read_seqbegin(&seqlock_a);
//读操作代码块
...
} while (read_seqretry(&seqlock_a, seqnum));
```

7.4.4 读－拷贝－更新

RCU（Read-Copy Update，读－拷贝－更新），它是基于其原理命名的。RCU 并不是新的锁机制，它对于 Linux 内核而言是新的。早在 20 世纪 80 年代就有了这种机制，而在 Linux 系统中，开发 2.5.43 内核时引入该技术，并正式包含在 2.6 内核中。

对于被 RCU 保护的共享数据结构，读执行单元不需要获得任何锁就可以访问它，不使用原子指令，而且在除 Alpha 的所有架构上也不需要内存栅（Memory Barrier），因此不会导致锁竞争、内存延迟以及流水线停滞。不需要锁也使得使用更容易，因为死锁问题就不需要考虑了。

使用 RCU 的写执行单元在访问它前需首先复制一个副本，然后对副本进行修改，最后使用一个回调机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据，这个时机就是所有引用该数据的 CPU 都退出对共享数据的操作的时候。读执行单元没有任何同步开销，而写执行单元的同步开销则取决于使用的写执行单元间的同步机制。

RCU 可以看做读写锁的高性能版本，相比读写锁，RCU 的优点在于既允许多个读执行单元同时访问被保护的数据，又允许多个读执行单元和多个写执行单元同时访问被保护的数据。

但是，RCU 不能替代读写锁，因为如果写比较多时，对读执行单元的性能提高不能弥补写执行单元导致的损失。因为使用 RCU 时，写执行单元之间的同步开销会比较大，它需要延迟数据结构的释放，复制被修改的数据结构，它也必须使用某种锁机制同步并行的其他写执行单元的修改操作。

Linux 系统中提供的 RCU 操作如下 4 种。

1．读锁定

```
rcu_read_lock()
rcu_read_lock_bh()
```

2．读解锁

```
rcu_read_unlock()
rcu_read_unlock_bh()
```

使用 RCU 进行读的模式如下：

```
rcu_read_lock()
...//读临界区
rcu_read_unlock()
```

其中 rcu_read_lock()和 rcu_read_unlock()实质只是禁止和使能内核的抢占调度，如下所示：

```
#define rcu_read_lock()        preempt_disable()
#define rcu_read_unlock()      preempt_enable()
```

其变种 rcu_read_lock_bh()、rcu_read_unlock_bh()则定义为：

```
#define rcu_read_lock_bh()      local_bh_disable()
#define rcu_read_unlock_bh()    local_bh_enable()
```

3．同步 RCU

```
synchronize_rcu()
```

该函数由 RCU 写执行单元调用，它将阻塞写执行单元，直到所有的读执行单元已经完成读执行单元临界区，写执行单元才可以继续下一步操作。如果有多个 RCU 写执行单元调用该函数，它们将在一个 grace period（即所有的读执行单元已经完成对临界区的访问）之后全部被唤醒。synchronize_rcu()保证所有 CPU 都处理完正在运行的读执行单元临界区。

```
synchronize_kernel()
```

内核代码使用该函数来等待所有 CPU 处于可抢占状态，目前功能等同于 synchronize_rcu()，但现在已经不建议使用，而是使用 synchronize_sched()，该函数用于等待所有 CPU 都处在可抢占状态，它能保证正在运行的中断处理函数处理完毕，但不能保证正在运行的软中断处理完毕。

4．挂接回调

```
void fastcall call_rcu(struct rcu_head *head,
              void (*func)(struct rcu_head *rcu));
```

函数 call_rcu()也由 RCU 写执行单元调用，它不会使写执行单元阻塞，因而可以在中断上下文或软中断中使用。该函数将把函数 func 挂接到 RCU 回调函数链上，然后立即返回。函数 synchronize_rcu()的实现实际上使用了 call_rcu()函数。

```
void fastcall call_rcu_bh(struct rcu_head *head,
                        void (*func)(struct rcu_head *rcu));
```

call_ruc_bh()函数的功能几乎与 call_rcu()完全相同，唯一差别就是它把软中断的完成也当做经历一个 quiescent state（静默状态），因此如果写执行单元使用了该函数，

在进程上下文的读执行单元必须使用 rcu_read_lock_bh()。

每个 CPU 维护两个数据结构 rcu_data 和 rcu_bh_data，它们用于保存回调函数，函数 call_rcu()把回调函数注册到 rcu_data，而 call_rcu_bh()则把回调函数注册到 rcu_bh_data，在每一个数据结构上，回调函数被组成一个链表，先注册的排在前头，后注册的排在末尾。

使用 RCU 时，读执行单元必须提供一个信号给写执行单元以便写执行单元能够确定数据可以被安全地释放或修改的时机。有一个专门的垃圾收集器来探测读执行单元的信号，一旦所有的读执行单元都已经发送信号告知它们都不在使用被 RCU 保护的数据结构，垃圾收集器就调用回调函数完成最后的数据释放或修改操作。

RCU 还增加了链表操作函数的 RCU 版本，如下所示：

```
static inline void list_add_rcu(struct list_head *new, struct list_head *head);
```

该函数把链表元素 new 插入到 RCU 保护的链表 head 的开头，内存栅保证了在引用这个新插入的链表元素之前，新链表元素的链接指针的修改对所有读执行单元是可见的。

```
static inline void list_add_tail_rcu(struct list_head *new,
    struct list_head *head);
```

该函数类似于 list_add_rcu()，它将把新的链表元素 new 添加到被 RCU 保护的链表的末尾。

```
static inline void list_del_rcu(struct list_head *entry);
```

该函数从 RCU 保护的链表中删除指定的链表元素 entry。

```
static inline void list_replace_rcu(struct list_head *old, struct list_head *new);
```

该函数是 RCU 新添加的函数，并不存在非 RCU 版本。它使用新的链表元素 new 取代旧的链表元素 old，内存栅保证在引用新的链表元素之前，它对链接指针的修正对所有读执行单元是可见的。

```
list_for_each_rcu(pos, head)
```

该宏用于遍历由 RCU 保护的链表 head，只要在读执行单元临界区使用该函数，它就可以安全地和其他_rcu 链表操作函数并发运行如 list_add_rcu()。

```
list_for_each_safe_rcu(pos, n, head)
```

该宏类似于 list_for_each_rcu，不同之处在于它允许安全地删除当前链表元素 pos。

```
list_for_each_entry_rcu(pos, head, member)
```

该宏类似于 list_for_each_rcu，不同之处在于它用于遍历指定类型的数据结构链表，当前链表元素 pos 为一个包含 struct list_head 结构的特定的数据结构。

```
static inline void hlist_del_rcu(struct hlist_node *n)
```

它从由 RCU 保护的哈希链表中移走链表元素 n。

```
static inline void hlist_add_head_rcu(struct hlist_node *n,
    struct hlist_head *h);
```

该函数用于把链表元素 n 插入到被 RCU 保护的哈希链表的开头，但同时允许读执行单元对该哈希链表的遍历。内存栅确保在引用新链表元素之前，它对指针的修改对所有读执行单元可见。

```
hlist_for_each_rcu(pos, head)
```

该宏用于遍历由 RCU 保护的哈希链表 head，只要在读端临界区使用该函数，它就可以安全地和其他_rcu 哈希链表操作函数（如 hlist_add_rcu）并发运行。

```
    hlist_for_each_entry_rcu(tpos, pos, head, member)
```

类似于 hlist_for_each_rcu()，不同之处在于它用于遍历指定类型的数据结构哈希链表，当前链表元素 pos 为一个包含 struct list_head 结构的特定的数据结构。

目前，RCU 的使用在内核中已经非常普遍，内核中大量原先使用读写锁的代码被 RCU 替换，下面的表单左右两列平行地分别给出使用读写锁和 RCU 实现链表元素读、删除、添加和修改的代码。

```
/*
  原先的audit_filter_task 函数，读链表前获得读写锁 */
static enum audit_state
  audit_filter_task(struct
task_struct
  *tsk)
{
  struct audit_entry *e;
  enum audit_state state;
  read_lock(&auditsc_lock);
  /* 遍历链表 */
  list_for_each_entry(e,    &audit_
tsklist,
    list)
  {
    if   (audit_filter_rules(tsk,
&e
      ->rule, NULL, &state))
    {
      read_unlock(&auditsc_lock);
      return state;
    }
  }
  read_unlock(&auditsc_lock);
  return AUDIT_BUILD_CONTEXT;
}

/* 原先的audit_del_rule，删除链表元素前获得读写锁 */
static          inline          int
audit_del_rule(struct
  audit_rule *rule, struct
```

```
/*
修改后的audit_filter_task 函数，采用 RCU
*/
static enum audit_state
  audit_filter_task(struct
task_struct
  *tsk)
{
  struct audit_entry *e;
  enum audit_state state;
  rcu_read_lock();
  /* 遍历链表 */
  list_for_each_entry_rcu(e,
    &audit_tsklist, list)
  {
    if (audit_filter_rules(tsk, &e
      ->rule, NULL, &state))
    {
      rcu_read_unlock();
      return state;
    }
  }
  rcu_read_unlock();
  return AUDIT_BUILD_CONTEXT;
}

/* 修改后的 audit_del_rule，使用 list_del_rcu 删除链表元素，并调用 call_rcu 注册回调函数 */
static          inline          int
audit_del_rule(struct
  audit_rule *rule, struct
  list_head*list)
{
  struct audit_entry *e;

  /* 遍历链表 */
  list_for_each_entry(e, list, list)
  {
    if (!audit_compare_rule(rule, &e
      ->rule))
    {
     list_del_rcu(&e->list);


call_rcu(&e->rcu,audit_free_rule,e);
      return 0;
    }
```

```
  list_head*list)
{
  struct audit_entry *e;
  write_lock(&auditsc_lock);
  /* 遍历链表 */
  list_for_each_entry(e,    list,
list)
  {
    if (!audit_compare_rule(rule,
&e
      ->rule))
    {
      list_del(&e->list); //删除链
表元素

write_unlock(&auditsc_lock);
      return 0;
    }
  }
  write_unlock(&auditsc_lock);
  return  - EFAULT;
}


/* 原先的audit_add_rule，添加链表元
素前获得读写锁 */
static   inline   int   audit_add_
rule(struct
  audit_entry *entry, struct
  list_head*list)
{
  write_lock(&auditsc_lock);
  if  (entry->rule.flags  &AUDIT_
PREPEND)
  {
    entry->rule.flags  &=  ~AUDIT_
PREPEND;
    list_add(&entry->list, list);
  }
  else
  {
  }

  return  - EFAULT;
}

/* 修改后的audit_add_rule，在添加链表元素
时使用list_add_xxx函数 */
static           inline           int
audit_add_rule(struct
  audit_entry *entry, struct
  list_head*list)
{

  if                (entry->rule.flags
&AUDIT_PREPEND)
  {
    entry->rule.flags    &=    ~AUDIT_
PREPEND;
    list_add_rcu(&entry->list, list);
  }
  else
  {
    list_add_tail_rcu(&entry->list,
      list);
  }
  return 0;
}

/* 修改后的 audit_upd_rule，使用 list_
replace_
rcu 修改链表元素，并用 call_rcu 注册回调函数
*/
static           inline           int
audit_upd_rule(struct
  audit_rule *rule, struct list_head
  *list, __u32 newaction, __u32
  newfield_count)
{
  struct audit_entry *e;
  struct audit_newentry *ne;

  /* 遍历链表 */
  list_for_each_entry(e, list, list)
  {
    if (!audit_compare_rule(rule, &e
      ->rule))
    {
      ne = kmalloc(sizeof(*entry),
        GFP_ATOMIC);//分配新元素内存
      if (ne == NULL)
        return  - ENOMEM;
      audit_copy_rule(&ne->rule, &e
        ->rule);//写前拷贝
      ne->rule.action = newaction;
      ne->rule.file_count =
        newfield_count;
      list_replace_rcu(e, ne);
```

```
    list_add_tail(&entry->list,
list);
  }
  write_unlock(&auditsc_lock);
  return 0;
}


/*原先的 audit_upd_rule 函数，在修改
链表元素前获得读写锁 */
static   inline   int   audit_upd_
rule(struct
  audit_rule    *rule,    struct
list_head
  *list, __u32 newaction, __u32
  newfield_count)
{
  struct audit_entry *e;
  struct audit_newentry *ne;
  write_lock(&auditsc_lock);
  /* 遍历链表 */
  list_for_each_entry(e,   list,
list)
  {
    if (!audit_compare_rule(rule,
&e
      ->rule))
    {
      e->rule.action = newaction;
      e->rule.file_count =
        newfield_count;

write_unlock(&auditsc_lock);
call_rcu(&e->rcu,audit_free_rule,e);
      return 0;
    }
  }


  return  - EFAULT;
}
```

华清远见

7.5 信号量

7.5.1 信号量的使用

信号量（semaphore）是用于保护临界区的一种常用方法，它的使用方式和自旋锁类似。与自旋锁相同，只有得到信号量的进程才能执行临界区代码。但是，与自旋锁不同的是，当获取不到信号量时，进程不会原地打转而是进入休眠等待状态。

Linux 系统中与信号量相关的操作主要有如下 4 种。

1．定义信号量

下列代码定义名称为 sem 的信号量。

```
struct semaphore sem;
```

2．初始化信号量

```
void sema_init (struct semaphore *sem, int val);
```

该函数初始化信号量，并设置信号量 sem 的值为 val。尽管信号量可以被初始化为大于 1 的值从而成为一个计数信号量，但是它通常不被这样使用。

```
void init_MUTEX(struct semaphore *sem);
```

该函数用于初始化一个用于互斥的信号量，它把信号量 sem 的值设置为 1，等同于 sema_init (struct semaphore *sem, 1)。

```
void init_MUTEX_LOCKED (struct semaphore *sem);
```

该函数也用于初始化一个信号量，但它把信号量 sem 的值设置为 0，等同于 sema_init (struct semaphore *sem, 0)。

此外，下面两个宏是定义并初始化信号量的“快捷方式”。

```
DECLARE_MUTEX(name)
DECLARE_MUTEX_LOCKED(name)
```

前者定义一个名为 name 的信号量并初始化为 1，后者定义一个名为 name 的信号量并初始化为 0。

3．获得信号量

```
void down(struct semaphore * sem);
```

该函数用于获得信号量 sem，它会导致睡眠，因此不能在中断上下文使用。

```
int down_interruptible(struct semaphore * sem);
```

该函数功能与 down()类似，不同之处为，因为 down()而进入睡眠状态的进程不能被信号打断，而因为 down_interruptible()而进入睡眠状态的进程能被信号打断，信号也会导致该函数返回，这时候函数的返回值非 0。

```
int down_trylock(struct semaphore * sem);
```

该函数尝试获得信号量 sem，如果能够立刻获得，它就获得该信号量并返回 0，否则，返回非 0 值。它不会导致调用者睡眠，可以在中断上下文使用。

在使用 down_interruptible()获取信号量时，对返回值一般会进行检查，如果非 0，通常立即返回-ERESTARTSYS，如：

```
if (down_interruptible(&sem))
{
return  - ERESTARTSYS;
}
```

4．释放信号量

```
void up(struct semaphore * sem);
```

该函数释放信号量 sem，唤醒等待者。

信号量一般这样被使用，如下所示：

```
//定义信号量
DECLARE_MUTEX(mount_sem);
down(&mount_sem);//获取信号量，保护临界区
...
critical section //临界区
...
up(&mount_sem);//释放信号量
```

Linux 自旋锁和信号量所采用的“获取锁—访问临界区—释放锁”的方式存在于几乎所有的多任务操作系统之中。

代码清单 7.3 给出了使用信号量实现设备只能被一个进程打开的例子，等同于代码清单 7.1 和代码清单 7.2。

代码清单 7.3　使用信号量实现设备只能被一个进程打开

```
static DECLARE_MUTEX(xxx_lock);//定义互斥锁

static int xxx_open(struct inode *inode, struct file *filp)
{
  ...
  if (down_trylock(&xxx_lock))  //获得打开锁
      return  - EBUSY;  //设备忙
  ...
  return 0; /* 成功 */
}

static int xxx_release(struct inode *inode, struct file *filp)
{
  up(&xxx_lock);  //释放打开锁
  return 0;
}
```

7.5.2　信号量用于同步

如果信号量被初始化为 0，则它可以用于同步，同步意味着一个执行单元的继续执行需等待另一执行单元完成某事，保证执行的先后顺序。如图 7.4 所示，执行单元 A 执行代码区域 b 之前，必须等待执行单元 B 执行完代码单元 c，信号量可辅助这一同步过程。

图 7.4　信号量用于同步

7.5.3　完成量用于同步

Linux 系统提供了一种比信号量更好的同步机制，即完成量（completion，关于这个名词，至今没有好的翻译，笔者将其译为“完成量”），它用于一个执行单元等待另一个执行单元执行完某事。

Linux 系统中与 completion 相关的操作主要有以下 4 种。

1．定义完成量

下列代码定义名为 my_completion 的完成量。

```
struct completion my_completion;
```

2．初始化 completion

下列代码初始化 my_completion 这个完成量。

```
init_completion(&my_completion);
```

对 my_completion 的定义和初始化可以通过如下快捷方式实现。

```
DECLARE_COMPLETION(my_completion);
```

3．等待完成量

下列函数用于等待一个 completion 被唤醒。

```
void wait_for_completion(struct completion *c);
```

4．唤醒完成量

下面两个函数用于唤醒完成量。

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```

前者只唤醒一个等待的执行单元，后者释放所有等待同一完成量的执行单元。

图 7.5 描述了使用完成量实现的同步功能。

图 7.5 完成量用于同步

7.5.4 自旋锁 vs 信号量

自旋锁和信号量都是解决互斥问题的基本手段，面对特定的情况，应该如何进行选择呢？选择的依据是临界区的性质和系统的特点。

从严格意义上说，信号量和自旋锁属于不同层次的互斥手段，前者的实现依赖于后者。在信号量本身的实现上，为了保证信号量结构存取的原子性，在多 CPU 中需要自旋锁来互斥。

信号量是进程级的，用于多个进程之间对资源的互斥，虽然也是在内核中，但是该内核执行路径是以进程的身份，代表进程来争夺资源的。如果竞争失败，会发生进程上下文切换，当前进程进入睡眠状态，CPU 将运行其他进程。鉴于进程上下文切换的开销也很大，因此，只有当进程占用资源时间较长时，用信号量才是较好的选择。

当所要保护的临界区访问时间比较短时，用自旋锁是非常方便的，因为它节省上下文切换的时间。但是 CPU 得不到自旋锁会在那里空转直到其他执行单元解锁为止，所以要求锁不能在临界区里长时间停留，否则会降低系统的效率。

由此，可以总结出自旋锁和信号量选用的 3 项原则。

- 当锁不能被获取时，使用信号量的开销是进程上下文切换时间 Tsw，使用自旋锁的开销是等待获取自旋锁（由临界区执行时间决定）Tcs，若 Tcs 比较小，应使用自旋锁，若 Tcs 很大，应使用信号量。
- 信号量所保护的临界区可包含可能引起阻塞的代码，而自旋锁则绝对要避免用来保护包含这样代码的临界区。因为阻塞意味着要进行进程的切换，如果进程被切换出去后，另一个进程企图获取本自旋锁，死锁就会发生。
- 信号量存在于进程上下文，因此，如果被保护的共享资源需要在中断或软中断情况下使用，则在信号量和自旋锁之间只能选择自旋锁。当然，如果一定要使用信号量，则只能通过 down_trylock()方式进行，不能获取就立即返回以避免阻塞。

华清远见

7.5.5 读写信号量

读写信号量与信号量的关系与读写自旋锁和自旋锁的关系类似，读写信号量可能引起进程阻塞，但它可允许 *N* 个读执行单元同时访问共享资源，而最多只能有一个写执行单元。因此，读写信号量是一种相对放宽条件的粒度稍大于信号量的互斥机制。

读写自旋锁涉及的操作包括如下 5 种。

1．定义和初始化读写信号量

```
struct rw_semaphore my_rws;  /*定义读写信号量*/
void init_rwsem(struct rw_semaphore *sem);  /*初始化读写信号量*/
```

2．读信号量获取

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
```

3．读信号量释放

```
void up_read(struct rw_semaphore *sem);
```

4．写信号量获取

```
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
```

5．写信号量释放

```
void up_write(struct rw_semaphore *sem);
```

读写信号量一般这样被使用，如下所示：

```
rw_semaphore rw_sem;   //定义读写信号量
init_rwsem(&rw_sem); //初始化读写信号量

//读时获取信号量
down_read(&rw_sem);
...  //临界资源
up_read(&rw_sem);

//写时获取信号量
down_write(&rw_sem);
... //临界资源
up_write(&rw_sem);
```

7.6 互斥体

尽管信号量已经可以实现互斥的功能，而且包含 DECLARE_MUTEX()、init_MUTEX ()等定义信号量的宏或函数，从名字上看就体现出了互斥体的概念，但是 mutex 在 Linux 内核中还是真实地存在的。

下面代码定义名为 my_mutex 的互斥体并初始化它。

```
struct mutex my_mutex;
```

```
mutex_init(&my_mutex);
```

下面的两个函数用于获取互斥体。

```
void fastcall mutex_lock(struct mutex *lock);
int fastcall mutex_lock_interruptible(struct mutex *lock);
int fastcall mutex_trylock(struct mutex *lock);
```

mutex_lock()与 mutex_lock_interruptible()的区别和 down()与 down_trylock()的区别完全一致，前者引起的睡眠不能被信号打断，而后者可以。mutex_trylock()用于尝试获得 mutex，获取不到 mutex 时不会引起进程睡眠。

下列函数用于释放互斥体。

```
void fastcall mutex_unlock(struct mutex *lock);
```

mutex 的使用方法和信号量用于互斥的场合完全一样，如下所示：

```
struct mutex my_mutex;  //定义 mutex
mutex_init(&my_mutex);  //初始化 mutex

mutex_lock(&my_mutex);  //获取 mutex
...//临界资源
mutex_unlock(&my_mutex); //释放 mutex
```

7.7 增加并发控制后的 globalmem 驱动

在 globalmem()的读写函数中，由于要调用 copy_from_user()、copy_to_user()这些可能导致阻塞的函数，因此不能使用自旋锁，宜使用信号量。

驱动工程师习惯将某设备所使用的自旋锁、信号量等辅助手段也放在设备结构中，因此，可如代码清单 7.4 那样修改 globalmem_dev 结构体的定义，并在模块初始化函数中初始化这个信号量，如代码清单 7.5 所示。

代码清单 7.4　增加并发控制后的 globalmem 设备结构体

```
1 struct globalmem_dev
2 {
3   struct cdev cdev; /*cdev 结构体*/
4   unsigned char mem[GLOBALMEM_SIZE]; /*全局内存*/
5   struct semaphore sem; /*并发控制用的信号量*/
6 };
```

代码清单 7.5　增加并发控制后的 globalmem 设备驱动模块加载函数

```
1  int globalmem_init(void)
2  {
3    int result;
4    dev_t devno = MKDEV(globalmem_major, 0);
```

```
    5
    6   /* 申请设备号*/
    7   if (globalmem_major)
    8     result = register_chrdev_region(devno, 1, "globalmem");
    9   else  /* 动态申请设备号 */
    10  {
    11    result = alloc_chrdev_region(&devno, 0, 1, "globalmem");
    12    globalmem_major = MAJOR(devno);
    13  }
    14  if (result < 0)
    15    return result;
    16
    17  /* 动态申请设备结构体的内存*/
    18      globalmem_devp  =  kmalloc(sizeof(struct  globalmem_dev),
GFP_KERNEL);
    19  if (!globalmem_devp)   /*申请失败*/
    20  {
    21    result =  - ENOMEM;
    22    goto fail_malloc;
    23  }
    24  memset(globalmem_devp, 0, sizeof(struct globalmem_dev));
    25
    26  globalmem_setup_cdev(globalmem_devp, 0);
    27  init_MUTEX(&globalfifo_devp->sem);  /*初始化信号量*/
    28  return 0;
    29
    30  fail_malloc: unregister_chrdev_region(devno, 1);
    31  return result;
    32 }
```

在访问 globalmem_dev 中的共享资源时，需先获取这个信号量，访问完成后，随即释放这个信号量。驱动中新的 globalmem 读、写操作如代码清单 7.6 所示。

代码清单 7.6　增加并发控制后的 globalmem 读写操作

```
    1  /*增加并发控制后的 globalmem 读函数*/
    2  static ssize_t globalmem_read(struct file *filp, char __user *buf,
size_t size,
    3    loff_t *ppos)
    4  {
    5    unsigned long p =  *ppos;
    6    unsigned int count = size;
    7    int ret = 0;
```

```
    8   struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指
针*/
    9
    10  /*分析和获取有效的写长度*/
    11  if (p >= GLOBALMEM_SIZE)
    12    return count ?  - ENXIO: 0;
    13  if (count > GLOBALMEM_SIZE - p)
    14    count = GLOBALMEM_SIZE - p;
    15
    16  if (down_interruptible(&dev->sem)) //获得信号量
    17  {
    18    return  - ERESTARTSYS;
    19  }
    20
    21  /*内核空间->用户空间*/
    22  if (copy_to_user(buf, (void*)(dev->mem + p), count))
    23  {
    24    ret =  - EFAULT;
    25  }
    26  else
    27  {
    28    *ppos += count;
    29    ret = count;
    30
    31    printk(KERN_INFO "read %d bytes(s) from %d\n", count, p);
    32  }
    33  up(&dev->sem); //释放信号量
    34
    35  return ret;
    36 }
    37
    38 /*增加并发控制后的globalmem写函数*/
    39 static ssize_t globalmem_write(struct file *filp, const char __user
*buf,
    40  size_t size, loff_t *ppos)
    41 {
    42  unsigned long p =  *ppos;
    43  unsigned int count = size;
    44  int ret = 0;
    45  struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指
针*/
    46
    47  /*分析和获取有效的写长度*/
    48  if (p >= GLOBALMEM_SIZE)
```

```
49    return count ?  - ENXIO: 0;
50   if (count > GLOBALMEM_SIZE - p)
51    count = GLOBALMEM_SIZE - p;
52
53   if (down_interruptible(&dev->sem)) //获得信号量 54
54   {
55    return  - ERESTARTSYS;
56   }
57   /*用户空间->内核空间*/
58   if (copy_from_user(dev->mem + p, buf, count))
59    ret =  - EFAULT;
60   else
61   {
62    *ppos += count;
63    ret = count;
64
65    printk(KERN_INFO "written %d bytes(s) from %d\n", count, p);
66   }
67   up(&dev->sem); //释放信号量
68   return ret;
69 }
```

代码第 16～19 和第 53～56 行用于获取信号量，如果 down_interruptible()返回值非 0，则意味着其在获得信号量之前已被打断，这时写函数返回-ERESTARTSYS。代码第 33 和第 67 行用于在对临界资源访问结束后释放信号量。

除了 globalmem 的读写操作之外，如果在读写的同时，另一执行单元执行 MEM_CLEAR IO 控制命令，也会导致全局内存的混乱，因此，globalmem_ioctl()函数也需被重写，如代码清单 7.7 所示。

代码清单 7.7　增加并发控制后的 globalmem 设备驱动 ioctl()函数

```
1  static int globalmem_ioctl(struct inode *inodep, struct file *filp,
unsigned
2   int cmd, unsigned long arg)
3  {
4   struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指
针*/
5
6   switch (cmd)
7   {
8    case MEM_CLEAR:
9     if (down_interruptible(&dev->sem))//获得信号量
10    {
11     return  - ERESTARTSYS;
12    }
13    memset(dev->mem, 0, GLOBALMEM_SIZE);
14    up(&dev->sem); //释放信号量
15
16    printk(KERN_INFO "globalmem is set to zero\n");
17    break;
18
19   default:
20    return  - EINVAL;
21  }
22  return 0;
23 }
```

下面是增加并发控制后的 globalmem 驱动针对代码清单 6.16 未考虑并发问题

globalmem 驱动的 patch。

```
28a30
>   struct semaphore sem; /*并发控制用的信号量*/
53a58,61
>   if (down_interruptible(&dev->sem))
>   {
>     return  - ERESTARTSYS;
>   }
54a63,64
>   up(&dev->sem); //释放信号量
>
78a89,93
>   if (down_interruptible(&dev->sem))
>   {
>    return  - ERESTARTSYS;
>   }
>
90a106
>   up(&dev->sem); //释放信号量
109a126,129
>   if (down_interruptible(&dev->sem))//获得信号量
>   {
>    return  - ERESTARTSYS;
>   }
120c140
<
---
>   up(&dev->sem); //释放信号量
216a237
>   init_MUTEX(&globalfifo_devp->sem);   /*初始化信号量*/
```

7.8 总结

并发和竞态广泛存在，中断屏蔽、原子操作、自旋锁和信号量都是解决并发问题的机制。中断屏蔽很少单独被使用，原子操作只能针对整数进行，因此自旋锁和信号量应用最为广泛。

自旋锁会导致死循环，锁定期间不允许阻塞，因此要求锁定的临界区小。信号量允许临界区阻塞，可以适用于临界区大的情况。

读写自旋锁和读写信号量分别是放宽了条件的自旋锁和信号量，它们允许多个执行单元对共享资源的并发读。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

· 招生简章：http://www.embedu.org/courses/index.htm

第 8 章 Linux 设备驱动中的阻塞与非阻塞 I/O

阻塞和非阻塞 I/O 是设备访问的两种不同模式，驱动程序可以灵活地支持用户空间对设备的这两种访问方式。

8.1 节讲解阻塞 I/O 和非阻塞 I/O 的区别，实现阻塞 I/O 的等待队列机制，以及在 globalfifo 设备驱动中增加对阻塞 I/O 支持的方法，并进行了用户空间的验证。

8.2 节讲解设备驱动的轮询（poll）操作的概念和编程方法，poll 操作可以帮助用户了解是否能对设备进行无阻塞访问。

8.3 节讲解在 globalfifo 中增加 poll 操作的方法，并进行了用户空间的验证。

华清远见

8.1 阻塞与非阻塞 I/O

阻塞操作是指在执行设备操作时若不能获得资源则挂起进程，直到满足可操作的条件后再进行操作。被挂起的进程进入休眠状态，被从调度器的运行队列移走，直到等待的条件被满足。而非阻塞操作的进程在不能进行设备操作时并不挂起，它或者放弃，或者不停地查询，直至可以进行操作为止。

驱动程序通常需要提供这样的能力：当应用程序进行 read()、write()等系统调用时，若设备的资源不能获取，而用户又希望以阻塞的方式访问设备，驱动程序应在设备驱动的 xxx_read()、xxx_write()等操作中将进程阻塞直到资源可以获取，此后，应用程序的 read()、write()等调用才返回，整个过程仍然进行了正确的设备访问，用户并没有感知到；若用户以非阻塞的方式访问设备文件，则当设备资源不可获取时，设备驱动的 xxx_read()、xxx_write()等操作应立即返回，read()、write()等系统调用也随即被返回。

阻塞从字面上听起来似乎意味着低效率，实则不然，如果设备驱动不阻塞，则用户想获取设备资源只能不停地查询，这反而会无谓地耗费 CPU 资源。而阻塞访问时，不能获取资源的进程将进入休眠，它将 CPU 资源让给其他进程。

因为阻塞的进程会进入休眠状态，因此，必须确保有一个地方能够唤醒休眠的进程。唤醒进程的地方最大可能发生在中断里面，因为硬件资源获得的同时往往伴随着一个中断。

代码清单 8.1 和代码清单 8.2 分别演示了以阻塞和非阻塞方式读取串口一个字符的代码。实际的串口编程中，若使用非阻塞模式，还可借助信号（sigaction）以异步方式访问串口以提高 CPU 利用率，而这里仅仅是为了说明阻塞与非阻塞的区别。

代码清单 8.1　阻塞地读取串口一个字符

```
char buf;
fd = open("/dev/ttyS1", O_RDWR);
...
res = read(fd,&buf,1); //当串口上有输入时才返回
if(res==1)
        printf("%c\n", buf);
```

代码清单 8.2　非阻塞地读取串口一个字符

```
char buf;
fd = open("/dev/ttyS1", O_RDWR| O_NONBLOCK);
...
while(read(fd,&buf,1)!=1); //串口上无输入也返回，所以要循环尝试读取串口
printf("%c\n", buf);
```

8.1.1 等待队列

在 Linux 驱动程序中，可以使用等待队列（wait queue）来实现阻塞进程的唤醒。wait queue 很早就作为一个基本的功能单位出现在 Linux 内核里了，它以队列为基础数据结构，与进程调度机制紧密结合，能够用于实现内核中的异步事件通知机制。等待队列可以用来同步对系统资源的访问，第 7 章中所讲述的信号量在内核中也依赖等待队列来实现。

Linux 2.6 提供如下关于等待队列的操作。

1．定义“等待队列头”。

```
wait_queue_head_t my_queue;
```

2．初始化“等待队列头”。

```
init_waitqueue_head(&my_queue);
```

而下面的 DECLARE_WAIT_QUEUE_HEAD()宏可以作为定义并初始化等待队列头的“快捷方式”。

```
DECLARE_WAIT_QUEUE_HEAD (name)
```

3．定义等待队列。

```
DECLARE_WAITQUEUE(name, tsk)
```

该宏用于定义并初始化一个名为 name 的等待队列。

4．添加/移除等待队列。

```
void fastcall add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);
void fastcall remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);
```

add_wait_queue()用于将等待队列 wait 添加到等待队列头 q 指向的等待队列链表中，而 remove_wait_queue()用于将等待队列 wait 从附属的等待队列头 q 指向的等待队列链表中移除。

5．等待事件。

```
wait_event(queue, condition)
wait_event_interruptible(queue, condition)
wait_event_timeout(queue, condition, timeout)
wait_event_interruptible_timeout(queue, condition, timeout)
```

等待第一个参数 queue 作为等待队列头的等待队列被唤醒，而且第二个参数 condition 必须满足，否则阻塞。wait_event()和 wait_event_interruptible()的区别在于后者可以被信号打断，而前者不能。加上_timeout 后的宏意味着阻塞等待的超时时间，以 jiffy 为单位，在第三个参数的 timeout 到达时，不论 condition 是否满足，均返回。

wait()的定义如代码清单 8.3 所示，从其源代码可以看出，当 condition 满足时，

wait_event()会立即返回，否则，阻塞等待 condition 满足。

代码清单 8.3 wait_event()函数

```
#define wait_event(wq, condition)                                \
do {                                                     \
 if (condition)  /*条件满足立即返回*/                     \
     break;                                              \
 __wait_event(wq, condition);/*添加等待队列并阻塞*/
} while (0)

#define __wait_event(wq, condition)                              \
do {                                                     \
 DEFINE_WAIT(__wait);                                    \
                                                         \
 for (;;) {                                              \
     prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);  \
     if (condition)                                      \
         break;                                          \
     schedule();/*放弃 CPU*/                              \
 }                                                       \
 finish_wait(&wq, &__wait);                              \
} while (0)

void fastcall
prepare_to_wait(wait_queue_head_t *q, wait_queue_t *wait, int state)
{
 unsigned long flags;

 wait->flags &= ~WQ_FLAG_EXCLUSIVE;
 spin_lock_irqsave(&q->lock, flags);
 if (list_empty(&wait->task_list))
     __add_wait_queue(q, wait);//添加等待队列
 if (is_sync_wait(wait))
     set_current_state(state);//改变当前进程的状态为休眠
 spin_unlock_irqrestore(&q->lock, flags);
}

void fastcall finish_wait(wait_queue_head_t *q, wait_queue_t *wait)
{
 unsigned long flags;
```

```
    38
    39  __set_current_state(TASK_RUNNING);// 恢复当前进程的状态为
TASK_RUNNING
    40  if (!list_empty_careful(&wait->task_list)) {
    41      spin_lock_irqsave(&q->lock, flags);
    42      list_del_init(&wait->task_list);
    43      spin_unlock_irqrestore(&q->lock, flags);
    44  }
    45 }
```

6．唤醒队列。

```
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

上述操作会唤醒以 queue 作为等待队列头的所有等待队列中所有属于该等待队列头的等待队列对应的进程。

wake_up() 应与 wait_event() 或 wait_event_timeout() 成对使用，而 wake_up_interruptible() 则应与 wait_event_interruptible() 或 wait_event_interruptible_timeout() 成对使用。wake_up() 可唤醒处于 TASK_INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE 的进程，而 wake_up_interruptible()只能唤醒处于 TASK_INTERRUPTIBLE 的进程。

7．在等待队列上睡眠。

```
sleep_on(wait_queue_head_t *q );
interruptible_sleep_on(wait_queue_head_t *q );
```

sleep_on()函数的作用就是将目前进程的状态置成 TASK_UNINTERRUPTIBLE，并定义一个等待队列，之后把它附属到等待队列头 q，直到资源可获得，q 引导的等待队列被唤醒。

interruptible_sleep_on()与 sleep_on()函数类似，其作用是将目前进程的状态置成 TASK_ INTERRUPTIBLE，并定义一个等待队列，之后把它附属到等待队列头 q，直到资源可获得，q 引导的等待队列被唤醒或者进程收到信号。

sleep_on() 函数应该与 wake_up() 成对使用，interruptible_sleep_on() 应该与 wake_up_interruptible()成对使用。

代码清单 8.4 和代码清单 8.5 分别列出了 sleep_on()和 interruptible_sleep_on()函数的源代码。

代码清单 8.4　sleep_on()函数

```
void fastcall __sched interruptible_sleep_on(wait_queue_head_t *q)
{
 SLEEP_ON_VAR
  /* #define SLEEP_ON_VAR                    \
    unsigned long flags;                     \
    wait_queue_t wait;                       \
    init_waitqueue_entry(&wait, current); */
 current->state = TASK_UNINTERRUPTIBLE;//改变当前进程状态

 SLEEP_ON_HEAD
  /* #define SLEEP_ON_HEAD                   \
   spin_lock_irqsave(&q->lock,flags);        \
   __add_wait_queue(q, &wait);               \
   spin_unlock(&q->lock);*/

 schedule();//放弃 CPU
 SLEEP_ON_TAIL
  /* #define SLEEP_ON_TAIL                   \
    spin_lock_irq(&q->lock);                 \
    __remove_wait_queue(q, &wait);           \
    spin_unlock_irqrestore(&q->lock, flags); */
}
```

代码清单 8.5　interruptible_sleep_on()函数

```
void fastcall __sched interruptible_sleep_on(wait_queue_head_t *q)
{
 SLEEP_ON_VAR

 current->state = TASK_INTERRUPTIBLE;//改变当前进程状态

 SLEEP_ON_HEAD
 schedule();//放弃 CPU
 SLEEP_ON_TAIL
}
```

从代码清单 8.4 和代码清单 8.5 可以看出，不论是 sleep_on()还是 interruptible_sleep_on()，其流程都如下所示。

（1）定义并初始化一个等待队列，将进程状态改变为 TASK_UNINTERRUPTIBLE（不能被信号打断）或 TASK_INTERRUPTIBLE（可以被信号打断），并将等待队列添加到等待队列头。

（2）通过 schedule()放弃 CPU，调度其他进程执行。

（3）进程被其他地方唤醒，将等待队列移出等待队列头。

在内核中使用 set_current_state()函数或__add_wait_queue()函数来实现目前进程状态的改变，直接采用 current->state = TASK_UNINTERRUPTIBLE 类似的赋值语句也是可行的。通常而言，set_current_state()函数在任何环境下都可以使用，不会存在并发问题，但是效率要低于__add_ wait_queue()。

因此，在许多设备驱动中，并不调用 sleep_on()或 interruptible_sleep_on()，而是亲自进行进程的状态改变和切换，如代码清单 8.6 所示。

代码清单 8.6　在驱动程序中改变进程状态并调用 schedule()

```
static ssize_t xxx_write(struct file *file, const char *buffer, size_t count,
  loff_t *ppos)
{
  ...
  DECLARE_WAITQUEUE(wait, current); //定义等待队列
  __add_wait_queue(&xxx_wait, &wait);  //添加等待队列

  ret = count;
  /* 等待设备缓冲区可写 */
  do
  {
    avail = device_writable(...);
    if (avail < 0)
      __set_current_state(TASK_INTERRUPTIBLE);//改变进程状态

    if (avail < 0)
    {
      if (file->f_flags &O_NONBLOCK)  //非阻塞
      {
        if (!ret)
          ret =  - EAGAIN;
        goto out;
      }
      schedule();   //调度其他进程执行
      if (signal_pending(current))//如果是因为信号唤醒
      {
        if (!ret)
          ret =  - ERESTARTSYS;
        goto out;
      }
    }
  }while (avail < 0);

  /* 写设备缓冲区 */
  device_write(...)
  out:
  remove_wait_queue(&xxx_wait, &wait);//将等待队列移出等待队列头
  set_current_state(TASK_RUNNING);//设置进程状态为 TASK_RUNNING
```

```
39  return ret;
40 }
```

8.1.2 支持阻塞操作的 globalfifo 设备驱动

现在我们给 globalfifo 增加这样的约束：把 globalfifo 中的全局内存变成一个 FIFO，只有当 FIFO 中有数据的时候（即有进程把数据写到这个 FIFO 而且没有被读进程读空），读进程才能把数据读出，而且读取后的数据会从 globalfifo 的全局内存中被拿掉；只有当 FIFO 非满时（即还有一些空间未被写，或写满后被读进程从这个 FIFO 中读出了数据），写进程才能往这个 FIFO 中写入数据。

现在，将 globalfifo 重命名为“globalfifo”，在 globalfifo 中，读 FIFO 将唤醒写 FIFO，而写 FIFO 也将唤醒读 FIFO。首先，需要修改设备结构体，在其中增加两个等待队列头，分别对应于读和写，如代码清单 8.7 所示。

代码清单 8.7　globalfifo 设备结构体

```
1 struct globalfifo_dev
2 {
3    struct cdev cdev; /*cdev 结构体*/
4    unsigned int current_len;   /*fifo 有效数据长度*/
5    unsigned char mem[GLOBALFIFO_SIZE]; /*全局内存*/
6    struct semaphore sem; /*并发控制用的信号量*/
7    wait_queue_head_t r_wait; /*阻塞读用的等待队列头*/
8    wait_queue_head_t w_wait; /*阻塞写用的等待队列头*/
9 };
```

与 globalfifo 设备结构体的另一个不同是增加了 current_len 成员用于表征目前 FIFO 中有效数据的长度。

这个等待队列需在设备驱动模块加载函数中调用 init_waitqueue_head()被初始化，新的设备驱动模块加载函数如代码清单 8.8 所示。

代码清单 8.8　globalfifo 设备驱动模块加载函数

```
1  int globalfifo_init(void)
2  {
3    int ret;
4    dev_t devno = MKDEV(globalfifo_major, 0);
5
6    /* 申请设备号*/
7    if (globalfifo_major)
8      ret = register_chrdev_region(devno, 1, "globalfifo");
9    else  /* 动态申请设备号 */
10   {
11     ret = alloc_chrdev_region(&devno, 0, 1, "globalfifo");
12     globalfifo_major = MAJOR(devno);
```

```
    }
    if (ret < 0)
      return ret;
    /* 动态申请设备结构体的内存*/
      globalfifo_devp = kmalloc(sizeof(struct globalfifo_dev), GFP_KERNEL);
    if (!globalfifo_devp)   /*申请失败*/
    {
      ret = - ENOMEM;
      goto fail_malloc;
    }

    memset(globalfifo_devp, 0, sizeof(struct globalfifo_dev));

    globalfifo_setup_cdev(globalfifo_devp, 0);

    init_MUTEX(&globalfifo_devp->sem);   /*初始化信号量*/
    init_waitqueue_head(&globalfifo_devp->r_wait); /*初始化读等待队列头*/
    init_waitqueue_head(&globalfifo_devp->w_wait); /*初始化写等待队列头*/

    return 0;

    fail_malloc: unregister_chrdev_region(devno, 1);
    return ret;
}
```

设备驱动读写操作需要被修改，在读函数中需增加等待 globalfifo_devp->w_wait 被唤醒的语句，而在写操作中唤醒 globalfifo_devp->r_wait，如代码清单 8.9 所示。

代码清单 8.9　增加等待队列后的 globalfifo 读写函数

```
/*globalfifo 读函数*/
static ssize_t globalfifo_read(struct file *filp, char __user *buf, size_t
    count, loff_t *ppos)
{
    int ret;
    struct globalfifo_dev *dev = filp->private_data; //获得设备结构体指针
    DECLARE_WAITQUEUE(wait, current); //定义等待队列

```

```
    down(&dev->sem); //获得信号量
    add_wait_queue(&dev->r_wait, &wait); //进入读等待队列头

    /* 等待 FIFO 非空 */
    if (dev->current_len == 0)
    {
        if (filp->f_flags &O_NONBLOCK)
        {
            ret =  - EAGAIN;
            goto out;
        }
        __set_current_state(TASK_INTERRUPTIBLE); //改变进程状态为睡眠
        up(&dev->sem);

        schedule(); //调度其他进程执行
        if (signal_pending(current))
        //如果是因为信号唤醒
        {
            ret =  - ERESTARTSYS;
            goto out2;
        }

        down(&dev->sem);
    }

    /* 复制到用户空间 */
    if (count > dev->current_len)
        count = dev->current_len;

    if (copy_to_user(buf, dev->mem, count))
    {
        ret =  - EFAULT;
        goto out;
    }
    else
    {
        memcpy(dev->mem, dev->mem + count, dev->current_len - count); //fifo 数据前移
        dev->current_len -= count; //有效数据长度减少
```

```
        printk(KERN_INFO "read %d bytes(s),current_len:%d\n", count, dev
            ->current_len);

        wake_up_interruptible(&dev->w_wait); //唤醒写等待队列

        ret = count;
    }
    out: up(&dev->sem); //释放信号量
    out2: remove_wait_queue(&dev->w_wait, &wait); //从附属的等待队列头移除
    set_current_state(TASK_RUNNING);
    return ret;
}


/*globalfifo 写操作*/
static ssize_t globalfifo_write(struct file *filp, const char __user *buf,
    size_t count, loff_t *ppos)
{
    struct globalfifo_dev *dev = filp->private_data; //获得设备结构体指针
    int ret;
    DECLARE_WAITQUEUE(wait, current); //定义等待队列

    down(&dev->sem); //获取信号量
    add_wait_queue(&dev->w_wait, &wait); //进入写等待队列头

    /* 等待 FIFO 非满 */
    if (dev->current_len == GLOBALFIFO_SIZE)
    {
        if (filp->f_flags &O_NONBLOCK)
        //如果是非阻塞访问
        {
            ret =  - EAGAIN;
            goto out;
        }
        __set_current_state(TASK_INTERRUPTIBLE); //改变进程状态为睡眠
        up(&dev->sem);

        schedule(); //调度其他进程执行
        if (signal_pending(current))
        //如果是因为信号唤醒
        {
            ret =  - ERESTARTSYS;
            goto out2;
        }

        down(&dev->sem); //获得信号量
    }

    /*从用户空间复制到内核空间*/
```

```
96      if (count > GLOBALFIFO_SIZE - dev->current_len)
97          count = GLOBALFIFO_SIZE - dev->current_len;
98
99      if (copy_from_user(dev->mem + dev->current_len, buf, count))
100     {
101         ret =  - EFAULT;
102         goto out;
103     }
104     else
105     {
106         dev->current_len += count;
107          printk(KERN_INFO "written %d bytes(s),current_len:%d\n",
count, dev
108            ->current_len);
109
110         wake_up_interruptible(&dev->r_wait); //唤醒读等待队列
111
112         ret = count;
113     }
114
115     out: up(&dev->sem); //释放信号量
116     out2: remove_wait_queue(&dev->w_wait, &wait); //从附属的等待队列
头移除
117     set_current_state(TASK_RUNNING);
118     return ret;
119 }
```

代码清单 8.9 处理了等待队列进出和进程切换的过程。是否可以把读函数中一大段用于等待 dev->current_len != 0 的内容直接用 wait_event_interruptible（dev->r_wait, dev->current_len != 0）替换，把写函数中一大段用于等待 dev->current_len != GLOBALFIFO_SIZE 的代码用 wait_event_ interruptible（dev->w_wait, dev->current_len != 0）替换呢？

实际上，就控制等待队列非空和非满的角度而言，wait_event_interruptible（dev->r_wait, dev-> current_len != 0）和第 13～32 行代码的功能完全一样，wait_event_interruptible（dev->w_wait, dev-> current_len != 0）和第 73～93 行代码的功能完全一样。细微的区别体现在第 13～32 行代码和第 73～93 行代码在进行 schedule() 即切换进程前，通过 up（&dev->sem）释放了信号量。这一细微的动作意义重大，非如此，则死锁将不可避免。

如图 8.1（a）所示，假设目前的 FIFO 为空，即 dev->current_len 为 0，此时如果有一个读进程，它会先获得信号量，因为条件不满足，它将因为 wait_event_interruptible(dev->r_wait, dev-> current_len != 0) 而阻塞，而释放 dev->r_wait 等待队列及让 dev->current_len != 0 的操作又需要在写进程中进行，写进程在执行写操作前又必须等待读进程释放信号量，造成互相等待对方资源的矛盾局面，从而死锁。

如图 8.1(b)所示，假设目前的 FIFO 为满即 dev->current_len 为 GLOBALFIFO_SIZE，此时如果有一个写进程，它先获得信号量，因为条件不满足，它将因为 wait_event_interruptible(dev->w_wait, dev->current_len != GLOBALFIFO_SIZE)而阻塞，

而释放 dev->w_wait 等待队列及让 dev->current_ len != GLOBALFIFO_SIZE 的操作又需要在读进程中进行，读进程在执行读操作前又必须等待写进程释放信号量，造成互相等待对方资源的矛盾局面，从而死锁。

所谓死锁，就是多个进程循环等待它方占有的资源而无限期地僵持下去的局面。如果没有外力的作用，那么死锁涉及的各个进程都将永远处于封锁状态。因此，驱动工程师一定要注意：当多个等待队列、信号量等机制同时出现时，谨防死锁。

现在回过来看一下代码清单 8.9 的第 15 行和第 75 行，发现在设备驱动的 read()、write()等功能函数中，可以通过 filp->f_flags 标志获得用户空间是否要求非阻塞访问。驱动中可以依据此标志判断用户究竟要求阻塞还是非阻塞访问，从而进行不同的处理。

图 8.1　等待队列、信号量等引起的死锁

8.1.3　在用户空间验证 globalfifo 的读写

编译 globalfifo.c 并 insmod 模块即创建设备文件节点“/dev/globalfifo”后，启动两个进程，一个进程“cat /dev/globalfifo&”在后台执行，另一个进程“echo 字符串 /dev/globalfifo”在前台执行，如下所示：

```
[root@localhost root]# cat /dev/globalfifo&
[1] 4059
[root@localhost root]# echo 'I want to be' > /dev/globalfifo
[root@localhost root]# I want to be

[root@localhost root]# echo 'a great Chinese Linux driver Engineer' >
/dev/globalfifo
```

```
[root@localhost root]# a great Chinese Linux driver Engineer
```

每当 echo 进程向/dev/globalfifo 写入一串数据，cat 进程就立即将该串数据显现出来。

8.2 轮询操作

8.2.1 轮询的概念与作用

在用户程序中，select()和 poll()也是与设备阻塞与非阻塞访问息息相关的论题。使用非阻塞 I/O 的应用程序通常会使用 select()和 poll()系统调用查询是否可对设备进行无阻塞的访问。select()和 poll()系统调用最终会引发设备驱动中的 poll()函数被执行，在 2.5.45 内核中还引入了 epoll()，即扩展的 poll()。

select()和 poll()系统调用的本质一样，前者在 BSD UNIX 中引入的，后者在 System V 中引入的。

华清远见

8.2.2 应用程序中的轮询编程

应用程序中最广泛用到的是 BSD UNIX 中引入的 select()系统调用，其原型如下：

```
int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);
```

其中 readfds、writefds、exceptfds 分别是被 select()监视的读、写和异常处理的文件描述符集合，numfds 的值是需要检查的号码最高的文件描述符加 1。timeout 参数是一个指向 struct timeval 类型的指针，它可以使 select()在等待 timeout 时间后若没有文件描述符准备好则返回。struct timeval 数据结构的定义如代码清单 8.10 所示。

代码清单 8.10 timeval 结构体定义

```
struct timeval
{
    int tv_sec; /* 秒 */
    int tv_usec; /* 微妙 */
};
```

下列操作用来设置、清除、判断文件描述符集合。

```
FD_ZERO(fd_set *set)
```

清除一个文件描述符集。

```
FD_SET(int fd,fd_set *set)
```

将一个文件描述符加入文件描述符集中。

```
FD_CLR(int fd,fd_set *set)
```

将一个文件描述符从文件描述符集中清除。

```
FD_ISSET(int fd,fd_set *set)
```

判断文件描述符是否被置位。

8.2.3 设备驱动中的轮询编程

设备驱动中 poll()函数的原型如下：

```
unsigned int(*poll)(struct file * filp, struct poll_table* wait);
```

第一个参数为 file 结构体指针，第二个参数为轮询表指针。这个函数应该进行以下两项工作。

- 对可能引起设备文件状态变化的等待队列调用 poll_wait()函数，将对应的等待队列头添加到 poll_table。
- 返回表示是否能对设备进行无阻塞读、写访问的掩码。

关键的用于向 poll_table 注册等待队列的 poll_wait()函数的原型如下：

```
void poll_wait(struct file *filp, wait_queue_heat_t *queue, poll_table
* wait);
```

poll_wait()函数的名称非常容易让人产生误会，以为它和 wait_event()等一样，会阻塞地等待某事件的发生，其实这个函数并不会引起阻塞。poll_wait()函数所做的工作是把当前进程添加到 wait 参数指定的等待列表（poll_table）中。

驱动程序 poll()函数应该返回设备资源的可获取状态，即 POLLIN、POLLOUT、POLLPRI、POLLERR、POLLNVAL 等宏的位“或”结果。每个宏的含义都表明设备

的一种状态，如 POLLIN（定义为 0x0001）意味着设备可以无阻塞地读，POLLOUT（定义为 0x0004）意味着设备可以无阻塞地写。

通过以上分析，可得出设备驱动中 poll()函数的典型模板，如代码清单 8.11 所示。

华清远见

代码清单 8.11 poll()函数典型模板

```
static unsigned int xxx_poll(struct file *filp, poll_table *wait)
{
    unsigned int mask = 0;
    struct xxx_dev *dev = filp->private_data; /*获得设备结构体指针*/

    ...
    poll_wait(filp, &dev->r_wait, wait);//加读等待队列头
    poll_wait(filp, &dev->w_wait, wait);//加写等待队列头

    if (...)//可读
    {
        mask |= POLLIN | POLLRDNORM; /*标示数据可获得*/
    }

    if (...)//可写
    {
        mask |= POLLOUT | POLLWRNORM; /*标示数据可写入*/
    }

    ...
    return mask;
}
```

8.3 支持轮询操作的 globalfifo 驱动

8.3.1 在 globalfifo 驱动中增加轮询操作

在 globalfifo 的 poll()函数中，首先将设备结构体中的 r_wait 和 w_wait 等待队列头添加到等待列表，然后通过判断 dev->current_len 是否等于 0 来获得设备的可读状态，通过判断 dev->current_len 是否等于 GLOBALFIFO_SIZE 来获得设备的可写状态，如代码清单 8.12 所示。

代码清单 8.12 globalfifo 设备驱动的 poll()函数

```
static unsigned int globalfifo_poll(struct file *filp, poll_table *wait)
{
    unsigned int mask = 0;
```

```
    struct globalfifo_dev *dev = filp->private_data; /*获得设备结构体指针*/

    down(&dev->sem);

    poll_wait(filp, &dev->r_wait, wait);
    poll_wait(filp, &dev->w_wait, wait);
    /*fifo 非空*/
    if (dev->current_len != 0)
    {
        mask |= POLLIN | POLLRDNORM; /*标示数据可获得*/
    }
    /*fifo 非满*/
    if (dev->current_len != GLOBALFIFO_SIZE)
    {
        mask |= POLLOUT | POLLWRNORM; /*标示数据可写入*/
    }

    up(&dev->sem);
    return mask;
}
```

注意要把 globalfifo_poll 赋给 globalfifo_fops 的 poll 成员，如下所示：

```
static const struct file_operations globalfifo_fops =
{
  ..
  .poll = globalfifo_poll,
  ...
};
```

8.3.2 在用户空间验证 globalfifo 设备的轮询

编写一个应用程序 pollmonitor.c 用于监控 globalfifo 的可读写状态，这个程序如代码清单 8.13 所示。

代码清单 8.13 监控 globalfifo 是否可非阻塞读写的应用程序

```
#include ...

#define FIFO_CLEAR 0x1
#define BUFFER_LEN 20
main()
{
  int fd, num;
```

```
8   char rd_ch[BUFFER_LEN];
9   fd_set rfds,wfds; //读/写文件描述符集
10
11  /*以非阻塞方式打开/dev/globalfifo 设备文件*/
12  fd = open("/dev/globalfifo", O_RDONLY | O_NONBLOCK);
13  if (fd !=  - 1)
14  {
15    /*FIFO 清零*/
16    if (ioctl(fd, FIFO_CLEAR, 0) < 0)
17    {
18      printf("ioctl command failed\n");
19    }
20    while (1)
21    {
22      FD_ZERO(&rfds);
23      FD_ZERO(&wfds);
24      FD_SET(fd, &rfds);
25      FD_SET(fd, &wfds);
26
27      select(fd + 1, &rfds, &wfds, NULL, NULL);
28      /*数据可获得*/
29      if (FD_ISSET(fd, &rfds))
30      {
31          printf("Poll monitor:can be read\n");
32      }
33      /*数据可写入*/
34      if (FD_ISSET(fd, &wfds))
35      {
36          printf("Poll monitor:can be written\n");
37      }
38    }
39  }
40  else
41  {
42    printf("Device open failure\n");
43  }
44 }
```

运行时看到，当没有任何输入，即 FIFO 为空时，程序不断地输出“Poll monitor:can be written”；当通过 echo 向/dev/globalfifo 写入一些数据后，将输出“Poll monitor:can be read”和“Poll monitor:can be written”；如果不断地通过 echo 向/dev/globalfifo 写

入数据直至写满 FIFO，发现 pollmonitor 程序将只输出“Poll monitor:can be read”。对于 globalfifo 而言，不会出现既不能读、又不能写的情况。

8.4 总结

阻塞与非阻塞访问是 I/O 操作的两种不同模式，前者在 I/O 操作暂时不可进行时会让进程睡眠。

在设备驱动中阻塞 I/O 一般基于等待队列来实现，等待队列可用于同步驱动中事件发生的先后顺序。使用非阻塞 I/O 的应用程序也可借助轮询函数来查询设备是否能立即被访问，用户空间调用 select()和 poll()接口，设备驱动提供 poll()函数。设备驱动的 poll()本身不会阻塞，但是 poll()和 select()系统调用则会阻塞地等待文件描述符集合中的至少一个可访问或超时。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm
- 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程： 华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：
  http://www.farsight.com.cn/courses/TS-LinuxApp.htm
- 嵌入式 Linux 系统开发班：
  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm
- 嵌入式 Linux 驱动开发班：
  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm

华清远见

第 9 章 Linux 设备驱动中的异步通知与异步 I/O

在设备驱动中使用异步通知可以使得对设备的访问可进行时，由驱动主动通知应用程序进行访问。这样，使用无阻塞 I/O 的应用程序无须轮询设备是否可访问，而阻塞访问也可以被类似“中断”的异步通知所取代。

9.1 节讲解了异步通知的概念和作用，9.2 节讲解了 Linux 异步通知的编程方法，9.3 节讲解了增加异步通知的 globalfifo 驱动及其在用户空间的验证。

华清远见

9.1 异步通知的概念与作用

阻塞与非阻塞访问、poll()函数提供了较好的解决设备访问的机制，但是如果有了异步通知整套机制就更加完整了。

异步通知的意思是：一旦设备就绪，则主动通知应用程序，这样应用程序根本就不需要查询设备状态，这一点非常类似于硬件上“中断”的概念，比较准确的称谓是“信号驱动的异步 I/O”。信号是在软件层次上对中断机制的一种模拟，在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。

阻塞 I/O 意味着一直等待设备可访问后再访问，非阻塞 I/O 中使用 poll()意味着查询设备是否可访问，而异步通知则意味着设备通知自身可访问，实现了异步 I/O。由此可见，这几种方式 I/O 可以互为补充。

图 9.1 呈现了阻塞 I/O，结合 poll()的非阻塞 I/O 及异步通知在时间先后顺序上的不同。

图 9.1　阻塞、非阻塞 I/O、异步通知的区别

阻塞、非阻塞 I/O、异步通知本身没有优劣，应该根据不同的应用场景合理选择。

9.2 Linux 异步通知编程

9.2.1　Linux 信号

使用信号进行进程间通信（IPC）是 UNIX 系统中的一种传统机制，Linux 系统也支持这种机制。在 Linux 系统中，异步通知使用信号来实现，Linux 系统中可用的信号及其定义如表 9-1 所示。

华清远见

表 9-1 Linux 信号

信　　号	值	含　　义
SIGHUP	1	挂起
SIGINT	2	终端中断
SIGQUIT	3	终端退出
SIGILL	4	无效命令
SIGTRAP	5	跟踪陷阱
SIGIOT	6	IOT 陷阱
SIGBUS	7	BUS 错误
SIGFPE	8	浮点异常
SIGKILL	9	强行终止（不能被捕获或忽略）
SIGUSR1	10	用户定义的信号 1
SIGSEGV	11	无效的内存段处理
SIGUSR2	12	用户定义的信号 2
SIGPIPE	13	半关闭管道发生写操作
SIGALRM	14	计时器到期
SIGTERM	15	终止
SIGSTKFLT	16	堆栈错误
SIGCHLD	17	子进程已经停止或退出
SIGCONT	18	如果停止了，继续执行
SIGSTOP	19	停止执行（不能被捕获或忽略）
SIGTSTP	20	终端停止信号
SIGTTIN	21	后台进程需要从终端读取输入
SIGTTOU	22	后台进程需要向从终端写出
SIGURG	23	紧急的套接字事件
SIGXCPU	24	超额使用 CPU 分配的时间
SIGXFSZ	25	文件尺寸超额
SIGVTALRM	26	虚拟时钟信号
SIGPROF	27	时钟信号描述
SIGWINCH	28	窗口尺寸变化
SIGIO	29	I/O
SIGPWR	30	断电重启

华清远见

除了 SIGSTOP 和 SIGKILL 两个信号外，进程能够忽略或捕获其他的全部信号。一个信号被捕获的意思是当一个信号到达时有相应的代码处理它。如果一个信号没有被这个进程所捕获，内核将采用默认行为处理。

9.2.2　信号的接收

在用户程序中，为了捕获信号，可以使用 signal()函数来设置对应信号的处理函数，如下所示：

```
void (*signal(int signum, void (*handler))(int))(int);
```

该函数原型较难理解，它可以分解如下：

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

第一个参数指定信号的值，第二个参数指定针对前面信号值的处理函数，若为 SIG_IGN，表示忽略该信号；若为 SIG_DFL，表示采用系统默认方式处理信号；若为用户自定义的函数，则信号被捕获到后，该函数将被执行。

如果 signal()调用成功，它返回最后一次为信号 signum 绑定的处理函数 handler 值，失败则返回 SIG_ERR。

在进程执行时，按下［Ctrl+c］组合键将向其发出 SIGINT 信号，kill 正在运行的进程将向其发出 SIGTERM 信号，代码清单 9.1 的进程捕获这两个信号并输出信号值。

代码清单 9.1　signal()捕获信号范例

```
void sigterm_handler(int signo)
{
  printf("Have caught sig N.O. %d\n", signo);
  exit(0);
}

int main(void)
{
  signal(SIGINT, sigterm_handler);
  signal(SIGTERM, sigterm_handler);
  while(1);

  return 0;
}
```

除了 signal()函数外，sigaction()函数可用于改变进程接收到特定信号后的行为，它的原型如下：

```
int sigaction(int signum,const struct sigaction *act,struct sigaction *oldact);
```

该函数的第一个参数为信号的值，可以为除 SIGKILL 及 SIGSTOP 外的任何一个特定有效的信号。第二个参数是指向结构体 sigaction 的一个实例的指针，在结构体 sigaction 的实例中，指定了对特定信号的处理函数，若为空，则进程会以默认方式对信号处理。第三个参数 oldact 指向的对象用来保存原来对相应信号的处理函数，可指定 oldact 为 NULL。如果把第二、第三个参数都设为 NULL，那么该函数可用于检查信号的有效性。

先来看一个使用信号实现异步通知的例子，它通过 signal（SIGIO, input_handler）对标准输入文件描述符 STDIN_FILENO 启动信号机制。用户输入后，应用程序将接

收到 SIGIO 信号，其处理函数 input_handler()将被调用，如代码清单 9.2 所示。

代码清单 9.2 使用信号实现异步通知的应用程序实例

```
1  #include <sys/types.h>
2  #include <sys/stat.h>
3  #include <stdio.h>
4  #include <fcntl.h>
5  #include <signal.h>
6  #include <unistd.h>
7  #define MAX_LEN 100
8  void input_handler(int num)
9  {
10   char data[MAX_LEN];
11   int len;
12
13   //读取并输出 STDIN_FILENO 上的输入
14   len = read(STDIN_FILENO, &data, MAX_LEN);
15   data[len] = 0;
16   printf("input available:%s\n", data);
17 }
18
19 main()
20 {
21   int oflags;
22
23   //启动信号驱动机制
24   signal(SIGIO, input_handler);
25   fcntl(STDIN_FILENO, F_SETOWN, getpid());
26   oflags = fcntl(STDIN_FILENO, F_GETFL);
27   fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);
28
29   //最后进入一个死循环，仅为保持进程不终止，如果程序中没有这个死循环
30   //会立即执行完毕
31   while (1);
32 }
```

上述代码 24 行为 SIGIO 信号安装 input_handler()作为处理函数，第 25 行设置本进程为 STDIN_FILENO 文件的拥有者（owner），没有这一步内核不会知道应该将信号发给哪个进程。而为了启用异步通知机制，还需对设备设置 FASYNC 标志，第 26～27 行代码实现此目的。整个程序的执行效果如下：

```
[root@localhost driver_study]# ./signal_test
I am Chinese.
input available: I am Chinese.

I love Linux driver.
input available: I love Linux driver.
```

从中可以看出，当用户输入一串字符后，标准输入设备释放 SIGIO 信号，这个信号“中断”驱使对应的应用程序中的 input_handler()得以执行，将用户输入显示出来。

由此可见，为了在用户空间中能处理一个设备释放的信号，它必须完成以下 3 项工作。

- 通过 F_SETOWN IO 控制命令设置设备文件的拥有者为本进程，这样从设备驱动发出的信号才能被本进程接收到。
- 通过 F_SETFL IO 控制命令设置设备文件支持 FASYNC，即异步通知模式。

- 通过 signal()函数连接信号和信号处理函数。

9.2.3　信号的释放

在设备驱动和应用程序的异步通知交互中，仅仅在应用程序端捕获信号是不够的，因为信号没有的源头在设备驱动端。因此，应该在合适的时机让设备驱动释放信号，在设备驱动程序中增加信号释放的相关代码。

为了使设备支持异步通知机制，驱动程序中涉及以下 3 项工作。

- 支持 F_SETOWN 命令，能在这个控制命令处理中设置 filp->f_owner 为对应进程 ID。不过此项工作已由内核完成，设备驱动无须处理。
- 支持 F_SETFL 命令的处理，每当 FASYNC 标志改变时，驱动程序中的 fasync()函数将得以执行。因此，驱动中应该实现 fasync()函数。
- 在设备资源可获得时，调用 kill_fasync()函数激发相应的信号。

驱动中的上述 3 项工作和应用程序中的 3 项工作是一一对应的，图 9.2 所示为异步通知处理过程中用户空间和设备驱动的交互。

图 9.2　异步通知中设备驱动和异步通知的交互

设备驱动中异步通知编程比较简单，主要用到一项数据结构和两个函数。数据结构是 fasync_struct 结构体，两个函数分别如下。

处理 FASYNC 标志变更的函数。

```
int fasync_helper(int fd, struct file *filp, int mode, struct fasync_struct **fa);
```

释放信号用的函数。

```
void kill_fasync(struct fasync_struct **fa, int sig, int band);
```

和其他的设备驱动一样，将 fasync_struct 结构体指针放在设备结构体中仍然是最佳选择，代码清单 9.3 给出了支持异步通知的设备结构体模板。

代码清单 9.3　支持异步通知的设备结构体模板

```
struct xxx_dev
{
    struct cdev cdev; /*cdev 结构体*/
    ...
    struct fasync_struct *async_queue; /* 异步结构体指针 */
};
```

在设备驱动的 fasync()函数中，只需要简单地将该函数的 3 个参数以及

fasync_struct 结构体指针的指针作为第 4 个参数传入 fasync_helper()函数即可。代码清单 9.4 给出了支持异步通知的设备驱动程序 fasync()函数的模板。

华清远见

代码清单 9.4　支持异步通知的设备驱动 fasync()函数的模板

```
static int xxx_fasync(int fd, struct file *filp, int mode)
{
    struct xxx_dev *dev = filp->private_data;
    return fasync_helper(fd, filp, mode, &dev->async_queue);
}
```

在设备资源可以获得时，应该调用 kill_fasync()释放 SIGIO 信号，可读时第 3 个参数设置为 POLL_IN，可写时第 3 个参数设置为 POLL_OUT。代码清单 9.5 为释放信号的范例。

代码清单 9.5　支持异步通知的设备驱动信号释放的模板

```
 static ssize_t xxx_write(struct file *filp, const char __user *buf, size_t count,
                         loff_t *f_pos)
 {
  struct xxx_dev *dev = filp->private_data;
  ...
  /* 产生异步读信号 */
  if (dev->async_queue)
      kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
  ...
}
```

最后，在文件关闭时，即在设备驱动的 release()函数中，应调用设备驱动的 fasync()函数将文件从异步通知的列表中删除。代码清单 9.6 给出了支持异步通知的设备驱动 release()函数的模板。

代码清单 9.6　支持异步通知的设备驱动 release()函数的模板

```
static int xxx_release(struct inode *inode, struct file *filp)
{
    struct xxx_dev *dev = filp->private_data;
    /* 将文件从异步通知列表中删除 */
    xxx_fasync(-1, filp, 0);
    ...
    return 0;
}
```

9.3 支持异步通知的 globalfifo 驱动

9.3.1 在 globalfifo 驱动中增加异步通知

首先，参照代码清单 9.2 将异步结构体指针添加到 globalfifo_dev 设备结构体内，如代码清单 9.7 所示。

代码清单 9.7 增加异步通知后的 globalfifo 设备结构体

```
struct globalfifo_dev
{
    struct cdev cdev; /*cdev 结构体*/
    unsigned int current_len;   /*fifo 有效数据长度*/
    unsigned char mem[GLOBALFIFO_SIZE]; /*全局内存*/
    struct semaphore sem; /*并发控制用的信号量*/
    wait_queue_head_t r_wait; /*阻塞读用的等待队列头*/
    wait_queue_head_t w_wait; /*阻塞写用的等待队列头*/
    struct fasync_struct *async_queue; /* 异步结构体指针，用于读 */
};
```

参考代码清单 9.3 的 fasync()函数模板，globalfifo 的这个函数如代码清单 9.8 所示。

代码清单 9.8 支持异步通知的 globalfifo 设备驱动的 fasync()函数

```
static int globalfifo_fasync(int fd, struct file *filp, int mode)
{
   struct globalfifo_dev *dev = filp->private_data;
   return fasync_helper(fd, filp, mode, &dev->async_queue);
}
```

在 globalfifo 设备被正确写入之后，它变得可读，这个时候驱动应释放 SIGIO 信号以便应用程序捕获，代码清单 9.9 给出了支持异步通知的 globalfifo 设备驱动的写函数。

代码清单 9.9 支持异步通知的 globalfifo 设备驱动的写函数

```
static ssize_t globalfifo_write(struct file *filp, const char __user *buf,
  size_t count, loff_t *ppos)
{
  struct globalfifo_dev *dev = filp->private_data; //获得设备结构体指针
  int ret;
  DECLARE_WAITQUEUE(wait, current); //定义等待队列

  down(&dev->sem); //获取信号量
  add_wait_queue(&dev->w_wait, &wait); //进入写等待队列头

  /* 等待 FIFO 非满 */
```

```
if (dev->current_len == GLOBALFIFO_SIZE)
{
  if (filp->f_flags &O_NONBLOCK)
  //如果是非阻塞访问
  {
    ret =  - EAGAIN;
    goto out;
  }
  __set_current_state(TASK_INTERRUPTIBLE); //改变进程状态为睡眠
  up(&dev->sem);

  schedule(); //调度其他进程执行
  if (signal_pending(current))
  //如果是因为信号唤醒
  {
    ret =  - ERESTARTSYS;
    goto out2;
  }

  down(&dev->sem); //获得信号量
}

/*从用户空间复制到内核空间*/
if (count > GLOBALFIFO_SIZE - dev->current_len)
  count = GLOBALFIFO_SIZE - dev->current_len;

if (copy_from_user(dev->mem + dev->current_len, buf, count))
{
  ret =  - EFAULT;
  goto out;
}
else
{
  dev->current_len += count;
  printk(KERN_INFO "written %d bytes(s),current_len:%d\n", count,
  dev->current_len);

  wake_up_interruptible(&dev->r_wait); //唤醒读等待队列
  /* 产生异步读信号 */
  if (dev->async_queue)
```

```
52        kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
53
54     ret = count;
55   }
56
57   out: up(&dev->sem); //释放信号量
58   out2:remove_wait_queue(&dev->w_wait, &wait); //从附属的等待队列头移除
59   set_current_state(TASK_RUNNING);
60   return ret;
61 }
```

参考代码清单 9.6，增加异步通知后的 globalfifo 设备驱动的 release()函数中需调用 globalfifo_ fasync()函数将文件从异步通知列表中删除，代码清单 9.10 给出了支持异步通知的 globalfifo_ release()函数。

代码清单 9.10　增加异步通知后的 globalfifo 设备驱动的 release()函数

```
1 int globalfifo_release(struct inode *inode, struct file *filp)
2 {
3     struct globalfifo_dev *dev = filp->private_data;
4     /* 将文件从异步通知列表中删除 */
5     globalfifo_fasync( - 1, filp, 0);
6     return 0;
7 }
```

9.3.2　在用户空间验证 globalfifo 的异步通知

现在，我们可以采用与代码清单 9.2 类似的方法，编写一个在用户空间验证 globalfifo 异步通知的程序，这个程序在接收到由 globalfifo 发出的信号后将输出信号值，如代码清单 9.11 所示。

代码清单 9.11　监控 globalfifo 异步通知信号的应用程序

```
1  #include ...
2
3  /*接收到异步读信号后的动作*/
4  void input_handler(int signum)
5  {
6    printf("receive a signal from globalfifo,signalnum:%d\n",signum);
7  }
8
9  main()
10 {
11   int fd, oflags;
12   fd = open("/dev/globalfifo", O_RDWR, S_IRUSR | S_IWUSR);
```

```
  if (fd !=  - 1)
  {
    //启动信号驱动机制
    signal(SIGIO, input_handler); //让input_handler()处理SIGIO信号
    fcntl(fd, F_SETOWN, getpid());
    oflags = fcntl(fd, F_GETFL);
    fcntl(fd, F_SETFL, oflags | FASYNC);
    while(1)
    {
      sleep(100);
    }
  }
  else
  {
    printf("device open failure\n");
  }
}
```

加载新的 globalfifo 设备驱动并创建设备文件节点，运行上述程序后，每当通过 echo 向/dev/globalfifo 写入新的数据时，input_handler()将会被调用，如下所示：

```
[root@localhost driver_study]# echo 0 > /dev/globalfifo
[root@localhost driver_study]# receive a signal from globalfifo,signalnum:29

[root@localhost driver_study]# echo 0 > /dev/globalfifo
[root@localhost driver_study]# receive a signal from globalfifo,signalnum:29

[root@localhost driver_study]# echo 0 > /dev/globalfifo
[root@localhost driver_study]# receive a signal from globalfifo,signalnum:29
```

9.4 Linux 2.6 异步 I/O

9.4.1　AIO 概念与 GNU C 库函数

Linux 系统中最常用的输入/输出（I/O）模型是同步 I/O。在这个模型中，当请求发出之后，应用程序就会阻塞，直到请求满足为止。这是很好的一种解决方案，因为调用应用程序在等待 I/O 请求完成时不需要使用任何中央处理单元（CPU）。但是在某些情况下，I/O 请求可能需要与其他进程产生交叠。可移植操作系统接口（POSIX）异步 I/O（AIO）应用程序接口（API）就提供了这种功能。

Linux 异步 I/O 是 2.6 版本内核的一个标准特性，但是我们在 2.4 版本内核的补丁中也可以找到它。AIO 基本思想是允许进程发起很多 I/O 操作，而不用阻塞或等待任何操作完成。稍后或在接收到 I/O 操作完成的通知时，进程就可以检索 I/O 操作的结果。

select()函数所提供的功能（异步阻塞 I/O）与 AIO 类似，它对通知事件进行阻塞，而不是对 I/O 调用进行阻塞。

在异步非阻塞 I/O 中，我们可以同时发起多个传输操作。这需要每个传输操作都

有惟一的上下文，这样才能在它们完成时区分到底是哪个传输操作完成了。在 AIO 中，通过 aiocb（AIO I/O Control Block）结构体进行区分。这个结构体包含了有关传输的所有信息，包括为数据准备的用户缓冲区。在产生 I/O（称为完成）通知时，aiocb 结构就被用来惟一标识所完成的 I/O 操作。

AIO 系列 API 被 GNU C 库函数所包含，它被 POSIX.1b 所要求，主要包括如下函数。

1．aio_read

aio_read()函数请求对一个有效的文件描述符进行异步读操作。这个文件描述符可以表示一个文件、套接字甚至管道。aio_read 函数的原型如下：

```
int aio_read( struct aiocb *aiocbp );
```

aio_read()函数在请求进行排队之后会立即返回。如果执行成功，返回值就为 0；如果出现错误，返回值就为−1，并设置 errno 的值。

2．aio_write

aio_write()函数用来请求一个异步写操作，其函数原型如下：

```
int aio_write( struct aiocb *aiocbp );
```

aio_write()函数会立即返回，说明请求已经进行排队（成功时返回值为 0，失败时返回值为−1，并相应地设置 errno。

3．aio_error

aio_error 函数被用来确定请求的状态，其原型如下：

```
int aio_error( struct aiocb *aiocbp );
```

这个函数可以返回以下内容。

EINPROGRESS：说明请求尚未完成。

ECANCELLED：说明请求被应用程序取消了。

-1：说明发生了错误，具体错误原因由 errno 记录。

4．aio_return

异步 I/O 和标准块 I/O 之间的另外一个区别是不能立即访问这个函数的返回状态，因为并没有阻塞在 read()调用上。在标准的 read()调用中，返回状态是在该函数返回时提供的。但是在异步 I/O 中，我们要使用 aio_return()函数。这个函数的原型如下：

```
ssize_t aio_return( struct aiocb *aiocbp );
```

只有在 aio_error()调用确定请求已经完成（可能成功，也可能发生了错误）之后，才会调用这个函数。aio_return()的返回值就等价于同步情况中 read 或 write 系统调用的返回值（所传输的字节数，如果发生错误，返回值就为−1）。

代码清单 9.12 给出了用户空间应用程序进行异步读操作的一个例程，它首先打开文件，然后准备 aiocb 结构体，之后调用 aio_read（&my_aiocb）进行提出异步读请求，当 aio_error(&my_aiocb) == EINPROGRESS 即操作还在进行中时，一直等待，结束后

通过 aio_return（&my_aiocb）获得返回值。

代码清单 9.12　用户空间异步读例程

```
1  #include <aio.h>
2  ...
3  int fd, ret;
4  struct aiocb my_aiocb;
5
6  fd = open("file.txt", O_RDONLY);
7  if (fd < 0)
8    perror("open");
9
10 /* 清零 aiocb 结构体 */
11 bzero((char*) &my_aiocb, sizeof(struct aiocb));
12
13 /* 为 aiocb 请求分配数据缓冲区 */
14 my_aiocb.aio_buf = malloc(BUFSIZE + 1);
15 if (!my_aiocb.aio_buf)
16   perror("malloc");
17
18 /* 初始化 aiocb 的成员 */
19 my_aiocb.aio_fildes = fd;
20 my_aiocb.aio_nbytes = BUFSIZE;
21 my_aiocb.aio_offset = 0;
22
23 ret = aio_read(&my_aiocb);
24 if (ret < 0)
25   perror("aio_read");
26
27 while (aio_error(&my_aiocb) == EINPROGRESS)
28   ;
29
30 if ((ret = aio_return(&my_iocb)) > 0)
31 {
32   /* 获得异步读的返回值 */
33 }
34 else
35 {
36   /* 读失败，分析 errorno */
37 }
```

用户可以使用 aio_suspend()函数来挂起（或阻塞）调用进程，直到异步请求完成为止，此时会产生一个信号，或者发生其他超时操作。调用者提供了一个 aiocb 引用列表，其中任何一个完成都会导致 aio_suspend()返回。aio_suspend 的函数原型如下：

```
int aio_suspend( const struct aiocb *const cblist[],
                                int n, const struct timespec *timeout );
```

代码清单 9.13 给出了用户空间异步读操作时使用 aio_suspend()函数的例子。

代码清单 9.13　用户空间异步 I/O aio_suspend()函数使用例程

```
1 struct aioct *cblist[MAX_LIST]
2 /* 清零 aioct 结构体链表 */
3 bzero((char *)cblist, sizeof(cblist));
4 /* 将一个或更多的 aiocb 放入 aioct 结构体链表 */
```

```
cblist[0] = &my_aiocb;
ret = aio_read( &my_aiocb );
ret = aio_suspend( cblist, MAX_LIST, NULL );
```

aio_cancel()函数允许用户取消对某个文件描述符执行的一个或所有 I/O 请求。其原型如下：

```
int aio_cancel( int fd, struct aiocb *aiocbp );
```

如果要取消一个请求，用户需提供文件描述符和 aiocb 引用。如果这个请求被成功取消了，那么这个函数就会返回 AIO_CANCELED。如果请求完成了，这个函数就会返回 AIO_NOTCANCELED。

如果要取消对某个给定文件描述符的所有请求，用户需要提供这个文件的描述符以及一个对 aiocbp 的 NULL 引用。如果所有的请求都取消了，这个函数就会返回 AIO_CANCELED；如果至少有一个请求没有被取消，那么这个函数就会返回 AIO_NOT_CANCELED；如果没有一个请求可以被取消，那么这个函数就会返回 AIO_ALLDONE。然后，可以使用 aio_error()来验证每个 AIO 请求，如果某请求已经被取消了，那么 aio_error()就会返回−1，并且 errno 会被设置为 ECANCELED。

lio_listio()函数可用于同时发起多个传输。这个函数非常重要，它使得用户可以在一个系统调用（一次内核上下文切换）中启动大量的 I/O 操作。lio_listio API 函数的原型如下：

```
int lio_listio( int mode, struct aiocb *list[], int nent, struct sigevent
*sig );
```

mode 参数可以是 LIO_WAIT 或 LIO_NOWAIT。LIO_WAIT 会阻塞这个调用，直到所有的 I/O 都完成为止。在操作进行排队之后，LIO_NOWAIT 就会返回。list 是一个 aiocb 引用的列表，最大元素的个数是由 nent 定义的。如果 list 的元素为 NULL，lio_listio()会将其忽略。

代码清单 9.14 给出了用户空间异步 I/O 操作时使用 lio_listio()函数的例子。

代码清单 9.14 用户空间异步 I/O lio_listio()函数使用例程

```
struct aiocb aiocb1, aiocb2;
struct aiocb *list[MAX_LIST];
...
/* 准备第一个aiocb */
aiocb1.aio_fildes = fd;
aiocb1.aio_buf = malloc( BUFSIZE+1 );
aiocb1.aio_nbytes = BUFSIZE;
aiocb1.aio_offset = next_offset;
aiocb1.aio_lio_opcode = LIO_READ; /*异步读操作*/
... /*准备多个aiocb */
bzero( (char *)list, sizeof(list) );

/*将aiocb填入链表*/
list[0] = &aiocb1;
```

```
15 list[1] = &aiocb2;
16 ...
17 ret = lio_listio( LIO_WAIT, list, MAX_LIST, NULL );/*发起大量 I/O 操作*/
```

上述代码第 9 行中，因为是进行异步读操作，所以操作码为 LIO_READ，对于写操作来说，应该使用 LIO_WRITE 作为操作码，而 LIO_NOP 意味着空操作。

9.4.2　使用信号作为 AIO 的通知

9.1～9.3 节讲述的信号作为异步通知的机制在 AIO 中仍然是适用的，为使用信号，使用 AIO 的应用程序同样需要定义信号处理程序，在指定的信号被产生时会触发调用这个处理程序。作为信号上下文的一部分，特定的 aiocb 请求被提供给信号处理函数用来区分 AIO 请求。

代码清单 9.15 给出了使用信号作为 AIO 异步 I/O 通知机制的例子。

代码清单 9.15　使用信号作为 AIO 异步 I/O 通知机制例程

```
1  /*设置异步 I/O 请求*/
2  void setup_io(...)
3  {
4    int fd;
5    struct sigaction sig_act;
6    struct aiocb my_aiocb;
7    ...
8    /* 设置信号处理函数 */
9    sigemptyset(&sig_act.sa_mask);
10   sig_act.sa_flags = SA_SIGINFO;
11   sig_act.sa_sigaction = aio_completion_handler;
12
13   /* 设置 AIO 请求 */
14   bzero((char*) &my_aiocb, sizeof(struct aiocb));
15   my_aiocb.aio_fildes = fd;
16   my_aiocb.aio_buf = malloc(BUF_SIZE + 1);
17   my_aiocb.aio_nbytes = BUF_SIZE;
18   my_aiocb.aio_offset = next_offset;
19
20   /* 连接 AIO 请求和信号处理函数 */
21   my_aiocb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
22   my_aiocb.aio_sigevent.sigev_signo = SIGIO;
23   my_aiocb.aio_sigevent.sigev_value.sival_ptr = &my_aiocb;
24
25   /* 将信号与信号处理函数绑定 */
26   ret = sigaction(SIGIO, &sig_act, NULL);
27   ...
28   ret = aio_read(&my_aiocb); /*发出异步读请求*/
29 }
30
31 /*信号处理函数*/
32  void  aio_completion_handler(int  signo,  siginfo_t  *info,  void
*context)
```

```
33 {
34   struct aiocb *req;
35
36   /* 确定是我们需要的信号*/
37   if (info->si_signo == SIGIO)
38   {
39     req = (struct aiocb*)info->si_value.sival_ptr; /*获得 aiocb*/
40
41     /* 请求的操作完成了吗? */
42     if (aio_error(req) == 0)
43     {
44       /* 请求的操作完成，获取返回值 */
45       ret = aio_return(req);
46     }
47   }
48   return ;
49 }
```

特别要注意上述代码的第 39 行通过(struct aiocb*)info->si_value.sival_ptr 获得了信号对应的 aiocb。

9.4.3 使用回调函数作为 AIO 的通知

除了信号之外，应用程序还可提供一个回调（Callback）函数给内核，以便 AIO 的请求完成后内核调用这个函数。

一般来说，下层对上层（如内核对应用）的调用都称为“回调”，而上层对下层（如进行 Linux 系统调用）的调用称为“调用”，如图 9.3 所示。

图 9.3 回调与调

代码清单 9.16 给出了使用回调函数作为 AIO 异步 I/O 请求完成的通知机制的例子。

华清远见

代码清单 9.16　使用回调函数作为 AIO 异步 I/O 通知机制例程

```
1  /*设置异步 I/O 请求*/
2  void setup_io(...)
3  {
4    int fd;
5    struct aiocb my_aiocb;
6    ...
7    /* 设置 AIO 请求 */
8    bzero((char*) &my_aiocb, sizeof(struct aiocb));
9    my_aiocb.aio_fildes = fd;
10   my_aiocb.aio_buf = malloc(BUF_SIZE + 1);
11   my_aiocb.aio_nbytes = BUF_SIZE;
12   my_aiocb.aio_offset = next_offset;
13
14   /* 连接 AIO 请求和线程回调函数 */
15   my_aiocb.aio_sigevent.sigev_notify = SIGEV_THREAD;
16   my_aiocb.aio_sigevent.notify_function = aio_completion_handler;
17     /*设置回调函数*/
18   my_aiocb.aio_sigevent.notify_attributes = NULL;
19   my_aiocb.aio_sigevent.sigev_value.sival_ptr = &my_aiocb;
20   ... ret = aio_read(&my_aiocb); //发起 AIO 请求
21 }
22
23 /* 异步 I/O 完成回调函数 */
24 void aio_completion_handler(sigval_t sigval)
25 {
26   struct aiocb *req;
27   req = (struct aiocb*)sigval.sival_ptr;
28
29   /* AIO 请求完成? */
30   if (aio_error(req) == 0)
31   {
32     /* 请求完成，获得返回值 */
33     ret = aio_return(req);
34   }
35
36   return ;
37 }
```

上述程序在创建 aiocb 请求之后，使用 SIGEV_THREAD 请求了一个线程回调函数来作为通知方法。在回调函数中，通过(struct aiocb*)sigval.sival_ptr 可以获得对应的

aiocb 指针，使用 AIO 函数可验证请求是否已经完成。

proc 文件系统包含了两个虚拟文件，它们可以用来对异步 I/O 的性能进行优化。

- /proc/sys/fs/aio-nr 文件提供了系统范围异步 I/O 请求的数目。
- /proc/sys/fs/aio-max-nr 文件是所允许的并发请求的最大个数，最大个数通常是 64KB，这对于大部分应用程序来说都已经足够了。

9.4.4 AIO 与设备驱动

在内核中，每个 I/O 请求都对应于一个 kiocb 结构体，其 ki_filp 成员指向对应的 file 指针，通过 is_sync_kiocb()可以判断某 kiocb 是否为同步 I/O 请求，如果返回非真，表示为异步 I/O 请求。

块设备和网络设备本身是异步的，只有字符设备必须明确表明应支持 AIO。AIO 对于大多数字符设备而言都不是必须的，只有极少数设备需要。比如，对于磁带机，由于 I/O 操作很慢，这时候使用异步 I/O 将改善性能。

字符设备驱动程序中，file_operations 包含 3 个与 AIO 相关的成员函数，如下所示：

```
ssize_t (*aio_read) (struct kiocb *iocb, char *buffer,
size_t count, loff_t offset);
ssize_t (*aio_write) (struct kiocb *iocb, const char *buffer,
size_t count, loff_t offset);
int (*aio_fsync) (struct kiocb *iocb, int datasync);
```

aio_read()和 aio_write()与 file_operations 中的 read()和 write()中的 offset 参数不同，它直接传递值，而后者传递的是指针，这是因为 AIO 从来不需要改变文件的位置。

aio_read()和 aio_write()函数本身不一定完成了读和写操作，它只是发起、初始化读和写操作，代码清单 9.17 给出了驱动程序中 aio_read()和 aio_write()函数的实现例子。

代码清单 9.17 设备驱动中的异步 I/O 函数

```
1  /*异步读*/
2  static ssize_t xxx_aio_read(struct kiocb *iocb, char *buf, size_t
count, loff_t
3    pos)
4  {
5    return xxx_defer_op(0, iocb, buf, count, pos);
6  }
7
8  /*异步写*/
9  static ssize_t xxx_aio_write(struct kiocb *iocb, const char *buf,
size_t count,
10   loff_t pos)
11 {
12   return xxx_defer_op(1, iocb, (char*)buf, count, pos);
```

```
13 }
14
15 /*初始化异步 I/O*/
16 static int xxx_defer_op(int write, struct kiocb *iocb, char *buf,
size_t count,
17   loff_t pos)
18 {
19   struct async_work *async_wk;
20   int result;
21   /* 当可以访问 buffer 时进行复制*/
22   if (write)
23     result = xxx_write(iocb->ki_filp, buf, count, &pos);
24   else
25     result = xxx_read(iocb->ki_filp, buf, count, &pos);
26   /* 如果是同步 IOCB，立即返回状态 */
27   if (is_sync_kiocb(iocb))
28     return result;
29
30   /* 否则，推后几μs 执行 */
31   async_wk = kmalloc(sizeof(*async_wk), GFP_KERNEL);
32   if (async_wk == NULL)
33     return result;
34   /*调度延迟的工作*/
35   async_wk->iocb = iocb;
36   async_wk->result = result;
37   INIT_WORK(&async_wk->work, xxx_do_deferred_op, async_wk);
38   schedule_delayed_work(&async_wk->work, HZ / 100);
39   return  - EIOCBQUEUED; /*控制权返回用户空间*/
40 }
41
42 /*延迟后执行*/
43 static void xxx_do_deferred_op(void *p)
44 {
45   struct async_work *async_wk = (struct async_work*)p;
46   aio_complete(async_wk->iocb, async_wk->result, 0);
47   kfree(async_wk);
48 }
```

上述代码中最核心的是使用 aync_work（异步工作）结构体将操作延后执行，aync_work 结构体定义如代码清单 9.18 所示，通过 schedule_delayed_work()函数可以调度其执行。第 46 行对 aio_complete()的调用用于通知内核驱动程序已经完成了操作。

代码清单 9.18 async_work 结构体

```
struct async_work
{
    struct kiocb *iocb;  //kiocb 结构体指针
    int result;  //执行结果
    struct work_struct work; //工作结构体
};
```

9.5 总结

本章主要讲解了 Linux 中的异步 I/O，异步 I/O 可以使得应用程序在等待 I/O 操作的同时进行其他操作。

使用信号可以实现设备驱动与用户程序之间的异步通知，总体而言，设备驱动和用户空间要分别完成以下工作：用户空间设置文件的拥有者、FASYNC 标志及捕获信号，内核空间响应对文件的拥有者、FASYNC 标志的设置，并在资源可获得时释放信号。

Linux 2.6 内核包含对 AIO 的支持为用户空间提供统一的异步 I/O 接口。在 AIO 中，信号和回调函数是实现内核空间对用户空间应用程序通知的两种机制。

华清远见

推荐课程：嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm
- 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程：华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：

http://www.farsight.com.cn/courses/TS-LinuxApp.htm

第 10 章 中断与时钟

本章主要讲解 Linux 设备驱动编程中的中断与定时器处理。由于中断服务程序的执行并不存在于进程上下文，因此，要求中断服务程序的时间尽可能地短。因此，Linux 在中断处理中引入了顶半部和底半部分离的机制。另外，内核中对时钟的处理也采用中断方式，而内核软件定时器最终依赖于时钟中断。

10.1 节讲解中断和定时器的概念及处理流程。

10.2 节讲解 Linux 中断处理程序的架构，顶半部、底半部之间的关系。

10.3 节讲解 Linux 中断编程的方法，涉及申请和释放中断，禁止和使能中断，以及中断底半部 tasklet、工作队列、软中断机制等。

10.4 节讲解多个设备共享同一个中断号时的中断处理过程。

10.5 节和 10.6 节分别讲解 Linux 设备驱动编程中定时器的编程以及内核延时的方法。

华清远见

10.1 中断与定时器

所谓中断是指 CPU 在执行程序的过程中，出现了某些突发事件时 CPU 必须暂停执行当前的程序，转去处理突发事件，处理完毕后 CPU 又返回原程序被中断的位置并继续执行。

根据中断的来源，中断可分为内部中断和外部中断，内部中断的中断源来自 CPU 内部（软件中断指令、溢出、除法错误等，例如，操作系统从用户态切换到内核态需借助 CPU 内部的软件中断），外部中断的中断源来自 CPU 外部，由外设提出请求。

根据是否可以屏蔽中断分为可屏蔽中断与不屏蔽中断（NMI），可屏蔽中断可以通过屏蔽字被屏蔽，屏蔽后，该中断不再得到响应，而不屏蔽中断不能被屏蔽。

根据中断入口跳转方法的不同，中断分为向量中断和非向量中断。采用向量中断的 CPU 通常为不同的中断分配不同的中断号，当检测到某中断号的中断到来后，就自动跳转到与该中断号对应的地址执行。不同中断号的中断有不同的入口地址。非向量中断的多个中断共享一个入口地址，进入该入口地址后再通过软件判断中断标志来识别具体是哪个中断。也就是说，向量中断由硬件提供中断服务程序入口地址，非向量中断由软件提供中断服务程序入口地址。

一个典型的非向量中断服务程序如代码清单 10.1 所示，它先判断中断源，然后调用不同中断源的中断服务程序。

代码清单 10.1　非向量中断服务程序的典型结构

```
irq_handler()
{
  ...
  int int_src = read_int_status(); /*读硬件的中断相关寄存器*/
  switch (int_src) /*判断中断源*/
  {
    case DEV_A:
      dev_a_handler();
      break;
    case DEV_B:
      dev_b_handler();
      break;
    ...
    default:
      break;
  }
```

```
...
}
```

由于向量中断使用方便，目前有许多工程师创造了一些在非向量中断的处理器上模拟向量中断的方法。

嵌入式系统以及 X86 PC 中大多包含可编程中断控制器（PIC），许多 MCU 内部就集成了 PIC。如在 80386 中，PIC 是两片 i8259A 芯片的级联。通过读写 PIC 的寄存器，程序员可以屏蔽/使能某中断及获得中断状态，前者一般通过中断 MASK 寄存器完成，后者一般通过中断 PEND 寄存器完成。

定时器在硬件上也依赖中断来实现，图 10.1 给出了典型的嵌入式微处理内可编程间隔定时器（PIT）的工作原理，它接收一个时钟输入，当时钟脉冲到来时，将目前计数值增 1 并与预先设置的计数值（计数目标）比较，若相等，证明计数周期满，产生定时器中断并复位目前计数值。

图 10.1　定时器工作原理

10.2 Linux 中断处理程序架构

设备的中断会打断内核中进程的正常调度和运行，系统对更高吞吐率的追求势必要求中断服务程序尽可能地短小精悍。但是，这个良好的愿望往往与现实并不吻合。在大多数真实的系统中，当中断到来时，要完成的工作往往并不会是短小的，它可能要进行较大量的耗时处理。

图 10.2 描述了 Linux 内核的中断处理机制。为了在中断执行时间尽可能短和中断处理需完成大量工作之间找到一个平衡点，Linux 将中断处理程序分解为两个半部：顶半部（top half）和底半部（bottom half）。

图 10.2　Linux 中断处理机制

顶半部完成尽可能少的比较紧急的功能，它往往只是简单地读取寄存器中的中断状态并清除中断标志后就进行“登记中断”的工作。“登记中断”意味着将底半部处理程序挂到该设备的底半部执行队列中去。这样，顶半部执行的速度就会很快，可以服务更多的中断请求。

现在，中断处理工作的重心就落在了底半部的头上，它来完成中断事件的绝大多数任务。底半部几乎做了中断处理程序所有的事

情，而且可以被新的中断打断，这也是底半部和顶半部的最大不同，因为顶半部往往被设计成不可中断。底半部则相对来说并不是非常紧急的，而且相对比较耗时，不在硬件中断服务程序中执行。

尽管顶半部、底半部的结合能够改善系统的响应能力，但是，僵化地认为 Linux 设备驱动中的中断处理一定要分两个半部则是不对的。如果中断要处理的工作本身很少，则完全可以直接在顶半部全部完成。

其他操作系统中对中断的处理也采用了类似于 Linux 系统的方法，真正的硬件中断服务程序都应该尽可能短。因此，许多操作系统都提供了中断上下文和非中断上下文相结合的机制，将中断的耗时工作保留到非中断上下文去执行。例如，在 VxWorks 系统中，网络设备包接收中断到来后，中断服务程序会通过 netJobAdd()函数将耗时的包接收和上传工作交给 tNetTask 任务去执行。

在 Linux 系统中，查看/proc/interrupts 文件可以获得系统中断的统计信息，如下所示。在单处理器的系统中，第一列是中断号，第二列是向 CPU0 产生该中断的次数，之后的是对于中断的描述。

```
              CPU0
   0:       135253          XT-PIC          timer
   1:           22          XT-PIC          i8042
   2:            0          XT-PIC          cascade
   8:            1          XT-PIC          rtc
   10:         108          XT-PIC          etho
   11:        3707          XT-PIC          BusLogic BT-958
   12:         313          XT-PIC          i8042
   15:           4          XT-PIC          idel
NMI:             0
ERR              0
```

10.3 Linux 中断编程

10.3.1 申请和释放中断

在 Linux 设备驱动中，使用中断的设备需要申请和释放对应的中断，分别使用内核提供的 request_irq()和 free_irq()函数。

1. 申请 IRQ

```
int request_irq(unsigned int irq,
                void (*handler)(int irq, void *dev_id, struct pt_regs
*regs),
                unsigned long irqflags,
                const char * devname,
                void *dev_id);
```

irq 是要申请的硬件中断号。

handler 是向系统登记的中断处理函数，是一个回调函数，中断发生时，系统调用这个函数，dev_id 参数将被传递给它。

irqflags 是中断处理的属性，若设置了 SA_INTERRUPT，则表示中断处理程序是快速处理程序，快速处理程序被调用时屏蔽所有中断，慢速处理程序不屏蔽；若设置了 SA_SHIRQ，则表示多个设备共享中断，dev_id 在中断共享时会用到，一般设置为这个设备的设备结构体或者 NULL。

request_irq()返回 0 表示成功，返回-INVAL 表示中断号无效或处理函数指针为 NULL，返回-EBUSY 表示中断已经被占用且不能共享。

2．释放 IRQ

与 request_irq()向对应的函数为 free_irq()，free_irq()的原型如下：

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

free_irq()中参数的定义与 request_irq()相同。

华清远见

10.3.2 使能和屏蔽中断

下列 3 个函数用于屏蔽一个中断源。

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

disable_irq_nosync()与 disable_irq()的区别在于前者立即返回，而后者等待目前的中断处理完成。注意，这 3 个函数作用于可编程中断控制器，因此，对系统内的所有 CPU 都生效。

下列两个函数将屏蔽本 CPU 内的所有中断。

```
void local_irq_save(unsigned long flags);
void local_irq_disable(void);
```

前者会将目前的中断状态保留在 flags 中，注意 flags 被直接传递，而不是通过指针传递。后者直接禁止中断。

与上述两个禁止中断对应的恢复中断的方法如下：

```
void local_irq_restore(unsigned long flags);
void local_irq_enable(void);
```

以上各 local_开头的方法的作用范围是本 CPU 内。

10.3.3 底半部机制

Linux 系统实现底半部的机制主要有 tasklet、工作队列和软中断。

1. tasklet

tasklet 的使用较简单，我们只需要定义 tasklet 及其处理函数并将两者关联，例如：

```
void my_tasklet_func(unsigned long); /*定义一个处理函数*/
DECLARE_TASKLET(my_tasklet, my_tasklet_func, data);
  /*定义一个 tasklet 结构 my_tasklet，与 my_tasklet_func(data)函数相关联*/
```

代码 DECLARE_TASKLET（my_tasklet,my_tasklet_func,data）实现了定义名称为 my_tasklet 的 tasklet 并将其与 my_tasklet_func()这个函数绑定，而传入这个函数的参数为 data。

在需要调度 tasklet 的时候引用一个 tasklet_schedule()函数就能使系统在适当的时候进行调度运行，如下所示：

```
tasklet_schedule(&my_tasklet);
```

使用 tasklet 作为底半部处理中断的设备驱动程序模板如代码清单 10.2 所示（仅包含与中断相关的部分）。

代码清单 10.2 tasklet 使用模板

```
1 /*定义 tasklet 和底半部函数并关联*/
2 void xxx_do_tasklet(unsigned long);
3 DECLARE_TASKLET(xxx_tasklet, xxx_do_tasklet, 0);
4
```

```
5  /*中断处理底半部*/
6  void xxx_do_tasklet(unsigned long)
7  {
8   ...
9  }
10
11 /*中断处理顶半部*/
12 irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
13 {
14  ...
15  tasklet_schedule(&xxx_tasklet);
16  ...
17 }
18
19 /*设备驱动模块加载函数*/
20 int __init xxx_init(void)
21 {
22  ...
23  /*申请中断*/
24  result = request_irq(xxx_irq, xxx_interrupt,
25      SA_INTERRUPT, "xxx", NULL);
26  ...
27 }
28
29 /*设备驱动模块卸载函数*/
30 void __exit xxx_exit(void)
31 {
32  ...
33  /*释放中断*/
34  free_irq(xxx_irq, xxx_interrupt);
35  ...
36 }
```

上述程序在模块加载函数中申请中断（第 24～25 行），并在模块卸载函数中释放它（第 34 行）。对应于 xxx_irq 的中断处理程序被设置为 xxx_interrupt()函数，在这个函数中，第 15 行的 tasklet_schedule（&xxx_tasklet）调度的 tasklet 函数 xxx_do_tasklet()在适当的时候得到 执行。

上述代码第 12 行显示中断处理程序顶半部的返回类型为 irqreturn_t，它定义为 int，中断处理程序顶半部一般返回 IRQ_HANDLED。

2．工作队列

工作队列的使用方法和 tasklet 非常相似，下面的代码用于定义一个工作队列和一个底半部执行函数。

```
struct work_struct my_wq; /*定义一个工作队列*/
void my_wq_func(unsigned long); /*定义一个处理函数*/
```

通过 INIT_WORK()可以初始化这个工作队列并将工作队列与处理函数绑定，如下所示：

```
INIT_WORK(&my_wq, (void (*)(void *)) my_wq_func, NULL);
  /*初始化工作队列并将其与处理函数绑定*/
```

与 tasklet_schedule()对应的用于调度工作队列执行的函数为 schedule_work()，如：

```
schedule_work(&my_wq);/*调度工作队列执行*/
```

与代码清单 10.2 对应的使用工作队列处理中断底半部的设备驱动程序模板如代码清单 10.3 所示（仅包含与中断相关的部分）。

代码清单 10.3　工作队列使用模板

```
1  /*定义工作队列和关联函数*/
2  struct work_struct xxx_wq;
3  void xxx_do_work(unsigned long);
4
5  /*中断处理底半部*/
6  void xxx_do_work(unsigned long)
7  {
8   ...
9  }
10
11 /*中断处理顶半部*/
12 irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
13 {
14  ...
15  schedule_work(&xxx_wq);
16  ...
17 }
18
19 /*设备驱动模块加载函数*/
20 int xxx_init(void)
21 {
22  ...
23  /*申请中断*/
24  result = request_irq(xxx_irq, xxx_interrupt,
25      SA_INTERRUPT, "xxx", NULL);
26  ...
27  /*初始化工作队列*/
28  INIT_WORK(&xxx_wq, (void (*)(void *)) xxx_do_work, NULL);
29  ...
30 }
31
32 /*设备驱动模块卸载函数*/
33 void xxx_exit(void)
34 {
35  ...
36  /*释放中断*/
37  free_irq(xxx_irq, xxx_interrupt);
38  ...
39 }
```

与代码清单 10.2 不同的是，上述程序在设计驱动模块加载函数中增加了初始化工作队列的代码（第 28 行）。

尽管 Linux 专家们多建议在设备第一次打开时才申请设备的中断并在最后一次关闭时释放中断以尽量减少中断被这个设备占用的时间，但是，大多数情况下，为求省事，大多数驱动工程师还是将中断申请和释放的工作放在了设备驱动的模块加载和卸载函数中。

3．软中断

软中断是用软件方式模拟硬件中断的概念，实现宏观上的异步执行效果，tasklet 也是基于软中断实现的。

第 9 章异步通知所基于的信号也类似于中断，现在，总结一下硬中断、软中断和信号的区别：硬中断是外部设备对 CPU 的中断，软中断通常是硬中断服务程序对内核的中断，而信号则是由内核（或其他进程）对某个进程的中断。

在 Linux 内核中，用 softirq_action 结构体表征一个软中断，这个结构体中包含软中断处理函数指针和传递给该函数的参数。使用 open_softirq()函数可以注册软中断对应的处理函数，而 raise_softirq()函数可以触发一个软中断。

软中断和 tasklet 仍然运行于中断上下文，而工作队列则运行于进程上下文。因此，软中断和 tasklet 处理函数中不能睡眠，而工作队列处理函数中允许睡眠。

local_bh_disable()和 local_bh_enable()是内核中用于禁止和使能软中断和 tasklet 底半部机制的函数。

10.3.4 实例：S3C2410 实时钟中断

S3C2410 处理器内部集成了实时钟（RTC）模块，该模块能够在系统断电的情况下由后备电池供电继续工作，其主要功能相对于一个时钟，记录年、月、日、时、分、秒等。S3C2410 的 RTC 可产生两种中断：周期节拍（tick）中断和报警（alarm）中断，前者相当于一个周期性的定时器，后者相当于一个“闹钟”，它在预先设定的时间到来时产生中断。

S3C2410 实时钟设备驱动的 open()函数中，会申请它将要使用的中断，如代码清单 10.4 所示。

华清远见

代码清单 10.4 S3C2410 实时钟驱动的 open()函数

```
/*S3C2410 实时钟驱动的open()函数 */
static int s3c2410_rtc_open(void)
{
  int ret;
  /*申请alarm中断*/
  ret = request_irq(s3c2410_rtc_alarmno, s3c2410_rtc_alarmirq,
                    SA_INTERRUPT, "s3c2410-rtc alarm", NULL);
  /*中断号被占用*/
  if (ret)
    printk(KERN_ERR "IRQ%d already in use\n", s3c2410_rtc_alarmno);

  /*申请tick中断*/
    ret = request_irq(s3c2410_rtc_tickno, s3c2410_rtc_tickirq, SA_INTERRUPT,
    "s3c2410-rtc tick", NULL);
  /*中断号被占用*/
  if (ret)
  {
    printk(KERN_ERR "IRQ%d already in use\n", s3c2410_rtc_tickno);
    goto tick_err;
  }

```

```
22  return ret;
23
24  tick_err: free_irq(s3c2410_rtc_alarmno, NULL); /*释放 alarm 中断*/
25  return ret;
26 }
```

S3C2410 实时钟设备驱动的 release()函数中，会释放它将要使用的中断，如代码清单 10.5 所示。

代码清单 10.5　S3C2410 实时钟驱动的 release()函数

```
1 static void s3c2410_rtc_release(void)
2 {
3     s3c2410_rtc_setpie(0);
4     /* 释放中断 */
5     free_irq(s3c2410_rtc_alarmno, NULL);
6     free_irq(s3c2410_rtc_tickno, NULL);
7 }
```

S3C2410 实时钟驱动的中断处理比较简单，不需要分为上下两个半部，而只存在顶半部，如代码清单 10.6 所示。

代码清单 10.6　S3C2410 实时钟驱动的中断处理程序

```
1  /*中断处理*/
2  static irqreturn_t s3c2410_rtc_alarmirq(int irq, void *id, struct
pt_regs *r)
3  {
4   rtc_update(1, RTC_AF | RTC_IRQF);
5   return IRQ_HANDLED;
6  }
7
8  static irqreturn_t s3c2410_rtc_tickirq(int irq, void *id, struct
pt_regs *r)
9  {
10  rtc_update(1, RTC_PF | RTC_IRQF);
11  return IRQ_HANDLED;
12 }
```

上述代码中调用的 rtc_update()函数定义于 linux-2.6.16\arch\arm\common\Rtctime.c 文件中，被各种实时钟共享，如代码清单 10.7 所示。

代码清单 10.7　实时钟更新的 rtc_update()函数

```
1  void rtc_update(unsigned long num, unsigned long events)
2  {
3   spin_lock(&rtc_lock);
4   rtc_irq_data = (rtc_irq_data + (num << 8)) | events;
5   spin_unlock(&rtc_lock);
6
7   wake_up_interruptible(&rtc_wait);/*唤醒等待队列*/
8   kill_fasync(&rtc_async_queue, SIGIO, POLL_IN);/*释放信号*/
9  }
10 EXPORT_SYMBOL(rtc_update);
```

上述中断处理程序并没有底半部（或者说没有严格意义上的 tasklet、工作队列或软中断底半部），实际上，它只是唤醒一个等待队列 rtc_wait，而这个等待队列的唤醒也将导致一个阻塞的进程被执行（这个阻塞的进程可看做底半部）。现在我们看到，等待队列可以作为中断处理程序顶半部和进程同步的一种良好机制。但是，任何情况下，都不能在顶半部等待一个等待队列，而只能唤醒。

10.4 中断共享

多个设备共享一根硬件中断线的情况在实际的硬件系统中广泛存在，PCI 设备即是如此。Linux 2.6 支持这种中断共享。下面是中断共享的使用方法。

- 共享中断的多个设备在申请中断时都应该使用 SA_SHIRQ 标志，而且一个设备以 SA_SHIRQ 申请某中断成功的前提是之前申请该中断的所有设备也都以 SA_SHIRQ 标志申请该中断。
- 尽管内核模块可访问的全局地址都可以作为 request_irq（…, void *dev_id）的最后一个参数 dev_id，但是设备结构体指针是可传入的最佳参数。
- 在中断到来时，所有共享此中断的中断处理程序都会被执行，在中断处理程序顶半部中，应迅速地根据硬件寄存器中的信息比照传入的 dev_id 参数判断是否是本设备的中断，若不是，应迅速返回，如图 10.3 所示。

图 10.3 共享中断的处理

代码清单 10.8 给出了使用共享中断的设备驱动程序的模板（仅包含与共享中断机制相关的部分）。

代码清单 10.8 共享中断编程的模板

```
/*中断处理顶半部*/
irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
 ...
 int status = read_int_status();/*获知中断源*/
 if(!is_myint(dev_id,status))/*判断是否是本设备中断*/
 {
 return IRQ_NONE;  /*立即返回*/
 }
 ...
 return IRQ_HANDLED;
}

/*设备驱动模块加载函数*/
int xxx_init(void)
{
 ...
```

华清远见

```
18  /*申请共享中断*/
19  result = request_irq(sh_irq, xxx_interrupt,
20      SA_SHIRQ, "xxx", xxx_dev);
21  ...
22 }
23
24 /*设备驱动模块卸载函数*/
25 void xxx_exit(void)
26 {
27  ...
28  /*释放中断*/
29  free_irq(xxx_irq, xxx_interrupt);
30  ...
31 }
```

10.5 内核定时器

10.5.1 内核定时器编程

软件意义上的定时器最终依赖硬件定时器来实现，内核在时钟中断发生后检测各定时器是否到期，到期后的定时器处理函数将作为软中断在底半部执行。实质上，时钟中断处理程序执行 update_process_timers()函数，该函数调用 run_local_timers()函数，这个函数处理 TIMER_SOFTIRQ 软中断，运行当前处理器上到期的所有定时器。

在 Linux 设备驱动编程中，可以利用 Linux 内核中提供的一组函数和数据结构来完成定时触发工作或者完成某周期性的事务。这组函数和数据结构使得驱动工程师多数情况下不用关心具体的软件定时器究竟对应着怎样的内核和硬件行为。

Linux 内核所提供的用于操作定时器的数据结构和函数如下。

1．timer_list

在 Linux 内核中，timer_list 结构体的一个实例对应一个定时器，如代码清单 10.9 所示。

代码清单 10.9　timer_list 结构体

```
1 struct timer_list {
2    struct list_head entry; //定时器列表
3    unsigned long expires; //定时器到期时间
4    void (*function)(unsigned long); //定时器处理函数
5    unsigned long data; //作为参数被传入定时器处理函数
6    struct timer_base_s *base;
7 };
```

当定时器期满后，其中第 5 行的 function()成员将被执行，而第 4 行的 data 成员则是传入其中的参数，第 3 行的 expires 则是定时器到期的时间（jiffies）。

如下代码定义一个名为 my_timer 的定时器：

```
struct timer_list my_timer;
```

2．初始化定时器

```
void init_timer(struct timer_list * timer);
```

上述 init_timer()函数初始化 timer_list 的 entry 的 next 为 NULL，并给 base 指针赋值。

TIMER_INITIALIZER（_function, _expires, _data）宏用于赋值定时器结构体的 function、expires、data 和 base 成员，这个宏的定义如下所示：

```
#define TIMER_INITIALIZER(_function, _expires, _data) {        \
        .function = (_function),                               \
        .expires = (_expires),                                 \
        .data = (_data),                                       \
        .base = &__init_timer_base,                            \
    }
```

DEFINE_TIMER（_name, _function, _expires, _data）宏是定义并初始化定时器成员的“快捷方式”，这个宏定义如下所示：

```
#define DEFINE_TIMER(_name, _function, _expires, _data)        \
    struct timer_list _name =                                  \
        TIMER_INITIALIZER(_function, _expires, _data)
```

此外，setup_timer()也可用于初始化定时器并赋值其成员，其源代码如下：

```
static inline void setup_timer(struct timer_list * timer,
                    void (*function)(unsigned long),
                    unsigned long data)
{
     timer->function = function;
     timer->data = data;
     init_timer(timer);
}
```

3．增加定时器

```
void add_timer(struct timer_list * timer);
```

上述函数用于注册内核定时器，将定时器加入到内核动态定时器链表中。

4．删除定时器

```
int del_timer(struct timer_list * timer);
```

上述函数用于删除定时器。

del_timer_sync()是 del_timer()的同步版，主要在多处理器系统中使用，如果编译内核时不支持 SMP，del_timer_sync()和 del_timer()等价。

5．修改定时器的 expire

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

上述函数用于修改定时器的到期时间，在新的被传入的 expires 到来后才会执行定

时器函数。

代码清单 10.10 给出了一个完整的内核定时器使用模板，大多数情况下，设备驱动都如这个模板那样使用定时器。

代码清单 10.10　内核定时器使用模板

```
/*xxx 设备结构体*/
struct xxx_dev
{
 struct cdev cdev;
 ...
 timer_list xxx_timer;/*设备要使用的定时器*/
};

/*xxx 驱动中的某函数*/
xxx_func1(...)
{
 struct xxx_dev *dev = filp->private_data;
 ...
 /*初始化定时器*/
 init_timer(&dev->xxx_timer);
 dev->xxx_timer.function = &xxx_do_timer;
 dev->xxx_timer.data = (unsigned long)dev;
                        /*设备结构体指针作为定时器处理函数参数*/
 dev->xxx_timer.expires = jiffies + delay;
 /*添加（注册）定时器*/
 add_timer(&dev->xxx_timer);
 ...
}

/*xxx 驱动中的某函数*/
xxx_func2(…)
{
 ...
 /*删除定时器*/
 del_timer (&dev->xxx_timer);
 ...
}

/*定时器处理函数*/
static void xxx_do_timer(unsigned long arg)
```

```
36 {
37  struct xxx_device *dev = (struct xxx_device *)(arg);
38  ...
39  /*调度定时器再执行*/
40  dev->xxx_timer.expires = jiffies + delay;
41  add_timer(&dev->xxx_timer);
42  ...
43 }
```

从代码清单第 19、40 行可以看出，定时器的到期时间往往是在目前 jiffies 的基础上添加一个时延，若为 Hz，则表示延迟 1s。

在定时器处理函数中，在做完相应的工作后，往往会延后 expires 并将定时器再次添加到内核定时器链表，以便定时器能再次被触发。

10.5.2 实例：秒字符设备

下面我们编写一个字符设备“second”（即“秒”）的驱动，它在被打开的时候初始化一个定时器并将其添加到内核定时器链表，每秒输出一次当前的 jiffies（为此，定时器处理函数中每次都要修改新的 expires），整个程序如代码清单 10.11。

代码清单 10.11 使用内核定时器的 second 字符设备驱动

```
1   #include ...
2
3   #define SECOND_MAJOR 252    /*预设的second的主设备号*/
4
5   static int second_major = SECOND_MAJOR;
6
7   /*second设备结构体*/
8   struct second_dev
9   {
10    struct cdev cdev; /*cdev结构体*/
11    atomic_t counter;/* 一共经历了多少秒？ */
12    struct timer_list s_timer; /*设备要使用的定时器*/
13  };
14
15  struct second_dev *second_devp; /*设备结构体指针*/
16
17  /*定时器处理函数*/
18  static void second_timer_handle(unsigned long arg)
19  {
20    mod_timer(&second_devp->s_timer,jiffies + HZ);
21    atomic_inc(&second_devp->counter);
22
```

```
  printk(KERN_NOTICE "current jiffies is %ld\n", jiffies);
}

/*文件打开函数*/
int second_open(struct inode *inode, struct file *filp)
{
  /*初始化定时器*/
  init_timer(&second_devp->s_timer);
  second_devp->s_timer.function = &second_timer_handle;
  second_devp->s_timer.expires = jiffies + HZ;

  add_timer(&second_devp->s_timer); /*添加（注册）定时器*/

  atomic_set(&second_devp->counter,0); //计数清零

  return 0;
}
/*文件释放函数*/
int second_release(struct inode *inode, struct file *filp)
{
  del_timer(&second_devp->s_timer);

  return 0;
}

/*globalfifo 读函数*/
static ssize_t second_read(struct file *filp, char __user *buf, size_t count,
  loff_t *ppos)
{
  int counter;

  counter = atomic_read(&second_devp->counter);
  if(put_user(counter, (int*)buf))
    return - EFAULT;
  else
    return sizeof(unsigned int);
}

/*文件操作结构体*/
```

```
62 static const struct file_operations second_fops =
63 {
64   .owner = THIS_MODULE,
65   .open = second_open,
66   .release = second_release,
67   .read = second_read,
68 };
69
70 /*初始化并注册cdev*/
71 static void second_setup_cdev(struct second_dev *dev, int index)
72 {
73   int err, devno = MKDEV(second_major, index);
74
75   cdev_init(&dev->cdev, &second_fops);
76   dev->cdev.owner = THIS_MODULE;
77   dev->cdev.ops = &second_fops;
78   err = cdev_add(&dev->cdev, devno, 1);
79   if (err)
80     printk(KERN_NOTICE "Error %d adding LED%d", err, index);
81 }
82
83 /*设备驱动模块加载函数*/
84 int second_init(void)
85 {
86   int ret;
87   dev_t devno = MKDEV(second_major, 0);
88
89   /* 申请设备号*/
90   if (second_major)
91     ret = register_chrdev_region(devno, 1, "second");
92   else  /* 动态申请设备号 */
93   {
94     ret = alloc_chrdev_region(&devno, 0, 1, "second");
95     second_major = MAJOR(devno);
96   }
97   if (ret < 0)
98     return ret;
99   /* 动态申请设备结构体的内存*/
100  second_devp = kmalloc(sizeof(struct second_dev), GFP_KERNEL);
101  if (!second_devp)   /*申请失败*/
```

```
  {
    ret =  - ENOMEM;
    goto fail_malloc;
  }

  memset(second_devp, 0, sizeof(struct second_dev));

  second_setup_cdev(second_devp, 0);

  return 0;

  fail_malloc: unregister_chrdev_region(devno, 1);
}

/*模块卸载函数*/
void second_exit(void)
{
  cdev_del(&second_devp->cdev);   /*注销 cdev*/
  kfree(second_devp);    /*释放设备结构体内存*/
  unregister_chrdev_region(MKDEV(second_major, 0), 1); /*释放设备号*/
}

MODULE_AUTHOR("Song Baohua");
MODULE_LICENSE("Dual BSD/GPL");

module_param(second_major, int, S_IRUGO);

module_init(second_init);
module_exit(second_exit);
```

在 second 的 open()函数中，将启动定时器，此后每 1s 会再次运行定时器处理函数，在 second 的 release()函数中，定时器被删除。

second_dev 结构体中的原子变量 counter 用于秒计数，每次在定时器处理函数中将被 atomic_inc()调用原子的增 1，second 的 read()函数会将这个值返回给用户空间。

编译驱动，加载该内核模块并创建“/dev/second”设备文件结点后，使用代码清单 10.12 的应用程序打开“/dev/second”。

代码清单 10.12 的应用程序 second_test 会不断地读取自打开“/dev/second”设备文件以来经历的秒数。

代码清单 10.12　second 设备用户空间测试程序

```
1  #include ...
2
3  main()
4  {
5    int fd;
6    int counter = 0;
7    int old_counter = 0;
8
9    /*打开/dev/second 设备文件*/
10   fd = open("/dev/second", O_RDONLY);
11   if (fd !=  - 1)
12   {
13     while (1)
14     {
15       read(fd,&counter, sizeof(unsigned int));//读目前经历的秒数
16       if(counter!=old_counter)
17       {
18           printf("seconds after open /dev/second :%d\n",counter);
19           old_counter = counter;
20       }
21     }
22   }
23   else
24   {
25     printf("Device open failure\n");
26   }
27 }
```

运行 second_test 后，内核将不断地输出目前的 jiffies 值，如下所示：

```
current jiffies is 17216
current jiffies is 17316
current jiffies is 17416
current jiffies is 17516
current jiffies is 17616
current jiffies is 17716
current jiffies is 17816
current jiffies is 17916
current jiffies is 18016
current jiffies is 18116
current jiffies is 18216
current jiffies is 18316
```

而应用程序将不断输出自打开/dec/second 以来经历的秒数，如下所示：

```
[root@localhost driver_study]# ./second_test
seconds after open /dev/second :1
seconds after open /dev/second :2
seconds after open /dev/second :3
seconds after open /dev/second :4
seconds after open /dev/second :5
seconds after open /dev/second :6
seconds after open /dev/second :7
seconds after open /dev/second :8
seconds after open /dev/second :9
```

```
seconds after open /dev/second :10
```

10.6 内核延时

10.6.1 短延迟

Linux 内核中提供了如下 3 个函数分别进行纳秒、微秒和毫秒延迟。

```
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

上述延迟的实现原理本质上是忙等待，它根据 CPU 频率进行一定次数的循环。有时候，可以在软件中进行这样的延迟：

```
void delay(unsigned int time)
{
    while (time--);
}
```

ndelay()、udelay()和 mdelay()函数的实现方式机理与此类似。

毫秒时延（以及更大的秒时延）已经比较大了，在内核中，最好不要直接使用 mdelay()函数，这将无谓地耗费 CPU 资源，对于毫秒级以上时延，内核提供了下述函数：

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
```

上述函数将使得调用它的进程睡眠参数指定的时间，msleep()、ssleep()不能被打断，而 msleep_interruptible()则可以被打断。

受系统 HZ 以及进程调度的影响，msleep()类似函数的精度是有限的。

10.6.2 长延迟

内核中进行延迟的一个很直观的方法是比较当前的 jiffies 和目标 jiffies（设置为当前 jiffies 加上时间间隔的 jiffies），直到未来的 jiffies 达到目标 jiffies。代码清单 10.13 给出了使用忙等待先延迟 100 个 jiffies 再延迟 2s 的实例。

代码清单 10.13 忙等待时延实例

```
1 /*延迟100个jiffies*/
2 unsigned long delay = jiffies + 100;
3 while (time_before(jiffies, delay));
4
5 /*再延迟2s*/
6 unsigned long delay = jiffies + 2*HZ;
7 while (time_before(jiffies, delay));
```

与 time_before()对应的还有一个 time_after()，它们在内核中定义为（实际上只是将传入的未来时间 jiffies 和被调用时的 jiffies 进行一个简单的比较）：

```
#define time_after(a,b)            \
    (typecheck(unsigned long, a) &&    \
      typecheck(unsigned long, b) &&   \
      ((long)(b) - (long)(a) < 0))
#define time_before(a,b)   time_after(b,a)
```

为了防止 time_before()和 time_after()的比较过程中编译器对 jiffies 的优化，内核将其定义为 volatile 变量，这将保证它每次都被重新读取。

10.6.3 睡着延迟

睡着延迟无疑是比忙等待更好的方式，随着延迟在等待的时间到来之间进程处于睡眠状态，CPU 资源被其他进程使用。schedule_timeout()可以使当前任务睡眠指定的 jiffies 之后重新被调度执行，msleep()和 msleep_interruptible()在本质上都是依靠包含了 schedule_timeout() 的 schedule_ timeout_uninterruptible() 和 schedule_timeout_interruptible()实现的，如代码清单 10.14 所示。

代码清单 10.14 schedule_timeout()的使用

```
1  void msleep(unsigned int msecs)
2  {
3    unsigned long timeout = msecs_to_jiffies(msecs) + 1;
4
5    while (timeout)
6      timeout = schedule_timeout_uninterruptible(timeout);
7  }
8
9  unsigned long msleep_interruptible(unsigned int msecs)
10 {
11   unsigned long timeout = msecs_to_jiffies(msecs) + 1;
12
13   while (timeout && !signal_pending(current))
14     timeout = schedule_timeout_interruptible(timeout);
15   return jiffies_to_msecs(timeout);
16 }
```

实际上，schedule_timeout()的实现原理是向系统添加一个定时器，在定时器处理函数中唤醒参数对应的进程。

代码清单 10.14 第 6 行和第 14 行分别调用 schedule_timeout_uninterruptible()和 schedule_ timeout_interruptible()，这两个函数的区别在于前者在调用 schedule_timeout()之前置进程状态为 TASK_INTERRUPTIBLE，后者置进程状态为 TASK_UNINTERRUPTIBLE，如代码清单 10.15 所示。

代码清单 10.15 schedule_timeout_interruptible()和 schedule_timeout_interruptible()

```
signed long __sched schedule_timeout_interruptible(signed long timeout)
{
  __set_current_state(TASK_INTERRUPTIBLE);
  return schedule_timeout(timeout);
}

signed long __sched schedule_timeout_uninterruptible(signed long timeout)
{
  __set_current_state(TASK_UNINTERRUPTIBLE);
  return schedule_timeout(timeout);
}
```

另外，下面两个函数可以将当前进程添加到等待队列中，从而在等待队列上睡眠。当超时发生时，进程将被唤醒（后者可以在超时前被打断），如下所示：

```
sleep_on_timeout(wait_queue_head_t *q, unsigned long timeout);
interruptible_sleep_on_timeout(wait_queue_head_t*q, unsigned long timeout);
```

10.7 总结

Linux 的中断处理分为两个半部，顶半部处理紧急的硬件操作，底半部处理不紧急的耗时操作。tasklet 和工作队列都是调度中断底半部的良好机制，tasklet 基于软中断实现。内核定时器也依靠软中断实现。

内核中的延时是忙等待或者睡眠等待，为了充分利用 CPU 资源，使系统有更好的吞吐性能，在对延迟时间的要求并不是很精确的情况下，睡眠等待通常是值得推荐的。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

· 招生简章：http://www.embedu.org/courses/index.htm

· 课程内容：http://www.embedu.org/courses/course1.htm

嵌入式学院 华清远见旗下品牌

第 11 章　内存与 I/O 访问

由于 Linux 系统中提供了复杂的内存管理功能，所以内存的概念在 Linux 系统中变得相对复杂，出现了常规内存、高端内存、虚拟地址、逻辑地址、总线地址、物理地址、I/O 内存、设备内存、预留内存等概念。本章将系统地讲解内存和 I/O 的访问编程，带您走出内存和 I/O 的概念迷宫。

11.1 节讲解内存和 I/O 的硬件机制，主要涉及内存空间、I/O 空间和 MMU。

11.2 节讲解 Linux 的内存管理、内存区域的分布、常规内存与高端内存的区别。

11.3 节讲解 Linux 内存存取的方法，主要涉及内存动态申请以及通过虚拟地址存取物理地址的方法。

11.4 节讲解设备 I/O 内存和 I/O 端口的访问流程，这一节对于编写设备驱动意义非常重大，设备驱动使用此节的方法访问物理设备。

11.5 节讲解设备驱动中的 DMA 与 CACHE 一致性问题以及 DMA 编程方法。

华清远见

11.1 CPU 与内存和 I/O

11.1.1 内存空间与 I/O 空间

在 X86 处理器中存在着 I/O 空间的概念，I/O 空间是相对于内存空间而言的，它通过特定的指令 in、out 来访问。端口号标识了外设的寄存器地址。Intel 语法的 in、out 指令格式如下：

```
IN 累加器, {端口号|DX}
OUT {端口号|DX},累加器
```

目前，大多数嵌入式微控制器如 ARM、PowerPC 等中并不提供 I/O 空间，而仅存在内存空间。内存空间可以直接通过地址、指针来访问，程序和程序运行中使用的变量和其他数据都存在于内存空间中。

内存地址可以直接由 C 语言指针操作，例如在 186 处理器中执行如下代码：

```
unsigned char *p = (unsigned char *)0xF000FF00;
*p=11;
```

以上程序的意义为在绝对地址 0xF0000+0xFF00（186 使用 16 位段地址和 16 位偏移地址）写入 11。

而在 ARM、PowerPC 等未采用段地址的处理器中，p 指向的内存空间就是 0xF000FF00，而*p = 11 就是在该地址写入 11。

再如，186 处理器启动后会在绝对地址 0xFFFF0（对应 C 语言指针是 0xF000FFF0，0xF000 为段地址，0xFFF0 为段内偏移）执行，请看下面的代码：

```
typedef void (*lpFunction) ( ); /* 定义一个无参数、无返回类型的函数指针类型 */
lpFunction lpReset = (lpFunction)0xF000FFF0; /* 定义一个函数指针，指向*/
/* CPU 启动后所执行第一条指令的位置 */
lpReset(); /* 调用函数 */
```

在以上程序中，没有定义任何一个函数实体，但是程序中却执行了这样的函数调用：lpReset()，它实际上起到了“软重启”的作用，跳转到 CPU 启动后第一条要执行的指令的位置。因此，可以通过函数指针调用一个没有函数体的“函数”，本质上只是换一个地址开始执行。

即便是在 X86 处理器中，虽然提供了 I/O 空间，如果由我们自己设计电路板，外设仍然可以只挂接在内存空间。此时，CPU 可以像访问一个内存单元那样访问外设 I/O 端口，而不需要设立专门的 I/O 指令。因此，内存空间是必须的，而 I/O 空间是可选的。图 11.1 给出了内存空间和 I/O 空间的对比。

图 11.1　内存空间和 I/O 空间

11.1.2　内存管理单元 MMU

高性能处理器一般会提供一个内存管理单元（MMU），该单元辅助操作系统进行内存管理，提供虚拟地址和物理地址的映射、内存访问权限保护和 Cache 缓存控制等硬件支持。操作系统内核借助 MMU，可以让用户感觉到好像程序可以使用非常大的内存空间，从而使得编程人员在写程序时不用考虑计算机中的物理内存的实际容量。

为了理解基本的 MMU 操作原理，需先明晰几个概念。

- TLB：Translation Lookaside Buffer，即转换旁路缓存，TLB 是 MMU 的核心部件，它缓存少量的虚拟地址与物理地址的转换关系，是转换表的 Cache，因此也经常被称为“快表”。
- TTW：Translation Table walk，即转换表漫游，当 TLB 中没有缓冲对应的地址转换关系时，需要通过对内存中转换表（大多数处理器的转换表为多级页表，如图 11.2 所示）的访问来获得虚拟地址和物理地址的对应关系。TTW 成功后，结果应写入 TLB。

图 11.2　内存中的转换表

图 11.3 给出了一个典型的 ARM 处理器访问内存的过程，其他处理器也执行类似过程。当 ARM 要访问存储器时，MMU 先查找 TLB 中的虚拟地址表。如果 ARM 的结构支持分开的数据 TLB（DTLB）和指令 TLB（ITLB），则除取指令使用 ITLB 外，

其他的都使用 DTLB。ARM 处理器的 MMU 如图 11.3 所示。

图 11.3　ARM 的内存管理单元

若 TLB 中没有虚拟地址的入口，则转换表遍历硬件从存放于主存储器中的转换表中获取地址转换信息和访问权限（即执行 TTW），同时将这些信息放入 TLB，它或者被放在一个没有使用的入口或者替换一个已经存在的入口。之后，在 TLB 条目中控制信息的控制下，当访问权限允许时，对真实物理地址的访问将在 Cache 或者在内存中发生，如图 11.4 所示。

图 11.4　ARM CPU 进行数据访问的流程

ARM 中的 TLB 条目中的控制信息用于控制对对应地址的访问权限以及 Cache 的操作。

- C（高速缓存）和 B（缓冲）位被用来控制对应地址的高速缓存和写缓冲，并决定是否高速缓存。
- 访问权限和域位用来控制读写访问是否被允许。如果不允许，则 MMU 将向 ARM 处理器发送一个存储器异常，否则访问将被允许进行。

上述描述的 MMU 机制针对的虽然是 ARM 处理器，但 PowerPC、MIPS 等其他处理器也均有类似的操作。

MMU 具有虚拟地址和物理地址转换、内存访问权限保护等功能，这将使得 Linux 操作系统能单独为系统的每个用户进程分配独立的内存空间并保证用户空间不能访问内核空间的地址，为操作系统的虚拟内存管理模块提供硬件基础。

但是，MMU 并非对所有处理器都是必须的，例如常用的 SAMSUNG 基于 ARM7TDMI 系列的 S3C44B0X 不附带 MMU，其上无法运行老版的 Linux，而只能运行改版的µcLinux，但是新版的 Linux 2.6 则支持不带 MMU 的处理器。在嵌入式系统中，仍存在大量无 MMU 的处理器，Linux 2.6 为了更广泛地应用于嵌入式系统，融合了µcLinux，以支持 MMU-less 系统，如 Dragonball、ColdFire、Hitachi H8/300 等。

在 S3C2410 的 vivi 这个 Bootloader 中，建立了一个 4GB 物理地址与虚拟地址一一映射的一级页表，我们来追踪一下其创建过程。

vivi 的 main()函数首先会调用 mem_map_init()函数创建页表，其后调用 mmu_init()初始化并使能 MMU。mem_map_init()函数如代码清单 12.1 所示。

代码清单 12.1　vivi Bootloader 的 mem_map_init()函数

```
void mem_map_init(void)
{
  #ifdef CONFIG_S3C2410_NAND_BOOT
    /*CONFIG_S3C2410_NAND_BOOT = y，在文件 include/autoconf.h 中定义 */
    mem_map_nand_boot();
    /* 最终调用 mem_mepping_linear, 建立页表 */
  #else
    mem_map_nor();
  #endif
  cache_clean_invalidate(); /* 清空 cache,使无效 cache */
  tlb_invalidate(); /* 使无效快表 TLB */
}
```

在 mem_map_nand_boot()中会调用 mem_mapping_linear()进行页表创建工作，如代码清单 12.2 所示。

代码清单 12.2　vivi Bootloader 中的页表创建

```
static inline void mem_mapping_linear(void)
```

```
{
  unsigned long pageoffset, sectionNumber;
      putstr_hex("MMU table base address = 0x%", (unsigned long)mmu_tlb_base);

  /* 4GB 虚拟地址映射到相同的物理地址，均不能缓存 */
  for (sectionNumber = 0; sectionNumber < 4096; sectionNumber++)
  {
    pageoffset = (sectionNumber << 20);
    *(mmu_tlb_base + (pageoffset >> 20)) = pageoffset | MMU_SECDESC;
  }

  /* 使 DRAM 的区域可缓存 */
  /* SDRAM 物理地址 0x3000000-0x33ffffff,
  DRAM_BASE=0x30000000,DRAM_SIZE=64M
   */
  for (pageoffset = DRAM_BASE; pageoffset < (DRAM_BASE + DRAM_SIZE);
    pageoffset += SZ_1M)
  {
      //DPRINTK(3, "Make DRAM section cacheable: 0x%08lx\n", pageoffset);
    *(mmu_tlb_base + (pageoffset >> 20)) =
      pageoffset | MMU_SECDESC | MMU_CACHEABLE;
  }
}
```

内核为 ARM920T 的 S3C2410 是一个 32 位的 CPU，它的地址空间为 4GB。共有 4 种内存映射模式，即 Fault（无映射）、Coarse Page（粗页表）、Section（段）、Fine Page（细页表）。

代码清单 12.2 使用了 ARM920T 内存映射的 Section 模式（实际可等同于页大小为 1MB 的情况），4GB 的虚拟空间被分成一个一个称为 Section 的单位。4GB 的虚拟内存总共可以被分成 4096 个段（1MB*4096 = 4GB），因此我们必须用 4096 个描述符来对这组段进行描述，每个描述符占用 4 个字节，故这组描述符的大小为 16KB，这 4096 个描述符构成的表格就是转换表。对于 SDRAM 区域，在描述符中使能了 Cache。

mmu_init()函数直接调用 arm920_setup()初始化 MMU，arm920_setup()函数包含一系列内嵌的汇编代码，用于控制 S3C2410 的 CP15 协处理器对 MMU 的处理。

11.2 Linux 内存管理

对于包含 MMU 的处理器而言，Linux 系统提供了复杂的存储管理系统，使得进程所能访问的内存达到 4GB。

在 Linux 系统中，进程的 4GB 内存空间被分为两个部分——用户空间与内核空间。用户空间地址一般分布为 0～3GB（即 PAGE_OFFSET，在 0x86 中它等于 0xC0000000），这样，剩下的 3～4GB 为内核空间，如图 11.5 所示。用户进程通常情况下只能访问用户空间的虚拟地址，不能访问内核空间虚拟地址。用户进程只有通过系统调用（代表

用户进程在内核态执行）等方式才可以访问到内核空间。

图 11.5　用户空间与内核空间

每个进程的用户空间都是完全独立、互不相干的，用户进程各自有不同的页表。而内核空间是由内核负责映射，它并不会跟着进程改变，是固定的。内核空间地址有自己对应的页表，内核的虚拟空间独立于其他程序。

Linux 中 1GB 的内核地址空间又被划分为物理内存映射区、虚拟内存分配区、高端页面映射区、专用页面映射区和系统保留映射区这几个区域，如图 11.6 所示。

一般情况下，物理内存映射区最大长度为 896MB，系统的物理内存被顺序映射在内核空间的这个区域中。当系统物理内存大于 896MB 时，超过物理内存映射区的那部分内存称为高端内存（而未超过物理内存映射区的内存通常被称为常规内存），内核在存取高端内存时必须将它们映射到高端页面映射区。

Linux 保留内核空间最顶部 FIXADDR_TOP～4GB 的区域作为保留区。

图 11.6　Linux 内核地址空间

紧接着最顶端的保留区以下的一段区域为专用页面映射区（FIXADDR_START～FIXADDR_ TOP），它的总尺寸和每一页的用途由 fixed_address 枚举结构在编译时预定义，用__fix_to_virt(index)可获取专用区内预定义页面的逻辑地址。其开始地址和结束地址宏定义如下：

```
#define FIXADDR_START       (FIXADDR_TOP - __FIXADDR_SIZE)
#define FIXADDR_TOP     ((unsigned long)__FIXADDR_TOP)
#define __FIXADDR_TOP  0xfffff000
```

接下来，如果系统配置了高端内存，则位于专用页面映射区之下的就是一段高端内存映射区，其起始地址为 PKMAP_BASE，定义如下：

```
#define PKMAP_BASE ( (FIXADDR_BOOT_START - PAGE_SIZE*(LAST_PKMAP + 1))
& PMD_MASK )
```

其中所涉及的宏定义如下：

```
#define FIXADDR_BOOT_START (FIXADDR_TOP - __FIXADDR_BOOT_SIZE)
#define LAST_PKMAP  PTRS_PER_PTE
#define PTRS_PER_PTE    512
#define PMD_MASK    (~(PMD_SIZE-1))
# define PMD_SIZE   (1UL << PMD_SHIFT)
#define PMD_SHIFT   21
```

在物理区和高端映射区之间为虚存内存分配区（VMALLOC_START～VMALLOC_END），用于 vmalloc()函数，它的前部与物理内存映射区有一个隔离带，后部与高端映射区也有一个隔离带，vmalloc 区域定义如下：

```
#define VMALLOC_OFFSET (8*1024*1024)
#define VMALLOC_START  (((unsigned long) high_memory +
   vmalloc_earlyreserve + 2*VMALLOC_OFFSET-1) & ~(VMALLOC_OFFSET-1))

#ifdef CONFIG_HIGHMEM   /*支持高端内存*/
# define VMALLOC_END    (PKMAP_BASE-2*PAGE_SIZE)
#else                 /*不支持高端内存*/
# define VMALLOC_END    (FIXADDR_START-2*PAGE_SIZE)
#endif
```

当系统物理内存超过 4GB 时，必须使用 CPU 的扩展分页（PAE）模式所提供的 64 位页目录项才能存取到 4GB 以上的物理内存，这需要 CPU 的支持。加入了 PAE 功能的 Intel Pentium Pro 及其后的 CPU 允许内存最大可配置到 64GB，具备 36 位物理地址空间寻址能力。

由此可见，在 3～4GB 之间的内核空间中，从低地址到高地址依次为：物理内存映射区—隔离带—vmalloc 虚拟内存分配器—隔离带—高端内存映射区—专用页面映射区—保留区。

11.3 内存存取

11.3.1 用户空间内存动态申请

在用户空间动态申请内存的函数为 malloc()，这个函数在各种操作系统上的使用是一致的，malloc()申请的内存的释放函数为 free()。

malloc()的内存一定要被 free()，否则会造成内存泄漏。理想情况下，malloc()和 free()应成对出现，即谁申请，就由谁释放。例如下面的一段程序：

```
char * function(void)
{
  char *p;
  p = (char *)malloc(…);
  if(p==NULL)
    ...;
    ... /* 一系列针对 p 的操作 */
  return p;
}
```

在某处调用 function()，用完 function()中动态申请的内存后将其 free，如下：

```
char *q = function();
…
free(q);
```

上述代码并不合理的，因为违反了 malloc()和 free()成对出现的原则。不满足这个原则，会导致代码的耦合度增大，因为用户在调用 function()函数时需要知道其内部细节。

较好的做法是在调用处申请内存，并传入 function()函数，如下所示：

```
char *p=malloc(…);
if(p==NULL)
...;
function(p);
...
free(p);
p=NULL;
```

而函数 function()则接收参数 p，如下所示：

```
void function(char *p)
{
  ... /* 一系列针对 p 的操作 */
}
```

完全让 malloc()和 free()成对出现有时候很难做到，即便如此，也应尽力将 malloc()申请内存的释放限制在本模块范围之内。如果在 A 模块申请的内存需在 B 模块释放，一般而言，软件结构的设计可能存在问题。

11.3.2 内核空间内存动态申请

在 Linux 内核空间申请内存涉及的函数主要包括 kmalloc()、__get_free_pages()和 vmalloc()等。kmalloc()和_ _get_free_pages()（及其类似函数）申请的内存位于物理内存映射区域，而且在物理上也是连续的，它们与真实的物理地址只有一个固定的偏移，因此存在较简单的转换关系。而 vmalloc()在虚拟内存空间给出一块连续的内存区，实质上，这片连续的虚拟内存在物理内存中并不一定连续，而 vmalloc()申请的虚拟内存和物理内存之间也没有简单的换算关系。

1．kmalloc()

```
void *kmalloc(size_t size, int flags);
```

给 kmalloc()的第一个参数是要分配的块的大小，第二个参数为分配标志，用于控制 kmalloc()的行为。

最常用的分配标志是 GFP_KERNEL，其含义是在内核空间的进程中申请内存。kmalloc()的底层依赖__get_free_pages()实现，分配标志的前缀 GFP 正好是这个底层函数的缩写。使用 GFP_ KERNEL 标志申请内存时，若暂时不能满足，则进程会睡眠等待页，即会引起阻塞，因此不能在中断上下文或持有自旋锁的时候使用 GFP_KERNE 申请内存。

在中断处理函数、tasklet 和内核定时器等非进程上下文中不能阻塞，此时驱动应当使用 GFP_ATOMIC 标志来申请内存。当使用 GFP_ATOMIC 标志申请内存时，若不存在空闲页，则不等待，直接返回。

其他的相对不常用的申请标志还包括 GFP_USER（用来为用户空间页分配内存，可

能阻塞）、GFP_HIGHUSER（类似 GFP_USER，但是从高端内存分配）、GFP_NOIO（不允许任何 I/O 初始化）、GFP_NOFS（不允许进行任何文件系统调用）、__GFP_DMA（要求分配在能够 DMA 的内存区）、__GFP_HIGHMEM（指示分配的内存可以位于高端内存）、__GFP_COLD（请求一个较长时间不访问的页）、__GFP_NOWARN（当一个分配无法满足时，阻止内核发出警告）、__GFP_HIGH（高优先级请求，允许获得被内核保留给紧急状况使用的最后的内存页）、__GFP_REPEAT（分配失败则尽力重复尝试）、__GFP_NOFAIL（标志只许申请成功，不推荐）和__GFP_NORETRY（若申请不到，则立即放弃）。

使用 kmalloc()申请的内存应使用 kfree()释放，这个函数的用法和用户空间的 free()类似。

2. __get_free_pages ()

__get_free_pages()系列函数/宏是 kmalloc()实现的基础，__get_free_pages()系列函数/宏包括 get_zeroed_page()、__get_free_page()和__get_free_pages()。

```
get_zeroed_page(unsigned int flags);
```

该函数返回一个指向新页的指针并且将该页清零。

```
__get_free_page(unsigned int flags);
```

该宏返回一个指向新页的指针但是该页不清零，它实际上为：

```
#define __get_free_page(gfp_mask) \
        __get_free_pages((gfp_mask),0)
```

就是调用了下面的__get_free_pages()申请 1 页。

```
__get_free_pages(unsigned int flags, unsigned int order);
```

该函数可分配多个页并返回分配内存的首地址，分配的页数为 2^{order}，分配的页也不清零。order 允许的最大值是 10（即 1024 页）或者 11（即 2048 页），依赖于具体的硬件平台。

__get_free_pages ()和 get_zeroed_page ()的实现中调用了 alloc_pages()函数，alloc_pages()既可以在内核空间分配，也可以在用户空间分配，其原型为：

```
struct page * alloc_pages(int gfp_mask, unsigned long order);
```

参数含义与__get_free_pages()类似，但它返回分配的第一个页的描述符而非首地址。

使用__get_free_pages()系列函数/宏申请的内存应使用下列函数释放：

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

如果申请和释放的 order 不一样，则会引起内存的混乱。

__get_free_pages 等函数在使用时，其申请标志的值与 kmalloc()完全一样，各标志的含义也与 kmalloc()完全一致，最常用的是 GFP_KERNEL 和 GFP_ATOMIC。

3．vmalloc()

vmalloc()一般用在为只存在于软件中（没有对应的硬件意义）的较大的顺序缓冲区分配内存，vmalloc()远大于__get_free_pages()的开销，为了完成 vmalloc()，新的页表需要被建立。因此，只是调用 vmalloc()来分配少量的内存（如 1 页）是不妥的。

vmalloc()申请的内存应使用 vfree()释放，vmalloc()和 vfree()的函数原型如下：

```
void *vmalloc(unsigned long size);
void vfree(void * addr);
```

vmalloc()不能用在原子上下文中，因为它的内部实现使用了标志为 GFP_KERNEL 的 kmalloc()。

使用 vmalloc 函数的一个例子函数是 create_module()系统调用，它利用 vmalloc()函数来获取被创建模块需要的内存空间。

4．slab 与内存池

在操作系统的运作过程中，经常会涉及大量对象的重复生成、使用和释放问题。在 Linux 系统中所用到的对象，比较典型的例子是 inode、task_struct 等。如果我们能够用合适的方法使得在对象前后两次被使用时分配在同一块内存或同一类内存空间且保留了基本的数据结构，就可以大大提高效率。slab 算法就是针对上述特点设计的。

- 创建 slab 缓存。

```
struct kmem_cache *kmem_cache_create(const char *name, size_t size,
    size_t align, unsigned long flags,
    void (*ctor)(void*, struct kmem_cache *, unsigned long),
    void (*dtor)(void*, struct kmem_cache *, unsigned long));
```

kmem_cache_create()用于创建一个 slab 缓存，它是一个可以驻留任意数目全部同样大小的后备缓存。参数 size 是要分配的每个数据结构的大小，参数 flags 是控制如何进行分配的位掩码，包括 SLAB_NO_REAP（即使内存紧缺也不自动收缩这块缓存）、SLAB_HWCACHE_ALIGN（每个数据对象被对齐到一个缓存行）、SLAB_CACHE_DMA（要求数据对象在 DMA 内存区分配）等。

- 分配 slab 缓存。

```
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);
```

上述函数在 kmem_cache_create()创建的 slab 后备缓冲中分配一块并返回首地址指针。

- 释放 slab 缓存。

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

上述函数释放由 kmem_cache_alloc()分配的缓存。

- 收回 slab 缓存。

```
int kmem_cache_destroy(struct kmem_cache *cachep);
```

代码清单 11.3 给出了 slab 缓存的使用范例。

代码清单 11.3　slab 缓存使用范例

```
/*创建slab缓存*/
static kmem_cache_t  *xxx_cachep;
xxx_cachep = kmem_cache_create("xxx", sizeof(struct xxx),
              0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL, NULL);
/*分配slab缓存*/
struct xxx *ctx;
ctx = kmem_cache_alloc(xxx_cachep, GFP_KERNEL);
...//使用slab缓存
/*释放slab缓存*/
kmem_cache_free(xxx_cachep, ctx);
kmem_cache_destroy(xxx_cachep);
```

除了 slab 以外，在 Linux 内核中还包含对内存池的支持，内存池技术也是一种非常经典的用于分配大量小对象的后备缓存技术。

Linux 内核中，与内存池相关的操作包括如下几种。

- 创建内存池。

```
mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn,
     mempool_free_t *free_fn, void *pool_data);
```

mempool_create()函数用于创建一个内存池，min_nr 参数是需要预分配对象的数目，alloc_fn 和 free_fn 是指向内存池机制提供的标准对象分配和回收函数的指针，其原型分别为：

```
typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data);
```

和

```
typedef void (mempool_free_t)(void *element, void *pool_data);
```

pool_data 是分配和回收函数用到的指针，gfp_mask 是分配标记。只有当__GFP_WAIT 标记被指定时，分配函数才会休眠。

- 分配和回收对象。

在内存池中分配和回收对象需由以下函数来完成：

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

mempool_alloc()用来分配对象，如果内存池分配器无法提供内存，那么就可以用预分配的池。

- 回收内存池。

```
void mempool_destroy(mempool_t *pool);
```

mempool_create()函数创建的内存池需由 mempool_destroy()来回收。

11.3.3　虚拟地址与物理地址关系

对于内核物理内存映射区的虚拟内存，使用 virt_to_phys()可以实现内核虚拟地址转化为物理地址，virt_to_phys()的定义如代码清单 11.4 所示。

代码清单 11.4　virt_to_phys()函数

```
1 #define __pa(x)  ((unsigned long)(x)-PAGE_OFFSET)
2 extern inline unsigned long virt_to_phys(volatile void * address)
3 {
4   return __pa(address);
5 }
```

上面转换过程中调用的__pa()会将虚拟地址减去 PAGE_OFFSET，通常为 3GB。

与之对应的函数为 phys_to_virt()，它将物理地址转化为内核虚拟地址，phys_to_virt()的定义如代码清单 11.5 所示。

代码清单 11.5　phys_to_virt()函数

```
1 #define __va(x)  ((void *)((unsigned long)(x)+PAGE_OFFSET))
2 extern inline void * phys_to_virt(unsigned long address)
3 {
4   return __va(address);
5 }
```

上面转换过程中调用的__va()会将物理地址加上 PAGE_OFFSET ，通常为 3GB。

注意，上述方法仅适用于常规内存，高端内存的虚拟地址与物理地址之间不存在如此简单的换算关系。

11.4 设备 I/O 端口和 I/O 内存的访问

设备通常会提供一组寄存器来用于控制设备、读写设备和获取设备状态，即控制寄存器、数据寄存器和状态寄存器。这些寄存器可能位于 I/O 空间，也可能位于内存空间。当位于 I/O 空间时，通常被称为 I/O 端口，位于内存空间时，对应的内存空间被称为 I/O 内存。

11.4.1　Linux I/O 端口和 I/O 内存访问接口

1．I/O 端口

在 Linux 设备驱动中，应使用 Linux 内核提供的函数来访问定位于 I/O 空间的端口，这些函数包括如下几种。

- 读写字节端口（8 位宽）。

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

- 读写字端口（16 位宽）。

```
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

- 读写长字端口（32 位宽）。

```
unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
```

- 读写一串字节。

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
```

- insb()从端口 port 开始读 count 个字节端口，并将读取结果写入 addr 指向的内存；outsb()将 addr 指向的内存的 count 个字节连续地写入 port 开始的端口。
- 读写一串字。

```
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
```

- 读写一串长字。

```
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

上述各函数中 I/O 端口号 port 的类型高度依赖于具体的硬件平台，因此，只是写出了 unsigned。

2．I/O 内存

在内核中访问 I/O 内存之前，需首先使用 ioremap()函数将设备所处的物理地址映射到虚拟地址。ioremap()的原型如下：

```
void *ioremap(unsigned long offset, unsigned long size);
```

ioremap()与 vmalloc()类似，也需要建立新的页表，但是它并不进行 vmalloc()中所执行的内存分配行为。ioremap()返回一个特殊的虚拟地址，该地址可用来存取特定的物理地址范围。通过 ioremap()获得的虚拟地址应该被 iounmap()函数释放，其原型如下：

```
void iounmap(void * addr);
```

在设备的物理地址被映射到虚拟地址之后，尽管可以直接通过指针访问这些地址，但是可以使用 Linux 内核的如下一组函数来完成设备内存映射的虚拟地址的读写，这些函数如下所示。

- 读 I/O 内存。

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
```

与上述函数对应的较早版本的函数为（这些函数在 Linux 2.6 中仍然被支持）：

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
```

- 写 I/O 内存。

```
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

与上述函数对应的较早版本的函数为（这些函数在 Linux 2.6 中仍然被支持）：

```
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

- 读一串 I/O 内存。

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);
```

- 写一串 I/O 内存。

```
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);
```

- 复制 I/O 内存。

```
void memcpy_fromio(void *dest, void *source, unsigned int count);
void memcpy_toio(void *dest, void *source, unsigned int count);
```

- 设置 I/O 内存。

```
void memset_io(void *addr, u8 value, unsigned int count);
```

3. 把 I/O 端口映射到内存空间

```
void *ioport_map(unsigned long port, unsigned int count);
```

通过这个函数，可以把 port 开始的 count 个连续的 I/O 端口重映射为一段“内存空间”。然后就可以在其返回的地址上像访问 I/O 内存一样访问这些 I/O 端口。当不再需要这种映射时，需要调用下面的函数来撤销。

```
void ioport_unmap(void *addr);
```

实际上，分析 ioport_map()的源代码可发现，映射到内存空间行为实际上是给开发人员制造的一个“假象”，并没有映射到内核虚拟地址，仅仅是为了让工程师可使用统一的 I/O 内存访问接口访问 I/O 端口。

11.4.2 申请与释放设备 I/O 端口和 I/O 内存

1. I/O 端口申请

Linux 内核提供了一组函数用于申请和释放 I/O 端口。

```
struct resource *request_region(unsigned long first, unsigned long n,
const char *name);
```

这个函数向内核申请 *n* 个端口，这些端口从 first 开始，name 参数为设备的名称。如果分配成功返回值是非 NULL，如果返回 NULL，则意味着申请端口失败。

当用 request_region()申请的 I/O 端口使用完成后，应当使用 release_region()函数将它们归还给系统，这个函数的原型如下：

```
void release_region(unsigned long start, unsigned long n);
```

2. I/O 内存申请

同样，Linux 内核也提供了一组函数用于申请和释放 I/O 内存的范围。

```
struct resource *request_mem_region(unsigned long start, unsigned long
len, char *name);
```

这个函数向内核申请 *n* 个内存地址，这些地址从 first 开始，name 参数为设备的名称。如果分配成功返回值是非 NULL，如果返回 NULL，则意味着申请 I/O 内存失败。

当用 request_mem_region() 申请的 I/O 内存使用完成后，应当使用

release_mem_region()函数将它们归还给系统，这个函数的原型如下：

```
void release_mem_region(unsigned long start, unsigned long len);
```

上述 request_region()和 release_mem_region()都不是必须的，但建议使用。其任务是检查申请的资源是否可用，如果可用则申请成功，并标志为已经使用，其他驱动想再次申请该资源时就会失败。

有许多设备驱动程序在没有申请 I/O 端口和 I/O 内存之前就直接访问了，这不够安全。

11.4.3 设备 I/O 端口和 I/O 内存访问流程

综合 11.3 节和本节的内容，可以归纳出设备驱动访问 I/O 端口和 I/O 内存的步骤。

I/O 端口访问的一种途径是直接使用 I/O 端口操作函数：在设备打开或驱动模块被加载时申请 I/O 端口区域，之后使用 inb()、outb()等进行端口访问，最后，在设备关闭或驱动被卸载时释放 I/O 端口范围。整个流程如图 11.7 所示。

图 11.7 I/O 端口的访问流程（不映射到内存空间）

I/O 端口访问的另一种途径是将 I/O 端口映射为内存进行访问：在设备打开或驱动模块被加载时，申请 I/O 端口区域并使用 ioport_map()映射到内存，之后使用 I/O 内存的函数进行端口访问，最后，在设备关闭或驱动被卸载时释放 I/O 端口并释放映射。整个流程如图 11.8 所示。

I/O 内存的访问步骤如图 11.9 所示，首先是调用 request_mem_region()申请资源，接着将寄存器地址通过 ioremap()映射到内核空间虚拟地址，之后就可以通过 Linux 设备访问编程接口访问这些设备的寄存器了。访问完成后，应对 ioremap()申请的虚拟地址进行释放，并释放 release_mem_ region()申请的 I/O 内存资源。

图 11.8　I/O 端口的访问流程（映射到内存空间）　　图 11.9　I/O 内存访问流程

11.4.4　将设备地址映射到用户空间

1．内存映射与 VMA

一般情况下，用户空间是不可能也不应该直接访问设备的，但是，设备驱动程序中可实现 mmap()函数，这个函数可使得用户空间能直接访问设备的物理地址。实际上，mmap()实现了这样的一个映射过程：它将用户空间的一段内存与设备内存关联，当用户访问用户空间的这段地址范围时，实际上会转化为对设备的访问。

这种能力对于显示适配器一类的设备非常有意义，如果用户空间可直接通过内存映射访问显存的话，屏幕帧的各点的像素将不再需要一个从用户空间到内核空间的复制的过程。

mmap()必须以 PAGE_SIZE 为单位进行映射，实际上，内存只能以页为单位进行映射，若要映射非 PAGE_SIZE 整数倍的地址范围，要先进行页对齐，强行以 PAGE_SIZE 的倍数大小进行映射。

从 file_operations 文件操作结构体可以看出，驱动中 mmap()函数的原型如下：

```
int(*mmap)(struct file *, struct vm_area_struct*);
```

驱动中的 mmap()函数将在用户进行 mmap()系统调用时最终被调用，mmap()系统调用的原型与 file_operations 中 mmap()的原型区别很大，如下所示：

```
caddr_t  mmap (caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset);
```

参数 fd 为文件描述符，一般由 open()返回，fd 也可以指定为−1，此时须指定 flags 参数中的 MAP_ANON，表明进行的是匿名映射。

len 是映射到调用用户空间的字节数，它从被映射文件开头 offset 个字节开始算起，offset 参数一般设为 0，表示从文件头开始映射。

prot 参数指定访问权限，可取如下几个值的“或”：PROT_READ（可读）、PROT_WRITE（可写）、PROT_EXEC（可执行）和 PROT_NONE（不可访问）。

参数 addr 指定文件应被映射到用户空间的起始地址，一般被指定为 NULL，这样，选择起始地址的任务将由内核完成，而函数的返回值就是映射到用户空间的地址。其类型 caddr_t 实际上就是 void *。

当用户调用 mmap()的时候，内核会进行如下处理。

① 在进程的虚拟空间查找一块 VMA。

② 将这块 VMA 进行映射。

③ 如果设备驱动程序或者文件系统的 file_operations 定义了 mmap()操作，则调用它。

④ 将这个 VMA 插入到进程的 VMA 链表中。

file_operations 中 mmap()函数的第一个参数就是步骤①中找到的 VMA。

由 mmap()系统调用映射的内存可由 munmap()解除映射，这个函数的原型如下：

int munmap(caddr_t addr, size_t len);

驱动程序中 mmap()的实现机制是建立页表，并填充 VMA 结构体中 vm_operations_struct 指针。VMA 即 vm_area_struct，用于描述一个虚拟内存区域，VMA 结构体的定义如代码清单 11.6 所示。

代码清单 11.6　VMA 结构体

```
struct vm_area_struct
{
  struct mm_struct *vm_mm; /* 所处的地址空间 */
  unsigned long vm_start; /* 开始虚拟地址 */
  unsigned long vm_end; /* 结束虚拟地址 */

  pgprot_t vm_page_prot; /* 访问权限 */
  unsigned long vm_flags; /* 标志 */
  ...
  /* 操作 VMA 的函数集指针 */
  struct vm_operations_struct *vm_ops;

  unsigned long vm_pgoff; /* 偏移（页帧号） */
  struct file *vm_file;
  void *vm_private_data;
  ...
};
```

VMA 结构体描述的虚地址介于 vm_start 和 vm_end 之间，而其 vm_ops 成员指向

这个 VMA 的操作集。针对 VMA 的操作都被包含在 vm_operations_struct 结构体中，vm_operations_struct 结构体的定义如代码清单 11.7 所示。

代码清单 11.7　vm_operations_struct 结构体

```
struct vm_operations_struct
{
  void(*open)(struct vm_area_struct *area); /*打开 VMA 的函数*/
  void(*close)(struct vm_area_struct *area); /*关闭 VMA 的函数*/
  struct page *(*nopage)(struct vm_area_struct *area, unsigned long address,
    int *type); /*访问的页不在内存时调用*/
  int(*populate)(struct vm_area_struct *area, unsigned long address, unsigned
    long len, pgprot_t prot, unsigned long pgoff, int nonblock);
  ...
};
```

在内核生成一个 VMA 后，它会调用该 VMA 的 open()函数，例如 fork 一个继承父继承资源的子进程时。但是，当用户进行 mmap()系统调用后，尽管 VMA 在设备驱动文件操作结构体的 mmap()被调用前就已产生，内核却不会调用 VMA 的 open()函数，通常需要在驱动的 mmap()函数中显示调用 vma->vm_ops->open()。代码清单 11.8 给出了一个 vm_operations_struct 的操作范例。

代码清单 11.8　vm_operations_struct 操作范例

```
static int xxx_mmap(struct file *filp, struct vm_area_struct *vma)
{
  if (remap_pfn_range(vma, vma->vm_start, vm->vm_pgoff, vma->vm_end - vma
    ->vm_start, vma->vm_page_prot))  /* 建立页表 */
    return  - EAGAIN;
  vma->vm_ops = &xxx_remap_vm_ops;
  xxx_vma_open(vma);
  return 0;
}

void xxx_vma_open(struct vm_area_struct *vma)   //VMA 打开函数
{
  ...
   printk(KERN_NOTICE "xxx VMA open, virt %lx, phys %lx\n", vma->vm_start,
    vma->vm_pgoff << PAGE_SHIFT);
}

void xxx_vma_close(struct vm_area_struct *vma)   //VMA 关闭函数
{
  ...
  printk(KERN_NOTICE "xxx VMA close.\n");
}

static struct vm_operations_struct xxx_remap_vm_ops =   //VMA 操作
```

```
结构体
    25 {
    26   .open = xxx_vma_open,
    27   .close = xxx_vma_close,
    28   ...
    29 };
```

第 2 行调用的 remap_pfn_range()创建页表，以 VMA 结构体的成员（VMA 的数据成员是内核根据用户的请求自己填充的）作为 remap_pfn_range()的参数，映射的虚拟地址范围是 vma->vm_ start 至 vma->vm_end。

remap_pfn_range()函数的原型如下：

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long addr,
          unsigned long pfn, unsigned long size, pgprot_t prot);
```

其中的 addr 参数表示内存映射开始处的虚拟地址。remap_pfn_range()函数为 addr～addr+size 之间的虚拟地址构造页表。

pfn 是虚拟地址应该映射到的物理地址的页帧号，实际上就是物理地址右移 PAGE_SHIFT 位。若 PAGE_SIZE 为 4KB，则 PAGE_SHIFT 为 12，因为 PAGE_SIZE 等于 1<<PAGE_SHIFT。

prot 是新页所要求的保护属性。

在驱动程序中，我们能使用 remap_pfn_range()映射内存中的保留页（如 X86 系统中的 640KB～1MB 区域）和设备 I/O 内存，另外，kmalloc()申请的内存若要被映射到用户空间可以通过 mem_map_ reserve()设置为保留后进行。代码清单 11.9 给出了映射 kmalloc()申请的内存到用户空间的典型范例。

代码清单 11.9　映射 kmalloc()申请的内存到用户空间范例

```
1  /*内核模块加载函数*/
2  int __init kmalloc_map_init(void)
3  {
4    ...//申请设备号、添加 cdev 结构体
5    buffer = kmalloc(BUF_SIZE, GFP_KERNEL); //申请 buffer
6
7    for (page = virt_to_page(buffer); page < virt_to_page(buffer +
BUF_SIZE);
8      page++)
9    {
10     mem_map_reserve(page); //置页为保留
11   }
12 }
13 /*mmap()函数*/
14 static int kmalloc_map_mmap(struct file *filp, struct vm_area_struct
*vma)
15 {
16   unsigned long page, pos;
```

```
17  unsigned long start = (unsigned long)vma->vm_start;
18  unsigned long size = (unsigned long)(vma->vm_end - vma->vm_start);
19  printk(KERN_INFO "mmaptest_mmap called\n");
20  /* 用户要映射的区域太大 */
21  if (size > BUF_SIZE)
22    return  - EINVAL;
23
24  pos = (unsigned long)buffer;
25  /* 映射 buffer 中的所有页 */
26  while (size > 0)
27  {
28    /* 每次映射一页 */
29    page = virt_to_phys((void*)pos);
30    if (remap_page_range(start, page, PAGE_SIZE, PAGE_SHARED))
31      return  - EAGAIN;
32    start += PAGE_SIZE;
33    pos += PAGE_SIZE;
34    size -= PAGE_SIZE;
35  } return 0;
36 }
```

第 30 行调用 remap_page_range(start, page, PAGE_SIZE, PAGE_SHARED)的第 4 个参数 PAGE_ SHARED 实际上是_PAGE_PRESENT | _PAGE_USER | _PAGE_RW，表明可读写并映射到用户空间。

通常，I/O 内存被映射时需要是 nocache 的，这时候，我们应该对 vma->vm_page_prot 设置 nocache 标志之后再映射，如代码清单 11.10 所示。

代码清单 11.10　以 nocache 方式将内核空间映射到用户空间

```
1  static int xxx_nocache_mmap(struct file *filp, struct vm_area_struct
*vma)
2  {
3      vma->vm_page_prot  =  pgprot_noncached(vma->vm_page_prot);// 赋
nocache 标志
4    vma->vm_pgoff = ((u32)map_start >> PAGE_SHIFT);
5    //映射
6          if  (remap_pfn_range(vma,  vma->vm_start,  vma->vm_pgoff,
vma->vm_end - vma
7      ->vm_start, vma->vm_page_prot))
8      return  - EAGAIN;
9    return 0;
10 }
```

上述代码第 3 行的 pgprot_noncached()是一个宏，它高度依赖于 CPU 体系结构，

ARM 的 pgprot_noncached()定义如下：

```
    #define pgprot_noncached(prot) __pgprot(pgprot_val(prot)          &
~(L_PTE_CACHEABLE | L_PTE_ BUFFERABLE))
```

另一个比 pgprot_noncached()稍微少一些限制的宏是 pgprot_writecombine()，它的定义如下：

```
    #define   pgprot_writecombine(prot)   __pgprot(pgprot_val(prot)   &
~L_PTE_CACHEABLE)
```

pgprot_noncached()实际禁止了相关页的 Cache 和写缓冲（write buffer），pgprot_writecombine()则没有禁止写缓冲。ARM 的写缓冲器是一个非常小的 FIFO 存储器，位于处理器核与主存之间，其目的在于将处理器核和 Cache 从较慢的主存写操作中解脱出来。写缓冲区与 Cache 在存储层次上处于同一层次，但是它只作用于写主存。

2．nopage()函数

除了 remap_pfn_range()以外，在驱动程序中实现 VMA 的 nopage()函数通常可以为设备提供更加灵活的内存映射途径。当访问的页不在内存，即发生缺页异常时，nopage()会被内核自动调用。这是因为，当发生缺页异常时，系统会经过如下处理过程。

① 找到缺页的虚拟地址所在的 VMA。

② 如果必要，分配中间页目录表和页表。

③ 如果页表项对应的物理页面不存在，则调用这个 VMA 的 nopage()方法，它返回物理页面的页描述符。

④ 将物理页面的地址填充到页表中。

实现 nopage()后，用户空间可以通过 mremap()系统调用重新绑定映射区域所绑定的地址，代码清单 11.11 给出了一个设备驱动中使用 nopage()的典型范例。

代码清单 11.11　nopage()函数使用范例

```
    1  static int xxx_mmap(struct file *filp, struct vm_area_struct *vma)
    2  {
    3    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
    4    if (offset >= __pa(high_memory) || (filp->f_flags &O_SYNC))
    5      vma->vm_flags |= VM_IO;
    6    vma->vm_flags |= VM_RESERVED; //预留
    7    vma->vm_ops = &xxx_nopage_vm_ops;
    8    xxx_vma_open(vma);
    9    return 0;
    10 }
    11
    12 struct page *xxx_vma_nopage(struct vm_area_struct *vma, unsigned
long
    13   address, int *type)
    14 {
    15   struct page *pageptr;
    16   unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
    17   unsigned long physaddr = address - vma->vm_start + offset; /* 物
理地址 */
```

```
18   unsigned long pageframe = physaddr >> PAGE_SHIFT; /* 页帧号 */
19   if (!pfn_valid(pageframe))    /* 页帧号有效? */
20     return NOPAGE_SIGBUS;
21   pageptr = pfn_to_page(pageframe);  /* 页帧号->页描述符 */
22   get_page(pageptr);    /* 获得页，增加页的使用计数 */
23   if (type)
24     *type = VM_FAULT_MINOR;
25   return pageptr;   /*返回页描述符 */
26 }
```

上述函数对常规内存进行映射，返回一个页描述符，可用于扩大或缩小映射的内存区域。

由此可见，nopage()与 remap_pfn_range()的一个较大区别在于 remap_pfn_range()一般用于设备内存映射，而 nopage()还可用于 RAM 映射。

大多数设备驱动都不需要提供设备内存到用户空间的映射能力，因为，对于串口等面向流的设备而言，实现这种映射毫无意义。

11.5 I/O 内存静态映射

在将 Linux 移植到目标电路板的过程中，通常会建立外设 I/O 内存物理地址到虚拟地址的静态映射，这个映射通过在电路板对应的 map_desc 结构体数组中添加新的成员来完成，map_desc 结构体的定义如代码清单 11.12 所示。

代码清单 11.12　map_desc 结构体

```
1 struct map_desc
2 {
3   unsigned long virtual;//虚拟地址
4   unsigned long pfn ;// __phys_to_pfn(phy_addr)
5   unsigned long length; //大小
6   unsigned int type;//类型
7 };
```

例如，S3C2440 ARM 评估板 SMDK2440 的 map_desc 结构体数组定义如代码清单 11.13 所示，位于文件/arch/ arm/mach-smdk2440.c。

代码清单 11.13　SMDK2440 评估板的 map_desc 数组

```
1  static struct map_desc smdk2440_iodesc[] __initdata = {
2   /* ISA IO 空间映射 */
3   {
4       .virtual    = (u32)S3C24XX_VA_ISA_WORD,
5       .pfn        = __phys_to_pfn(S3C2410_CS2),
6       .length     = 0x10000,
7       .type       = MT_DEVICE,
8   }, {
9       .virtual    = (u32)S3C24XX_VA_ISA_WORD + 0x10000,
10      .pfn        = __phys_to_pfn(S3C2410_CS2 + (1<<24)),
```

```
    .length     = SZ_4M,
    .type       = MT_DEVICE,
 }, {
    .virtual    = (u32)S3C24XX_VA_ISA_BYTE,
    .pfn        = __phys_to_pfn(S3C2410_CS2),
    .length     = 0x10000,
    .type       = MT_DEVICE,
 }, {
    .virtual    = (u32)S3C24XX_VA_ISA_BYTE + 0x10000,
    .pfn        = __phys_to_pfn(S3C2410_CS2 + (1<<24)),
    .length     = SZ_4M,
    .type       = MT_DEVICE,
 }
};
```

Linux 操作系统移植到特定平台上，MACHINE_START 到 MACHINE_END 宏之间的定义针对特定电路板而设计，其中的 map_io()成员函数完成 I/O 内存的静态映射，代码清单 11.14 给出了 SMDK2440 电路板的 MACHINE_START、MACHINE_END 宏的例子。

代码清单 11.14　SMDK2440 MACHINE_START、MACHINE_END 宏

```
1  MACHINE_START(S3C2440, "SMDK2440")
2   /* Maintainer: Ben Dooks <ben@fluff.org> */
3   .phys_io    = S3C2410_PA_UART,
4   .io_pg_offst    = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
5   .boot_params    = S3C2410_SDRAM_PA + 0x100,
6
7   .init_irq   = s3c24xx_init_irq,
8   .map_io     = smdk2440_map_io,
9   .init_machine   = smdk2440_machine_init,
10  .timer      = &s3c24xx_timer,
11  MACHINE_END
```

第 8 行赋值给 map_io 的 smdk2440_map_io()函数完成 SMDK2440 电路板 I/O 内存的静态映射，最终调用的是 cpu->map_io()建立 map_desc 数组中物理内存和虚拟内存的静态映射关系。

在一个已经移植好 OS 的内核中，驱动工程师完全可以对非常规内存区域的 I/O 内存（外设控制器寄存器、MCU 内部集成的外设控制器寄存器等）依照电路板的资源使用情况添加到 map_ desc 数组中，代码清单 11.15 的例子给出了内存空间资源的使用情况（注释部分）与 map_desc 数组的对应关系。

代码清单 11.15　根据电路板内存资源情况定义 map_desc

```
1  /*
2   * 逻辑地址 物理地址
3   * e8000000  40000000    PCI memory      PHYS_PCI_MEM_BASE   (max
512M)
4   * ec000000  61000000    PCI 配置空间  PHYS_PCI_CONFIG_BASE    (max
16M)
5   * ed000000  62000000    PCI V3 regs     PHYS_PCI_V3_BASE    (max
64k)
6   * ee000000  60000000    PCI IO          PHYS_PCI_IO_BASE    (max
16M)
7   * ef000000          Cache flush
8   * f1000000  10000000    核心模块寄存器
```

```
 * f1100000  11000000    系统控制寄存器
 * f1200000  12000000    EBI 寄存器
 * f1300000  13000000    计数器/定时器
 * f1400000  14000000    中断控制器
 * f1600000  16000000    UART 0
 * f1700000  17000000    UART 1
 * f1a00000  1a000000    调试用 LEDs
 * f1b00000  1b000000    GPIO
 */

static struct map_desc ap_io_desc[] __initdata = {
 {
     .virtual    = IO_ADDRESS(INTEGRATOR_HDR_BASE),
     .pfn        = __phys_to_pfn(INTEGRATOR_HDR_BASE),
     .length     = SZ_4K,
     .type       = MT_DEVICE
 }, {
     .virtual    = IO_ADDRESS(INTEGRATOR_SC_BASE),
     .pfn        = __phys_to_pfn(INTEGRATOR_SC_BASE),
     .length     = SZ_4K,
     .type       = MT_DEVICE
 }, {
     .virtual    = IO_ADDRESS(INTEGRATOR_EBI_BASE),
     .pfn        = __phys_to_pfn(INTEGRATOR_EBI_BASE),
     .length     = SZ_4K,
     .type       = MT_DEVICE
 }, {
     .virtual    = IO_ADDRESS(INTEGRATOR_CT_BASE),
     .pfn        = __phys_to_pfn(INTEGRATOR_CT_BASE),
     .length     = SZ_4K,
     .type       = MT_DEVICE
 }, {
     .virtual    = IO_ADDRESS(INTEGRATOR_IC_BASE),
     .pfn        = __phys_to_pfn(INTEGRATOR_IC_BASE),
     .length     = SZ_4K,
     .type       = MT_DEVICE
 }, {
     .virtual    = IO_ADDRESS(INTEGRATOR_UART0_BASE),
     .pfn        = __phys_to_pfn(INTEGRATOR_UART0_BASE),
     .length     = SZ_4K,
     .type       = MT_DEVICE
 }, {
```

```
51      .virtual    = IO_ADDRESS(INTEGRATOR_UART1_BASE),
52      .pfn        = __phys_to_pfn(INTEGRATOR_UART1_BASE),
53      .length     = SZ_4K,
54      .type       = MT_DEVICE
55  }, {
56      .virtual    = IO_ADDRESS(INTEGRATOR_DBG_BASE),
57      .pfn        = __phys_to_pfn(INTEGRATOR_DBG_BASE),
58      .length     = SZ_4K,
59      .type       = MT_DEVICE
60  }, {
61      .virtual    = IO_ADDRESS(INTEGRATOR_GPIO_BASE),
62      .pfn        = __phys_to_pfn(INTEGRATOR_GPIO_BASE),
63      .length     = SZ_4K,
64      .type       = MT_DEVICE
65  }, {
66      .virtual    = PCI_MEMORY_VADDR,
67      .pfn        = __phys_to_pfn(PHYS_PCI_MEM_BASE),
68      .length     = SZ_16M,
69      .type       = MT_DEVICE
70  }, {
71      .virtual    = PCI_CONFIG_VADDR,
72      .pfn        = __phys_to_pfn(PHYS_PCI_CONFIG_BASE),
73      .length     = SZ_16M,
74      .type       = MT_DEVICE
75  }, {
76      .virtual    = PCI_V3_VADDR,
77      .pfn        = __phys_to_pfn(PHYS_PCI_V3_BASE),
78      .length     = SZ_64K,
79      .type       = MT_DEVICE
80  }, {
81      .virtual    = PCI_IO_VADDR,
82      .pfn        = __phys_to_pfn(PHYS_PCI_IO_BASE),
83      .length     = SZ_64K,
84      .type       = MT_DEVICE
85  }
86 };
```

此后，在设备驱动中访问经过 map_desc 数组映射后的 I/O 内存时，直接在 map_desc 中该段的虚拟地址上加上相应的偏移即可，不再需要使用 ioremap()。

11.6 DMA

DMA 是一种无须 CPU 的参与就可以让外设与系统内存之间进行双向数据传输的硬件机制。使用 DMA 可以使系统 CPU 从实际的 I/O 数据传输过程中摆脱出来，从而大大提高系统的吞吐率。DMA 通常与硬件体系结构特别是外设的总线技术密切相关。

DMA 方式的数据传输由 DMA 控制器（DMAC）控制，在传输期间，CPU 可以并发地执行其他任务。当 DMA 结束后，DMAC 通过中断通知 CPU 数据传输已经结束，然后由 CPU 执行相应的中断服务程序进行后处理。

11.6.1 DMA 与 Cache 一致性

Cache 和 DMA 本身似乎是两个毫不相关的事物。Cache 被用做 CPU 针对内存的缓存，利用程序的空间局部性和时间局部性原理，达到较高的命中率从而避免 CPU 每次都一定要与相对慢速的内存交互数据来提高数据的访问速率。DMA 可以用做内存与外设之间传输数据的方式，这种传输方式之下，数据并不需要经过 CPU 中转。

假设 DMA 针对内存的目的地址与 Cache 缓存的对象没有重叠区域（如图 11.10 所示），DMA 和 Cache 之间将相安无事。但是，如果 DMA 的目的地址与 Cache 所缓存的内存地址访问有重叠（如图 11.11 所示），经过 DMA 操作，Cache 缓存对应的内存的数据已经被修改，而 CPU 本身并不知道，它仍然认为 Cache 中的数据就是内存中的数据，以后访问 Cache 映射的内存时，它仍然使用陈旧的 Cache 数据。这样就发生 Cache 与内存之间数据“不一致性”的错误。

所谓 Cache 数据与内存数据的不一致性，是指在采用 Cache 的系统中，同样一个数据可能既存在于 Cache 中，也存在于主存中，Cache 与主存中的数据一样则具有一致性，数据若不一样则具有不一致性。

华清远见

图 11.10　DMA 目的地址与 Cache 对象不交叉

图 11.11　DMA 目的地址与 Cache 对象交叉

需要特别注意的是，Cache 与内存的一致性问题经常被初学者遗忘。在发生 Cache 与内存不一致性错误后，驱动将无法正常运行。如果没有相关的背景知识，工程师几乎无法定位错误的原因，因为看起来所有的程序都是完全正确的。

图 11.12　跷跷板两端的 Cache 与 D

解决由于 DMA 导致的 Cache 一致性问题的最简单方法是直接禁止 DMA 目标地址范围内内存的 Cache 功能。当然，这将牺牲性能，但是却更可靠，图 11.12 所示为 Cache 和 DMA 在考虑性能和易用两个方面时处于跷跷板两端的比重。

Cache 的不一致问题并发只是发生在 DMA 的情况下，实际上，还存在于其他情况。例如，对于带 MMU 功能的 ARM 处理器，在开启 MMU 之前，需要先置 Cache 无效，TLB 也是如此，代码清单 11.16 所给出的一段汇编被用来完成此任务。

代码清单 11.16　置 ARM 的 Cache 无效

```
/* 使 cache 无效 */
"mov  r0, #0\n"
"mcr  p15, 0, r0, c7, c7, 0\n"    /* 使数据和指令 cache 无效 */
"mcr  p15, 0, r0, c7, c10, 4\n"   /* 放空写缓冲 */
"mcr  p15, 0, r0, c8, c7, 0\n"    /* 使 TLB 无效 */
```

11.6.2　Linux 下的 DMA 编程

首先 DMA 本身不属于一种等同于字符设备、块设备和网络设备的外设，它只是外设与内存交互数据的一种方式。因此，本节的标题不是“Linux 下的 DMA 驱动”而是“Linux 下的 DMA 编程”。

内存中用于与外设交互数据的一块区域被称做 DMA 缓冲区，在设备不支持 scatter/gather CSG，分散/聚集操作的情况下，DMA 缓冲区必须是物理上连续的。

对于 ISA 设备而言，其 DMA 操作只能在 16MB 以下的内存中进行，因此，在使用 kmalloc()和_ _get_free_pages()及其类似函数申请 DMA 缓冲区时应使用 GFP_DMA 标志，这样能保证获得的内存是具备 DMA 能力的（DMA-capable）。

内核中定义了__get_free_pages()针对 DMA 的“快捷方式”__get_dma_pages()，它在申请标志中添加了 GFP_DMA，如下所示：

```
#define __get_dma_pages(gfp_mask, order) \
        __get_free_pages((gfp_mask) | GFP_DMA,(order))
```

如果不想使用 $\log_2$size 即 order 为参数申请 DMA 内存，则可以使用另一个函数 dma_mem_ alloc()，其源代码如代码清单 11.17 所示。

代码清单 11.17　dma_mem_alloc()函数

```
1 static unsigned long dma_mem_alloc(int size)
2 {
3   int order = get_order(size);//大小->指数
4   return __get_dma_pages(GFP_KERNEL, order);
5 }
```

基于 DMA 的硬件使用总线地址而非物理地址，总线地址是从设备角度上看到的内存地址，物理地址则是从 CPU 角度上看到的未经转换的内存地址（经过转换的为虚拟地址）。虽然在 PC 上，对于 ISA 和 PCI 而言，总线地址即为物理地址，但并非每个平台都是如此。因为有时候接口总线通过桥接电路被连接，桥接电路会将 I/O 地址映射为不同的物理地址。例如，在 PReP（PowerPC Reference Platform）系统中，物理地址 0 在设备端看起来是 0x80000000，而 0 通常又被映射为虚拟地址 0xC0000000，所以同一地址就具备了三重身份：物理地址 0、总线地址 0x80000000 及虚拟地址 0xC0000000。还有一些系统提供了页面映射机制，它能将任意的页面映射为连续的外设总线地址。内核提供了如下函数用于进行简单的虚拟地址/总线地址转换：

```
unsigned long virt_to_bus(volatile void *address);
void *bus_to_virt(unsigned long address);
```

在使用 IOMMU 或反弹缓冲区的情况下，上述函数一般不会正常工作。而且，这两个函数并不建议使用。如图 11.13 所示，IOMMU 的工作原理与 CPU 内的 MMU 非常类似，不过它针对的是外设总线地址和内存地址之间的转化。由于 IOMMU 可以使得外设看到“虚拟地址”，因此在使用 IOMMU 的情况下，在修改映射寄存器后，可以使得 SG 中分段的缓冲区地址对外设变得连续。

图 11.13　MMU 与 IOMMU

设备并不一定能在所有的内存地址上执行 DMA 操作，在这种情况下应该通过下列函数执行 DMA 地址掩码：

```
int dma_set_mask(struct device *dev, u64 mask);
```

例如，对于只能在 24 位地址上执行 DMA 操作的设备而言，就应该调用 dma_set_mask (dev, 0xffffff)。

DMA 映射包括两个方面的工作：分配一片 DMA 缓冲区；为这片缓冲区产生设备可访问的地址。同时，DMA 映射也必须考虑 Cache 一致性问题。内核中提供了以下函数用于分配一个 DMA 一致性的内存区域：

```
void * dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *handle, gfp_t gfp);
```

上述函数的返回值为申请到的 DMA 缓冲区的虚拟地址，此外，该函数还通过参数 handle 返回 DMA 缓冲区的总线地址。handle 的类型为 dma_addr_t，代表的是总线地址。

dma_alloc_coherent()申请一片 DMA 缓冲区，进行地址映射并保证该缓冲区的 Cache 一致性。与 dma_alloc_coherent()对应的释放函数为：

```
void dma_free_coherent(struct device *dev, size_t size, void *cpu_addr, dma_addr_t handle);
```

以下函数用于分配一个写合并（writecombining）的 DMA 缓冲区：

```
void * dma_alloc_writecombine(struct device *dev, size_t size, dma_addr_t *handle, gfp_t gfp);
```

与 dma_alloc_writecombine()对应的释放函数 dma_free_writecombine()实际上就是 dma_free_ coherent()，因为它定义为：

```
#define dma_free_writecombine(dev,size,cpu_addr,handle) \
        dma_free_coherent(dev,size,cpu_addr,handle)
```

此外，Linux 内核还提供了 PCI 设备申请 DMA 缓冲区的函数 pci_alloc_consistent()，其原型为：

```
void * pci_alloc_consistent(struct pci_dev *pdev, size_t size, dma_addr_t *dma_addrp);
```

对应的释放函数为 pci_free_consistent()，其原型为：

```
void pci_free_consistent(struct pci_dev *pdev, size_t size, void *cpu_addr,     dma_addr_t dma_addr);
```

相对于一致性 DMA 映射而言，流式 DMA 映射的接口较为复杂。对于单个已经分配的缓冲区而言，使用 dma_map_single()可实现流式 DMA 映射，该函数原型为：

```
dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size,
    enum dma_data_direction direction);
```

如果映射成功，返回的是总线地址，否则，返回 NULL。第 4 个参数为 DMA 的方向，可能的值包括 DMA_TO_DEVICE、DMA_FROM_DEVICE、DMA_BIDIRECTIONAL 和 DMA_NONE。

dma_map_single()的反函数为 dma_unmap_single()，原型是：

```
void dma_unmap_single(struct device *dev, dma_addr_t dma_addr, size_t size,
    enum dma_data_direction direction);
```

通常情况下，设备驱动不应该访问 unmap 的流式 DMA 缓冲区，如果一定要这么做，可先使用如下函数获得 DMA 缓冲区的拥有权：

```
void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr,
size_t size, enum dma_data_direction direction);
```

在驱动访问完 DMA 缓冲区后，应该将其所有权返还给设备，通过如下函数完成：

```
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr,
size_t size, enum dma_data_direction direction);
```

如果设备要求较大的 DMA 缓冲区，在其支持 SG 模式的情况下，申请多个不连续的、相对较小的 DMA 缓冲区通常是防止申请太大的连续物理空间的方法。在 Linux 内核中，使用如下函数映射 SG：

```
int dma_map_sg(struct device *dev, struct scatterlist *sg, int nents,
enum dma_data_direction direction);
```

nents 是散列表（scatterlist）入口的数量，该函数的返回值是 DMA 缓冲区的数量，可能小于 nents。对于 scatterlist 中的每个项目，dma_map_sg()为设备产生恰当的总线地址，它会合并物理上临近的内存区域。

scatterlist 结构体的定义如代码清单 11.18 所示，它包含了 scatterlist 对应的 page 结构体指针、缓冲区在 page 中的偏移（offset）、缓冲区长度（length）以及总线地址（dma_address）。

代码清单 11.18　scatterlist 结构体

```
1 struct scatterlist
2 {
3   struct page *page;
4   unsigned int offset;
5   dma_addr_t dma_address;
6   unsigned int length;
7 };
```

执行 dma_map_sg()后，通过 sg_dma_address()可返回 scatterlist 对应缓冲区的总线地址，sg_dma_len()可返回 scatterlist 对应缓冲区的长度，这两个函数的原型为：

```
dma_addr_t sg_dma_address(struct scatterlist *sg);
unsigned int sg_dma_len(struct scatterlist *sg);
```

在 DMA 传输结束后，可通过 dma_map_sg()的反函数 dma_unmap_sg()去除 DMA 映射：

```
void dma_unmap_sg(struct device *dev, struct scatterlist *list,
int nents, enum dma_data_direction direction);
```

SG 映射属于流式 DMA 映射，与单一缓冲区情况下的流式 DMA 映射类似，如果设备驱动一定要访问映射情况下的 SG 缓冲区，应该先调用如下函数：

```
void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg,
int nents, enum dma_data_direction direction);
```

访问完后，通过下列函数将所有权返回给设备：

```
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg,
int nents, enum dma_data_direction direction);
```

Linux 系统中可以有一个相对简单的方法预先分配缓冲区，那就是同步“mem=”参数预留内存。例如，对于内存为 64MB 的系统，通过给其传递 mem=62MB 命令行参数可以使得顶部的 2MB 内存被预留出来作为 I/O 内存使用，这 2MB 内存可以被静态映射（11.5

节），也可以被执行 ioremap()。

1．申请和释放 DMA 通道

和中断一样，在使用 DMA 之前，设备驱动程序需首先向系统申请 DMA 通道，申请 DMA 通道的函数如下：

```
int request_dma(unsigned int dmanr, const char * device_id);
```

同样的，设备结构体指针可作为传入 device_id 的最佳参数。

使用完 DMA 通道后，应该利用如下函数释放该通道：

```
void free_dma(unsigned int dmanr);
```

现在可以总结出在 Linux 设备驱动中 DMA 相关代码的流程，如图 11.14 所示。

图 11.14　Linux 中 DMA 使用流程

2．实例：8237 DMA 控制器

最典型的 DMAC 是 8237，PC 的 DMA 子系统建立在 Intel® 8237 控制器之上。8237 控制器包含了 4 个 DMA 通道，每个通道都可以被独立地编程控制而且任何一个通道在任何时候都可以是活动的。这些通道被编号为 0、1、2 和 3。从 PC/AT 开始，IBM 加入了第二个 8237 芯片，然后把加入的通道命名为 4、5、6 和 7。第 1 个 DMAC 以字节为单位传输，第 2 个 DMAC 以 16 比特数据为单位传输。通过一组寄存器和命令可以控制 8237，代码清单 11.19 给出了 8237 的寄存器及命令定义。

代码清单 11.19　8237 DMAC 的寄存器和命令定义

```
/* 8237 DMA 控制器 */
#define IO_DMA1_BASE 0x00    /* 8 位从 DMA，0～3 通道 */
#define IO_DMA2_BASE 0xC0    /* 16 位主 DMA，4～7 通道 */

/* DMA 控制器 1 控制寄存器 */
#define DMA1_CMD_REG     0x08    /* 命令寄存器 (w) */
#define DMA1_STAT_REG        0x08    /* 状态寄存器 (r) */
#define DMA1_REQ_REG      0x09    /* 请求寄存器 (w) */
```

```
9  #define DMA1_MASK_REG        0x0A    /* 单通道屏蔽 (w) */
10 #define DMA1_MODE_REG        0x0B    /* 模式寄存器 (w) */
11 #define DMA1_CLEAR_FF_REG    0x0C    /* 清除 flip-flop (w) */
12 #define DMA1_TEMP_REG      0x0D    /* 临时寄存器 (r) */
13 #define DMA1_RESET_REG       0x0D    /* 主清除 (w) */
14 #define DMA1_CLR_MASK_REG       0x0E    /* 清除屏蔽 */
15 #define DMA1_MASK_ALL_REG       0x0F    /* 所有通道屏蔽 */
16
17 /* DMA 控制器 2 控制寄存器 */
18 #define DMA2_CMD_REG     0xD0    /* 命令寄存器 (w) */
19 …          /*类似于 DMA 控制器 1*/
20
21 /* DMA0～7 通道地址寄存器 */
22 #define DMA_ADDR_0             0x00
23 …
24
25 /* DMA0～7 通道计数寄存器 */
26 #define DMA_CNT_0              0x01
27 …
28
29 /* DMA0～7 通道页寄存器 */
30 #define DMA_PAGE_0             0x87
31 …
32
33 #define DMA_MODE_READ    0x44    /* I/O 到内存 */
34 #define DMA_MODE_WRITE   0x48    /* 内存到 I/O */
35 …
```

代码清单 11.20 给出的一组函数可以从硬件上设置 8237 DMA 控制器，包括使能和禁止 DMA 通道、设置 DMA 的模式、地址和尺寸等。DMA flip-flop 用来控制对 16 位寄存器（由两个 8 位字节组成）的访问，清除时选中低字节，设置时访问高字节。

代码清单 11.20　8237 DMA 硬件设置

```
1  /*使能 DMA 通道*/
2  static __inline__ void enable_dma(unsigned int dmanr)
3  {
4    if (dmanr <= 3)
5      dma_outb(dmanr, DMA1_MASK_REG);
6    else
7      dma_outb(dmanr &3, DMA2_MASK_REG);
8  }
9
10 /*禁止 DMA 通道*/
11 static __inline__ void disable_dma(unsigned int dmanr)
12 {
13   if (dmanr <= 3)
14     dma_outb(dmanr | 4, DMA1_MASK_REG);
15   else
16     dma_outb((dmanr &3) | 4, DMA2_MASK_REG);
17 }
```

```
18
19 /* 设置传输尺寸(通道 0~3 最大 64KB, 通道 5~7 最大 128KB */
20 static __inline__ void set_dma_count(unsigned int dmanr, unsigned int count)
21 {
22   count--;
23   if (dmanr <= 3)
24   {
25     dma_outb(count &0xff, ((dmanr &3) << 1) + 1+IO_DMA1_BASE);
26     dma_outb((count >> 8) &0xff, ((dmanr &3) << 1) + 1+IO_DMA1_BASE);
27   }
28   else
29   {
30     dma_outb((count >> 1) &0xff, ((dmanr &3) << 2) + 2+IO_DMA2_BASE);
31     dma_outb((count >> 9) &0xff, ((dmanr &3) << 2) + 2+IO_DMA2_BASE);
32   }
33 }
34
35 /* 设置传输地址和页位 */
36 static __inline__ void set_dma_addr(unsigned int dmanr, unsigned int a)
37 {
38   set_dma_page(dmanr, a >> 16);
39   if (dmanr <= 3)
40   {
41     dma_outb(a &0xff, ((dmanr &3) << 1) + IO_DMA1_BASE);
42     dma_outb((a >> 8) &0xff, ((dmanr &3) << 1) + IO_DMA1_BASE);
43   }
44   else
45   {
46     dma_outb((a >> 1) &0xff, ((dmanr &3) << 2) + IO_DMA2_BASE);
47     dma_outb((a >> 9) &0xff, ((dmanr &3) << 2) + IO_DMA2_BASE);
48   }
49 }
50
51 /* 设置页寄存器位，当已知 DMA 当前地址寄存器的低 16 位时，连续传输 */
52 static __inline__ void set_dma_page(unsigned int dmanr, char pagenr)
53 {
54   switch (dmanr)
55   {
```

```
56    case 0:
57      dma_outb(pagenr, DMA_PAGE_0);
58      break;
59    case 1:
60      dma_outb(pagenr, DMA_PAGE_1);
61      break;
62    case 2:
63      dma_outb(pagenr, DMA_PAGE_2);
64      break;
65    case 3:
66      dma_outb(pagenr, DMA_PAGE_3);
67      break;
68    case 5:
69      dma_outb(pagenr &0xfe, DMA_PAGE_5);
70      break;
71    case 6:
72      dma_outb(pagenr &0xfe, DMA_PAGE_6);
73      break;
74    case 7:
75      dma_outb(pagenr &0xfe, DMA_PAGE_7);
76      break;
77  }
78 }
79
80 /*清除DMA flip-flop*/
81 static __inline__ void clear_dma_ff(unsigned int dmanr)
82 {
83  if (dmanr <= 3)
84    dma_outb(0, DMA1_CLEAR_FF_REG);
85  else
86    dma_outb(0, DMA2_CLEAR_FF_REG);
87 }
88
89 /*设置某通道的DMA模式*/
90 static __inline__ void set_dma_mode(unsigned int dmanr, char mode)
91 {
92  if (dmanr <= 3)
93    dma_outb(mode | dmanr, DMA1_MODE_REG);
94  else
95    dma_outb(mode | (dmanr &3), DMA2_MODE_REG);
```

```
96 }
```

假设设备 xxx 使用了 DMA，DMA 相关的信息应该被添加到设备结构体内。在模块加载函数或打开函数中，DMA 通道和中断应该被申请，而 DMA 本身也应被初始化。不论是内存到 I/O 的 DMA，还是 I/O 到内存的 DMA，都应使用代码清单 11.19 给出的硬件设置函数，不过命令不一样。中断服务程序完成 DMA 的善后处理，有时候，正是中断事件的到来才触发了一次 DMA 的传输。

内存到 I/O 的 DMA 发送通常由上层触发，而 I/O 到内存的 DMA 通常由外设收到了数据之后的中断触发。代码清单 11.21 给出了外设使用 8237 DMA 控制器进行数据传输的设备驱动典型范例。

代码清单 11.21　外设使用 8237 DMA 驱动范例

```
1   /*xxx 设备结构体*/
2   typedef struct
3   {
4   ...
5   void *dma_buffer; //DMA 缓冲区
6   /*当前 DMA 的相关信息*/
7   struct
8   {
9       unsigned int direction; //方向
10      unsigned int length;    //尺寸
11          void *target;          //目标
12      unsigned long start_time;  //开始时间
13  } current_dma;
14
15  unsigned char       dma;//DMA 通道
16  } xxx_device;
17
18  static int xxx_open(...)
19  {
20  ...
21  /*安装中断服务程序 */
22  if ((retval = request_irq(dev->irq, &xxx_interrupt, 0, dev->name,
dev))) {
23      printk(KERN_ERR "%s: could not allocate IRQ%d\n", dev->name,
dev->irq);
24      return retval;
25  }
26  /*申请 DMA*/
27  if ((retval = request_dma(dev->dma, dev->name))) {
```

```
    free_irq(dev->irq, dev);
    printk(KERN_ERR "%s: could not allocate DMA%d channel\n", ...);
    return retval;
}
/*申请 DMA 缓冲区*/
dev->dma_buffer = (void *) dma_mem_alloc(DMA_BUFFER_SIZE);
if (!dev->dma_buffer) {
    printk(KERN_ERR "%s: could not allocate DMA buffer\n", dev->name);
    free_dma(dev->dma);
    free_irq(dev->irq, dev);
    return -ENOMEM;
}
/*初始化 DMA*/
init_dma();
...
}

/*内存到外设*/
static int mem_to_xxx(const byte *buf,int len)
{
...
dev->current_dma.direction = 1; /*DMA 方向*/
dev->current_dma.start_time = jiffies; /*记录 DMA 开始时间*/

memcpy(dev->dma_buffer, buf, len); /*复制要发送的数据到 DMA 缓冲区*/
target = isa_virt_to_bus(dev->dma_buffer);/*假设 xxx 挂接在 ISA 总线*/

 /*进行一次 DMA 写操作*/
flags=claim_dma_lock();
disable_dma(dev->dma); /*禁止 DMA*/
clear_dma_ff(dev->dma);  /*清除 DMA flip-flop*/
set_dma_mode(dev->dma, 0x48);  /* DMA 内存 -> io */
set_dma_addr(dev->dma, target); /*设置 DMA 地址*/
set_dma_count(dev->dma, len);  /*设置 DMA 长度*/
outb_control(dev->x_ctrl | DMAE | TCEN, dev);/*让设备接收 DMA*/
enable_dma(dev->dma);    /*使能 DMA*/
release_dma_lock(flags);

```

```
  printk(KERN_DEBUG "%s: DMA transfer started\n", dev->name);
 ...
}

/*外设到内存*/
static void xxx_to_mem(const char *buf, int len,char * target)
{
  ...
  /*记录 DMA 信息*/
  dev->current_dma.target = target;
  dev->current_dma.direction = 0;
  dev->current_dma.start_time = jiffies;
  dev->current_dma.length = len;

  /*进行一次 DMA 读操作*/
  outb_control(dev->x_ctrl | DIR | TCEN | DMAE, dev);
  flags = claim_dma_lock();
  disable_dma(dev->dma);
  clear_dma_ff(dev->dma);
  set_dma_mode(dev->dma, 0x04); /* I/O->mem */
  set_dma_addr(dev->dma, isa_virt_to_bus(target));
  set_dma_count(dev->dma, len);
  enable_dma(dev->dma);
  release_dma_lock(flags);
  ...
}

/*设备中断处理*/
 static irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs *reg_ptr)
{
  ...
  do
  {
    /* DMA 传输完成? */
    if (int_type==DMA_DONE)
    {
      outb_control(dev->x_ctrl &~(DMAE | TCEN | DIR), dev);
      if (dev->current_dma.direction)
      {
```

```
      /*内存->I/O*/
      ...
    }
    else
    /*I/O->内存*/
    {
      memcpy(dev->current_dma.target, dev->dma_buffer, dev
        ->current_dma.length);
    }
  }
   else if(int_type==RECV_DATA) /*收到数据*/
   {
       xxx_to_mem(...);/*通过 DMA 读接收到的数据到内存*/
   }
   ...
 }
 ...
}
```

11.7 总结

外设可处于 CPU 的内存空间和 I/O 空间，除 X86 外，嵌入式处理器一般只存在内存空间。在 Linux 系统中，为 I/O 内存和 I/O 端口的访问提高了一套统一的方法，访问流程一般为“申请资源→映射→访问→去映射→释放资源”。

对于有 MMU 的处理器而言，Linux 系统的内部布局比较复杂，可直接映射的物理内存称为常规内存，超出部分为高端内存。kmalloc()和__get_free_pages()申请的内存在物理上连续，而 vmalloc()申请的内存在物理上不连续。

DMA 操作可能导致 Cache 的不一致问题，因此，对于 DMA 缓冲，应该使用 dma_alloc_coherent()等方法申请。在 DMA 操作中涉及总线地址、物理地址和虚拟地址等概念，区分这 3 类地址非常重要。Linux 内核中对 DMA 通道的申请和释放采用了和中断类似的方法。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

· 招生简章：http://www.embedu.org/courses/index.htm

嵌入式学院 华清远见旗下品牌

第 12 章　Linux 字符设备驱动综合实例

本章将分析 5 个典型的字符设备驱动，在这些驱动中，将灵活地运用到前面各章所讲解的内容。

12.1 节讲解按键的设备驱动，加深读者对字符设备驱动架构、阻塞与非阻塞、中断、定时器等相关知识的理解。

12.2 节讲解触摸屏的设备驱动，触摸屏的设备驱动比按键的设备驱动稍微复杂一些，但是很类似。

12.3 节讲解 TI 的 DSP 提供给通用 CPU 的 HPI（主机并行接口）的设备驱动，硬件结构为 ARM+DSP，ARM 的总线连接 DSP 的 HPI 接口。

12.4 节讲解通用 NVRAM 的设备驱动，并会引入一个新的概念，即 miscdevice（混杂设备）。

12.5 节讲解看门狗的设备驱动，它也被归入 miscdevice，这一节还会引入两个新的概念，即 platform_device（平台设备）和 platform_driver（平台驱动）。

NVRAM 和看门狗的设备驱动与普通字符设备驱动有细微的差别。

华清远见

12.1 按键的设备驱动

12.1.1 按键的硬件原理

在嵌入式系统中，按键的硬件原理比较简单，通过一个上拉电阻将处理器的外部中断（或 GPIO）引脚拉高，电阻的另一端连接按钮并接地即可实现。如图 12.1 所示，当按钮被按下时，EINT10、EIN13、EINT14、EINT15 上将产生低电平，这个低电平将中断 CPU（图中的 CPU 为 S3C2410），CPU 可以依据中断判断按键被按下。

但是，仅仅依据中断被产生就认定有一次按键行为是很不准确的，所有按键、触摸屏等机械设备都存在一个固有的问题，那就是“抖动”，按键从最初接通到稳定接通要经过数毫秒，其间可能发生多次“接通—断开”的过程。如果不消除“抖动”的影响，一次按键可能被理解为多次按键。

消除按键抖动影响的方法是：在判断有键按下后，进行软件延时（如 20ms，在延时过程中要屏蔽对应中断），再判断键盘状态，如果仍处于按键按下状态，则可以断定该按键被按下，流程如图 12.2（a）所示。如果按键对应的引脚本身不具备中断输入功能，则可以改为完全查询方式，流程如图 12.2（b）所示。

图 12.1　按键的硬件原理

图 12.2　确认按键的流程

12.1.2　按键驱动中的数据结构

设备驱动中主要要设计的数据结构是设备结构体，按键的设备结构体中应包含一个缓冲区，因为多次按键可能无法被及时处理，可以用该缓冲区缓存按键。此外，在按键设备结构体中，还包含按键状态标志和一个实现过程中要借助的等待队列、cdev 结构体。为了实现软件延时，定时器也是必要的，但可以不包含在设备结构体中。代码清单 12.1 给出了按键设备结构体及定时器。

代码清单 12.1　按键驱动的设备结构体、定时器

```
1  #define MAX_KEY_BUF 16 //按键缓冲区大小
2  typedef unsigned char KEY_RET;
3  //设备结构体:
4  typedef struct
5  {
6    unsigned int keyStatus[KEY_NUM]; //4 个按键的按键状态
7    KEY_RET buf[MAX_KEY_BUF]; //按键缓冲区
8    unsigned int head, tail; //按键缓冲区头和尾
9    wait_queue_head_t wq; //等待队列
10   struct cdev cdev;     //cdev 结构体
10 } KEY_DEV;
11 static struct timer_list key_timer[KEY_NUM];//4 个按键去抖定时器
```

在按键设备驱动中，可用一个结构体记录每个按键所对应的中断/GPIO 引脚及键值，如代码清单 12.2 所示。

代码清单 12.2　按键硬件资源、键值信息结构体

```
1  static struct key_info
2  {
3    int irq_no;     //中断号
4    unsigned int gpio_port; //GPIO 端口
5    int key_no;    //键值
6  } key_info_tab[4] =
7  {
8     /*按键所使用的 CPU 资源*/
9    {
```

```
    IRQ_EINT10, GPIO_G2, 1
  }
  ,
  {
    IRQ_EINT13, GPIO_G5, 2
  }
  ,
  {
    IRQ_EINT14, GPIO_G6, 3
  }
  ,
  {
    IRQ_EINT15, GPIO_G7, 4
  }
  ,
};
```

按键设备驱动的文件操作结构体如代码清单 12.3 所示，主要实现了打开、释放和读函数，因为按键只是一个输入设备，所以不存在写函数。

代码清单 12.3　按键设备驱动文件操作结构体

```
static struct file_operations s3c2410_key_fops =
{
  owner: THIS_MODULE,
  open: s3c2410_key_open,  //启动设备
  release: s3c2410_key_release,  //关闭设备
  read: s3c2410_key_read,  //读取按键的键值
};
```

12.1.3　按键驱动的模块加载和卸载函数

按键设备作为一种字符设备，在其模块加载和卸载函数中分别包含了设备号申请和释放、cdev 的添加和删除行为，在模块加载函数中，还需申请中断、初始化定时器和等待队列等，模块卸载函数完成相反的行为，代码清单 12.4 和 12.5 分别给出了按键设备驱动的模块加载和卸载函数，代码清单 12.6 和 12.7 分别给出了模块加载和卸载所调用的申请和释放 4 个中断的函数。

代码清单 12.4　按键设备驱动的模块加载函数

```
static int __init s3c2410_key_init(void)
{
  ...//申请设备号，添加 cdev

  request_irqs(); //注册中断函数
  keydev.head = keydev.tail = 0; //初始化结构体
  for (i = 0; i < KEY_NUM; i++)
    keydev.keyStatus[i] = KEYSTATUS_UP;
  init_waitqueue_head(&(keydev.wq)); //等待队列

  //初始化定时器，实现软件的去抖动
  for (i = 0; i < KEY_NUM; i++)
    setup_timer(&key_timer[i], key_timer_handler, i);
  //把按键的序号作为传入定时器处理函数的参数
}
```

代码清单 12.5　按键设备驱动的模块卸载函数

```
1 static void __exit s3c2410_key_exit(void)
2 {
3   free_irqs(); //注销中断
4   ...//释放设备号，删除 cdev
5 }
```

代码清单 12.6　按键设备驱动的中断申请函数

```
1  /*申请系统中断，中断方式为下降沿触发*/
2  static int request_irqs(void)
3  {
4    struct key_info *k;
5    int i;
6    for (i = 0; i < sizeof(key_info_tab) / sizeof(key_info_tab[1]); i++)
7    {
8      k = key_info_tab + i;
9      set_external_irq(k->irq_no, EXT_LOWLEVEL, GPIO_PULLUP_DIS);
10                  //设置低电平触发
11         if  (request_irq(k->irq_no,  &buttons_irq,  SA_INTERRUPT,
DEVICE_NAME,
12       i))  //申请中断，将按键序号作为参数传入中断服务程序
13     {
14       return  - 1;
15     }
16   }
17   return 0;
18 }
```

代码清单 12.7　按键设备驱动的中断释放函数

```
1  /*释放中断*/
2  static void free_irqs(void)
3  {
4    struct key_info *k;
5    int i;
6    for (i = 0; i < sizeof(key_info_tab) / sizeof(key_info_tab[1]); i++)
7    {
8      k = key_info_tab + i;
9      free_irq(k->irq_no, buttons_irq); //释放中断
10   }
11 }
```

12.1.4　按键设备驱动中断、定时器处理程序

在键被按下后，将发生中断，在中断处理程序中，应该关闭中断进入查询模式，延迟 20ms 以实现去抖动，如代码清单 12.8 所示，这个中断处理过程只包含顶半部，无底半部。

代码清单 12.8　按键设备驱动的中断处理程序

```
1 static void s3c2410_eint_key(int irq, void *dev_id, struct pt_regs
*reg)
2 {
3   int key = dev_id;
4   disable_irq(key_info_tab[key].irq_no); //关中断，转入查询模式
5
6   keydev.keyStatus[key] = KEYSTATUS_DOWNX;//状态为按下
7   key_timer[key].expires == jiffies + KEY_TIMER_DELAY1;//延迟
8   add_timer(&key_timer[key]); //启动定时器
9 }
```

在定时器处理程序中，查询按键是否仍然被按下，如果是被按下的状态，则将该

按键记录入缓冲区。同时启动新的定时器延迟，延迟一个相对于去抖更长的时间（如 100ms），每次定时器到期后，查询按键是否仍然处于按下状态，如果是，则重新启用新的 100ms 延迟；若查询到已经没有按下，则认定键已抬起，这个时候应该开启对应按键的中断，等待新的按键。每次记录新的键值时，应唤醒等待队列。定时器处理流程如图 12.3 所示，代码清单如 12.9 所示。

代码清单 12.9　按键设备驱动的定时器处理函数

```
static void key_timer_handler(unsigned long data)
{
  int key = data;
  if (ISKEY_DOWN(key))
  {
    if (keydev.keyStatus[key] == KEYSTATUS_DOWNX)
    //从中断进入
    {
      keydev.keyStatus[key] = KEYSTATUS_DOWN;
      key_timer[key].expires == jiffies + KEY_TIMER_DELAY; //延迟
      keyEvent();   //记录键值，唤醒等待队列
      add_timer(&key_timer[key]);
    }
    else
    {
      key_timer[key].expires == jiffies + KEY_TIMER_DELAY; //延迟
      add_timer(&key_timer[key]);
    }
  }
  else      //键已抬起
  {
    keydev.keyStatus[key] = KEYSTATUS_UP;
    enable_irq(key_info_tab[key].irq_no);
  }
}
```

图 12.3　定时器处理函数流程

12.1.5　按键设备驱动的打开、释放函数

按键设备驱动的打开和释放函数比较简单，主要是设置 keydev.head、keydev.tail 和按键事件函数指针 keyEvent 的值，如代码清单 12.10 所示。

代码清单 12.10　按键设备驱动的打开、释放函数

```
static int s3c2410_key_open(struct inode *inode, struct file *filp)
{
  keydev.head = keydev.tail = 0; //清空按键动作缓冲区
  keyEvent = keyEvent_raw; //函数指针指向按键处理函数 keyEvent_raw
  return 0;
}

static int s3c2410_key_release(struct inode *inode, struct file *filp)
{
  keyEvent = keyEvent_dummy; //函数指针指向空函数
  return 0;
}
```

12.1.6　按键设备驱动读函数

代码清单 12.11 给出了按键设备驱动的读函数，按键设备驱动的读函数主要提供对按键设备结构体中缓冲区的读并复制到用户空间。当 keydev.head ! = keydev.tail 时，意味着缓冲区有数据，使用 copy_to_user()拷贝到用户空间，否则，根据用户空间是阻塞读还是非阻塞读，分为如下两种情况。

- 若采用非阻塞读，则因为没有按键缓存，直接返回- EAGAIN；
- 若采用阻塞读，则在 keydev.wq 等待队列上睡眠，直到有按键被记录入缓冲区后被唤醒。

代码清单 12.11　按键设备驱动的读函数

```
static ssize_t s3c2410_key_read(struct file *filp, char *buf, ssize_t count,
  loff_t*ppos)
{
  retry: if (keydev.head != keydev.tail)
  //当前循环队列中有数据
  {
    key_ret = keyRead(); //读取按键
    copy_to_user(..); //把数据从内核空间传送到用户空间
  }
  else
  {
    if (filp->f_flags &O_NONBLOCK)
    //若用户采用非阻塞方式读取
    {
      return  - EAGAIN;
    }
    interruptible_sleep_on(&(keydev.wq));
      //用户采用阻塞方式读取，调用该函数使进程睡眠
    goto retry;
  }
  return 0;
}
```

最后，解释一下代码清单 12.9 第 11 行的 keyEvent()函数和代码清单 12.11 的 keyRead()函数。在设备驱动的打开函数中，keyEvent 被赋值为 keyEvent_raw，这个函数完成记录键值，并使用 wait_up_interrupt(&(keydev.wq))语句唤醒 s3c2410_key_read()第 17 行所期待的等待队列。而 keyRead()函数则直接从按键缓冲区中读取键值。

12.2 触摸屏的设备驱动

12.2.1 触摸屏的硬件原理

按照触摸屏的工作原理和传输信息的介质，我们把触摸屏分为 4 种：电阻式、电容感应式、红外线式以及表面声波式。

电阻式触摸屏利用压力感应进行控制，包含上下叠合的两个透明层，通常还要用一种弹性材料来将两层隔开。在触摸某点时，两层会在此点接通。四线和八线触摸屏由两层具有相同表面电阻的透明阻性材料组成，五线和七线触摸屏由一个阻性层和一个导电层组成。

所有的电阻式触摸屏都采用分压器原理来产生代表 X 坐标和 Y 坐标的电压。如图 12.4 所示，分压器是通过将两个电阻进行串联来实现的。电阻 R1 连接正参考电压 V_{REF}，电阻 R2 接地。两个电阻连接点处的电压测量值与 R2 的阻值成正比。

为了在电阻式触摸屏上的特定方向测量一个坐标，需要对一个阻性层进行偏置：将它的一边接 V_{REF}，另一边接地。同时，将未偏置的那一层连接到一个 ADC 的高阻抗输入端。当触摸屏上的压力足够大，两层之间发生接触时，电阻性表面被分隔为两个电阻。它们的阻值与触摸点到偏置边缘的距离成正比。触摸点与接地边之间的电阻相当于分压器中下面的那个电阻。因此，在未偏置层上测得的电压与触摸点到接地边之间的距离成正比。

四线触摸屏包含两个阻性层。其中一层在屏幕的左右边缘各有一条垂直总线，另一层在屏幕的底部和顶部各有一条水平总线，如图 12.5 所示。为了在 X 轴方向进行测量，将左侧总线偏置为 0V，右侧总线偏置为 V_{REF}。将顶部或底部总线连接到 ADC，当顶层和底层相接触时即可作一次测量。为了在 Y 轴方向进行测量，将顶部总线偏置为 V_{REF}，底部总线偏置为 0V。将 ADC 输入端接左侧总线或右侧总线，当顶层与底层相接触时即可对电压进行测量。

华清远见

图 12.4　电阻触摸屏分压

图 12.5　四线电阻式触摸屏

S3C2410 接 4 线电阻式触摸屏的电路原理如图 12.6 所示。S3C2410 提供了 nYMON、YMON、nXPON 和 XMON 直接作为触摸屏的控制信号，它通过连接 FDC6321 场效应管触摸屏驱动器控制触摸屏。输入信号在经过阻容式低通滤器滤除坐标信号噪声后被接入 S3C2410 内集成的 ADC（模数转换器）的模拟信号输入通道 AIN5、AIN7。

华清远见

图 12.6　S3C2410 连接 4 线电阻式触摸屏

S3C2410 内置了一个 8 信道的 10 位 ADC，该 ADC 能以 500KS/S 的采样速率将外部的模拟信号转换为 10 位分辨率的数字量。因此，ADC 能与触摸屏控制器协同工作，完成对触摸屏绝对地址的测量。

S3C2410 的 ADC 和触摸屏接口可工作于 5 种模式，分别如下。

1．普通转换模式（Normal Converson Mode）

普通转换模式（AUTO_PST = 0，XY_PST = 0）用来进行一般的 ADC 转换，例如通过 ADC 测量电池电压等。

2．独立 *X*/*Y* 位置转换模式（Separate *X*/*Y* Position Conversion Mode）

独立 *X*/*Y* 轴坐标转换模式其实包含了 *X* 轴模式和 *Y* 轴模式。为获得 *X*、*Y* 坐标，需首先进行 *X* 轴的坐标转换（AUTO_PST = 0，XY_PST = 1），*X* 轴的转换资料会写到 ADCDAT0 寄存器的 XPDAT 中，等待转换完成后，触摸屏控制器会产生 INT_ADC 中断。然后，进行 *Y* 轴的坐标转换（AUTO_PST = 0，XY_PST = 2），*Y* 轴的转换资料会写到 ADCDAT1 寄存器的 YPDAT 中，等待转换完成后，触摸屏控制器也会产生 INT_ADC 中断。

3．自动（连续）*X*/*Y* 位置转换模式（Auto *X*/*Y* Position Conversion Mode）

自动（连续）*X*/*Y* 位置转换模式（AUTO_PST = 1，XY_PST = 0）运行方式是触摸屏控制自动转换 *X* 位置和 *Y* 位置。触摸屏控制器在 ADCDAT0 的 XPDATA 位写入 *X* 测定数据，在 ADCDAT1 的 YPADATA 位写入 *Y* 测定数据。自动（连续）位置转换后，触摸屏控制器产生 INT_ADC 中断。

4．等待中断模式（Wait for Interrupt Mode）

当触摸屏控制器等待中断模式时，它等待触摸屏触点信号的到来。当触点信号到来时，控制器产生 INT_TC 中断信号。然后，*X* 位置和 *Y* 位置能被适当地转换模式（独立 *X*/*Y* 位置转换模式或自动 *X*/*Y* 位置转换模式）读取到。

5．待机模式（Standby Mode）

当 ADCCON 寄存器的 STDBM 位置 1 时，待机模式被激活。在这种模式下，A/D 转换动作被禁止，ADCDAT0 的 XPDATA 位和 ADXDATA1 的 YPDAT 保留以前被转换的数据。

12.2.2　触摸屏设备驱动中数据结构

触摸屏设备结构体的成员与按键设备结构体的成员类似，也包含一个缓冲区，同时包括自旋锁、等待队列和 fasync_struct 指针，如代码清单 12.12 所示。

代码清单 12.12　触摸屏设备结构体

```
typedef struct
{
  unsigned int penStatus; /* PEN_UP, PEN_DOWN, PEN_SAMPLE */
  TS_RET buf[MAX_TS_BUF]; /* 缓冲区 */
  unsigned int head, tail; /* 缓冲区头和尾 */
  wait_queue_head_t wq; /*等待队列*/
  spinlock_t lock;
  #ifdef USE_ASYNC
    struct fasync_struct *aq;
  #endif
  struct cdev cdev;
} TS_DEV;
```

触摸屏结构体中包含的 TS_RET 值的类型定义如代码清单 12.13 所示，包含 *X*、*Y* 坐标和状态（PEN_DOWN、PEN_UP）等信息，这个信息会在用户读取触摸信息时复制到用户空间。

代码清单 12.13　TS_RET 结构体

```
typedef struct
```

```
2 {
3   unsigned short pressure;//PEN_DOWN、PEN_UP
4   unsigned short x;//x 坐标
5   unsigned short y;//y 坐标
6   unsigned short pad;
7 } TS_RET;
```

在触摸屏设备驱动中，将实现 open()、release()、read()、fasync()和 poll()函数，因此，其文件操作结构体定义如代码清单 12.14 所示。

代码清单 12.14　触摸屏驱动文件操作结构体

```
1  static struct file_operations s3c2410_fops =
2  {
3    owner: THIS_MODULE,
4    open: s3c2410_ts_open, //打开
5    read: s3c2410_ts_read, //读坐标
6    release:
7      s3c2410_ts_release,
8    #ifdef USE_ASYNC
9      fasync: s3c2410_ts_fasync, // fasync()函数
10   #endif
11   poll: s3c2410_ts_poll,//轮询
12 };
```

12.2.3　触摸屏驱动中的硬件控制

代码清单 12.15 中的一组宏用于控制触摸屏和 ADC 进入不同的工作模式，如等待中断、*X*/*Y* 位置转换等。

代码清单 12.15　触摸屏和 ADC 硬件控制

```
1  #define wait_down_int()  { ADCTSC = DOWN_INT | XP_PULL_UP_EN |\
2                XP_AIN | XM_HIZ | YP_AIN | YM_GND | \
3                XP_PST(WAIT_INT_MODE); }
4  #define wait_up_int()    { ADCTSC = UP_INT | XP_PULL_UP_EN | XP_AIN
|\
5                XM_HIZ |YP_AIN | YM_GND | XP_PST(WAIT_INT_MODE); }
6  #define mode_x_axis()    { ADCTSC = XP_EXTVLT | XM_GND | YP_AIN  \
7                | YM_HIZ |XP_PULL_UP_DIS | XP_PST(X_AXIS_MODE); }
8  #define mode_x_axis_n()  { ADCTSC = XP_EXTVLT | XM_GND | YP_AIN | \
9                 YM_HIZ |XP_PULL_UP_DIS | XP_PST(NOP_MODE); }
10 #define mode_y_axis()    { ADCTSC = XP_AIN | XM_HIZ | YP_EXTVLT  \
11               | YM_GND |XP_PULL_UP_DIS | XP_PST(Y_AXIS_MODE); }
12 #define start_adc_x()    { ADCCON = PRESCALE_EN | PRSCVL(49) | \
13               ADC_INPUT(ADC_IN5) | ADC_START_BY_RD_EN | \
14               ADC_NORMAL_MODE; \
15             ADCDAT0; }
16 #define start_adc_y()    { ADCCON = PRESCALE_EN | PRSCVL(49) | \
17               ADC_INPUT(ADC_IN7) | ADC_START_BY_RD_EN | \
18               ADC_NORMAL_MODE; \
19             ADCDAT1; }
20 #define disable_ts_adc(){ ADCCON &= ~(ADCCON_READ_START); }
```

12.2.4 触摸屏驱动模块加载和卸载函数

在触摸屏设备驱动的模块加载函数中，要完成申请设备号、添加 cdev、申请中断、设置触摸屏控制引脚（YPON、YMON、XPON、XMON）等多项工作，如代码清单 12.16 所示。

代码清单 12.16 触摸屏设备驱动的模块加载函数

```
1  static int __init s3c2410_ts_init(void)
2  {
3    int ret;
4    tsEvent = tsEvent_dummy;
5    ...//申请设备号，添加 cdev
6
7    /* 设置 XP、YM、YP 和 YM 对应引脚 */
8    set_gpio_ctrl(GPIO_YPON);
9    set_gpio_ctrl(GPIO_YMON);
10   set_gpio_ctrl(GPIO_XPON);
11   set_gpio_ctrl(GPIO_XMON);
12
13   /* 使能触摸屏中断 */
14   ret = request_irq(IRQ_ADC_DONE, s3c2410_isr_adc,
15     SA_INTERRUPT, DEVICE_NAME,s3c2410_isr_adc);
16   if (ret)
17     goto adc_failed;
18   ret = request_irq(IRQ_TC, s3c2410_isr_tc, SA_INTERRUPT,
19     DEVICE_NAME,s3c2410_isr_tc);
20   if (ret)
21     goto tc_failed;
22
23   /*置于等待触点中断模式*/
24   wait_down_int();
25
26   printk(DEVICE_NAME " initialized\n");
27
28   return 0;
29   tc_failed:
30   free_irq(IRQ_ADC_DONE, s3c2410_isr_adc);
31   adc_failed:
32   return ret;
33 }
```

在触摸屏设备驱动的模块卸载函数中，要完成释放设备号、删除 cdev、释放中断等工作，如代码清单 12.17 所示。

代码清单 12.17 触摸屏设备驱动模块卸载函数

```
1 static void __exit s3c2410_ts_exit(void)
2 {
3   ...//释放设备号，删除 cdev
4   free_irq(IRQ_ADC_DONE, s3c2410_isr_adc);
5   free_irq(IRQ_TC, s3c2410_isr_tc);
6 }
```

12.2.5 触摸屏驱动中断、定时器处理程序

由 12.2.1 小节对触摸屏和 ADC 模式的分析，可知触摸屏驱动中会产生两类中断，一类是触点中断（INT-TC），一类是 *X*/*Y* 位置转换中断（INT-ADC）。在前一类中断发生后，若之前处于 PEN_UP 状态，则应该启动 *X*/*Y* 位置转换。另外，将抬起中断也放

在 INT-TC 处理程序中，它会调用 tsEvent()完成等待队列和信号的释放，如代码清单 12.18 所示。

代码清单 12.18　触摸屏设备驱动的触点/抬起中断处理程序

```
1  static void s3c2410_isr_tc(int irq, void *dev_id, struct pt_regs *reg)
2  {
3    spin_lock_irq(&(tsdev.lock));
4    if (tsdev.penStatus == PEN_UP)
5    {
6      start_ts_adc(); //开始 X/Y 位置转换
7    }
8    else
9    {
10     tsdev.penStatus = PEN_UP;
11     DPRINTK("PEN UP: x: %08d, y: %08d\n", x, y);
12     wait_down_int();//置于等待触点中断模式
13     tsEvent();
14   }
15   spin_unlock_irq(&(tsdev.lock));
16 }
```

当 *X*/*Y* 位置转换中断发生后，应读取 *X*、*Y* 的坐标值，填入缓冲区，如代码清单 12.19 所示。

代码清单 12.19　触摸屏设备驱动 *X*/*Y* 位置转换中断处理程序

```
1  static void s3c2410_isr_adc(int irq, void *dev_id, struct pt_regs *reg)
2  {
3    spin_lock_irq(&(tsdev.lock));
4    if (tsdev.penStatus == PEN_UP)
5      s3c2410_get_XY(); //读取坐标
6    #ifdef HOOK_FOR_DRAG
7      else
8        s3c2410_get_XY();
9    #endif
10   spin_unlock_irq(&(tsdev.lock));
11 }
```

上述程序中调用的 s3c2410_get_XY()用于获得 *X*、*Y* 坐标，它使用代码清单 12.15 的硬件操作宏实现，如代码清单 12.20 所示。

代码清单 12.20　触摸屏设备驱动中获得 *X*、*Y* 坐标

```
1  static inline void s3c2410_get_XY(void)
2  {
```

```
  if (adc_state == 0)
  {
    adc_state = 1;
    disable_ts_adc();   //禁止 INT-ADC
    y = (ADCDAT0 &0x3ff); //读取坐标值
    mode_y_axis();
    start_adc_y();   //开始 y 位置转换
  }
  else if (adc_state == 1)
  {
    adc_state = 0;
    disable_ts_adc(); //禁止 INT-ADC
    x = (ADCDAT1 &0x3ff);  //读取坐标值
    tsdev.penStatus = PEN_DOWN;
    DPRINTK("PEN DOWN: x: %08d, y: %08d\n", x, y);
    wait_up_int();   //置于等待抬起中断模式
    tsEvent();
  }
}
```

代码清单 12.18、12.20 中调用的 tsEvent 最终为 tsEvent_raw()，这个函数很关键，当处于 PEN_DOWN 状态时调用该函数，它会完成缓冲区的填充、等待队列的唤醒以及异步通知信号的释放；否则（处于 PEN_UP 状态），将缓冲区头清 0，也唤醒等待队列并释放信号，如代码清单 12.21 所示。

代码清单 12.21　触摸屏设备驱动的 tsEvent_raw()函数

```
static void tsEvent_raw(void)
{
  if (tsdev.penStatus == PEN_DOWN)
  {
    /*填充缓冲区*/
    BUF_HEAD.x = x;
    BUF_HEAD.y = y;
    BUF_HEAD.pressure = PEN_DOWN;

    #ifdef HOOK_FOR_DRAG
      ts_timer.expires = jiffies + TS_TIMER_DELAY;
      add_timer(&ts_timer);//启动定时器
    #endif
  }
  else
  {
    #ifdef HOOK_FOR_DRAG
      del_timer(&ts_timer);
    #endif

    /*填充缓冲区*/
    BUF_HEAD.x = 0;
    BUF_HEAD.y = 0;
    BUF_HEAD.pressure = PEN_UP;
  }

  tsdev.head = INCBUF(tsdev.head, MAX_TS_BUF);
  wake_up_interruptible(&(tsdev.wq));  //唤醒等待队列

```

```
#ifdef USE_ASYNC
  if (tsdev.aq)
    kill_fasync(&(tsdev.aq), SIGIO, POLL_IN);//异步通知
#endif
}
```

在包含了对拖动轨迹支持的情况下，定时器会被启用，周期为 10ms，在每次定时器处理函数被引发时，调用 start_ts_adc()开始 *X*/*Y* 位置转换过程，如代码清单 12.22 所示。

华清远见

代码清单 12.22　触摸屏设备驱动的定时器处理函数

```
1  #ifdef HOOK_FOR_DRAG
2    static void ts_timer_handler(unsigned long data)
3    {
4      spin_lock_irq(&(tsdev.lock));
5      if (tsdev.penStatus == PEN_DOWN)
6      {
7        start_ts_adc();  //开始 X/Y 位置转换
8      }
9      spin_unlock_irq(&(tsdev.lock));
10   }
11 #endif
```

12.2.6　触摸屏设备驱动的打开、释放函数

在触摸屏设备驱动的打开函数中，应初始化缓冲区、penStatus 和定期器、等待队列及 tsEvent 时间处理函数指针，如代码清单 12.23 所示。

代码清单 12.23　触摸屏设备驱动的打开函数

```
1  static int s3c2410_ts_open(struct inode *inode, struct file *filp)
2  {
3   tsdev.head = tsdev.tail = 0;
4   tsdev.penStatus = PEN_UP;//初始化触摸屏状态为 PEN_UP
5  #ifdef HOOK_FOR_DRAG //如果定义了拖动钩子函数
6   init_timer(&ts_timer);//初始化定时器
7   ts_timer.function = ts_timer_handler;
8  #endif
9   tsEvent = tsEvent_raw;
10  init_waitqueue_head(&(tsdev.wq));//初始化等待队列
11
12  return 0;
13 }
```

触摸屏设备驱动的释放函数非常简单，删除为用于拖动轨迹所使用的定时器即可，如代码清单 12.24 所示。

代码清单 12.24　触摸屏设备驱动的释放函数

```
1 static int s3c2410_ts_release(struct inode *inode, struct file *filp)
2 {
3   #ifdef HOOK_FOR_DRAG
4     del_timer(&ts_timer);//删除定时器
5   #endif
6   return 0;
7 }
```

12.2.7　触摸屏设备驱动的读函数

触摸屏设备驱动的读函数实现缓冲区中信息向用户空间的复制，当缓冲区有内容时，直接复制；否则，如果用户阻塞访问触摸屏，则进程在等待队列上睡眠，否则，立即返回-EAGAIN，如代码清单 12.25 所示。

代码清单 12.25　触摸屏设备驱动的读函数

```
1  static ssize_t s3c2410_ts_read(struct file *filp, char *buffer, size_t count,
2    loff_t *ppos)
3  {
4    TS_RET ts_ret;
5
6    retry:
7    if (tsdev.head != tsdev.tail)  //缓冲区有信息
8    {
9      int count;
10     count = tsRead(&ts_ret);
11     if (count)
12       copy_to_user(buffer, (char*) &ts_ret, count);//复制到用户空间
13     return count;
14   }
15   else
16   {
17     if (filp->f_flags &O_NONBLOCK)    //非阻塞读
18       return  - EAGAIN;
19     interruptible_sleep_on(&(tsdev.wq));  //在等待队列上睡眠
20     if (signal_pending(current))
21       return  - ERESTARTSYS;
22     goto retry;
23   }
24
25   return sizeof(TS_RET);
26 }
```

12.2.8　触摸屏设备驱动的轮询与异步通知

在触摸屏设备驱动中，通过 s3c2410_ts_poll()函数实现了轮询接口，这个函数的实现非常简单。它将等待队列添加到 poll_table，当缓冲区有数据时，返回资源可读取标志，否则返回 0，如代码清单 12.26 所示。

代码清单 12.26　触摸屏设备驱动的 poll()函数

```
1  static unsigned int s3c2410_ts_poll(struct file *filp, struct poll_table_struct *wait)
2 {
3   poll_wait(filp, &(tsdev.wq), wait);//添加等待队列到 poll_table
4   return (tsdev.head == tsdev.tail) ? 0 : (POLLIN | POLLRDNORM);
5 }
```

而为了实现触摸屏设备驱动对应用程序的异步通知，设备驱动中要实现 s3c2410_ts_fasync()函数，这个函数与第 9 章给出的模板完全一样，如代码清单 12.27 所示。

代码清单 12.27　触摸屏设备驱动的 fasync()函数

```
1 #ifdef USE_ASYNC
2 static int s3c2410_ts_fasync(int fd, struct file *filp, int mode)
```

```
{
  return fasync_helper(fd, filp, mode, &(tsdev.aq));
}
#endif
```

华清远见

12.2.9 Linux 输入子系统

12.1 及 12.2.1～12.2.8 节分别讲解按键与触摸屏的设备驱动，实际上，在 Linux 系统中，一种更值得推荐的实现这类设备驱动的方法是利用 input 子系统。

Linux 系统提供了 input 子系统，按键、触摸屏、键盘、鼠标等输入都可以利用 input 接口函数来实现设备驱动，因此，12.1～12.2 节的按键和触摸屏设备驱动都可以作为 input 设备驱动而实现。

在 Linux 内核中，input 设备用 input_dev 结构体描述，使用 input 子系统实现输入设备驱动的时候，驱动的核心工作是向系统报告按键、触摸屏、键盘、鼠标等输入事件（event，通过 input_event 结构体描述），不再需要关心文件操作接口，因为 input 子系统已经完成了文件操作接口。驱动报告的事件经过 InputCore 和 Eventhandler 最终到达用户空间。

通过 input 子系统，具体的输入设备驱动只需要完成如下工作。

- 在模块加载函数中告知 input 子系统它可以报告的事件。

设备驱动通过 set_bit()告诉 input 子系统它支持哪些事件，如下所示：

```
set_bit(EV_KEY, button_dev.evbit);
```

- 在模块加载函数中注册输入设备。

注册输入设备的函数为：

```
int input_register_device(struct input_dev *dev);
```

- 在键被按下/抬起、触摸屏被触摸/抬起/移动、鼠标被移动/单击/抬起时通过 input_ report_xxx()报告发生的事件及对应的键值/坐标等状态。

主要的事件类型包括 EV_KEY（按键事件）、EV_REL（相对值，如光标移动，报告的是相对最后一次位置的偏移）和 EV_ABS（绝对值，如触摸屏和操纵杆，它们工作在绝对坐标系统）。

用于报告 EV_KEY、EV_REL 和 EV_ABS 事件的函数分别为：

```
void input_report_key(struct input_dev *dev, unsigned int code, int value);
void input_report_rel(struct input_dev *dev, unsigned int code, int value);
void input_report_abs(struct input_dev *dev, unsigned int code, int value);
```

input_sync()用于事件同步，它告知事件的接收者驱动已经发出了一个完整的报告。

例如，在触摸屏设备驱动中，一次坐标及按下状态的整个报告过程如下：

```
input_report_abs(input_dev, ABS_X, x);  //X 坐标
input_report_abs(input_dev, ABS_Y, y);   //Y 坐标
input_report_abs(input_dev, ABS_PRESSURE, pres); //压力
input_sync(input_dev);  //同步
```

- 在模块卸载函数中注销输入设备。

注销输入设备的函数为：

```
void input_unregister_device(struct input_dev *dev);
```

代码清单 12.28 给出了一个最简单的使用 input 接口实现按键设备驱动的范例，它在中断服务程序中向系统报告按键及同步事件。

代码清单 12.28　最简单的 input 设备驱动

```
/*在按键中断中报告事件*/
static void button_interrupt(int irq, void *dummy, struct pt_regs *fp)
{
  input_report_key(&button_dev, BTN_1, inb(BUTTON_PORT) &1);
  input_sync(&button_dev);
}

static int __init button_init(void)
{
  /*申请中断*/
  if (request_irq(BUTTON_IRQ, button_interrupt, 0, "button", NULL))
  {
    printk(KERN_ERR "button.c: Can't allocate irq %d\n", button_irq);
    return  - EBUSY;
  }

  button_dev.evbit[0] = BIT(EV_KEY);    //支持 EV_KEY 事件
  button_dev.keybit[LONG(BTN_0)] = BIT(BTN_0);

  input_register_device(&button_dev);   //注册 input 设备
}

static void __exit button_exit(void)
{
  input_unregister_device(&button_dev);   //注销 input 设备
  free_irq(BUTTON_IRQ, button_interrupt); //释放中断
}
```

12.3 DSP HPI 的设备驱动

12.3.1 HPI 接口的硬件原理

TI 公司的一些 DSP 提供了一个典型的主机并行接口（HPI）供外部 CPU 通过存储总线读取和写入 DSP 的内存。通过 HPI 接口，DSP 可以与外部 CPU 通信，甚至包括DSP所用的程序也可以直接由CPU通过HPI下载到DSP的内存中。典型的host CPU 连接 DSP HPI 接口的电路原理如图 12.7 所示。

图 12.7　host 与 DSP HPI 连接

HINT 也是一个输出信号，由 HPIC 中的 HINT 比特位确定，HPI 可以利用该引脚中断主机。HR/W 为 HPI 的读写控制信号，高电平为读操作，低电平为写操作。HRDY 为 HPI 的输出信号，高电平表示 HPI 准备就绪。HHWIL 的功能就是区分当前在 HPI 总线上的数据是前一个半字，还是后一个半字。HBE[1:0]是字节使能信号。HCS 为 HPI 接口的片选信号。HDS1 和 HDS2 为 HPI 数据有效信号。

DSP 的 HPI 接口向主机提供了 3 个寄存器：地址寄存器（HPIA）、数据寄存器（HPID）和控制寄存器（HPIC），通过 HPI 接口的 HCNTL[0]、HCNTL[1]可以选择相应的寄存器，如表 12.1 所示。

表 12.1　　HPI 接口寄存器的选择

HCNTL[0]	HCNTL[1]	功 能 描 述
0	0	HPIC 的读写
0	1	读写 HPID，HPIA 自动增加（HPIA 寄存器只能由主机 CPU 读写）
1	0	HPIA 的读写
1	1	读写 HPID，HPIA 不自动增加

通过 DSP HPI 接口的地址、数据和控制寄存器即可以完成整个 HPI 接口的操作，由于 HCNTL[1 :0]、HR/W 都直接由主机的地址线连接，因此地址的不同就可区分不同的寄存器。根据图 12.7 的连接，HPI 寄存器的读写访问方式如代码清单 12.29 所示。

代码清单 12.29　HPI 接口地址、数据和控制寄存器访问

```
1  #define HPI_BASEADDR      0x08000000     //     BANK 1
2  #define bHPI(Nb)          __REG1(HPI_BASEADDR +(Nb))
3  #define HPIC_WRITE bHPI(0x0) //EA3(HCNTL0):0 EA4(HCNTL1):0 EA5:0(w)
4  #define HPIC_READ bHPI(0x20) //EA3(HCNTL0):0 EA4(HCNTL1):0 EA5:1(r)
5  #define HPIA_WRITE bHPI(0x08)
6  #define HPIA_READ bHPI(0x48)
7  #define HPID_WRITE bHPI(0x10)
8  #define HPID_READ bHPI(0x30)
```

HPI 接口地址寄存器是 host 要读取或写入的 DSP 存储空间地址，读 DSP 存储空间时，数据寄存器保存读取的数据，写 DSP 存储空间时，数据寄存器保存写入的数据，控制寄存器则控制是读还是写，读写过程中地址是否自增等。

12.3.2　HPI 接口设备驱动中数据结构

代码清单 12.30 列出了 HPI 接口的设备结构体，包括用于并发控制的信号量、读和写缓冲区指针及 cdev 结构体。

代码清单 12.30　HPI 接口设备结构体

```
1 typedef struct
2 {
3   struct semaphore sem;   //信号量
4   char *HpiBaseBufRead;   //读缓冲区指针
5   char *HpiBaseBufWrite;  //写缓冲区指针
6   struct cdev cdev;
7 } HPI_DEVICE;
```

在 HPI 接口的驱动中，实现了 open()、release()、read()、write()等函数，因此，其文件操作结构体如代码清单 12.31 所示。

代码清单 12.31　HPI 接口设备驱动的文件操作结构体

```
1  static struct file_operations hpi_fops =
2  {
3    owner: THIS_MODULE,
4    open: hpi_open,//打开函数
5    read: hpi_read, //读函数
6    write: hpi_write, //写函数
7    ioctl: hpi_ioctl, //ioctl()函数
8    release: hpi_release, //释放函数
9  };
```

12.3.3　HPI 接口设备驱动的读写函数

代码清单 12.32 给出了 HPI 接口的读函数，它首先在控制器寄存器写入自增读标志，然后在地址寄存器写入要读取的地址，之后连续循环地读数据寄存器，读完之后，将数据复制到用户空间。

代码清单 12.32　HPI 接口设备驱动的读函数

```
1  static ssize_t hpi_read(struct file *file, char *buf, size_t count,
2    loff_t *oppos)
```

```
3 {
4   ...
5   down(&(pHpiDevice->sem));  //获取并发访问信号量
6   HPIC_WRITE = AUTO_READ_FLAGS; //表明要读 DSP 内存
7   HPIA_WRITE =  *oppos; //读的起始地址
8   count = min(count, MAX_DSP_ADDRESS - *oppos);
9   for (i = 0; i < count; i++)    //连续地读 count 个数据
10  {
11    read_buf[i] = HPID_READ;
12    //读取到 HpiBaseBufRead 缓冲区
13  }
14
15  ret = copy_to_user(buf, read_buf, count) ?  -
16    EFAULT: ret;      //复制到用户空间
17  //...
18  up(&(pHpiDevice->sem));   //释放并发访问信号量
19  return count;
20 }
```

代码清单 12.33 给出了 HPI 接口的写函数，它首先把数据从用户空间复制到内核空间，然后在控制器寄存器写入自增写标志，然后在地址寄存器填入要写的 DSP 内存目标地址，之后连续循环地写数据寄存器。

代码清单 12.33　HPI 接口设备驱动的写函数

```
1 static ssize_t hpi_write(struct file *file, char *buf, size_t count,
2   loff_t *oppos)
3 {
4   ...
5   ret = copy_from_user(write_buf, buf, count) ?  -
6     EFAULT: ret;
7   ...
8   down(&(pHpiDevice->sem));
9   HPIC_WRITE = AUTO_WRITE_FLAGS; //表明要读 DSP 内存
10  HPIA_WRITE =  *oppos; //写的起始地址
11  count = min(count,MAX_DSP_ADDRESS-*oppos);
12  for (i = 0; i < count; i++)
13  {
14    HPID_WRITE = write_buf[i];
15    //读取到 HpiBaseBufRead 缓冲区
```

```
16  }
17
18  //...
19  up(&(pHpiDevice->sem));
20  return count;
21 }
```

读和写都涉及公共的寄存器（HPI 的控制寄存器、地址寄存器和数据寄存器）及 HPI 接口提供给 host 端读写的内存的操作，所以一定要用信号量等互斥机制加以保护，否则，会出现读写错乱的现象。

DSP 通过 HPI 接口可以中断 host，具体的中断处理程序受到系统中 host 和 DSP 软件通信协议的约束。

12.4 NVRAM 设备驱动

12.4.1 NVRAM 设备驱动的数据结构

在 Linux 内核中，把 NVRAM 归入了 miscdevice（混杂设备），Linux 内核所提供的 miscdevice 包括 NVRAM、看门狗、DS1286 等实时钟、字符 LCD、鼠标、AMD 768 随机数发生器等。

miscdevice 共享一个主设备号 MISC_MAJOR（即 10），但次设备号不同。所有的 miscdevice 设备形成一个链表，对设备访问时内核根据次设备号查找对应的 miscdevice 设备，然后调用其 struct file_operations 中注册的文件操作接口进行操作。

在 Linux 内核中，用 struct miscdevice 结构体表示 miscdevice 设备，这个结构体的定义如代码清单 12.34 所示，而代码清单 12.35 则给出了对应于 NVRAM 的该结构体实例。从代码清单可以看出，这一部分代码仍然没有及时更新，例如 devfs 仍然在其中。

代码清单 12.34　miscdevice 结构体

```
1  struct miscdevice
2  {
3    int minor;
4    const char *name;//设备名
5    struct file_operations *fops;//文件操作
6    struct list_head list;//链表头
7    struct device *dev;
8    struct class_device *class ;
9    char devfs_name[64];//devfs 文件节点名
10 };
```

代码清单 12.35　NVRAM 设备结构体

```
static struct miscdevice nvram_dev =
{
  NVRAM_MINOR, //次设备号
  "nvram",   //设备名
  &nvram_fops   //文件操作结构体
};
```

在 NVRAM 设备驱动中，将实现读、写、seek()和 ioctl()函数，因此 NVRAM 驱动的文件操作结构体如代码清单 12.36 所示。

代码清单 12.36　NVRAM 设备驱动文件操作结构体

```
struct file_operations nvram_fops =
{
  .owner = THIS_MODULE,
  .llseek = nvram_llseek,  //seek
  .read = read_nvram,  //读
  .write = write_nvram,//写
  .ioctl = nvram_ioctl,//IO 控制
};
```

12.4.2　NVRAM 设备驱动的模块加载与卸载函数

在 NVRAM 设备驱动的模块加载和卸载函数，分别要注册和注销 miscdevice 结构体，注册用 misc_register()函数，注销用 misc_deregister()函数，如代码清单 12.37 所示。

代码清单 12.37　NVRAM 设备驱动的模块加载和卸载函数

```
int __init nvram_init(void)
{
 printk(KERN_INFO "Macintosh non-volatile memory driver v%s\n",
     NVRAM_VERSION);
 return misc_register(&nvram_dev);//注册 miscdevice
}

void __exit nvram_cleanup(void)
{
  misc_deregister( &nvram_dev );//注销 miscdevice
}
```

12.4.3　NVRAM 设备驱动读写函数

NVRAM 的读函数主要实现将 NVRAM 中读出的字节通过_ _put_user()函数复制到用户空间，写函数主要实现将用户空间的字节通过__get_user()复制到内核空间后再写入 NVRAM，如代码清单 12.38 所示。

代码清单 12.38　NVRAM 设备驱动的读写函数

```
/*读函数*/
static ssize_t read_nvram(struct file *file, char __user *buf, size_t count,
  loff_t *ppos)
{
  unsigned int i;
  char __user *p = buf;

  if (!access_ok(VERIFY_WRITE, buf, count))
```

```
    9     return  - EFAULT;
    10  //偏移过大
    11  if (*ppos >= NVRAM_SIZE)
    12    return 0;
    13  for (i = *ppos; count > 0 && i < NVRAM_SIZE; ++i, ++p, --count)
    14  //读取 1 字节并复制到用户空间
    15  {
    16    if (__put_user(nvram_read_byte(i), p))
    17      return - EFAULT;
    18  }
    19  *ppos = i;
    20  return p - buf;
    21 }
    22
    23 /*写函数*/
    24 static ssize_t write_nvram(struct file *file, const char __user *buf,
size_t
    25  count, loff_t *ppos)
    26 {
    27  unsigned int i;
    28  const char __user *p = buf;
    29  char c;
    30
    31  if (!access_ok(VERIFY_READ, buf, count))
    32    return  - EFAULT;
    33  //偏移过大
    34  if (*ppos >= NVRAM_SIZE)
    35    return 0;
    36  for (i =  *ppos; count > 0 && i < NVRAM_SIZE; ++i, ++p, --count)
    37  {
    38    //从用户空间取得 1 字节并写入 NVRAM
    39    if (__get_user(c, p))
    40      return  - EFAULT;
    41    nvram_write_byte(c, i);
    42  }
    43  *ppos = i;
    44  return p - buf;
    45 }
```

12.4.4 NVRAM 设备驱动的 seek 函数

NVRAM 设备驱动中的定位函数实现了相对于当前位置和相对于 NVRAM 末尾的偏移，如代码清单 12.39 所示。

代码清单 12.39　NVRAM 设备驱动的 seek 函数

```
    1  static loff_t nvram_llseek(struct file *file, loff_t offset, int
origin)
    2  {
    3    lock_kernel();//大内核锁，已经过时
    4    switch (origin)
    5    {
    6      case 1://相对当前位置偏移
    7        offset += file->f_pos;
    8        break;
    9      case 2: //相对文件尾偏移
    10       offset += NVRAM_SIZE;
    11       break;
    12   }
    13   if (offset < 0)
    14   {
```

```
    unlock_kernel();
    return  - EINVAL;
  }
  file->f_pos = offset;
  unlock_kernel();
  return file->f_pos;
}
```

12.5 看门狗设备驱动

12.5.1 看门狗硬件原理

看门狗（watchdog）分硬件看门狗和软件看门狗。硬件看门狗是利用一个定时器电路，其定时输出连接到电路的复位端，程序在一定时间范围内对定时器清零（俗称“喂狗”），因此程序正常工作时，定时器总不能溢出，也就不能产生复位信号。如果程序出现故障，不在定时周期内复位看门狗，就使得看门狗定时器溢出产生复位信号并重启系统。软件看门狗原理上一样，只是将硬件电路上的定时器用处理器的内部定时器代替，这样可以简化硬件电路设计，但在可靠性方面不如硬件定时器。

S3C2410 内部集成了 watchdog，提供 3 个寄存器对 watchdog 进行操作，这 3 个寄存器分别为 WTCON（watchdog 控制寄存器）、WTDAT（watchdog 数据寄存器）和 WTCNT（watchdog 记数寄存器）。

对于 S3C2410 而言，开启 watchdog 的过程中首先必须在寄存器 WTDAT 里面填入计数目标，watchdog 开启之后该值会被自动加载进寄存器 WTCNT 中。watchdog 从 CPU 内部时钟分频和选择电路中获得一个工作周期 t_watchdog，每个 t_watchdog 周期结束时 WTCNT 中的值便会减 1，若在 WTCNT 递减为 0（即超时）的时候软件还没有重新往 WTCNT 中写入数值（即“喂狗”），则 watchdog 触发系统复位。

例如，通过代码清单 12.40 的一段程序可以开启 S3C2410 的看门狗。

代码清单 12.40　开启 S3C2410 的看门狗

```
void enable_watchdog()
{
  rWTCON = WTCON_DIV64 | WTCON_RSTEN;//64 分频、开启复位信号
  rWTDAT = 0x8000;//计数目标
  rWTCON |= WTCON_ENABLE;//开启看门狗
}
```

在系统开启 watchdog 后，软件必须在 WTCNT 中的值递减到 0 之前重新往该寄存器写入一个非 0 值，否则将引起系统重启，代码清单 12.41 给出了 S3C2410 的“喂狗”函数。

代码清单 12.41　S3C2410 的看门狗“喂狗”

```
void feed_dog()
{
```

```
  rWTCNT=0x8000;
}
```

代码清单 12.42 给出了看门狗的启动和“喂狗”例程，它在 while(1)死循环中不断地“喂狗”，如果程序跑飞了，while(1)将进不去，不会再“喂狗”，这会导致看门狗超时，系统会复位以求使软件的工作恢复正常。

代码清单 12.42　看门狗的使用例程

```
void main()
{
  init_system();
  ...
  enable_watchdog();//启动看门狗
  ...
  while(1)
  {
     ...
     feed_dog(); //喂狗
  }
}
```

12.5.2　看门狗驱动中的数据结构

在 Linux 设备驱动中，有一类设备被称为“平台设备”，通常 SoC 系统中集成的独立的外设单元都被当作平台设备处理。平台设备用 platform_device 结构体来描述，这个结构体的定义如代码清单 12.43 所示。

代码清单 12.43　platform_device 结构体

```
struct platform_device
{
  const char  * name;//设备名
  u32     id;
  struct device dev;
  u32     num_resources;//设备所使用各类资源数量
  struct resource * resource;//资源
};
```

注意，所谓的“平台设备”并不是与字符设备、块设备和网络设备并列的概念，而是 Linux 系统提供的一种附加手段，例如，在 S3C2410 处理器中，把内部集成的 I^2C、IIS、RTC、看门狗等都归纳为平台设备，而它们本身就是字符设备。代码清单 12.44 列出了 S3C2410 开发板中的平台设备，代码清单 12.45 给出了看门狗的 platform_device 结构体。

代码清单 12.44　S3C2410 中的平台设备

```
struct platform_device *s3c24xx_uart_devs[];
```

```

struct platform_device s3c_device_usb; //USB 控制器
struct platform_device s3c_device_lcd; //LCD 控制器
struct platform_device s3c_device_wdt; //看门狗
struct platform_device s3c_device_i2c; //I2C 控制器
struct platform_device s3c_device_iis; //IIS
struct platform_device s3c_device_rtc; //实时钟
 ...
/*SMDK2410 开发板使用的平台设备*/
static struct platform_device *smdk2410_devices[]__initdata =
{
  &s3c_device_usb, //USB
  &s3c_device_lcd, //LCD
  &s3c_device_wdt, //看门狗
  &s3c_device_i2c, //I2C
  &s3c_device_iis,  //IIS
};
```

代码清单 12.45　S3C2410 看门狗的 platform_device 结构体

```
struct platform_device s3c_device_wdt =
{
  .name = "s3c2410-wdt",  //设备名
  .id =  - 1, .
  num_resources = ARRAY_SIZE(s3c_wdt_resource),  //资源数量
  .resource = s3c_wdt_resource,  //看门狗所使用资源
};
```

通过 platform_add_devices()函数可以将平台设备添加到系统中，这个函数的原型为：

```
int platform_add_devices(struct platform_device **devs, int num);
```

该函数的第一个参数为平台设备数组的指针，第二个参数为平台设备的数量，它内部调用了 platform_device_register()函数用于注册单个的平台设备，如代码清单 12.46 所示。

代码清单 12.46　int platform_add_devices()函数

```
int platform_add_devices(struct platform_device **devs, int num)
{
  int i, ret = 0;

  for (i = 0; i < num; i++)
  {
```

```
    ret = platform_device_register(devs[i]);/*注册平台设备*/
    if (ret) /*注册失败*/
    {
      while (--i >= 0)
         platform_device_unregister(devs[i]);/*注销已经注册的平台设备*/
      break;
    }
  }

  return ret;
}
```

被定义为平台设备的字符设备的设备驱动中，文件操作结构体中的 open()、release()、read()、write()、ioctl()等函数仍然是需要实现的，除此之外，Linux 系统还为平台设备定义了平台驱动结构体 platform_driver，这个结构体中包含 probe()、remove()、shutdown()、suspend()、resume()函数，通常也需要由驱动实现，如代码清单 12.47。

代码清单 12.47　platform_driver 结构体

```
struct platform_driver
{
 int (*probe)(struct platform_device *);//探测
 int (*remove)(struct platform_device *);//移除
 void (*shutdown)(struct platform_device *);//关闭
 int (*suspend)(struct platform_device *, pm_message_t state);//挂起
 int (*resume)(struct platform_device *);//恢复
 struct device_driver driver;
};
```

代码清单 12.48 给出了 S3C2410 看门狗的平台驱动结构体。

代码清单 12.48　S3C2410 看门狗的 platform_driver 结构体

```
static struct platform_driver s3c2410wdt_driver =
{
 .probe      = s3c2410wdt_probe,//S3C2410 看门狗探测
 .remove     = s3c2410wdt_remove,// S3C2410 看门狗移除
 .shutdown   = s3c2410wdt_shutdown,//S3C2410 看门狗关闭
 .suspend    = s3c2410wdt_suspend,//S3C2410 看门狗挂起
 .resume     = s3c2410wdt_resume, //S3C2410 看门狗恢复
 .driver     = {
     .owner  = THIS_MODULE,
     .name   = "s3c2410-wdt",//设备名
 },
```

```
12 };
```

注意，代码清单 12.48 中第 10 行和代码清单 12.45 第 3 行定义了相同的设备名“s3c2410-wdt”，正是依靠这个相同的设备名，使得 platform_driver 和 platform_device 被建立关联。

在平台设备结构体中，包含其所用的资源数量和指针，代码清单 12.49 所示为 S3C2410 看门狗所使用的 I/O 内存和 IRQ 资源。

代码清单 12.49　S3C2410 看门狗所用资源

```
1  static struct resource s3c_wdt_resource[] =
2  {
3    [0] =
4    {
5      .start = S3C24XX_PA_WATCHDOG,   //看门狗 I/O 内存开始位置
6      .end = S3C24XX_PA_WATCHDOG + S3C24XX_SZ_WATCHDOG - 1,
7          //看门狗 I/O 内存结束位置
8      .flags = IORESOURCE_MEM,  //I/O 内存资源
9    } ,
10   [1] =
11   {
12     .start = IRQ_WDT, //看门狗开始 IRQ 号
13     .end = IRQ_WDT, //看门狗结束 IRQ 号
14     .flags = IORESOURCE_IRQ,  //IRQ 资源
15   }
16 };
```

在 Linux 内核中，看门狗仍然被放在混杂设备（即主设备号为 10）内，因此，S3C2410 看门狗也对应一个 miscdevice 结构体实例，如代码清单 12.50 所示。

代码清单 12.50　S3C2410 看门狗驱动的 miscdevice 结构体

```
1 static struct miscdevice s3c2410wdt_miscdev =
2 {
3   .minor      = WATCHDOG_MINOR,//次设备号
4   .name       = "watchdog",
5   .fops       = &s3c2410wdt_fops,//文件操作结构体
6 };
```

S3C2410 看门狗驱动 miscdevice 结构体中的文件操作结构体 s3c2410wdt_fops 的定义如代码清单 12.51 所示。

代码清单 12.51　S3C2410 看门狗驱动的文件操作结构体

```
1 static struct file_operations s3c2410wdt_fops =
```

```
{
  .owner      = THIS_MODULE,
  .llseek     = no_llseek,   //seek
  .write      = s3c2410wdt_write,   //写函数
  .ioctl      = s3c2410wdt_ioctl, //ioctl 函数
  .open       = s3c2410wdt_open,  //打开函数
  .release    = s3c2410wdt_release,//释放函数
};
```

图 12.8　看门狗与字符设备、混杂设备和平台设备

S3C2410 的看门狗同时具备多重身份：字符设备、混杂设备、平台设备，这究竟是怎么回事？

字符设备描述了看门狗的访问方式是串行、顺序的，而不是随机、缓冲的；混杂设备意味着看门狗这个字符设备被丢在了使用同一设备号的混合设备里面；平台设备意味着看门狗这个设备是属于平台的独立的模块，它完全是一项附加信息。

对于看门狗而言，“字符设备”是对其“本质”的描述，“混杂设备”是存放这个字符设备的“容器”，“平台设备”则描述了看门狗的一种“特征”或“属性”。

例如一颗花生，如果我们把它本身看作一个字符设备，当放在八宝粥里面，它就成了八宝粥混杂设备之一，而花生本身挂接在花生树苗的根部，成为整个花生树苗中一个独立的硬件单元而存在，因此，也可以定义为一个平台设备。图 12.8 所示为看门狗设备驱动中其作为字符设备、混杂设备和平台设备时的各组成元素，看门狗设备集三重身份于一身。

12.5.3　看门狗驱动模块的加载和卸载函数

在 S3C2410 驱动模块的加载和卸载函数分别调用 platform_driver_register()和 platform_driver_ unregister()注册和注销 platform_driver，如代码清单 12.52 所示。

代码清单 12.52　S3C2410 看门狗驱动

```
static int __init watchdog_init(void)
{
 printk(banner);
 return   platform_driver_register(&s3c2410wdt_driver);//  注  册 platform_driver
}

static void __exit watchdog_exit(void)
{
```

```
platform_driver_unregister(&s3c2410wdt_driver);//注销platform_driver
}
```

12.5.4 看门狗驱动探测和移除函数

在 S3C2410 看门狗 platform_driver 结构体成员的探测函数中，需要通过 platform_get_resource()函数分别获得看门狗所使用的内存资源和 IRQ 资源，并调用 request_mem_region()、request_irq()分别申请 I/O 内存和中断号。看门狗所使用的时钟源也采用与资源类似的方式获得和使能。最后，探测函数调用 misc_register()注册看门狗这个混杂设备，如代码清单 12.53 所示。

代码清单 12.53 S3C2410 看门狗驱动的探测函数

```
static int s3c2410wdt_probe(struct platform_device *pdev)
{
  struct resource *res;
  int started = 0;
  int ret;
  int size;

  DBG("%s: probe=%p\n", __FUNCTION__, pdev);

  /* 获得看门狗的内存区域 */

  res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
  if (res == NULL)
  {
    printk(KERN_INFO PFX "failed to get memory region resouce\n");
    return  - ENOENT;
  }

  size = (res->end - res->start) + 1;
  //申请 I/O 内存
  wdt_mem = request_mem_region(res->start, size, pdev->name);
  if (wdt_mem == NULL)
  {
    printk(KERN_INFO PFX "failed to get memory region\n");
    return  - ENOENT;
  }

  wdt_base = ioremap(res->start, size); //设备内存->虚拟地址
  if (wdt_base == 0)
  {
    printk(KERN_INFO PFX "failed to ioremap() region\n");
    return  - EINVAL;
  }

  DBG("probe: mapped wdt_base=%p\n", wdt_base);

  /* 获得看门狗的中断 */

  res = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
  if (res == NULL)
```

```
41  {
42    printk(KERN_INFO PFX "failed to get irq resource\n");
43    return  - ENOENT;
44  }
45  //申请中断
46   ret = request_irq(res->start, s3c2410wdt_irq, 0, pdev->name,
pdev);
47  if (ret != 0)
48  {
49    printk(KERN_INFO PFX "failed to install irq (%d)\n", ret);
50    return ret;
51  }
52
53  wdt_clock = clk_get(&pdev->dev, "watchdog"); //获得看门狗时钟源
54  if (wdt_clock == NULL)
55  {
56    printk(KERN_INFO PFX "failed to find watchdog clock source\n");
57    return  - ENOENT;
58  }
59
60  clk_enable(wdt_clock);
61
62  /* 看看是否能设置定时器的超时时间为期望的值，如果不能，使用缺省值 */
63
64  if (s3c2410wdt_set_heartbeat(tmr_margin))
65  {
66    started = s3c2410wdt_set_heartbeat(
67        CONFIG_S3C2410_WATCHDOG_DEFAULT_TIME);
68    if (started == 0)
69    {
70      printk(KERN_INFO PFX "tmr_margin value out of range, default
%d
71        used\n",CONFIG_S3C2410_WATCHDOG_DEFAULT_TIME);
72    }
73    else
74    {
75      printk(KERN_INFO PFX
76        "default timer value is out of range, cannot start\n");
77    }
78  }
79
80  //注册 miscdevice
81  ret = misc_register(&s3c2410wdt_miscdev);
82  if (ret)
83  {
84    printk(KERN_ERR PFX "cannot register miscdev on minor=%d (%d)\n",
85      WATCHDOG_MINOR, ret);
86    return ret;
87  }
88
89  if (tmr_atboot && started == 0)
90  {
91    printk(KERN_INFO PFX "Starting Watchdog Timer\n");
92    s3c2410wdt_start();
93  }
94
95  return 0;
96 }
```

第 39 行调用的 platform_get_resource()函数的实现原理非常简单，它直接返回 platform_device 结构体中对应序号、对应类型的 resource 指针，如代码清单 12.54 所示。

代码清单 12.54　platform_get_resource()函数

```
1  struct resource *platform_get_resource(struct platform_device *dev, unsigned
2    int type, unsigned int num)
3  {
4    int i;
5
6    for (i = 0; i < dev->num_resources; i++)
7    {
8      struct resource *r = &dev->resource[i];
9      if ((r->flags &(IORESOURCE_IO | IORESOURCE_MEM |
10       IORESOURCE_IRQ |IORESOURCE_DMA)) == type) /*如果类型匹配*/
11       if (num-- == 0)/*序号*/
12         return r; /*返回 resource 指针*/
13   }
14   return NULL;
15 }
```

S3C2410 看门狗驱动移除函数实现与探测函数相反的工作，包括释放 I/O 内存资源、IRQ 资源并禁止时钟源，最后注销混杂设备，如代码清单 12.55 所示。

代码清单 12.55　S3C2410 看门狗驱动的探测函数

```
1  static int s3c2410wdt_remove(struct platform_device *dev)
2  {
3    if (wdt_mem != NULL)
4    {
5      release_resource(wdt_mem); //释放资源
6      kfree(wdt_mem);//释放内存
7      wdt_mem = NULL;
8    }
9
10   //释放中断
11   if (wdt_irq != NULL)
12   {
13     free_irq(wdt_irq->start, dev);
14     wdt_irq = NULL;
```

```
15  }
16
17  //禁止时钟源
18  if (wdt_clock != NULL)
19  {
20    clk_disable(wdt_clock);
21    clk_put(wdt_clock);
22    wdt_clock = NULL;
23  }
24
25  misc_deregister(&s3c2410wdt_miscdev);//注销 miscdevice
26  return 0;
27 }
```

12.5.5 看门狗驱动的挂起和恢复函数

在看门狗设备驱动的挂起函数中，应该保存看门狗的控制寄存器和数据寄存器，并停止该看门狗，如代码清单 12.56 所示。

代码清单 12.56 看门狗设备驱动的挂起函数

```
1  static int s3c2410wdt_suspend(struct platform_device *dev, pm_message_t state)
2  {
3   /* 保存看门狗状态，停止它 */
4   wtcon_save = readl(wdt_base + S3C2410_WTCON);
5   wtdat_save = readl(wdt_base + S3C2410_WTDAT);
6
7   s3c2410wdt_stop();
8
9   return 0;
10 }
```

在看门狗设备驱动的恢复函数中，应该将先前保存的看门狗的控制寄存器和数据寄存器还原（计数寄存器也由先前的数据寄存器填充），并启动该看门狗，如代码清单 12.57 所示。

代码清单 12.57 看门狗设备驱动恢复函数

```
1  static int s3c2410wdt_resume(struct platform_device *dev)
2  {
3   /* 恢复看门狗状态 */
4   writel(wtdat_save, wdt_base + S3C2410_WTDAT);
5   writel(wtdat_save, wdt_base + S3C2410_WTCNT);
6   writel(wtcon_save, wdt_base + S3C2410_WTCON);
7
8   printk(KERN_INFO PFX "watchdog %sabled\n", (wtcon_save
9     &S3C2410_WTCON_ENABLE)? "en" : "dis");
10
11  return 0;
12 }
```

12.5.6 看门狗驱动的打开和释放函数

S3C2410 看门狗的设备驱动使用了一个互斥锁 open_lock 使同时最多只能有 1 个进程打开看门狗对应的设备文件，在 s3c2410wdt_open()函数中，用 down_trylock()尝试获得锁，若不能获得该锁，则证明其他进程获得了这个锁，返回- EBUSY，否则打开函数启动看门狗，如代码清单 12.58 所示。

代码清单 12.58 看门狗设备驱动的打开函数

```
1  static int s3c2410wdt_open(struct inode *inode, struct file *file)
2  {
3    if (down_trylock(&open_lock))  //获得打开锁
4      return  - EBUSY;
5
6    if (nowayout)
7    {
8      __module_get(THIS_MODULE);
9    }
10   else
11   {
12     allow_close = CLOSE_STATE_ALLOW;
13   }
14
15   /* 启动看门狗 */
16   s3c2410wdt_start();
17   return nonseekable_open(inode, file);
18 }
```

为了使得看门狗设备文件被关闭后，其他进程可以打开该设备，应该在 s3c2410wdt_release()函数中释放 open_lock 锁。如果看门狗设备被允许关闭，则还需要停止看门狗，如代码清单 12.59 所示。

代码清单 12.59 看门狗设备驱动的释放函数

```
1  static int s3c2410wdt_release(struct inode *inode, struct file *file)
2  {
3    /* 停止看门狗 */
4    if (allow_close == CLOSE_STATE_ALLOW)
5    {
6      s3c2410wdt_stop();
7    }
8    else
9    {
10         printk(KERN_CRIT  PFX  "Unexpected  close,  not  stopping
watchdog!\n");
11     s3c2410wdt_keepalive();
12   }
13
14   allow_close = CLOSE_STATE_NOT;
15   up(&open_lock);  //释放打开锁
16   return 0;
17 }
```

代码清单 12.55 第 16 行和代码清单 12.56 第 6 行所调用的看门狗启动和停止函数通过读写 S3C2410 看门狗的寄存器来实现，完全硬件操作，如代码清单 12.60 所示。

代码清单 12.60　看门狗设备驱动中启停看门狗函数

```
/*停止看门狗*/
static int s3c2410wdt_stop(void)
{
  unsigned long wtcon;

  wtcon = readl(wdt_base + S3C2410_WTCON);
  //停止看门狗，禁止复位
  wtcon &= ~(S3C2410_WTCON_ENABLE | S3C2410_WTCON_RSTEN);
  writel(wtcon, wdt_base + S3C2410_WTCON);

  return 0;
}

/*开启看门狗*/
static int s3c2410wdt_start(void)
{
  unsigned long wtcon;

  s3c2410wdt_stop();

  wtcon = readl(wdt_base + S3C2410_WTCON);
  //使能看门狗，128 分频
  wtcon |= S3C2410_WTCON_ENABLE | S3C2410_WTCON_DIV128;

  if (soft_noboot)
  {
    wtcon |= S3C2410_WTCON_INTEN; //使能中断
    wtcon &= ~S3C2410_WTCON_RSTEN; //禁止复位
  }
  else
  {
    wtcon &= ~S3C2410_WTCON_INTEN; //禁止中断
    wtcon |= S3C2410_WTCON_RSTEN; //使能复位
  }

  DBG("%s: wdt_count=0x%08x, wtcon=%08lx\n", __FUNCTION__,
     wdt_count, wtcon);
  writel(wdt_count, wdt_base + S3C2410_WTDAT);
  writel(wdt_count, wdt_base + S3C2410_WTCNT);
  writel(wtcon, wdt_base + S3C2410_WTCON);

  return 0;
}
```

12.5.7　看门狗驱动写函数

如果向看门狗设备写入了一个字符“V”，将使得看门狗可以被关闭，置 allow_close 标志为 CLOSE_STATE_ALLOW，如代码清单 12.61 所示。

代码清单 12.61　看门狗设备驱动的释放函数

```
static ssize_t s3c2410wdt_write(struct file *file, const char __user *data,
  size_t len, loff_t *ppos)
{
  /* 刷新看门狗 */
  if (len)
  {
    if (!nowayout)
```

```
    {
      size_t i;
      allow_close = CLOSE_STATE_NOT;
      for (i = 0; i != len; i++)
      {
        char c;
        if (get_user(c, data + i))//用户空间->内核空间
          return  - EFAULT;
        if (c == 'V')  //如果写入了'V' ，允许关闭
          allow_close = CLOSE_STATE_ALLOW;
      }
    }
    s3c2410wdt_keepalive();
  }
  return len;
}
```

12.6 总结

到目前为止，字符设备驱动的整个讲解就暂时划上了一个句号。虽然以字符设备为依托进行讲解，但是，第 6～12 章中所描述的关于阻塞与非阻塞、异步通知、轮询、内存与 I/O 访问、并发控制等机制并非只适用于字符设备，对其他的任何设备，都存在同样的问题，也采用完全相同的处理方法。

学完本章读者已经掌握了扎实的基础理论，后面各章所讲解的设备其难点在于其复杂、庞大的结构，而不在于设备本身。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm

第 13 章　Linux 块设备驱动

华清远见

块设备是与字符设备并列的概念，这两类设备在 Linux 中驱动的结构有较大差异，总体而言，块设备驱动比字符设备驱动要复杂得多，在 I/O 操作上表现出极大的不同，缓冲、I/O 调度、请求队列等都是与块设备驱动相关的概念。本章将详细讲解 Linux 块设备驱动的编程方法。

13.1 节讲解块设备 I/O 操作的特点，分析字符设备与块设备在 I/O 操作上的差异。

13.2 节从整体上描述 Linux 块设备驱动的结构，分析主要的数据结构、函数及其关系。

13.3～13.5 节分别讲解块设备驱动模块加载与卸载、打开与释放和 ioctl()函数。

13.6 节非常重要，讲述了块设备 I/O 操作所依赖的请求队列的概念及用法。

13.2 节与 13.3～13.6 节是整体与部分的关系，13.2～13.6 节与 13.7 节是迭代递进的关系。

13.7 节在 13.1～13.6 节讲解内容的基础上，总结 Linux 下块设备的读写流程，而 13.7 节则讲解了块设备驱动的具体实例，即 RamDisk 的驱动。

13.1 块设备的 I/O 操作特点

字符设备与块设备 I/O 操作的不同如下。

（1）块设备只能以块为单位接受输入和返回输出，而字符设备则以字节为单位。大多数设备是字符设备，因为它们不需要缓冲而且不以固定块大小进行操作。

（2）块设备对于 I/O 请求有对应的缓冲区，因此它们可以选择以什么顺序进行响应，字符设备无须缓冲且被直接读写。对于存储设备而言调整读写的顺序作用巨大，因为在读写连续的扇区比分离的扇区更快。

（3）字符设备只能被顺序读写，而块设备可以随机访问。虽然块设备可随机访问，但是对于磁盘这类机械设备而言，顺序地组织块设备的访问可以提高性能，如图 13.1 所示，对扇区 1、10、3、2 的请求被调整为对扇区 1、2、3、10 的请求。而对 SD 卡、RamDisk 等块设备而言，不存在机械上的原因，进行这样的调整没有必要。

图 13.1　调整块设备 I/O 操作的顺序

13.2 Linux 块设备驱动结构

13.2.1　block_device_operations 结构体

在块设备驱动中，有 1 个类似于字符设备驱动中 file_operations 结构体的 block_device_operations 结构体，它是对块设备操作的集合，定义如代码清单 13.1 所示。

代码清单 13.1　block_device_operations 结构体

```
1  struct block_device_operations
2  {
3    int(*open)(struct inode *, struct file*);  //打开
```

```
    int(*release)(struct inode *, struct file*);  //释放
    int(*ioctl)(struct inode *, struct file *, unsigned, unsigned long);
//ioctl
    long(*unlocked_ioctl)(struct file *, unsigned, unsigned long);
    long(*compat_ioctl)(struct file *, unsigned, unsigned long);
     int(*direct_access)(struct block_device *, sector_t, unsigned
long*);
    int(*media_changed)(struct gendisk*);  //介质被改变?
    int(*revalidate_disk)(struct gendisk*);  //使介质有效
    int(*getgeo)(struct block_device *, struct hd_geometry*);//填充驱
动器信息
    struct module *owner; //模块拥有者
 };
```

下面对其主要的成员函数进行分析。

1. 打开和释放

```
int (*open)(struct inode *inode, struct file *filp);
int (*release)(struct inode *inode, struct file *filp);
```

与字符设备驱动类似，当设备被打开和关闭时将调用它们。

2. IO 控制

```
int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd,
unsigned long arg);
```

上述函数是 ioctl()系统调用的实现，块设备包含大量的标准请求，这些标准请求由 Linux 块设备层处理，因此大部分块设备驱动的 ioctl()函数相当短。

3. 介质改变

```
int (*media_changed) (struct gendisk *gd);
```

被内核调用来检查是否驱动器中的介质已经改变，如果是，则返回一个非 0 值，否则返回 0。这个函数仅适用于支持可移动介质的驱动器，通常需要在驱动中增加一个表示介质状态是否改变的标志变量，非可移动设备的驱动不需要实现这个方法。

4. 使介质有效

```
int (*revalidate_disk) (struct gendisk *gd);
```

revalidate_disk()函数被调用来响应一个介质改变，它给驱动一个机会来进行必要的工作以使新介质准备好。

5. 获得驱动器信息

```
int (*getgeo)(struct block_device *, struct hd_geometry *);
```

该函数根据驱动器的几何信息填充一个 hd_geometry 结构体，hd_geometry 结构体

包含磁头、扇区、柱面等信息。

6．模块指针

```
struct module *owner;
```

一个指向拥有这个结构体的模块的指针，它通常被初始化为 THIS_MODULE。

13.2.2 gendisk 结构体

在 Linux 内核中，使用 gendisk（通用磁盘）结构体来表示 1 个独立的磁盘设备（或分区），这个结构体的定义如代码清单 13.2 所示。

代码清单 13.2 gendisk 结构体

```
struct gendisk
{
  int major; /* 主设备号 */
  int first_minor;  /*第 1 个次设备号*/
  int minors; /* 最大的次设备数，如果不能分区，则为 1*/
  char disk_name[32]; /* 设备名称 */
  struct hd_struct **part; /* 磁盘上的分区信息 */
  struct block_device_operations *fops; /*块设备操作结构体*/
  struct request_queue *queue;  /*请求队列*/
  void *private_data;  /*私有数据*/
  sector_t capacity; /*扇区数，512 字节为 1 个扇区*/

  int flags;
  char devfs_name[64];
  int number;
  struct device *driverfs_dev;
  struct kobject kobj;

  struct timer_rand_state *random;
  int policy;

  atomic_t sync_io; /* RAID */
  unsigned long stamp;
  int in_flight;
  #ifdef CONFIG_SMP
    struct disk_stats *dkstats;
  #else
    struct disk_stats dkstats;
  #endif
};
```

major、first_minor 和 minors 共同表征了磁盘的主、次设备号，同一个磁盘的各个分区共享一个主设备号，而次设备号则不同。fops 为 block_device_operations，即上节描述的块设备操作集合。queue 是内核用来管理这个设备的 I/O 请求队列的指针。capacity 表明设备的容量，以 512 个字节为单位。private_data 可用于指向磁盘的任何私有数据，用法与字符设备驱动 file 结构体的 private_data 类似。

Linux 内核提供了一组函数来操作 gendisk，如下所示。

1．分配 gendisk

gendisk 结构体是一个动态分配的结构体，它需要特别的内核操作来初始化，驱动不能自己分配这个结构体，而应该使用下列函数来分配 gendisk：

```
struct gendisk *alloc_disk(int minors);
```

minors 参数是这个磁盘使用的次设备号的数量，一般也就是磁盘分区的数量，此后 minors 不能被修改。

2．增加 gendisk

gendisk 结构体被分配之后，系统还不能使用这个磁盘，需要调用如下函数来注册这个磁盘设备。

```
void add_disk(struct gendisk *gd);
```

特别要注意的是对 add_disk()的调用必须发生在驱动程序的初始化工作完成并能响应磁盘的请求之后。

3．释放 gendisk

当不再需要一个磁盘时，应当使用如下函数释放 gendisk。

```
void del_gendisk(struct gendisk *gd);
```

4．gendisk 引用计数

gendisk 中包含一个 kobject 成员，因此，它是一个可被引用计数的结构体。通过 get_disk()和 put_disk()函数可用来操作引用计数，这个工作一般不需要驱动亲自做。通常对 del_gendisk()的调用会去掉 gendisk 的最终引用计数，但是这一点并不是必须的。因此，在 del_gendisk()被调用后，这个结构体可能继续存在。

5．设置 gendisk 容量

```
void set_capacity(struct gendisk *disk, sector_t size);
```

块设备中最小的可寻址单元是扇区，扇区大小一般是 2 的整数倍，最常见的大小是 512 字节。扇区的大小是设备的物理属性，扇区是所有块设备的基本单元，块设备无法对比它还小的单元进行寻址和操作，不过许多块设备能够一次就传输多个扇区。虽然大多数块设备的扇区大小都是 512 字节，不过其他大小的扇区也很常见，比如，很多 CD-ROM 盘的扇区都是 2KB。

不管物理设备的真实扇区大小是多少，内核与块设备驱动交互的扇区都以 512 字节为单位。因此，set_capacity()函数也以 512 字节为单位。

13.2.3 request 与 bio 结构体

1．请求

在 Linux 块设备驱动中，使用 request 结构体来表征等待进行的 I/O 请求，这个结构体的定义如代码清单 13.3 所示。

代码清单 13.3　request 结构体

```
struct request
{
  struct list_head queuelist; /*链表结构*/
  unsigned long flags; /* REQ_ */

  sector_t sector; /* 要传送输的下一个扇区 */
  unsigned long nr_sectors; /*要传送的扇区数目*/
  /*当前要传送的扇区数目*/
  unsigned int current_nr_sectors;

  sector_t hard_sector; /*要完成的下一个扇区*/
  unsigned long hard_nr_sectors; /*要被完成的扇区数目*/
  /*当前要被完成的扇区数目*/
  unsigned int hard_cur_sectors;

  struct bio *bio; /*请求的 bio 结构体的链表*/
  struct bio *biotail; /*请求的 bio 结构体的链表尾*/

  void *elevator_private;

  unsigned short ioprio;

  int rq_status;
  struct gendisk *rq_disk;
  int errors;
  unsigned long start_time;

  /*请求在物理内存中占据的不连续的段的数目，scatter/gather 列表的尺寸*/
  unsigned short nr_phys_segments;

  /*与 nr_phys_segments 相同，但考虑了系统 I/O MMU 的 remap */
  unsigned short nr_hw_segments;

  int tag;
  char *buffer; /*传送的缓冲，内核虚拟地址*/

  int ref_count; /* 引用计数 */
  ...
};
```

request 结构体的主要成员包括：

```
sector_t hard_sector;
unsigned long hard_nr_sectors;
unsigned int hard_cur_sectors;
```

上述 3 个成员标识还未完成的扇区，hard_sector 是第一个尚未传输的扇区，hard_nr_sectors 是尚待完成的扇区数，hard_cur_sectors 是当前 I/O 操作中待完成的扇区数。这些成员只用于内核块设备层，驱动不应当使用它们，如下所示：

```
sector_t sector;
unsigned long nr_sectors;
```

```
unsigned int current_nr_sectors;
```

驱动中会经常与这 3 个成员打交道，这 3 个成员在内核和驱动交互中发挥着重大作用。它们以 512 字节大小为一个扇区，如果硬件的扇区大小不是 512 字节，则需要进行相应的调整。例如，如果硬件的扇区大小是 2048 字节，则在进行硬件操作之前，需要用 4 来除起始扇区号。

hard_sector 、 hard_nr_sectors 、 hard_cur_sectors 与 sector 、 nr_sectors 、 current_nr_sectors 之间可认为是“副本”关系。

```
struct bio *bio;
```

bio 是这个请求中包含的 bio 结构体的链表，驱动中不宜直接存取这个成员，而应该使用后文将介绍的 rq_for_each_bio()。

```
char *buffer;
```

指向缓冲区的指针，数据应当被传送到或者来自这个缓冲区，这个指针是一个内核虚拟地址，可被驱动直接引用。

```
unsigned short nr_phys_segments;
```

该值表示相邻的页被合并后，这个请求在物理内存中占据的段的数目。

如果设备支持分散/聚集（SG，scatter/gather）操作，可依据此字段申请 sizeof(scatterlist)* nr_phys_segments 的内存，并使用下列函数进行 DMA 映射：

```
int blk_rq_map_sg(request_queue_t) *q, struct request *req,
          struct scatterlist *sglist;
```

该函数与 dma_map_sg()类似，它返回 scatterlist 列表入口的数量。

```
struct list_head queuelist;
```

用于链接这个请求到请求队列的链表结构，blkdev_dequeue_request()可用于从队列中移除请求。

使用如下宏可以从 request 获得数据传送的方向。

```
rq_data_dir(struct request *req);
```

0 返回值表示从设备中读，非 0 返回值表示向设备写。

2．请求队列

一个块请求队列是一个块 I/O 请求的队列，其定义如代码清单 13.4。

代码清单 13.4　request 队列结构体

```
struct request_queue
{
  ...
  /* 保护队列结构体的自旋锁 */
  spinlock_t __queue_lock;
  spinlock_t *queue_lock;

  /* 队列 kobject */
  struct kobject kobj;

```

```
11  /* 队列设置 */
12  unsigned long nr_requests; /* 最大的请求数量 */
13  unsigned int nr_congestion_on;
14  unsigned int nr_congestion_off;
15  unsigned int nr_batching;
16
17  unsigned short max_sectors;  /* 最大的扇区数 */
18  unsigned short max_hw_sectors;
19  unsigned short max_phys_segments; /* 最大的段数 */
20  unsigned short max_hw_segments;
21  unsigned short hardsect_size;  /* 硬件扇区尺寸 */
22  unsigned int max_segment_size;  /* 最大的段尺寸 */
23
24  unsigned long seg_boundary_mask; /* 段边界掩码 */
25  unsigned int dma_alignment;  /* DMA 传送的内存对齐限制 */
26
27  struct blk_queue_tag *queue_tags;
28
29  atomic_t refcnt; /* 引用计数 */
30
31  unsigned int in_flight;
32
33  unsigned int sg_timeout;
34  unsigned int sg_reserved_size;
35  int node;
36
37  struct list_head drain_list;
38
39  struct request *flush_rq;
40  unsigned char ordered;
41 };
```

请求队列跟踪等候的块 I/O 请求，它存储用于描述这个设备能够支持的请求的类型信息、它们的最大大小、多少不同的段可进入一个请求、硬件扇区大小、对齐要求等参数，其结果是：如果请求队列被配置正确了，它不会交给该设备一个不能处理的请求。

请求队列还实现一个插入接口，这个接口允许使用多个 I/O 调度器，I/O 调度器（也称电梯）的工作是以最优性能的方式向驱动提交 I/O 请求。大部分 I/O 调度器累积批量的 I/O 请求，并将它们排列为递增（或递减）的块索引顺序后提交给驱动。进行这些工作的原因在于，对于磁头而言，当给定顺序排列的请求时，可以使得磁盘顺序地从一头到另一头工作，非常像一个满载的电梯，在一个方向移动直到所有它的“请求”被满足。

另外，I/O 调度器还负责合并邻近的请求，当一个新 I/O 请求被提交给调度器后，它会在队列里搜寻包含邻近扇区的请求。如果找到一个，并且如果结果的请求不是太大，调度器将合并这两个请求。

对磁盘等块设备进行 I/O 操作顺序的调度类似于电梯的原理，先服务完上楼的乘客，再服务下楼的乘客效率会更高，而顺序响应用户的请求则电梯会无序地忙乱。

Linux 2.6 内核包含 4 个 I/O 调度器，它们分别是 No-op I/O scheduler、Anticipatory I/O scheduler、Deadline I/O scheduler 与 CFQ I/O scheduler。

Noop I/O scheduler 是一个简化的调度程序，它只作最基本的合并与排序。

Anticipatory I/O scheduler 是当前内核中默认的 I/O 调度器，它拥有非常好的性能，在 Linux 2.5 内核中它就相当引人注意。在与 Linux 2.4 内核进行的对比测试中，在 Linux 2.4 内核中多项以分钟为单位完成的任务，它则是以秒为单位来完成的，正因为如此它成为目前 Linux 2.6 内核中默认的 I/O 调度器。Anticipatory I/O scheduler 的缺点是比较庞大与复杂，在一些特殊的情况下，特别是在数据吞吐量非常大的数据库系统中它会变得比较缓慢。

Deadline I/O scheduler 是针对 Anticipatory I/O scheduler 的缺点进行改善而来的，表现出的性能几乎与 Anticipatory I/O scheduler 一样好，但是比 Anticipatory 小巧。

CFQ I/O scheduler 为系统内的所有任务分配相同的带宽，提供一个公平的工作环境，它比较适合桌面环境。事实上在测试中它也有不错的表现，mplayer、xmms 等多媒体播放器与它配合的相当好，回放平滑，几乎没有因访问磁盘而出现的跳帧现象。

内核 block 目录中的 noop-iosched.c、as-iosched.c、deadline-iosched.c 和 cfq-iosched.c 文件分别实现了上述调度算法。

可以通过给 kernel 添加启动参数，选择使用的 IO 调度算法，如：

```
kernel elevator=deadline
```

（1）初始化请求队列。

```
request_queue_t *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock);
```

该函数的第一个参数是请求处理函数的指针，第二个参数是控制访问队列权限的自旋锁，这个函数会发生内存分配的行为，它可能会失败，因此一定要检查它的返回值。这个函数一般在块设备驱动的模块加载函数中调用。

（2）清除请求队列。

```
void blk_cleanup_queue(request_queue_t * q);
```

这个函数完成将请求队列返回给系统的任务，一般在块设备驱动模块卸载函数中调用。

而 blk_put_queue()宏则定义为：

```
#define blk_put_queue(q) blk_cleanup_queue((q))
```

（3）分配“请求队列”。

```
request_queue_t *blk_alloc_queue(int gfp_mask);
```

对于 Flash、RAM 盘等完全随机访问的非机械设备，并不需要进行复杂的 I/O 调度，这个时候，应该使用上述函数分配一个“请求队列”，并使用如下函数来绑定请求队列和“制造请求”函数。

```
void blk_queue_make_request(request_queue_t * q, make_request_fn * mfn);
```

在 13.6.2 节我们会看到，这种方式分配的“请求队列”实际上不包含任何请求，所以给其加上引号。

（4）提取请求。

```
struct request *elv_next_request(request_queue_t *queue);
```

上述函数用于返回下一个要处理的请求（由 I/O 调度器决定），如果没有请求则返回 NULL。elv_next_request()不会清除请求，它仍然将这个请求保留在队列上，但是标识它为活动的，这个标识将阻止 I/O 调度器合并其他的请求到已开始执行的请求。因为 elv_next_request()不从队列里清除请求，因此连续调用它两次，两次会返回同一个请求结构体。

（5）去除请求。

```
void blkdev_dequeue_request(struct request *req);
```

上述函数从队列中去除一个请求。如果驱动中同时从同一个队列中操作了多个请求，它必须以这样的方式将它们从队列中去除。

如果需要将一个已经出列的请求归还到队列中，可以进行以下调用：

```
void elv_requeue_request(request_queue_t *queue, struct request *req);
```

另外，块设备层还提供了一套函数，这些函数可被驱动用来控制一个请求队列的操作，主要包括以下操作。

（6）启停请求队列。

```
void blk_stop_queue(request_queue_t *queue);
void blk_start_queue(request_queue_t *queue);
```

如果块设备到达不能处理等候的命令的状态，应调用 blk_stop_queue()来告知块设备层。之后，请求函数将不被调用，除非再次调用 blk_start_queue()将设备恢复到可处理请求的状态。

（7）参数设置。

```
void blk_queue_max_sectors(request_queue_t *queue, unsigned short max);
void blk_queue_max_phys_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_hw_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_segment_size(request_queue_t *queue, unsigned int max);
```

这些函数用于设置描述块设备可处理的请求的参数。blk_queue_max_sectors()描述任一请求可包含的最大扇区数，默认值为 255；blk_queue_max_phys_segments()和 blk_queue_max_hw_segments()都控制一个请求中可包含的最大物理段（系统内存中不相邻的区），blk_queue_max_hw_segments()考虑了系统 I/O 内存管理单元的重映射，这两个参数缺省都是 128。blk_queue_max_segment_size 告知内核请求段的最大字节数，缺省值为 65,536。

（8）通告内核。

```
void blk_queue_bounce_limit(request_queue_t *queue, u64 dma_addr);
```

该函数用于告知内核块设备执行 DMA 时可使用的最高物理地址 dma_addr，如果一个请求包含超出这个限制的内存引用，系统将会给这个操作分配一个“反弹”缓冲

区。这种方式的代价昂贵，因此应尽量避免使用。

可以给 dma_addr 参数提供任何可能的值或使用预先定义的宏，如 BLK_BOUNCE_HIGH（对高端内存页使用反弹缓冲区）、BLK_BOUNCE_ISA（驱动只可在 16MB 的 ISA 区执行 DMA）或者 BLK_BOUCE_ANY（驱动可在任何地址执行 DMA），缺省值是 BLK_BOUNCE_HIGH。

```
blk_queue_segment_boundary(request_queue_t *queue, unsigned long mask);
```

如果我们正在驱动编写的设备无法处理跨越一个特殊大小内存边界的请求，应该使用这个函数来告知内核这个边界。例如，如果设备处理跨 4MB 边界的请求有困难，应该传递一个 0x3fffff 掩码，缺省的掩码是 0xffffffff（对应 4GB 边界）。

```
void blk_queue_dma_alignment(request_queue_t *queue, int mask);
```

告知内核块设备施加于 DMA 传送的内存对齐限制，所有请求都匹配这个对齐，缺省的屏蔽是 0x1ff，它导致所有的请求被对齐到 512 字节边界。

```
void blk_queue_hardsect_size(request_queue_t *queue, unsigned short max);
```

该函数告知内核块设备硬件扇区的大小，所有由内核产生的请求都是这个大小的倍数并且被正确对界。但是，内核块设备层和驱动之间的通信还是以 512 字节扇区为单位进行。

3. 块 I/O

通常一个 bio 对应一个 I/O 请求，代码清单 13.5 给出了 bio 结构体的定义。I/O 调度算法可将连续的 bio 合并成一个请求。所以，一个请求可以包含多个 bio。

代码清单 13.5　bio 结构体

```
struct bio
{
  sector_t bi_sector; /* 要传输的第一个扇区 */
  struct bio *bi_next; /* 下一个bio */
  struct block_device    *bi_bdev;
  unsigned long bi_flags; /* 状态、命令等 */
  unsigned long bi_rw; /* 低位表示READ/WRITE，高位表示优先级*/

  unsigned short bi_vcnt; /* bio_vec数量 */
  unsigned short bi_idx; /* 当前bvl_vec索引 */

  /*不相邻的物理段的数目*/
  unsigned short bi_phys_segments;

  /*物理合并和DMA remap合并后不相邻的物理段的数目*/
  unsigned short bi_hw_segments;

  unsigned int bi_size; /* 以字节为单位所需传输的数据大小 */

  /* 为了明了最大的hw尺寸，我们考虑这个bio中第一个和最后一个
   虚拟的可合并的段的尺寸 */
  unsigned int bi_hw_front_size;
  unsigned int bi_hw_back_size;

  unsigned int bi_max_vecs; /* 我们能持有的最大bvl_vecs数 */

  struct bio_vec *bi_io_vec; /* 实际的vec列表 */

```

```
bio_end_io_t *bi_end_io;
atomic_t bi_cnt;

void *bi_private;

bio_destructor_t *bi_destructor; /* destructor */
};
```

下面我们对其中的核心成员进行分析：

```
sector_t bi_sector;
```

标识这个 bio 要传送的第一个（512 字节）扇区。

```
unsigned int bi_size;
```

被传送的数据大小，以字节为单位，驱动中可以使用 bio_sectors(bio)宏获得以扇区为单位的大小。

```
unsigned long bi_flags;
```

一组描述 bio 的标志，如果这是一个写请求，最低有效位被置位，可以使用 bio_data_dir(bio)宏来获得读写方向。

```
unsigned short bio_phys_segments;
unsigned short bio_hw_segments;
```

分别表示包含在这个 BIO 中要处理的不连续的物理内存段的数目和考虑 DMA 重映像后的不连续的内存段的数目。

bio 的核心是一个称为 bi_io_vec 的数组，它由 bio_vec 结构体组成，bio_vec 结构体的定义如代码清单 13.6 所示。

代码清单 13.6　bio_vec 结构体

```
struct bio_vec
{
  struct page *bv_page;  /* 页指针 */
  unsigned int bv_len;  /* 传输的字节数 */
  unsigned int bv_offset; /* 偏移位置 */
};
```

我们不应该直接访问 bio 的 bio_vec 成员，而应该使用 bio_for_each_segment()宏来进行这项工作，可以用这个宏循环遍历整个 bio 中的每个段，这个宏的定义如代码清单 13.7 所示。

代码清单 13.7　bio_for_each_segment 宏

```
#define __bio_for_each_segment(bvl, bio, i, start_idx) \
        for (bvl = bio_iovec_idx((bio), (start_idx)), i = (start_idx); \
            i < (bio)->bi_vcnt; \
            bvl++, i++)

#define bio_for_each_segment(bvl, bio, i) \
```

```
7           __bio_for_each_segment(bvl, bio, i, (bio)->bi_idx
```

图 13.2（a）所示为 request 队列、request 与 bio 数据结构之间的关系，13.2（b）所示为 request、bio 和 bio_vec 数据结构之间的关系，13.2（c）所示为 bio 与 bio_vec 数据结构之间的关系，因此整个图 13.2 递归地呈现了 request 队列、request、bio 和 bio_vec 这 4 个结构体之间的关系。

（a）request 与 bio

（b）request、bio 和 bio_vec

（c） bio 与 bio_vec

图 13.2　request 队列、request、bio 和 bio_vec 结构体之间的关系

内核还提供了一组函数（宏）用于操作 bio：

```
int bio_data_dir(struct bio *bio);
```

这个函数可用于获得数据传输的方向是 READ 还是 WRITE。

```
struct page *bio_page(struct bio *bio) ;
```

这个函数可用于获得目前的页指针。

```
int bio_offset(struct bio *bio) ;
```

这个函数返回操作对应的当前页内的偏移，通常块 I/O 操作本身就是页对齐的。

```
int bio_cur_sectors(struct bio *bio) ;
```

这个函数返回当前 bio_vec 要传输的扇区数。

```
char *bio_data(struct bio *bio) ;
```

这个函数返回数据缓冲区的内核虚拟地址。

```
char *bvec_kmap_irq(struct bio_vec *bvec, unsigned long *flags) ;
```

这个函数返回一个内核虚拟地址，这个地址可用于存取被给定的 bio_vec 入口指向的数据缓冲区。它也会屏蔽中断并返回一个原子 kmap，因此，在 bvec_kunmap_irq()被调用以前，驱动不应该睡眠。

```
void bvec_kunmap_irq(char *buffer, unsigned long *flags);
```

这个函数是 bvec_kmap_irq()函数的“反函数”，它撤销 bvec_kmap_irq()创建的映射。

```
char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
```

这个函数是对 bvec_kmap_irq()的包装，它返回给定的 bio 的当前 bio_vec 入口的映射。

```
char *__bio_kmap_atomic(struct bio *bio, int i, enum km_type type);
```

这个函数通过 kmap_atomic()获得返回给定 bio 的第 i 个缓冲区的虚拟地址。

```
void __bio_kunmap_atomic(char *addr, enum km_type type);
```

这个函数返还由__bio_kmap_atomic()获得的内核虚拟地址。

另外，对 bio 的引用计数通过如下函数完成：

```
void bio_get(struct bio *bio);  //引用 bio
void bio_put(struct bio *bio);  //释放对 bio 的引用
```

13.2.4 块设备驱动注册与注销

块设备驱动中的第一个工作通常是注册它们自己到内核，完成这个任务的函数是 register_ blkdev()，其原型为：

```
int register_blkdev(unsigned int major, const char *name);
```

major 参数是块设备要使用的主设备号，name 为设备名，它会在/proc/devices 中被显示。如果 major 为 0，内核会自动分配一个新的主设备号，register_blkdev()函数的返回值就是这个主设备号。如果 register_blkdev()返回一个负值，表明发生了一个错误。

与 register_blkdev()对应的注销函数是 unregister_blkdev()，其原型为：

```
int unregister_blkdev(unsigned int major, const char *name);
```

这里，传递给 register_blkdev()的参数必须与传递给 register_blkdev()的参数匹配，否则这个函数返回-EINVAL。

值得一提的是，在 Linux 2.6 内核中，对 register_blkdev()的调用完全是可选的，register_blkdev()的功能已随时间正在减少，这个调用最多只完成两件事。

① 如果需要，分配一个动态主设备号。

② 在/proc/devices 中创建一个入口。

在将来的内核中，register_blkdev()可能会被去掉。但是目前的大部分驱动仍然调用它。代码清单 13.8 给出了一个块设备驱动注册的模板。

代码清单 13.8　块设备驱动注册模板

```
xxx_major = register_blkdev(xxx_major, "xxx");
if (xxx_major <= 0) //注册失败
{
  printk(KERN_WARNING "xxx: unable to get major number\n");
  return -EBUSY;
}
```

13.3 Linux 块设备驱动的模块加载与卸载

在块设备驱动的模块加载函数中通常需要完成如下工作。

① 分配、初始化请求队列，绑定请求队列和请求函数。

② 分配、初始化 gendisk，给 gendisk 的 major、fops、queue 等成员赋值，最后添加 gendisk。

③ 注册块设备驱动。

代码清单 13.9 和 13.10 分别给出了使用 blk_alloc_queue()分配请求队列并使用 blk_queue_ make_request()绑定“请求队列”和“制造请求”的函数，以及使用 blk_init_queue()初始化请求队列并绑定请求队列与请求处理函数两种不同情况下的块设备驱动模块加载函数模板。

代码清单 13.9　块设备驱动的模块加载函数模板（使用 blk_alloc_queue）

```
static int __init xxx_init(void)
{
  //分配 gendisk
  xxx_disks = alloc_disk(1);
  if (!xxx_disks)
  {
    goto out;
  }

  //块设备驱动注册
  if (register_blkdev(XXX_MAJOR, "xxx"))
  {
    err =  - EIO;
    goto out;
  }

  //“请求队列”分配
  xxx_queue = blk_alloc_queue(GFP_KERNEL);
  if (!xxx_queue)
  {
    goto out_queue;
  }

  blk_queue_make_request(xxx_queue, &xxx_make_request); //绑定“制造请求”函数
  blk_queue_hardsect_size(xxx_queue, xxx_blocksize); //硬件扇区尺寸设置

  //gendisk 初始化
```

```
28  xxx_disks->major = XXX_MAJOR;
29  xxx_disks->first_minor = 0;
30  xxx_disks->fops = &xxx_op;
31  xxx_disks->queue = xxx_queue;
32  sprintf(xxx_disks->disk_name, "xxx%d", i);
33  set_capacity(xxx_disks, xxx_size); //xxx_size 以 512bytes 为单位
34  add_disk(xxx_disks); //添加 gendisk
35
36  return 0;
37  out_queue: unregister_blkdev(XXX_MAJOR, "xxx");
38  out: put_disk(xxx_disks);
39  blk_cleanup_queue(xxx_queue);
40
41  return  - ENOMEM;
42 }
```

代码清单 13.10　块设备驱动的模块加载函数模板（使用 blk_init_queue）

```
1  static int __init xxx_init(void)
2  {
3    //块设备驱动注册
4    if (register_blkdev(XXX_MAJOR, "xxx"))
5    {
6      err =  - EIO;
7      goto out;
8    }
9
10   //请求队列初始化
11   xxx_queue = blk_init_queue(xxx_request, xxx_lock);
12   if (!xxx_queue)
13   {
14     goto out_queue;
15   }
16
17   blk_queue_hardsect_size(xxx_queue, xxx_blocksize); //硬件扇区尺寸设置
18
19   //gendisk 初始化
20   xxx_disks->major = XXX_MAJOR;
21   xxx_disks->first_minor = 0;
22   xxx_disks->fops = &xxx_op;
23   xxx_disks->queue = xxx_queue;
24   sprintf(xxx_disks->disk_name, "xxx%d", i);
25   set_capacity(xxx_disks, xxx_size *2);
26   add_disk(xxx_disks); //添加 gendisk
27
28   return 0;
29   out_queue: unregister_blkdev(XXX_MAJOR, "xxx");
30   out: put_disk(xxx_disks);
31   blk_cleanup_queue(xxx_queue);
32
33   return  - ENOMEM;
34 }
```

在块设备驱动的模块卸载函数中完成与模块加载函数相反的工作。

① 清除请求队列。

② 删除 gendisk 和对 gendisk 的引用。

③ 删除对块设备的引用，注销块设备驱动。

代码清单 13.11 给出了块设备驱动的模块卸载函数的模板。

代码清单 13.11　块设备驱动的模块卸载函数模板

```
static void __exit xxx_exit(void)
{
  if (bdev)
  {
    invalidate_bdev(xxx_bdev, 1);
    blkdev_put(xxx_bdev);
  }
  del_gendisk(xxx_disks); //删除 gendisk
  put_disk(xxx_disks);
  blk_cleanup_queue(xxx_queue[i]); //清除请求队列
  unregister_blkdev(XXX_MAJOR, "xxx");
}
```

13.4 块设备的打开与释放

块设备驱动的 open()和 release()函数并非是必须的，一个简单的块设备驱动可以不提供 open()和 release()函数。

块设备驱动的 open()函数和其字符设备驱动的对等体非常类似，都以相关的 inode 和 file 结构体指针作为参数。当一个节点引用一个块设备时，inode->i_bdev->bd_disk 包含一个指向关联 gendisk 结构体的指针。因此，类似于字符设备驱动，我们也可以将 gendisk 的 private_data 赋给 file 的 private_data，private_data 同样最好是指向描述该设备的设备结构体 xxx_dev 的指针，如代码清单 13.12 所示。

代码清单 13.12　在块设备的 open()函数中赋值 private_data

```
static int xxx_open(struct inode *inode, struct file *filp)
{
  struct xxx_dev *dev = inode->i_bdev->bd_disk->private_data;
  filp->private_data = dev;  //赋值 file 的 private_data
  ...
  return 0;
}
```

在一个处理真实的硬件设备的驱动中，open()和 release()方法还应当设置驱动和硬件的状态，这些工作可能包括启停磁盘、加锁一个可移出设备和分配 DMA 缓冲等。

13.5 块设备驱动的 ioctl 函数

与字符设备驱动一样，块设备可以包含一个 ioctl()函数以提供对设备的 I/O 控制能力。实际上，高层的块设备层代码处理了绝大多数 ioctl()，因此，具体的块设备驱动中通常不再需要实现很多 ioctl 命令。

代码清单 13.13 给出的 ioctl()函数只实现一个命令 HDIO_GETGEO，用于获得磁盘的几何信息（geometry，指 CHS，即 Cylinder、Head、Sector/Track）。

代码清单 13.13　块设备驱动的 I/O 控制函数模板

```
int xxx_ioctl(struct inode *inode, struct file *filp, unsigned int cmd,
  unsigned long arg)
{
  long size;
  struct hd_geometry geo;
  struct xxx_dev *dev = filp->private_data; //通过 file->private 获得设备结构体

  switch (cmd)
  {
    case HDIO_GETGEO:
      size = dev->size *(hardsect_size / KERNEL_SECTOR_SIZE);
      geo.cylinders = (size &~0x3f) >> 6;
      geo.heads = 4;
      geo.sectors = 16;
      geo.start = 4;
      if (copy_to_user((void __user*)arg, &geo, sizeof(geo)))
      {
        return  - EFAULT;
      }
      return 0;
  }

  return  - ENOTTY; //不知道的命令
}
```

13.6 块设备驱动的 I/O 请求处理

13.6.1　使用请求队列

块设备驱动请求函数的原型为：

```
void request(request_queue_t *queue);
```

这个函数不能由驱动自己调用，只有当内核认为是时候让驱动处理对设备的读写等操作时，它才调用这个函数。

请求函数可以在没有完成请求队列中的所有请求的情况下返回，甚至它一个请求不完成都可以返回。但是，对大部分设备而言，在请求函数中处理完所有请求后再返回通常是值得推荐的方法。代码清单 13.14 给出了一个简单的 request()函数的例子。

代码清单 13.14　块设备驱动请求函数例程

```
1  static void xxx_request(request_queue_t *q)
2  {
3    struct request *req;
4    while ((req = elv_next_request(q)) != NULL)
5    {
6      struct xxx_dev *dev = req->rq_disk->private_data;
7      if (!blk_fs_request(req)) //不是文件系统请求
8      {
9        printk(KERN_NOTICE "Skip non-fs request\n");
10       end_request(req, 0);//通知请求处理失败
11       continue;
12     }
13         xxx_transfer(dev,  req->sector,  req->current_nr_sectors,
req->buffer,
14       rq_data_dir(req)); //处理这个请求
15     end_request(req, 1); //通知成功完成这个请求
16   }
17 }
18
19 //完成具体的块设备 I/O 操作
20 static void xxx_transfer(struct xxx_dev *dev, unsigned long sector,
unsigned
21   long nsect, char *buffer, int write)
22 {
23   unsigned long offset = sector * KERNEL_SECTOR_SIZE;
24   unsigned long nbytes = nsect * KERNEL_SECTOR_SIZE;
25   if ((offset + nbytes) > dev->size)
26   {
27       printk(KERN_NOTICE "Beyond-end write (%ld %ld)\n", offset,
nbytes);
28     return ;
29   }
30   if (write)
31   {
32     write_dev(offset, buffer, nbytes);  //向设备写 nbytes 个字节的数据
33   }
34   else
35   {
36     read_dev(offset, buffer, nbytes); //从设备读 nbytes 个字节的数据
37   }
38 }
```

上述代码第 4 行使用 elv_next_request()获得队列中第一个未完成的请求，

end_request()会将请求从请求队列中剥离。第 7 行判断请求是否为文件系统请求，如果不是，则直接清除，调用 end_ request()，传递给 end_request()的第二个参数为 0 意味着处理该请求失败。而第 15 行传递给 end_request()的第二个参数为 1 意味着该请求处理成功。

end_request()函数非常重要，其源代码如代码清单 13.15 所示。

代码清单 13.15　end_request()函数源代码

```
1  void end_request(struct request *req, int uptodate)
2  {
3
4                  if     (!end_that_request_first(req,     uptodate,
req->hard_cur_sectors))
5    {
6      add_disk_randomness (req->rq_disk);
7      blkdev_dequeue_request (req);
8      end_that_request_last(req);
9    }
10 }
```

当设备已经完成一个 I/O 请求的部分或者全部扇区传输后，它必须通告块设备层，上述代码中的第 4 行完成这个工作。end_that_request_first()函数的原型为：

```
int end_that_request_first(struct request *req, int success, int
count);
```

这个函数告知块设备层，块设备驱动已经完成 count 个扇区的传送。end_that_request_first()的返回值是一个标志，指示是否这个请求中的所有扇区已经被传送。返回值为 0 表示所有的扇区已经被传送并且这个请求完成，之后，我们必须使用 blkdev_dequeue_request()来从队列中清除这个请求。最后，将这个请求传递给 end_that_request_last()函数。

```
void end_that_request_last(struct request *req);
```

end_that_request_last()通知所有正在等待这个请求完成的对象请求已经完成并回收这个请求结构体。

第 6 行的 add_disk_randomness()函数的作用是使用块 I/O 请求的定时来给系统的随机数池贡献熵，它不影响块设备驱动。但是，仅当磁盘的操作时间是真正随机的时候（大部分机械设备如此），才应该调用它。

代码清单 13.16 给出了一个更复杂的请求函数，它进行了 3 层遍历：遍历请求队列中的每个请求，遍历请求中的每个 bio，遍历 bio 中的每个段。

代码清单 13.16　请求函数遍历请求、bio 和段

```
1  static void xxx_full_request(request_queue_t *q)
2  {
3    struct request *req;
4    int sectors_xferred;
5    struct xxx_dev *dev = q->queuedata;
6    /* 遍历每个请求 */
7    while ((req = elv_next_request(q)) != NULL)
8    {
9      if (!blk_fs_request(req))
10     {
```

```
    11      printk(KERN_NOTICE "Skip non-fs request\n");
    12
    13      end_request(req, 0);
    14      continue;
    15    }
    16    sectors_xferred = xxx_xfer_request(dev, req);
    17    if (!end_that_request_first(req, 1, sectors_xferred))
    18    {
    19      blkdev_dequeue_request(req);
    20      end_that_request_last(req);
    21    }
    22  }
    23 }
    24 /* 请求处理 */
    25 static int xxx_xfer_request(struct xxx_dev *dev, struct request *req)
    26 {
    27  struct bio *bio;
    28  int nsect = 0;
    29  /* 遍历请求中的每个 bio */
    30  rq_for_each_bio(bio, req)
    31  {
    32    xxx_xfer_bio(dev, bio);
    33    nsect += bio->bi_size / KERNEL_SECTOR_SIZE;
    34  }
    35  return nsect;
    36 }
    37 /* bio 处理 */
    38 static int xxx_xfer_bio(struct xxx_dev *dev, struct bio *bio)
    39 {
    40  int i;
    41  struct bio_vec *bvec;
    42  sector_t sector = bio->bi_sector;
    43
    44  /* 遍历每一段 */
    45  bio_for_each_segment(bvec, bio, i)
    46  {
    47    char *buffer = __bio_kmap_atomic(bio, i, KM_USER0);
    48        xxx_transfer(dev,  sector,  bio_cur_sectors(bio),  buffer,
bio_data_dir(bio)
    49      == WRITE);
    50    sector += bio_cur_sectors(bio);
    51    __bio_kunmap_atomic(bio, KM_USER0);
    52  }
    53  return 0;
    54 }
```

图 13.3 所示为一个请求队列内 request、bio 以及 bio 中 segment 的层层遍历关系。

图 13.3　遍历一个请求队列

13.6.2　不使用请求队列

使用请求队列对于一个机械的磁盘设备而言的确有助于提高系统的性能，但是对于许多块设备，如数码相机的存储卡、RAM 盘等完全可真正随机访问的设备而言，无法从高级的请求队列逻辑中获益。对于这些设备，块层支持“无队列”的操作模式，为使用这个模式，驱动必须提供一个“制造请求”函数，而不是一个请求函数，“制造请求”函数的原型为：

```
typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);
```

上述函数的第一个参数仍然是“请求队列”，但是这个“请求队列”实际不包含任何请求。因此，“制造请求”函数的主要参数是 bio 结构体，这个 bio 结构体表示一个或多个要传送的缓冲区。“制造请求”函数或者直接进行传输，或者把请求重定向给其他设备。

在“制造请求”函数中处理 bio 的方式与 13.6.1 小节中讲解的完全一致，但是在处理完成后应该使用 bio_endio()函数通知处理结束，如下所示：

```
void bio_endio(struct bio *bio, unsigned int bytes, int error);
```

参数 bytes 是已经传送的字节数，它可以比这个 bio 所代表的字节数少，这意味着“部分完成”，同时 bio 结构体中的当前缓冲区指针需要更新。当设备进一步处理这个 bio 后，驱动应该再次调用 bio_endio()，如果不能完成这个请求，应指出一个错误，错误码赋值给 error 参数。

不管对应的 I/O 处理成功与否，“制造请求”函数都应该返回 0。如果“制造请求”函数返回一个非零值，bio 将被再次提交。

代码清单 13.17 所示为一个“制造请求”函数的例子。

代码清单 13.17　“制造请求”函数例程

```
static int xxx_make_request(request_queue_t *q, struct bio *bio)
{
  struct xxx_dev *dev = q->queuedata;
  int status;
  status = xxx_xfer_bio(dev, bio); //处理 bio
  bio_endio(bio, bio->bi_size, status); //通告结束
  return 0;
}
```

为了使用无队列的 I/O 请求处理，驱动模块的加载函数应遵循代码清单 13.9

的模板而非代码清单 13.10 的模板，而使用请求队列时，驱动模块的加载函数应遵循代码清单 13.10 的模板。

13.7 实例 1：RamDisk 驱动

13.7.1 RamDisk 的硬件原理

RamDisk（RAM 盘）是一种模拟磁盘，其数据实际上存储在 RAM 中，它使用一部分内存空间来模拟出一个磁盘，以块设备的方式来访问这片内存，RamDisk 对应的设备文件一般为/dev/ram%d。

使用如下一组命令就可以创建并挂载 RamDisk：

```
mkdir /tmp/ramdisk0     //创建装载点
mke2fs /dev/ram0    //创建一个文件系统
mount /dev/ram0 /tmp/ramdisk0   //装载 RamDisk
```

其中，mke2f /dev/ram0 命令的执行会回馈如下信息：

```
mke2fs 1.14, 9-Jan-1999 for EXT2 FS 0.5b, 95/08/09
Linux ext2 filesystem format
Filesystem label=
1024 inodes, 4096 blocks
204 blocks (4.98%) reserved for the super user
First data block=1
Block size=1024 (log=0)
Fragment size=1024 (log=0)
1 block group
8192 blocks per group, 8192 fragments per group
1024 inodes per group
```

表明创建了 1 个 4MB 的块设备，共包含 4096 块，每块 1024 字节。

13.7.2 RamDisk 驱动模块的加载与卸载

RamDisk 驱动的模块加载函数完成的工作与 13.3 节给出的模板完全一致，由于 RAM 盘属于完全的随机设备，宜使用无队列的 I/O 处理方式，其驱动中实现了如图 13.4 所示的与块设备驱动模板对应的函数。

代码清单 13.18 给出了 RamDisk 设备驱动的模块加载与卸载函数，实现的功能与模块是一致的。

图 13.4　块设备驱动模板与 RamDisk 设备驱动的映射

代码清单 13.18　RamDisk 设备驱动的模块加载与卸载函数

```
1  static int __init rd_init(void)
2  {
3    int i;
4    int err =  - ENOMEM;
5    //调整块尺寸
6    if (rd_blocksize > PAGE_SIZE || rd_blocksize < 512 || (rd_blocksize
&
7      (rd_blocksize - 1)))
8    {
9      printk("RAMDISK: wrong blocksize %d, reverting to defaults\n",
rd_blocksize) ;
10
11     rd_blocksize = BLOCK_SIZE;
12   }
13   //分配 gendisk
14   for (i = 0; i < CONFIG_BLK_DEV_RAM_COUNT; i++)
15   {
16     rd_disks[i] = alloc_disk(1); //分配 gendisk
17     if (!rd_disks[i])
18       goto out;
19   }
20   //块设备注册
21   if (register_blkdev(RAMDISK_MAJOR, "ramdisk"))
22   //注册块设备
23   {
24     err =  - EIO;
25     goto out;
26   }
27
28   devfs_mk_dir("rd"); //创建 devfs 目录
29
30   for (i = 0; i < CONFIG_BLK_DEV_RAM_COUNT; i++)
31   {
32     struct gendisk *disk = rd_disks[i];
33     //分配并绑定请求队列与“制造请求”函数
34     rd_queue[i] = blk_alloc_queue(GFP_KERNEL);
35     if (!rd_queue[i])
```

```
    36      goto out_queue;
    37
    38    blk_queue_make_request(rd_queue[i], &rd_make_request);   //绑定
“制造请求”函数
    39    blk_queue_hardsect_size(rd_queue[i], rd_blocksize); //硬件扇区尺
寸设置
    40
    41    //初始化 gendisk
    42    disk->major = RAMDISK_MAJOR;
    43    disk->first_minor = i;
    44    disk->fops = &rd_bd_op;
    45    disk->queue = rd_queue[i];
    46    disk->flags |= GENHD_FL_SUPPRESS_PARTITION_INFO;
    47    sprintf(disk->disk_name, "ram%d", i);
    48    sprintf(disk->devfs_name, "rd/%d", i);
    49    set_capacity(disk, rd_size *2);
    50    add_disk(rd_disks[i]); //添加 gendisk
    51  }
    52
    53  // rd_size 以 KB 为单位
    54  printk("RAMDISK driver initialized: "
    55    "%d RAMDISKs of %dK size %d blocksize\n",
    56    CONFIG_BLK_DEV_RAM_COUNT,rd_size, rd_blocksize);
    57
    58  return 0;
    59  out_queue: unregister_blkdev(RAMDISK_MAJOR, "ramdisk");
    60  out:
    61  while (i--)
    62  {
    63    put_disk(rd_disks[i]);
    64    blk_cleanup_queue(rd_queue[i]);
    65  }
    66  return err;
    67 }
    68
    69 static void __exit rd_cleanup(void)
    70 {
    71  int i;
    72
    73  for (i = 0; i < CONFIG_BLK_DEV_RAM_COUNT; i++)
    74  {
    75    struct block_device *bdev = rd_bdev[i];
    76    rd_bdev[i] = NULL;
    77    if (bdev)
    78    {
    79      invalidate_bdev(bdev, 1);
    80      blkdev_put(bdev);
    81    }
    82    del_gendisk(rd_disks[i]); //删除 gendisk
    83    put_disk(rd_disks[i]); //释放对 gendisk 的引用
    84    blk_cleanup_queue(rd_queue[i]); //清除请求队列
    85  }
```

```
86  devfs_remove("rd");
87  unregister_blkdev(RAMDISK_MAJOR, "ramdisk"); //块设备注销
88 }
```

华清远见

13.7.3 RamDisk 设备驱动 block_device_operations 及成员函数

RamDisk 提供 block_device_operations 结构体中两个成员函数的实现：open()和 ioctl()，代码清单 13.19 给出了 RamDisk 设备驱动的 block_device_operations 结构体定义及 open()和 ioctl()结构。

代码清单 13.19 RamDisk 设备驱动 block_device_operations 结构体及成员函数

```
static struct block_device_operations rd_bd_op =
{
  .owner = THIS_MODULE,
  .open = rd_open,
  .ioctl = rd_ioctl,
};

static int rd_open(struct inode *inode, struct file *filp)
{
 unsigned unit = iminor(inode);//获得次设备号

 if (rd_bdev[unit] == NULL) {
     struct block_device *bdev = inode->i_bdev;//获得 block_device 结构体指针
     struct address_space *mapping; //地址空间
     unsigned bsize;
     gfp_t gfp_mask;
    /* 设置inode成员 */
     inode = igrab(bdev->bd_inode);
     rd_bdev[unit] = bdev;
     bdev->bd_openers++;
     bsize = bdev_hardsect_size(bdev);
     bdev->bd_block_size = bsize;
     inode->i_blkbits = blksize_bits(bsize);
     inode->i_size = get_capacity(bdev->bd_disk)<<9;

     mapping = inode->i_mapping;
     mapping->a_ops = &ramdisk_aops;
     mapping->backing_dev_info = &rd_backing_dev_info;
     bdev->bd_inode_backing_dev_info = &rd_file_backing_dev_info;

     gfp_mask = mapping_gfp_mask(mapping);
     gfp_mask &= ~(__GFP_FS|__GFP_IO);
     gfp_mask |= __GFP_HIGH;
     mapping_set_gfp_mask(mapping, gfp_mask);
 }

 return 0;
}

static int rd_ioctl(struct inode *inode, struct file *file,
         unsigned int cmd, unsigned long arg)
{
 int error;
 struct block_device *bdev = inode->i_bdev;

 if (cmd != BLKFLSBUF) /* 不是flush buffer cache 命令 */
     return -ENOTTY;
```

```
48  /* 刷新buffer cache */
49  error = -EBUSY;
50  down(&bdev->bd_sem);
51  if (bdev->bd_openers <= 2) {
52      truncate_inode_pages(bdev->bd_inode->i_mapping, 0);
53      error = 0;
54  }
55  up(&bdev->bd_sem);
56  return error;
57 }
```

13.7.4 RamDisk I/O 请求处理

鉴于 RamDisk 是一种完全随机设备，其驱动中宜使用“制造请求”函数而非请求函数，这个函数的实现如代码清单 13.20 所示。

代码清单 13.20 RamDisk 设备驱动的“制造请求”函数

```
1  static int rd_make_request(request_queue_t *q, struct bio *bio)
2  {
3    struct block_device *bdev = bio->bi_bdev;
4    struct address_space *mapping = bdev->bd_inode->i_mapping;
5    sector_t sector = bio->bi_sector;
6    unsigned long len = bio->bi_size >> 9;
7    int rw = bio_data_dir(bio);//数据传输方向: 读/写?
8    struct bio_vec *bvec;
9    int ret = 0, i;
10
11   if (sector + len > get_capacity(bdev->bd_disk))
12   //超过容量
13     goto fail;
14
15   if (rw == READA)
16     rw = READ;
17   //遍历每个段
18   bio_for_each_segment(bvec, bio, i)
19   {
20     ret |= rd_blkdev_pagecache_IO(rw, bvec, sector, mapping);
21     sector += bvec->bv_len >> 9;
22   }
23   if (ret)
24     goto fail;
25
26   bio_endio(bio, bio->bi_size, 0); //处理结束
27   return 0;
28   fail: bio_io_error(bio, bio->bi_size);
29   return 0;
30 }
```

13.8 实例 2：IDE 硬盘设备驱动

13.8.1 IDE 硬盘设备原理

IDE（Integrated Drive Electronics）接口，也就是集成驱动器电路接口，原名为 ATA（AT Attachment，AT 嵌入式）接口，其本意为将硬盘控制器与盘体集成在一起的硬盘

驱动器，经历了 ATA-1 到 ATA-7 以及 SATA-1 和 SATA-2 的发展历史。ATA-1 至 ATA-4 采用 40 芯排线缆，ATA-5 至 ATA-7 则采用 40 针 80 芯线缆，虽然线缆数量增加了，但是逻辑原理没有变，只是通过物理上的改变来达到改善 PCB 信号完整性的目的，它提供更多的地线并使信号线临近地线，从而减少电流回流的面积。SATA-1 和 SATA-2 与 ATA-1 至 ATA-7 相比，数据传输方式由并行转变为串行。

IDE 接口的硬件原理实际上非常简单，对 CPU 的外围总线进行简单扩展后就可外接 IDE 控制器，表 13.1 所示为 40 针 IDE 接口的引脚定义。

表 13.1　　IDE 接口的引脚定义

引脚	信号	信号描述	信号方向	引脚	信号	信号描述	信号方
1	RSET	复位	I	2	GND	地	I/O
3	DD7	数据位 7	I/O	4	DD8	数据位 8	I/O
5	DD6	数据位 6	I/O	6	DD9	数据位 9	I/O
7	DD5	数据位 5	I/O	8	DD10	数据位 10	I/O
9	DD4	数据位 4	I/O	10	DD11	数据位 11	I/O
11	DD3	数据位 3	I/O	12	DD12	数据位 12	I/O
13	DD2	数据位 2	I/O	14	DD13	数据位 13	I/O
15	DD1	数据位 1	I/O	16	DD14	数据位 14	I/O
17	DD0	数据位 0	I/O	18	DD15	数据位 15	I/O
19	GND	地		20	N.C	未用	
21	DMARQ	DMA 请求	O	22	GND	地	
23	$\overline{\text{IDOW}}$	写选通	I	24	GND	地	
25	$\overline{\text{DIOR}}$	读选通	I	26	GND	地	
27	IORDY	通道就绪	O	28	DPSYNC:CXEL	同步电缆选择	
29	$\overline{\text{DMACK}}$	DMA 应答	O	30	GND	地	
31	$\overline{\text{INTRQ}}$	中断请求	O	32	$\overline{\text{IOCS16}}$	16 位 I/O 片选	O
33	DA1	地址 1	I	34	$\overline{\text{PDIAG}}$	诊断完成	O
35	DA0	地址 0	I	36	DA2	地址 2	I
37	$\overline{\text{CS0}}$	片选 0	I	38	$\overline{\text{CS1}}$	片选 1	I
39	$\overline{\text{DASP}}$	驱动器状态指示	O	40	GND	地	

IDE 控制器提供了一组寄存器，通过这些寄存器，主机能控制 IDE 驱动器的行为和查询其状态，表 13.2 所示 IDE 接口寄存器的定义。

华清远见

表 13.2　　IDE 接口寄存器定义

片选 1	片选 0	地址 2	地址 1	地址 0	读	写	位数
1	0	0	0	0	数据寄存器	数据寄存器	16
1	0	0	0	1	错误寄存器	特征寄存器	8

续表

片选 1	片选 0	地址 2	地址 1	地址 0	读	写	位数
1	0	0	1	0	扇区数寄存器	扇区数寄存器	8
1	0	0	1	1	扇区号寄存器	扇区号寄存器	8
1	0	1	0	0	柱面号寄存器（低 8 位）	柱面号寄存器（低 8 位）	8
1	0	1	0	1	柱面号寄存器（高 8 位）	柱面号寄存器（高 8 位）	8
1	0	1	1	0	驱动器选择/磁头寄存器	驱动器选择/磁头寄存器	8
1	0	1	1	1	状态寄存器	命令寄存器	8
0	1	1	1	0	状态寄存器	设备控制器寄存器	8

IDE 硬盘的传输模式有以下 3 种。

- PIO（Programmed I/O）模式：PIO 模式是一种通过 CPU 执行 I/O 端口指令来进行数据读写的数据交换模式，是最早的硬盘数据传输模式，数据传输速率低下，CPU 占有率也很高。
- DMA（Driect Memory Access）模式：DMA 模式是一种不经过 CPU 而直接从内存存取数据的数据交换模式。PIO 模式下硬盘和内存之间的数据传输是由 CPU 来控制的；而在 DMA 模式下，CPU 只需向 DMA 控制器下达指令，让 DMA 控制器来处理数的传送，数据传送完毕再把信息反馈给 CPU，这样就在很大程度上减轻了 CPU 资源占有率。
- Ultra DMA（简称 UDMA）模式：它在包含了 DMA 模式的优点的基础上又增加了 CRC 校验技术，提高数据传输过程中的准确性，安全性得到保障。另外，在以往的硬盘数据传输模式下，一个时钟周期只传输一次数据，而在 UDMA 模式中逐渐应用了 Double Data Rate（双倍数据传输）技术，它在时钟的上升沿和下降沿各自进行一次数据传输，使数据传输速度成倍增长。

除了可以以 CHS（Cylinder、Head 和 Sector）的方式定位硬盘的扇区外，还可以用 LBA（逻辑块线性地址）的方式来定位，CHS 可以换算为 LBA。CHS 设计最多只允许 65536 个柱面、16 个磁头以及 255 个扇区/磁轨。这就将容量限制为 267386880 个扇区，即大约 137GB。

假设用 c 表示当前柱面号，h 表示当前磁头号，cs 表示起始柱面号，hs 表示起始磁头号，ss 表示起始扇区号，ps 表示每磁道有多少个扇区，ph 表示每柱面有多少个磁道（一般情况下，$cs = 0$、$hs = 0$、$ss = 1$、$ps = 63$、$ph = 255$），LBA 与 CHS 有如下对应关系：

```
lba=（c-cs）*ph*ps+（h-hs）*ps+（s-ss）
```

LBA 使得系统忽略硬盘的几何结构，交由驱动器来完成。系统不需要去查询 CHS 值，而只需要查询逻辑块地址（Logical Block Address，LBA），驱动器电子装置会找出要读或写的实际扇区。而 LBA48（48 位逻辑块地址）则可以使系统支持超过 137GB 的硬盘。

Linux 内核中，与 IDE 驱动相关的文件被放置在/drivers/ide 目录下，这个目录包含 ide.c、ide-cd.c、ide-cd.h、ide-disk.c、ide-dma.c、ide-floppy.c、ide-generic.c、ide-io.c、ide-iops.c、ide-lib.c、ide-pnp.c、ide-probe.c、ide-proc.c、ide-tape.c、ide-taskfile.c、ide-timing.h 文件以及针对 ARM、PPC、MIPS 等外围 IDE 设备驱动的目录。整个 IDE 设备驱动的体系结构很复杂，但驱动工程师要使 Linux 支持某嵌入式系统中的 IDE 硬盘，所需编写的代码量是非常少的。

华清远见

13.8.2 IDE 硬盘设备驱动的 block_device_operations 及成员函数

IDE 硬盘驱动的 block_device_operations 中包含了打开、释放、I/O 控制、获得几何信息、媒介改变和使介质有效的成员函数，这些函数的实现较简单，如代码清单 13.21 所示。

代码清单 13.21 IDE 硬盘驱动 block_device_operations 结构体及其成员函数

```
1  static struct block_device_operations idedisk_ops =
2  {
3   .owner      = THIS_MODULE,
4   .open       = idedisk_open,
5   .release    = idedisk_release,
6   .ioctl      = idedisk_ioctl,
7   .getgeo     = idedisk_getgeo, //得到几何信息
8   .media_changed  = idedisk_media_changed,  //媒介改变
9   .revalidate_disk= idedisk_revalidate_disk //使介质有效
10 };
11
12 static int idedisk_ioctl(struct inode *inode, struct file *file,
13          unsigned int cmd, unsigned long arg)
14 {
15  struct block_device *bdev = inode->i_bdev;
16  struct ide_disk_obj *idkp = ide_disk_g(bdev->bd_disk);
17  return generic_ide_ioctl(idkp->drive, file, bdev, cmd, arg);//通用 IDE 的 I/O 控制
18 }
19
20
21  static  int  idedisk_getgeo(struct  block_device  *bdev,  struct hd_geometry *geo)
22 {
23  struct ide_disk_obj *idkp = ide_disk_g(bdev->bd_disk);
24  ide_drive_t *drive = idkp->drive;
25  /* 得到几何信息，CHS */
26  geo->heads = drive->bios_head;
27  geo->sectors = drive->bios_sect;
28  geo->cylinders = (u16)drive->bios_cyl; /* truncate */
29  return 0;
30 }
31
32 static int idedisk_open(struct inode *inode, struct file *filp)
33 {
34  struct gendisk *disk = inode->i_bdev->bd_disk;
35  struct ide_disk_obj *idkp;
36  ide_drive_t *drive;
37
38  if (!(idkp = ide_disk_get(disk)))
39      return -ENXIO;
40
41  drive = idkp->drive;
42
43  drive->usage++;  //使用计数加 1
44  if (drive->removable && drive->usage == 1) {
45      ide_task_t args;
46      memset(&args, 0, sizeof(ide_task_t));
47      args.tfRegister[IDE_COMMAND_OFFSET] = WIN_DOORLOCK;
48      args.command_type = IDE_DRIVE_TASK_NO_DATA;
```

```
    49      args.handler      = &task_no_data_intr;
    50      check_disk_change(inode->i_bdev);
    51
    52      if  (drive->doorlocking  &&  ide_raw_taskfile(drive,  &args,
NULL))
    53          drive->doorlocking = 0;
    54  }
    55  return 0;
    56 }
    57
    58 static int idedisk_release(struct inode *inode, struct file *filp)
    59 {
    60  struct gendisk *disk = inode->i_bdev->bd_disk;
    61  struct ide_disk_obj *idkp = ide_disk_g(disk);
    62  ide_drive_t *drive = idkp->drive;
    63
    64  if (drive->usage == 1)
    65      ide_cacheflush_p(drive);
    66  if (drive->removable && drive->usage == 1) {
    67      ide_task_t args;
    68      memset(&args, 0, sizeof(ide_task_t));
    69      args.tfRegister[IDE_COMMAND_OFFSET] = WIN_DOORUNLOCK;
    70      args.command_type = IDE_DRIVE_TASK_NO_DATA;
    71      args.handler      = &task_no_data_intr;
    72      if  (drive->doorlocking  &&  ide_raw_taskfile(drive,  &args,
NULL))
    73          drive->doorlocking = 0;
    74  }
    75  drive->usage--;   //使用计数减 1
    76
    77  ide_disk_put(idkp);
    78  return 0;
    79 }
```

13.8.3 IDE 硬盘设备驱动的 I/O 请求处理

Linux 系统对 IDE 驱动进行了再封装，定义了 ide_driver_t 结构体，这个结构体容纳了 IDE 硬盘的探测、移除、请求处理和结束请求处理等函数指针。结束请求处理函数 ide_end_request()是对 end_request()函数针对 IDE 的修改。代码清单 13.21 所示为 ide_driver_t 结构体的定义。

代码清单 13.22　ide_driver_t 结构体

```
    1  static ide_driver_t idedisk_driver = {
    2   .gen_driver = {
    3       .owner      = THIS_MODULE,
    4       .name       = "ide-disk",
    5       .bus        = &ide_bus_type,
    6   },
    7   .probe          = ide_disk_probe,  //探测
    8   .remove         = ide_disk_remove,  //移除
    9   .shutdown       = ide_device_shutdown, //关闭
    10  .version        = IDEDISK_VERSION,
    11  .media          = ide_disk,  //媒介类型
```

```
12  .supports_dsc_overlap   = 0,
13  .do_request     = ide_do_rw_disk,  //请求处理函数
14  .end_request        = ide_end_request,  //请求处理结束
15  .error          = __ide_error,
16  .abort          = __ide_abort,
17  .proc           = idedisk_proc,
18 };
```

代码清单 13.22 第 13 行的 ide_do_rw_disk()函数完成硬盘 I/O 操作请求的处理，如代码清单 13.23 所示。

代码清单 13.23　IDE 硬盘驱动 I/O 请求处理

```
1   static ide_startstop_t ide_do_rw_disk(ide_drive_t *drive, struct
request *rq,
2     sector_t block)
3   {
4     ide_hwif_t *hwif = HWIF(drive);
5
6     BUG_ON(drive->blocked);
7
8     if (!blk_fs_request(rq)) //不是文件系统请求
9     {
10      blk_dump_rq_flags(rq, "ide_do_rw_disk - bad command");
11      ide_end_request(drive, 0, 0);  //以失败结束该请求
12      return ide_stopped;
13    }
14
15    pr_debug("%s: %sing: block=%llu, sectors=%lu, buffer=0x%08lx\n",
16      drive->name,rq_data_dir(rq) == READ ? "read" : "writ", (unsigned
long
17      long)block, rq->nr_sectors, (unsigned long)rq->buffer);
18
19    if (hwif->rw_disk)
20      hwif->rw_disk(drive, rq);
21
22    return __ide_do_rw_disk(drive, rq, block);  //具体的请求处理
23  }
24
25  static ide_startstop_t __ide_do_rw_disk(ide_drive_t *drive, struct
request *rq,
26    sector_t block)
27  {
28    ide_hwif_t *hwif = HWIF(drive);
29    unsigned int dma = drive->using_dma;
30    u8 lba48 = (drive->addressing == 1) ? 1 : 0;
31    task_ioreg_t command = WIN_NOP;
32    ata_nsector_t nsectors;
33
34    nsectors.all = (u16)rq->nr_sectors; //要传送的扇区数
35
36    if (hwif->no_lba48_dma && lba48 && dma)
37    {
38      if (block + rq->nr_sectors > 1ULL << 28)
39        dma = 0;
40      else
41        lba48 = 0;
42    }
43
```

```
    if (!dma)
    {
      ide_init_sg_cmd(drive, rq);
      ide_map_sg(drive, rq);
    }

    if (IDE_CONTROL_REG)
      hwif->OUTB(drive->ctl, IDE_CONTROL_REG);

    if (drive->select.b.lba)
    {
      if (lba48) //48 位 LBA
      {
        ...
      }
      else
      {
        //LBA 方式，写入要读写的位置信息到 IDE 寄存器
        hwif->OUTB(0x00, IDE_FEATURE_REG);
        hwif->OUTB(nsectors.b.low, IDE_NSECTOR_REG);
        hwif->OUTB(block, IDE_SECTOR_REG);
        hwif->OUTB(block >>= 8, IDE_LCYL_REG);
        hwif->OUTB(block >>= 8, IDE_HCYL_REG);
                        hwif->OUTB(((block   >>   8)   &0x0f)   |
drive->select.all,IDE_SELECT_REG);
      }
    }
    else
    {
      unsigned int sect, head, cyl, track;
      track = (int)block / drive->sect;
      sect = (int)block % drive->sect + 1;
      hwif->OUTB(sect, IDE_SECTOR_REG);
      head = track % drive->head;
      cyl = track / drive->head;

      pr_debug("%s: CHS=%u/%u/%u\n", drive->name, cyl, head, sect);
      //CHS 方式，写入要读写的位置信息到 IDE 寄存器
      hwif->OUTB(0x00, IDE_FEATURE_REG);
      hwif->OUTB(nsectors.b.low, IDE_NSECTOR_REG);
      hwif->OUTB(cyl, IDE_LCYL_REG);
      hwif->OUTB(cyl >> 8, IDE_HCYL_REG);
      hwif->OUTB(head | drive->select.all, IDE_SELECT_REG);
    }

    if (dma)  //DMA 方式
    {
      if (!hwif->dma_setup(drive))   //设置 DMA 成功
      {
        if (rq_data_dir(rq))
        {
          command = lba48 ? WIN_WRITEDMA_EXT : WIN_WRITEDMA;
          if (drive->vdma)
            command = lba48 ? WIN_WRITE_EXT : WIN_WRITE;
        }
```

```
98        else
99        {
100         command = lba48 ? WIN_READDMA_EXT : WIN_READDMA;
101         if (drive->vdma)
102           command = lba48 ? WIN_READ_EXT : WIN_READ;
103       }
104       hwif->dma_exec_cmd(drive, command);
105       hwif->dma_start(drive);
106       return ide_started;
107     }
108     /* 回到PIO模式 */
109     ide_init_sg_cmd(drive, rq);
110   }
111
112   if (rq_data_dir(rq) == READ) //数据传输方向是读
113   {
114     if (drive->mult_count)
115     {
116       hwif->data_phase = TASKFILE_MULTI_IN;
117       command = lba48 ? WIN_MULTREAD_EXT : WIN_MULTREAD;
118     }
119     else
120     {
121       hwif->data_phase = TASKFILE_IN;
122       command = lba48 ? WIN_READ_EXT : WIN_READ;
123     }
124     //执行读命令
125     ide_execute_command(drive, command, &task_in_intr, WAIT_CMD,
NULL);
126     return ide_started;
127   }
128   else  //数据传输方向是写
129   {
130     if (drive->mult_count)
131     {
132       hwif->data_phase = TASKFILE_MULTI_OUT;
133       command = lba48 ? WIN_MULTWRITE_EXT : WIN_MULTWRITE;
134     }
135     else
136     {
137       hwif->data_phase = TASKFILE_OUT;
138       command = lba48 ? WIN_WRITE_EXT : WIN_WRITE;
139     }
140
141     //写IDE命令寄存器/写入写命令
142     hwif->OUTB(command, IDE_COMMAND_REG);
143
144     return pre_task_out_intr(drive, rq);
145   }
146 }
```

从代码清单 13.23 可知，真正开始执行 I/O 操作的是其 22 行引用的__ide_do_rw_disk()函数。这个函数会根据不同的操作模式，将要读写的 LBA 或 CHS 信息写入 IDE 寄存器内，并给其命令寄存器写入读、写命令。

为了进行硬盘读写操作，第 61～67 行和第 80～85 行将参数写入地址寄存器和特性寄存器，如果是读，第 125 行调用的 ide_execute_command()会将读命令写入命令寄

存器；如果是写，第 142 行将写命令写入 IDE 命令寄存器 IDE_COMMAND_REG。

真正调用 ide_driver_t 结构体中 do_request()成员函数即 ide_do_rw_disk()的是 ide-io.c 文件中的 start_request()函数，这个函数会过滤掉一些请求，最终将读写 I/O 操作请求传递给 ide_do_rw_ disk()函数，如代码清单 13.24 所示。

代码清单 13.24　开始执行一个 IDE 请求的 start_request()函数

```
1  static ide_startstop_t start_request(ide_drive_t *drive, struct request *rq)
2  {
3    ide_startstop_t startstop;
4    sector_t block;
5
6    BUG_ON(!(rq->flags &REQ_STARTED));
7
8    /* 超过了最大失败次数 */
9            if (drive->max_failures && (drive->failures > drive->max_failures))
10   {
11     goto kill_rq;
12   }
13
14   block = rq->sector; //要传输的下一个扇区
15   if (blk_fs_request(rq) && (drive->media == ide_disk || drive->media ==
16     ide_floppy)) //是文件系统请求，是 IDE 盘
17   {
18     block += drive->sect0;
19   }
20   /* 如果将 0 扇区重映射到一扇区 */
21   if (block == 0 && drive->remap_0_to_1 == 1)
22     block = 1;
23
24   if (blk_pm_suspend_request(rq) && rq->pm->pm_step ==
25     ide_pm_state_start_suspend)
26     drive->blocked = 1;
27   else if (blk_pm_resume_request(rq) && rq->pm->pm_step ==
28     ide_pm_state_start_resume)
29   {
30     /* 醒来后的第一件事就是等待 BSY 位不忙 */
31     int rc;
32
33     rc = ide_wait_not_busy(HWIF(drive), 35000);
34     if (rc)
35           printk(KERN_WARNING "%s: bus not ready on wakeup\n", drive->name);
36     SELECT_DRIVE(drive);
37                                                HWIF(drive)->OUTB(8, HWIF(drive)->io_ports[IDE_CONTROL_OFFSET]);
38     rc = ide_wait_not_busy(HWIF(drive), 10000);
39     if (rc)
40          printk(KERN_WARNING "%s: drive not ready on wakeup\n", drive->name);
41   }
```

```
    42
    43   SELECT_DRIVE(drive);
    44   if (ide_wait_stat(&startstop, drive, drive->ready_stat, BUSY_STAT
| DRQ_STAT,
    45     WAIT_READY))  //等待驱动器 READY
    46   {
    47         printk(KERN_ERR  "%s:  drive  not  ready  for  command\n",
drive->name);
    48     return startstop;
    49   }
    50   if (!drive->special.all)
    51   {
    52     ide_driver_t *drv;
    53
    54     //其他非读写请求
    55     if (rq->flags &(REQ_DRIVE_CMD | REQ_DRIVE_TASK))
    56       return execute_drive_cmd(drive, rq);
    57     else if (rq->flags &REQ_DRIVE_TASKFILE)
    58       return execute_drive_cmd(drive, rq);
    59     else if (blk_pm_request(rq))
    60     {
    61       startstop = ide_start_power_step(drive, rq);
    62            if  (startstop  ==  ide_stopped  &&  rq->pm->pm_step  ==
ide_pm_state_completed)
    63         ide_complete_pm_request(drive, rq);
    64       return startstop;
    65     }
    66
    67     drv = *(ide_driver_t **)rq->rq_disk->private_data;
    68     return drv->do_request(drive, rq, block); //处理 I/O 操作请求
    69   }
    70   return do_special(drive);
    71   kill_rq: ide_kill_rq(drive, rq);
    72   return ide_stopped;
    73 }
```

IDE 硬盘驱动对 I/O 请求结束处理进行了针对 IDE 的整理，并填充在 ide_driver_t 结构体的 end_request 成员中，对应的函数为 ide_end_request()，如代码清单 13.25 所示。

代码清单 13.25　IDE I/O 请求结束处理

```
    1  /*   ide_end_request: 完成了一个 IDE 请求
    2   参数:  nr_sectors 被完成的扇区数量
    3  */
    4   int  ide_end_request  (ide_drive_t  *drive,  int  uptodate,  int
nr_sectors)
    5  {
    6   struct request *rq;
    7   unsigned long flags;
    8   int ret = 1;
    9
    10  spin_lock_irqsave(&ide_lock, flags);  //获得自旋锁
    11  rq = HWGROUP(drive)->rq;
    12
    13  if (!nr_sectors)
    14      nr_sectors = rq->hard_cur_sectors;
    15
    16  ret = __ide_end_request(drive, rq, uptodate, nr_sectors); //具体
```

```
的结束请求处理
    17
    18  spin_unlock_irqrestore(&ide_lock, flags);  //释放获得自旋锁
    19  return ret;
    20 }
    21
    22 static int __ide_end_request(ide_drive_t *drive, struct request *rq,
int
    23   uptodate, int nr_sectors)
    24 {
    25   int ret = 1;
    26
    27   BUG_ON(!(rq->flags &REQ_STARTED));
    28
    29   /*  如果对请求设置了 failfast，立即完成整个请求 */
    30   if (blk_noretry_request(rq) && end_io_error(uptodate))
    31     nr_sectors = rq->hard_nr_sectors;
    32
    33   if (!blk_fs_request(rq) && end_io_error(uptodate) && !rq->errors)
    34     rq->errors =  - EIO;
    35
    36   /* 决定是否再使能 DMA，如果 DMA 超过 3 次，采用 PIO 模式 */
    37   if (drive->state == DMA_PIO_RETRY && drive->retry_pio <= 3)
    38   {
    39     drive->state = 0;
    40     HWGROUP(drive)->hwif->ide_dma_on(drive);
    41   }
    42
    43   //结束请求
    44   if (!end_that_request_first(rq, uptodate, nr_sectors))
    45   {
    46     add_disk_randomness(rq->rq_disk);   //给系统的随机数池贡献熵
    47     blkdev_dequeue_request(rq);
    48     HWGROUP(drive)->rq = NULL;
    49     end_that_request_last(rq, uptodate);
    50     ret = 0;
    51   }
    52
    53   return ret;
    54 }
```

13.8.4 在内核中增加对新系统 IDE 设备的支持

尽管 IDE 设备的驱动非常复杂，但是由于其访问方式符合 ATA 标准，因而内核提供的 I/O 操作的代码是通用的，为了使内核能找到新系统中的 IDE 硬盘，工程师只需编写少量的针对特定硬件平台的底层代码。

使用 ide_register_hw()函数可注册 IDE 硬件接口，其原型为：

```
int ide_register_hw(hw_regs_t *hw, ide_hwif_t **hwifp);
```

这个函数接收的两个参数对应的数据结构为 hw_regs_s 和 ide_hwif_t， 其定义如代码清单 13.26 所示。

代码清单 13.26　hw_regs_s 和 ide_hwif_t 结构体的定义

```
1  typedef struct hw_regs_s
2  {
3    unsigned long io_ports[IDE_NR_PORTS]; /* task file 寄存器 */
4    int irq; /* 中断号 */
5    int dma; /* DMA 入口 */
6    ide_ack_intr_t *ack_intr; /* 确认中断 */
7    hwif_chipset_t chipset;
8    struct device *dev;
9  } hw_regs_t;
10
11 typedef struct hwif_s
12 {
13   ...
14
15   char name[6]; /* 接口名，如"ide0" */
16
17   hw_regs_t hw; /* 硬件信息 */
18   ide_drive_t drives[MAX_DRIVES]; /* 驱动器信息 */
19
20   u8 major; /* 主设备号 */
21   u8 index; /* 索引，如果为 0，则对应 ide0，如果为 1，则对应 ide1 */
22   ...
23   //DMA 操作
24   int(*dma_setup)(ide_drive_t*);
25   void(*dma_exec_cmd)(ide_drive_t *, u8);
26   void(*dma_start)(ide_drive_t*);
27   int(*ide_dma_end)(ide_drive_t *drive);
28   int(*ide_dma_check)(ide_drive_t *drive);
29   int(*ide_dma_on)(ide_drive_t *drive);
30   int(*ide_dma_off_quietly)(ide_drive_t *drive);
31   int(*ide_dma_test_irq)(ide_drive_t *drive);
32   int(*ide_dma_host_on)(ide_drive_t *drive);
33   int(*ide_dma_host_off)(ide_drive_t *drive);
34   int(*ide_dma_lostirq)(ide_drive_t *drive);
35   int(*ide_dma_timeout)(ide_drive_t *drive);
36   //寄存器访问
37   void(*OUTB)(u8 addr, unsigned long port);
38   void(*OUTBSYNC)(ide_drive_t *drive, u8 addr, unsigned long port);
39   void(*OUTW)(u16 addr, unsigned long port);
40   void(*OUTL)(u32 addr, unsigned long port);
41   void(*OUTSW)(unsigned long port, void *addr, u32 count);
42   void(*OUTSL)(unsigned long port, void *addr, u32 count);
43
44   u8(*INB)(unsigned long port);
45   u16(*INW)(unsigned long port);
46   u32(*INL)(unsigned long port);
47   void(*INSW)(unsigned long port, void *addr, u32 count);
48   void(*INSL)(unsigned long port, void *addr, u32 count);
49
50   ...
51 } ____cacheline_internodealigned_in_smp ide_hwif_t;
```

hw_regs_s 结构体描述了 IDE 接口的寄存器、用到的中断号和 DMA 入口，是对寄存器和硬件资源的描述，而 ide_hwif_t 是对 IDE 接口硬件访问方法的描述。

因此，为使得新系统支持 IDE 并被内核侦测到，工程师只需要初始化 hw_regs_s 和 ide_hwif_t 这两个结构体并使用 ide_register_hw()注册 IDE 接口即可。代码清单 13.27 所示为 H8/300 系列单片机 IDE 驱动的适配器注册代码。

代码清单 13.27　H8/300 系列单片机 IDE 接口注册

```
/* 寄存器操作函数 */
static void mm_outw(u16 d, unsigned long a)
{
__asm__("mov.b %w0,r2h\n\t"
    "mov.b %x0,r2l\n\t"
    "mov.w r2,@%1"
    :
    :"r"(d),"r"(a)
    :"er2");
}

static u16 mm_inw(unsigned long a)
{
register u16 r __asm__("er0");
__asm__("mov.w @%1,r2\n\t"
    "mov.b r2l,%x0\n\t"
    "mov.b r2h,%w0"
    :"=r"(r)
    :"r"(a)
    :"er2");
return r;
}

static void mm_outsw(unsigned long addr, void *buf, u32 len)
{
unsigned short *bp = (unsigned short *)buf;
for (; len > 0; len--, bp++)
    *(volatile u16 *)addr = bswap(*bp);
}

static void mm_insw(unsigned long addr, void *buf, u32 len)
{
unsigned short *bp = (unsigned short *)buf;
for (; len > 0; len--, bp++)
    *bp = bswap(*(volatile u16 *)addr);
}

#define H8300_IDE_GAP (2)

/* hw_regs_t 结构体初始化 */
static inline void hw_setup(hw_regs_t *hw)
```

```
{
int i;

memset(hw, 0, sizeof(hw_regs_t));
for (i = 0; i <= IDE_STATUS_OFFSET; i++)
    hw->io_ports[i] = CONFIG_H8300_IDE_BASE + H8300_IDE_GAP*i;
hw->io_ports[IDE_CONTROL_OFFSET] = CONFIG_H8300_IDE_ALT;
hw->irq = EXT_IRQ0 + CONFIG_H8300_IDE_IRQ;
hw->dma = NO_DMA;
hw->chipset = ide_generic;
}

/* ide_hwif_t 结构体初始化 */
static inline void hwif_setup(ide_hwif_t *hwif)
{
default_hwif_iops(hwif);

hwif->mmio  = 2;
hwif->OUTW  = mm_outw;
hwif->OUTSW = mm_outsw;
hwif->INW   = mm_inw;
hwif->INSW  = mm_insw;
hwif->OUTL  = NULL;
hwif->INL   = NULL;
hwif->OUTSL = NULL;
hwif->INSL  = NULL;
}

/* 注册 IDE 适配器 */
void __init h8300_ide_init(void)
{
hw_regs_t hw;
ide_hwif_t *hwif;
int idx;

  /* 申请内存区域 */
if   (!request_region(CONFIG_H8300_IDE_BASE,   H8300_IDE_GAP*8, "ide-h8300"))
    goto out_busy;
if    (!request_region(CONFIG_H8300_IDE_ALT,    H8300_IDE_GAP, "ide-h8300")) {
    release_region(CONFIG_H8300_IDE_BASE, H8300_IDE_GAP*8);
    goto out_busy;
}

```

```
85  hw_setup(&hw); //初始化 hw_regs_t
86
87  /* 注册 IDE 接口 */
88  idx = ide_register_hw(&hw, &hwif);
89  if (idx == -1) {
90      printk(KERN_ERR "ide-h8300: IDE I/F register failed\n");
91      return;
92  }
93
94  hwif_setup(hwif);   //设置 ide_hwif_t
95  printk(KERN_INFO "ide%d: H8/300 generic IDE interface\n", idx);
96  return;
97
98  out_busy:
99  printk(KERN_ERR "ide-h8300: IDE I/F resource already used.\n");
100 }
```

第 1～36 行定义了寄存器读写函数，第 41 行的 hw_setup()函数用于初始化 hw_regs_t 结构体，第 55 行的 hwif_setup()函数用于初始化 ide_hwif_t 结构体，第 71 行的 h8300_ide_init()函数中使用 ide_register_hw()注册了这个接口。

13.9 总结

块设备的 I/O 操作方式与字符设备存在较大的不同，因而引入了 request_queue、request、bio 等一系列数据结构。在整个块设备的 I/O 操作中，贯穿于始终的就是“请求”，字符设备的 I/O 操作则是直接进行不绕弯，块设备的 I/O 操作会排队和整合。

驱动的任务是处理请求，对请求的排队和整合由 I/O 调度算法解决，因此，块设备驱动的核心就是请求处理函数或“制造请求”函数。

尽管在块设备驱动中仍然存在 block_device_operations 结构体及其成员函数，但其不再包含读写一类的成员函数，而只是包含打开、释放及 I/O 控制等与具体读写无关的函数。

块设备驱动的结构相当复杂的，但幸运的是，块设备不像字符设备那么包罗万象，它通常就是存储设备，而且驱动的主体已经由 Linux 内核提供，针对一个特定的硬件系统，驱动工程师所涉及的工作往往只是编写少量的与硬件直接交互的代码。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

· 招生简章：http://www.embedu.org/courses/index.htm

第 10 章 中断与时钟

本章主要讲解 Linux 设备驱动编程中的中断与定时器处理。由于中断服务程序的执行并不存在于进程上下文，因此，要求中断服务程序的时间尽可能地短。因此，Linux 在中断处理中引入了顶半部和底半部分离的机制。另外，内核中对时钟的处理也采用中断方式，而内核软件定时器最终依赖于时钟中断。

10.1 节讲解中断和定时器的概念及处理流程。

10.2 节讲解 Linux 中断处理程序的架构，顶半部、底半部之间的关系。

10.3 节讲解 Linux 中断编程的方法，涉及申请和释放中断，禁止和使能中断，以及中断底半部 tasklet、工作队列、软中断机制等。

10.4 节讲解多个设备共享同一个中断号时的中断处理过程。

10.5 节和 10.6 节分别讲解 Linux 设备驱动编程中定时器的编程以及内核延时的方法。

华清远见

10.1 中断与定时器

所谓中断是指 CPU 在执行程序的过程中，出现了某些突发事件时 CPU 必须暂停执行当前的程序，转去处理突发事件，处理完毕后 CPU 又返回原程序被中断的位置并继续执行。

根据中断的来源，中断可分为内部中断和外部中断，内部中断的中断源来自 CPU 内部（软件中断指令、溢出、除法错误等，例如，操作系统从用户态切换到内核态需借助 CPU 内部的软件中断），外部中断的中断源来自 CPU 外部，由外设提出请求。

根据是否可以屏蔽中断分为可屏蔽中断与不屏蔽中断（NMI），可屏蔽中断可以通过屏蔽字被屏蔽，屏蔽后，该中断不再得到响应，而不屏蔽中断不能被屏蔽。

根据中断入口跳转方法的不同，中断分为向量中断和非向量中断。采用向量中断的 CPU 通常为不同的中断分配不同的中断号，当检测到某中断号的中断到来后，就自动跳转到与该中断号对应的地址执行。不同中断号的中断有不同的入口地址。非向量中断的多个中断共享一个入口地址，进入该入口地址后再通过软件判断中断标志来识别具体是哪个中断。也就是说，向量中断由硬件提供中断服务程序入口地址，非向量中断由软件提供中断服务程序入口地址。

一个典型的非向量中断服务程序如代码清单 10.1 所示，它先判断中断源，然后调用不同中断源的中断服务程序。

代码清单 10.1　非向量中断服务程序的典型结构

```
irq_handler()
{
  ...
  int int_src = read_int_status(); /*读硬件的中断相关寄存器*/
  switch (int_src) /*判断中断源*/
  {
    case DEV_A:
      dev_a_handler();
      break;
    case DEV_B:
      dev_b_handler();
      break;
    ...
    default:
      break;
  }
```

```
...
}
```

由于向量中断使用方便，目前有许多工程师创造了一些在非向量中断的处理器上模拟向量中断的方法。

嵌入式系统以及 X86 PC 中大多包含可编程中断控制器（PIC），许多 MCU 内部就集成了 PIC。如在 80386 中，PIC 是两片 i8259A 芯片的级联。通过读写 PIC 的寄存器，程序员可以屏蔽/使能某中断及获得中断状态，前者一般通过中断 MASK 寄存器完成，后者一般通过中断 PEND 寄存器完成。

定时器在硬件上也依赖中断来实现，图 10.1 给出了典型的嵌入式微处理内可编程间隔定时器（PIT）的工作原理，它接收一个时钟输入，当时钟脉冲到来时，将目前计数值增 1 并与预先设置的计数值（计数目标）比较，若相等，证明计数周期满，产生定时器中断并复位目前计数值。

图 10.1　定时器工作原理

10.2 Linux 中断处理程序架构

设备的中断会打断内核中进程的正常调度和运行，系统对更高吞吐率的追求势必要求中断服务程序尽可能地短小精悍。但是，这个良好的愿望往往与现实并不吻合。在大多数真实的系统中，当中断到来时，要完成的工作往往并不会是短小的，它可能要进行较大量的耗时处理。

图 10.2 描述了 Linux 内核的中断处理机制。为了在中断执行时间尽可能短和中断处理需完成大量工作之间找到一个平衡点，Linux 将中断处理程序分解为两个半部：顶半部（top half）和底半部（bottom half）。

图 10.2　Linux 中断处理机制

顶半部完成尽可能少的比较紧急的功能，它往往只是简单地读取寄存器中的中断状态并清除中断标志后就进行“登记中断”的工作。“登记中断”意味着将底半部处理程序挂到该设备的底半部执行队列中去。这样，顶半部执行的速度就会很快，可以服务更多的中断请求。

现在，中断处理工作的重心就落在了底半部的头上，它来完成中断事件的绝大多数任务。底半部几乎做了中断处理程序所有的事

情，而且可以被新的中断打断，这也是底半部和顶半部的最大不同，因为顶半部往往被设计成不可中断。底半部则相对来说并不是非常紧急的，而且相对比较耗时，不在硬件中断服务程序中执行。

尽管顶半部、底半部的结合能够改善系统的响应能力，但是，僵化地认为 Linux 设备驱动中的中断处理一定要分两个半部则是不对的。如果中断要处理的工作本身很少，则完全可以直接在顶半部全部完成。

其他操作系统中对中断的处理也采用了类似于 Linux 系统的方法，真正的硬件中断服务程序都应该尽可能短。因此，许多操作系统都提供了中断上下文和非中断上下文相结合的机制，将中断的耗时工作保留到非中断上下文去执行。例如，在 VxWorks 系统中，网络设备包接收中断到来后，中断服务程序会通过 netJobAdd()函数将耗时的包接收和上传工作交给 tNetTask 任务去执行。

在 Linux 系统中，查看/proc/interrupts 文件可以获得系统中断的统计信息，如下所示。在单处理器的系统中，第一列是中断号，第二列是向 CPU0 产生该中断的次数，之后的是对于中断的描述。

```
              CPU0
   0:       135253          XT-PIC          timer
   1:           22          XT-PIC          i8042
   2:            0          XT-PIC          cascade
   8:            1          XT-PIC          rtc
   10:         108          XT-PIC          etho
   11:        3707          XT-PIC          BusLogic BT-958
   12:         313          XT-PIC          i8042
   15:           4          XT-PIC          idel
NMI:             0
ERR              0
```

10.3 Linux 中断编程

10.3.1 申请和释放中断

在 Linux 设备驱动中，使用中断的设备需要申请和释放对应的中断，分别使用内核提供的 request_irq()和 free_irq()函数。

1. 申请 IRQ

```
int request_irq(unsigned int irq,
                void (*handler)(int irq, void *dev_id, struct pt_regs
*regs),
                unsigned long irqflags,
                const char * devname,
                void *dev_id);
```

irq 是要申请的硬件中断号。

handler 是向系统登记的中断处理函数，是一个回调函数，中断发生时，系统调用这个函数，dev_id 参数将被传递给它。

irqflags 是中断处理的属性，若设置了 SA_INTERRUPT，则表示中断处理程序是快速处理程序，快速处理程序被调用时屏蔽所有中断，慢速处理程序不屏蔽；若设置了 SA_SHIRQ，则表示多个设备共享中断，dev_id 在中断共享时会用到，一般设置为这个设备的设备结构体或者 NULL。

request_irq()返回 0 表示成功，返回-INVAL 表示中断号无效或处理函数指针为 NULL，返回-EBUSY 表示中断已经被占用且不能共享。

2．释放 IRQ

与 request_irq()向对应的函数为 free_irq()，free_irq()的原型如下：

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

free_irq()中参数的定义与 request_irq()相同。

华清远见

10.3.2 使能和屏蔽中断

下列 3 个函数用于屏蔽一个中断源。

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

disable_irq_nosync()与 disable_irq()的区别在于前者立即返回，而后者等待目前的中断处理完成。注意，这 3 个函数作用于可编程中断控制器，因此，对系统内的所有 CPU 都生效。

下列两个函数将屏蔽本 CPU 内的所有中断。

```
void local_irq_save(unsigned long flags);
void local_irq_disable(void);
```

前者会将目前的中断状态保留在 flags 中，注意 flags 被直接传递，而不是通过指针传递。后者直接禁止中断。

与上述两个禁止中断对应的恢复中断的方法如下：

```
void local_irq_restore(unsigned long flags);
void local_irq_enable(void);
```

以上各 local_开头的方法的作用范围是本 CPU 内。

10.3.3 底半部机制

Linux 系统实现底半部的机制主要有 tasklet、工作队列和软中断。

1. tasklet

tasklet 的使用较简单，我们只需要定义 tasklet 及其处理函数并将两者关联，例如：

```
void my_tasklet_func(unsigned long); /*定义一个处理函数*/
DECLARE_TASKLET(my_tasklet, my_tasklet_func, data);
  /*定义一个 tasklet 结构 my_tasklet，与 my_tasklet_func(data)函数相关联*/
```

代码 DECLARE_TASKLET（my_tasklet,my_tasklet_func,data）实现了定义名称为 my_tasklet 的 tasklet 并将其与 my_tasklet_func()这个函数绑定，而传入这个函数的参数为 data。

在需要调度 tasklet 的时候引用一个 tasklet_schedule()函数就能使系统在适当的时候进行调度运行，如下所示：

```
tasklet_schedule(&my_tasklet);
```

使用 tasklet 作为底半部处理中断的设备驱动程序模板如代码清单 10.2 所示（仅包含与中断相关的部分）。

代码清单 10.2 tasklet 使用模板

```
1  /*定义 tasklet 和底半部函数并关联*/
2  void xxx_do_tasklet(unsigned long);
3  DECLARE_TASKLET(xxx_tasklet, xxx_do_tasklet, 0);
4
```

```
5  /*中断处理底半部*/
6  void xxx_do_tasklet(unsigned long)
7  {
8   ...
9  }
10
11 /*中断处理顶半部*/
12 irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
13 {
14  ...
15  tasklet_schedule(&xxx_tasklet);
16  ...
17 }
18
19 /*设备驱动模块加载函数*/
20 int __init xxx_init(void)
21 {
22  ...
23  /*申请中断*/
24  result = request_irq(xxx_irq, xxx_interrupt,
25      SA_INTERRUPT, "xxx", NULL);
26  ...
27 }
28
29 /*设备驱动模块卸载函数*/
30 void __exit xxx_exit(void)
31 {
32  ...
33  /*释放中断*/
34  free_irq(xxx_irq, xxx_interrupt);
35  ...
36 }
```

上述程序在模块加载函数中申请中断（第 24～25 行），并在模块卸载函数中释放它（第 34 行）。对应于 xxx_irq 的中断处理程序被设置为 xxx_interrupt()函数，在这个函数中，第 15 行的 tasklet_schedule（&xxx_tasklet）调度的 tasklet 函数 xxx_do_tasklet()在适当的时候得到 执行。

上述代码第 12 行显示中断处理程序顶半部的返回类型为 irqreturn_t，它定义为 int，中断处理程序顶半部一般返回 IRQ_HANDLED。

2．工作队列

工作队列的使用方法和 tasklet 非常相似，下面的代码用于定义一个工作队列和一个底半部执行函数。

```
struct work_struct my_wq; /*定义一个工作队列*/
void my_wq_func(unsigned long); /*定义一个处理函数*/
```

通过 INIT_WORK()可以初始化这个工作队列并将工作队列与处理函数绑定，如下所示：

```
INIT_WORK(&my_wq, (void (*)(void *)) my_wq_func, NULL);
  /*初始化工作队列并将其与处理函数绑定*/
```

与 tasklet_schedule()对应的用于调度工作队列执行的函数为 schedule_work()，如：

```
schedule_work(&my_wq);/*调度工作队列执行*/
```

与代码清单 10.2 对应的使用工作队列处理中断底半部的设备驱动程序模板如代码清单 10.3 所示（仅包含与中断相关的部分）。

代码清单 10.3　工作队列使用模板

```
1  /*定义工作队列和关联函数*/
2  struct work_struct xxx_wq;
3  void xxx_do_work(unsigned long);
4
5  /*中断处理底半部*/
6  void xxx_do_work(unsigned long)
7  {
8   ...
9  }
10
11 /*中断处理顶半部*/
12 irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
13 {
14  ...
15  schedule_work(&xxx_wq);
16  ...
17 }
18
19 /*设备驱动模块加载函数*/
20 int xxx_init(void)
21 {
22  ...
23  /*申请中断*/
24  result = request_irq(xxx_irq, xxx_interrupt,
25      SA_INTERRUPT, "xxx", NULL);
26  ...
27  /*初始化工作队列*/
28  INIT_WORK(&xxx_wq, (void (*)(void *)) xxx_do_work, NULL);
29  ...
30 }
31
32 /*设备驱动模块卸载函数*/
33 void xxx_exit(void)
34 {
35  ...
36  /*释放中断*/
37  free_irq(xxx_irq, xxx_interrupt);
38  ...
39 }
```

与代码清单 10.2 不同的是，上述程序在设计驱动模块加载函数中增加了初始化工作队列的代码（第 28 行）。

尽管 Linux 专家们多建议在设备第一次打开时才申请设备的中断并在最后一次关闭时释放中断以尽量减少中断被这个设备占用的时间，但是，大多数情况下，为求省事，大多数驱动工程师还是将中断申请和释放的工作放在了设备驱动的模块加载和卸载函数中。

3．软中断

软中断是用软件方式模拟硬件中断的概念，实现宏观上的异步执行效果，tasklet 也是基于软中断实现的。

第 9 章异步通知所基于的信号也类似于中断，现在，总结一下硬中断、软中断和信号的区别：硬中断是外部设备对 CPU 的中断，软中断通常是硬中断服务程序对内核的中断，而信号则是由内核（或其他进程）对某个进程的中断。

在 Linux 内核中，用 softirq_action 结构体表征一个软中断，这个结构体中包含软中断处理函数指针和传递给该函数的参数。使用 open_softirq()函数可以注册软中断对应的处理函数，而 raise_softirq()函数可以触发一个软中断。

软中断和 tasklet 仍然运行于中断上下文，而工作队列则运行于进程上下文。因此，软中断和 tasklet 处理函数中不能睡眠，而工作队列处理函数中允许睡眠。

local_bh_disable()和 local_bh_enable()是内核中用于禁止和使能软中断和 tasklet 底半部机制的函数。

10.3.4 实例：S3C2410 实时钟中断

S3C2410 处理器内部集成了实时钟（RTC）模块，该模块能够在系统断电的情况下由后备电池供电继续工作，其主要功能相对于一个时钟，记录年、月、日、时、分、秒等。S3C2410 的 RTC 可产生两种中断：周期节拍（tick）中断和报警（alarm）中断，前者相当于一个周期性的定时器，后者相当于一个“闹钟”，它在预先设定的时间到来时产生中断。

S3C2410 实时钟设备驱动的 open()函数中，会申请它将要使用的中断，如代码清单 10.4 所示。

华清远见

代码清单 10.4 S3C2410 实时钟驱动的 open()函数

```
/*S3C2410 实时钟驱动的open()函数 */
static int s3c2410_rtc_open(void)
{
  int ret;
  /*申请alarm中断*/
  ret = request_irq(s3c2410_rtc_alarmno, s3c2410_rtc_alarmirq,
                    SA_INTERRUPT, "s3c2410-rtc alarm", NULL);
  /*中断号被占用*/
  if (ret)
    printk(KERN_ERR "IRQ%d already in use\n", s3c2410_rtc_alarmno);

  /*申请tick中断*/
    ret = request_irq(s3c2410_rtc_tickno, s3c2410_rtc_tickirq, SA_INTERRUPT,
    "s3c2410-rtc tick", NULL);
  /*中断号被占用*/
  if (ret)
  {
    printk(KERN_ERR "IRQ%d already in use\n", s3c2410_rtc_tickno);
    goto tick_err;
  }

```

```
22  return ret;
23
24  tick_err: free_irq(s3c2410_rtc_alarmno, NULL); /*释放 alarm 中断*/
25  return ret;
26 }
```

S3C2410 实时钟设备驱动的 release()函数中，会释放它将要使用的中断，如代码清单 10.5 所示。

代码清单 10.5　S3C2410 实时钟驱动的 release()函数

```
1 static void s3c2410_rtc_release(void)
2 {
3     s3c2410_rtc_setpie(0);
4     /* 释放中断 */
5     free_irq(s3c2410_rtc_alarmno, NULL);
6     free_irq(s3c2410_rtc_tickno, NULL);
7 }
```

S3C2410 实时钟驱动的中断处理比较简单，不需要分为上下两个半部，而只存在顶半部，如代码清单 10.6 所示。

代码清单 10.6　S3C2410 实时钟驱动的中断处理程序

```
1  /*中断处理*/
2  static irqreturn_t s3c2410_rtc_alarmirq(int irq, void *id, struct
pt_regs *r)
3  {
4   rtc_update(1, RTC_AF | RTC_IRQF);
5   return IRQ_HANDLED;
6  }
7
8  static irqreturn_t s3c2410_rtc_tickirq(int irq, void *id, struct
pt_regs *r)
9  {
10  rtc_update(1, RTC_PF | RTC_IRQF);
11  return IRQ_HANDLED;
12 }
```

上述代码中调用的 rtc_update()函数定义于 linux-2.6.16\arch\arm\common\Rtctime.c 文件中，被各种实时钟共享，如代码清单 10.7 所示。

代码清单 10.7　实时钟更新的 rtc_update()函数

```
1  void rtc_update(unsigned long num, unsigned long events)
2  {
3   spin_lock(&rtc_lock);
4   rtc_irq_data = (rtc_irq_data + (num << 8)) | events;
5   spin_unlock(&rtc_lock);
6
7   wake_up_interruptible(&rtc_wait);/*唤醒等待队列*/
8   kill_fasync(&rtc_async_queue, SIGIO, POLL_IN);/*释放信号*/
9  }
10 EXPORT_SYMBOL(rtc_update);
```

上述中断处理程序并没有底半部（或者说没有严格意义上的 tasklet、工作队列或软中断底半部），实际上，它只是唤醒一个等待队列 rtc_wait，而这个等待队列的唤醒也将导致一个阻塞的进程被执行（这个阻塞的进程可看做底半部）。现在我们看到，等待队列可以作为中断处理程序顶半部和进程同步的一种良好机制。但是，任何情况下，都不能在顶半部等待一个等待队列，而只能唤醒。

10.4 中断共享

多个设备共享一根硬件中断线的情况在实际的硬件系统中广泛存在，PCI 设备即是如此。Linux 2.6 支持这种中断共享。下面是中断共享的使用方法。

- 共享中断的多个设备在申请中断时都应该使用 SA_SHIRQ 标志，而且一个设备以 SA_SHIRQ 申请某中断成功的前提是之前申请该中断的所有设备也都以 SA_SHIRQ 标志申请该中断。
- 尽管内核模块可访问的全局地址都可以作为 request_irq（…, void *dev_id）的最后一个参数 dev_id，但是设备结构体指针是可传入的最佳参数。
- 在中断到来时，所有共享此中断的中断处理程序都会被执行，在中断处理程序顶半部中，应迅速地根据硬件寄存器中的信息比照传入的 dev_id 参数判断是否是本设备的中断，若不是，应迅速返回，如图 10.3 所示。

图 10.3 共享中断的处理

代码清单 10.8 给出了使用共享中断的设备驱动程序的模板（仅包含与共享中断机制相关的部分）。

代码清单 10.8 共享中断编程的模板

```
/*中断处理顶半部*/
irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
 ...
 int status = read_int_status();/*获知中断源*/
 if(!is_myint(dev_id,status))/*判断是否是本设备中断*/
 {
 return IRQ_NONE;  /*立即返回*/
 }
 ...
 return IRQ_HANDLED;
}

/*设备驱动模块加载函数*/
int xxx_init(void)
{
 ...
```

华清远见

```
18  /*申请共享中断*/
19  result = request_irq(sh_irq, xxx_interrupt,
20      SA_SHIRQ, "xxx", xxx_dev);
21  ...
22 }
23
24 /*设备驱动模块卸载函数*/
25 void xxx_exit(void)
26 {
27  ...
28  /*释放中断*/
29  free_irq(xxx_irq, xxx_interrupt);
30  ...
31 }
```

10.5 内核定时器

10.5.1 内核定时器编程

软件意义上的定时器最终依赖硬件定时器来实现，内核在时钟中断发生后检测各定时器是否到期，到期后的定时器处理函数将作为软中断在底半部执行。实质上，时钟中断处理程序执行 update_process_timers()函数，该函数调用 run_local_timers()函数，这个函数处理 TIMER_SOFTIRQ 软中断，运行当前处理器上到期的所有定时器。

在 Linux 设备驱动编程中，可以利用 Linux 内核中提供的一组函数和数据结构来完成定时触发工作或者完成某周期性的事务。这组函数和数据结构使得驱动工程师多数情况下不用关心具体的软件定时器究竟对应着怎样的内核和硬件行为。

Linux 内核所提供的用于操作定时器的数据结构和函数如下。

1．timer_list

在 Linux 内核中，timer_list 结构体的一个实例对应一个定时器，如代码清单 10.9 所示。

代码清单 10.9　timer_list 结构体

```
1 struct timer_list {
2    struct list_head entry; //定时器列表
3    unsigned long expires; //定时器到期时间
4    void (*function)(unsigned long); //定时器处理函数
5    unsigned long data; //作为参数被传入定时器处理函数
6    struct timer_base_s *base;
7 };
```

当定时器期满后，其中第 5 行的 function()成员将被执行，而第 4 行的 data 成员则是传入其中的参数，第 3 行的 expires 则是定时器到期的时间（jiffies）。

如下代码定义一个名为 my_timer 的定时器：

```
struct timer_list my_timer;
```

2．初始化定时器

```
void init_timer(struct timer_list * timer);
```

上述 init_timer()函数初始化 timer_list 的 entry 的 next 为 NULL，并给 base 指针赋值。

TIMER_INITIALIZER（_function, _expires, _data）宏用于赋值定时器结构体的 function、expires、data 和 base 成员，这个宏的定义如下所示：

```
#define TIMER_INITIALIZER(_function, _expires, _data) {        \
        .function = (_function),                               \
        .expires = (_expires),                                 \
        .data = (_data),                                       \
        .base = &__init_timer_base,                            \
    }
```

DEFINE_TIMER（_name, _function, _expires, _data）宏是定义并初始化定时器成员的“快捷方式”，这个宏定义如下所示：

```
#define DEFINE_TIMER(_name, _function, _expires, _data)        \
    struct timer_list _name =                                  \
        TIMER_INITIALIZER(_function, _expires, _data)
```

此外，setup_timer()也可用于初始化定时器并赋值其成员，其源代码如下：

```
static inline void setup_timer(struct timer_list * timer,
                    void (*function)(unsigned long),
                    unsigned long data)
{
     timer->function = function;
     timer->data = data;
     init_timer(timer);
}
```

3．增加定时器

```
void add_timer(struct timer_list * timer);
```

上述函数用于注册内核定时器，将定时器加入到内核动态定时器链表中。

4．删除定时器

```
int del_timer(struct timer_list * timer);
```

上述函数用于删除定时器。

del_timer_sync()是 del_timer()的同步版，主要在多处理器系统中使用，如果编译内核时不支持 SMP，del_timer_sync()和 del_timer()等价。

5．修改定时器的 expire

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

上述函数用于修改定时器的到期时间，在新的被传入的 expires 到来后才会执行定

时器函数。

代码清单 10.10 给出了一个完整的内核定时器使用模板，大多数情况下，设备驱动都如这个模板那样使用定时器。

代码清单 10.10　内核定时器使用模板

```
/*xxx 设备结构体*/
struct xxx_dev
{
 struct cdev cdev;
 ...
 timer_list xxx_timer;/*设备要使用的定时器*/
};

/*xxx 驱动中的某函数*/
xxx_func1(...)
{
 struct xxx_dev *dev = filp->private_data;
 ...
 /*初始化定时器*/
 init_timer(&dev->xxx_timer);
 dev->xxx_timer.function = &xxx_do_timer;
 dev->xxx_timer.data = (unsigned long)dev;
                        /*设备结构体指针作为定时器处理函数参数*/
 dev->xxx_timer.expires = jiffies + delay;
 /*添加（注册）定时器*/
 add_timer(&dev->xxx_timer);
 ...
}

/*xxx 驱动中的某函数*/
xxx_func2(…)
{
 ...
 /*删除定时器*/
 del_timer (&dev->xxx_timer);
 ...
}

/*定时器处理函数*/
static void xxx_do_timer(unsigned long arg)
```

```
36 {
37  struct xxx_device *dev = (struct xxx_device *)(arg);
38  ...
39  /*调度定时器再执行*/
40  dev->xxx_timer.expires = jiffies + delay;
41  add_timer(&dev->xxx_timer);
42  ...
43 }
```

从代码清单第 19、40 行可以看出，定时器的到期时间往往是在目前 jiffies 的基础上添加一个时延，若为 Hz，则表示延迟 1s。

在定时器处理函数中，在做完相应的工作后，往往会延后 expires 并将定时器再次添加到内核定时器链表，以便定时器能再次被触发。

10.5.2 实例：秒字符设备

下面我们编写一个字符设备“second”（即“秒”）的驱动，它在被打开的时候初始化一个定时器并将其添加到内核定时器链表，每秒输出一次当前的 jiffies（为此，定时器处理函数中每次都要修改新的 expires），整个程序如代码清单 10.11。

代码清单 10.11 使用内核定时器的 second 字符设备驱动

```
1   #include ...
2
3   #define SECOND_MAJOR 252    /*预设的second的主设备号*/
4
5   static int second_major = SECOND_MAJOR;
6
7   /*second设备结构体*/
8   struct second_dev
9   {
10    struct cdev cdev; /*cdev结构体*/
11    atomic_t counter;/* 一共经历了多少秒？ */
12    struct timer_list s_timer; /*设备要使用的定时器*/
13  };
14
15  struct second_dev *second_devp; /*设备结构体指针*/
16
17  /*定时器处理函数*/
18  static void second_timer_handle(unsigned long arg)
19  {
20    mod_timer(&second_devp->s_timer,jiffies + HZ);
21    atomic_inc(&second_devp->counter);
22
```

```
  printk(KERN_NOTICE "current jiffies is %ld\n", jiffies);
}

/*文件打开函数*/
int second_open(struct inode *inode, struct file *filp)
{
  /*初始化定时器*/
  init_timer(&second_devp->s_timer);
  second_devp->s_timer.function = &second_timer_handle;
  second_devp->s_timer.expires = jiffies + HZ;

  add_timer(&second_devp->s_timer); /*添加（注册）定时器*/

  atomic_set(&second_devp->counter,0); //计数清零

  return 0;
}
/*文件释放函数*/
int second_release(struct inode *inode, struct file *filp)
{
  del_timer(&second_devp->s_timer);

  return 0;
}

/*globalfifo 读函数*/
static ssize_t second_read(struct file *filp, char __user *buf, size_t count,
  loff_t *ppos)
{
  int counter;

  counter = atomic_read(&second_devp->counter);
  if(put_user(counter, (int*)buf))
    return - EFAULT;
  else
    return sizeof(unsigned int);
}

/*文件操作结构体*/
```

```
62 static const struct file_operations second_fops =
63 {
64   .owner = THIS_MODULE,
65   .open = second_open,
66   .release = second_release,
67   .read = second_read,
68 };
69
70 /*初始化并注册cdev*/
71 static void second_setup_cdev(struct second_dev *dev, int index)
72 {
73   int err, devno = MKDEV(second_major, index);
74
75   cdev_init(&dev->cdev, &second_fops);
76   dev->cdev.owner = THIS_MODULE;
77   dev->cdev.ops = &second_fops;
78   err = cdev_add(&dev->cdev, devno, 1);
79   if (err)
80     printk(KERN_NOTICE "Error %d adding LED%d", err, index);
81 }
82
83 /*设备驱动模块加载函数*/
84 int second_init(void)
85 {
86   int ret;
87   dev_t devno = MKDEV(second_major, 0);
88
89   /* 申请设备号*/
90   if (second_major)
91     ret = register_chrdev_region(devno, 1, "second");
92   else  /* 动态申请设备号 */
93   {
94     ret = alloc_chrdev_region(&devno, 0, 1, "second");
95     second_major = MAJOR(devno);
96   }
97   if (ret < 0)
98     return ret;
99   /* 动态申请设备结构体的内存*/
100  second_devp = kmalloc(sizeof(struct second_dev), GFP_KERNEL);
101  if (!second_devp)   /*申请失败*/
```

```
  {
    ret =  - ENOMEM;
    goto fail_malloc;
  }

  memset(second_devp, 0, sizeof(struct second_dev));

  second_setup_cdev(second_devp, 0);

  return 0;

  fail_malloc: unregister_chrdev_region(devno, 1);
}

/*模块卸载函数*/
void second_exit(void)
{
  cdev_del(&second_devp->cdev);   /*注销 cdev*/
  kfree(second_devp);    /*释放设备结构体内存*/
  unregister_chrdev_region(MKDEV(second_major, 0), 1); /*释放设备号*/
}

MODULE_AUTHOR("Song Baohua");
MODULE_LICENSE("Dual BSD/GPL");

module_param(second_major, int, S_IRUGO);

module_init(second_init);
module_exit(second_exit);
```

在 second 的 open()函数中，将启动定时器，此后每 1s 会再次运行定时器处理函数，在 second 的 release()函数中，定时器被删除。

second_dev 结构体中的原子变量 counter 用于秒计数，每次在定时器处理函数中将被 atomic_inc()调用原子的增 1，second 的 read()函数会将这个值返回给用户空间。

编译驱动，加载该内核模块并创建“/dev/second”设备文件结点后，使用代码清单 10.12 的应用程序打开“/dev/second”。

代码清单 10.12 的应用程序 second_test 会不断地读取自打开“/dev/second”设备文件以来经历的秒数。

代码清单 10.12　second 设备用户空间测试程序

```
1  #include ...
2
3  main()
4  {
5    int fd;
6    int counter = 0;
7    int old_counter = 0;
8
9    /*打开/dev/second 设备文件*/
10   fd = open("/dev/second", O_RDONLY);
11   if (fd !=  - 1)
12   {
13     while (1)
14     {
15       read(fd,&counter, sizeof(unsigned int));//读目前经历的秒数
16       if(counter!=old_counter)
17       {
18           printf("seconds after open /dev/second :%d\n",counter);
19           old_counter = counter;
20       }
21     }
22   }
23   else
24   {
25     printf("Device open failure\n");
26   }
27 }
```

运行 second_test 后，内核将不断地输出目前的 jiffies 值，如下所示：

```
current jiffies is 17216
current jiffies is 17316
current jiffies is 17416
current jiffies is 17516
current jiffies is 17616
current jiffies is 17716
current jiffies is 17816
current jiffies is 17916
current jiffies is 18016
current jiffies is 18116
current jiffies is 18216
current jiffies is 18316
```

而应用程序将不断输出自打开/dec/second 以来经历的秒数，如下所示：

```
[root@localhost driver_study]# ./second_test
seconds after open /dev/second :1
seconds after open /dev/second :2
seconds after open /dev/second :3
seconds after open /dev/second :4
seconds after open /dev/second :5
seconds after open /dev/second :6
seconds after open /dev/second :7
seconds after open /dev/second :8
seconds after open /dev/second :9
```

```
seconds after open /dev/second :10
```

10.6 内核延时

10.6.1 短延迟

Linux 内核中提供了如下 3 个函数分别进行纳秒、微秒和毫秒延迟。

```
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

上述延迟的实现原理本质上是忙等待，它根据 CPU 频率进行一定次数的循环。有时候，可以在软件中进行这样的延迟：

```
void delay(unsigned int time)
{
    while (time--);
}
```

ndelay()、udelay()和 mdelay()函数的实现方式机理与此类似。

毫秒时延（以及更大的秒时延）已经比较大了，在内核中，最好不要直接使用 mdelay()函数，这将无谓地耗费 CPU 资源，对于毫秒级以上时延，内核提供了下述函数：

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
```

上述函数将使得调用它的进程睡眠参数指定的时间，msleep()、ssleep()不能被打断，而 msleep_interruptible()则可以被打断。

受系统 HZ 以及进程调度的影响，msleep()类似函数的精度是有限的。

10.6.2 长延迟

内核中进行延迟的一个很直观的方法是比较当前的 jiffies 和目标 jiffies（设置为当前 jiffies 加上时间间隔的 jiffies），直到未来的 jiffies 达到目标 jiffies。代码清单 10.13 给出了使用忙等待先延迟 100 个 jiffies 再延迟 2s 的实例。

代码清单 10.13 忙等待时延实例

```
1 /*延迟100个jiffies*/
2 unsigned long delay = jiffies + 100;
3 while (time_before(jiffies, delay));
4
5 /*再延迟2s*/
6 unsigned long delay = jiffies + 2*HZ;
7 while (time_before(jiffies, delay));
```

与 time_before()对应的还有一个 time_after()，它们在内核中定义为（实际上只是将传入的未来时间 jiffies 和被调用时的 jiffies 进行一个简单的比较）：

```
#define time_after(a,b)            \
    (typecheck(unsigned long, a) &&    \
      typecheck(unsigned long, b) &&   \
      ((long)(b) - (long)(a) < 0))
#define time_before(a,b)   time_after(b,a)
```

为了防止 time_before()和 time_after()的比较过程中编译器对 jiffies 的优化，内核将其定义为 volatile 变量，这将保证它每次都被重新读取。

10.6.3 睡着延迟

睡着延迟无疑是比忙等待更好的方式，随着延迟在等待的时间到来之间进程处于睡眠状态，CPU 资源被其他进程使用。schedule_timeout()可以使当前任务睡眠指定的 jiffies 之后重新被调度执行，msleep()和 msleep_interruptible()在本质上都是依靠包含了 schedule_timeout() 的 schedule_ timeout_uninterruptible() 和 schedule_timeout_interruptible()实现的，如代码清单 10.14 所示。

代码清单 10.14 schedule_timeout()的使用

```
1  void msleep(unsigned int msecs)
2  {
3    unsigned long timeout = msecs_to_jiffies(msecs) + 1;
4
5    while (timeout)
6      timeout = schedule_timeout_uninterruptible(timeout);
7  }
8
9  unsigned long msleep_interruptible(unsigned int msecs)
10 {
11   unsigned long timeout = msecs_to_jiffies(msecs) + 1;
12
13   while (timeout && !signal_pending(current))
14     timeout = schedule_timeout_interruptible(timeout);
15   return jiffies_to_msecs(timeout);
16 }
```

实际上，schedule_timeout()的实现原理是向系统添加一个定时器，在定时器处理函数中唤醒参数对应的进程。

代码清单 10.14 第 6 行和第 14 行分别调用 schedule_timeout_uninterruptible()和 schedule_ timeout_interruptible()，这两个函数的区别在于前者在调用 schedule_timeout()之前置进程状态为 TASK_INTERRUPTIBLE，后者置进程状态为 TASK_UNINTERRUPTIBLE，如代码清单 10.15 所示。

代码清单 10.15 schedule_timeout_interruptible()和 schedule_timeout_interruptible()

```
signed long __sched schedule_timeout_interruptible(signed long timeout)
{
  __set_current_state(TASK_INTERRUPTIBLE);
  return schedule_timeout(timeout);
}

signed long __sched schedule_timeout_uninterruptible(signed long timeout)
{
  __set_current_state(TASK_UNINTERRUPTIBLE);
  return schedule_timeout(timeout);
}
```

另外，下面两个函数可以将当前进程添加到等待队列中，从而在等待队列上睡眠。当超时发生时，进程将被唤醒（后者可以在超时前被打断），如下所示：

```
sleep_on_timeout(wait_queue_head_t *q, unsigned long timeout);
interruptible_sleep_on_timeout(wait_queue_head_t*q, unsigned long timeout);
```

10.7 总结

Linux 的中断处理分为两个半部，顶半部处理紧急的硬件操作，底半部处理不紧急的耗时操作。tasklet 和工作队列都是调度中断底半部的良好机制，tasklet 基于软中断实现。内核定时器也依靠软中断实现。

内核中的延时是忙等待或者睡眠等待，为了充分利用 CPU 资源，使系统有更好的吞吐性能，在对延迟时间的要求并不是很精确的情况下，睡眠等待通常是值得推荐的。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm

嵌入式学院 华清远见旗下品牌

第 15 章　Linux 的 I2C 核心、总线与设备驱动

I^2C 总线仅仅使用 SCL、SDA 这两根信号线就实现了设备之间的数据交互，极大地简化了对硬件资源和 PCB 板布线空间的占用。因此，I^2C 总线被非常广泛地应用在 EEPROM、实时钟、小型 LCD 等设备与 CPU 的接口中。

Linux 系统定义了 I^2C 驱动体系结构，在 Linux 系统中，I^2C 驱动由 3 部分组成，即 I^2C 核心、I^2C 总线驱动和 I^2C 设备驱动。这 3 部分相互协作，形成了非常通用、可适应性很强的 I^2C 框架。

6.1 节对 Linux I^2C 体系结构进行分析，讲解 3 个组成部分各自的功能及相互联系。

6.2 节对 Linux I^2C 核心进行分析，讲解 i2c-core.c 文件的功能和主要函数的实现。

6.3 节、6.4 节分别详细介绍 I^2C 总线驱动和 I^2C 设备驱动的编写方法，给出可供参考的设计模板。

6.5 节、6.6 节以 6.3 节和 6.4 节给出的设计模板为基础，讲解 S3C2410 ARM 处理器 I^2C 总线驱动及挂接在其上的 SAA7113H 视频模拟/数字转换芯片设备驱动的编写方法。

15.1 Linux 的 I²C 体系结构

Linux 的 I²C 体系结构分为 3 个组成部分。

（1）I²C 核心。

I²C 核心提供了 I²C 总线驱动和设备驱动的注册、注销方法，I²C 通信方法（即“algorithm”，笔者认为直译为“运算方法”并不合适，为免引起误解，下文将直接使用“algorithm”）上层的、与具体适配器无关的代码以及探测设备、检测设备地址的上层代码等。

图 15.1　Linux I²C 体系结构

（2）I²C 总线驱动。

I²C 总线驱动是对 I²C 硬件体系结构中适配器端的实现，适配器可由 CPU 控制，甚至可以直接集成在 CPU 内部。

I²C 总线驱动主要包含了 I²C 适配器数据结构 i2c_adapter、I²C 适配器的 algorithm 数据结构 i2c_algorithm 和控制 I²C 适配器产生通信信号的函数。

经由 I²C 总线驱动的代码，我们可以控制 I²C 适配器以主控方式产生开始位、停止位、读写周期，以及以从设备方式被读写、产生 ACK 等。

（3）I²C 设备驱动。

I²C 设备驱动是对 I²C 硬件体系结构中设备端的实现，设备一般挂接在受 CPU 控制的 I²C 适配器上，通过 I²C 适配器与 CPU 交换数据。

I²C 设备驱动主要包含了数据结构 i2c_driver 和 i2c_client，我们需要根据具体设备实现其中的成员函数。

在 Linux 2.6 内核中，所有的 I²C 设备都在 sysfs 文件系统中显示，存于/sys/bus/i2c/目录，以适配器地址和芯片地址的形式列出，例如：

```
$ tree /sys/bus/i2c/
/sys/bus/i2c/
|-- devices
|   |-- 0-0048 -> ../../../devices/legacy/i2c-0/0-0048
|   |-- 0-0049 -> ../../../devices/legacy/i2c-0/0-0049
|   |-- 0-004a -> ../../../devices/legacy/i2c-0/0-004a
|   |-- 0-004b -> ../../../devices/legacy/i2c-0/0-004b
|   |-- 0-004c -> ../../../devices/legacy/i2c-0/0-004c
```

```
    |   |-- 0-004d -> ../../../devices/legacy/i2c-0/0-004d
    |   |-- 0-004e -> ../../../devices/legacy/i2c-0/0-004e
    |   '-- 0-004f -> ../../../devices/legacy/i2c-0/0-004f
    '-- drivers
        |-- i2c_adapter
        '-- lm75
            |-- 0-0048 -> ../../../../devices/legacy/i2c-0/0-0048
            |-- 0-0049 -> ../../../../devices/legacy/i2c-0/0-0049
            |-- 0-004a -> ../../../../devices/legacy/i2c-0/0-004a
            |-- 0-004b -> ../../../../devices/legacy/i2c-0/0-004b
            |-- 0-004c -> ../../../../devices/legacy/i2c-0/0-004c
            |-- 0-004d -> ../../../../devices/legacy/i2c-0/0-004d
            |-- 0-004e -> ../../../../devices/legacy/i2c-0/0-004e
            '-- 0-004f -> ../../../../devices/legacy/i2c-0/0-004f
```

在 Linux 内核源代码中的 drivers 目录下包含一个 i2c 目录，而在 i2c 目录下又包含如下文件和文件夹。

（1）i2c-core.c。

这个文件实现了 I^2C 核心的功能以及/proc/bus/i2c*接口。

（2）i2c-dev.c。

实现了 I^2C 适配器设备文件的功能，每一个 I^2C 适配器都被分配一个设备。通过适配器访问设备时的主设备号都为 89，次设备号为 0～255。应用程序通过“i2c-%d”（i2c-0, i2c-1,…, i2c-10,…）文件名并使用文件操作接口 open()、write()、read()、ioctl()和 close()等来访问这个设备。

i2c-dev.c 并没有针对特定的设备而设计，只是提供了通用的 read()、write()和 ioctl()等接口，应用层可以借用这些接口访问挂接在适配器上的 I^2C 设备的存储空间或寄存器，并控制 I^2C 设备的工作方式。

（3）chips 文件夹。

这个目录中包含了一些特定的 I^2C 设备驱动，如 Dallas 公司的 DS1337 实时钟芯片、EPSON 公司的 RTC8564 实时钟芯片和 I^2C 接口的 EEPROM 驱动等。

（4）busses 文件夹。

这个文件中包含了一些 I^2C 总线的驱动，如 S3C2410 的 I^2C 控制器驱动为 i2c-s3c2410.c。

（5）algos 文件夹。

实现了一些 I^2C 总线适配器的 algorithm。

此外，内核中的 i2c.h 这个头文件对 i2c_driver、i2c_client、i2c_adapter 和 i2c_algorithm 这 4 个数据结构进行了定义。理解这 4 个结构体的作用十分关键，代码清单 15.1、15.2、15.3、15.4 分别给出了它们的定义。

代码清单 15.1　i2c_adapter 结构体

```
1  struct i2c_adapter {
2   struct module *owner;/*所属模块*/
3   unsigned int id;   /*algorithm 的类型，定义于 i2c-id.h，以 I2C_ALGO_
开始*/
4   unsigned int class;
```

```
5   struct i2c_algorithm *algo;/*总线通信方法结构体指针    */
6   void *algo_data;    /* algorithm 数据 */
7   int (*client_register)(struct i2c_client *);  /*client 注册时调用*/
8   int (*client_unregister)(struct i2c_client *); /*client 注销时调用
*/
9   struct semaphore bus_lock;    /*控制并发访问的自旋锁*/
10  struct semaphore clist_lock;
11  int timeout;
12  int retries;    /*重试次数*/
13  struct device dev;      /* 适配器设备 */
14  struct class_device class_dev; /* 类设备 */
15  int nr;
16  struct list_head clients;  /* client 链表头*/
17  struct list_head list;
18  char name[I2C_NAME_SIZE];  /*适配器名称*/
19  struct completion dev_released;    /*用于同步*/
20  struct completion class_dev_released;
21};
```

代码清单 15.2　i2c_algorithm 结构体

```
1  struct i2c_algorithm {
2   int (*master_xfer)(struct i2c_adapter *adap,struct i2c_msg *msgs,
3                    int num);  /*i2c 传输函数指针*/
4   int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr,   /*smbus
传输函数指针*/
5                    unsigned short flags, char read_write,
6                    u8 command, int size, union i2c_smbus_data * data);
7   int (*slave_send)(struct i2c_adapter *,char*,int);/*当 I2C 适配器为 slave
时，发送函数*/
8   int (*slave_recv)(struct i2c_adapter *,char*,int); /*当 I2C 适配器为 slave
时，接收函数*/
9   int (*algo_control)(struct i2c_adapter *, unsigned int, unsigned long);
/*类似 ioctl*/
10  u32 (*functionality) (struct i2c_adapter *);/*返回适配器支持的功能*/
11 };
```

上述代码第 4 行对应为 SMBus 传输函数指针，SMBus 大部分基于 I^2C 总线规范，SMBus 不需增加额外引脚。与 I^2C 总线相比，SMBus 增加了一些新的功能特性，在访问时序也有一定的差异。

代码清单 15.3　i2c_driver 结构体

```
1  struct i2c_driver {
2   int id;
3   unsigned int class;
4   int (*attach_adapter)(struct i2c_adapter *);   /*依附 i2c_adapter
函数指针 */
5   int (*detach_adapter)(struct i2c_adapter *);   /*脱离 i2c_adapter
函数指针*/
6   int (*detach_client)(struct i2c_client *);         /*i2c  client
脱离函数指针*/
7   int (*command)(struct i2c_client *client,unsigned int cmd, void *arg);
/*类似 ioctl*/
```

```
8   struct device_driver driver;        /*设备驱动结构体*/
9   struct list_head list;              /*链表头*/
10 };
```

代码清单 15.4　i2c_client 结构体

```
1  struct i2c_client {
2   unsigned int flags;         /* 标志 */
3   unsigned short addr;            /* 低 7 位为芯片地址 */
4   struct i2c_adapter *adapter;    /*依附的 i2c_adapter*/
5   struct i2c_driver *driver;      /*依附的 i2c_driver */
6   int usage_count;                /* 访问计数  */
7   struct device dev;          /* 设备结构体 */
8   struct list_head list;          /* 链表头 */
9   char name[I2C_NAME_SIZE];       /* 设备名称 */
10  struct completion released;     /* 用于同步 */
11 };
```

下面分析 i2c_driver、i2c_client、i2c_adapter 和 i2c_algorithm 这 4 个数据结构的作用及其盘根错节的关系。

（1）i2c_adapter 与 i2c_algorithm。

i2c_adapter 对应于物理上的一个适配器，而 i2c_algorithm 对应一套通信方法。一个 I^2C 适配器需要 i2c_algorithm 中提供的通信函数来控制适配器上产生特定的访问周期。缺少 i2c_algorithm 的 i2c_adapter 什么也做不了，因此 i2c_adapter 中包含其使用的 i2c_algorithm 的指针。

i2c_algorithm 中的关键函数 master_xfer()用于产生 I^2C 访问周期需要的信号，以 i2c_msg（即 I^2C 消息）为单位。i2c_msg 结构体也非常关键，代码清单 15.5 给出了它的定义。

代码清单 15.5　i2c_msg 结构体

```
1 struct i2c_msg {
2   __u16 addr;     /* 设备地址*/
3   __u16 flags;        /* 标志 */
4   __u16 len;      /* 消息长度*/
5   __u8 *buf;      /* 消息数据*/
6 };
```

（2）i2c_driver 与 i2c_client。

i2c_driver 对应一套驱动方法，是纯粹的用于辅助作用的数据结构，它不对应于任何的物理实体。i2c_client 对应于真实的物理设备，每个 I^2C 设备都需要一个 i2c_client 来描述。i2c_client 一般被包含在 I^2C 字符设备的私有信息结构体中。

i2c_driver 与 i2c_client 发生关联的时刻在 i2c_driver 的 attach_adapter()函数被运行时。attach_adapter()会探测物理设备，当确定一个 client 存在时，把该 client 使用的 i2c_client 数据结构的 adapter 指针指向对应的 i2c_adapter，driver 指针指向该 i2c_driver，并会调用 i2c_adapter 的 client_register()函数。相反的过程发生在 i2c_driver 的 detach_client()函数被调用的时候。

（3）i2c_adpater 与 i2c_client。

华清远见

i2c_adpater 与 i2c_client 的关系与 I^2C 硬件体系中适配器和设备的关系一致，即 i2c_client 依附于 i2c_adpater。由于一个适配器上可以连接多个 I^2C 设备，所以一个 i2c_adpater 也可以被多个 i2c_client 依附，i2c_adpater 中包括依附于它的 i2c_client 的链表。

假设 I^2C 总线适配器 xxx 上有两个使用相同驱动程序的 yyy I^2C 设备，在打开该 I^2C 总线的设备结点后相关数据结构之间的逻辑组织关系将如图 15.2 所示。

从上面的分析可知，虽然 I^2C 硬件体系结构比较简单，但是 I^2C 体系结构在 Linux 中的实现却相当复杂。当工程师拿到实际的电路板，面对复杂的 Linux I^2C 子系统，应该如何下手写驱动呢？究竟有哪些是需要亲自做的，哪些是内核已经提供的呢？理清这个问题非常有意义，可以使我们面对具体问题时迅速地抓住重点。

一方面，适配器驱动可能是 Linux 内核本身还不包含的；另一方面，挂接在适配器上的具体设备驱动可能也是 Linux 内核还不包含的。即便上述设备驱动都存在于 Linux 内核中，其基于的平台也可能与我们的电路板不一样。因此，工程师要实现的主要工作如下。

- 提供 I^2C 适配器的硬件驱动，探测、初始化 I^2C 适配器（如申请 I^2C 的 I/O 地址和中断号）、驱动 CPU 控制的 I^2C 适配器从硬件上产生各种信号以及处理 I^2C 中断等。
- 提供 I^2C 适配器的 algorithm，用具体适配器的 xxx_xfer()函数填充 i2c_algorithm 的 master_xfer 指针，并把 i2c_algorithm 指针赋值给 i2c_adapter 的 algo 指针。
- 实现 I^2C 设备驱动与 i2c_driver 接口，用具体设备 yyy 的 yyy_attach_adapter()函数指针、yyy_detach_client()函数指针和 yyy_command()函数指针的赋值给 i2c_driver 的 attach_ adapter、detach_adapter 和 detach_client 指针。
- 实现 I^2C 设备驱动的文件操作接口，即实现具体设备 yyy 的 yyy_read()、yyy_write()和 yyy_ioctl()函数等。

华清远见

图 15.2　I²C 驱动的各数据结构的关系

上述工作中前两个属于 I²C 总线驱动，后两个属于 I²C 设备驱动，做完这些工作，系统会增加两个内核模块。15.3～15.4 节将详细分析这些工作的实施方法，给出设计模板，而 15.5～15.6 节将给出两个具体的实例。

15.2 Linux I²C 核心

I²C 核心（drivers/i2c/i2c-core.c）中提供了一组不依赖于硬件平台的接口函数，这个文件一般不需要被工程师修改，但是理解其中的主要函数非常关键，因为 I²C 总线驱动和设备驱动之间依赖于 I²C 核心作为纽带。I²C 核心中的主要函数如下。

（1）增加/删除 i2c_adapter。

```
int i2c_add_adapter(struct i2c_adapter *adap);
int i2c_del_adapter(struct i2c_adapter *adap);
```

（2）增加/删除 i2c_driver。

```
int i2c_register_driver(struct module *owner, struct i2c_driver *driver);
int i2c_del_driver(struct i2c_driver *driver);
inline int i2c_add_driver(struct i2c_driver *driver);
```

（3）i2c_client 依附/脱离。

```
int i2c_attach_client(struct i2c_client *client);
int i2c_detach_client(struct i2c_client *client);
```

当一个具体的 client 被侦测到并被关联的时候，设备和 sysfs 文件将被注册。相反地，在 client 被取消关联的时候，sysfs 文件和设备也被注销，如代码清单 15.6 所示。

代码清单 15.6 I²C 核心的 client attach/detach 函数

```
int i2c_attach_client(struct i2c_client *client)
{
  ...
 device_register(&client->dev);
 device_create_file(&client->dev, &dev_attr_client_name);

 return 0;
}

int i2c_detach_client(struct i2c_client *client)
{
  ...
 device_remove_file(&client->dev, &dev_attr_client_name);
 device_unregister(&client->dev);
  ...
}
```

（4）I²C 传输、发送和接收。

```
int i2c_transfer(struct i2c_adapter * adap, struct i2c_msg *msgs, int num);
int i2c_master_send(struct i2c_client *client,const char *buf ,int
```

```
count);
    int i2c_master_recv(struct i2c_client *client, char *buf ,int count);
```

i2c_transfer()函数用于进行 I^2C 适配器和 I^2C 设备之间的一组消息交互，i2c_master_send()函数和 i2c_master_recv()函数内部会调用 i2c_transfer()函数分别完成一条写消息和一条读消息，如代码清单 15.7、代码清单 15.8 所示。

华清远见

代码清单 15.7　I^2C 核心的 i2c_master_send 函数

```
int i2c_master_send(struct i2c_client *client,const char *buf ,int count)
{
 int ret;
 struct i2c_adapter *adap=client->adapter;
 struct i2c_msg msg;
  /*构造一个写消息*/
 msg.addr = client->addr;
 msg.flags = client->flags & I2C_M_TEN;
 msg.len = count;
 msg.buf = (char *)buf;
 /*传输消息*/
 ret = i2c_transfer(adap, &msg, 1);

 return (ret == 1) ? count : ret;
}
```

代码清单 15.8　I^2C 核心的 i2c_master_recv 函数

```
int i2c_master_recv(struct i2c_client *client, char *buf ,int count)
{
 struct i2c_adapter *adap=client->adapter;
 struct i2c_msg msg;
 int ret;
 /*构造一个读消息*/
 msg.addr = client->addr;
 msg.flags = client->flags & I2C_M_TEN;
 msg.flags |= I2C_M_RD;
 msg.len = count;
 msg.buf = buf;
 /*传输消息*/
 ret = i2c_transfer(adap, &msg, 1);

 /* 成功（1 条消息被处理）， 返回读的字节数 */
 return (ret == 1) ? count : ret;
}
```

i2c_transfer()函数本身不具备驱动适配器物理硬件完成消息交互的能力，它只是寻找到 i2c_adapter 对应的 i2c_algorithm，并使用 i2c_algorithm 的 master_xfer()函数真正

驱动硬件流程，如代码清单 15.9 所示。

代码清单 15.9　I^2C 核心的 i2c_transfer 函数

```
int i2c_transfer(struct i2c_adapter * adap, struct i2c_msg *msgs, int num)
{
 int ret;

 if (adap->algo->master_xfer) {
     down(&adap->bus_lock);
     ret = adap->algo->master_xfer(adap,msgs,num); /* 消息传输 */
     up(&adap->bus_lock);
     return ret;
 } else {
     dev_dbg(&adap->dev, "I2C level transfers not supported\n");
     return -ENOSYS;
 }
}
```

（5）I^2C 控制命令分配。

下面函数有助于将发给 I^2C 适配器设备文件 ioctl 的命令分配给对应适配器的 algorithm 的 algo_control()函数或 i2c_driver 的 command()函数，如下所示：

```
int i2c_control(struct i2c_client *client, unsigned int cmd, unsigned long arg);
void i2c_clients_command(struct i2c_adapter *adap, unsigned int cmd, void *arg);
```

15.3 Linux I^2C 总线驱动

15.3.1　I^2C 适配器驱动加载与卸载

I^2C 总线驱动模块的加载函数要完成两个工作。

- 初始化 I^2C 适配器所使用的硬件资源，如申请 I/O 地址、中断号等。
- 通过 i2c_add_adapter()添加 i2c_adapter 的数据结构，当然这个 i2c_adapter 数据结构的成员已经被 xxx 适配器的相应函数指针所初始化。

I^2C 总线驱动模块的卸载函数要完成的工作与加载函数相反。

- 释放 I^2C 适配器所使用的硬件资源，如释放 I/O 地址、中断号等。
- 通过 i2c_del_adapter()删除 i2c_adapter 的数据结构。

代码清单 15.10 所示为 I^2C 适配器驱动的模块加载和卸载函数的模板。

代码清单 15.10　I²C 总线驱动的模块加载和卸载函数模板

```
1  static int __init i2c_adapter_xxx_init(void)
2  {
3    xxx_adpater_hw_init();
4    i2c_add_adapter(&xxx_adapter);
5  }
6
7  static void __exit i2c_adapter_xxx_exit(void)
8  {
9    xxx_adpater_hw_free();
10   i2c_del_adapter(&xxx_adapter);
11 }
```

上述代码中 xxx_adpater_hw_init()和 xxx_adpater_hw_free()函数的实现都与具体的 CPU 和 I²C 设备硬件直接相关。

15.3.2　I²C 总线通信方法

我们需要为特定的 I²C 适配器实现其通信方法，主要实现 i2c_algorithm 的 master_xfer()函数和 functionality()函数。

functionality()函数非常简单，用于返回 algorithm 所支持的通信协议，如 I2C_FUNC_I2C、I2C_FUNC_10BIT_ADDR、I2C_FUNC_SMBUS_READ_BYTE、I2C_FUNC_SMBUS_WRITE_BYTE 等。

master_xfer()函数在 I²C 适配器上完成传递给它的 i2c_msg 数组中的每个 I²C 消息，代码清单 15.11 所示为 xxx 设备的 master_xfer()函数模板。

代码清单 15.11　I²C 总线驱动 master_xfer 函数模板

```
1  static int i2c_adapter_xxx_xfer(struct i2c_adapter *adap, struct
i2c_msg *msgs,
2    int num)
3  {
4    ...
5    for (i = 0; i < num; i++)
6    {
7      i2c_adapter_xxx_start(); /*产生开始位*/
8      /*是读消息*/
9      if (msgs[i]->flags &I2C_M_RD)
10     {
11       i2c_adapter_xxx_setaddr((msg->addr << 1) | 1); /*发送从设备读地
址*/
12       i2c_adapter_xxx_wait_ack(); /*获得从设备的 ack*/
13       i2c_adapter_xxx_readbytes(msgs[i]->buf, msgs[i]->len); /*读取
msgs[i] ->len
14         长的数据到 msgs[i]->buf*/
15     }
16     else
```

```
    /*是写消息*/
   {
     i2c_adapter_xxx_setaddr(msg->addr << 1); /*发送从设备写地址*/
     i2c_adapter_xxx_wait_ack(); /*获得从设备的 ack*/
     i2c_adapter_xxx_writebytes(msgs[i]->buf, msgs[i]->len); /*读取 msgs[i] ->len
       长的数据到 msgs[i]->buf*/
   }
  }
  i2c_adapter_xxx_stop(); /*产生停止位*/
}
```

上述代码实际上给出了一个 master_xfer()函数处理 I^2C 消息数组的流程，对于数组中的每个消息，判断消息类型，若为读消息，则赋从设备地址为（msg->addr << 1）|1，否则为 msg->addr << 1。对每个消息产生一个开始位，紧接着传送从设备地址，然后开始数据的发送或接收，对最后的消息还需产生一个停止位。如图 15.3 所示为整个 master_xfer()完成的时序。

图 15.3 algorithm 中 master_xfer 的时序

master_xfer()函数模板中的 i2c_adapter_xxx_start()、i2c_adapter_xxx_setaddr()、i2c_adapter_ xxx_wait_ack()、i2c_adapter_xxx_readbytes()、i2c_adapter_xxx_writebytes()和 i2c_adapter_xxx_stop()函数用于完成适配器的底层硬件操作，与 I^2C 适配器和 CPU 的具体硬件直接相关，需要由工程师根据芯片的数据手册来实现。

i2c_adapter_xxx_readbytes() 用于从从设备上接收一串数据，i2c_adapter_xxx_writebytes()用于向从设备写入一串数据，这两个函数的内部也会涉及 I^2C 总线协议中的 ACK 应答。

master_xfer()函数的实现在形式上会很多样，即便是 Linux 内核源代码中已经给出的一些 I^2C 总线驱动的 master_xfer()函数，由于由不同的组织或个人完成，风格上的差别也非常大，不一定能与模板完全对应，如 master_xfer()函数模板给出的消息处理是顺序进行的，而有的驱动以中断方式来完成这个流程（15.5 节的实例即是如此）。不管具体怎么实施，流程的本质都是不变的。因为这个流程不以驱动工程师的意志为转移，最终由 I^2C 总线硬件上的通信协议决定。

多数 I^2C 总线驱动会定义一个 xxx_i2c 结构体，作为 i2c_adapter 的 algo_data（类似“私有数据”），其中包含 I^2C 消息数组指针、数组索引及 I^2C 适配器 algorithm 访问控制用的自旋锁、等待队列等，而 master_xfer()函数完成消息数组中消息的处理也可通过对 xxx_i2c 结构体相关成员的访问来控制。代码清单 15.12 所示为 xxx_i2c 结构体的定义，与图 15.2 中的 xxx_i2c 是对应的。

代码清单 15.12　xxx_i2c 结构体模板

```
struct xxx_i2c
{
 spinlock_t      lock;
 wait_queue_head_t   wait;
 struct i2c_msg  *msg;
 unsigned int        msg_num;
 unsigned int        msg_idx;
 unsigned int        msg_ptr;
 ...
 struct i2c_adapter  adap;
};
```

15.4 Linux I²C 设备驱动

I²C 设备驱动要使用 i2c_driver 和 i2c_client 数据结构并填充其中的成员函数。i2c_client 一般被包含在设备的私有信息结构体 yyy_data 中，而 i2c_driver 则适合被定义为全局变量并初始化，代码清单 15.13 所示为已被初始化的 i2c_driver。

代码清单 15.13　已被初始化的 i2c_driver

```
static struct i2c_driver yyy_driver =
{
  .driver =
  {
    .name = "yyy",
  } ,
  .attach_adapter = yyy_attach_adapter,
  .detach_client =  yyy_detach_client,
  .command = yyy_command,
};
```

15.4.1 Linux I²C 设备驱动的模块加载与卸载

I²C 设备驱动的模块加载函数通用的方法是在 I²C 设备驱动的模块加载函数中完成两件事。

（1）通过 register_chrdev()函数将 I²C 设备注册为一个字符设备。

（2）通过 I²C 核心的 i2c_add_driver()函数添加 i2c_driver。

在模块卸载函数中需要做相反的两件事。

（1）通过 I²C 核心的 i2c_del_driver()函数删除 i2c_driver。

（2）通过 unregister_chrdev()函数注销字符设备。

代码清单 15.14 所示为 I²C 设备驱动的加载与卸载函数模板。

代码清单 15.14　I²C 设备驱动的模块加载与卸载函数模板

```
1  static int __init yyy_init(void)
2  {
3    int res;
4    /*注册字符设备*/
5    res = register_chrdev(YYY_MAJOR, "yyy", &yyy_fops); //老内核接口
6    if (res)
7      goto out;
8    /*添加 i2c_driver*/
9    res = i2c_add_driver(&yyy_driver);
10   if (res)
11     goto out_unreg_class;
12   return 0;
13
14   out_unreg_chrdev: unregister_chrdev(I2C_MAJOR, "i2c");
15   out: printk(KERN_ERR "%s: Driver Initialisation failed\n", __FILE_
_);
16   return res;
17 }
18
19 static void __exit yyy_exit(void)
20 {
21   i2c_del_driver(&i2cdev_driver);
22   unregister_chrdev(YYY_MAJOR, "yyy");
23 }
```

第 5 行代码说明注册“yyy”这个字符设备时，使用的 file_operations 结构体为 yyy_fops，15.4.3 小节将讲解这个结构体中成员函数的实现。

15.4.2 Linux I²C 设备驱动的 i2c_driver 成员函数

i2c_add_driver(&yyy_driver)的执行会引发 i2c_driver 结构体中 yyy_attach_adapter()函数的执行，我们可以在 yyy_attach_adapter()函数里探测物理设备。为了实现探测，yyy_attach_adapter()函数里面也只需简单地调用 I²C 核心的 i2c_probe()函数，如代码清单 15.15 所示。

代码清单 15.15　I²C 设备驱动的 i2c_attach_adapter 函数模板

```
1 static int yyy_attach_adapter(struct i2c_adapter *adapter)
```

```
2 {
3   return i2c_probe(adapter, &addr_data, yyy_detect);
4 }
```

代码第 3 行传递给 i2c_probe()函数的第 1 个参数是 i2c_adapter 指针，第 2 个参数是要探测的地址数据，第 3 个参数是具体的探测函数。要探测的地址实际列表在一个 16 位无符号整型数组中，这个数组以 I2C_CLIENT_END 为最后一个元素。

i2c_probe()函数会引发 yyy_detect()函数的调用，可以在 yyy_detect()函数中初始化 i2c_client，如代码清单 15.16 所示。

代码清单 15.16　I^2C 设备驱动的 detect 函数模板

```
1  static int yyy_detect(struct i2c_adapter *adapter, int address, int
kind)
2  {
3   struct i2c_client *new_client;
4   struct yyy_data *data;
5   int err = 0;
6
7   if (!i2c_check_functionality(adapter, I2C_FUNC_XXX)
8       goto exit;
9
10  if (!(data = kzalloc(sizeof(struct yyy_data), GFP_KERNEL)))
11  {
12      err =  - ENOMEM;
13      goto exit;
14  }
15
16  new_client = &data->client;
17  new_client->addr = address;
18  new_client->adapter = adapter;
19  new_client->driver = &yyy_driver;
20  new_client->flags = 0;
21
22  /* 新的client将依附于adapter */
23  if ((err = i2c_attach_client(new_client)))
24      goto exit_kfree;
25
26  yyy_init_client(new_client);
27  return 0;
28  exit_kfree: kfree(data);
29  exit: return err;
30}
```

代码第 10 行分配私有信息结构体的内存，i2c_client 也被创建。第 16～20 行对新创建的 i2c_client 进行初始化。第 23 行调用内核的 i2c_attach_client()知会 I^2C 核心系统中包含了一个新的 I^2C 设备。第 26 行代码初始化 i2c_client 对应的 I^2C 设备，这个函数是硬件相关的。

如图 15.4 所示为当 I^2C 设备驱动的模块加载函数被调用的时候引发的连锁反应的

流程。

I^2C 设备驱动的卸载函数进行 i2c_del_driver（&yyy_driver）调用后，会引发与 yyy_driver 关联的每个 i2c_client 与之解除关联，即 yyy_detach_client()函数将因此而被调用，代码清单 15.17 所示为函数 yyy_detach_client()的设计。

华清远见

代码清单 15.17　I²C 设备驱动的 i2c_detach_client 函数模板

```
1  static int yyy_detach_client(struct i2c_client *client)
2  {
3   int err;
4   struct yyy_data *data = i2c_get_clientdata(client);
5
6   if ((err = i2c_detach_client(client)))
7       return err;
8
9   kfree(data);
10  return 0;
11 }
```

上述函数中第 4 行的 i2c_get_clientdata()函数用于从 yyy_data 私有信息结构中的 i2c_client 的指针获取 yyy_data 的指针。第 6 行调用 I²C 核心函数 i2c_detach_client()，这个函数会引发 i2c_adapter 的 client_ unregister()函数被调用。第 9 行代码释放 yyy_data 的内存。

图 15.5 所示为当 I²C 设备驱动的模块卸载函数被调用的时候引发的连锁反应的流程。

图 15.4　I²C 设备驱动的模块加载连锁反应

图 15.5　I²C 设备驱动的模块卸载连锁反应

下面讲解 i2c_driver 中重要函数 yyy_command()的实现，它实现了针对设备的控制命令。具体的控制命令是设备相关的，如对于实时钟而言，命令将是设置时间和获取时间，而对于视频 AD 设备而言，命令会是设置采样方式、选择通道等。

假设 yyy 设备接受两类命令 YYY_CMD1 和 YYY_CMD2，而处理这两个命令的函数分别为 yyy_cmd1()和 yyy_cmd2()，代码清单 15.18 所示为 yyy_command()函数的

设计。

代码清单 15.18　I²C 设备驱动的 command 函数模板

```
static int yyy_command(struct i2c_client *client, unsigned int cmd, void
  *arg)
{
  switch (cmd)
  {
    case YYY_CMD1:
      return yyy_cmd1(client, arg);
    case YYY_CMD2:
      return yyy_cmd2(client, arg);
    default:
      return  - EINVAL;
  }
}
```

具体命令的实现是通过组件 i2c_msg 消息数组，并调用 I²C 核心的传输、发送和接收函数，由 I²C 核心的传输、发送和接收函数调用 I²C 适配器对应的 algorithm 相关函数来完成的。代码清单 15.19 所示为一个 yyy_cmd1()的例子。

代码清单 15.19　I²C 设备具体命令处理函数模板

```
static int yyy_cmd1(struct i2c_client *client, struct rtc_time *dt)
{
 struct i2c_msg msg[2];
 /*第一条消息是写消息*/
 msg[0].addr = client->addr;
 msg[0].flags = 0;
 msg[0].len = 1;
 msg[0].buf = &offs;
 /*第二条消息是读消息*/
 msg[1].addr = client->addr;
 msg[1].flags = I2C_M_RD;
 msg[1].len = sizeof(buf);
 msg[1].buf = &buf[0];

 i2c_transfer(client->adapter, msg, 2);
  ...
}
```

15.4.3 Linux I²C 设备驱动的文件操作接口

作为一种字符类设备，Linux I²C 设备驱动的文件操作接口与普通的设备驱动是完全一致的，但是在其中要使用 i2c_client、i2c_driver、i2c_adapter 和 i2c_algorithm 结构体和 I²C 核心，并且对设备的读写和控制需要借助体系结构中各组成部分的协同合作。代码清单 15.20 所示为一个 I²C 设备写函数的例子。

代码清单 15.20 I²C 设备文件的接口写函数范例

```
1  static ssize_t yyy_write(struct file *file, char *buf, size_t count, loff_t off)
2  {
3                  struct     i2c_client     *client     =     (struct i2c_client*)file->private_data;
4    i2c_msg msg[1];
5    char *tmp;
6    int ret;
7
8    tmp = kmalloc(count, GFP_KERNEL);
9    if (tmp == NULL)
10     return  - ENOMEM;
11   if (copy_from_user(tmp, buf, count))
12   {
13     kfree(tmp);
14     return  - EFAULT;
15   }
16
17   msg[0].addr = client->addr;//地址
18   msg[0].flags = 0;          //0 为写
19   msg[0].len = count;        //要写的字节数
20   msg[0].buf = tmp;          //要写的数据
21   ret = i2c_transfer(client->adapter, msg, 1);  //传输 I²C 消息
22   return (ret == 1) ? count : ret;
23 }
```

上述程序给出的仅仅是一个写函数的例子，具体的写操作与设备密切相关。下面详细讲解 I²C 设备读写过程中数据的流向和函数的调用关系。I²C 设备的写操作经历了如下几个步骤。

（1）从用户空间到字符设备驱动写函数接口，写函数构造 I²C 消息数组。

（2）写函数把构造的 I²C 消息数组传递给 I²C 核心的传输函数 i2c_transfer()。

（3）I²C 核心的传输函数 i2c_transfer()找到对应适配器 algorithm 的通信方法函数 master_xfer()去最终完成 I²C 消息的处理。

图 15.6 I²C 设备读写完整流程

如图 15.6 所示为从用户空间发起读写操作到 algorithm 进行消息传输的流程。

通常，如果 I²C 设备不是一个输入/输出设备或存储设备，就并不需要给 I²C 设备提供读写函数。许多 I²C 设备只是需要被设置

以某种方式工作，而不是被读写。另外，I^2C 设备驱动的文件操作接口也不是必需的，甚至极少被需要。大多数情况下，我们只需要通过 i2c-dev.c 文件提供的 I^2C 适配器设备文件接口就可完成对 I^2C 设备的读写。

15.4.4 Linux 的 i2c-dev.c 文件分析

i2c-dev.c 文件完全可以被看作一个 I^2C 设备驱动，其结构与 15.4.1～15.4.3 小节的描述是基本一致的，不过，它实现的一个 i2c_client 是虚拟、临时的，随着设备文件的打开而产生，并随设备文件的关闭而撤销，并没有被添加到 i2c_adapter 的 clients 链表中。i2c-dev.c 针对每个 I^2C 适配器生成一个主设备号为 89 的设备文件，实现了 i2c_driver 的成员函数以及文件操作接口，所以 i2c-dev.c 的主体是“i2c_driver 成员函数 + 字符设备驱动”。

i2c-dev.c 中提供 i2cdev_read()、i2cdev_write()函数来对应用户空间要使用的 read()和 write()文件操作接口，这两个函数分别调用 I^2C 核心的 i2c_master_recv()和 i2c_master_send()函数来构造一条 I^2C 消息并引发适配器 algorithm 通信函数的调用，完成消息的传输，对应于如图 15.7 所示的时序。但是，很遗憾，大多数稍微复杂一点 I^2C 设备的读写流程并不对应于一条消息，往往需要两条甚至更多的消息来进行一次读写周期（即如图 15.8 所示的重复开始位 RepStart 模式），这种情况下，在应用层仍然调用 read()、write()文件 API 来读写 I^2C 设备，将不能正确地读写。许多工程师碰到过类似的问题，往往经过相当长时间的调试都没法解决 I^2C 设备的读写，连错误的原因也无法找到，显然是对 i2cdev_read()和 i2cdev_write()函数的作用有所误解。

图 15.7　i2cdev_read 和 i2cdev_write 函数对应时序

图 15.8　RepStart 模式

鉴于上述原因，i2c-dev.c 中 i2cdev_read()和 i2cdev_write()函数不具备太强的通用性，没有太大的实用价值，只能适用于非 RepStart 模式的情况。对于两条以上消息组成的读写，在用户空间需要组织 i2c_msg 消息数组并调用 I2C_RDWR IOCTL 命令。代码清单 15.21 所示为 i2cdev_ioctl()函数的框架，其中详细列出了 I2C_RDWR 命令的处理过程。

代码清单 15.21　i2c-dev.c 中的 i2cdev_ioctl 函数

```
static int i2cdev_ioctl(struct inode *inode, struct file *file,
      unsigned int cmd, unsigned long arg)
{
 struct    i2c_client    *client    =    (struct    i2c_client *)file->private_data;
  ...
 switch ( cmd ) {
 case I2C_SLAVE:
```

```
case I2C_SLAVE_FORCE:
    ...   /*设置从设备地址*/
case I2C_TENBIT:
    ...
case I2C_PEC:
    ...
case I2C_FUNCS:
    ...
case I2C_RDWR:
  if (copy_from_user(&rdwr_arg,
            (struct i2c_rdwr_ioctl_data __user *)arg,
            sizeof(rdwr_arg)))
    return -EFAULT;
  /* 一次传入的消息太多 */
  if (rdwr_arg.nmsgs > I2C_RDRW_IOCTL_MAX_MSGS)
    return -EINVAL;
  /*获得用户空间传入的消息数组
  rdwr_pa = (struct i2c_msg *)
    kmalloc(rdwr_arg.nmsgs * sizeof(struct i2c_msg),
    GFP_KERNEL);
  if (rdwr_pa == NULL) return -ENOMEM;
  if (copy_from_user(rdwr_pa, rdwr_arg.msgs,
            rdwr_arg.nmsgs * sizeof(struct i2c_msg))) {
    kfree(rdwr_pa);
    return -EFAULT;
  }
  data_ptrs = kmalloc(rdwr_arg.nmsgs * sizeof(u8 __user *), GFP_KERNEL);
  if (data_ptrs == NULL) {
    kfree(rdwr_pa);
    return -ENOMEM;
  }
  res = 0;
  for( i=0; i<rdwr_arg.nmsgs; i++ ) {
    /* 限制消息的长度 */
    if (rdwr_pa[i].len > 8192) {
        res = -EINVAL;
        break;
```

```
    }
    data_ptrs[i] = (u8 __user *)rdwr_pa[i].buf;
    rdwr_pa[i].buf = kmalloc(rdwr_pa[i].len, GFP_KERNEL);
    if(rdwr_pa[i].buf == NULL) {
        res = -ENOMEM;
        break;
    }
    if(copy_from_user(rdwr_pa[i].buf,
        data_ptrs[i],
        rdwr_pa[i].len)) {
            ++i; /* Needs to be kfreed too */
            res = -EFAULT;
        break;
    }
  }
  if (res < 0) {
    int j;
    for (j = 0; j < i; ++j)
        kfree(rdwr_pa[j].buf);
    kfree(data_ptrs);
    kfree(rdwr_pa);
    return res;
  }
  /*把这些消息交给通信方法去处理*/
  res = i2c_transfer(client->adapter,
    rdwr_pa,
    rdwr_arg.nmsgs);
  while(i-- > 0) {   /*如果是读消息，把值复制到用户空间*/
    if( res>=0 && (rdwr_pa[i].flags & I2C_M_RD)) {
        if(copy_to_user(
            data_ptrs[i],
            rdwr_pa[i].buf,
            rdwr_pa[i].len)) {
            res = -EFAULT;
        }
    }
    kfree(rdwr_pa[i].buf);
  }
  kfree(data_ptrs);
  kfree(rdwr_pa);
```

```
85    return res;
86  case I2C_SMBUS:
87      ...
88  default:
89      return i2c_control(client,cmd,arg);
90  }
91  return 0;
92 }
```

常用的 IOCTL 包括 I2C_SLAVE（设置从设备地址）、I2C_RETRIES（没有收到设备 ACK 情况下的重试次数，默认为 1）、I2C_TIMEOU（超时）以及 I2C_RDWR。

由第 17～19 行可以看出，应用程序需要通过 i2c_rdwr_ioctl_data 结构体来给内核传递 I^2C 消息，这个结构体定义如代码清单 15.22 所示，i2c_msg 数组指针及消息数量就被包含在 i2c_rdwr_ ioctl_data 结构体中。

代码清单 15.22　i2c_rdwr_ioctl_data 结构体

```
1 struct i2c_rdwr_ioctl_data {
2   struct i2c_msg __user *msgs;   /* I2C 消息指针 */
3   __u32 nmsgs;                   /* I2C 消息数量 */
4 };
```

代码清单 15.23 和代码清单 15.24 所示为直接通过 read()、write()接口和 O_RDWR IOCTL 读写 I^2C 设备的例子。

代码清单 15.23　直接通过 read()/write()读写 I^2C 设备

```
1  #include <stdio.h>
2  #include <linux/types.h>
3  #include <fcntl.h>
4  #include <unistd.h>
5  #include <stdlib.h>
6  #include <sys/types.h>
7  #include <sys/ioctl.h>
8
9  #define I2C_RETRIES     0x0701
10 #define I2C_TIMEOUT     0x0702
11 #define I2C_SLAVE       0x0703
12
13 int main(int argc, char **argv)
14 {
15   unsigned int fd;
16   unsigned short mem_addr;
```

```
17  unsigned short size;
18  unsigned short idx;
19  #define BUFF_SIZE   32
20  char buf[BUFF_SIZE];
21  char cswap;
22  union
23  {
24    unsigned short addr;
25    char bytes[2];
26  } tmp;
27
28  if (argc < 3)
29  {
30    printf("Use:\n%s /dev/i2c-x mem_addr size\n", argv[0]);
31    return 0;
32  }
33  sscanf(argv[2], "%d", &mem_addr);
34  sscanf(argv[3], "%d", &size);
35
36  if (size > BUFF_SIZE)
37    size = BUFF_SIZE;
38
39  fd = open(argv[1], O_RDWR);
40
41  if (!fd)
42  {
43    printf("Error on opening the device file\n");
44    return 0;
45  }
46
47  ioctl(fd, I2C_SLAVE, 0x50); /* 设置 E2PROM 地址 */
48  ioctl(fd, I2C_TIMEOUT, 1); /* 设置超时 */
49  ioctl(fd, I2C_RETRIES, 1); /* 设置重试次数 */
50
51  for (idx = 0; idx < size; ++idx, ++mem_addr)
52  {
53    tmp.addr = mem_addr;
54    cswap = tmp.bytes[0];
55    tmp.bytes[0] = tmp.bytes[1];
56    tmp.bytes[1] = cswap;
57    write(fd, &tmp.addr, 2);
58    read(fd, &buf[idx], 1);
```

```
59   }
60   buf[size] = 0;
61   close(fd);
62   printf("Read %d char: %s\n", size, buf);
63   return 0;
64 }
```

代码清单 15.24　通过 O_RDWR IOCTL 读写 I^{2}C 设备

```
1  #include <stdio.h>
2  #include <linux/types.h>
3  #include <fcntl.h>
4  #include <unistd.h>
5  #include <stdlib.h>
6  #include <sys/types.h>
7  #include <sys/ioctl.h>
8  #include <errno.h>
9  #include <assert.h>
10 #include <string.h>
11
12 #define MAX_I2C_MSG        2
13
14 #define I2C_RETRIES    0x0701
15 #define I2C_TIMEOUT    0x0702
16 #define I2C_RDWR       0x0707
17
18 struct i2c_msg
19 {
20   __u16 addr; /* 从地址 */
21   __u16 flags;
22   #define I2C_M_RD       0x01
23   __u8 *buf; /* 消息数据指针 */
24 };
25 struct i2c_rdwr_ioctl_data
26 {
27   struct i2c_msg *msgs; /* i2c_msg[]指针 */
28   int nmsgs; /* i2c_msg 数量 */
29 };
30
31 int main(int argc, char **argv)
32 {
33   struct i2c_rdwr_ioctl_data work_queue;
34   unsigned int idx;
35   unsigned int fd;
36   unsigned short start_address;
37   int ret;
38
39   if (argc < 4)
40   {
```

```
    printf("Usage:\n%s /dev/i2c-x start_addr\n", argv[0]);
    return 0;
  }

  fd = open(argv[1], O_RDWR);

  if (!fd)
  {
    printf("Error on opening the device file\n");
    return 0;
  }
  sscanf(argv[2], "%x", &start_address);
  work_queue.nmsgs = MAX_I2C_MSG; /* 消息数量 */

    work_queue.msgs  =  (struct  i2c_msg*)malloc(work_queue.nmsgs *sizeof(struct
    i2c_msg));
  if (!work_queue.msgs)
  {
    printf("Memory alloc error\n");
    close(fd);
    return 0;
  }

  for (idx = 0; idx < work_queue.nmsgs; ++idx)
  {
    (work_queue.msgs[idx]).len = 0;
    (work_queue.msgs[idx]).addr = start_address + idx;
    (work_queue.msgs[idx]).buf = NULL;
  }

  ioctl(fd, I2C_TIMEOUT, 2); /* 设置超时 */
  ioctl(fd, I2C_RETRIES, 1); /* 设置重试次数 */

  ret = ioctl(fd, I2C_RDWR, (unsigned long) &work_queue);

  if (ret < 0)
  {
    printf("Error during I2C_RDWR ioctl with error code: %d\n", ret);
  }

  close(fd);
  return ;
}
```

15.5 S3C2410 I²C 总线驱动实例

15.5.1 S3C2410 I^2C 控制器硬件描述

S3C2410 处理器内部集成了一个 I^2C 控制器，通过 4 个寄存器就可方便地对其进行控制，这 4 个寄存器如下。

- IICCON：I^2C 控制寄存器。
- IICSTAT：I^2C 状态寄存器。
- IICDS：I^2C 收发数据移位寄存器。
- IICADD：I^2C 地址寄存器。

S3C2410 处理器内部集成的 I^2C 控制器可支持主、从两种模式，我们主要使用其主模式。通过对 IICCON、IICDS 和 IICADD 寄存器的操作，可在 I^2C 总线上产生开始位、停止位、数据和地址，而传输的状态则通过 IICSTAT 寄存器获取。

15.5.2 S3C2410 I^2C 总线驱动总体分析

S3C2410 的 I^2C 总线驱动设计主要要完成以下工作。

- 设计对应于 i2c_adapter_xxx_init()模板的 S3C2410 的模块加载函数和对应于 i2c_ adapter_xxx_exit()函数模板的模块卸载函数。
- 设计对应于 i2c_adapter_xxx_xfer()模板的 S3C2410 适配器的通信方法函数。

针对 S3C2410，functionality()函数只需简单地返回 I2C_FUNC_I2C | I2C_FUNC_SMBUS_ EMUL | I2C_FUNC_PROTOCOL_MANGLING 表明其支持的功能。

内核源代码中的/drivers/i2c/busses/i2c-s3c2410.c 文件包含了由 Ben Dooks<ben@simtec.co.uk>编写的 S3C2410 的 I^2C 总线驱动，在 Internet 上我们还可以获得其他版本的 S3C2410 I^2C 总线驱动源代码，如 Steve Hein<ssh@sgi.com>为三星 SMDK2410 参考板编写的 i2c-s3c2410.c、i2c-algo- s3c2410.c 文件，对比发现，它与 Ben Dooks 版本的驱动实现风格迥异。

这里分析内核中自带的 Ben Dooks 版本驱动，它同时支持 S3C2410 和 S3C2440。图 15.9 给出了 S3C2410 驱动中的主要函数与 15.3 节模板函数的对应关系，由于实现通信方法的方式不一样，模板的一个函数可能对应于 S3C2410 I^2C 总线驱动的多个函数。

图 15.9　I²C 总线驱动模板与 S3C2410 I²C 总线驱动的映射

15.5.3　S3C2410 I²C 适配器驱动的模块加载与卸载

I²C 适配器驱动被作为一个单独的模块加载进内核，在模块的加载和卸载函数中，只需注册和注销一个 platform_driver 结构体，如代码清单 15.25 所示。

代码清单 15.25　S3C2410 I²C 总线驱动的模块加载与卸载

```
static int __init i2c_adap_s3c_init(void)
{
 int ret;

 ret = platform_driver_register(&s3c2410_i2c_driver);
 if (ret == 0) {
     ret = platform_driver_register(&s3c2440_i2c_driver);
     if (ret)
         platform_driver_unregister(&s3c2410_i2c_driver);
 }

 return ret;
}

static void __exit i2c_adap_s3c_exit(void)
{
 platform_driver_unregister(&s3c2410_i2c_driver);
 platform_driver_unregister(&s3c2440_i2c_driver);
}
module_init(i2c_adap_s3c_init);
module_exit(i2c_adap_s3c_exit);
```

platform_driver 结构体包含了具体适配器的 probe()函数、remove()函数、resume()函数指针等信息，它需要被定义和赋值，如代码清单 15.26 所示。

代码清单 15.26　platform_driver 结构体

```
static struct platform_driver s3c2410_i2c_driver = {
  .probe      = s3c24xx_i2c_probe,
  .remove     = s3c24xx_i2c_remove,
  .resume     = s3c24xx_i2c_resume,
  .driver     = {
      .owner  = THIS_MODULE,
```

```
    7       .name   = "s3c2410-i2c",
    8   },
    9 };
```

当通过 Linux 内核源代码/drivers/base/platform.c 文件中定义 platform_driver_unregister()函数注册 platform_driver 结构体时，其中 probe 指针指向的 s3c24xx_i2c_probe()函数将被调用，以初始化适配器硬件，如代码清单 15.27 所示。

代码清单 15.27　S3C2410 I²C 总线驱动中的 s3c24xx_i2c_probe 函数

```
    1  static int s3c24xx_i2c_probe(struct platform_device *pdev)
    2  {
    3   struct s3c24xx_i2c *i2c = &s3c24xx_i2c;
    4   struct resource *res;
    5   int ret;
    6
    7   /* 使能 I2C 的时钟 */
    8   i2c->dev = &pdev->dev;
    9   i2c->clk = clk_get(&pdev->dev, "i2c");
    10  if (IS_ERR(i2c->clk)) {
    11      dev_err(&pdev->dev, "cannot get clock\n");
    12      ret = -ENOENT;
    13      goto out;
    14  }
    15  clk_enable(i2c->clk);
    16
    17  /* 映射寄存器 */
    18  res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    19  if (res == NULL) {
    20      dev_err(&pdev->dev, "cannot find IO resource\n");
    21      ret = -ENOENT;
    22      goto out;
    23  }
    24  i2c->ioarea           =           request_mem_region(res->start,
(res->end-res->start)+1,
    25                  pdev->name);
    26  if (i2c->ioarea == NULL) {
    27      dev_err(&pdev->dev, "cannot request IO\n");
    28      ret = -ENXIO;
    29      goto out;
    30  }
    31
    32  i2c->regs = ioremap(res->start, (res->end-res->start)+1);
    33  if (i2c->regs == NULL) {
    34      dev_err(&pdev->dev, "cannot map IO\n");
    35      ret = -ENXIO;
    36      goto out;
    37  }
    38
    39  /* 设置 I2C 的信息块 */
    40  i2c->adap.algo_data = i2c;
    41  i2c->adap.dev.parent = &pdev->dev;
```

```
42
43  /* 初始化 I2C 控制器 */
44  ret = s3c24xx_i2c_init(i2c);
45  if (ret != 0)
46      goto out;
47
48  /* 申请中断 */
49  res = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
50  if (res == NULL) {
51      ret = -ENOENT;
52      goto out;
53  }
54  ret = request_irq(res->start, s3c24xx_i2c_irq, SA_INTERRUPT,
55            pdev->name, i2c);
56  if (ret != 0) {
57      goto out;
58  }
59  i2c->irq = res;
60
61  ret = i2c_add_adapter(&i2c->adap); /*添加 i2c_adapter*/
62  if (ret < 0) {
63      goto out;
64  }
65
66  platform_set_drvdata(pdev, i2c);
67
68  out:
69  if (ret < 0)
70      s3c24xx_i2c_free(i2c);
71  return ret;
72 }
```

上述代码中的主体工作是使能硬件并申请 I^2C 适配器使用的 I/O 地址、中断号等，在这些工作都完成无误后，通过 I^2C 核心提供的 i2c_add_adapter()函数添加这个适配器。因为 S3C2410 内部集成 I^2C 控制器，可以确定 I^2C 适配器一定存在，s3c24xx_i2c_probe()函数虽然命名“探测”，但实际没有也不必进行任何探测工作，之所以这样命名完全是一种设计习惯。

与 s3c24xx_i2c_probe()函数完成相反功能的函数是 s3c24xx_i2c_remove()函数，它在适配器模块卸载函数调用 platform_driver_unregister()函数时所示通过 platform_driver 的 remove 指针方式被调用。xxx_i2c_remove()的设计模板如代码清单 15.28 所示。

代码清单 15.28　S3C2410 I^2C 总线驱动中的 s3c24xx_i2c_remove 函数

```
1  static int s3c24xx_i2c_remove(struct platform_device *pdev)
2  {
3   struct s3c24xx_i2c *i2c = platform_get_drvdata(pdev);
```

```

 if (i2c != NULL) {
     s3c24xx_i2c_free(i2c);
     platform_set_drvdata(pdev, NULL);
 }

 return 0;
}
```

代码清单 15.27 和代码清单 15.28 中用到的 s3c24xx_i2c 结构体进行适配器所有信息的封装，类似于私有信息结构体，它与代码清单 15.12 所示的 xxx_i2c 结构体模板对应。代码清单 15.29 所示的 s3c24xx_i2c 结构体的定义，以及驱动模块定义的一个 s3c24xx_i2c 结构体全局实例。

代码清单 15.29　s3c24xx_i2c 结构体

```
struct s3c24xx_i2c {
 spinlock_t              lock;
 wait_queue_head_t           wait;
 struct i2c_msg          *msg;
 unsigned int                msg_num;
 unsigned int                msg_idx;
 unsigned int                msg_ptr;
 enum s3c24xx_i2c_state  state;
 void __iomem                *regs;
 struct clk              *clk;
 struct device               *dev;
 struct resource         *irq;
 struct resource         *ioarea;
 struct i2c_adapter      adap;
};

static struct s3c24xx_i2c s3c24xx_i2c = {
 .lock   = SPIN_LOCK_UNLOCKED,  /*自旋锁未锁定*/
 .wait   = __WAIT_QUEUE_HEAD_INITIALIZER(s3c24xx_i2c.wait),/*等待队列初始化*/
 .adap   = {
     .name           = "s3c2410-i2c",
     .owner          = THIS_MODULE,
     .algo           = &s3c24xx_i2c_algorithm,
     .retries            = 2,
     .class          = I2C_CLASS_HWMON,
```

```
26  },
27 };
```

15.5.4 S3C2410 I²C 总线通信方法

由代码清单 15.29 的第 23 行可以看出，I²C 适配器对应的 i2c_algorithm 结构体实例为 s3c24xx_i2c_algorithm，代码清单 15.30 所示 s3c24xx_i2c_algorithm 的定义。

代码清单 15.30 S3C2410 的 i2c_algorithm 结构体

```
1 static struct i2c_algorithm s3c24xx_i2c_algorithm = {
2   .master_xfer            = s3c24xx_i2c_xfer,
3   .functionality      = s3c24xx_i2c_func,
4 };
```

上述代码第 1 行指定了 S3C2410 I²C 总线通信传输函数 s3c24xx_i2c_xfer()，这个函数非常关键，所有 I²C 总线上对设备的访问最终应该由它来完成，代码清单 15.31 所示为这个重要函数以及其依赖的 s3c24xx_i2c_doxfer() 函数和 s3c24xx_i2c_message_start()函数的源代码。

代码清单 15.31 S3C2410 I²C 总线驱动的 master_xfer 函数

```
1  static int s3c24xx_i2c_xfer(struct i2c_adapter *adap,
2           struct i2c_msg *msgs, int num)
3  {
4   struct s3c24xx_i2c *i2c = (struct s3c24xx_i2c *)adap->algo_data;
5   int retry;
6   int ret;
7
8   for (retry = 0; retry < adap->retries; retry++) {
9       ret = s3c24xx_i2c_doxfer(i2c, msgs, num);
10      if (ret != -EAGAIN)
11          return ret;
12      udelay(100);
13  }
14
15  return -EREMOTEIO;
16 }
17
18  static  int  s3c24xx_i2c_doxfer(struct  s3c24xx_i2c  *i2c,  struct
i2c_msg *msgs, int num)
19 {
20  unsigned long timeout;
21  int ret;
22
23  ret = s3c24xx_i2c_set_master(i2c);
```

```
if (ret != 0) {
    ret = -EAGAIN;
    goto out;
}

spin_lock_irq(&i2c->lock);
i2c->msg    = msgs;
i2c->msg_num = num;
i2c->msg_ptr = 0;
i2c->msg_idx = 0;
i2c->state   = STATE_START;
s3c24xx_i2c_enable_irq(i2c);
s3c24xx_i2c_message_start(i2c, msgs);
spin_unlock_irq(&i2c->lock);

timeout = wait_event_timeout(i2c->wait, i2c->msg_num == 0, HZ * 5);

ret = i2c->msg_idx;

if (timeout == 0)
    dev_dbg(i2c->dev, "timeout\n");
else if (ret != num)
    dev_dbg(i2c->dev, "incomplete xfer (%d)\n", ret);

msleep(1);/* 确保停止位已经被传递 */

out:
return ret;
}

static void s3c24xx_i2c_message_start(struct s3c24xx_i2c *i2c,
                  struct i2c_msg *msg)
{
unsigned int addr = (msg->addr & 0x7f) << 1;
unsigned long stat;
unsigned long iiccon;

stat = 0;
```

```
62  stat |=  S3C2410_IICSTAT_TXRXEN;
63
64  if (msg->flags & I2C_M_RD) {
65      stat |= S3C2410_IICSTAT_MASTER_RX;
66      addr |= 1;
67  } else
68      stat |= S3C2410_IICSTAT_MASTER_TX;
69  if (msg->flags & I2C_M_REV_DIR_ADDR)
70      addr ^= 1;
71
72  s3c24xx_i2c_enable_ack(i2c);/*如果要使能 ACK，则使能*/
73
74  iiccon = readl(i2c->regs + S3C2410_IICCON);
75  writel(stat, i2c->regs + S3C2410_IICSTAT);
76  writeb(addr, i2c->regs + S3C2410_IICDS);
77
78  udelay(1);/*在发送新的开始位前延迟 1 位*/
79  writel(iiccon, i2c->regs + S3C2410_IICCON);
80  stat |=  S3C2410_IICSTAT_START;
81  writel(stat, i2c->regs + S3C2410_IICSTAT);
82 }
```

s3c24xx_i2c_xfer()函数调用 s3c24xx_i2c_doxfer()函数传输 I^2C 消息，第 8 行的循环意味着最多可以重试 adap->retries 次。

s3c24xx_i2c_doxfer()首先将 S3C2410 的 I^2C 适配器设置为 I^2C 主设备，其后初始化 s3c24xx_i2c 结构体，使能 I^2C 中断，并调用 s3c24xx_i2c_message_start()函数启动 I^2C 消息的传输。

s3c24xx_i2c_message_start()函数写 S3C2410 适配器对应的控制寄存器，向 I^2C 从设备传递开始位和从设备地址。

上述代码只是启动了 I^2C 消息数组的传输周期，并没有完整实现图 15.3 中给出的 algorithm master_xfer 流程。这个流程的完整实现需要借助 I^2C 适配器上的中断来步步推进。代码清单 15.32 所示为 S3C2410 I^2C 适配器中断处理函数以及其依赖的 i2s_s3c_irq_nextbyte()函数的源代码。

代码清单 15.32　S3C2410 I^2C 适配器中断处理函数

```
1   static irqreturn_t s3c24xx_i2c_irq(int irqno, void *dev_id,
2                   struct pt_regs *regs)
3   {
4       struct s3c24xx_i2c *i2c = dev_id;
5       unsigned long status;
6       unsigned long tmp;
7
8       status = readl(i2c->regs + S3C2410_IICSTAT);
9       if (status & S3C2410_IICSTAT_ARBITR) {
10          ...
11      }
12
```

```
    if (i2c->state == STATE_IDLE) {
        tmp = readl(i2c->regs + S3C2410_IICCON);
        tmp &= ~S3C2410_IICCON_IRQPEND;
        writel(tmp, i2c->regs +  S3C2410_IICCON);
        goto out;
    }

    i2s_s3c_irq_nextbyte(i2c, status);/* 把传输工作进一步推进 */

     out:
        return IRQ_HANDLED;
    }

       static  int  i2s_s3c_irq_nextbyte(struct  s3c24xx_i2c  *i2c,
unsigned long iicstat)
    {
    unsigned long tmp;
    unsigned char byte;
    int ret = 0;

    switch (i2c->state) {
    case STATE_IDLE:
        goto out;
        break;
    case STATE_STOP:
        s3c24xx_i2c_disable_irq(i2c);
        goto out_ack;
    case STATE_START:
        /* 我们最近做的一件事是启动一个新 I2C 消息*/
        if (iicstat  & S3C2410_IICSTAT_LASTBIT &&
            !(i2c->msg->flags & I2C_M_IGNORE_NAK)) {
            /* 没有收到 ACK */
            s3c24xx_i2c_stop(i2c, -EREMOTEIO);
            goto out_ack;
        }

        if (i2c->msg->flags & I2C_M_RD)
            i2c->state = STATE_READ;
        else
```

```
            i2c->state = STATE_WRITE;

        /* 仅一条消息，而且长度为 0（主要用于适配器探测），发送停止位*/
        if (is_lastmsg(i2c) && i2c->msg->len == 0) {
            s3c24xx_i2c_stop(i2c, 0);
            goto out_ack;
        }

        if (i2c->state == STATE_READ)
            goto prepare_read;
        /* 进入写状态 */
    case STATE_WRITE:
    retry_write:
        if (!is_msgend(i2c)) {
            byte = i2c->msg->buf[i2c->msg_ptr++];
            writeb(byte, i2c->regs + S3C2410_IICDS);

        } else if (!is_lastmsg(i2c)) {
            /* 推进到下一条消息 */
            i2c->msg_ptr = 0;
            i2c->msg_idx ++;
            i2c->msg++;

            /* 检查是否要为该消息产生开始位 */
            if (i2c->msg->flags & I2C_M_NOSTART) {
                if (i2c->msg->flags & I2C_M_RD) {
                    s3c24xx_i2c_stop(i2c, -EINVAL);
                }
                goto retry_write;
            } else {
                /* 发送新的开始位 */
                s3c24xx_i2c_message_start(i2c, i2c->msg);
                i2c->state = STATE_START;
            }
        } else {
            s3c24xx_i2c_stop(i2c, 0);/* send stop */
        }
        break;
        case STATE_READ:
        /* 有一个字节可读，看是否还有消息要处理    */
```

```
        if (!(i2c->msg->flags & I2C_M_IGNORE_NAK) &&
           !(is_msglast(i2c) && is_lastmsg(i2c))) {

            if (iicstat & S3C2410_IICSTAT_LASTBIT) {
                dev_dbg(i2c->dev, "READ: No Ack\n");

                s3c24xx_i2c_stop(i2c, -ECONNREFUSED);
                goto out_ack;
            }
    }
    byte = readb(i2c->regs + S3C2410_IICDS);
    i2c->msg->buf[i2c->msg_ptr++] = byte;

prepare_read:
    if (is_msglast(i2c)) {/* last byte of buffer */
        if (is_lastmsg(i2c))
            s3c24xx_i2c_disable_ack(i2c);

    } else if (is_msgend(i2c)) {
        /* 还有消息要处理吗? */
        if (is_lastmsg(i2c)) {
            s3c24xx_i2c_stop(i2c, 0);/* last message, send stop and complete */
        } else {
            /* 推进到下一条消息 */
            i2c->msg_ptr = 0;
            i2c->msg_idx++;
            i2c->msg++;
        }
    }
    break;
}

/* irq 清除 */
    out_ack:
tmp = readl(i2c->regs + S3C2410_IICCON);
tmp &= ~S3C2410_IICCON_IRQPEND;
writel(tmp, i2c->regs + S3C2410_IICCON);
```

```
out:
return ret;
}
```

中断处理函数 s3c24xx_i2c_irq()主要通过调用 i2s_s3c_irq_nextbyte()函数进行传输工作的进一步推进。i2s_s3c_irq_nextbyte()函数通过 switch(i2c->state)语句分成 i2c->state 的不同状态进行处理，在每种状态下，先检查 i2c->state 的状态与硬件寄存器应该处于的状态是否一致，如果不一致，则证明有误，直接返回。当 I^2C 处于读状态 STATE_READ 或写状态 STATE_WRITE 时，通过 is_lastmsg()函数判断是否传输的是最后一条 I^2C 消息，如果是，则产生停止位，否则通过 i2c->msg_idx++、i2c->msg++推进到下一条消息。

15.6 SAA7113H 视频 AD 芯片的 I^2C 设备驱动实例

15.6.1 SAA7113H 视频 AD 芯片硬件描述

如图 15.10 所示，SAA7113H 是飞利浦半导体推出的 9 位视频 AD 芯片，它可以选择 4 路视频输入中的 1 路，并采样为 9 位的数字信号。

图 15.10 SAA7113H 输入、输出与 I^2C 接口

对 SAA7113H 输入通道的选择以及采样方式的设置都需通过其 I^2C 接口进行，以 0x4A 地址可读 SAA7113H 寄存器，以 0x4B 可写 SAA7113H 寄存器。SAA7113H 的 I^2C 接口连接在 S3C2410 的 I^2C 控制器上，作为 S3C2410 I^2C 控制器的从设备。SAA7113H 的写时序与图 15.7 给出的写时序是一致的，但是读时序则需通过两条 I^2C 消息解决，在第 1 条消息中应该给 SAA7113H 写入寄存器子地址，如图 15.11 所示。

图 15.11 SAA7113H 读时序

15.6.2 SAA7113H 视频 AD 芯片驱动的模块加载与卸载

由于不需要给 SAA7113H 实现文件操作接口，在设备驱动模块的加载和卸载函数中均不再需要注册和注销字符设备的操作，代码清单 15.33 所示为 SAA7113H 设备驱动的模块加载和卸载的源代码。

代码清单 15.33 SAA7113H 设备驱动的模块加载和卸载函数

```
static int __init saa711x_init (void)
{
  return i2c_add_driver(&i2c_driver_saa711x); //注册 i2c_driver
}

static void __exit saa711x_exit (void)
{
  i2c_del_driver(&i2c_driver_saa711x); //注销 i2c_driver
}

module_init(saa711x_init);
module_exit(saa711x_exit);
```

15.6.3 SAA7113H 设备驱动的 i2c_driver 成员函数

如代码清单 15.34 所示，SAA7113H 设备驱动模块加载和卸载中添加和删除的 i2c_driver_ saa711x 结构体的 attach_adapter 指针被赋值为 saa711x_attach_adapter，detach_client 指针被赋值为 saa711x_detach_client，而 command 指针被赋值为 saa711x_command，代码清单 15.35 所示为这 3 个函数的实现。

代码清单 15.34 SAA7113H 设备驱动 i2c_driver 结构体

```
static struct i2c_driver i2c_driver_saa711x = {
  .driver = {
  .name = "saa711x",
 },
  .id = I2C_DRIVERID_SAA711X,
  /* 成员函数 */
  .attach_adapter = saa711x_attach_adapter,
  .detach_client = saa711x_detach_client,
  .command = saa711x_command,
};
```

代码清单 15.35 SAA7113H 设备驱动 i2c_driver 成员函数

```
saa711x_attach_adapter (struct i2c_adapter *adapter)
{
 dprintk(1, KERN_INFO "saa711x.c: starting probe for adapter %s (0x%x)\n",
      I2C_NAME(adapter), adapter->id);
```

```
    5
    6   return i2c_probe(adapter, &addr_data, &saa711x_detect_client);
    7    //saa711x_detect_client   会   调   用   i2c_set_clientdata()   、
i2c_attach_client()
    8  }
    9
    10 static int saa711x_detach_client (struct i2c_client *client)
    11 {
    12  struct saa711x *decoder = i2c_get_clientdata(client);
    13  int err;
    14
    15  err = i2c_detach_client(client);
    16  if (err) {
    17      return err;
    18  }
    19
    20  kfree(decoder);
    21  kfree(client);
    22
    23  return 0;
    24 }
    25
    26 static int saa711x_command(struct i2c_client *client, unsigned int
cmd, void
    27   *arg)
    28 {
    29   struct saa711x *decoder = i2c_get_clientdata(client);
    30
    31   switch (cmd)  //处理不同的命令
    32   {
    33     case 0:
    34     case DECODER_INIT:
    35       ...
    36     case DECODER_DUMP:
    37       ...
    38     case DECODER_GET_CAPABILITIES:
    39       ...
    40     case DECODER_GET_STATUS:
    41       ...
    42     case DECODER_SET_VBI_BYPASS:
    43       ...
    44     case DECODER_SET_NORM:
```

```
    ...
  case DECODER_SET_INPUT:
    {
      int *iarg = arg;
      if (*iarg < 0 ||  *iarg > 9)
      {
        return  - EINVAL;
      }
      if (decoder->input !=  *iarg)
      {
        decoder->input =  *iarg;
        saa711x_write(client, 0x02, (decoder->reg[0x02] &0xf0) | decoder
          ->input);
        saa711x_write(client, 0x09, (decoder->reg[0x09] &0x7f) | ((decoder
          ->input > 3) ? 0x80 : 0));
      }
    }
    break;

  case DECODER_SET_OUTPUT:
    ...
  case DECODER_ENABLE_OUTPUT:
    ...
  case DECODER_SET_PICTURE:
    ...
  default:
    return  - EINVAL;
}

return 0;
}
```

15.8 总结

Linux I^2C 驱动体系结构有相当的复杂度，它主要由 3 部分组成，即 I^2C 核心、I^2C 总线驱动和 I^2C 设备驱动。I^2C 核心是 I2C 总线驱动和 I^2C 设备驱动的中间枢纽，它以通用的、与平台无关的接口实现了 I^2C 中设备与适配器的沟通。I^2C 总线驱动填充 i2c_adapter 和 i2c_algorithm 结构体，I^2C 设备驱动填充 i2c_driver 和 i2c_client 结构体。

另外，系统中 i2c-dev.c 文件定义的主设备号为 89 的设备可以方便地给应用程序提供读写 I^2C 设备寄存器的能力，使得工程师大多数时候并不需要为具体的 I^2C 设备驱动定义文件操作接口。

最后，工程师在设计 I^2C 设备驱动的程序，并不是一定要遵守 Linux I^2C 驱动的体系结构，完全可以把它当作一个普通的字符设备处理。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm
- 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程： 华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：
  http://www.farsight.com.cn/courses/TS-LinuxApp.htm
- 嵌入式 Linux 系统开发班：
  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm
- 嵌入式 Linux 驱动开发班：
  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm

华清远见

第 16 章　Linux 网络设备驱动

网络设备是完成用户数据包在网络媒介上发送和接收的设备，它将上层协议传递下来的数据包以特定的媒介访问控制方式进行发送，并将接收到的数据包传递给上层协议。

与字符设备和块设备不同，网络设备并不对应于/dev 目录下的文件，应用程序最终使用套接字（socket）完成与网络设备的接口。因而在网络设备身上并不能体现出“一切都是文件”的思想。

Linux 系统对网络设备驱动定义了 4 个层次，这 4 个层次为网络协议接口层、网络设备接口层、提供实际功能的设备驱动功能层和网络设备与媒介层。

在本章中，16.1 节讲解 Linux 网络设备驱动的层次结构，描述其 4 个层次各自的作用以及它们是如何协同合作以实现向下驱动网络设备硬件、向上提供数据包收发接口能力的。16.2～16.8 节主要讲解设备驱动功能层的各主要函数和数据结构，包括设备注册与注销、设备初始化、数据包收发函数、打开与释放函数等，在分析的基础上给出了抽象的设计模板。16.9 节是对前面各节讲解内容的例化，是对抽象设计模板的具体填充。

16.1 节与 16.2～16.8 节的内容是整体与部分的关系，而 16.1～16.8 节讲解的理论和设计模板与 16.9 节针对具体设备 CS8900 网卡进行的设计是抽象与具体的关系。

16.1 Linux 网络设备驱动的结构

Linux 网络设备驱动程序的体系结构如图 16.1 所示，从上到下可以划分为 4 层，依次为网络协议接口层、网络设备接口层、提供实际功能的设备驱动功能层以及网络设备与媒介层，这 4 层的作用如下所示。

- 网络协议接口层向网络层协议提供统一的数据包收发接口，不论上层协议为 ARP 还是 IP，都通过 dev_queue_xmit()函数发送数据，并通过 netif_rx()函数接收数据。这一层的存在使得上层协议独立于具体的设备。
- 网络设备接口层向协议接口层提供统一的用于描述具体网络设备属性和操作的结构体 net_device，该结构体是设备驱动功能层中各函数的容器。实际上，网络设备接口层从宏观上规划了具体操作硬件的设备驱动功能层的结构。
- 设备驱动功能层各函数是网络设备接口层 net_device 数据结构的具体成员，是驱使网络设备硬件完成相应动作的程序，它通过 hard_start_xmit()函数启动发送操作，并通过网络设备上的中断触发接收操作。
- 网络设备与媒介层是完成数据包发送和接收的物理实体，包括网络适配器和具体的传输媒介，网络适配器被设备驱动功能层中的函数物理上驱动。对于 Linux 系统而言，网络设备和媒介都可以是虚拟的。

图 16.1　Linux 网络设备驱动程序的体系结构

在设计具体的网络设备驱动程序时，我们需要完成的主要工作是编写设备驱动功能层的相关函数以填充 net_device 数据结构的内容并将 net_device 注册入内核。

16.1.1 网络协议接口层

网络协议接口层最主要的功能是给上层协议提供了透明的数据包发送和接收接口。当上层 ARP 或 IP 协议需要发送数据包时，它将调用网络协议接口层的 dev_queue_xmit()函数发送该数据包，同时需传递给该函数一个指向 struct sk_buff 数据结构的指针。dev_queue_xmit()函数的原型为：

```
dev_queue_xmit (struct sk_buff * skb );
```

同样地，上层对数据包的接收也通过向 netif_rx()函数传递一个 struct sk_buff 数据结构的指针来完成。netif_rx()函数的原型为：

```
int netif_rx(struct sk_buff *skb);
```

sk_buff 结构体非常重要，它的含义为“套接字缓冲区”，用于在 Linux 网络子系统中的各层之间传递数据，是 Linux 网络子系统数据传递的“中枢神经”。

当发送数据包时，Linux 内核的网络处理模块必须建立一个包含要传输的数据包的 sk_buff，然后将 sk_buff 递交给下层，各层在 sk_buff 中添加不同的协议头直至交给网络设备发送。同样地，当网络设备从网络媒介上接收到数据包后，它必须将接收到的数据转换为 sk_buff 数据结构并传递给上层，各层剥去相应的协议头直至交给用户。

1．套接字缓冲区成员

参看 linux/skbuff.h 中的源代码，sk_buff 结构体包含的主要成员如下

（1）各层协议头 h、nh 和 mac。

sk_buff 结构体中定义了 3 个协议头以对应于网络协议的不同层次，这 3 个协议头为传输层 TCP/UDP（及 ICMP 和 IGMP）协议头 h、网络层协议头 nh 和链路层协议头 mac。这 3 个协议头数据结构都被定义为联合体，如代码清单 16.1 所示。

代码清单 16.1　sk_buff 结构体协议头

```
union{
  struct tcphdr *th;        /* TCP 头部 */
  struct udphdr *uh;            /* UDP 头部*/
  struct icmphdr *icmph;        /* ICMP 头部*/
  struct igmphdr *igmph;        /* IGMP 头部*/
  struct iphdr *ipiph;          /* IP 头部*/
  struct ipv6hdr *ipv6h;    /* IPv6 头部*/
  unsigned char *raw;           /* 数据链路层头部*/
}h;

union{
  struct iphdr *iph;            /* IP 头部*/
  struct ipv6hdr *ipv6h;        /* IPv6 头部*/
  struct arphdr *arph;          /*ARP 头部*/
  unsigned char *raw;       /* 数据链路层头部*/
}nh;

union{
  unsigned char *raw;           /* 数据链路层头部*/
} mac;
```

（2）数据缓冲区指针 head、data、tail 和 end。

Linux 内核必须分配用于容纳数据包的缓冲区，sk_buff 结构体定义了 4 个指向这片缓冲区不同位置的指针 head、data、tail 和 end。

head 指针指向内存中已分配的用于承载网络数据的缓冲区的起始地址，sk_buff 和相关数据块在分配之后，该指针的值就被固定了。

data 指针则指向对应当前协议层有效数据的起始地址。每个协议层的有效数据含义并不相同，各层的有效数据信息包含的内容如下。

- 对于传输层而言，用户数据和传输层协议头属于有效数据。
- 对于网络层而言，用户数据、传输层协议头和网络层协议头是其有效数据。
- 对于数据链路层而言，用户数据、传输层协议头、网络层协议头和链路层头部都属于有效数据。

因此，data 指针的值需随着当前拥有 sk_buff 的协议层的变化进行相应的移动。

tail 指针则指向对应当前协议层有效数据负载的结尾地址，与 data 指针对应。

end 指针指向内存中分配的数据缓冲区的结尾，与 head 指针对应。和 head 指针一样，sk_buff 被分配之后，end 指针的值也就固定不变了。

很显然，有效数据必须位于分配的数据缓冲区内，即（data,tail）区间位于（head,end）区间内，如图 16.2 所示。因此，head、data、tail 和 end 这 4 个指针间存在如下关系：head <- data <- tail <- end。

图 16.2　head、data、tail 和 end 指针

从图 16.2 中可以看出，end 指针所指地址数据缓冲区的末尾还包括一个 skb_shared_info 结构体的空间，这个结构体存放分隔存储的数据片段，意味着可以将数据包的有效数据分成几片存储在不同的内存空间中。图中的 frags 为分片数组，每一个分片的长度上限是一页。

（3）长度信息 len、data_len、truesize。

sk_buff 结构体中定义的 len 是指数据包有效数据的长度，包括协议头和负载（Payload）。

为了支持数据包的分片存放，sk_buff 中增加了 data_len 这个成员，它记录分片的数据长度。

truesize 表示缓存区的整体长度，置为 sizeof(struct sk_buff)加上传入 alloc_skb()函数或 dev_alloc_skb()函数（下文将要介绍）的长度，但不包括结构体 skb_shared_info 的长度。

2．套接字缓冲区操作

下面我们来分析套接字缓冲区涉及到的操作函数，Linux 套接字缓冲区支持分配、释放、指针移动等功能函数。

（1）分配。

Linux 内核用于分配套接字缓冲区的函数有：

```
struct sk_buff *alloc_skb(unsigned int len,int priority);
```

```
struct sk_buff *dev_alloc_skb(unsigned int len);
```

alloc_skb()函数分配一个套接字缓冲区和一个数据缓冲区，参数 len 为数据缓冲区的空间大小，以 16 字节对齐，参数 priority 为内存分配的优先级。

dev_alloc_skb()函数只是以 GFP_ATOMIC 优先级（代表分配过程不能被中断）调用上面的 alloc_skb()函数，并保存 skb->head 和 skb->data 之间的 16 个字节。

分配成功之后，因为还没有存放具体的网络数据包，所以 sk_buff 的 data、tail 指针都指向存储空间的起始地址 head，而 len 的大小则为 0。

（2）释放。

Linux 内核用于释放套接字缓冲区的函数有：

```
void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
void dev_kfree_skb_irq(struct sk_buff *skb);
void dev_kfree_skb_any(struct sk_buff *skb);
```

上述函数用于释放被 alloc_skb()函数分配的套接字缓冲区和数据缓冲区。

Linux 内核内部使用 kree_skb()函数，而网络设备驱动程序中则必须用 dev_kfree_skb()、dev_kfree_skb_irq()或 dev_kfree_skb_any()函数进行套接字缓冲区的释放。其中，dev_kfree_skb()函数用于非中断上下文，dev_kfree_skb_irq()函数用于中断上下文，而 dev_kfree_skb_any()函数则在中断和非中断上下文中皆可采用。

（3）指针移动。

Linux 套接字缓冲区中的数据缓冲区指针移动操作包括 put（放置）、push（推）、pull（拉）、reserve（保留）等。

① put 操作

数据缓冲区指针 put 操作以下列函数完成：

```
unsigned char *skb_put(struct sk_buff *skb, unsigned int len);
unsigned char *__skb_put(struct sk_buff *skb, unsigned int len);
```

上述函数将 tail 指针下移，增加 sk_buff 的 len 值，并返回 skb->tail 的当前值。skb_put()和__ skb_put()的区别在于前者会检测放入缓冲区的数据，而后者不会检查。这两个函数主要用于在缓冲区尾部添加数据。

② push 操作

数据缓冲区指针 push 操作以下列函数完成：

```
unsigned char *skb_push(struct sk_buff *skb, unsigned int len);
unsigned char *__skb_push(struct sk_buff *skb, unsigned int len);
```

与 skb_put()和_ _skb_put()不同，skb_push()和_ _skb_push()会将 data 指针上移，因此也要增加 sk_buff 的 len 值。push 操作在存储空间的头部增加一段可以存储网络数据包的空间，而 put 操作则在存储空间的尾部增加一段可以存储网络数据包的空间，因此主要用于在数据包发送时添加头部。skb_push()与_ _skb_push()的区别和 skb_put()和_ _skb_put()的区别类似。

③ pull 操作

数据缓冲区指针 pull 操作以下列函数完成：

```
unsigned char * skb_pull(struct sk_buff *skb, unsigned int len);
```

skb_pull()函数将 data 指针下移，并减小 skb 的 len 值。这个操作一般用于下层协议向上层协议移交数据包，使 data 指针指向上一层协议的协议头。

④ reserve 操作

数据缓冲区指针 reserve 操作以下列函数完成：

```
void skb_reserve(struct sk_buff *skb, unsigned int len);
```

skb_reserve()函数将 data 指针和 tail 指针同时下移，这个操作主要用于在存储空间的头部预留 len 长度的空隙。

下面我们以一个具体的 UDP 数据包接收的 Linux 处理流程为例来说明 sk_buff 的操作过程，这一过程的绝大部分工作都由 Linux 内核完成，驱动工程师只需完成涉及的数据链路层部分。

假设以太网适配器（以太网卡）收到了一个 UDP 数据包，Linux 从底层到应用层处理这一数据包的流程如下。

（1）网卡收到一个 UDP 数据包后，驱动程序需要创建一个 sk_buff 结构体和数据缓冲区，将收到的数据全部复制到 data 指向的空间，并将 skb->mac.raw 指向 data。此时，有效数据的开始位置是一个以太网头，skb->mac.raw 指向链路层的以太网头部。

代码清单 16.2 演示了驱动程序在接收到数据包后分配 sk_buff 和数据缓冲区并将数据包复制到缓冲区的过程。

代码清单 16.2　分配 sk_buff 和数据缓冲区

```
1   /* 分配新的套接字缓冲区和数据缓冲区 */
2   skb = dev_alloc_skb(length + 2);
3   if (skb == NULL)
4   {
5     .../* 分配失败 */
6     return ;
7   }
8
9   skb_reserve(skb, 2); /* 预留空间以使网络层协议头对齐 */
10
11  /* 将硬件上接收到的数据复制到数据缓冲区 */
12  readwords(ioaddr, RX_FRAME_PORT, skb_put(skb, length), length >> 1);
13  if (length &1)
14    skb->data[length - 1] = readword(ioaddr, RX_FRAME_PORT);
```

（2）数据链路层通过调用 skb_pull()函数剥掉以太网协议头，向网络层 IP 传递数据包。在剥离过程中，data 指针会下移一个以太网头部的长度 sizeof(struct ethhdr)，而 len 则减去 sizeof(struct ethhdr)。此时，有效数据的开始位置是一个 IP 头部，skb->nh.raw

指向 data，即 IP 头部，而 skb->mac.raw 仍指向以太网头部。

（3）网络层通过 skb_pull()函数剥掉 IP 头，向传输层 UDP 传递数据包。在剥离过程中，data 指针会下移一个 IP 头部的长度 sizeof(struct iphdr)，而 len 则减去 sizeof(struct iphdr)。此时，有效数据的开始位置是一个 UDP 头部，skb->h.raw 指向 data，即 UDP 首部，而 skb->nh.raw 指向 IP 头部，skb->mac.raw 仍指向以太网头部。

（4）应用程序在调用 recv()接收数据时，从 skb->data+sizeof(struct udphdr)的位置开始复制到应用层缓冲区。由此可见，UDP 头部到最后也没有被剥离。

如图 16.3 所示为上述过程中各指针的指向和移动过程。

16.1.2 网络设备接口层

网络设备接口层的主要功能是为千变万化的网络设备定义了统一、抽象的数据结构 net_device 结构体，以不变应万变，实现多种硬件在软件层次上的统一。

net_device 结构体在内核中指代一个网络设备，网络设备驱动程序只需通过填充 net_device 的具体成员并注册 net_device 即可实现硬件操作函数与内核的挂接。

net_device 本身是一个巨型结构体，包含网络设备的属性描述和操作接口。当我们编写网络设备驱动程序时，只需要了解其中的一部分。

（1）全局信息。

```
char name[IFNAMESIZ];
```

name 是网络设备的名称。

```
int (*init)(struct net_device *dev);
```

init 为设备初始化函数指针，如果这个指针被设置了，则网络设备被注册时将调用该函数完成对 net_device 结构体的初始化。但是，设备驱动程序可以不实现这个函数并将其赋值为 NULL。

（2）硬件信息。

```
unsigned long mem_end;
unsigned long mem_start;
```

mem_start 和 mem_end 分别定义了设备所使用的共享内存的起始和结束地址。

```
unsigned long base_addr;
unsigned char irq;
unsigned char if_port;
unsigned char dma;
```

base_addr 为网络设备 I/O 基地址。

irq 为设备使用的中断号。

if_port 指定多端口设备使用哪一个端口，该字段仅针对多端口设备。例如，如果设备同时支持 IF_PORT_10BASE2（同轴电缆）和 IF_PORT_10BASET（双绞线），则可使用该字段。

dma 指定分配给设备的 DMA 通道。

（3）接口信息。

```
unsigned short hard_header_len;
```

hard_header_len 是网络设备的硬件头长度，在以太网设备的初始化函数中，该成员被赋为 ETH_HLEN，即 14。

```
unsigned short type;
```

type 是接口的硬件类型。

```
unsigned mtu;
```

mtu 指最大传输单元（MTU）。

```
unsigned char dev_addr[MAX_ADDR_LEN];
unsigned char broadcast[MAX_ADDR_LEN];
```

dev_addr[ ]、broadcast[ ]无符号字符数组，分别用于存放设备的硬件地址和广播地址。对于以太网而言，这两个地址的长度都为 6 个字节。以太网设备的广播地址为 6 个 0xFF，而 MAC 地址需由驱动程序从硬件上读出并填充到 dev_addr[ ]中。

```
unsigned short flags;
```

flags 指网络接口标志，以 IFF_（interface flags）开头，部分标志由内核来管理，其他的在接口初始化时被设置以说明设备接口的能力和特性。接口标志包括 IFF_UP（当设备被激活并可以开始发送数据包时，内核设置该标志）、IFF_AUTOMEDIA（设备可在多种媒介间切换）、IFF_BROADCAST（允许广播）、IFF_DEBUG（调试模式，可用于控制 printk 调用的详细程度）、IFF_LOOPBACK（回环）、IFF_MULTICAST（允许组播）、IFF_NOARP（接口不能执行 ARP）、IFF_POINTOPOINT（接口连接到点到点链路）等。

（4）设备操作函数。

```
int (*open)(struct net_device *dev);
int (*stop)(struct net_device *dev);
```

open()函数的作用是打开网络接口设备，获得设备需要的 I/O 地址、IRQ、DMA 通道等。stop()函数的作用是停止网络接口设备，与 open()函数的作用相反。

```
int  (*hard_start_xmit) (struct sk_buff *skb,struct net_device *dev);
```

hard_start_xmit()函数会启动数据包的发送，当系统调用驱动程序的 hard_start_xmit()函数时，需要向其传入一个 sk_buff 结构体指针，以使得驱动程序能获取从上层传递下来的数据包。

```
void (*tx_timeout)(struct net_device *dev);
```

当数据包的发送超时时，tx_timeout ()函数会被调用，该函数需采取重新启动数据包发送过程或重新启动硬件等策略来恢复网络设备到正常状态。

```
int  (*hard_header) (struct sk_buff *skb,
                     struct net_device *dev,
                     unsigned short type,
                     void *daddr,
                     void *saddr,
                     unsigned len);
```

hard_header()函数完成硬件帧头填充，返回填充的字节数。传入该函数的参数包括 sk_buff 指针、设备指针、协议类型、目的地址、源地址以及数据长度。对于以太网设备而言，将内核提供的 eth_header()函数赋值给 hard_header 指针即可。

```
struct net_device_stats* (*get_stats)(struct net_device *dev);
```

get_stats()函数用于获得网络设备的状态信息，它返回一个 net_device_stats 结构体。net_device_stats 结构体保存了网络设备详细的流量统计信息，如发送和接收到的

数据包数、字节数等，详见 16.8 节。

```
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
int (*set_config)(struct net_device *dev, struct ifmap *map);
int (*set_mac_address)(struct net_device *dev, void *addr);
```

do_ioctl()函数用于进行设备特定的 I/O 控制。

set_config()函数用于配置接口，可用于改变设备的 I/O 地址和中断号。

set_mac_address()函数用于设置设备的 MAC 地址。

```
int (*poll)( struct net_device *dev,int quota);
```

对于 NAPI（网络中断缓和）兼容的设备驱动，将以轮询方式操作接口，接收数据包。NAPI 是 Linux 系统上采用的一种提高网络处理效率的技术，它的核心概念就是不采用中断的方式读取数据包，而是采用首先借助中断唤醒数据包接收的服务程序，然后以轮询方式获取数据包。

net_device 结构体的上述成员需要在设备初始化时被填充，详见 16.3 节。

（5）辅助成员。

```
unsigned long trans_start;
unsigned long last_rx;
```

trans_start 记录最后的数据包开始发送时的时间戳，last_rx 记录最后一次接收到数据包时的时间戳，这两个时间戳记录的都是 jiffies，驱动程序应维护这两个成员。

```
void *priv;
```

priv 为设备的私有信息指针，与 filp->private_data 的地位相当。设备驱动程序中应该以 netdev_priv()函数获得该指针。

```
spinlock_t xmit_lock;
int xmit_lock_owner;
```

xmit_lock 是避免 hard_start_xmit()函数被同时多次调用的自旋锁。xmit_lock_owner 则指当前拥有 xmit_lock 自旋锁的 CPU 的编号。

16.1.3 设备驱动功能层

net_device 结构体的成员（属性和函数指针）需要被设备驱动功能层的具体数值和函数赋予。对于具体的设备 xxx，工程师应该编写设备驱动功能层的函数，这些函数形如 xxx_open()、xxx_stop()、xxx_tx()、xxx_ hard_header()、xxx_get_stats()、xxx_tx_timeout()、xxx_poll()等。

由于网络数据包的接收可由中断引发，设备驱动功能层中另一个主体部分将是中断处理函数，它负责读取硬件上接收的数据包并传送给上层协议，可能包含 xxx_interrupt()和 xxx_rx()函数，前者完成中断类型判断等基本的工作，后者则需完成数据包的生成和递交上层等复杂工作。

16.2～16.8 节将对上述 xxx_xxx()函数进行详细地分析并给出参考设计模板。

对于特定的设备，我们还可以定义其相关私有数据和操作，并封装为一个私有信息结构体 xxx_private，让其指针被赋值给 net_device 的 priv 成员。xxx_private 结构体中可包含设备特殊的属性和操作、自旋锁与信号量、定时器以及统计信息等，由工程师自定义。

16.1.4 网络设备与媒介层

网络设备与媒介层直接对应于实际的硬件设备。为了给设备的物理配置和寄存器

操作一个更一般的描述，我们可以定义一组宏和一组访问设备内部寄存器的函数，具体的宏和函数与特定的硬件紧密相关。代码清单 16.3 所示为相应的设计范例。

代码清单 16.3　网络设备底层硬件操作

```
/*  寄存器定义 */
#define DATA_REG 0x0004
#define CMD_REG 0x0008

/* 寄存器读写函数 */
static u16 xxx_readword(u32 base_addr, int portno)
{
  ...  /*读取寄存器的值并返回*/
}

static void xxx_writeword(u32 base_addr, int portno, u16 value)
{
  ...  /*向寄存器写入数值*/
}
```

16.2 网络设备驱动的注册与注销

网络设备驱动的注册与注销使用成对出现的 register_netdev()和 unregister_netdev()函数完成，这两个函数的原型为：

```
int register_netdev(struct net_device *dev);
void unregister_netdev(struct net_device *dev);
```

这两个函数都接收一个 net_device 结构体指针为参数，可见 net_device 数据结构在网络设备驱动中的核心地位。

net_device 的生成和成员的赋值并非一定要由工程师逐个亲自动手完成，可以利用下面的函数帮助我们填充：

```
struct net_device *alloc_netdev(int sizeof_priv, const char *name, void(*setup)
    (struct net_device*));
struct net_device *alloc_etherdev(int sizeof_priv);
```

alloc_netdev()函数生成一个 net_device 结构体，对其成员赋值并返回该结构体的指针。第一个参数为设备私有成员的大小，第二个参数为设备名，第三个参数为 net_device 的 setup()函数指针。setup()函数接收的参数也为 struct net_device 指针，用于预置 net_device 成员的值。

alloc_etherdev()是 alloc_netdev()针对以太网的“快捷”函数，这从 alloc_etherdev()的源代码可以看出，如代码清单 16.4 所示。

代码清单 16.4　alloc_etherdev()函数

```
1   struct net_device *alloc_etherdev(int sizeof_priv)
2   {
3     /* 以 ether_setup 为 alloc_netdev 的 setup 参数 */
4     return alloc_netdev(sizeof_priv, "eth%d", ether_setup);
5   }
```

上述代码中的第 4 行传入 alloc_netdev ()函数的第三个参数为 ether_setup()函数地址，ether_setup()是由 Linux 内核提供的一个对以太网设备 net_device 结构体中公有成员快速赋值的函数。

完成与 alloc_enetdev()和 alloc_etherdev()函数相反功能，即释放 net_device 结构体的函数为：

```
void free_netdev(struct net_device *dev);
```

net_device 结构体的分配和网络设备驱动注册需在网络设备驱动程序的模块加载函数中进行，而 net_device 结构体的释放和网络设备驱动的注销则需在模块卸载函数中完成，如代码清单 16.5 所示。

代码清单 16.5　网络设备驱动程序的模块加载函数模板

```
1   int xxx_init_module(void)
2   {
3     ...
4     /* 分配 net_device 结构体并对其成员赋值 */
5     xxx_dev = alloc_netdev(sizeof(struct xxx_priv), "sn%d", xxx_init);
6     if (xxx_dev == NULL)
7     ... /* 分配 net_device 失败 */
8
9     /* 注册 net_device 结构体 */
10     if ((result = register_netdev(xxx_dev)))
11     ...
12  }
13
14  void xxx_cleanup(void)
15  {
16    ...
17    /* 注销 net_device 结构体 */
18    unregister_netdev(xxx_dev);
19    /* 释放 net_device 结构体 */
20    free_netdev(xxx_dev);
21  }
```

16.3 网络设备的初始化

网络设备的初始化主要需要完成如下几个方面的工作。

- 进行硬件上的准备工作，检查网络设备是否存在，如果存在，则检测设备所

使用的硬件资源。

- 进行软件接口上的准备工作，分配 net_device 结构体并对其数据和函数指针成员赋值。
- 获得设备的私有信息指针并初始化其各成员的值。如果私有信息中包括自旋锁或信号量等并发或同步机制，则需对其进行初始化。

对 net_device 结构体成员及私有数据的赋值都可能需要与硬件初始化工作协同进行，即硬件检测出了相应的资源，需要根据检测结果填充 net_device 结构体成员和私有数据。

一个网络设备驱动初始化函数的模板如代码清单 16.6 所示，具体的设备驱动初始化函数并不一定完全和本模板一样，但是其本质过程是一致的。

代码清单 16.6　网络设备驱动的初始化函数模板

```
void xxx_init(struct net_device *dev)
{
  /*设备的私有信息结构体*/
  struct xxx_priv *priv;

  /* 检查设备是否存在和设备所使用的硬件资源 */
  xxx_hw_init();

  /* 初始化以太网设备的公用成员 */
  ether_setup(dev);

  /*设置设备的成员函数指针*/
  dev->open = xxx_open;
  dev->stop = xxx_release;
  dev->set_config = xxx_config;
  dev->hard_start_xmit = xxx_tx;
  dev->do_ioctl = xxx_ioctl;
  dev->get_stats = xxx_stats;
  dev->change_mtu = xxx_change_mtu;
  dev->rebuild_header = xxx_rebuild_header;
  dev->hard_header = xxx_header;
  dev->tx_timeout = xxx_tx_timeout;
  dev->watchdog_timeo = timeout;

  /*如果使用 NAPI，设置 pool 函数*/
  if (use_napi)
  {
    dev->poll = xxx_poll;
  }

  /* 取得私有信息，并初始化它*/
  priv = netdev_priv(dev);
  ...  /* 初始化设备私有数据区 */
}
```

上述代码第 7 行的 xxx_hw_init()函数完成硬件相关的初始化操作，如下所示。

- 探测 xxx 网络设备是否存在。探测的方法类似于数学上的“反证法”，即先假设存在设备 xxx，访问该设备，如果设备的表现与预期的一致，就确定设备存在；否则，假设错误，设备 xxx 不存在。
- 探测设备的具体硬件配置。一些设备驱动编写得非常通用，对于同类的设备使用统一的驱动，我们需要在初始化时探测设备的具体型号。另外，即便是同一设备，在硬件上的配置也可能不一样，我们也可以探测设备所使用的硬件资源。
- 申请设备所需要的硬件资源，如用 request_region()函数进行 I/O 端口的申请等，但是这个过程可以放在设备的打开函数 xxx_open()中完成。

16.4 网络设备的打开与释放

网络设备的打开函数需要完成如下工作。

- 使能设备使用的硬件资源，申请 I/O 区域、中断和 DMA 通道等。
- 调用 Linux 内核提供的 netif_start_queue()函数，激活设备发送队列。

网络设备的关闭函数需要完成如下工作。

- 调用 Linux 内核提供的 netif_stop_queue()函数，停止设备传输包。
- 释放设备所使用的 I/O 区域、中断和 DMA 资源。

Linux 内核提供的 netif_start_queue()和 netif_stop_queue()两个函数的原型为：

```
void netif_start_queue(struct net_device *dev);
void netif_stop_queue (struct net_device *dev);
```

根据以上分析，可得出如代码清单 16.7 所示的网络设备打开和释放函数的模板。

代码清单 16.7　网络设备打开和释放函数模板

```
1   int xxx_open(struct net_device *dev)
2   {
3     /* 申请端口、IRQ 等，类似于 fops->open */
4     ret = request_irq(dev->irq, &xxx_interrupt, 0, dev->name, dev);
5     ...
6     netif_start_queue(dev);
7     ...
8   }
9
10  int xxx_release(struct net_device *dev)
11  {
12    /* 释放端口、IRQ 等，类似于 fops->close */
13    free_irq(dev->irq, dev);
14    ...
15    netif_stop_queue(dev); /* can't transmit any more */
16    ...
17  }
```

16.5 数据发送流程

从 16.1 节网络设备驱动程序的结构分析可知，Linux 网络子系统在发送数据包时，会调用驱动程序提供的 hard_start_transmit()函数，该函数用于启动数据包的发送。在设备初始化的时候，这个函数指针需被初始化指向设备的 xxx_tx()函数。

网络设备驱动完成数据包发送的流程如下。

（1）网络设备驱动程序从上层协议传递过来的 sk_buff 参数获得数据包的有效数据和长度，将有效数据放入临时缓冲区。

（2）对于以太网，如果有效数据的长度小于以太网冲突检测所要求数据帧的最小长度 ETH_ZLEN，则给临时缓冲区的末尾填充 0。

（3）设置硬件的寄存器，驱使网络设备进行数据发送操作。

完成以上 3 个步骤的网络设备驱动程序的数据包发送函数的模板如代码清单 16.8 所示。

代码清单 16.8　网络设备驱动程序的数据包发送函数模板

```
int xxx_tx(struct sk_buff *skb, struct net_device *dev)
{
  int len;
  char *data, shortpkt[ETH_ZLEN];
 /* 获得有效数据指针和长度 */
  data = skb->data;
  len = skb->len;
  if (len < ETH_ZLEN)
  {
     /* 如果帧长小于以太网帧最小长度，补0 */
    memset(shortpkt, 0, ETH_ZLEN);
    memcpy(shortpkt, skb->data, skb->len);
    len = ETH_ZLEN;
    data = shortpkt;
  }

  dev->trans_start = jiffies; /* 记录发送时间戳 */

  /* 设置硬件寄存器让硬件把数据包发送出去 */
  xxx_hw_tx(data, len, dev);
  ...
}
```

当数据传输超时时，意味着当前的发送操作失败，此时，数据包发送超时处理函数 xxx_tx_ timeout()将被调用。这个函数需要调用 Linux 内核提供的 netif_wake_queue()

函数重新启动设备发送队列，如代码清单 16.9 所示。

华清远见

代码清单 16.9　网络设备驱动程序的数据包发送超时函数模板

```
1   void xxx_tx_timeout(struct net_device *dev)
2   {
3     ...
4     netif_wake_queue(dev);  /* 重新启动设备发送队列 */
5   }
```

16.6 数据接收流程

网络设备接收数据的主要方法是由中断引发设备的中断处理函数，中断处理函数判断中断类型，如果为接收中断，则读取接收到的数据，分配 sk_buffer 数据结构和数据缓冲区，将接收到的数据复制到数据缓冲区，并调用 netif_rx()函数将 sk_buffer 传递给上层协议。代码清单 16.10 所示为完成这一过程的函数模板。

代码清单 16.10　网络设备驱动的中断处理函数模板

```
1   static void xxx_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
2   {
3     ...
4     switch (status &ISQ_EVENT_MASK)
5     {
6       case ISQ_RECEIVER_EVENT:
7         /* 获取数据包 */
8         xxx_rx(dev);
9         break;
10        /* 其他类型的中断 */
11    }
12  }
13  static void xxx_rx(struct xxx_device *dev)
14  {
15    ...
16    length = get_rev_len (...);
17    /* 分配新的套接字缓冲区 */
18    skb = dev_alloc_skb(length + 2);
19
```

```
20    skb_reserve(skb, 2); /* 对齐 */
21    skb->dev = dev;
22
23    /* 读取硬件上接收到的数据 */
24    insw(ioaddr + RX_FRAME_PORT, skb_put(skb, length), length >> 1);
25    if (length &1)
26      skb->data[length - 1] = inw(ioaddr + RX_FRAME_PORT);
27
28    /* 获取上层协议类型 */
29    skb->protocol = eth_type_trans(skb, dev);
30
31    /* 把数据包交给上层 */
32    netif_rx(skb);
33
34    /* 记录接收时间戳 */
35    dev->last_rx = jiffies;
36    ...
37  }
```

从上述代码的第 4～7 行可以看出，当设备的中断处理程序判断中断类型为数据包接收中断时，它调用第 13～37 定义的 xxx_rx()函数完成更深入的数据包接收工作。xxx_rx()函数代码中的第 16 行从硬件读取到接收数据包有效数据的长度，第 17～20 行分配 sk_buff 和数据缓冲区，第 23～26 行读取硬件上接收到的数据并放入数据缓冲区，第 28～29 行解析接收数据包上层协议的类型，最后，第 31～32 行代码将数据包上交给上层协议。

如果是 NAPI 兼容的设备驱动，则可以通过 poll 方式接收数据包。这种情况下，我们需要为该设备驱动提供 xxx_poll()函数，如代码清单 16.11 所示。

代码清单 16.11　网络设备驱动的 poll 函数模板

```
1   static int xxx_poll(struct net_device *dev, int *budget)
2   {
3     int npackets = 0, quota = min(dev->quota,  *budget);
4     struct sk_buff *skb;
5     struct xxx_priv *priv = netdev_priv(dev);
6     struct xxx_packet *pkt;
7
8     while (npackets < quota && priv->rx_queue)
9     {
10      /*从队列中取出数据包*/
11      pkt = xxx_dequeue_buf(dev);
12
13      /*接下来的处理，和中断触发的数据包接收一致*/
14      skb = dev_alloc_skb(pkt->datalen + 2);
15      if (!skb)
16      {
17        ...
18        continue;
19      }
```

```
20      skb_reserve(skb, 2);
21      memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);
22      skb->dev = dev;
23      skb->protocol = eth_type_trans(skb, dev);
24      /*调用 netif_receive_skb 而不是 net_rx 将数据包交给上层协议*/
25      netif_receive_skb(skb);
26
27      /*更改统计数据 */
28      priv->stats.rx_packets++;
29      priv->stats.rx_bytes += pkt->datalen;
30      xxx_release_buffer(pkt);
31    }
32    /* 处理完所有数据包*/
33    *budget -= npackets;
34    dev->quota -= npackets;
35
36    if (!priv->rx_queue)
37    {
38      netif_rx_complete(dev);
39      xxx_enable_rx_int (…); /* 再次使能网络设备的接收中断 */
40      return 0;
41    }
42
43    return 1;
44  }
```

上述代码第 3 行中的 dev->quota 是当前 CPU 能够从所有接口中接收数据包的最大数目，budget 是在初始化阶段分配给接口的 weight 值，poll 函数必须接收两者之间的最小值，表示轮询函数本次要处理的数据包个数。第 8 行的 while()循环读取设备的接收缓冲区，读取数据包并提交给上层。这个过程和中断触发的数据包接收过程一致，但是最后使用 netif_receive_skb()函数而非 netif_rx()函数将数据包提交给上层。这里体现出了中断处理机制和轮询机制之间的差别。

当网络设备接收缓冲区中的数据包都被读取完后（即 priv->rx_queue 为 NULL），一个轮询过程结束，第 38 行代码调用 netif_rx_complete()把当前指定的设备从 poll 队列中清除，第 39 行代码再次启动网络设备的接收中断。

虽然 NAPI 兼容的设备驱动以 poll 方式接收数据包，但是仍然需要首次数据包接收中断来触发 poll 过程。与数据包的中断接收方式不同的是，以轮询方式接收数据包时，当第一次中断发生后，中断处理程序要禁止设备的数据包接收中断，如代码清单 16.12 所示。

代码清单 16.12　网络设备驱动的 poll 中断处理函数模板

```
1   static void xxx_poll_interrupt(int irq, void *dev_id, struct
pt_regs *regs)
2   {
3     switch (status &ISQ_EVENT_MASK)
4     {
5       case ISQ_RECEIVER_EVENT:
6         … /* 获取数据包 */
7         xxx_disable_rx_int(...);  /* 禁止接收中断 */
8         netif_rx_schedule(dev);
9         break;
```

```
10      …  /* 其他类型的中断 */
11    }
12  }
```

上述代码第 8 行的 netif_rx_schedule()函数被轮询方式驱动的中断程序调用，将设备的 poll 方法添加到网络层的 poll 处理队列中，排队并且准备接收数据包，最终触发一个 NET_RX_SOFTIRQ 软中断，通知网络层接收数据包。图 16.4 所示为 NAPI 驱动程序各部分的调用关系。

图 16.4　NAPI 调用关系

16.7 网络连接状态

网络适配器硬件电路可以检测出链路上是否有载波，载波反映了网络的连接是否正常。网络设备驱动可以通过 netif_carrier_on()和 netif_carrier_off()函数改变设备的连接状态，如果驱动检测到连接状态发生变化，也应该以 netif_carrier_on()和 netif_carrier_off()函数显式地通知内核。

除了 netif_carrier_on()和 netif_carrier_off()函数以外，另一个函数 netif_carrier_ok()可用于向调用者返回链路上的载波信号是否存在。

这几个函数都接收一个 net_device 设备结构体指针为参数，原型分别为：

```
void netif_carrier_on(struct net_device *dev);
void netif_carrier_off(struct net_device *dev);
int netif_carrier_ok(struct net_device *dev);
```

网络设备驱动程序中往往设置一个定时器来对链路状态进行周期性地检查。当定时器到期之后，在定时器处理函数中读取物理设备的相关寄存器获得载波状态，从而更新设备的连接状态，如代码清单 16.13 所示。

代码清单 16.13　网络设备驱动用定时器周期检查链路状态

```
static void xxx_timer(unsigned long data)
{
  struct net_device *dev = (struct net_device*)data;
  u16 link;
  …
  if (!(dev->flags &IFF_UP))
  {
    goto set_timer;
  }
```

```
10
11    /* 获得物理上的连接状态 */
12    if (link = xxx_chk_link(dev))
13    {
14      if (!(dev->flags &IFF_RUNNING))
15      {
16        netif_carrier_on(dev);
17        dev->flags |= IFF_RUNNING;
18        printk(KERN_DEBUG "%s: link up\n", dev->name);
19      }
20    }
21    else
22    {
23      if (dev->flags &IFF_RUNNING)
24      {
25        netif_carrier_off(dev);
26        dev->flags &= ~IFF_RUNNING;
27        printk(KERN_DEBUG "%s: link down\n", dev->name);
28      }
29    }
30
31    set_timer:
32    priv->timer.expires = jiffies + 1 * HZ;
33    priv->timer.data = (unsigned long)dev;
34    priv->timer.function = &xxx_timer; /* timer handler */
35    add_timer(&priv->timer);
36  }
```

上述代码第 12 行调用的 xxx_chk_link()函数用于读取网络适配器硬件的相关寄存器以获得链路连接状态，具体实现由硬件决定。当链路连接上时，第 16 行的 netif_carrier_on()函数显式地通知内核链路正常；反之，则第 25 行的 netif_carrier_off()则同样显式地通知内核链路失去连接。

此外，从上述源代码还可以看出，定时器处理函数会不停地利用第 31～35 行代码启动新的定时器以实现周期检测的目的。那么最初启动定时器的地方在哪里呢？很显然，它最适合在设备的打开函数中完成，如代码清单 16.14 所示。

代码清单 16.14　在网络设备驱动的打开函数中初始化定时器

```
1   static int xxx_open(struct net_device *dev)
2   {
3     struct xxx_priv *priv = (struct xxx_priv*)dev->priv;
4
5     ...
6     priv->timer.expires = jiffies + 3 * HZ;
7     priv->timer.data = (unsigned long)dev;
8     priv->timer.function = &xxx_timer; /* timer handler */
9     add_timer(&priv->timer);
10    ...
```

```
}
```

16.8 参数设置和统计数据

在网络设备的驱动程序中还提供一些方法供系统对设备的参数进行设置或读取设备相关的信息。

当用户调用 ioctl()函数，并指定 SIOCSIFHWADDR 命令时，意味着要设置这个设备的 MAC 地址。一般来说，这个操作没有太大的意义。设置网络设备的 MAC 地址可用如代码清单 16.15 所示的模板。

代码清单 16.15　设置网络设备的 MAC 地址

```
static int set_mac_address(struct net_device *dev, void *addr)
{
    if (netif_running(dev))
        return -EBUSY;  /* 设备忙 */

    /* 设置以太网的 MAC 地址 */
    xxx_set_ mac(dev, addr);

    return 0;
}
```

上述程序首先用 netif_running()宏判断设备是否正在运行，如果是，则意味着设备忙，此时不允许设置 MAC 地址；否则，调用 xxx_set_ mac()函数在网络适配器硬件内写入新的 MAC 地址。这要求设备在硬件上支持 MAC 地址的修改，而实际上，许多设备并不提供修改 MAC 地址的接口。

netif_running()宏的定义为：

```
#define netif_running(dev) (dev->flags & IFF_UP)
```

当用户调用 ioctl()函数时，若命令为 SIOCSIFMAP（如在控制台中运行网络配置命令 ifconfig 就会引发这一调用），系统会调用驱动程序的 set_config()函数。

系统会向 set_config()函数传递一个 ifmap 结构体，该结构体中主要包含用户欲设置的设备要使用的 I/O 地址、中断等信息。注意，并非 ifmap 结构体中给出的所有修改都是可以被接受的。实际上，大多数设备并不宜包含 set_config()函数。set_config()函数的例子如代码清单 16.16 所示。

代码清单 16.16　网络设备驱动的 set_config 函数模板

```
int xxx_config(struct net_device *dev, struct ifmap *map)
{
  if (netif_running(dev))   /* 不能设置一个正在运行状态的设备 */
    return  - EBUSY;

  /* 假设不允许改变 I/O 地址 */
  if (map->base_addr != dev->base_addr)
```

```
  {
    printk(KERN_WARNING "xxx: Can't change I/O address\n");
    return  - EOPNOTSUPP;
  }

  /* 假设允许改变 IRQ */
  if (map->irq != dev->irq)
  {
    dev->irq = map->irq;
  }

  return 0;
}
```

上述代码中的 set_config()函数接受 IRQ 的修改，拒绝设备 I/O 地址的修改。具体的设备是否接收这些信息的修改，要视硬件的设计而定。

如果用户调用 ioctl()时，命令类型在 SIOCDEVPRIVATE 和 SIOCDEVPRIVATE+15 之间，系统会调用驱动程序的 do_ioctl()函数，进行设备专用数据的设置。这个设置大多数情况下也并不需要。

驱动程序还应提供 get_stats()函数用以向用户反馈设备状态和统计信息，该函数返回的是一个 net_device_stats 结构体，如代码清单 16.17 所示。

代码清单 16.17　网络设备驱动的 get_stats 函数模板

```
struct net_device_stats *xxx_stats(struct net_device *dev)
{
  struct xxx_priv *priv = netdev_priv(dev);
  return  &priv->stats;
}
```

net_device_stats 结构体定义在内核的 include/linux/netdevice.h 文件中，它包含了比较完整的统计信息，如代码清单 16.18 所示。

代码清单 16.18　net_device_stats 结构体

```
struct net_device_stats
{
  unsigned long rx_packets;        /* 收到的数据包数 */
  unsigned long tx_packets;        /* 发送的数据包数 */
  unsigned long rx_bytes;      /* 收到的字节数  */
  unsigned long tx_bytes;      /* 发送的字节数  */
  unsigned long rx_errors;         /* 收到的错误数据包数 */
  unsigned long tx_errors;         /* 发生发送错误的数据包数 */
  ...
};
```

上述代码清单只是列出了 net_device_stats 包含的主项目统计信息，实际上，这些项目还可以进一步细分，net_device_stats 中的其他信息给出了更详细的子项目统计，详见 Linux 源代码。

net_device_stats 结构体适宜包含在设备的私有信息结构体中，而其中统计信息的修改则应该在设备驱动的与发送和接收相关的具体函数中完成，这些函数包括中断处理程序、数据包发送函数、数据包发送超时函数和数据包接收相关函数等。我们应该在这些函数中添加相应的代码，如代码清单 16.19 所示。

代码清单 16.19　net_device_stats 结构体中统计信息的维护

```
1   /* 发送超时函数 */
2   void xxx_tx_timeout(struct net_device *dev)
3   {
4     struct xxx_priv *priv = netdev_priv(dev);
5     ...
6     priv->stats.tx_errors++;  /* 发送错误包数加 1 */
7     ...
8   }
9
10  /* 中断处理函数 */
11  static void xxx_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
12  {
13    switch (status &ISQ_EVENT_MASK)
14    {
15      ...
16      case ISQ_TRANSMITTER_EVENT:  /
17        priv->stats.tx_packets++;   /* 数据包发送成功，tx_packets 信息加
1 */
18        netif_wake_queue(dev); /* 通知上层协议 */
19        if ((status &(TX_OK | TX_LOST_CRS | TX_SQE_ERROR |
20        TX_LATE_COL | TX_16_COL)) != TX_OK) /*读取硬件上的出错标志*/
21        {
22          /* 根据错误的不同情况，对 net_device_stats 的不同成员加 1 */
23          if ((status &TX_OK) == 0)
24            priv->stats.tx_errors++;
25          if (status &TX_LOST_CRS)
26            priv->stats.tx_carrier_errors++;
27          if (status &TX_SQE_ERROR)
28            priv->stats.tx_heartbeat_errors++;
29          if (status &TX_LATE_COL)
30            priv->stats.tx_window_errors++;
31          if (status &TX_16_COL)
32            priv->stats.tx_aborted_errors++;
33        }
34        break;
35      case ISQ_RX_MISS_EVENT:
36        priv->stats.rx_missed_errors += (status >> 6);
37        break;
38      case ISQ_TX_COL_EVENT:
39        priv->stats.collisions += (status >> 6);
40        break;
41    }
42  }
```

上述代码的第 6 行意味着在发送数据包超时时，将发生发送错误的数据包数增 1。

而第 13～41 行则意味着当网络设备中断产生时，中断处理程序读取硬件的相关信息以决定修改 net_device_ stats 统计信息中的哪些项目和子项目，并将相应的项目增 1。

16.9 CS8900 网卡设备驱动实例

16.9.1 CS8900 网卡硬件描述

CS8900 芯片是 Cirrus Logic 公司生产的一种以太网处理芯片，提供 ISA 总线接口，内部集成了 E^2PROM 控制器、802.3 MAC 控制器、片内 RAM 和 10BASE-T 收发滤波器，支持 I/O 和内存模式两种模式，可采用 DMA 方式与主机以 16 位或 8 位数据长度进行数据交换。图 16.5 所示为 CS8900 的内部结构框架。

图 16.5　CS8900 以太网芯片的内部结构

在一般的嵌入式系统硬件原理设计时，CS8900 可以直接挂接在 CPU 的存储器总线上。将 CS8900A 网卡连接在 S3C2410A ARM 处理器的数据总线、控制总线和地址总线上的原理如图 16.6 所示。

CS8900 采用了独特的 PacketPage（封包页）机制来实现总线对其内部 4KB RAM 空间的访问。位于 CS8900 PacketPage RAM 空间内的地址只需要通过直接映射到 CPU 内存或 I/O 空间的 16 字节连续的可直接访问的端口中的 PacketPage 指针端口和 PacketPage 数据端口即可间接访问，最大程度地减少了 CS8900 对硬件资源的占用。

图 16.6　CS8900 与 S3C2410A 连接原理

当 CS8900 处于 I/O 模式下时，可以通过以下几个 PacketPage 空间内的寄存器来控制 CS8900 的行为（括号内给出的是寄存器地址相对于 PacketPage 基地址的偏移）。

- LINECTL(0112H)。

LINECTL 决定 CS8900 的基本配置和物理接口，可选择使用 10BASE-T 接口、AUI 接口或者自动选择。

- RXCTL(0104H)。

RXCTL 控制 CS8900 接收特定数据包，控制是否接收多播、广播和单播包。

- RXCFG(0102H)。

RXCFG 控制 CS8900 接收到特定数据包后引发接收中断，并控制是否使用接收 DMA 和 CRC 校验。

- BUSCT(0116H)。

BUSCT 可控制芯片的工作模式、DMA 方式、是否使能外部中断引脚。

- BUSST(0138H)。

BUSST 标志网络设备的发送状态，如设备是否准备好发送。

- ISQ(0120H)。

ISQ 是网卡芯片的中断状态寄存器。

当 CS8900 复位时，16 字节连续的端口被分配在偏移地址 300H 处，这个地址就是 CS8900 的 I/O 基地址。当然，I/O 基地址可以通过 PacketPage 空间内的相对于 PacketPage 基地址偏移 0020H 地址处的 I/O 基地址寄存器重新设置，但一般没有这个修改的必要。

通过下面的 3 个寄存器可完成数据包的发送和接收（括号内为寄存器地址相对 I/O 基地址的偏移）。

- PORT0(0000H)。

发送或接收数据包时，CPU 都通过 PORT0 写入或读取数据。

- TXCMD(0004H)。

发送控制寄存器，填写相应的发送命令后会引发数据包的发送。

- TXLENG(0006H)。

发送数据包长度寄存器，发送数据时，应该首先写入发送数据长度，然后将数据通过 PORT0 写入芯片。

CS8900 工作时，应首先对其进行初始化，即在寄存器 LINECTL、RXCTL、RCCFG、BUSCT 中写入正确的配置字。

如图 16.7 所示，在 I/O 模式下，CS8900 发送数据包的步骤如下。

（1）向控制寄存器 TXCMD 寄存器写入发送命令。

（2）将发送数据长度写入 TXLENG 寄存器。

（3）读取 PacketPage 空间内的 BUSST 寄存器，确定其第 8 位被设置为 Rdy4TxNOW，即设备处于准备发送状态。

（4）将要发送的数据循环写入 PORT0 寄存器。

完成上述寄存器操作后，CS8900 会将数据组织为以太网帧并添加填充位和 CRC 校验信息，然后将数据转化为比特流传送到网络媒介。

如图 16.8 所示，在 I/O 模式下，CS8900 接收数据包的方法如下。

（1）接收到网络适配器产生的中断，查询相对于 I/O 基地址偏移 0008H 中断状态队列端口，判断中断类型为接收中断。

（2）读 PORT0 寄存器依次获得接收状态 rxStatus、接收数据长度 rxLength。
（3）循环继续对 PORT0 寄存器读取 rxLength 次，获得整个数据包。

图 16.7　CS8900 数据包发送流程　　图 16.8　CS8900 数据包接收流程

这里所说的 CS8900 处于 I/O 模式并非意味着它一定位于 CPU 的 I/O 空间，实际上，CS8900 I/O 模式下的寄存器仍然映射 ARM 处理器的内存空间。因此，我们直接通过读/写寄存器地址 ioremap()之后的虚拟地址即可完成与代码清单 16.4 对应的 CS8900 寄存器读写函数。

16.9.2 CS8900 网卡驱动设计分析

总的来说，CS8900 是一种具体的网络设备，是通用的抽象的网络设备驱动体系结构和设计模板的例化。

对于一种网络设备的驱动而言，工程师主要需完成设备驱动功能层的设计。在 16.2～16.8 节已经给出了设备驱动功能层主要数据结构和函数的设计模板，因此，在编写 CS8900 的这些数据结构和函数时，实际要完成的工作就是用具体的针对 CS8900 的操作来填充模板，具体包括以下工作。

- 填充 CS8900 的私有信息结构体，把 CS8900 特定的数据放入这个私有结构体中。在 CS8900 驱动程序中，这个数据结构为 struct net_local。
- 填充设备初始化模板，初始化 net_device 结构体，将其注册入内核。net_device 的注册与注销在模块加载与注销函数中完成。在 CS8900 驱动程序中，与此相关的函数有：

```
struct net_device * __init cs89x0_probe(int unit);
int cs89x0_probe1(struct net_device *dev, int ioaddr, int modular);
int init_module(void);
void cleanup_module(void);
```

- 填充设备发送数据包函数模板，把真实的数据包发送硬件操作填充入 xxx_tx()函数，并填充发送超时函数 xxx_tx_timeout()。在 CS8900 驱动程序中，与此相关的函数有：

```
static int net_send_packet(struct sk_buff *skb, struct net_device *dev);
static void net_timeout(struct net_device *dev);
```

- 填充设备驱动程序的中断处理程序 xxx_interrupt()和具体的数据包接收函数 xxx_rx()，填入真实的硬件操作。在 CS8900 驱动程序中，与此相关的函数有：

```
irqreturn_t net_interrupt(int irq, void *dev_id, struct pt_regs *regs);
void net_rx(struct net_device *dev);
```

- 填充设备打开 xxx_open()与释放 xxx_release()函数代码。在 CS8900 驱动程序中，与此相关的函数有：

```
int net_open(struct net_device *dev);
int net_close(struct net_device *dev);
```

- 填充设备配置与数据统计的具体代码，填充返回设备冲突的 xxx_stats()函数。

图 16.9 所示为 16.2～16.8 节给出的设备驱动功能层抽象设计模板与上述 CS8900 的具体函数和数据结构间的映射关系。

图 16.9　CS8900 驱动与设计模板的映射

下面将具体讲解 CS8900 驱动的若干数据结构和函数，由于整个代码量相当庞大，我们只对其比较复杂的注册与注销、初始化、数据包发送与接口函数进行分析。

16.9.3　CS8900 网卡注册、初始化与注销

在 CS8900 的设备驱动程序中，核心数据结构 net_device 以全局变量的方式定义，其数个成员的初始值都被置为空，如代码清单 16.20 所示。

代码清单 16.20　CS8900 的 net_device 实例

```
static struct net_device dev_cs89x0 = {
        "",
        0, 0, 0, 0,
        0, 0,
        0, 0, 0, NULL, NULL };
```

驱动程序为 CS8900 网卡设备定义的私有信息结构体为 net_local，如代码清单 16.21 所示。

代码清单 16.21　CS8900 私有信息结构体 net_local

```
struct net_local {
    struct net_device_stats stats;   /* 网络设备状态结构体 */
    int chip_type;   /* 区分芯片类型: CS89x0 */
    char chip_revision; /* 芯片版本字母，如"A" */
    int send_cmd; /* 发送命令: TX_NOW, TX_AFTER_381 或 TX_AFTER_ALL
*/
```

```
    6       ...
    7       spinlock_t lock;    /* 并发控制自旋锁 */
    8   };
```

当芯片的版本字母不同时，net_local 结构体中记录的 send_cmd 命令将不同。例如，同样是 CS8900 网卡，若芯片版本字母为大于等于“F”，则发送命令为 TX_NOW，而对于 CS8900A，发送命令为 TX_AFTER_ALL。

CS8900 设备驱动程序的初始化由 net_device 结构体中的 init()函数完成，这个函数将在 net_device 被注册时自动被调用。init()函数在 CS8900 网卡的驱动程序中对应于 cs89x0_probe()函数，如代码清单 16.22 所示。

代码清单 16.22　CS8900 设备驱动的初始化函数

```
    1   int __init cs89x0_probe(struct net_device *dev) {
    2       int i;
    3
    4       SET_MODULE_OWNER(dev);
    5       DPRINTK(1, "cs89x0:cs89x0_probe(0x%x)\n", base_addr);
    6
    7       BWSCON = (BWSCON & ~(BWSCON_ST3 | BWSCON_WS3 | BWSCON_DW3)) |
    8         (BWSCON_ST3 | BWSCON_WS3 | BWSCON_DW(3, BWSCON_DW_16));
    9       BANKCON3= BANKCON_Tacs0 | BANKCON_Tcos4 | BANKCON_Tacc14 |
    10        BANKCON_Toch1 | BANKCON_Tcah4 | BANKCON_Tacp6 | BANKCON_PMC1;
    11
    12      set_external_irq(IRQ_CS8900,                EXT_RISING_EDGE,
GPIO_PULLUP_DIS);
    13
    14      for (i = 0; netcard_portlist[i]; i++) {
    15        if (cs89x0_probe1(dev, netcard_portlist[i]) == 0)
    16              return 0;
    17      }
    18      printk(KERN_WARNING "cs89x0: no cs8900 or cs8920 detected."
    19         "Be sure to disable PnP with SETUP\n");
    20      return -ENODEV;
    21  }
```

上述函数第 7～10 行设置 S3C2410A ARM 处理器的片选，第 12 行设置 ARM 与 CS8900 网卡对应的中断，第 14～17 行循环检索 netcard_portlist[ ]数组中定义的基地址处是否存在 CS8900 网卡。对于我们的嵌入式硬件平台，netcard_portlist[ ]数组的定义将很简单，基地址为 vCS8900_BASE + 0x300，因而 netcard_portlist[]数组的定义如代码清单 16.23 所示。

代码清单 16.23　要探测的 CS8900 网卡基地址列表

```
1   static unsigned int netcard_portlist[] __initdata =
2     { vCS8900_BASE + 0x300, 0};
```

代码清单 16.22 中第 15 行调用的 cs89x0_probe1()函数读取 vCS8900_BASE + 0x300 基地址处的 CS8900 相关的寄存器，以“反证法”验证网卡的存在，并获取 CS8900 所使用的硬件资源。代码清单 16.24 所示为 cs89x0_probe1()函数。

代码清单 16.24　探测 CS8900 并获取硬件资源

```
1   static int __init cs89x0_probe1(struct net_device *dev, int ioaddr)
2   {
3     struct net_local *lp;
4     unsigned rev_type = 0;
5     int ret;
6
7     /* 初始化设备结构体私有信息 */
8     if (dev->priv == NULL)
9     {
10      dev->priv = kmalloc(sizeof(struct net_local), GFP_KERNEL);
11      if (dev->priv == 0)
12      {
13        ret =  - ENOMEM;
14        goto before_kmalloc;
15      }
16      lp = (struct net_local*)dev->priv;
17      memset(lp, 0, sizeof(*lp));
18      spin_lock_init(&lp->lock);
19    }
20    lp = (struct net_local*)dev->priv;
21
22    dev->base_addr = ioaddr;
23    /* 读取芯片类型 */
24    rev_type = readreg(dev, PRODUCT_ID_ADD);
25    lp->chip_type = rev_type &~REVISON_BITS;
26    lp->chip_revision = ((rev_type &REVISON_BITS) >> 8) + 'A';30
27    if (lp->chip_type != CS8900)
28    {
29      printk(__FILE__ ": wrong device driver!\n");
30      ret =  - ENODEV;
31      goto after_kmalloc;
```

```
    32    }
    33    /* 根据芯片类型和版本确定正确的发送命令 */
    34    lp->send_cmd = TX_AFTER_ALL;
    35    if (lp->chip_type == CS8900 && lp->chip_revision >= 'F')
    36      lp->send_cmd = TX_NOW;
    37
    38    reset_chip(dev);
    39
    40    lp->adapter_cnf = A_CNF_10B_T | A_CNF_MEDIA_10B_T;
    41    lp->auto_neg_cnf = EE_AUTO_NEG_ENABLE;43
    42    printk(KERN_INFO   "cs89x0   media   %s%s",   (lp->adapter_cnf
&A_CNF_10B_T) ?
    43      "RJ-45": "", (lp->adapter_cnf &A_CNF_AUI) ? "AUI" : "");
    44
    45    /* 设置 CS8900 的 MAC 地址 */
    46    dev->dev_addr[0] = 0x00;
    47    dev->dev_addr[1] = 0x00;
    48    dev->dev_addr[2] = 0xc0;
    49    dev->dev_addr[3] = 0xff;
    50    dev->dev_addr[4] = 0xee;
    51    dev->dev_addr[5] = 0x08;
    52    set_mac_address(dev, dev->dev_addr);
    53
    54    /* 设置设备中断号 */
    55    dev->irq = IRQ_LAN;
    56    printk(", IRQ %d", dev->irq);
    57
    58    /* 填充设备结构体的成员函数指针 */
    59    dev->open = net_open;
    60    dev->stop = net_close;
    61    dev->tx_timeout = net_timeout;
    62    dev->watchdog_timeo = 3 * HZ;
    63    dev->hard_start_xmit = net_send_packet;
    64    dev->get_stats = net_get_stats;
    65    dev->set_multicast_list = set_multicast_list;
    66    dev->set_mac_address = set_mac_address;
    67
    68    /* 填充以太网公用数据和函数指针 */
    69    ether_setup(dev);
    70
    71    printk("\n");
    72    DPRINTK(1, "cs89x0_probe1() successful\n");
    73    return 0;
```

```
74
75    after_kmalloc: kfree(dev->priv);
76    before_kmalloc: return ret;
77  }
```

上述 cs89x0_probe1()函数的流程如下。

（1）第 8～20 行分配设备的私有信息结构体内存并初始化，若分配失败，则直接跳入第 78 行的代码返回。

（2）第 24～26 行从寄存器中读取芯片的具体类型。

（3）第 27～32 行判断芯片类型，若不是 CS8900 则直接跳入第 77 行的代码，释放私有信息结构体并返回。

（4）当芯片类型为 CS8900 时，第 34～69 行完成 net_device 设备结构体的初始化，赋值其属性和函数指针。

下面讲解 CS8900 的模块加载函数，该函数用来注册经过 cs89x0_probe()和 cs89x0_probe1()函数初始化过的 net_device 设备结构体，如代码清单 16.25 所示。

代码清单 16.25　CS8900 网卡设备驱动的模块加载函数

```
1   static int __init init_cs8900a_s3c2410(void)
2   {
3     struct net_local *lp;
4     int ret = 0;
5
6     dev_cs89x0.irq = irq;
7     dev_cs89x0.base_addr = io;
8     dev_cs89x0.init = cs89x0_probe;
9     dev_cs89x0.priv = kmalloc(sizeof(struct net_local), GFP_KERNEL);
10    if (dev_cs89x0.priv == 0)
11    {
12      printk(KERN_ERR "cs89x0.c: Out of memory.\n");
13      return  - ENOMEM;
14    }
15    memset(dev_cs89x0.priv, 0, sizeof(struct net_local));
16    lp = (struct net_local*)dev_cs89x0.priv;
17
18    request_region(dev_cs89x0.base_addr,        NETCARD_IO_EXTENT,
"cs8900a");
19
20    spin_lock_init(&lp->lock);
21
22    /* 设置物理接口的正确类型*/
```

```
23    if (!strcmp(media, "rj45"))
24      lp->adapter_cnf = A_CNF_MEDIA_10B_T | A_CNF_10B_T;
25    else if (!strcmp(media, "aui"))
26      lp->adapter_cnf = A_CNF_MEDIA_AUI | A_CNF_AUI;
27    else if (!strcmp(media, "bnc"))
28      lp->adapter_cnf = A_CNF_MEDIA_10B_2 | A_CNF_10B_2;
29    else
30      lp->adapter_cnf = A_CNF_MEDIA_10B_T | A_CNF_10B_T;
31
32    if (duplex ==  - 1)
33      lp->auto_neg_cnf = AUTO_NEG_ENABLE;
34
35    if (io == 0)
36    {
37      printk(KERN_ERR "cs89x0.c: Module autoprobing not allowed.\n");
38      printk(KERN_ERR "cs89x0.c: Append io=0xNNN\n");
39      ret =  - EPERM;
40      goto out;
41    }
42
43    if (register_netdev(&dev_cs89x0) != 0)
44    {
45      printk(KERN_ERR "cs89x0.c: No card found at 0x%x\n", io);
46      ret =  - ENXIO;
47      goto out;
48    }
49    out: if (ret)
50      kfree(dev_cs89x0.priv);
51    return ret;
52  }
```

上述代码第 8 行将 cs89x0_probe()函数赋值给结构体 net_device 的 init 指针，这样，在使用 register_netdev()函数注册 net_device 设备结构体时，cs89x0_probe()函数会被自动调用以完成 net_device 结构体的初始化。

第 18 行调用 request_region()函数为 CS8900 网卡申请了 NETCARD_IO_EXTENT 大小的 I/O 地址区域。

第 43 行完成已经被 cs89x0_probe()函数初始化的 net_device 设备结构体的注册。

CS8900 驱动的模块卸载函数需完成与模块加载函数相反的功能，如代码清单 16.26 所示。

代码清单 16.26　CS8900 网卡设备驱动的模块卸载函数

```
1   static void __exit cleanup_cs8900a_s3c2410(void)
2   {
3     if (dev_cs89x0.priv != NULL)
4     {
5       /* 释放私有信息结构体  */
6       unregister_netdev(&dev_cs89x0);
7       outw(PP_ChipID, dev_cs89x0.base_addr + ADD_PORT);
8       kfree(dev_cs89x0.priv);
9       dev_cs89x0.priv = NULL;
10      /* 释放 CS8900 申请的 I/O 地址区域 */
```

```
11      release_region(dev_cs89x0.base_addr, NETCARD_IO_EXTENT);
12    }
13  }
```

上述代码首先注销了 net_device 设备结构体，然后释放 CS8900 的私有信息结构体和在模块加载函数中申请的 I/O 地址区域。

16.9.4 CS8900 网卡发送数据流程

从代码清单 16.24 即 CS8900 的初始化函数可以看出，CS8900 的数据包发送函数指针 hard_ start_xmit 被赋值为 CS8900 驱动程序中的 net_send_packet()，这个函数完成 16.9.1 小节所描述的硬件发送序列，代码清单为 16.27 所示。

代码清单 16.27　CS8900 网卡驱动的发送函数

```
1   static int net_send_packet(struct sk_buff *skb, struct net_device
*dev)
2   {
3     struct net_local *lp = (struct net_local*)dev->priv;
4
5     writereg(dev, PP_BusCTL, 0x0);
6     writereg(dev, PP_BusCTL, readreg(dev, PP_BusCTL) | ENABLE_IRQ);
7
8     spin_lock_irq(&lp->lock);/* 使用自旋锁阻止多个数据包被同时写入硬件
*/
9     netif_stop_queue(dev);
10
11    /* 初始化硬件发送序列 */
12    writeword(dev, TX_CMD_PORT, lp->send_cmd);
13    writeword(dev, TX_LEN_PORT, skb->len);
14
15    /* 检测硬件是否处于发送 READY 状态 */
16    if ((readreg(dev, PP_BusST) &READY_FOR_TX_NOW) == 0)
17    {
18      spin_unlock_irq(&lp->lock);
19      DPRINTK(1, "cs89x0: Tx buffer not free!\n");
20      return 1;
21    }
22
23    writeblock(dev, skb->data, skb->len);/* 将数据写入硬件 */
24
25    spin_unlock_irq(&lp->lock);       /* 解锁自旋锁 */
26    dev->trans_start = jiffies;       /* 记录发送开始的时间戳 */
27    dev_kfree_skb(skb);               /* 释放 sk_buff 和数据缓冲区 */
28
29    return 0;
30  }
```

上述代码第 12～23 行完成硬件操作，即按照发送序列在 CS8900 的寄存器中写入相应的命令或数据。整个硬件发送序列都处于第 8 行锁定的自选锁的控制之下，直到

数据包被写入硬件，才在代码的第 25 行对自旋锁解除锁定。

第 26 行记录最后数据包发送的时间戳。

第 27 行释放了从上层协议传递下来的 sk_buff 套接字缓冲区和数据缓冲区。

当发送数据包超时时，CS8900 驱动程序的数据包发送超时函数将被调用，它重新启动设备发送队列，如代码清单 16.28 所示。

代码清单 16.28　CS8900 网卡驱动的发送超时函数

```
1   static void net_timeout(struct net_device *dev)
2   {
3       DPRINTK(1, "%s: transmit timed out, %s?\n", dev->name,
4           tx_done(dev) ? "IRQ conflict ?" : "network cable problem");
5
6       net_close(dev);
7       writereg(dev,   PP_SelfCTL,   readreg(dev,   PP_SelfCTL)   |
POWER_ON_RESET);
8       net_open(dev);
9   }
```

上述代码处理发送超时的策略是：第 6 行停止网卡，第 7 行进行网卡硬复位，在复位后第 8 行再次启动 CS8900 网卡。

16.9.5　CS8900 网卡接收数据流程

CS8900 网卡设备的数据包接收流程由中断引发，在中断服务程序中会判断中断的类型，如果是接收到数据包中断，则完成套接字缓冲区的创建和填充，并调用 netif_rx()函数将套接字缓冲区传递给上层协议，如代码清单 16.29 所示。

代码清单 16.29　CS8900 网卡驱动的中断处理函数

```
static void net_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
{
  struct net_device *dev = dev_id;
  struct net_local *lp;
  int ioaddr, status;

  ioaddr = dev->base_addr;
  lp = (struct net_local*)dev->priv;

  /* 读取中断事件类型 */
  while ((status = readword(dev, ISQ_PORT)))
  {
    DPRINTK(4, "%s: event=%04x\n", dev->name, status);
    switch (status &ISQ_EVENT_MASK)
    {
```

```
16        case ISQ_RECEIVER_EVENT:
17          /* 获得数据包 */
18          net_rx(dev);
19          break;
20        ... /* 其他类型中断 */
21      }
22    }
23  }
```

上述程序的第 18 行意味着网卡中断为接收中断时，调用 net_rx()函数完成数据包的获取与向上层协议传递。因此，net_rx()函数相当关键，如代码清单 16.30 所示。

代码清单 16.30　CS8900 网卡驱动的数据包接收函数

```
1   static void net_rx(struct net_device *dev)
2   {
3     struct net_local *lp = (struct net_local*)dev->priv;
4     struct sk_buff *skb;
5     int status, length;
6
7     int ioaddr = dev->base_addr;
8
9     status = inw(ioaddr + RX_FRAME_PORT);
10    if ((status &RX_OK) == 0)
11    {
12      count_rx_errors(status, lp);
13      return ;
14    }
15
16    length = inw(ioaddr + RX_FRAME_PORT);/* 读取接收数据包的长度 */
17
18    /* 分配新的套接字缓冲区和数据缓冲区 */
19    skb = dev_alloc_skb(length + 2);
20    if (skb == NULL)
21    {
22      lp->stats.rx_dropped++; /* 分配失败，统计被丢弃的包数 */
23      return ;
24    }
25    skb_reserve(skb, 2);
26    skb->len = length;
27    skb->dev = dev;29
28    readblock(dev, skb->data, skb->len); /* 将硬件中读取数据包放入数据
缓冲区 */
29
30    skb->protocol = eth_type_trans(skb, dev);/* 解析协议类型 */

31    netif_rx(skb);      /* 传递给上层协议 */
32
33    dev->last_rx = jiffies;   /* 记录最后收到数据包的时间戳 */
34    /* 统计接收数据包数和接收字节数 */
35    lp->stats.rx_packets++;
```

```
36    lp->stats.rx_bytes += length;
37  }
```

上述代码的第 16 行从硬件中读取收到的数据包长度 length，根据这一结果，第 19 行分配套接字缓冲区和数据缓冲区，第 28 行从硬件中读取数据包到分配的数据缓冲区中，第 30 行利用内核提供的 eth_type_trans()函数解析收到数据包的网络层协议类型并赋值给 skb->protocol，第 31 行利用网络协议接口层的 netif_rx()函数将数据包递交给上层协议。程序中最后的第 33～36 行记录接收时间戳并完成信息统计。

16.10 总结

Linux 网络设备驱动体系结构的层次化设计实现了对上层协议接口的统一和硬件驱动的对下层多样化硬件设备的可适应。程序员需要完成的工作集中在设备驱动功能层，网络设备接口层 net_device 结构体的存在将千变万化的网络设备得以抽象，使得设备功能层中除数据包接收以外的主体工作都由填充 net_device 的属性和函数指针完成。

在分析 net_device 数据结构的基础上，本章给出了设备驱动功能层设备初始化、数据包收发、打开和释放等函数的设计模板，这些模板对实际设备驱动的开发具有直接指导意义。有了这些模板，我们在设计具体设备的驱动时，不再需要关心程序的体系，而可以集中精力于硬件操作本身。从对 CS8900 网卡的驱动分析可知，其驱动的主要函数和数据结构和模板可以良好地映射。

在 Linux 网络子系统和设备驱动中，套接字缓冲区 sk_buff 发挥着巨大的作用，是所有数据流动的载体。网络设备驱动和上层协议之间也依赖此结构进行数据包交互，因此，我们要特别牢记它的操作方法。

华清远见

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

· 招生简章： http://www.embedu.org/courses/index.htm

· 课程内容： http://www.embedu.org/courses/course1.htm

· 项目实战： http://www.embedu.org/courses/project.htm

· 出版教材： http://www.embedu.org/courses/course3.htm

嵌入式学院 华清远见旗下品牌

第 17 章　Linux 音频设备驱动

在 Linux 系统中，先后出现了音频设备的两种框架：OSS 和 ALSA，本节将在介绍数字音频设备及音频设备硬件接口的基础上讲解 OSS 和 ALSA 驱动的结构。

17.1～17.2 节讲解了音频设备及 PCM、IIS 和 AC97 硬件接口。

17.3 节讲解了 Linux OSS 音频设备驱动的组成、mixer 接口、dsp 接口及用户空间编程方法。

17.4 节讲解了 Linux ALSA 音频设备驱动的组成、card 和组件管理、PCM 设备、control 接口、AC97 API 及用户空间编程方法。

17.5 节以 S3C2410 通过 IIS 接口外接 UDA1341 编解码器的实例讲解了 OSS 驱动。

17.6 节以 PXA255 通过 AC97 接口外接 AC97 编解码器的实例讲解了 ALSA 驱动。

华清远见

17.1 数字音频设备

目前，手机、PDA、MP3 等许多嵌入式设备中包含了数字音频设备，一个典型的数字音频系统的电路组成如图 17.1 所示。图 17.1 中的嵌入式微控制器/DSP 中集成了 PCM、IIS 或 AC97 音频接口，通过这些接口连接外部的音频编解码器即可实现声音的 AD 和 DA 转换，图中的功放完成模拟信号的放大功能。

图 17.1　典型的数字音频系统电路

音频编解码器是数字音频系统的核心，衡量它的主要指标如下。

1．采样频率

采样的过程就是将通常的模拟音频信号的电信号转换成二进制码 0 和 1 的过程，这些 0 和 1 便构成了数字音频文件。图 17.2 中的正弦曲线代表原始音频曲线，方格代表采样后得到的结果，二者越吻合说明采样结果越好。

图 17.2　数字音频采样

采样频率是每秒钟的采样次数，我们常说的 44.1kHz 采样频率就是每秒钟采样 44100 次。理论上采样频率越高，转换精度越高，目前主流的采样频率是 48kHz。

2．量化精度

量化精度是指对采样数据分析的精度，比如 24bit 量化精度就是指将标准电平信号按照 2 的 24 次方进行分析，也就是说将图 17.2 中的纵坐标等分为 2^{24} 等分。量化精度越高，声音就越逼真。

17.2 音频设备硬件接口

17.2.1　PCM 接口

针对不同的数字音频子系统，出现了几种微处理器或 DSP 与音频器件间用于数字转换的接口。

最简单的音频接口是 PCM（脉冲编码调制）接口，该接口由时钟脉冲（BCLK）、帧同步信号（FS）及接收数据（DR）和发送数据（DX）组成。在 FS 信号的上升沿，数据传输从 MSB（Most Significant Bit）开始，FS 频率等于采样率。FS 信号之后开始数据字的传输，单个的数据位按顺序进行传输，一个时钟周期传输一个数据字。发送 MSB 时，信号的等级首先降到最低，以避免在不同终端的接口使用不同的数据方案时造成 MSB 的丢失。

PCM 接口很容易实现，原则上能够支持任何数据方案和任何采样率，但需要每个音频通道获得一个独立的数据队列。

17.2.2　IIS 接口

IIS 接口（Inter-IC Sound）在 20 世纪 80 年代首先被 PHILIPS 用于消费音频产品，并在

一个称为 LRCLK（Left/Right CLOCK）的信号机制中经过多路转换，将两路音频信号变成单一的数据队列。当 LRCLK 为高时，左声道数据被传输；LRCLK 为低时，右声道数据被传输。与 PCM 相比，IIS 更适合于立体声系统。对于多通道系统，在同样的 BCLK 和 LRCLK 条件下，并行执行几个数据队列也是可能的。

17.2.3 AC97 接口

AC'97（Audio Codec 1997）是以 Intel 为首的 5 个 PC 厂商 Intel、Creative Labs、NS、Analog Device 与 Yamaha 共同提出的规格标准。与 PCM 和 IIS 不同，AC'97 不只是一种数据格式，用于音频编码的内部架构规格，它还具有控制功能。AC'97 采用 AC-Link 与外部的编解码器相连，AC-Link 接口包括位时钟（BITCLK）、同步信号校正（SYNC）和从编码到处理器及从处理器中解码（SDATDIN 与 SDATAOUT）的数据队列。AC'97 数据帧以 SYNC 脉冲开始，包括 12 个 20 位时间段（时间段为标准中定义的不同的目的服务）及 16 位“tag”段，共计 256 个数据序列。例如，时间段“1”和“2”用于访问编码的控制寄存器，而时间段“3”和“4”分别负载左、右两个音频通道。“tag”段表示其他段中哪一个包含有效数据。把帧分成时间段使传输控制信号和音频数据仅通过 4 根线到达 9 个音频通道或转换成其他数据流成为可能。与具有分离控制接口的 IIS 方案相比，AC97 明显减少了整体管脚数。一般来说，AC'97 编解码器采用 TQFP48 封装，如图 17.3 所示。

PCM、IIS 和 AC97 各有其优点和应用范围，例如在 CD、MD、MP3 随身听多采用 IIS 接口，移动电话会采用 PCM 接口，具有音频功能的 PDA 则多使用和 PC 一样的 AC'97 编码格式。

图 17.3　AC97 Codec 芯片

17.3 Linux OSS 音频设备驱动

17.3.1　OSS 驱动的组成

OSS 标准中有两个最基本的音频设备：mixer（混音器）和 dsp（数字信号处理器）。

在声卡的硬件电路中，mixer 是一个很重要的组成部分，它的作用是将多个信号组合或者叠加在一起，对于不同的声卡来说，其混音器的作用可能各不相同。OSS 驱动中，/dev/mixer 设备文件是应用程序对 mixer 进行操作的软件接口。

混音器电路通常由两部分组成：输入混音器（input mixer）和输出混音器（output mixer）。输入混音器负责从多个不同的信号源接收模拟信号，这些信号源有时也被称为混音通道或者混音设备。模拟信号通过增益控制器和由软件控制的音量调节器，在不同的混音通道中进行级别（level）调制，然后被送到输入混音器中进行声音的合成。混音器上的电子开关可以控制哪些通道中有信号与混音器相连，有些声卡只允许连接一个混音通道作为录音的音源，而有些声卡则允许对混音通道做任意的连接。经过输入混音器处理后的信号仍然为模拟信号，它们将被送到 A/D 转换器进行数字化处理。

输出混音器的工作原理与输入混音器类似，同样也有多个信号源与混音器相连，并且事先都经过了增益调节。当输出混音器对所有的模拟信号进行了混合之后，通常还会有一个总控增益调节器来控制输出声音的大小，此外还有一些音调控制器来调节输出声音的音调。经过输出混音器处理后的信号也是模拟信号，它们最终会被送给喇叭或者其他的模拟输出设备。

对混音器的编程包括如何设置增益控制器的级别，以及怎样在不同的音源间进行切换，这些操作通常来讲是不连续的，而且不会像录音或者播放那样需要占用大量的计算机资源。由于混音器的操作不符合典型的读/写操作模式，因此除了 open()和 close()这两个系统调用之外，大部分的操作都是通过 ioctl()系统调用来完成的。与/dev/dsp 不同，/dev/mixer 允许多个应用程序同时访问，并且混音器的设置值会一直保持到对应的设备文件被关闭为止。

DSP 也称为编解码器，实现录音（录音）和放音（播放），其对应的设备文件是/dev/dsp 或/dev/sound/dsp。OSS 声卡驱动程序提供的/dev/dsp 是用于数字采样和数字录音的设备文件，向该设备写数据即意味着激活声卡上的 D/A 转换器进行播放，而向该设备读数据则意味着激活声卡上的 A/D 转换器进行录音。

从 DSP 设备读取数据时，从声卡输入的模拟信号经过 A/D 转换器变成数字采样后的样本，保存在声卡驱动程序的内核缓冲区中，当应用程序通过 read()系统调用从声卡读取数据时，保存在内核缓冲区中的数字采样结果将被复制到应用程序所指定的用户缓冲区中。

需要指出的是，声卡采样频率是由内核中的驱动程序所决定的，而不取决于应用程序从声卡读取数据的速度。如果应用程序读取数据的速度过慢，以致低于声卡的采

样频率，那么多余的数据将会被丢弃（即 overflow）；如果读取数据的速度过快，以致高于声卡的采样频率，那么声卡驱动程序将会阻塞那些请求数据的应用程序，直到新的数据到来为止。

向 DSP 设备写入数据时，数字信号会经过 D/A 转换器变成模拟信号，然后产生声音。应用程序写入数据的速度应该至少等于声卡的采样频率，过慢会产生声音暂停或者停顿的现象（即 underflow）。如果用户写入过快的话，它会被内核中的声卡驱动程序阻塞，直到硬件有能力处理新的数据为止。

与其他设备有所不同，声卡通常不需要支持非阻塞（non-blocking）的 I/O 操作。即便内核 OSS 驱动提供了非阻塞的 I/O 支持，用户空间也不宜采用。

无论是从声卡读取数据，或是向声卡写入数据，事实上都具有特定的格式（format），如无符号 8 位、单声道、8kHz 采样率，如果默认值无法达到要求，可以通过 ioctl()系统调用来改变它们。通常说来，在应用程序中打开设备文件/dev/dsp 之后，接下去就应该为其设置恰当的格式，然后才能从声卡读取或者写入数据。

17.3.2 mixer 接口

```
int register_sound_mixer(struct file_operations *fops, int dev);
```

上述函数用于注册一个混音器，第一个参数 fops 即是文件操作接口，第二个参数 dev 是设备编号，如果填入-1，则系统自动分配一个设备编号。mixer 是一个典型的字符设备，因此编码的主要工作是实现 file_operations 中的 open()、ioctl()等函数。

mixer 接口 file_operations 中的最重要函数是 ioctl()，它实现混音器的不同 I/O 控制命令，代码清单 17.1 所示为一个 ioctl()的范例。

代码清单 17.1 mixer()接口的 ioctl()函数范例

```
1   static int mixdev_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
unsigned long arg)
2   {
3    ...
4    switch (cmd)
5    {
6      case SOUND_MIXER_READ_MIC:
7        ...
8      case SOUND_MIXER_WRITE_MIC:
9        ...
10     case SOUND_MIXER_WRITE_RECSRC:
11       ...
12     case SOUND_MIXER_WRITE_MUTE:
13       ...
14   }
15   //其他命令
```

```
return mixer_ioctl(codec, cmd, arg);
}
```

17.3.3 dsp 接口

```
int register_sound_dsp(struct file_operations *fops, int dev);
```

上述函数与 register_sound_mixer()类似，它用于注册一个 dsp 设备，第一个参数 fops 即是文件操作接口，第二个参数 dev 是设备编号，如果填入−1，则系统自动分配一个设备编号。dsp 也是一个典型的字符设备，因此编码的主要工作是实现 file_operations 中的 read()、write()、ioctl()等函数。

dsp 接口 file_operations 中的 read()和 write()函数非常重要，read()函数从音频控制器中获取录音数据到缓冲区并复制到用户空间，write()函数从用户空间复制音频数据到内核空间缓冲区并最终发送到音频控制器。

dsp 接口 file_operations 中的 ioctl()函数处理对采样率、量化精度、DMA 缓冲区块大小等参数设置 I/O 控制命令的处理。

在数据从缓冲区复制到音频控制器的过程中，通常会使用 DMA，DMA 对声卡而言非常重要。例如，在放音时，驱动设置完 DMA 控制器的源数据地址（内存中的 DMA 缓冲区）、目的地址（音频控制器 FIFO）和 DMA 的数据长度，DMA 控制器会自动发送缓冲区的数据填充 FIFO，直到发送完相应的数据长度后才中断一次。

在 OSS 驱动中，建立存放音频数据的环形缓冲区（ring buffer）通常是值得推荐的方法。此外，在 OSS 驱动中，一般会将一个较大的 DMA 缓冲区分成若干个大小相同的块（这些块也被称为“段”，即 fragment），驱动程序使用 DMA 每次在声音缓冲区和声卡之间搬移一个 fragment。在用户空间，可以使用 ioctl()系统调用来调整块的大小和个数。

除了 read()、write()和 ioctl()外，dsp 接口的 poll()函数通常也需要被实现，以向用户反馈目前能否读写 DMA 缓冲区。

在 OSS 驱动初始化过程中，会调用 register_sound_dsp()和 register_sound_mixer()注册 dsp 和 mixer 设备；在模块卸载的时候，会调用代码清单 17.2。

代码清单 17.2　OSS 驱动初始化注册 dsp 和 mixer 设备

```
static int xxx_init(void)
{
  struct xxx_state *s = &xxx_state;
  ...
  //注册dsp设备
  if ((audio_dev_dsp = register_sound_dsp(&xxx_audio_fops,  - 1)) < 0)
      goto err_dev1;
  //设备mixer设备
  if ((audio_dev_mixer = register_sound_mixer(&xxx_mixer_fops,  - 1)) < 0)
      goto err_dev2;
  ...
}

void __exit xxx_exit(void)
```

```
{
  //注销 dsp 和 mixer 设备接口
  unregister_sound_dsp(audio_dev_dsp);
  unregister_sound_mixer(audio_dev_mixer);
  ...
}
```

根据 17.3.2 和 17.3.3 小节的分析，可以画出一个 Linux OSS 驱动结构的简图，如图 17.4 所示。

图 17.4　Linux OSS 驱动结构

17.3.4　OSS 用户空间编程

1．dsp 编程

对 OSS 驱动声卡的编程使用 Linux 文件接口函数，如图 17.5 所示，dsp 接口的操作一般包括如下几个步骤。

图 17.5　OSS dsp 接
用户空间操作流程

（1）打开设备文件/dev/dsp。

采用何种模式对声卡进行操作也必须在打开设备时指定，对于不支持全双工的声卡来说，应该使用只读或者只写的方式打开，只有那些支持全双工的声卡，才能以读写的方式打开，这还依赖于驱动程序的具体实现。Linux 允许应用程序多次打开或者关闭与声卡对应的设备文件，从而能够很方便地在放音状态和录音状态之间进行切换。

（2）如果有需要，设置缓冲区大小。

运行在 Linux 内核中的声卡驱动程序专门维护了一个缓冲区，其大小会影响到播放和录音时的效果，使用 ioctl()系统调用可以对它的尺寸进行恰当设置。调节驱动程序中缓冲区大小的操作不是必须的，如果没有特殊的要求，一般采用默认的缓冲区大小也就可以了。如果想设置缓冲区的大小，则通常应紧跟在设备文件打开之后，这是因为对声卡的其他操作有可能会导致驱动程序无法再修改其缓冲区的大小。

（3）设置声道（channel）数量。

根据硬件设备和驱动程序的具体情况，可以设置为单声道或者立体声。

（4）设置采样格式和采样频率

采样格式包括 AFMT_U8（无符号 8 位）、AFMT_S8（有符号 8 位）、AFMT_U16_LE

华清远见

（小端模式，无符号 16 位）、AFMT_U16_BE（大端模式，无符号 16 位）、AFMT_MPEG、AFMT_AC3 等。使用 SNDCTL_DSP_SETFMT IO 控制命令可以设置采样格式。

对于大多数声卡来说，其支持的采样频率范围一般为 5kHz～44.1kHz 或者 48kHz，但并不意味着该范围内的所有连续频率都会被硬件支持，在 Linux 系统下进行音频编程时最常用到的几种采样频率是 11025Hz、16000Hz、22050Hz、32000Hz 和 44100Hz。使用 SNDCTL_DSP_SPEED IO 控制命令可以设置采样频率。

（5）读写/dev/dsp 实现播放或录音。

代码清单 17.3 的程序实现了利用/dev/dsp 接口进行声音录制和播放的过程，它的功能是先录制几秒钟音频数据，将其存放在内存缓冲区中，然后再进行播放。

代码清单 17.3　OSS dsp 接口应用编程范例

```
1  #include <unistd.h>
2  #include <fcntl.h>
3  #include <sys/types.h>
4  #include <sys/ioctl.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <linux/soundcard.h>
8  #define LENGTH 3        /* 存储秒数 */
9  #define RATE 8000       /* 采样频率 */
10 #define SIZE 8          /* 量化位数 */
11 #define CHANNELS 1      /* 声道数目 */
12 /* 用于保存数字音频数据的内存缓冲区 */
13 unsigned char buf[LENGTH *RATE * SIZE * CHANNELS / 8];
14 int main()
15 {
16   int fd; /* 声音设备的文件描述符 */
17   int arg; /* 用于ioctl调用的参数 */
18   int status; /* 系统调用的返回值 */
19   /* 打开声音设备 */
20   fd = open("/dev/dsp", O_RDWR);
21   if(fd < 0)
22   {
23      perror("open of /dev/dsp failed");
24      exit(1);
25   }
26   /* 设置采样时的量化位数 */
27   arg = SIZE;
28   status = ioctl(fd, SOUND_PCM_WRITE_BITS, &arg);
29   if (status ==  - 1)
30      perror("SOUND_PCM_WRITE_BITS ioctl failed");
31   if (arg != SIZE)
32     perror("unable to set sample size");
33   /* 设置采样时的通道数目 */
34   arg = CHANNELS;
```

```
35  status = ioctl(fd, SOUND_PCM_WRITE_CHANNELS, &arg);
36  if(status ==  - 1)
37     perror("SOUND_PCM_WRITE_CHANNELS ioctl failed");
38  if(arg != CHANNELS)
39     perror("unable to set number of channels");
40  /* 设置采样率 */
41  arg = RATE;
42  status = ioctl(fd, SOUND_PCM_WRITE_RATE, &arg);
43  if(status ==  - 1)
44     perror("SOUND_PCM_WRITE_WRITE ioctl failed");
45  /* 循环，直到按下[ControltC]*/
46  while (1)
47  {
48    printf("Say something:\n");
49    status = read(fd, buf, sizeof(buf)); /* 录音 */
50    if(status != sizeof(buf))
51       perror("read wrong number of bytes");
52    printf("You said:\n");
53    status = write(fd, buf, sizeof(buf)); /* 放音 */
54    if(status != sizeof(buf))
55       perror("wrote wrong number of bytes");
56    /* 在继续录音前等待放音结束 */
57    status = ioctl(fd, SOUND_PCM_SYNC, 0);
58    if(status ==  - 1)
59       perror("SOUND_PCM_SYNC ioctl failed");
60  }
61 }
```

2．mixer 编程

声卡上的混音器由多个混音通道组成，它们可以通过驱动程序提供的设备文件/dev/mixer 进行编程。对混音器的操作一般都通过 ioctl()系统调用来完成，所有控制命令都以 SOUND_MIXER 或者 MIXER 开头，表 17.1 列出了常用的混音器控制命令。

表 17.1　　混音器常用命令

命　　令	作　　用
SOUND_MIXER_VOLUME	主音量调节
SOUND_MIXER_BASS	低音控制
SOUND_MIXER_TREBLE	高音控制
SOUND_MIXER_SYNTH	FM 合成器
SOUND_MIXER_PCM	主 D/A 转换器
SOUND_MIXER_SPEAKER	PC 喇叭

SOUND_MIXER_LINE	音频线输入
SOUND_MIXER_MIC	麦克风输入
SOUND_MIXER_CD	CD 输入
SOUND_MIXER_IMIX	放音音量
SOUND_MIXER_ALTPCM	从 D/A 转换器
SOUND_MIXER_RECLEV	录音音量

续表

命　　令	作　　用
SOUND_MIXER_IGAIN	输入增益
SOUND_MIXER_OGAIN	输出增益
SOUND_MIXER_LINE1	声卡的第 1 输入
SOUND_MIXER_LINE2	声卡的第 2 输入
SOUND_MIXER_LINE3	声卡的第 3 输入

对声卡的输入增益和输出增益进行调节是混音器的一个主要作用，目前大部分声卡采用的是 8 位或者 16 位的增益控制器，声卡驱动程序会将它们变换成百分比的形式，也就是说无论是输入增益还是输出增益，其取值范围都是从 0～100。

（1）SOUND_MIXER_READ 宏。

在进行混音器编程时，可以使用 SOUND_MIXER_READ 宏来读取混音通道的增益大小，例如，如下代码可以获得麦克风的输入增益：

```
ioctl(fd, SOUND_MIXER_READ(SOUND_MIXER_MIC), &vol);
```

对于只有一个混音通道的单声道设备来说，返回的增益大小保存在低位字节中。而对于支持多个混音通道的双声道设备来说，返回的增益大小实际上包括两个部分，分别代表左、右两个声道的值，其中低位字节保存左声道的音量，而高位字节则保存右声道的音量。下面的代码可以从返回值中依次提取左右声道的增益大小：

```
int left, right;
left = vol & 0xff;
right = (vol & 0xff00) >> 8;
```

（2）SOUND_MIXER_WRITE 宏。

如果想设置混音通道的增益大小，则可以通过 SOUND_MIXER_WRITE 宏来实现，例如下面的语句可以用来设置麦克风的输入增益：

```
vol = (right << 8) + left;
ioctl(fd, SOUND_MIXER_WRITE(SOUND_MIXER_MIC), &vol);
```

（3）查询 MIXER 信息。

声卡驱动程序提供了多个 ioctl()系统调用来获得混音器的信息，它们通常返回一个整型的位掩码，其中每一位分别代表一个特定的混音通道，如果相应的位为 1，则说明与之对应的混音通道是可用的。

通过 SOUND_MIXER_READ_DEVMASK 返回的位掩码查询出能够被声卡支持的每一个混音通道，而通过 SOUND_MIXER_READ_RECMAS 返回的位掩码则可以查询出能够被当作录音源的每一个通道。例如，如下代码可用来检查 CD 输入是否是一

个有效的混音通道：

```
ioctl(fd, SOUND_MIXER_READ_DEVMASK, &devmask);
if (devmask & SOUND_MIXER_CD)
   printf("The CD input is supported");
```

如下代码可用来检查 CD 输入是否是一个有效的录音源：

```
ioctl(fd, SOUND_MIXER_READ_RECMASK, &recmask);
if (recmask & SOUND_MIXER_CD)
   printf("The CD input can be a recording source");
```

大多数声卡提供了多个录音源，通过 SOUND_MIXER_READ_RECSRC 可以查询出当前正在使用的录音源，同一时刻可使用两个或两个以上的录音源，具体由声卡硬件本身决定。相应地，使用 SOUND_MIXER_WRITE_RECSRC 可以设置声卡当前使用的录音源，如下代码可以将 CD 输入作为声卡的录音源使用。

```
devmask = SOUND_MIXER_CD;
ioctl(fd, SOUND_MIXER_WRITE_RECSRC, &devmask);
```

此外，所有的混音通道都有单声道和双声道的区别，如果需要知道哪些混音通道提供了对立体声的支持，可以通过 SOUND_MIXER_READ_STEREODEVS 来获得。

代码清单 17.4 的程序实现了利用/dev/mixer 接口对混音器进行编程的过程，该程序可对各种混音通道的增益进行调节。

代码清单 17.4　OSS mixer 接口应用编程范例

```
1   #include <unistd.h>
2   #include <stdlib.h>
3   #include <stdio.h>
4   #include <sys/ioctl.h>
5   #include <fcntl.h>
6   #include <linux/soundcard.h>
7   /* 用来存储所有可用混音设备的名称 */
8   const char *sound_device_names[] = SOUND_DEVICE_NAMES;
9   int fd; /* 混音设备所对应的文件描述符 */
10  int devmask, stereodevs; /* 混音器信息对应的 bit 掩码 */
11  char *name;
12  /* 显示命令的使用方法及所有可用的混音设备 */
13  void usage()
14  {
15    int i;
16          fprintf(stderr,   "usage:   %s   <device>   <left-gain%%>
<right-gain%%>\n"
17     "%s <device> <gain%%>\n\n""Where <device> is one of:\n", name,
name);
18    for(i = 0; i < SOUND_MIXER_NRDEVICES; i++)
19      if ((1 << i) &devmask)
20     /* 只显示有效的混音设备 */
21       fprintf(stderr, "%s ", sound_device_names[i]);
22    fprintf(stderr, "\n");
23    exit(1);
24  }
25
```

```
26 int main(int argc, char *argv[])
27 {
28   int left, right, level; /* 增益设置 */
29   int status; /* 系统调用的返回值 */
30   int device; /* 选用的混音设备 */
31   char *dev; /* 混音设备的名称 */
32   int i;
33   name = argv[0];
34   /* 以只读方式打开混音设备 */
35   fd = open("/dev/mixer", O_RDONLY);
36   if (fd ==  - 1)
37   {
38     perror("unable to open /dev/mixer");
39     exit(1);
40   }
41
42   /* 获得所需要的信息 */
43   status = ioctl(fd, SOUND_MIXER_READ_DEVMASK, &devmask);
44   if(status ==  - 1)
45      perror("SOUND_MIXER_READ_DEVMASK ioctl failed");
46   status = ioctl(fd, SOUND_MIXER_READ_STEREODEVS, &stereodevs);
47   if (status ==  - 1)
48       perror("SOUND_MIXER_READ_STEREODEVS ioctl failed");
49   /* 检查用户输入 */
50   if (argc != 3 && argc != 4)
51       usage();
52   /* 保存用户输入的混音器名称 */
53   dev = argv[1];
54   /* 确定即将用到的混音设备 */
55   for (i = 0; i < SOUND_MIXER_NRDEVICES; i++)
56                   if  (((1  <<  i)  &devmask)  &&  !strcmp(dev,
sound_device_names[i]))
57           break;
58   if (i == SOUND_MIXER_NRDEVICES)
59   {
60       /* 没有找到匹配项 */
61     fprintf(stderr, "%s is not a valid mixer device\n", dev);
62     usage();
63   }
64   /* 查找到有效的混音设备 */
65   device = i;
66   /* 获取增益值 */
67   if (argc == 4)
68   {
69     /* 左、右声道均给定 */
70     left = atoi(argv[2]);
71     right = atoi(argv[3]);
72   }
73   else
74   {
75     /* 左、右声道设为相等 */
76     left = atoi(argv[2]);
77     right = atoi(argv[2]);
78   }
79
```

```
    80    /* 对非立体声设备给出警告信息 */
    81    if ((left != right) && !((1 << i) &stereodevs))
    82    {
    83      fprintf(stderr, "warning: %s is not a stereo device\n", dev);
    84    }
    85
    86    /* 将两个声道的值合到同一变量中 */
    87    level = (right << 8) + left;
    88
    89    /* 设置增益 */
    90    status = ioctl(fd, MIXER_WRITE(device), &level);
    91    if (status ==  - 1)
    92      {
    93        perror("MIXER_WRITE ioctl failed");
    94        exit(1);
    95    }
    96    /* 获得从驱动返回的左右声道的增益 */
    97       left = level &0xff;
    98       right = (level &0xff00) >> 8;
    99       /* 显示实际设置的增益 */
    100       fprintf(stderr, "%s gain set to %d%% / %d%%\n", dev, left,
right);
    101       /* 关闭混音设备 */
    102       close(fd);
    103       return 0;
    104 }
```

编译上述程序为可执行文件 mixer，执行./mixer <device> <left-gain%> <right-gain%>或./mixer <device> <gain%>可设置增益，device 可以是 vol、pcm、speaker、line、mic、cd、igain、line1、phin、video。

17.4 Linux ALSA 音频设备驱动

17.4.1 ALSA 的组成

虽然 OSS 已经非常成熟，但它毕竟是一个没有完全开放源代码的商业产品，而 ALSA （Advanced Linux Sound Architecture）恰好弥补了这一空白，它符合 GPL，是在 Linux 下进行音频编程时另一种可供选择的声卡驱动体系结构。ALSA 除了像 OSS 那样提供了一组内核驱动程序模块之外，还专门为简化应用程序的编写提供了相应的函数库，与 OSS 提供的基于 ioctl 的原始编程接口相比，ALSA

函数库使用起来要更加方便一些。ALSA 的主要特点如下。

- 支持多种声卡设备。
- 模块化的内核驱动程序。
- 支持 SMP 和多线程。
- 提供应用开发函数库（alsa-lib）以简化应用程序开发。
- 支持 OSS API，兼容 OSS 应用程序。

ALSA 具有更加友好的编程接口，并且完全兼容于 OSS，对应用程序员来讲无疑是一个更佳的选择。ALSA 系统包括驱动包 alsa-driver、开发包 alsa-libs、开发包插件 alsa-libplugins、设置管理工具包 alsa-utils、其他声音相关处理小程序包 alsa-tools、特殊音频固件支持包 alsa- firmware、OSS 接口兼容模拟层工具 alsa-oss 共 7 个子项目，其中只有驱动包是必需的。

alsa-driver 指内核驱动程序，包括硬件相关的代码和一些公共代码，非常庞大，代码总量达数十万行；alsa-libs 指用户空间的函数库，提供给应用程序使用，应用程序应包含头文件 asoundlib.h，并使用共享库 libasound.so；alsa-utils 包含一些基于 ALSA 的用于控制声卡的应用程序，如 alsaconf（侦测系统中声卡并写一个适合的 ALSA 配置文件）、alsactl（控制 ALSA 声卡驱动的高级设置）、alsamixer（基于 ncurses 的混音器程序）、amidi（用于读写 ALSA RawMIDI）、amixer（ALSA 声卡混音器的命令行控制）、aplay（基于命令行的声音文件播放）、arecord（基于命令行的声音文件录制）等。

目前 ALSA 内核提供给用户空间的接口有：

- 信息接口（Information Interface，/proc/asound）；
- 控制接口（Control Interface，/dev/snd/controlCX）；
- 混音器接口（Mixer Interface，/dev/snd/mixerCXDX）；
- PCM 接口（PCM Interface，/dev/snd/pcmCXDX）；
- Raw 迷笛接口（Raw MIDI Interface，/dev/snd/midiCXDX）；
- 音序器接口（Sequencer Interface，/dev/snd/seq）；
- 定时器接口（Timer Interface，/dev/snd/timer）。

和 OSS 类似，上述接口也以文件的方式被提供，不同的是这些接口被提供给 alsa-lib 使用，而不是直接给应用程序使用的。应用程序最好使用 alsa-lib，或者更高级的接口，比如 jack 提供的接口。

图 17.6 所示为 ALSA 声卡驱动与用户空间体系结构的简图，从中可以看出 ALSA 内核驱动与用户空间库及 OSS 之间的关系。

华清远见

图 17.6　ALSA 体系结构

17.4.2　card 和组件管理

对于每个声卡而言，必须创建一个 card 实例。card 是声卡的“总部”，它管理这个声卡上的所有设备（组件），如 PCM、mixers、MIDI、synthesizer 等。因此，card 和组件是 ALSA 声卡驱动中的主要组成元素。

1．创建 card

```
struct snd_card *snd_card_new(int idx, const char *xid,
                struct module *module, int extra_size);
```

idx 是 card 索引号，xid 是标识字符串，module 一般为 THIS_MODULE，extra_size 是要分配的额外数据的大小，分配的 extra_size 大小的内存将作为 card->private_data。

2．创建组件

```
int snd_device_new(struct snd_card *card, snd_device_type_t type,
            void *device_data, struct snd_device_ops *ops);
```

当 card 被创建后，设备（组件）能够被创建并关联于该 card。第 1 个参数是 snd_card_new()创建的 card 指针，第 2 个参数 type 指的是 device-level 即设备类型，形式为 SNDRV_DEV_XXX，包括 SNDRV_DEV_CODEC、SNDRV_DEV_CONTROL、SNDRV_DEV_PCM、SNDRV_DEV_ RAWMIDI 等，用户自定义设备的 device-level 是 SNDRV_DEV_LOWLEVEL，ops 参数是 1 个函数集（定义为 snd_device_ops 结构体）的指针，device_data 是设备数据指针，注意函数 snd_device_ new()本身不会分配设备数据的内存，因此应事先分配。

3．组件释放

每个 ALSA 预定义的组件在构造时需调用 snd_device_new()，而每个组件的析构方法则在函数集中被包含。对于 PCM、AC97 此类预定义组件，我们不需关心它们的析构，而对于自定义的组件，则需要填充 snd_device_ops 中的析构函数指针 dev_free，这样，当 snd_card_free()被调用时，组件将自动被释放。

4．芯片特定的数据（Chip-Specific Data）

芯片特定的数据一般以 struct xxxchip 结构体形式组织，这个结构体中包含芯片相关的 I/O 端口地址、资源指针、中断号等，其意义等同于字符设备驱动中的 file->private_data。

定义芯片特定的数据主要有两种方法，一种方法是将 sizeof(struct xxxchip)传入 snd_card_new()作为 extra_size 参数，它将自动成为 snd_card 的 private_data 成员，如代码清单 17.5 所示；另一种方法是在 snd_card_new()传入给 extra_size 参数 0，再分配 sizeof(struct xxxchip)的内存，将分配内存的地址传入 snd_device_new()的 device_data 的参数，如代码清单 17.6 所示。

代码清单 17.5　创建芯片特定的数据方法 1

```
struct xxxchip //芯片特定的数据结构体
{
   ...
};
card = snd_card_new(index, id, THIS_MODULE, sizeof(struct
xxxchip)); //创建声卡并申请 xxx_chi 内存作为 card-> private_data
struct xxxchip *chip = card->private_data;
```

代码清单 17.6　创建芯片特定的数据方法 2

```
struct snd_card *card;
struct xxxchip *chip;
//使用 0 作为第 4 个参数，并动态分配 xxx_chip 的内存
card = snd_card_new(index[dev], id[dev], THIS_MODULE, 0);
...
chip = kzalloc(sizeof(*chip), GFP_KERNEL);
//在 xxxchip 结构体中，应该包括声卡指针
struct xxxchip
{
   struct snd_card *card;
   ...
};
//并将其 card 成员赋值为 snd_card_new()创建的 card 指针
chip->card = card;
static struct snd_device_ops ops =
{
  . dev_free = snd_xxx_chip_dev_free,  //组件析构
};
...
//创建自定义组件
snd_device_new(card, SNDRV_DEV_LOWLEVEL, chip, &ops);
//在析构函数中释放 xxxchip 内存
static int snd_xxx_chip_dev_free(struct snd_device *device)
```

```
{
  return snd_xxx_chip_free(device->device_data); //释放
}
```

5．注册/释放声卡

当 snd_card 被准备好以后，可使用 snd_card_register()函数注册这个声卡，如下所示：

```
int snd_card_register(struct snd_card *card)
```

对应的 snd_card_free()完成相反的功能，如下所示：

```
int snd_card_free(struct snd_card *card);
```

17.4.3 PCM 设备

每个声卡最多可以有四个 PCM 实例，一个 PCM 实例对应一个设备文件。PCM 实例由 PCM 播放和录音流组成，而每个 PCM 流又由一个或多个 PCM 子流组成。有的声卡支持多重播放功能，例如，emu10k1 包含一个有 32 个立体声子流的 PCM 播放设备。

1．PCM 实例构造

```
int snd_pcm_new(struct snd_card *card, char *id, int device,
              int playback_count, int capture_count, struct snd_pcm **
rpcm);
```

第 1 个参数是 card 指针，第 2 个是标识字符串，第 3 个是 PCM 设备索引（0 表示第 1 个 PCM 设备），第 4 和第 5 个分别为播放和录音设备的子流数。当存在多个子流时，需要恰当地处理 open()、close()和其他函数。在每个回调函数中，可以通过 snd_pcm_substream 的 number 成员得知目前操作的究竟是哪个子流，如下所示：

```
struct snd_pcm_substream *substream;
int index = substream->number;
```

一种习惯的做法是在驱动中定义一个 PCM“构造函数”，负责 PCM 实例的创建，如代码清单 17.7 所示。

代码清单 17.7　PCM 设备的“构造函数”

```
static int __devinit snd_xxxchip_new_pcm(struct xxxchip *chip)
{
  struct snd_pcm *pcm;
  int err;
  //创建 PCM 实例
  if ((err = snd_pcm_new(chip->card, "xxx Chip", 0, 1, 1,  &pcm)) <
0)
    return err;
  pcm->private_data = chip; //置 pcm->private_data 为芯片特定数据
  strcpy(pcm->name, "xxx Chip");
  chip->pcm = pcm;
  ...
```

```
return 0;
}
```

2．设置 PCM 操作

```
void snd_pcm_set_ops(struct snd_pcm *pcm, int direction, struct snd_pcm_ops *ops);
```

第 1 个参数是 snd_pcm 的指针，第 2 个参数是 SNDRV_PCM_STREAM_PLAYBACK 或 SNDRV_ PCM_STREAM_CAPTURE，而第 3 个参数是 PCM 操作结构体 snd_pcm_ops，这个结构体的定义如代码清单 17.8 所示。

代码清单 17.8　snd_pcm_ops 结构体

```
struct snd_pcm_ops
{
int (*open)(struct snd_pcm_substream *substream);//打开
int (*close)(struct snd_pcm_substream *substream);//关闭
int (*ioctl)(struct snd_pcm_substream * substream,
            unsigned int cmd, void *arg);//I/O 控制
int (*hw_params)(struct snd_pcm_substream *substream,
         struct snd_pcm_hw_params *params);//硬件参数
int (*hw_free)(struct snd_pcm_substream *substream); //资源释放
int (*prepare)(struct snd_pcm_substream *substream);//准备
//在 PCM 被开始、停止或暂停时调用
int (*trigger)(struct snd_pcm_substream *substream, int cmd);
snd_pcm_uframes_t (*pointer)(struct snd_pcm_substream *substream);//当前缓冲区的硬件位置
//缓冲区复制
int (*copy)(struct snd_pcm_substream *substream, int channel,
           snd_pcm_uframes_t pos,
           void __user *buf, snd_pcm_uframes_t count);
int (*silence)(struct snd_pcm_substream *substream, int channel,
              snd_pcm_uframes_t pos, snd_pcm_uframes_t count);
struct page *(*page)(struct snd_pcm_substream *substream,
             unsigned long offset);
int (*mmap)(struct snd_pcm_substream *substream, struct vm_area_struct *vma);
int (*ack)(struct snd_pcm_substream *substream);
};
```

snd_pcm_ops 中的所有操作都需事先通过 snd_pcm_substream_chip()获得 xxxchip 指针，例如：

```
int xxx()
{
```

```
    struct xxxchip *chip = snd_pcm_substream_chip(substream);
   ...
}
```

当一个 PCM 子流被打开时，snd_pcm_ops 中的 open()函数将被调用，在这个函数中，至少需要初始化 runtime->hw 字段，代码清单 17.9 所示为 open()函数的范例。

代码清单 17.9　snd_pcm_ops 结构体中的 open()函数

```
1  static int snd_xxx_open(struct snd_pcm_substream *substream)
2  {
3    //从子流获得 xxxchip 指针
4    struct xxxchip *chip = snd_pcm_substream_chip(substream);
5    //获得 PCM 运行时信息指针
6    struct snd_pcm_runtime *runtime = substream->runtime;
7    ...
8    //初始化 runtime->hw
9    runtime->hw = snd_xxxchip_playback_hw;
10   return 0;
11 }
```

上述代码中的 snd_xxxchip_playback_hw 是预先定义的硬件描述。在 open()函数中，可以分配一段私有数据。如果硬件配置需要更多的限制，也需设置硬件限制。

当 PCM 子流被关闭时，close()函数将被调用。如果 open()函数中分配了私有数据，则在 close()函数中应该释放 substream 的私有数据，代码清单 17.10 所示为 close()函数的范例。

代码清单 17.10　snd_pcm_ops 结构体中的 close()函数

```
1 static int snd_xxx_close(struct snd_pcm_substream *substream)
2 {
3   //释放子流私有数据
4   kfree(substream->runtime->private_data);
5   //...
6 }
```

驱动中通常可以给 snd_pcm_ops 的 ioctl()成员函数传递通用的 snd_pcm_lib_ioctl()函数。

snd_pcm_ops 的 hw_params()成员函数将在应用程序设置硬件参数（PCM 子流的周期大小、缓冲区大小和格式等）的时候被调用，它的形式如下：

```
static int snd_xxx_hw_params(struct snd_pcm_substream *substream,struct snd_pcm_hw_params *hw_params);
```

在这个函数中，将完成大量硬件设置，甚至包括缓冲区分配，这时可调用如下辅

助函数：

```
snd_pcm_lib_malloc_pages(substream, params_buffer_bytes(hw_params));
```

仅当 DMA 缓冲区已被预先分配的情况下，上述调用才可成立。

与 hw_params()对应的函数是 hw_free()，它释放由 hw_params()分配的资源，例如，通过如下调用释放 snd_pcm_lib_malloc_pages()缓冲区：

```
snd_pcm_lib_free_pages(substream);
```

当 PCM 被“准备”时，prepare()函数将被调用，在其中可以设置采样率、格式等。prepare() 函数与 hw_params() 函数的不同在于对 prepare() 的调用发生在 snd_pcm_prepare()每次被调用的时候。prepare()的形式如下：

```
static int snd_xxx_prepare(struct snd_pcm_substream *substream);
```

trigger()成员函数在 PCM 被开始、停止或暂停时调用，函数的形式如下：

```
static int snd_xxx_trigger(struct snd_pcm_substream *substream, int
cmd);
```

cmd 参数定义了具体的行为，在 trigger() 成员函数中至少要处理 SNDRV_PCM_TRIGGER_ START 和 SNDRV_PCM_TRIGGER_STOP 命令，如果 PCM 支持暂停，还应处理 SNDRV_PCM_ TRIGGER_PAUSE_PUSH 和 SNDRV_PCM_TRIGGER_PAUSE_RELEASE 命令。如果设备支持挂起/恢复，当能量管理状态发生变化时将处理 SNDRV_PCM_TRIGGER_SUSPEND 和 SNDRV_PCM_ TRIGGER_RESUME 这两个命令。注意 trigger()函数是原子的，中途不能睡眠。代码清单 17.11 所示为 trigger()函数的范例。

代码清单 17.11　snd_pcm_ops 结构体中的 trigger()函数

```
1  static int snd_xxx_trigger(struct snd_pcm_substream *substream, int
cmd)
2  {
3    switch (cmd)
4    {
5       case SNDRV_PCM_TRIGGER_START:
6       // 开启 PCM 引擎
7       break;
8     case SNDRV_PCM_TRIGGER_STOP:
9       // 停止 PCM 引擎
10      break;
11    ...//其他命令
12    default:
13      return  - EINVAL;
14   }
15 }
```

pointer()函数用于 PCM 中间层查询目前缓冲区的硬件位置，该函数以帧的形式返回 0～buffer_size – 1 的位置（ALSA 0.5.x 中为字节形式），此函数也是原子的。

copy()和 silence()函数一般可以省略，但是，当硬件缓冲区不处于常规内存中时需

要。例如，一些设备有自己的不能被映射的硬件缓冲区，这种情况下，我们不得不将数据从内存缓冲区复制到硬件缓冲区。当内存缓冲区在物理和虚拟地址上都不连续时，这两个函数也必须被实现。

3．分配缓冲区

分配缓冲区的最简单方法是调用如下函数：

```
int snd_pcm_lib_preallocate_pages_for_all(struct snd_pcm *pcm,
                                          int type, void *data, size_t size, size_t max);
```

type 参数是缓冲区的类型，包含 SNDRV_DMA_TYPE_UNKNOWN（未知）、SNDRV_DMA_ TYPE_CONTINUOUS（连续的非 DMA 内存）、SNDRV_DMA_TYPE_DEV（连续的通用设备），SNDRV_DMA_TYPE_DEV_SG（通用设备 SG-buffer）和 SNDRV_DMA_TYPE_SBUS（连续的 SBUS）。如下代码将分配 64KB 的缓冲区：

```
snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
snd_dma_pci_data(chip->pci),64*1024, 64*1024);
```

4．设置标志

在构造 PCM 实例、设置操作集并分配缓冲区之后，如果有需要，应设置 PCM 的信息标志，例如，如果 PCM 设备只支持半双工，则这样定义标志：

```
pcm->info_flags = SNDRV_PCM_INFO_HALF_DUPLEX;
```

5．PCM 实例析构

PCM 实例的“析构函数”并非是必须的，因为 PCM 实例会被 PCM 中间层代码自动释放，如果驱动中分配了一些特别的内存空间，则必须定义“析构函数”，代码清单 17.12 所示为 PCM“析构函数”与对应的“构造函数”，“析构函数”会释放“构造函数”中创建的 xxx_private_pcm_data。

代码清单 17.12　PCM 设备“析构函数”

```
static void xxxchip_pcm_free(struct snd_pcm *pcm)
{
  /* 从 pcm 实例得到 chip */
  struct xxxchip *chip = snd_pcm_chip(pcm);
  /* 释放自定义用途的内存 */
  kfree(chip->xxx_private_pcm_data);
  ...
}

static int __devinit snd_xxxchip_new_pcm(struct xxxchip
```

```
*chip)
   11 {
   12   struct snd_pcm *pcm;
   13    ...
   14    /* 分配自定义用途的内存 */
   15    chip->xxx_private_pcm_data = kmalloc(...);
   16    pcm->private_data = chip;
   17    /* 设置“析构函数” */
   18    pcm->private_free = xxxchip_pcm_free;
   19    ...
   20 }
```

上述代码第 4 行的 snd_pcm_chip()从 PCM 实例指针获得 xxxchip 指针，实际上它就是返回第 16 行给 PCM 实例赋予的 xxxchip 指针。

6. PCM 信息运行时结构体

当 PCM 子流被打开后，PCM 运行时实例（定义为结构体 snd_pcm_runtime，如代码清单 17.13 所示）将被分配给这个子流，这个指针通过 substream->runtime 获得。运行时指针包含各种各样的信息：hw_params 及 sw_params 配置的拷贝、缓冲区指针、mmap 记录、自旋锁等，几乎 PCM 的所有控制信息均能从中取得。

代码清单 17.13 snd_pcm_runtime 结构体

```
struct snd_pcm_runtime
{
  /* 状态 */
  struct snd_pcm_substream *trigger_master;
  snd_timestamp_t trigger_tstamp; /* 触发时间戳 */
  int overrange;
  snd_pcm_uframes_t avail_max;
  snd_pcm_uframes_t hw_ptr_base; /* 缓冲区复位时的位置 */
  snd_pcm_uframes_t hw_ptr_interrupt; /* 中断时的位置*/
  /* 硬件参数 */
  snd_pcm_access_t access; /* 存取模式 */
  snd_pcm_format_t format; /* SNDRV_PCM_FORMAT_* */
  snd_pcm_subformat_t subformat; /* 子格式 */
  unsigned int rate; /* rate in Hz */
  unsigned int channels; /* 通道 */
  snd_pcm_uframes_t period_size; /* 周期大小 */
  unsigned int periods; /* 周期数 */
  snd_pcm_uframes_t buffer_size; /* 缓冲区大小 */
  unsigned int tick_time; /* tick time */
  snd_pcm_uframes_t min_align; /* 格式对应的最小对齐*/
  size_t byte_align;
  unsigned int frame_bits;
  unsigned int sample_bits;
  unsigned int info;
  unsigned int rate_num;
  unsigned int rate_den;
  /* 软件参数 */
  struct timespec tstamp_mode; /* mmap 时间戳被更新*/
```

```
    29  unsigned int period_step;
    30  unsigned int sleep_min; /* 睡眠的最小节拍 */
    31  snd_pcm_uframes_t xfer_align;
    32  snd_pcm_uframes_t start_threshold;
    33  snd_pcm_uframes_t stop_threshold;
    34  snd_pcm_uframes_t silence_threshold; /* Silence 填充阈值 */
    35  snd_pcm_uframes_t silence_size; /* Silence 填充大小 */
    36  snd_pcm_uframes_t boundary;
    37  snd_pcm_uframes_t silenced_start;
    38  snd_pcm_uframes_t silenced_size;
    39  snd_pcm_sync_id_t sync; /* 硬件同步 ID */
    40  /* mmap  */
    41  volatile struct snd_pcm_mmap_status *status;
    42  volatile struct snd_pcm_mmap_control *control;
    43  atomic_t mmap_count;
    44  /* 锁/调度 */
    45  spinlock_t lock;
    46  wait_queue_head_t sleep;
    47  struct timer_list tick_timer;
    48  struct fasync_struct *fasync;
    49  /* 私有段 */
    50  void *private_data;
    51  void(*private_free)(struct snd_pcm_runtime *runtime);
    52  /* 硬件描述 */
    53  struct snd_pcm_hardware hw;
    54  struct snd_pcm_hw_constraints hw_constraints;
    55  /* 中断回调函数 */
    56  void(*transfer_ack_begin)(struct snd_pcm_substream*substream);
    57  void(*transfer_ack_end)(struct snd_pcm_substream *substream);
    58  /* 定时器 */
    59  unsigned int timer_resolution; /* timer resolution */
    60  /* DMA */
    61  unsigned char *dma_area; /* DMA 区域*/
    62  dma_addr_t dma_addr; /* 总线物理地址*/
    64  size_t dma_bytes; /* DMA 区域大小 */
    65  struct snd_dma_buffer *dma_buffer_p; /* 被分配的缓冲区 */
    66                  #if       defined(CONFIG_SND_PCM_OSS)       ||
defined(CONFIG_SND_PCM_OSS_MODULE)
    67    /* OSS 信息 */
    68    struct snd_pcm_oss_runtime oss;
    69  #endif
    70 };
```

snd_pcm_runtime 中的大多数记录对被声卡驱动操作集中的函数是只读的，仅仅 PCM 中间层可从更新或修改这些信息，但是硬件描述、中断回调函数、DMA 缓冲区信息和私有数据是例外的。

下面解释 snd_pcm_runtime 结构体中的几个重要成员。

（1）硬件描述。

硬件描述（snd_pcm_hardware 结构体）包含了基本硬件配置的定义，需要在 open()函数中赋值。runtime 实例保存的是硬件描述的拷贝而非指针，这意味着在 open()函数中可以修改被拷贝的描述（runtime->hw），例如：

```
struct snd_pcm_runtime *runtime = substream->runtime;
...
runtime->hw = snd_xxchip_playback_hw; /* “大众”硬件描述 */
/* 特定的硬件描述 */
if (chip->model == VERY_OLD_ONE)
runtime->hw.channels_max = 1;
```

snd_pcm_hardware 结构体的定义如代码清单 17.14 所示。

代码清单 17.14　snd_pcm_hardware 结构体

```
struct snd_pcm_hardware
{
unsigned int info;          /* SNDRV_PCM_INFO_* /
u64 formats;                /* SNDRV_PCM_FMTBIT_* */
unsigned int rates;         /* SNDRV_PCM_RATE_* */
unsigned int rate_min;         /* 最小采样率 */
unsigned int rate_max;         /* 最大采样率 */
unsigned int channels_min;     /* 最小的通道数 */
unsigned int channels_max;     /* 最大的通道数 */
 size_t buffer_bytes_max;      /* 最大缓冲区大小 */
 size_t period_bytes_min;      /* 最小周期大小 */
 size_t period_bytes_max;      /* 最大奏曲大小 */
 unsigned int periods_min;     /* 最小周期数 */
 unsigned int periods_max;     /* 最大周期数 */
 size_t fifo_size;             /* FIFO 字节数 */
};
```

snd_pcm_hardware 结构体中的 info 字段标识 PCM 设备的类型和能力，形式为 SNDRV_PCM_ INFO_XXX。info 字段至少需要定义是否支持 mmap，当支持时，应设置 SNDRV_PCM_INFO_MMAP 标志；当硬件支持 interleaved 或 non-interleaved 格式时，应设置 SNDRV_PCM_INFO _INTERLEAVED 或 SNDRV_PCM_INFO_NONINTERLEAVED 标志；如果都支持，则两者都可设置。

MMAP_VALID 和 BLOCK_TRANSFER 标志针对 OSS mmap，只有 mmap 被真正支持时，才可设置 MMAP_VALID；SNDRV_PCM_INFO_PAUSE 意味着设备可支持暂停操作，而 SNDRV_ PCM_INFO_RESUME 意味着设备可支持挂起/恢复操作；当 PCM 子流能被同步，如同步播放和录音流的 start/stop，可设置 SNDRV_PCM_INFO_SYNC_START 标志。

formats 包含 PCM 设备支持的格式，形式为 SNDRV_PCM_FMTBIT_XXX，如果设备支持多种模式，应将各种模式标志进行“或”操作。

rates 包含了 PCM 设备支持的采样率，形式如 SNDRV_PCM_RATE_XXX，如果支持连续的采样率，则传递 CONTINUOUS。

rate_min 和 rate_max 分别定义了最大和最小的采样率，注意：要与 rates 字段相符。

channel_min 和 channel_max 定义了最大和最小的通道数量。

buffer_bytes_max 定义最大的缓冲区大小，注意：没有 buffer_bytes_min 字段，这

是因为它可以通过最小的周期大小和最小的周期数量计算出来。

period 信息与 OSS 中的 fragment 对应，定义了 PCM 中断产生的周期。更小的周期大小意味着更多的中断，在录音时，周期大小定义了输入延迟，在播放时，整个缓冲区大小对应着输出延迟。

PCM 可被应用程序通过 alsa-lib 发送 hw_params 来配置，配置信息将保存在运行时实例中。对缓冲区和周期大小的配置以帧形式存储，而 frames_to_bytes()和 bytes_to_frames()可完成帧和字节的转换，如：

```
period_bytes = frames_to_bytes(runtime, runtime->period_size);
```

（2）DMA 缓冲区信息。

包含 dma_area（逻辑地址）、dma_addr（物理地址）、dma_bytes（缓冲区大小）和 dma_private（被 ALSA DMA 分配器使用）。可以由 snd_pcm_lib_malloc_pages()实现，ALSA 中间层会设置 DMA 缓冲区信息的相关字段，这种情况下，驱动中不能再写这些信息，只能读取。也就是说，如果使用标准的缓冲区分配函数 snd_pcm_lib_malloc_pages()分配缓冲区，则我们不需要自己维护 DMA 缓冲区信息。如果缓冲区由自己分配，则需要在 hw_params()函数中管理缓冲区信息，至少需管理 dma_bytes 和 dma_addr，如果支持 mmap，则必须管理 dma_area，对 dma_private 的管理视情况而定。

（3）运行状态。

通过 runtime->status 可以获得运行状态，它是 snd_pcm_mmap_status 结构体的指针，例如，通过 runtime->status->hw_ptr 可以获得目前的 DMA 硬件指针。此外，通过 runtime->control 可以获得 DMA 应用指针，它指向 snd_pcm_mmap_control 结构体指针。

（4）私有数据。

驱动中可以为子流分配一段内存并赋值给 runtime->private_data，注意不要与 pcm->private_ data 混淆，后者一般指向 xxxchip，而前者是在 PCM 设备的 open()函数中分配的动态数据，例如：

```
static int snd_xxx_open(struct snd_pcm_substream *substream)
{
  struct xxx_pcm_data *data;
  ....
  data = kmalloc(sizeof(*data), GFP_KERNEL);
  substream->runtime->private_data = data; //赋值 runtime->private_data
  ....
}
```

（5）中断回调函数：

transfer_ack_begin()和 transfer_ack_end()函数分别在 snd_pcm_period_elapsed()的开始和结束时被调用。

根据以上分析，代码清单 17.15 给出了一个完整的 PCM 设备接口模板。

华清远见

代码清单 17.15　PCM 设备接口模板

```
1  #include <sound/pcm.h>
2  ....
3  /* 播放设备硬件定义 */
4  static struct snd_pcm_hardware snd_xxxchip_playback_hw =
5  {
6    .info = (SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_INTERLEAVED |
7      SNDRV_PCM_INFO_BLOCK_TRANSFER | SNDRV_PCM_INFO_MMAP_VALID),
8    .formats = SNDRV_PCM_FMTBIT_S16_LE,
9    .rates = SNDRV_PCM_RATE_8000_48000,
10   .rate_min = 8000,
11   .rate_max = 48000,
12   .channels_min = 2,
13   .channels_max = 2,
14   .buffer_bytes_max = 32768,
15   .period_bytes_min = 4096,
16   .period_bytes_max = 32768,
17   .periods_min = 1,
18   .periods_max = 1024,
19 };
20
21 /* 录音设备硬件定义 */
22 static struct snd_pcm_hardware snd_xxxchip_capture_hw =
23 {
24   .info = (SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_INTERLEAVED |
25                                SNDRV_PCM_INFO_BLOCK_TRANSFER   |
SNDRV_PCM_INFO_MMAP_VALID),
26   .formats = SNDRV_PCM_FMTBIT_S16_LE,
27   .rates = SNDRV_PCM_RATE_8000_48000,
28   .rate_min = 8000,
29   .rate_max = 48000,
30   .channels_min = 2,
31   .channels_max = 2,
32   .buffer_bytes_max = 32768,
33   .period_bytes_min = 4096,
34   .period_bytes_max = 32768,
35   .periods_min = 1,
36   .periods_max = 1024,
37 };
38
39 /* 播放: 打开函数 */
40            static      int      snd_xxxchip_playback_open(struct
snd_pcm_substream*substream)
41 {
42   struct xxxchip *chip = snd_pcm_substream_chip(substream);
43   struct snd_pcm_runtime *runtime = substream->runtime;
```

```
  runtime->hw = snd_xxxchip_playback_hw;
  ...  // 硬件初始化代码
  return 0;
}

/* 播放: 关闭函数 */
static int snd_xxxchip_playback_close(struct snd_pcm_substream*substream)
{
  struct xxxchip *chip = snd_pcm_substream_chip(substream);
  // 硬件相关的代码
  return 0;
}

/* 录音: 打开函数 */
static int snd_xxxchip_capture_open(struct snd_pcm_substream*substream)
{
  struct xxxchip *chip = snd_pcm_substream_chip(substream);
  struct snd_pcm_runtime *runtime = substream->runtime;
  runtime->hw = snd_xxxchip_capture_hw;
  ...  // 硬件初始化代码
  return 0;
}

/* 录音: 关闭函数 */
static int snd_xxxchip_capture_close(struct snd_pcm_substream*substream)
{
  struct xxxchip *chip = snd_pcm_substream_chip(substream);
  ... // 硬件相关的代码
  return 0;
}
/* hw_params 函数 */
static int snd_xxxchip_pcm_hw_params(struct snd_pcm_substream*substream, struct
  snd_pcm_hw_params *hw_params)
{
```

```
    78                    return     snd_pcm_lib_malloc_pages(substream,
params_buffer_bytes(hw_params));
    79  }
    80  /* hw_free 函数 */
    81              static       int       snd_xxxchip_pcm_hw_free(struct
snd_pcm_substream*substream)
    82  {
    83    return snd_pcm_lib_free_pages(substream);
    84  }
    85  /* prepare 函数 */
    86              static       int       snd_xxxchip_pcm_prepare(struct
snd_pcm_substream*substream)
    87  {
    88    struct xxxchip *chip = snd_pcm_substream_chip(substream);
    89    struct snd_pcm_runtime *runtime = substream->runtime;
    90    /* 根据目前的配置信息设置硬件
    91     * 例如:
    92     */
    93    xxxchip_set_sample_format(chip, runtime->format);
    94    xxxchip_set_sample_rate(chip, runtime->rate);
    95    xxxchip_set_channels(chip, runtime->channels);
    96                xxxchip_set_dma_setup(chip,    runtime->dma_addr,
chip->buffer_size, chip
    97      ->period_size);
    98    return 0;
    99  }
    100 /* trigger 函数 */
    101         static        int        snd_xxxchip_pcm_trigger(struct
snd_pcm_substream*substream, int cmd)
    102 {
    103    switch (cmd)
    104    {
    105      case SNDRV_PCM_TRIGGER_START:
    106        // do something to start the PCM engine
    107        break;
    108      case SNDRV_PCM_TRIGGER_STOP:
    109        // do something to stop the PCM engine
    110        break;
    111      default:
    112        return  - EINVAL;
```

```
113  }
114 }
115
116 /* pointer 函数 */
117   static   snd_pcm_uframes_t   snd_xxxchip_pcm_pointer(struct
snd_pcm_substream
118  *substream)
119 {
120  struct xxxchip *chip = snd_pcm_substream_chip(substream);
121  unsigned int current_ptr;
122  /*获得当前的硬件指针*/
123  current_ptr = xxxchip_get_hw_pointer(chip);
124  return current_ptr;
125 }
126 /* 放音设备操作集 */
127 static struct snd_pcm_ops snd_xxxchip_playback_ops =
128 {
129  .open = snd_xxxchip_playback_open,
130  .close = snd_xxxchip_playback_close,
131  .ioctl = snd_pcm_lib_ioctl,
132  .hw_params = snd_xxxchip_pcm_hw_params,
133  .hw_free = snd_xxxchip_pcm_hw_free,
134  .prepare = snd_xxxchip_pcm_prepare,
135  .trigger = snd_xxxchip_pcm_trigger,
136  .pointer = snd_xxxchip_pcm_pointer,
137 };
138 /* 录音设备操作集 */
139 static struct snd_pcm_ops snd_xxxchip_capture_ops =
140 {
141  .open = snd_xxxchip_capture_open,
142  .close = snd_xxxchip_capture_close,
143  .ioctl = snd_pcm_lib_ioctl,
144  .hw_params = snd_xxxchip_pcm_hw_params,
145  .hw_free = snd_xxxchip_pcm_hw_free,
146  .prepare = snd_xxxchip_pcm_prepare,
147  .trigger = snd_xxxchip_pcm_trigger,
148  .pointer = snd_xxxchip_pcm_pointer,
149 };
150
151 /* 创建一个 PCM 设备 */
152 static int __devinit snd_xxxchip_new_pcm(struct xxxchip *chip)
```

```
{
  struct snd_pcm *pcm;
  int err;
  if ((err = snd_pcm_new(chip->card, "xxx Chip", 0, 1, 1, &pcm)) < 0)
      return err;
  pcm->private_data = chip;
  strcpy(pcm->name, "xxx Chip");
  chip->pcm = pcm;
  /* 设置操作集 */
  snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK, &snd_xxxchip_playback_ops);
  snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_CAPTURE, &snd_xxxchip_capture_ops);
  /* 分配缓冲区 */
  snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
    snd_dma_pci_data(chip - > pci), 64 *1024, 64 *1024);
  return 0;
}
```

17.4.4 控制接口

1. control

控制接口对于许多开关（switch）和调节器（slider）而言应用相当广泛，它能从用户空间被存取。control 的最主要用途是 mixer，所有的 mixer 元素基于 control 内核 API 实现，在 ALSA 中，control 用 snd_kcontrol 结构体描述。

ALSA 有一个定义很好的 AC97 控制模块，对于仅支持 AC97 的芯片而言，不必实现本节的内容。

创建一个新的 control 至少需要实现 snd_kcontrol_new 中的 info()、get()和 put()这 3 个成员函数，snd_kcontrol_new 结构体的定义如代码清单 17.16 所示。

代码清单 17.16　snd_kcontrol_new 结构体

```
struct snd_kcontrol_new
{
   snd_ctl_elem_iface_t iface; /*接口 ID，SNDRV_CTL_ELEM_IFACE_XXX */
   unsigned int device; /* 设备号 */
   unsigned int subdevice; /* 子流（子设备）号 */
   unsigned char *name; /* 名称(ASCII 格式) */
   unsigned int index; /* 索引 */
   unsigned int access; /* 访问权限 */
   unsigned int count; /* 享用元素的数量 */
   snd_kcontrol_info_t *info;
   snd_kcontrol_get_t *get;
```

```
    snd_kcontrol_put_t *put;
    unsigned long private_value;
};
```

iface 字段定义了 control 的类型，形式为 SNDRV_CTL_ELEM_IFACE_XXX，通常是 MIXER，对于不属于 mixer 的全局控制，使用 CARD。如果关联于某类设备，则使用 HWDEP、PCM、RAWMIDI、TIMER 或 SEQUENCER。

name 是名称标识字符串，control 的名称非常重要，因为 control 的作用由名称来区分。对于名称相同的 control，则使用 index 区分。name 定义的标准是“SOURCE DIRECTION FUNCTION”即“源方向功能”，SOURCE 定义了 control 的源，如“Master”、“PCM”、“CD”和“Line”，方向则为“Playback”、“Capture”、“Bypass Playback”或“Bypass Capture”，如果方向省略，意味着 playback 和 capture 双向，第 3 个参数可以是“Switch”、“Volume”和“Route”等。

“SOURCE DIRECTION FUNCTION”格式的名称例子如 Master Capture Switch、PCM Playback Volume。

下面几种 control 的命名不采用“SOURCE DIRECTION FUNCTION”格式，属于例外。

（1）全局控制。

“Capture Source”、“Capture Switch”和“Capture Volume”用于全局录音源、输入开关和录音音量控制；“Playback Switch”、“Playback Volume”用于全局输出开关和音量控制。

（2）音调控制。

音调控制名称的形式为“Tone Control – XXX”，例如“Tone Control – Switch”、“Tone Control – Bas”和“Tone Control – Center”。

（3）3D 控制。

3D 控制名称的形式为“3D Control – XXX”，例如“3D Control – Switch”、“3D Control – Center”和“3D Control – Space”。

（4）麦克风增益（Mic boost）。

麦克风增益被设置为“Mic Boost”或“Mic Boost (6dB)”。

snd_kcontrol_new 结构体的 access 字段是访问控制权限，形式如 SNDRV_CTL_ELEM_ ACCESS_XXX。SNDRV_CTL_ELEM_ACCESS_READ 意味着只读，这时 put()函数不必实现；SNDRV_CTL_ELEM_ACCESS_WRITE 意味着只写，这时 get()函数不必实现。若 control 值频繁变化，则需定义 VOLATILE 标志。当 control 处于非激活状态时，应设置 INACTIVE 标志。

private_value 字段包含一个长整型值，可以通过它给 info()、get()和 put()函数传递参数。

2．info()函数

snd_kcontrol_new 结构体中的 info()函数用于获得该 control 的详细信息，该函数必须填充传递给它的第二个参数 snd_ctl_elem_info 结构体，info()函数的形式如下：

华清远见

```
static int snd_xxxctl_info(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_info *uinfo);
```

snd_ctl_elem_info 结构体的定义如代码清单 17.17 所示。

代码清单 17.17　snd_ctl_elem_info 结构体

```
struct snd_ctl_elem_info
{
   struct snd_ctl_elem_id id; /* W: 元素 ID */
   snd_ctl_elem_type_t type; /* R: 值类型 - SNDRV_CTL_ELEM_TYPE_* */
       unsigned int access; /* R: 值访问权限(位掩码) - SNDRV_CTL_ELEM_ACCESS_* */
   unsigned int count; /* 值的计数 */
   pid_t owner; /* 该 control 的拥有者 PID */
   union
   {
     struct
     {
       long min; /* R: 最小值 */
       long max; /* R: 最大值 */
       long step; /* R: 值步进 (0 可变的) */
     } integer;
     struct
     {
       long long min; /* R: 最小值 */
       long long max; /* R: 最大值 */
       long long step; /* R: 值步进 (0 可变的) */
     } integer64;
     struct
     {
       unsigned int items; /* R: 项目数 */
       unsigned int item; /* W: 项目号 */
       char name[64]; /* R: 值名称 */
     } enumerated; /* 枚举 */
     unsigned char reserved[128];
  }
  value;
  union
  {
    unsigned short d[4];
    unsigned short *d_ptr;
  } dimen;
  unsigned char reserved[64-4 * sizeof(unsigned short)];
};
```

snd_ctl_elem_info 结构体的 type 字段定义了 control 的类型，包括 BOOLEAN、INTEGER、ENUMERATED、BYTES、IEC958 和 INTEGER64。count 字段定义了这个 control 中包含的元素的数量，例如一个立体声音量 control 的 count = 2。value 是一个联合体，其所存储的值的具体类型依赖于 type。代码清单 17.18 所示为一个 info()函数填充 snd_ctl_elem_info 结构体的范例。

代码清单 17.18　snd_ctl_elem_info 结构体中的 info()函数范例

```
1 static int snd_xxxctl_info(struct snd_kcontrol *kcontrol, struct
2     snd_ctl_elem_info *uinfo)
3 {
4     uinfo->type = SNDRV_CTL_ELEM_TYPE_BOOLEAN;//类型为 BOOLEAN
5     uinfo->count = 1;//数量为 1
6     uinfo->value.integer.min = 0;//最小值为 0
7     uinfo->value.integer.max = 1;//最大值为 1
8     return 0;
9 }
```

枚举类型和其他类型略有不同，对枚举类型，应为目前项目索引设置名称字符串，如代码清单 17.19 所示。

代码清单 17.19　填充 snd_ctl_elem_info 结构体中的权举类型值

```
1  static int snd_xxxctl_info(struct snd_kcontrol *kcontrol, struct
2      snd_ctl_elem_info *uinfo)
3  {
4     //值名称字符串
5     static char *texts[4] =
6    {
7      "First", "Second", "Third", "Fourth"
8    };
9      uinfo->type = SNDRV_CTL_ELEM_TYPE_ENUMERATED;//枚举类型
10     uinfo->count = 1;//数量为 1
11     uinfo->value.enumerated.items = 4;//项目数量为 1
12     //超过 3 的项目号改为 3
13     if (uinfo->value.enumerated.item > 3)
14       uinfo->value.enumerated.item = 3;
15     //为目前项目索引复制名称字符串
16                                strcpy(uinfo->value.enumerated.name,
texts[uinfo->value.enumerated.item]);
17     return 0;
18 }
```

3．get()函数

get()函数用于得到 control 的目前值并返回用户空间，代码清单 17.20 所示为 get()函数的范例。

代码清单 17.20　snd_ctl_elem_info 结构体中的 get()函数范例

```
static int snd_xxxctl_get(struct snd_kcontrol *kcontrol, struct
    snd_ctl_elem_value *ucontrol)
{
    //从 snd_kcontrol 获得 xxxchip 指针
    struct xxxchip *chip = snd_kcontrol_chip(kcontrol);
    //从 xxxchip 获得值并写入 snd_ctl_elem_value
    ucontrol->value.integer.value[0] = get_some_value(chip);
    return 0;
}
```

get()函数的第二个参数的类型为 snd_ctl_elem_value，其定义如代码清单 10.21 所示。snd_ctl_ elem_ value 结构体的内部也包含一个由 integer、integer64、enumerated 等组成的值联合体，它的具体类型依赖于 control 的类型和 info()函数。

代码清单 17.21　snd_ctl_elem_value 结构体

```
struct snd_ctl_elem_value
{
    struct snd_ctl_elem_id id; /* W: 元素 ID */
    unsigned int indirect: 1; /* W: 使用间接指针(xxx_ptr 成员) */
    //值联合体
    union
    {
       union
      {
         long value[128];
         long *value_ptr;
      } integer;
      union
      {
         long long value[64];
         long long *value_ptr;
      } integer64;
      union
      {
         unsigned int item[128];
         unsigned int *item_ptr;
      } enumerated;
      union
      {
         unsigned char data[512];
         unsigned char *data_ptr;
       } bytes;
       struct snd_aes_iec958 iec958;
    }
    value; /* 只读 */
    struct timespec tstamp;
    unsigned char reserved[128-sizeof(struct timespec)];
};
```

4．put()函数

put()用于从用户空间写入值，如果值被改变，该函数返回 1，否则返回 0；如果发生错误，该函数返回一个错误码。代码清单 17.22 所示为一个 put()函数的范例。

代码清单 17.22　snd_ctl_elem_info 结构体中的 put()函数范例

```
static int snd_xxxctl_put(struct snd_kcontrol *kcontrol, struct
    snd_ctl_elem_value *ucontrol)
{
    //从 snd_kcontrol 获得 xxxchip 指针
    struct xxxchip *chip = snd_kcontrol_chip(kcontrol);
    int changed = 0;//默认返回值为 0
    //值被改变
    if (chip->current_value != ucontrol->value.integer.value[0])
    {
                                        change_current_value(chip,
ucontrol->value.integer.value[0]);
        changed = 1;//返回值为 1
    }
    return changed;
}
```

对于 get()和 put()函数而言，如果 control 有多于一个元素，即 count >1，则每个元素都需要被返回或写入。

5．构造 control

当所有事情准备好后，我们需要创建一个 control，调用 snd_ctl_add()和 snd_ctl_new1()这两个函数来完成，这两个函数的原型为：

```
int snd_ctl_add(struct snd_card *card, struct snd_kcontrol *kcontrol);


struct snd_kcontrol *snd_ctl_new1(const struct snd_kcontrol_new
*ncontrol,
                        void *private_data);
```

snd_ctl_new1()函数用于创建一个 snd_kcontrol 并返回其指针，snd_ctl_add()函数用于将创建的 snd_kcontrol 添加到对应的 card 中。

6．变更通知

如果驱动中需要在中断服务程序中改变或更新一个 control，可以调用 snd_ctl_notify()函数，此函数原型为：

```
void snd_ctl_notify(struct snd_card *card, unsigned int mask, struct
snd_ctl_elem_id *id);
```

该函数的第二个参数为事件掩码（event-mask），第三个参数为该通知的 control 元素 id 指针。

例如，如下语句定义的事件掩码 SNDRV_CTL_EVENT_MASK_VALUE 意味着

control 值的改变被通知：

```
snd_ctl_notify(card, SNDRV_CTL_EVENT_MASK_VALUE, id_pointer);
```

17.4.5 AC97 API 接口

ALSA AC97 编解码层被很好地定义，利用它，驱动工程师只需编写少量底层的控制函数。

1. AC97 实例构造

为了创建一个 AC97 实例，首先需要调用 snd_ac97_bus()函数构建 AC97 总线及其操作，这个函数的原型为：

```
int snd_ac97_bus(struct snd_card *card, int num, struct snd_ac97_bus_ops *ops,
            void *private_data, struct snd_ac97_bus **rbus);
```

该函数的第 3 个参数 ops 是一个 snd_ac97_bus_ops 结构体，其定义如代码清单 17.23 所示。

代码清单 17.23 snd_ac97_bus_ops 结构体

```
struct snd_ac97_bus_ops
{
    void(*reset)(struct snd_ac97 *ac97); //复位函数
    //写入函数
    void(*write)(struct snd_ac97 *ac97, unsigned short reg, unsigned short val);
    //读取函数
    unsigned short(*read)(struct snd_ac97 *ac97, unsigned short reg);
    void(*wait)(struct snd_ac97 *ac97);
    void(*init)(struct snd_ac97 *ac97);
};
```

接下来，调用 snd_ac97_mixer()函数注册混音器，这个函数的原型为：

```
int snd_ac97_mixer(struct snd_ac97_bus *bus, struct snd_ac97_template *template, struct snd_ac97 **rac97);
```

代码清单 17.24 所示为 AC97 实例的创建过程。

代码清单 17.24 AC97 实例的创建过程范例

```
struct snd_ac97_bus *bus;
//AC97 总线操作
static struct snd_ac97_bus_ops ops =
{
  .write = snd_mychip_ac97_write,
  .read = snd_mychip_ac97_read,
  };
  //AC97 总线与操作创建
  snd_ac97_bus(card, 0, &ops, NULL, &bus);
```

```
10 //AC97 模板
11 struct snd_ac97_template ac97;
12 int err;
13 memset(&ac97, 0, sizeof(ac97));
14 ac97.private_data = chip;//私有数据
15 //注册混音器
16 snd_ac97_mixer(bus, &ac97, &chip->ac97);
```

上述代码第一行的 snd_ac97_bus 结构体指针 bus 的指针被传入第 9 行的 snd_ac97_bus()函数并被赋值，chip->ac97 的指针被传入第 16 行的 snd_ac97_mixer()并被赋值，chip->ac97 将成员新创建 AC97 实例的指针。

如果一个声卡上包含多个编解码器，这种情况下，需要多次调用 snd_ac97_mixer()并对 snd_ac97 的 num 成员（编解码器序号）赋予相应的序号。驱动中可以为不同的编解码器编写不同的 snd_ac97_bus_ops 成员函数中，或者只是在相同的一套成员函数中通过 ac97.num 获得序号后再区分进行具体的操作。

2. snd_ac97_bus_ops 成员函数

snd_ac97_bus_ops 结构体中的 read()和 write()成员函数完成底层的硬件访问，reset()函数用于复位编解码器，wait()函数用于编解码器标准初始化过程中的特定等待，如果芯片要求额外的等待时间，则应实现这个函数，init()用于完成编解码器附加的初始化。代码清单 17.25 所示为 read()和 write()函数的范例。

代码清单 17.25　snd_ac97_bus_ops 结构体中的 read()和 write()函数范例

```
1  static unsigned short snd_xxxchip_ac97_read(struct snd_ac97 *ac97, unsigned
2      short reg)
3  {
4      struct xxxchip *chip = ac97->private_data;
5      ...
6      return the_register_value; //返回寄存器值
7  }
8
9  static void snd_xxxchip_ac97_write(struct snd_ac97 *ac97, unsigned short reg,
10     unsigned short val)
11 {
12     struct xxxchip *chip = ac97->private_data;
13     ...
14     // 将被给的寄存器值写入 codec
15 }
```

3．修改寄存器

如果需要在驱动中访问编解码器，可使用如下函数：

```
void snd_ac97_write(struct snd_ac97 *ac97, unsigned short reg, unsigned short value);

int snd_ac97_update(struct snd_ac97 *ac97, unsigned short reg, unsigned short value);

int snd_ac97_update_bits(struct snd_ac97 *ac97, unsigned short reg, unsigned short mask, unsigned short value);

unsigned short snd_ac97_read(struct snd_ac97 *ac97, unsigned short reg);
```

snd_ac97_update()与 void snd_ac97_write()的区别在于前者在值已经设置的情况下不会再设置，而后者则会再写一次。snd_ac97_update_bits()用于更新寄存器的某些位，由 mask 决定。

除此之外，还有一个函数可用于设置采样率：

```
int snd_ac97_set_rate(struct snd_ac97 *ac97, int reg, unsigned int rate);
```

这个函数的第二个参数 reg 可以是 AC97_PCM_MIC_ADC_RATE、AC97_PCM_FRONT_DAC_ RATE、AC97_PCM_LR_ADC_RATE 和 AC97_SPDIF，对于 AC97 _SPDIF 而言，寄存器并非真地被改变了，只是相应的 IEC958 状态位将被更新。

4．时钟调整

在一些芯片上，编解码器的时钟频率不是 48000Hz，而是使用 PCI 时钟以节省一个晶振，在这种情况下，我们应该改变 bus->clock 为相应的值，例如 intel8x0 和 es1968 包含时钟的自动测量函数。

5．proc 文件

ALSA AC97 接口会创建如/proc/asound/card0/codec97#0/ac97#0-0 和 ac97#0-0+regs 这样的 proc 文件，通过这些文件可以查看编解码器目前的状态和寄存器。

如果一个芯片上有多个 codecs，可多次调用 snd_ac97_mixer()。

17.4.6 ALSA 用户空间编程

ALSA 驱动的声卡在用户空间不宜直接使用文件接口，而应使用 alsa-lib，代码清单 17.26 所示为基于 ALSA 音频驱动的最简单的播放应用程序。

代码清单 17.26　ALSA 用户空间播放程序

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <alsa/asoundlib.h>
4
```

```
5  main(int argc, char *argv[])
6  {
7    int i;
8    int err;
9    short buf[128];
10   snd_pcm_t *playback_handle;     //PCM 设备句柄
11   snd_pcm_hw_params_t *hw_params; //硬件信息和 PCM 流配置
12   //打开 PCM，最后一个参数为 0 意味着标准配置
13        if   ((err   =   snd_pcm_open(&playback_handle,   argv[1],
SND_PCM_STREAM_PLAYBACK, 0)
14     ) < 0)
15   {
16     fprintf(stderr, "cannot open audio device %s (%s)\n", argv[1],
snd_strerror
17       (err));
18     exit(1);
19   }
20   //分配 snd_pcm_hw_params_t 结构体
21   if ((err = snd_pcm_hw_params_malloc(&hw_params)) < 0)
22   {
23       fprintf(stderr, "cannot allocate hardware parameter structure
(%s)\n",
24       snd_strerror(err));
25       exit(1);
26   }
27   //初始化 hw_params
28   if ((err = snd_pcm_hw_params_any(playback_handle, hw_params)) < 0)
29   {
30          fprintf(stderr,  "cannot  initialize  hardware  parameter
structure (%s)\n",
31         snd_strerror(err));
32       exit(1);
33   }
34   //初始化访问权限
35       if  ((err  =  snd_pcm_hw_params_set_access(playback_handle,
hw_params,
36       SND_PCM_ACCESS_RW_INTERLEAVED)) < 0)
37   {
38             fprintf(stderr,  "cannot  set  access  type  (%s)\n",
snd_strerror(err));
39       exit(1);
40   }
41   //初始化采样格式
42       if  ((err  =  snd_pcm_hw_params_set_format(playback_handle,
```

```
hw_params,
    43      SND_PCM_FORMAT_S16_LE)) < 0)
    44   {
    45            fprintf(stderr,  "cannot  set  sample  format  (%s)\n",
snd_strerror(err));
    46      exit(1);
    47   }
    48   //设置采样率，如果硬件不支持我们设置的采样率，将使用最接近的
    49     if  ((err  =  snd_pcm_hw_params_set_rate_near(playback_handle,
hw_params, 44100,
    50    0)) < 0)
    51   {
    52           fprintf(stderr,  "cannot  set  sample  rate  (%s)\n",
snd_strerror(err));
    53    exit(1);
    54   }
    55    //设置通道数量
    56      if  ((err  =  snd_pcm_hw_params_set_channels(playback_handle,
hw_params, 2)) < 0)
    57   {
    58           fprintf(stderr,  "cannot  set  channel  count  (%s)\n",
snd_strerror(err));
    59     exit(1);
    60   }
    61   //设置 hw_params
    62   if ((err = snd_pcm_hw_params(playback_handle, hw_params)) < 0)
    63   {
    64              fprintf(stderr,  "cannot  set  parameters  (%s)\n",
snd_strerror(err));
    65      exit(1);
    66   }
    67   //释放分配的 snd_pcm_hw_params_t 结构体
    68   snd_pcm_hw_params_free(hw_params);
    69   //完成硬件参数设置，使设备准备好
    70   if ((err = snd_pcm_prepare(playback_handle)) < 0)
    71   {
    72    fprintf(stderr, "cannot prepare audio interface for use (%s)\n",
    73       snd_strerror(err));
    74    exit(1);
    75   }
```

```
76
77  for (i = 0; i < 10; ++i)
78  {
79    //写音频数据到 PCM 设备
80    if ((err = snd_pcm_writei(playback_handle, buf, 128)) != 128)
81    {
82       fprintf(stderr, "write to audio interface failed (%s)\n",
snd_strerror
83        (err));
84      exit(1);
85    }
86  }
87  //关闭 PCM 设备句柄
88  snd_pcm_close(playback_handle);
89  exit(0);
90 }
```

由上述代码可以看出，ALSA 用户空间编程的流程与 17.3.4 小节给出的 OSS 驱动用户空间编程的流程基本是一致的，都经过了“打开－设置参数－读写音频数据”的过程，不同在于 OSS 打开的是设备文件，设置参数使用的是 ioctl()系统调用，读写音频数据使用的是 read()、write()文件 API，而 ALSA 则全部使用 alsa-lib 中的 API。

把上述代码第 80 行的 snd_pcm_writei()函数替换为 snd_pcm_readi()，变成了一个最简单的录音程序。

代码清单 17.27 的程序打开一个音频接口，配置它为立体声、16 位、44.1kHz 采样和基于 interleave 的读写。它阻塞等待直接接口准备好接收放音数据，这时候将数据复制到缓冲区。这种设计方法使得程序很容易移植到类似 JACK、LADSPA、Coreaudio、VST 等 callback 机制驱动的系统。

代码清单 17.27　ALSA 用户空间播放程序（基于“中断”）

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <poll.h>
5  #include <alsa/asoundlib.h>
6
7  snd_pcm_t *playback_handle;
8  short buf[4096];
9
10 int playback_callback(snd_pcm_sframes_t nframes)
11 {
```

```
  int err;
  printf("playback callback called with %u frames\n", nframes);
  /* 填充缓冲区 */
  if ((err = snd_pcm_writei(playback_handle, buf, nframes)) < 0)
  {
     fprintf(stderr, "write failed (%s)\n", snd_strerror(err));
  }

  return err;
}

main(int argc, char *argv[])
{

  snd_pcm_hw_params_t *hw_params;
  snd_pcm_sw_params_t *sw_params;
  snd_pcm_sframes_t frames_to_deliver;
  int nfds;
  int err;
  struct pollfd *pfds;

        if  ((err  =  snd_pcm_open(&playback_handle,  argv[1], SND_PCM_STREAM_PLAYBACK, 0)
       ) < 0)
  {
    fprintf(stderr, "cannot open audio device %s (%s)\n", argv[1], snd_strerror
       (err));
    exit(1);
  }

  if ((err = snd_pcm_hw_params_malloc(&hw_params)) < 0)
  {
      fprintf(stderr, "cannot allocate hardware parameter structure (%s)\n",
         snd_strerror(err));
    exit(1);
  }

  if ((err = snd_pcm_hw_params_any(playback_handle, hw_params)) < 0)
```

```
    {
        fprintf(stderr, "cannot initialize hardware parameter structure (%s)\n",
        snd_strerror(err));
    exit(1);
    }

    if ((err = snd_pcm_hw_params_set_access(playback_handle, hw_params,
    SND_PCM_ACCESS_RW_INTERLEAVED)) < 0)
    {
        fprintf(stderr, "cannot set access type (%s)\n", snd_strerror(err));
    exit(1);
    }

    if ((err = snd_pcm_hw_params_set_format(playback_handle, hw_params,
    SND_PCM_FORMAT_S16_LE)) < 0)
    {
        fprintf(stderr, "cannot set sample format (%s)\n", snd_strerror(err));
    exit(1);
    }

    if ((err = snd_pcm_hw_params_set_rate_near(playback_handle, hw_params, 44100,
    0)) < 0)
    {
        fprintf(stderr, "cannot set sample rate (%s)\n", snd_strerror(err));
    exit(1);
    }

    if ((err = snd_pcm_hw_params_set_channels(playback_handle, hw_params, 2)) < 0)
    {
        fprintf(stderr, "cannot set channel count (%s)\n",
```

```
snd_strerror(err));
    79     exit(1);
    80   }
    81
    82   if ((err = snd_pcm_hw_params(playback_handle, hw_params)) < 0)
    83   {
    84             fprintf(stderr,  "cannot  set  parameters  (%s)\n",
snd_strerror(err));
    85     exit(1);
    86   }
    87
    88   snd_pcm_hw_params_free(hw_params);
    89
    90   /* 告诉 ALSA 当 4096 个以上帧可以传递时唤醒我们 */
    91   if ((err = snd_pcm_sw_params_malloc(&sw_params)) < 0)
    92   {
    93     fprintf(stderr, "cannot allocate software parameters structure
(%s)\n",
    94         snd_strerror(err));
    95     exit(1);
    96   }
    97   if ((err = snd_pcm_sw_params_current(playback_handle, sw_params))
< 0)
    98   {
    99          fprintf(stderr, "cannot initialize software parameters
structure (%s)\n",
    100         snd_strerror(err));
    101       exit(1);
    102    }
    103    /* 设置 4096 帧传递一次数据 */
    104     if ((err = snd_pcm_sw_params_set_avail_min(playback_handle,
sw_params, 4096))
    105       < 0)
    106    {
    107      fprintf(stderr, "cannot set minimum available count (%s)\n",
snd_strerror
    108        (err));
    109     exit(1);
    110   }
    111   /* 一旦有数据就开始播放 */
```

```
    if ((err = snd_pcm_sw_params_set_start_threshold(playback_handle, sw_params,
      0U)) < 0)
  {
        fprintf(stderr, "cannot set start mode (%s)\n", snd_strerror(err));
    exit(1);
  }
  if ((err = snd_pcm_sw_params(playback_handle, sw_params)) < 0)
  {
      fprintf(stderr, "cannot set software parameters (%s)\n", snd_strerror(err));
     exit(1);
  }

  /* 每 4096 帧接口将中断内核，ALSA 将很快唤醒本程序 */

  if ((err = snd_pcm_prepare(playback_handle)) < 0)
  {
      fprintf(stderr, "cannot prepare audio interface for use (%s)\n",
        snd_strerror(err));
     exit(1);
  }

  while (1)
  {

    /* 等待，直到接口准备好传递数据，或者 1s 超时发生 */
    if ((err = snd_pcm_wait(playback_handle, 1000)) < 0)
  {
      fprintf(stderr, "poll failed (%s)\n", strerror(errno));
      break;
    }

    /* 查出有多少空间可放置 playback 数据 */
    if ((frames_to_deliver = snd_pcm_avail_update(playback_handle)) < 0)
```

```
    {
        if (frames_to_deliver ==  - EPIPE)
        {
            fprintf(stderr, "an xrun occured\n");
            break;
        }
        else
        {
           fprintf(stderr, "unknown ALSA avail update return value (%d)\n",
              frames_to_deliver);
           break;
       }
    }

        frames_to_deliver = frames_to_deliver > 4096 ? 4096 : frames_to_deliver;

    /* 传递数据 */
    if (playback_callback(frames_to_deliver) != frames_to_deliver)
    {
      fprintf(stderr, "playback callback failed\n");
      break;
    }
  }

  snd_pcm_close(playback_handle);
  exit(0);
}
```

17.5 S3C2410+UDA1341 OSS 驱动实例

17.5.1 S3C2410 与 UDA1341 接口硬件描述

如图 17.7 所示，S3C2410 处理器内置了 IIS 总线接口，S3C2410 的 IIS 总线时钟信号 SCK 与 Philips 公司的 UDA1341 的 BCK 连接，字段选择连接于 WS 引脚。UDA1341 提供两个音频通道，分别用于输入和输出，对应的引脚连接：IIS 总线的音频输出 IISSDO 对应于 UDA1341 的音频输入，IIS 总线的音频输入 IISSDI 对应于 UDA1341 的音频输出。

UDA1341 的 L3 接口相当于一个混音器控制接口，可以用来控制输入/输出音频信号的音量大小、低音等。L3 接口的引脚 L3MODE、L3DATA、L3CLOCK 分别连接到 S3C2410

的 3 个 GPIO 来控制。

Philips 公司的 UDA1341 支持 IIS 总线数据格式，采用位元流转换技术进行信号处理，完成声音信号的 A/D 转换，具有可编程增益放大器和数字自动增益控制器，其低功耗、低电压的特点使其非常适合用于 MD/CD、笔记本电脑等便携式设备。UDA1341 对外提供两组音频信号输入接口，每组包括左右两个声道。

华清远见

如图 17.8 所示，两组音频输入在 UDA1341 内部的处理存在很大差别：第一组音频信号输入后经过一个 0 dB/6 dB 开关后采样送入数字混音器；第二组音频信号输

入后先经过可编程增益放大器（PGA），然后再进行采样，采样后的数据要再经过数字自动增益控制器（AGC）送入数字混音器。

图 17.8　UDA1341 内部结构

设计硬件电路时选用第二组输入音频信号，这样可以通过软件的方法实现对系统输入音量大小的调节。显然选用第二组可以通过 L3 总线接口控制 AGC 来实现。另外，选择通道 2 还可以通过 PGA 对从 MIC 输入的信号进行片内放大。

S3C2410 与 UDA1341 之间的 IIS 接口有以下 3 种工作方式。

- 正常传输模式：该模式下使用 IISCON 寄存器对 FIFO 进行控制，CPU 通过轮询方式访问 FIFO 寄存器，以完成对 FIFO 缓存传输或接收的处理。
- DMA 模式：通过设置 IISFCON 寄存器使 IIS 接口工作于这种模式。在该模式下，FIFO 寄存器组的控制权掌握在 DMA 控制器上，当 FIFO 满时，由

DMA 控制器对 FIFO 中的数据进行处理。DMA 模式的选择由 IISCON 寄存器的第 4 位和第 5 位控制。

- 传输/接收模式：在该模式下，IIS 数据线将通过双通道 DMA 同时接收和发送音频数据。在 OSS 驱动中，将使用此模式。

17.5.2 注册 dsp 和 mixer 接口

在 UDA1341 OSS 驱动的模块加载函数中，将完成如下工作，如代码清单 17.28 所示。

- 初始化 IIS 接口硬件，设置 L3 总线对应的 GPIO。
- 申请用于音频数据传输的 DMA 通道。
- 初始化 UDA1341 到恰当的工作模式。
- 注册 dsp 和 mixer 接口。

代码清单 17.28　UDA1341 OSS 驱动的模块加载函数

```
1  //音频(dsp)文件操作
2  static struct file_operations smdk2410_audio_fops =
3  {
4      llseek: smdk2410_audio_llseek,
5      write: smdk2410_audio_write,
6      read:  smdk2410_audio_read,
7      poll: smdk2410_audio_poll,
8      ioctl: smdk2410_audio_ioctl,
9      open: smdk2410_audio_open,
10     release: smdk2410_audio_release
11 };
12 //混音器文件操作
13 static struct file_operations smdk2410_mixer_fops =
14 {
15     ioctl: smdk2410_mixer_ioctl,
16     open: smdk2410_mixer_open,
17     release: smdk2410_mixer_release
18 };
19
20 int __init s3c2410_uda1341_init(void)
21 {
22     unsigned long flags;
23
24     local_irq_save(flags);
25
26     /* 设置IIS接口引脚GPIO */
27
28     set_gpio_ctrl(GPIO_L3CLOCK); // GPB 4: L3CLOCK, 输出
```

```
    29    set_gpio_ctrl(GPIO_L3DATA);  // GPB 3: L3DATA, 输出
    30    set_gpio_ctrl(GPIO_L3MODE);  // GPB 2: L3MODE, 输出
    31
    32
    33     set_gpio_ctrl(GPIO_E3 | GPIO_PULLUP_EN | GPIO_MODE_IISSDI);
//GPE 3: IISSDI
    34     set_gpio_ctrl(GPIO_E0 | GPIO_PULLUP_EN | GPIO_MODE_IISSDI);
//GPE 0: IISLRCK
    35     set_gpio_ctrl(GPIO_E1 | GPIO_PULLUP_EN | GPIO_MODE_IISSCLK);
//GPE 1:IISSCLK
    36    set_gpio_ctrl(GPIO_E2 | GPIO_PULLUP_EN | GPIO_MODE_CDCLK); //GPE
2: CDCLK
    37     set_gpio_ctrl(GPIO_E4 | GPIO_PULLUP_EN | GPIO_MODE_IISSDO);
//GPE 4: IISSDO
    38
    39    local_irq_restore(flags);
    40
    41    init_uda1341();
    42
    43    /* 输出流采样 DMA 通道 2 */
    44    output_stream.dma_ch = DMA_CH2;
    45
    46    if (audio_init_dma(&output_stream, "UDA1341 out"))
    47    {
    48       audio_clear_dma(&output_stream);
    49       printk(KERN_WARNING AUDIO_NAME_VERBOSE ": unable to get DMA
channels\n");
    50       return  - EBUSY;
    51    }
    52    /* 输入流采样 DMA 通道 1 */
    53    input_stream.dma_ch = DMA_CH1;
    54
    55    if (audio_init_dma(&input_stream, "UDA1341 in"))
    56    {
    57      audio_clear_dma(&input_stream);
    58      printk(KERN_WARNING AUDIO_NAME_VERBOSE ": unable to get DMA
channels\n");
    59      return  - EBUSY;
```

```
    }

        /* 注册 dsp 和 mixer 设备接口 */
        audio_dev_dsp = register_sound_dsp(&smdk2410_audio_fops, -1);
        audio_dev_mixer = register_sound_mixer(&smdk2410_mixer_fops, - 1);

        printk(AUDIO_NAME_VERBOSE " initialized\n");

        return 0;
}
```

UDA1341 OSS 驱动的模块卸载函数中，将完成与模块加载函数相反的工作，如代码清单 17.29 所示。

代码清单 17.29　UDA1341 OSS 驱动的模块卸载函数

```
void __exit s3c2410_uda1341_exit(void)
{
  //注销 dsp 和 mixer 设备接口
 unregister_sound_dsp(audio_dev_dsp);
 unregister_sound_mixer(audio_dev_mixer);

 //注销 DMA 通道
 audio_clear_dma(&output_stream);
 audio_clear_dma(&input_stream); /* input */
 printk(AUDIO_NAME_VERBOSE " unloaded\n");
}
```

17.5.3　mixer 接口的 I/O 控制函数

UDA1341 OSS 驱动的 ioctl()函数处理多个 mixer 命令，如 SOUND_MIXER_INFO、SOUND_ MIXER_READ_STEREODEVS、SOUND_MIXER_WRITE_VOLUME 等，用于获得或设置音量和增益等信息，如代码清单 17.30 所示。

代码清单 17.30　UDA1341 OSS 驱动的 ioctl()函数

```
static int smdk2410_mixer_ioctl(struct inode *inode, struct file *file,
   unsigned int cmd, unsigned long arg)
{
   int ret;
   long val = 0;

   switch (cmd)
   {
      case SOUND_MIXER_INFO:   //获得 mixer 信息
        {
           mixer_info info;
```

```
        strncpy(info.id, "UDA1341", sizeof(info.id));
                        strncpy(info.name,  "Philips  UDA1341",
sizeof(info.name));
        info.modify_counter = audio_mix_modcnt;
        return copy_to_user((void*)arg, &info, sizeof(info));
     }

    case SOUND_OLD_MIXER_INFO:
      {
        _old_mixer_info info;
        strncpy(info.id, "UDA1341", sizeof(info.id));
                        strncpy(info.name,  "Philips  UDA1341",
sizeof(info.name));
        return copy_to_user((void*)arg, &info, sizeof(info));
      }

     case SOUND_MIXER_READ_STEREODEVS://获取设备对立体声的支持
        return put_user(0, (long*)arg);

    case SOUND_MIXER_READ_CAPS:  //获取声卡能力
        val = SOUND_CAP_EXCL_INPUT;
        return put_user(val, (long*)arg);

    case SOUND_MIXER_WRITE_VOLUME:   //设置音量
        ret = get_user(val, (long*)arg);
        if (ret)
            return ret;
        uda1341_volume = 63-(((val &0xff) + 1) *63) / 100;
        uda1341_l3_address(UDA1341_REG_DATA0);
        uda1341_l3_data(uda1341_volume);
        break;

    case SOUND_MIXER_READ_VOLUME:   //获取音量
         val = ((63-uda1341_volume) *100) / 63;
         val |= val << 8;
         return put_user(val, (long*)arg);

    case SOUND_MIXER_READ_IGAIN:   //获得增益
         val = ((31-mixer_igain) *100) / 31;
         return put_user(val, (int*)arg);

    case SOUND_MIXER_WRITE_IGAIN:  //设置增益
         ret = get_user(val, (int*)arg);
         if (ret)
            return ret;
         mixer_igain = 31-(val *31 / 100);
         /* 使用 mixer 增益通道 1 */
         uda1341_l3_address(UDA1341_REG_DATA0);
         uda1341_l3_data(EXTADDR(EXT0));
         uda1341_l3_data(EXTDATA(EXT0_CH1_GAIN(mixer_igain)));
         break;
```

```
    61
    62      default:
    63         DPRINTK("mixer ioctl %u unknown\n", cmd);
    64         return  - ENOSYS;
    65   }
    66
    67   audio_mix_modcnt++;
    68   return 0;
    69 }
```

17.5.4 dsp 接口音频数据传输

OSS 声卡驱动中，dsp 接口的读写函数是核心，直接对应着录音和播放的流程。

OSS 的读函数存在一个与普通字符设备驱动读函数不同的地方。一般来说，对于普通字符设备驱动，如果用户要求读 count 个字节，而实际上只有 count1 字节可获得（count1< count）时，它会将这 count1 字节复制给用户后即返回 count1；而 dsp 接口的读函数会分次复制，如果第一次不能满足，它会等待第二次，直到“count1 + count2 + … = count”为止再返回 count。这种设计是合理的，因为 OSS 驱动应该负责音频数据的流量控制。代码清单 17.31 所示为 UDA1341 OSS 驱动的读函数实现。

代码清单 17.31　UDA1341 OSS 驱动的读函数

```
    1  static ssize_t smdk2410_audio_read(struct file *file, char *buffer,
size_t
    2     count, loff_t *ppos)
    3  {
    4     const char *buffer0 = buffer;
    5     audio_stream_t *s = &input_stream; //得到数据区的指针
    6     int chunksize, ret = 0;
    7
    8     DPRINTK("audio_read: count=%d\n", count);
    9
    10    if (ppos !=  &file->f_pos)
    11       return  - ESPIPE;
    12
    13    if (!s->buffers)
    14    {
    15        int i;
    16
    17        if (audio_setup_buf(s))
    18            return  - ENOMEM;
    19        //依次从缓存区读取数据
    20        for (i = 0; i < s->nbfrags; i++)
    21        {
    22             audio_buf_t *b = s->buf;
    23      down(&b->sem);
    24       s3c2410_dma_queue_buffer(s->dma_ch, (void*)b, b->dma_addr,
s->fragsize,
    25         DMA_BUF_RD);
    26      NEXT_BUF(s, buf);
```

```
    }
  }

  //满足用户的所有读需求
  while (count > 0)
  {
    audio_buf_t *b = s->buf;

    if (file->f_flags &O_NONBLOCK) //非阻塞
    {
        ret =  - EAGAIN;
        if (down_trylock(&b->sem))
            break;
    }
    else
    {
       ret =  - ERESTARTSYS;
       if (down_interruptible(&b->sem))
           break;
    }

    chunksize = b->size;
    //从缓存区读取数据
    if (chunksize > count)
        chunksize = count;
    DPRINTK("read %d from %d\n", chunksize, s->buf_idx);
    if (copy_to_user(buffer, b->start + s->fragsize - b->size,  //
调用复制函数
    chunksize))
    {
       up(&b->sem);
       return  - EFAULT;
    }
    b->size -= chunksize;

    buffer += chunksize;
    count -= chunksize;  //已经给用户复制了一部分，count 减少
    if (b->size > 0)
    {
        up(&b->sem);
        break;
    }
```

```
    //将缓存区释放
      s3c2410_dma_queue_buffer(s->dma_ch, (void*)b, b->dma_addr, s->fragsize,
        DMA_BUF_RD);

    NEXT_BUF(s, buf);
  }

  if ((buffer - buffer0))
      ret = buffer - buffer0;

  return ret;
}
```

OSS 驱动 dsp 接口的写函数与读函数类似，一般来说，它也应该满足用户的所有写需求后再返回，如代码清单 17.32 所示。

代码清单 17.32　UDA1341 OSS 驱动的写函数

```
static ssize_t smdk2410_audio_write(struct file *file, const char *buffer,
    size_t count, loff_t *ppos)
{
    const char *buffer0 = buffer;
    audio_stream_t *s = &output_stream;
    int chunksize, ret = 0;

    DPRINTK("audio_write : start count=%d\n", count);

    switch (file->f_flags &O_ACCMODE)
    {
      case O_WRONLY: //只写
      case O_RDWR: //读写
        break;
      default:  //只读不合法
        return  - EPERM;
    }
    //设置 DMA 缓冲区
    if (!s->buffers && audio_setup_buf(s))
       return  - ENOMEM;

    count &= ~0x03;

    while (count > 0) //直到满足用户的所有写需求
    {
      audio_buf_t *b = s->buf;
```

```
    //非阻塞访问
    if (file->f_flags &O_NONBLOCK)
     {
        ret =  - EAGAIN;
        if (down_trylock(&b->sem))
            break;
    }
    else
    {
        ret =  - ERESTARTSYS;
        if (down_interruptible(&b->sem))
            break;
   }
   //从用户空间复制音频数据
   if (audio_channels == 2)
   {
       chunksize = s->fragsize - b->size;
       if (chunksize > count)
           chunksize = count;
       DPRINTK("write %d to %d\n", chunksize, s->buf_idx);
       if (copy_from_user(b->start + b->size, buffer, chunksize))
     {
        up(&b->sem);
        return  - EFAULT;
      }
      b->size += chunksize;
  }
  else
  {
      chunksize = (s->fragsize - b->size) >> 1;

      if (chunksize > count)
          chunksize = count;
      DPRINTK("write %d to %d\n", chunksize *2, s->buf_idx);
      if (copy_from_user_mono_stereo(b->start + b->size, buffer, chunksize))
      {
          up(&b->sem);
          return  - EFAULT;
```

```
        }

         b->size += chunksize * 2;
      }

       buffer += chunksize;
       count -= chunksize;  //已经从用户复制了一部分，count 减少
       if (b->size < s->fragsize)
       {
           up(&b->sem);
           break;
      }
      //发起 DMA 操作
        s3c2410_dma_queue_buffer(s->dma_ch, (void*)b, b->dma_addr, b->size,
         DMA_BUF_WR);
      b->size = 0;
      NEXT_BUF(s, buf);
    }

    if ((buffer - buffer0))
        ret = buffer - buffer0;

     DPRINTK("audio_write : end count=%d\n\n", ret);

     return ret;
}
```

17.6 SA1100+ UDA1341 ALSA 驱动实例

17.6.1 card 注册与注销

同样是 UDA1341 芯片，如果以 ALSA 体系结构来实现它的驱动，会和 OSS 大不一样。在模块初始化和卸载的时候，需要注册和注销 card，另外在模块加载的时候，也会注册 mixer 和 PCM 组件，如代码清单 17.33 所示。

代码清单 17.33　UDA1341 ALSA 驱动的模块初始化与卸载

```
static int __init sa11xx_uda1341_probe(struct platform_device *devptr)
{
int err;
struct snd_card *card;
struct sa11xx_uda1341 *chip;

/* 新建 card */
    card = snd_card_new(-1, id, THIS_MODULE, sizeof(struct
```

```
sa11xx_uda1341));
    9  if (card == NULL)
    10     return -ENOMEM;
    11
    12  chip = card->private_data;
    13  spin_lock_init(&chip->s[0].dma_lock);
    14  spin_lock_init(&chip->s[1].dma_lock);
    15
    16  card->private_free = snd_sa11xx_uda1341_free;//card 私有数据释放
    17  chip->card = card;
    18  chip->samplerate = AUDIO_RATE_DEFAULT;
    19
    20  // 注册 control(mixer)接口
    21  if ((err = snd_chip_uda1341_mixer_new(card, &chip->uda1341)))
    22    goto nodev;
    23
    24  // 注册 PCM 接口
    25  if ((err = snd_card_sa11xx_uda1341_pcm(chip, 0)) < 0)
    26    goto nodev;
    27
    28  strcpy(card->driver, "UDA1341");
    29  strcpy(card->shortname, "H3600 UDA1341TS");
    30   sprintf(card->longname,  "Compaq  iPAQ  H3600  with  Philips
UDA1341TS");
    31
    32  snd_card_set_dev(card, &devptr->dev);
    33   //注册 card
    34  if ((err = snd_card_register(card)) == 0) {
    35    printk( KERN_INFO "iPAQ audio support initialized\n" );
    36    platform_set_drvdata(devptr, card);
    37    return 0;
    38  }
    39
    40  nodev:
    41  snd_card_free(card);
    42  return err;
    43 }
    44
    45 static int __devexit sa11xx_uda1341_remove(struct platform_device
*devptr)
    46 {
    47  //释放 card
    48  snd_card_free(platform_get_drvdata(devptr));
    49  platform_set_drvdata(devptr, NULL);
    50  return 0;
    51 }
```

17.6.2 PCM 设备的实现

PCM 组件直接对应着 ALSA 驱动的录音和播放，由 17.4.2 小节可知，驱动从需要定义对应相应的 snd_pcm_hardware 结构体进行 PCM 设备硬件描述，如代码清单 17.34 所示。

代码清单 17.34 UDA1341 ALSA 驱动的 PCM 接口的 snd_pcm_hardware 结构体

```
static struct snd_pcm_hardware snd_sa11xx_uda1341_capture =
{
.info              = (SNDRV_PCM_INFO_INTERLEAVED |
                   SNDRV_PCM_INFO_BLOCK_TRANSFER |
                   SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_MMAP_VALID |
                   SNDRV_PCM_INFO_PAUSE | SNDRV_PCM_INFO_RESUME),
.formats      = SNDRV_PCM_FMTBIT_S16_LE,
.rates           = (SNDRV_PCM_RATE_8000 | SNDRV_PCM_RATE_16000 |\
                 SNDRV_PCM_RATE_22050 | SNDRV_PCM_RATE_32000 |\
                 SNDRV_PCM_RATE_44100 | SNDRV_PCM_RATE_48000 |\
                 SNDRV_PCM_RATE_KNOT),
.rate_min       = 8000,
.rate_max          = 48000,
.channels_min          = 2,
.channels_max          = 2,
.buffer_bytes_max      = 64*1024,
.period_bytes_min      = 64,
.period_bytes_max      = DMA_BUF_SIZE,
.periods_min           = 2,
.periods_max           = 255,
.fifo_size             = 0,
};

static struct snd_pcm_hardware snd_sa11xx_uda1341_playback =
{
.info              = (SNDRV_PCM_INFO_INTERLEAVED |
                   SNDRV_PCM_INFO_BLOCK_TRANSFER |
                   SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_MMAP_VALID |
                   SNDRV_PCM_INFO_PAUSE | SNDRV_PCM_INFO_RESUME),
.formats      =SNDRV_PCM_FMTBIT_S16_LE,
.rates           = (SNDRV_PCM_RATE_8000 | SNDRV_PCM_RATE_16000 |\
                 SNDRV_PCM_RATE_22050 | SNDRV_PCM_RATE_32000 |\
                 SNDRV_PCM_RATE_44100 | SNDRV_PCM_RATE_48000 |\
```

```
                    SNDRV_PCM_RATE_KNOT),
 .rate_min              = 8000,
 .rate_max          = 48000,
 .channels_min          = 2,
 .channels_max      = 2,
 .buffer_bytes_max      = 64*1024,
 .period_bytes_min      = 64,
 .period_bytes_max      = DMA_BUF_SIZE,
 .periods_min           = 2,
 .periods_max           = 255,
 .fifo_size             = 0,
};
```

PCM 接口的主要函数被封装在 snd_pcm_ops 结构体内，UDA1341 ALSA 驱动对 snd_pcm_ops 结构体的定义如代码清单 17.35 所示。

代码清单 17.35　UDA1341 ALSA 驱动的 PCM 接口的 snd_pcm_ops 结构体

```
static struct snd_pcm_ops snd_card_sa11xx_uda1341_playback_ops =
{
 .open                = snd_card_sa11xx_uda1341_open,
 .close               = snd_card_sa11xx_uda1341_close,
 .ioctl               = snd_pcm_lib_ioctl,
 .hw_params           = snd_sa11xx_uda1341_hw_params,
 .hw_free             = snd_sa11xx_uda1341_hw_free,
 .prepare             = snd_sa11xx_uda1341_prepare,
 .trigger         = snd_sa11xx_uda1341_trigger,
 .pointer         = snd_sa11xx_uda1341_pointer,
};

static struct snd_pcm_ops snd_card_sa11xx_uda1341_capture_ops =
{
 .open                = snd_card_sa11xx_uda1341_open,
 .close               = snd_card_sa11xx_uda1341_close,
 .ioctl               = snd_pcm_lib_ioctl,
 .hw_params           = snd_sa11xx_uda1341_hw_params,
 .hw_free             = snd_sa11xx_uda1341_hw_free,
 .prepare             = snd_sa11xx_uda1341_prepare,
 .trigger             = snd_sa11xx_uda1341_trigger,
 .pointer             = snd_sa11xx_uda1341_pointer,
```

```
23 };
```

代码清单 17.33 第 25 行调用的 snd_card_sa11xx_uda1341_pcm()即是 PCM 组件的“构造函数”，其实现如代码清单 17.36 所示。

代码清单 17.36　UDA1341 ALSA 驱动的 PCM 组件构造函数

```
1  static int __init snd_card_sa11xx_uda1341_pcm(struct sa11xx_uda1341 *sa11xx_ uda1341, int device)
2  {
3  struct snd_pcm *pcm;
4  int err;
5  /* 新建 PCM 设备，playback 和 capture 都为 1 个 */
6  if ((err = snd_pcm_new(sa11xx_uda1341->card, "UDA1341 PCM", device, 1, 1, &pcm)) < 0)
7      return err;
8
9  /* 建立初始缓冲区并设置 dma_type 为 isa */
10 snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
11                    snd_dma_isa_data(),
12                    64*1024, 64*1024);
13 /* 设置 PCM 的操作 */
14          snd_pcm_set_ops(pcm,          SNDRV_PCM_STREAM_PLAYBACK, snd_card_sa11xx_uda1341_playback_ops);
15           snd_pcm_set_ops(pcm,           SNDRV_PCM_STREAM_CAPTURE, &snd_card_sa11xx_uda1341_capture_ops);
16 pcm->private_data = sa11xx_uda1341;
17 pcm->info_flags = 0;
18 strcpy(pcm->name, "UDA1341 PCM");
19
20 sa11xx_uda1341_audio_init(sa11xx_uda1341);
21
22 /* 设置 DMA 控制器 */
23     audio_dma_request(&sa11xx_uda1341->s[SNDRV_PCM_STREAM_PLAYBACK], audio_dma_callback);
24      audio_dma_request(&sa11xx_uda1341->s[SNDRV_PCM_STREAM_CAPTURE], audio_dma_callback);
25
26 sa11xx_uda1341->pcm = pcm;
27
28 return 0;
29 }
```

在 snd_pcm_ops 结构体的打开成员函数中，需要根据具体的子流赋值 snd_pcm_runtime 的 hw，如代码清单 17.37 所示。

代码清单 17.37　UDA1341 ALSA 驱动的 snd_pcm_ops 结构体的 open/close 成员函数

```
1  static int snd_card_sa11xx_uda1341_open(struct snd_pcm_substream *substream)
2  {
3  struct sa11xx_uda1341 *chip = snd_pcm_substream_chip(substream);
```

```
    4  struct snd_pcm_runtime *runtime = substream->runtime;
    5  int stream_id = substream->pstr->stream;
    6  int err;
    7
    8  chip->s[stream_id].stream = substream;
    9
    10 if (stream_id == SNDRV_PCM_STREAM_PLAYBACK) //播放子流
    11       runtime->hw = snd_sa11xx_uda1341_playback;
    12 else    //录音子流
    13      runtime->hw = snd_sa11xx_uda1341_capture;
    14     if     ((err     =     snd_pcm_hw_constraint_integer(runtime,
SNDRV_PCM_HW_PARAM_PERIODS)) < 0)
    15        return err;
    16                              if((err=snd_pcm_hw_constraint_list(runtime,0,
SNDRV_PCM_HW_PARAM_RATE,&hw_constraints_rates)) < 0)
    17       return err;
    18
    19 return 0;
    20 }
    21
    22 static int snd_card_sa11xx_uda1341_close(struct snd_pcm_substream
*substream)
    23 {
    24   struct sa11xx_uda1341 *chip = snd_pcm_substream_chip(substream);
    25
    26   chip->s[substream->pstr->stream].stream = NULL;
    27   return 0;
    28 }
```

在 snd_pcm_ops 结构体的 trigger()成员函数中，控制播放和录音的启/停、挂起/恢复，如代码清单 17.38 所示。

代码清单 17.38　UDA1341 ALSA 驱动的 snd_pcm_ops 结构体的 trigger 成员函数

```
    1    static int snd_sa11xx_uda1341_trigger(struct snd_pcm_substream
*substream, int
    2     cmd)
    3   {
    4     struct sa11xx_uda1341 *chip = snd_pcm_substream_chip(substream);
    5     int stream_id = substream->pstr->stream;
```

```
    struct audio_stream *s = &chip->s[stream_id];
    struct audio_stream *s1 = &chip->s[stream_id ^ 1];
    int err = 0;

    /* 注意本地中断已经被中间层代码禁止 */
    spin_lock(&s->dma_lock);
    switch (cmd)
    {
      case SNDRV_PCM_TRIGGER_START://开启 PCM
        if (stream_id == SNDRV_PCM_STREAM_CAPTURE && !s1->active) //开启录音，不在播放
        {
          s1->tx_spin = 1;
          audio_process_dma(s1);
        }
        else
        {
          s->tx_spin = 0;
        }

        /* 被请求的流启动 */
        s->active = 1;
        audio_process_dma(s);
        break;
      case SNDRV_PCM_TRIGGER_STOP:
        /* 被请求的流关闭 */
        audio_stop_dma(s);
        if (stream_id == SNDRV_PCM_STREAM_PLAYBACK && s1->active) //在录音时开启播放
        {
          s->tx_spin = 1;
          audio_process_dma(s);  //启动 DMA
        }
        else
        {
          if (s1->tx_spin)
          {
            s1->tx_spin = 0;
            audio_stop_dma(s1);  //停止 DMA
         }
```

```
    }

    break;
  case SNDRV_PCM_TRIGGER_SUSPEND: //挂起
    s->active = 0;
    #ifdef HH_VERSION
      sa1100_dma_stop(s->dmach); //停止 DMA
    #endif
    s->old_offset = audio_get_dma_pos(s) + 1;
    #ifdef HH_VERSION
      sa1100_dma_flush_all(s->dmach);
    #endif
    s->periods = 0;
    break;
   case SNDRV_PCM_TRIGGER_RESUME:   //恢复
    s->active = 1;
    s->tx_spin = 0;
    audio_process_dma(s);  //开启 DMA
    if (stream_id == SNDRV_PCM_STREAM_CAPTURE && !s1->active)
    {
      s1->tx_spin = 1;
      audio_process_dma(s1);
    }
    break;
   case SNDRV_PCM_TRIGGER_PAUSE_PUSH:   //暂停
     #ifdef HH_VERSION
       sa1100_dma_stop(s->dmach);  //停止 DMA
     #endif
     s->active = 0;
     if (stream_id == SNDRV_PCM_STREAM_PLAYBACK)
    {
       if (s1->active)
       {
         s->tx_spin = 1;
         s->old_offset = audio_get_dma_pos(s) + 1;
         #ifdef HH_VERSION
           sa1100_dma_flush_all(s->dmach);
         #endif
         audio_process_dma(s); //开启 DMA
       }
    }
    else
```

```
    86        {
    87           if (s1->tx_spin)
    88           {
    89              s1->tx_spin = 0;
    90              #ifdef HH_VERSION
    91                 sa1100_dma_flush_all(s1->dmach);
    92              #endif
    93           }
    94        }
    95        break;
    96       case SNDRV_PCM_TRIGGER_PAUSE_RELEASE:   //暂停释放
    97        s->active = 1;
    98        if (s->old_offset)
    99        {
    100           s->tx_spin = 0;
    101           audio_process_dma(s);
    102           break;
    103        }
    104        if (stream_id == SNDRV_PCM_STREAM_CAPTURE && !s1->active)
    105        {
    106           s1->tx_spin = 1;
    107           audio_process_dma(s1);
    108        }
    109        #ifdef HH_VERSION
    110           sa1100_dma_resume(s->dmach);
    111        #endif
    112        break;
    113      default:
    114         err =  - EINVAL;
    115         break;
    116     }
    117     spin_unlock(&s->dma_lock);
    118     return err;
    119 }
```

snd_pcm_ops 结构体中其他的 hw_params()、prepare()、pointer()等成员函数实现较为简单，这里不再赘述。

17.6.3 控制接口的实现

代码清单 17.33 第 21 行调用的 snd_chip_uda1341_mixer_new()可以认为是 UDA1341 ALSA 驱动 mixer 控制组件的“构造函数”，其中会创建的控制元素的定义如代码清单 17.39 所示，包括一些枚举和单值元素。

代码清单 17.39 UDA1341 ALSA 驱动控制接口的 snd_kcontrol_new 结构体

```
    1  #define UDA1341_SINGLE(xname, where, reg, shift, mask, invert) \
    2   {  .iface  =  SNDRV_CTL_ELEM_IFACE_MIXER,  .name  =  xname,  .info  =
snd_uda1341_info_single,\
    3     .get = snd_uda1341_get_single, .put = snd_uda1341_put_single, \
    4     .private_value = where | (reg << 5) | (shift << 9) | (mask << 12)
| (invert <<18)\
    5   }
    6
    7   #define UDA1341_ENUM(xname, where, reg, shift, mask, invert) \
    8     {  .iface  =  SNDRV_CTL_ELEM_IFACE_MIXER,  .name  =  xname,  .info  =
snd_uda1341_info_enum, \
```

```
    9       .get = snd_uda1341_get_enum, .put = snd_uda1341_put_enum, \
    10      .private_value = where | (reg << 5) | (shift << 9) | (mask << 12) |
(invert << 18) \
    11 }
    12
    13 static struct snd_kcontrol_new snd_uda1341_controls[] =
    14 {
    15   UDA1341_SINGLE("Master Playback Switch", CMD_MUTE, data0_2, 2, 1,
1),
    16   UDA1341_SINGLE("Master Playback Volume", CMD_VOLUME, data0_0, 0,
63, 1),
    17
    18   UDA1341_SINGLE("Bass Playback Volume", CMD_BASS, data0_1, 2, 15,
0),
    19   UDA1341_SINGLE("Treble Playback Volume", CMD_TREBBLE, data0_1, 0,
3, 0),
    20
    21   UDA1341_SINGLE("Input Gain Switch", CMD_IGAIN, stat1, 5, 1, 0),
    22   UDA1341_SINGLE("Output Gain Switch", CMD_OGAIN, stat1, 6, 1, 0),
    23
    24   UDA1341_SINGLE("Mixer Gain Channel 1 Volume", CMD_CH1, ext0, 0, 31,
1),
    25   UDA1341_SINGLE("Mixer Gain Channel 2 Volume", CMD_CH2, ext1, 0, 31,
1),
    26
    27   UDA1341_SINGLE("Mic Sensitivity Volume", CMD_MIC, ext2, 2, 7, 0),
    28
    29   UDA1341_SINGLE("AGC Output Level", CMD_AGC_LEVEL, ext6, 0, 3, 0),
    30   UDA1341_SINGLE("AGC Time Constant", CMD_AGC_TIME, ext6, 2, 7, 0),
    31   UDA1341_SINGLE("AGC Time Constant Switch", CMD_AGC, ext4, 4, 1, 0),
    32
    33   UDA1341_SINGLE("DAC Power", CMD_DAC, stat1, 0, 1, 0),
    34   UDA1341_SINGLE("ADC Power", CMD_ADC, stat1, 1, 1, 0),
    35
    36   UDA1341_ENUM("Peak detection", CMD_PEAK, data0_2, 5, 1, 0),
    37   UDA1341_ENUM("De-emphasis", CMD_DEEMP, data0_2, 3, 3, 0),
    38   UDA1341_ENUM("Mixer mode", CMD_MIXER, ext2, 0, 3, 0),
    39   UDA1341_ENUM("Filter mode", CMD_FILTER, data0_2, 0, 3, 0),
    40
    41   UDA1341_2REGS("Gain Input Amplifier Gain (channel 2)", CMD_IG, ext4,
ext5,0,0,3,31, 0),
    42 };
```

从上述代码中宏 UDA1341_SINGLE 和 UDA1341_ENUM 的定义可知，单值元素的 info()、get()、put() 成员函数分别为 snd_uda1341_info_single()、snd_uda1341_get_single()和 snd_uda1341_put _single()；枚举元素的 info()、get()、put() 成员函数分别为 snd_uda1341_info_enum()、snd_ uda1341_get_enum()、和 snd_uda1341_put_enum()。作为例子，代码清单 17.40 给出了单值元素相关函数的实现。

代码清单 17.40　UDA1341 ALSA 驱动控制接口的单值元素 info/get/put 函数

```
    1  static int snd_uda1341_info_single(struct snd_kcontrol *kcontrol,
```

```
struct
    snd_ctl_elem_info *uinfo)
{
    int mask = (kcontrol->private_value >> 12) &63;

    uinfo->type = mask == 1 ? SNDRV_CTL_ELEM_TYPE_BOOLEAN :
       SNDRV_CTL_ELEM_TYPE_INTEGER;
    uinfo->count = 1;  //数量为 1
    uinfo->value.integer.min = 0;  //最小值
    uinfo->value.integer.max = mask; //最大值
    return 0;
}

static int snd_uda1341_get_single(struct snd_kcontrol *kcontrol,
struct
    snd_ctl_elem_value *ucontrol)
{
    struct l3_client *clnt = snd_kcontrol_chip(kcontrol);
    struct uda1341 *uda = clnt->driver_data;
    int where = kcontrol->private_value &31;
    int mask = (kcontrol->private_value >> 12) &63;
    int invert = (kcontrol->private_value >> 18) &1;

     ucontrol->value.integer.value[0] = uda->cfg[where]; //返回给
ucontrol
    if (invert) //如果反转
                    ucontrol->value.integer.value[0]  =  mask  -
ucontrol->value.integer.value[0];

     return 0;
}

static int snd_uda1341_put_single(struct snd_kcontrol *kcontrol,
struct
    snd_ctl_elem_value *ucontrol)
{
    struct l3_client *clnt = snd_kcontrol_chip(kcontrol);
    struct uda1341 *uda = clnt->driver_data;
    int where = kcontrol->private_value &31;
    int reg = (kcontrol->private_value >> 5) &15;
    int shift = (kcontrol->private_value >> 9) &7;
    int mask = (kcontrol->private_value >> 12) &63;
    int invert = (kcontrol->private_value >> 18) &1;
    unsigned short val;

    val = (ucontrol->value.integer.value[0] &mask);//从 ucontrol 获
得值
    if (invert) //如果反转
    val = mask - val;
```

```
    45
    46    uda->cfg[where] = val;
    47     return snd_uda1341_update_bits(clnt, reg, mask, shift, val,
FLUSH);//更新位
    48 }
```

17.7 PXA255+AC97 ALSA 驱动实例

Intel 公司的 XScale PXA255 是一款基于 ARM 内核技术的嵌入式处理器。它提供了符合 AC97 rev2.0 标准的 AC97 控制单元（ACUNIT）和音频控制连接（AC-Link）。ACUNIT 就是 CODEC 控制器，它通过 AC-Link 连接和控制 AC97 CODEC 芯片，例如 Philips 公司出品的一款符合 AC97 标准的多功能 CODEC 芯片 UCB1400。它不仅是一枚 CODEC 芯片，还集成了触摸和能量管理两个功能模块，在嵌入式系统中应用广泛。

AC-Link 连接了 ACUNIT 和 UCB1400Codec，只要对 ACUNIT 寄存器操作就可以实现同 UCBI400 之间的数据传输。通过 ACUNIT 还可读写 CODEC 内部寄存器，实现对音频采样和混音处理的控制。当然这些读写操作也是经由 AC-Link 传输的。

Intel Xscale PXA255 提供了 16 个 DMA 通道，可以很方便地为外围设备提供数据传送。ACUNIT 也为 CODEC 的立体声输入/输出和话筒输入提供了独立的 16 bit 数据通道。每个通道都有一个专门的 FIFO。

由于它是一个标准的 AC97 设备，因此，其驱动的控制部分实现非常简单，按照 17.4.4 小节的要求，需要实现 AC97 codec 寄存器读写的硬件级函数 pxa2xx_ac97_read() 和 pxa2xx_ac97_ write()，并在模块初始化时注册相关的 AC97 组件，如代码清单 17.41 所示。

代码清单 17.41　PXA255 连接 AC97 codec ALSA 驱动控制组件

```
    1    static unsigned short pxa2xx_ac97_read(struct snd_ac97 *ac97,
unsigned short
    2      reg)
    3   {
    4      unsigned short val =  - 1;
    5      volatile u32 *reg_addr;
    6
    7      down(&car_mutex);
    8
    9      /* 设置首/次 codec 空间 */
    10    reg_addr = (ac97->num &1) ? &SAC_REG_BASE: &PAC_REG_BASE;
```

```
    11    reg_addr += (reg >> 1);
    12
    13    /* 通过ac97 link读 */
    14    GSR = GSR_CDONE | GSR_SDONE;
    15    gsr_bits = 0;
    16    val =  *reg_addr;
    17    if (reg == AC97_GPIO_STATUS)
    18        goto out;
    19    if (wait_event_timeout(gsr_wq, (GSR | gsr_bits) &GSR_SDONE, 1)
<= 0
    20        && !((GSR | gsr_bits) &GSR_SDONE))
    21    //等待
    22    {
    23       printk(KERN_ERR "%s: read error (ac97_reg=%d GSR=%#lx)\n",
    24         __FUNCTION__,reg, GSR | gsr_bits);
    25       val =  - 1;
    26       goto out;
    27   }
    28
    29   /* 置数据有效 */
    30   GSR = GSR_CDONE | GSR_SDONE;
    31   gsr_bits = 0;
    32   val =  *reg_addr;
    33   /*已经开启另一个周期... */
    34   wait_event_timeout(gsr_wq, (GSR | gsr_bits) &GSR_SDONE, 1);
    35
    36   out: up(&car_mutex);
    37   return val;
    38 }
    39
    40 static void pxa2xx_ac97_write(struct snd_ac97 *ac97, unsigned short
reg,
    41    unsigned short val)
    42 {
    43    volatile u32 *reg_addr;
    44
    45    down(&car_mutex);
    46
    47    /*设置首/次codec空间*/
    48    reg_addr = (ac97->num &1) ? &SAC_REG_BASE: &PAC_REG_BASE;
    49    reg_addr += (reg >> 1);
    50
    51    GSR = GSR_CDONE | GSR_SDONE;
```

```
    gsr_bits = 0;
    *reg_addr = val; //通过AC97 link写
    if (wait_event_timeout(gsr_wq, (GSR | gsr_bits) &GSR_CDONE, 1) <= 0
       && !((GSR | gsr_bits) &GSR_CDONE))
       printk(KERN_ERR "%s: write error (ac97_reg=%d GSR=%#lx)\n",
         __FUNCTION__,reg, GSR | gsr_bits);

    up(&car_mutex);
}

 static int pxa2xx_ac97_probe(struct platform_device *dev)
{
      struct snd_card *card;
      struct snd_ac97_bus *ac97_bus;
      struct snd_ac97_template ac97_template;
      int ret;

      ret =  - ENOMEM;
      /* 新建card */
        card = snd_card_new(SNDRV_DEFAULT_IDX1, SNDRV_DEFAULT_STR1, THIS_MODULE, 0);
      if (!card)
         goto err;
      card->dev = &dev->dev;
                    strncpy(card->driver,   dev->dev.driver->name, sizeof(card->driver));

      /* 构造PCM组件 */
             ret  =  pxa2xx_pcm_new(card,  &pxa2xx_ac97_pcm_client, &pxa2xx_ac97_pcm);
      if (ret)
         goto err;

    /* 申请中断 */
    ret = request_irq(IRQ_AC97, pxa2xx_ac97_irq, 0, "AC97", NULL);
    if (ret < 0)
        goto err;

    ...

     /* 初始化AC97 bus */
```

```
    ret = snd_ac97_bus(card, 0, &pxa2xx_ac97_ops, NULL, &ac97_bus);
    if (ret)
       goto err;
    memset(&ac97_template, 0, sizeof(ac97_template));
    ret = snd_ac97_mixer(ac97_bus, &ac97_template, &pxa2xx_ac97_ac97);
    if (ret)
       goto err;
   ...

   /* 注册card */
   ret = snd_card_register(card);
   if (ret == 0)
  {
      platform_set_drvdata(dev, card);
      return 0;
  }

  err: if (card)
      snd_card_free(card);
  ...

   returns ret;
}
```

17.8 总结

音频设备接口包括 PCM、IIS 和 AC97 等，分别适用于不同的应用场合。针对音频设备，Linux 内核中包含了两类音频设备驱动框架，OSS 和 ALSA，前者包含 dsp 和 mixer 字符设备接口，在用户空间的编程中，完全使用文件操作；后者以 card 和组件（PCM、mixer 等）为主线，在用户空间的编程中不使用文件接口而使用 alsalib。

在音频设备驱动中，几乎必须使用 DMA，而 DMA 的缓冲区会被分割成一个一个的段，每次 DMA 操作进行其中的一段。OSS 驱动的阻塞读写具有流控能力，在用户空间不需要进行流量方面的定时工作，但是它需要及时地写（播放）和读（录音），以免出现缓冲区的 underflow 或 overflow。

第 18 章　LCD 设备驱动

在多媒体应用的推动下，彩色 LCD 越来越多地应用到了嵌入式系统中，掌上电脑（PDA）、手机等多采用 TFT 显示器件，支持彩色图形界面，能显示图片并进行视频媒体播放。帧缓冲（Framebuffer）是 Linux 为显示设备提供的一个接口，它允许上层应用程序在图形模式下直接对显示缓冲区进行读写操作。

本章主要讲解帧缓冲设备 Linux 驱动的架构及编程方法。

18.1 节讲解了 LCD 的底层硬件操作原理，18.2 节讲解了帧缓冲设备的概念及驱动中的重要数据结构和函数。

18.3 节讲解了帧缓冲设备驱动的整体结构，18.4～18.8 节分别讲解了帧缓冲设备的几个重要函数，18.3 节和 18.4～18.8 节的内容是整体与部分的关系。

18.9 节讲解了 Linux 帧缓冲设备用户空间的访问方法，并给出了两个常用的 GUI 环境—QT/E 及 Minigui 的基本用法。

18.10 节讲解了 S3C2410 LCD 控制器设备驱动的实例。

华清远见

18.1 LCD 硬件原理

利用液晶制成的显示器称为 LCD，依据驱动方式可分为静态驱动、简单矩阵驱动以及主动矩阵驱动 3 种。其中，简单矩阵型又可再细分扭转向列型（TN）和超扭转式向列型（STN）两种，而主动矩阵型则以薄膜式晶体管型（TFT）为主流。表 18.1 列出了 TN、STN 和 TFT 显示器的区别。

表 18.1　　TN、STN 和 TFT 显示器的区别

类　　别	TN	STN	TFT
原理	液晶分子，扭转 90°	扭转 180°～270°	液晶分子，扭转 90°
特性	黑白、单色低对比	黑白、彩色，低对比	彩色（1667 万色），可媲美 CRT 显示器的全彩，高对比
动画显示	否	否	是
视角	30°以下	40°以下	80°以下
面板尺寸	1～3 英寸	1～12 英寸	37 英寸

TN 型液晶显示技术是 LCD 中最基本的，其他种类的 LCD 都以 TN 型为基础改进而得。TN 型 LCD 显示质量很差，色彩单一，对比度低，反映速度很慢，故主要用于简单的数字符与文字的显示，如电子表及电子计算器等。

STN LCD 的显示原理与 TN 类似，区别在于 TN 型的液晶分子将入射光旋转 90°，而 STN 则可将入射光旋转 180°～270°。STN 改善了 TN 视角狭小的缺点，并提高了对比度，显示品质较 TN 高。

STN 搭配彩色滤光片，将单色显示矩阵的任一像素分成 3 个子像素，分别透过彩色滤光片显示红、绿、蓝三原色，再经由三原色按比例调和，显示出逼近全彩模式的色彩。STN 显示的画面色彩对比度仍较小，反应速度也较慢，可以作为一般的操作显示接口。

随后出现的 DSTN 通过双扫描方式来显示，显示效果相对 STN 而言有了较大幅度的提高。DSTN 的反应速度可达到 100ms，但是在电场反复改变电压的过程中，每一像素的恢复过程较慢。因此，当在屏幕画面快速变化时，会产生“拖尾”现象。

TN 与 STN 型液晶显示器都是使用场电压驱动方式，如果显示尺寸加大，中心部位对电极变化的反应时间就会拉长，显示器的速度跟不上。为了解决这个问题，主动式矩阵驱动被提出，主动式 TFT 型的液晶显示器的结构较为复杂，它包括背光管、导光板、偏光板、滤光板、玻璃基板、配向膜、液晶材料和薄膜式晶体管等。

在 TFT 型 LCD 中，晶体管矩阵依显示信号开启或关闭液晶分子的电压，使液晶分子轴转向而成“亮”或“暗”的对比，避免了显示器对电场效应的依靠。因此，TFT

华清远见

LCD 的显示质量较 TN/STN 更佳，画面显示对比度可达 150:1 以上，反应速度逼近 30ms 甚至更快，适用于 PDA、笔记本电脑、数码相机、MP4 等。

一块 LCD 屏显示图像不但需要 LCD 驱动器，还需要有相应的 LCD 控制器。通常 LCD 驱动器会以 COF/COG 的形式与 LCD 玻璃基板制作在一起，而 LCD 控制器则由外部电路来实现。许多 MCU 内部直接集成了 LCD 控制器，通过 LCD 控制器可以方便地控制 STN 和 TFT 屏。

TFT 屏是目前嵌入式系统应用的主流，图 18.1 给出了 TFT 屏的典型时序。时序图中的 VCLK、HSYNC 和 VSYNC 分别为像素时钟信号（用于锁存图像数据的像素时钟）、行同步信号和帧同步信号，VDEN 为数据有效标志信号，VD 为图像的数据信号。

图 18.1　TFT 屏工作时序

作为帧同步信号的 VSYNC，每发出一个脉冲，都意味着新的一屏图像数据开始发送。而作为行同步信号的 HSYNC，每发出一个脉冲都表明新的一行图像资料开始发送。在帧同步以及行同步的头尾都必须留有回扫时间。这样的时序安排起源于 CRT 显示器电子枪偏转所需要的时间，但后来成为实际上的工业标准，因此 TFT 屏也包含了回扫时间。

图 18.2 给出了 LCD 控制器中应该设置的 TFT 屏的参数，其中的上边界和下边界

即为帧切换的回扫时间，左边界和右边界即为行切换的回扫时间，水平同步和垂直同步分别是行和帧同步本身需要的时间。xres 和 yres 则分别是屏幕的水平和垂直分辨率，常见的嵌入式设备的 LCD 分辨率主要为 320*240、640*480 等。

图 18.2　LCD 控制器中的时序参数设置

18.2 帧缓冲

18.2.1　帧缓冲的概念

帧缓冲（framebuffer）是 Linux 系统为显示设备提供的一个接口，它将显示缓冲区抽象，屏蔽图像硬件的底层差异，允许上层应用程序在图形模式下直接对显示缓冲区进行读写操作。用户不必关心物理显示缓冲区的具体位置及存放方式，这些都由帧缓冲设备驱动本身来完成。对于帧缓冲设备而言，只要在显示缓冲区中与显示点对应的区域写入颜色值，对应的颜色会自动在屏幕上显示，18.2.2 小节将讲解显示缓冲区与显示点的对应关系。

帧缓冲设备为标准字符设备，主设备号为 29，对应于/dev/fb%d 设备文件。帧缓冲驱动的应用非常广泛，在 Linux 的桌面系统中，Xwindow 服务器就是利用帧缓冲进行窗口的绘制。嵌入式系统中的 Qt/Embedded 等图形用户界面环境也基于帧缓冲而设计。另外，通过帧缓冲可支持汉字点阵的显示，因此帧缓冲也成为 Linux 汉化的可行方案。

18.2.2　显示缓冲区与显示点

在帧缓冲设备中，对屏幕显示点的操作通过读写显示缓冲区来完成，在不同的色彩模式下，显示缓冲区和屏幕上的显示点有不同的对应关系，表 18.2～表 18.4 分别给

出了 16 级灰度、8 位色和 16 位情况下显示缓冲区与显示点的对应关系。

表 18.2　　16 级灰度显示缓冲区与显示点的对应关系

位	31～28	27～24	23～20	19～16	15～12	11～8	7～4	3～0
0x0	点 7	点 6	点 5	点 4	点 3	点 2	点 1	点 0
0x04	点 15	点 14	点 13	点 12	点 11	点 10	点 9	点 8
…	…	…	…	…	…	…	…	…

续表

位	31～28	27～24	23～20	19～16	15～12	11～8	7～4	3～0
0x0	点 0	点 1	点 2	点 3	点 4	点 5	点 6	点 7
0x04	点 8	点 9	点 10	点 11	点 12	点 13	点 14	点 15
…	…	…	…	…	…	…	…	…

表 18.3　　8 位色时显示缓冲区与显示点的对应关系

RGB			BGR		
7～5	4～2	1～0	7～5	4～2	1～0
R	G	B			

表 18.4　　16 位色时显示缓冲区与显示点的对应关系

位	15～11	10～5	4～0
RGB565	R	G	B
RGB555	R	G	B

18.2.3　Linux 帧缓冲相关数据结构与函数

1．fb_info 结构体

帧缓冲设备最关键的一个数据结构体是 fb_info 结构体（为了便于记忆，我们把它简称为“FBI”），FBI 中包括了关于帧缓冲设备属性和操作的完整描述，这个结构体的定义如代码清单 18.1 所示。

代码清单 18.1　fb_info 结构体

```
struct fb_info
{
   int node;
   int flags;
   struct fb_var_screeninfo var; /*可变参数 */
   struct fb_fix_screeninfo fix; /*固定参数 */
   struct fb_monspecs monspecs; /*显示器标准 */
   struct work_struct queue; /* 帧缓冲事件队列 */
   struct fb_pixmap pixmap; /* 图像硬件mapper */
   struct fb_pixmap sprite; /* 光标硬件mapper */
   struct fb_cmap cmap; /* 目前的颜色表*/
   struct list_head modelist;
   struct fb_videomode *mode; /* 目前的video模式 */

```

```
15    #ifdef CONFIG_FB_BACKLIGHT
16      struct mutex bl_mutex;
17      /* 对应的背光设备 */
18      struct backlight_device *bl_dev;
19      /* 背光调整 */
20      u8 bl_curve[FB_BACKLIGHT_LEVELS];
21    #endif
22
23    struct fb_ops *fbops; /* fb_ops，帧缓冲操作 */
24    struct device *device;
25    struct class_device *class_device; /
26    int class_flag; /* 私有sysfs标志 */
27    #ifdef CONFIG_FB_TILEBLITTING
28       struct fb_tile_ops *tileops; /* 图块Blitting */
29    #endif
30    char __iomem *screen_base; /* 虚拟基地址 */
31    unsigned long screen_size; /* ioremapped的虚拟内存大小 */
32    void *pseudo_palette; /* 伪16色颜色表 */
33    #define FBINFO_STATE_RUNNING 0
34    #define FBINFO_STATE_SUSPENDED        1
35    u32 state; /* 硬件状态，如挂起 */
36    void *fbcon_par;
37    void *par;
38 };
```

FBI 中记录了帧缓冲设备的全部信息，包括设备的设置参数、状态以及操作函数指针。每一个帧缓冲设备都必须对应一个 FBI。

2．fb_ops 结构体

FBI 的成员变量 fbops 为指向底层操作的函数的指针，这些函数是需要驱动程序开发人员编写的，其定义如代码清单 18.2 所示。

代码清单 18.2　fb_ops 结构体

```
1  struct fb_ops
2  {
3     struct module *owner;
4      /* 打开/释放 */
5     int(*fb_open)(struct fb_info *info, int user);
6     int(*fb_release)(struct fb_info *info, int user);
7
8     /* 对于非线性布局的/常规内存映射无法工作的帧缓冲设备需要 */
9     ssize_t(*fb_read)(struct file *file, char __user *buf, size_t
count,
10       loff_t*ppos);
11    ssize_t(*fb_write)(struct file *file, const char __user *buf,
size_t count,
12       loff_t *ppos);
13
14    /* 检测可变参数，并调整到支持的值*/
15        int(*fb_check_var)(struct  fb_var_screeninfo  *var,  struct
fb_info *info);
16
17   /* 根据info->var设置video模式 */
```

```
    int(*fb_set_par)(struct fb_info *info);

    /* 设置 color 寄存器 */
    int(*fb_setcolreg)(unsigned regno, unsigned red, unsigned green, unsigned
       blue, unsigned transp, struct fb_info *info);

    /* 批量设置 color 寄存器，设置颜色表 */
    int(*fb_setcmap)(struct fb_cmap *cmap, struct fb_info *info);

    /*显示空白 */
    int(*fb_blank)(int blank, struct fb_info *info);

    /* pan 显示 */
    int(*fb_pan_display)(struct fb_var_screeninfo *var, struct fb_info *info);

    /* 矩形填充 */
    void(*fb_fillrect)(struct fb_info *info, const struct fb_fillrect *rect);
    /* 数据复制 */
    void(*fb_copyarea)(struct fb_info *info, const struct fb_copyarea *region);
    /* 图形填充 */
    void(*fb_imageblit)(struct fb_info *info, const struct fb_image *image);

    /* 绘制光标 */
    int(*fb_cursor)(struct fb_info *info, struct fb_cursor *cursor);

    /* 旋转显示 */
    void(*fb_rotate)(struct fb_info *info, int angle);

    /* 等待 blit 空闲 (可选) */
    int(*fb_sync)(struct fb_info *info);

    /* fb 特定的 ioctl (可选) */
    int(*fb_ioctl)(struct fb_info *info, unsigned int cmd, unsigned long arg);

   /* 处理 32 位的 compat ioctl (可选) */
    int(*fb_compat_ioctl)(struct fb_info *info, unsigned cmd, unsigned long arg);

   /* fb 特定的 mmap */
   int(*fb_mmap)(struct fb_info *info, struct vm_area_struct *vma);

   /* 保存目前的硬件状态 */
   void(*fb_save_state)(struct fb_info *info);

   /* 恢复被保存的硬件状态 */
   void(*fb_restore_state)(struct fb_info *info);
};
```

fb_ops 的 fb_check_var()成员函数用于检查可以修改的屏幕参数并调整到合适的值，而 fb_set_par()则使得用户设置的屏幕参数在硬件上有效。

3．fb_var_screeninfo 和 fb_fix_screeninfo 结构体

FBI 的 fb_var_screeninfo 和 fb_fix_screeninfo 成员也是结构体，fb_var_screeninfo 记录用户可修改的显示控制器参数，包括屏幕分辨率和每个像素点的比特数。fb_var_screeninfo 中的 xres 定义屏幕一行有多少个点，yres 定义屏幕一列有多少个点，bits_per_pixel 定义每个点用多少个字节表示。而 fb_fix_screeninfo 中记录用户不能修改的显示控制器的参数，如屏幕缓冲区的物理地址、长度。当对帧缓冲设备进行映射操作的时候，就是从 fb_fix_screeninfo 中取得缓冲区物理地址的。上述数据成员都需要在驱动程序中初始化和设置。

fb_var_screeninfo 和 fb_fix_screeninfo 结构体的定义分别如代码清单 18.3 和代码清单 18.4 所示。

代码清单 18.3　fb_var_screeninfo 结构体

```
struct fb_var_screeninfo
{
  /* 可见解析度   */
  __u32 xres;
  __u32 yres;
  /* 虚拟解析度    */
  __u32 xres_virtual;
  __u32 yres_virtual;
  /* 虚拟到可见之间的偏移 */
  __u32 xoffset;
  __u32 yoffset;

  __u32 bits_per_pixel; /* 每像素位数，BPP */
  __u32 grayscale; /非 0 时指灰度 */

  /* fb 缓存的 R\G\B 位域 */
  struct fb_bitfield red;
  struct fb_bitfield green;
  struct fb_bitfield blue;
  struct fb_bitfield transp; /* 透明度 */

  __u32 nonstd; /* != 0 非标准像素格式 */

  __u32 activate;

  __u32 height; /*高度 */
  __u32 width; /*宽度 */

  __u32 accel_flags; /* 看 fb_info.flags */

  /* 定时: 除了 pixclock 本身外，其他的都以像素时钟为单位 */
  __u32 pixclock; /* 像素时钟（皮秒） */
```

```
33    __u32 left_margin; /* 行切换: 从同步到绘图之间的延迟    */
34    __u32 right_margin; /* 行切换: 从绘图到同步之间的延迟   */
35    __u32 upper_margin; /* 帧切换: 从同步到绘图之间的延迟  */
36    __u32 lower_margin; /* 帧切换: 从绘图到同步之间的延迟  */
37    __u32 hsync_len; /* 水平同步的长度     */
38    __u32 vsync_len; /* 垂直同步的长度     */
39    __u32 sync;
40    __u32 vmode;
41    __u32 rotate; /* 顺时钟旋转的角度 */
42    __u32 reserved[5]; /* 保留 */
43 };
```

代码清单 18.4　fb_fix_screeninfo 结构体

```
1  struct fb_fix_screeninfo
2  {
3      char id[16]; /* 字符串形式的标识符 */
4      unsigned long smem_start; /* fb 缓存的开始位置 */
5      __u32 smem_len; /* fb 缓存的长度 */
6      __u32 type; /* FB_TYPE_*       */
7      __u32 type_aux; /* 分界 */
8      __u32 visual; /* FB_VISUAL_* */
9      __u16 xpanstep; /* 如果没有硬件 panning，赋 0 */
10     __u16 ypanstep;
11     __u16 ywrapstep;/
12     __u32 line_length; /* 1 行的字节数 */
13     unsigned long mmio_start; /* 内存映射 I/O 的开始位置 */
14     __u32 mmio_len; /* 内存映射 I/O 的长度  */
15     __u32 accel;
16     __u16 reserved[3]; /* 保留*/
17 };
```

代码清单 18.4 中第 8 行的 visual 记录屏幕使用的色彩模式，在 Linux 系统中，支持的色彩模式包括如下几种。

- Monochrome（FB_VISUAL_MONO01、FB_VISUAL_MONO10），每个像素是黑或白。
- Pseudo color（FB_VISUAL_PSEUDOCOLOR、FB_VISUAL_STATIC_PSEUDOCOLOR），即伪彩色，采用索引颜色显示。
- True color（FB_VISUAL_TRUECOLOR），真彩色，分成红、绿、蓝三基色。
- Direct color（FB_VISUAL_DIRECTCOLOR），每个像素颜色也是有红、绿、蓝组成，不过每个颜色值是个索引，需要查表。
- Grayscale displays，灰度显示，红、绿、蓝的值都一样。

4．fb_bitfield 结构体

代码清单 18.3 第 17、18、19 行分别记录 R、G、B 的位域，fb_bitfield 结构体描述每一像素显示缓冲区的组织方式，包含位域偏移、位域长度和 MSB 指示，如代码清单 18.5 所示。

代码清单 18.5　fb_bitfield 结构体

```
1  struct fb_bitfield
2  {
3      __u32 offset; /* 位域偏移    */
4      __u32 length; /* 位域长度    */
```

```
    __u32 msb_right; /* MSB */
};
```

5．fb_cmap 结构体

fb_cmap 结构体记录设备无关的颜色表信息，用户空间可以通过 ioctl()的 FBIOGETCMAP 和 FBIOPUTCMAP 命令读取或设定颜色表。

代码清单 18.6　fb_cmap 结构体

```
struct fb_cmap
{
    __u32 start; /* 第1个元素入口    */
    __u32 len; /* 元素数量 */
    /* R、G、B、透明度*/
    __u16 *red;
    __u16 *green;
    __u16 *blue;
    __u16 *transp;
};
```

代码清单 18.7 所示为用户空间获取颜色表的例程，若 BPP 为 8 位，则颜色表长度为 256；若 BPP 为 4 位，则颜色表长度为 16；否则，颜色表长度为 0，这是因为，对于 BPP 大于等于 16 的情况，使用颜色表是不划算的。

代码清单 18.7　用户空间获取颜色表例程

```
// 读入颜色表
if ((vinfo.bits_per_pixel == 8) || (vinfo.bits_per_pixel == 4))
{
    screencols = (vinfo.bits_per_pixel == 8) ? 256 : 16;//颜色表大小
    int loopc;
    startcmap = new fb_cmap;
    startcmap->start = 0;
    startcmap->len = screencols;
    //分配颜色表的内存
    startcmap->red = (unsigned short int*)malloc(sizeof(unsigned short int)
        *screencols);
    startcmap->green = (unsigned short int*)malloc(sizeof(unsigned short int)
        *screencols);
    startcmap->blue = (unsigned short int*)malloc(sizeof(unsigned short int)
        *screencols);
    startcmap->transp = (unsigned short int*)malloc(sizeof(unsigned short int)
        *screencols);
    //获取颜色表
    ioctl(fd, FBIOGETCMAP, startcmap);
    for (loopc = 0; loopc < screencols; loopc++)
    {
            screenclut[loopc] = qRgb(startcmap->red[loopc] >> 8, startcmap
            ->green[loopc] >> 8, startcmap->blue[loopc] >> 8);
```

```
  }
}
else
{
    screencols = 0;
}
```

对于一个 256 色（BPP=8）的 800*600 分辨率的图像而言，若红、绿、蓝分别用一个字节描述，则需要 800*600*3=1440000Byte 的空间，而若使用颜色表，则只需要 800*600*1+256*3= 480768Byte 的空间。

6．文件操作结构体

作为一种字符设备，帧缓冲设备的文件操作结构体定义于/linux/drivers/vedio/fbmem.c 文件中，如代码清单 18.8 所示。

代码清单 18.8　帧缓冲设备文件操作结构体

```
static struct file_operations fb_fops =
{
    .owner = THIS_MODULE,
    .read = fb_read, //读函数
    .write = fb_write, //写函数
    .ioctl = fb_ioctl, //I/O 控制函数
    #ifdef CONFIG_COMPAT
       .compat_ioctl = fb_compat_ioctl,
    #endif
    .mmap = fb_mmap, //内存映射函数
    .open = fb_open,  //打开函数
    .release = fb_release, //释放函数
    #ifdef HAVE_ARCH_FB_UNMAPPED_AREA
       .get_unmapped_area = get_fb_unmapped_area,
    #endif
};
```

帧缓冲设备驱动的文件操作接口函数已经在 fbmem.c 中被统一实现，一般不需要由驱动工程师再编写。

7．注册与注销帧缓冲设备

Linux 内核提供了 register_framebuffer()和 unregister_framebuffer()函数分别注册和注销帧缓冲设备，这两个函数都接受 FBI 指针为参数，原型为：

```
int register_framebuffer(struct fb_info *fb_info);
int unregister_framebuffer(struct fb_info *fb_info);
```

对于 register_framebuffer()函数而言，如果注册的帧缓冲设备数超过了 FB_MAX（目前定义为 32），则函数返回-ENXIO，注册成功则返回 0。

18.3 Linux 帧缓冲设备驱动结构

图 18.3 所示为 Linux 帧缓冲设备驱动的主要结构，帧缓冲设备提供给用户空间的

file_operations 结构体由 fbmem.c 中的 file_operations 提供，而特定帧缓冲设备 fb_info 结构体的注册、注销以及其中成员的维护，尤其是 fb_ops 中成员函数的实现则由对应的 xxxfb.c 文件实现，fb_ops 中的成员函数最终会操作 LCD 控制器硬件寄存器。

图 18.3　帧缓冲设备驱动的程序结构

18.4 帧缓冲设备驱动的模块加载与卸载函数

在帧缓冲设备驱动的模块加载函数中，应该完成如下 4 个工作。

- 申请 FBI 结构体的内存空间，初始化 FBI 结构体中固定和可变的屏幕参数，即填充 FBI 中 fb_var_screeninfo var 和 struct fb_fix_screeninfo fix 成员。
- 根据具体 LCD 屏幕的特点，完成 LCD 控制器硬件的初始化。
- 申请帧缓冲设备的显示缓冲区空间。
- 注册帧缓冲设备。

在帧缓冲设备驱动的模块卸载函数中，应该完成相反的工作，包括释放 FBI 结构体内存、关闭 LCD、释放显示缓冲区以及注销帧缓冲设备。

由于 LCD 控制器经常被集成在 SoC 上作为一个独立的硬件模块而存在（成为 platform_device），因此，LCD 驱动中也经常包含平台驱动，这样，在帧缓冲设备驱动的模块加载函数中完成的工作只是注册平台驱动，而初始化 FBI 结构体中的固定和可变参数、LCD 控制器硬件的初始化、申请帧缓冲设备的显示缓冲区空间和注册帧缓冲设备的工作则移交到平台驱动的探测函数中完成。

同样地，在使用平台驱动的情况下，释放 FBI 结构体内存、关闭 LCD、释放显示缓冲区以及注销帧缓冲设备的工作也移交到平台驱动的移除函数中完成。

代码清单 18.9 所示为帧缓冲设备驱动的模块加载和卸载以及平台驱动的探测和移除函数中的模板。

代码清单 18.9　帧缓冲设备驱动的模块加载/卸载及平台驱动的探测/移除函数的模板

```
/* 平台驱动结构体 */
static struct platform_driver xxxfb_driver =
{
   .probe = xxxfb_probe,        //平台驱动探测函数
   .remove = xxxfb_remove,      //平台驱动移除函数
   .suspend = xxxfb_suspend,   .resume = xxxfb_resume, .driver =
   {
       .name = "xxx-lcd",  //驱动名
       .owner = THIS_MODULE,
   }
};

/* 平台驱动探测函数 */
static int __init xxxfb_probe(...)
{
    struct fb_info *info;

    /*分配 fb_info 结构体*/
    info = framebuffer_alloc(...);

    info->screen_base = framebuffer_virtual_memory;
    info->var = xxxfb_var;  //可变参数
   info->fix = xxxfb_fix;   //固定参数

   /*分配显示缓冲区*/
   alloc_dis_buffer(...);

   /*初始化 LCD 控制器*/
   lcd_init(...);

   /*检查可变参数*/
   xxxfb_check_var(&info->var, info);

   /*注册 fb_info*/
   if (register_framebuffer(info) < 0)
      return  - EINVAL;

    return 0;
}

/* 平台驱动移除函数 */
static void __exit xxxfb_remove(...)
{
   struct fb_info *info = dev_get_drv_data(dev);

   if (info)
   {
       unregister_framebuffer(info); //注销 fb_info
       dealloc_dis_buffer(...); //释放显示缓冲区
       framebuffer_release(info); //注销 fb_info
  }

```

```
53     return 0;
54 }
55
56 /* 帧缓冲设备驱动模块加载与卸载函数  */
57 int __devinit xxxfb_init(void)
58 {
59     return platform_driver_register(&xxxfb_driver); //注册平台设备
60 }
61
62 static void __exit xxxfb_cleanup(void)
63 {
64      platform_driver_unregister(&xxxfb_driver); //注销平台设备
65 }
66
67 module_init(xxxfb_init);
68 module_exit(xxxfb_cleanup);
```

上述代码中第 35 行、48 行成对出现的 register_framebuffer() 和 unregister_framebuffer()分别用于注册和注销帧缓冲设备。

18.5 帧缓冲设备显示缓冲区的申请与释放

在嵌入式系统中，一种常见的方式是直接在 RAM 空间中分配一段显示缓冲区，典型结构如图 18.4 所示。

图 18.4　在 RAM 中分配显示缓冲区

在分配显示缓冲区时一定要考虑 cache 的一致性问题，因为系统往往会通过 DMA 方式搬移显示数据。合适的方式是使用 dma_alloc_writecombine()函数分配一段 writecombining 区域，对应的 writecombining 区域由 dma_free_writecombine()函数释放，如代码清单 18.10 所示。

writecombining 意味着“写合并”，它允许写入的数据被合并，并临时保存在写合并缓冲区（WCB）中，直到进行一次 burst 传输而不再需要多次 single 传输。通过 dma_alloc_ writecombine()分配的显示缓冲区不会出现 cache 一致性问题，这一点类似于 dma_alloc_ coherent()。

代码清单 18.10　帧缓冲设备显示缓冲区的分配与释放

```
1  static int __init xxxfb_map_video_memory(struct xxxfb_info *fbi)
2  {
3      fbi->map_size = PAGE_ALIGN(fbi->fb->fix.smem_len + PAGE_SIZE);
4      fbi->map_cpu = dma_alloc_writecombine(fbi->dev, fbi->map_size,
5          &fbi->map_dma,GFP_KERNEL); //分配内存
6
7      fbi->map_size = fbi->fb->fix.smem_len; //显示缓冲区大小
8
9      if (fbi->map_cpu)
10     {
11         memset(fbi->map_cpu, 0xf0, fbi->map_size);
12
13         fbi->screen_dma = fbi->map_dma;
14         fbi->fb->screen_base = fbi->map_cpu;
15           fbi->fb->fix.smem_start = fbi->screen_dma; //赋值 fix 的
smem_start
16   }
17
18      return fbi->map_cpu ? 0 :  - ENOMEM;
19 }
20
21 static inline void xxxfb_unmap_video_memory(struct s3c2410fb_info
*fbi)
22 {
23   //释放显示缓冲区
24         dma_free_writecombine(fbi->dev,fbi->map_size,fbi->map_cpu,
fbi->map_dma);
25 }
```

18.6 帧缓冲设备的参数设置

18.6.1 定时参数

FBI 结构体可变参数 var 中的 left_margin、right_margin、upper_margin、lower_margin、hsync_len 和 vsync_len 直接查 LCD 的数据手册就可以得到，图 18.5 所示为某 LCD 数据手册中直接抓图获得的定时信息。

由图 18.5 可知对该 LCD 而言，var 中各参数的较合适值分别为：left_margin = 104，right_margin =

显示模式	参数	符号	条件	时间			单位
				最小值	典型值	最大值	
常规	Vertical cycle	VP		648	—	—	行
	Vertical data start	VDS	VS+VBP	4	—	—	行
	Vertical Sync Pulse width	VS		2	—	—	行
	Vertical front porch	VFP		4			行
	Vertical Back porch	VBP		2			行
	Vertical blanking period	VBL	VS+VBP+VFP	8	—	—	行
	Vertical active area	VDISP		640			行
	Horizontal cycle	HP		520			点
	Horizontal front porch	HFP		24	—	—	点
	Horizontal Sync Pulse width	HS		8	—	—	点
	Horizontal Back porch	HBP		8			点
	Horizontal Data start	HDS	HS+HBP	36	—	—	点
	Horizontal active area	HDISP		480	—	—	点
	Clock Frequency	fclk		20	TBD	TBD	MHz
		tclk		50	TBD	TBD	ns

图 18.5　LCD 数据手册中定时参数示例

8，upper_margin = 2，lower_margin = 2，hsync_len = 2，vsync_len = 2。在 QVGA 模式下也可类似得到。

18.6.2　像素时钟

FBI 可变参数 var 中的 pixclock 意味着像素时钟，例如，如果为 28.37516 MHz，那么画 1 个像素需要 35242 ps（皮秒）：

```
1/(28.37516E6 Hz) = 35.242E-9 s
```

如果屏幕的分辨率是 640×480，显示一行需要的时间是：

```
640*35.242E-9 s = 22.555E-6 s
```

每条扫描线是 640，但是水平回扫和水平同步也需要时间，假设水平回扫和同步需要 272 个像素时钟，因此，画一条扫描线完整的时间是：

```
(640+272)*35.242E-9 s = 32.141E-6 s
```

可以计算出水平扫描率大约是 31kHz：

```
1/(32.141E-6 s) = 31.113E3 Hz
```

完整的屏幕有 480 线，但是垂直回扫和垂直同步也需要时间，假设垂直回扫和垂直同步需要 49 个象素时钟，因此，画一个完整的屏幕的时间是：

```
(480+49)*32.141E-6 s = 17.002E-3 s
```

可以计算出垂直扫描率大约是 59kHz：

```
1/(17.002E-3 s) = 58.815 Hz
```

这意味着屏幕数据每秒钟大约刷新 59 次。

18.6.3　颜色位域

FBI 可变参数 var 中的 red、green 和 blue 位域的设置直接由显示缓冲区与显示点的对应关系决定，例如，对于 RGB565 模式，查表 18.4，red 占据 5 位，偏移为 11 位；green 占据 6 位，偏移为 5 位；blue 占据 5 位，偏移为 0 位，即：

```
fbinfo->var.red.offset = 11;
```

```
    fbinfo->var.green.offset = 5;
    fbinfo->var.blue.offset = 0;
    fbinfo->var.transp.offset = 0;
    fbinfo->var.red.length = 5;
    fbinfo->var.green.length = 6;
    fbinfo->var.blue.length = 5;
```

18.6.4 固定参数

FBI 固定参数 fix 中的 smem_start 指示帧缓冲设备显示缓冲区的首地址，smem_len 为帧缓冲设备显示缓冲区的大小，计算公式为：

```
    smem_len = max_xres * max_yres * max_bpp
```

即：帧缓冲设备显示缓冲区的大小 = 最大的 *x* 解析度 * 最大的 *y* 解析度 * 最大的 BPP。

18.7 帧缓冲设备驱动的 fb_ops 成员函数

FBI 中的 fp_ops 是使得帧缓冲设备工作所需函数的集合，它们最终与 LCD 控制器硬件打交道。

fb_check_var()用于调整可变参数，并修正为硬件所支持的值；fb_set_par()则根据屏幕参数设置具体读写 LCD 控制器的寄存器以使得 LCD 控制器进入相应的工作状态。

对于 fb_ops 中的 fb_fillrect()、fb_copyarea()和 fb_imageblit()成员函数，通常直接使用对应的通用的 cfb_fillrect()、cfb_copyarea()和 cfb_imageblit()函数即可。cfb_fillrect()函数定义在 drivers/video/cfbfillrect.c 文件中，cfb_copyarea()定义在 drivers/video/cfbcopyarea.c 文件中，cfb_imageblit()定义在 drivers/video/cfbimgblt.c 文件中。

fb_ops 中的 fb_setcolreg()成员函数实现伪颜色表（针对 FB_VISUAL_TRUECOLOR、FB_ VISUAL_DIRECTCOLOR 模式）和颜色表的填充，其模板如代码清单 18.11 所示。

代码清单 18.11 fb_setcolreg()函数模板

```
static int xxxfb_setcolreg(unsigned regno, unsigned red, unsigned green,
    unsigned blue, unsigned transp, struct fb_info *info)
{
    struct xxxfb_info *fbi = info->par;
    unsigned int val;

    switch (fbi->fb->fix.visual)
```

```
8     {
9       case FB_VISUAL_TRUECOLOR:
10        /* 真彩色，设置伪颜色表 */
11        if (regno < 16)
12        {
13           u32 *pal = fbi->fb->pseudo_palette;
14
15         val = chan_to_field(red, &fbi->fb->var.red);
16         val |= chan_to_field(green, &fbi->fb->var.green);
17         val |= chan_to_field(blue, &fbi->fb->var.blue);
18
19         pal[regno] = val;
20        }
21        break;
22
23    case FB_VISUAL_PSEUDOCOLOR:
24        if (regno < 256)
25        {
26            /* RGB565 模式 */
27           val = ((red >> 0) &0xf800);
28           val |= ((green >> 5) &0x07e0);
29           val |= ((blue >> 11) &0x001f);
30
31           writel(val, XXX_TFTPAL(regno));
32           schedule_palette_update(fbi, regno, val);
33        }
34        break;
35      ...
36    }
37
38    return 0;
39 }
```

上述代码第 11 行对 regno < 16 的判断意味着伪颜色表只有 16 个成员，实际上，它们对应 16 种控制台颜色，logo 显示也会使用伪颜色表。

18.8 LCD 设备驱动的读写、mmap 和 ioctl 函数

虽然帧缓冲设备的 file_operations 中的成员函数，即文件操作函数已经由内核在 fbmem.c 文件中实现，一般不再需要驱动工程师修改，但分析这些函数对于巩固字符设备驱动的知识以及加深对帧缓冲设备驱动的理解是大有裨益的。

代码清单 18.12 所示为 LCD 设备驱动的文件操作读写函数的源代码，从代码结构及习惯而言，与本书第二篇所讲解的字符设备驱动完全一致。

代码清单 18.12　帧缓冲设备驱动的读写函数

```
static ssize_t fb_read(struct file *file, char __user *buf, size_t count,
    loff_t *ppos)
{
    unsigned long p =  *ppos;
    struct inode *inode = file->f_dentry->d_inode;
    int fbidx = iminor(inode);
    struct fb_info *info = registered_fb[fbidx]; //获得FBI
    u32 *buffer,  *dst;
    u32 __iomem *src;
    int c, i, cnt = 0, err = 0;
    unsigned long total_size;

    if (!info || !info->screen_base)
        return  - ENODEV;

    if (info->state != FBINFO_STATE_RUNNING)
        return  - EPERM;

    if (info->fbops->fb_read)  //如果fb_ops中定义了特定的读函数
        return info->fbops->fb_read(file, buf, count, ppos);
    /*获得显示缓冲区总的大小*/
    total_size = info->screen_size;

    if (total_size == 0)
       total_size = info->fix.smem_len;

    if (p >= total_size)
       return 0;
    /*获得有效的读长度*/
    if (count >= total_size)
       count = total_size;

    if (count + p > total_size)
       count = total_size - p;
    /*分配用于临时存放显示缓冲区数据的buffer*/
      buffer = kmalloc((count > PAGE_SIZE) ? PAGE_SIZE : count, GFP_KERNEL);
    if (!buffer)
       return  - ENOMEM;

    src = (u32 __iomem*)(info->screen_base + p);  //获得源地址

    if (info->fbops->fb_sync)
        info->fbops->fb_sync(info);
```

```

    while (count)
    { /*读取显示缓冲区中的数据并复制到分配的buffer*/
      c = (count > PAGE_SIZE) ? PAGE_SIZE : count;
      dst = buffer;
      for (i = c >> 2; i--;)
         *dst++ = fb_readl(src++);
      if (c &3)
      {
         u8 *dst8 = (u8*)dst;
         u8 __iomem *src8 = (u8 __iomem*)src;

         for (i = c &3; i--;)
             *dst8++ = fb_readb(src8++);

         src = (u32 __iomem*)src8;
      }

      if (copy_to_user(buf, buffer, c))  //复制到用户空间
      {
          err =  - EFAULT;
          break;
     }
     *ppos += c;
     buf += c;
     cnt += c;
     count -= c;
    }

    kfree(buffer);

    return (err) ? err : cnt;
}

static ssize_t fb_write(struct file *file, const char __user *buf, size_t count,
    loff_t *ppos)
{
    unsigned long p =  *ppos;
    struct inode *inode = file->f_dentry->d_inode;
    int fbidx = iminor(inode);
    struct fb_info *info = registered_fb[fbidx];
    u32 *buffer,  *src;
    u32 __iomem *dst;
    int c, i, cnt = 0, err = 0;
    unsigned long total_size;

    if (!info || !info->screen_base)
        return  - ENODEV;

    if (info->state != FBINFO_STATE_RUNNING)
        return  - EPERM;
```

```
    if (info->fbops->fb_write) //如果 fb_ops 中定义了特定的写函数
       return info->fbops->fb_write(file, buf, count, ppos);
    /*获得显示缓冲区总的大小*/
    total_size = info->screen_size;

    if (total_size == 0)
        total_size = info->fix.smem_len;

    if (p > total_size)
       return 0;
    /*获得有效的写长度*/
    if (count >= total_size)
        count = total_size;

    if (count + p > total_size)
        count = total_size - p;
    /*分配用于存放用户空间传过来的显示缓冲区数据的 buffer*/
      buffer = kmalloc((count > PAGE_SIZE) ? PAGE_SIZE : count, GFP_KERNEL);
    if (!buffer)
        return  - ENOMEM;

    dst = (u32 __iomem*)(info->screen_base + p); //要写的显示缓冲区基地址

    if (info->fbops->fb_sync)
        info->fbops->fb_sync(info);

   while (count)
   { /*读取用户空间数据并复制到显示缓冲区*/
      c = (count > PAGE_SIZE) ? PAGE_SIZE : count;
      src = buffer;

      if (copy_from_user(src, buf, c))
      {
         err =  - EFAULT;
         break;
      }

      for (i = c >> 2; i--;)
          fb_writel(*src++, dst++);

      if (c &3)
      {
         u8 *src8 = (u8*)src;
         u8 __iomem *dst8 = (u8 __iomem*)dst;

       for (i = c &3; i--;)
           fb_writeb(*src8++, dst8++);

       dst = (u32 __iomem*)dst8;
    }

    *ppos += c;
```

```
    buf += c;
    cnt += c;
    count -= c;
  }

  kfree(buffer);

  return (err) ? err : cnt;
}
```

file_operations 中的 mmap()函数非常关键，它将显示缓冲区映射到用户空间，从而使得用户空间可以直接操作显示缓冲区而省去一次用户空间到内核空间的内存复制过程，提高效率，其源代码如代码清单 18.13 所示。

代码清单 18.13　帧缓冲设备驱动的 mmap 函数

```
static int fb_mmap(struct file *file, struct vm_area_struct *vma)
{
    int fbidx = iminor(file->f_dentry->d_inode);
    struct fb_info *info = registered_fb[fbidx];
    struct fb_ops *fb = info->fbops;
    unsigned long off;

    if (vma->vm_pgoff > (~0UL >> PAGE_SHIFT))
        return  - EINVAL;
   off = vma->vm_pgoff << PAGE_SHIFT;
    if (!fb)
        return  - ENODEV;
    if (fb->fb_mmap)  //FBI 中实现了 mmap，则调用 FBI 的 mmap
    {
       int res;
       lock_kernel();
       res = fb->fb_mmap(info, vma);
       unlock_kernel();
       return res;
  }

  /* !sparc32... */
  lock_kernel();

  /* 映射帧缓冲设备的显示缓冲区 */
  start = info->fix.smem_start; //开始地址
  len = PAGE_ALIGN((start &~PAGE_MASK) + info->fix.smem_len); //长度
  if (off >= len)
  {
      /* 内存映射的 I/O */
      off -= len;
      if (info->var.accel_flags)
      {
          unlock_kernel();
          return  - EINVAL;
```

```
    36      }
    37      start = info->fix.mmio_start;
    38      len = PAGE_ALIGN((start &~PAGE_MASK) + info->fix.mmio_len);
    39  }
    40  unlock_kernel();
    41  start &= PAGE_MASK;
    42  if ((vma->vm_end - vma->vm_start + off) > len)
    43      return  - EINVAL;
    44  off += start;
    45  vma->vm_pgoff = off >> PAGE_SHIFT;
    46  /* 这是 1 个 I/O 映射 */
    47  vma->vm_flags |= VM_I/O | VM_RESERVED;
    48  vma->vm_page_prot = pgprot_writecombine(vma->vm_page_prot);
    49
    50  if (io_remap_pfn_range(vma, vma->vm_start, off >> PAGE_SHIFT,
    51      vma->vm_end -vma->vm_start, vma->vm_page_prot)) //映射
    52    return  - EAGAIN;
    53
    54  return 0;
    55 }
```

fb_ioctl() 函数最终实现对用户 I/O 控制命令的执行，这些命令包括 FBIOGET_VSCREENINFO（获得可变的屏幕参数）、FBIOPUT_VSCREENINFO（设置可变的屏幕参数）、FBIOGET _FSCREENINFO（获得固定的屏幕参数设置，注意，固定的屏幕参数不能由用户设置）、FBIOPUTCMAP（设置颜色表）、FBIOGETCMAP（获得颜色表）等。代码清单 18.14 所示为帧缓冲设备 ioctl()函数的源代码。

代码清单 18.14 帧缓冲设备驱动的 ioctl 函数

```
    1  static int fb_ioctl(struct inode *inode, struct file *file, unsigned
int cmd,
    2     unsigned long arg)
    3  {
    4     int fbidx = iminor(inode);
    5     struct fb_info *info = registered_fb[fbidx];
    6     struct fb_ops *fb = info->fbops;
    7     struct fb_var_screeninfo var;
    8     struct fb_fix_screeninfo fix;
    9     struct fb_con2fbmap con2fb;
    10    struct fb_cmap_user cmap;
    11    struct fb_event event;
    12    void __user *argp = (void __user*)arg;
    13    int i;
    14
    15    if (!fb)
    16       return  - ENODEV;
    17    switch (cmd)
    18    {
    19      case FBIOGET_VSCREENINFO:// 获得可变的屏幕参数
    20          return copy_to_user(argp, &info->var, sizeof(var)) ?  -
EFAULT: 0;
    21      case FBIOPUT_VSCREENINFO: //设置可变的屏幕参数
    22        if (copy_from_user(&var, argp, sizeof(var)))
    23            return  - EFAULT;
    24        acquire_console_sem();
```

```
        info->flags |= FBINFO_MISC_USEREVENT;
        i = fb_set_var(info, &var);
        info->flags &= ~FBINFO_MISC_USEREVENT;
        release_console_sem();
        if (i)
          return i;
        if (copy_to_user(argp, &var, sizeof(var)))
          return  - EFAULT;
        return 0;
    case FBIOGET_FSCREENINFO: //获得固定的屏幕参数设置
         return copy_to_user(argp, &info->fix, sizeof(fix)) ?  -EFAULT: 0;
    case FBIOPUTCMAP:  //设置颜色表
        if (copy_from_user(&cmap, argp, sizeof(cmap)))
           return  - EFAULT;
        return (fb_set_user_cmap(&cmap, info));
    case FBIOGETCMAP:  //获得颜色表
        if (copy_from_user(&cmap, argp, sizeof(cmap)))
           return  - EFAULT;
        return fb_cmap_to_user(&info->cmap, &cmap);
    case FBIOPAN_DISPLAY:
        if (copy_from_user(&var, argp, sizeof(var)))
           return  - EFAULT;
        acquire_console_sem();
        i = fb_pan_display(info, &var);
        release_console_sem();
        if (i)
           return i;
        if (copy_to_user(argp, &var, sizeof(var)))
           return  - EFAULT;
        return 0;
    case FBIO_CURSOR:
        return  - EINVAL;
    case FBIOGET_CON2FBMAP:
        if (copy_from_user(&con2fb, argp, sizeof(con2fb)))
           return  - EFAULT;
        if (con2fb.console < 1 || con2fb.console > MAX_NR_CONSOLES)
           return  - EINVAL;
        con2fb.framebuffer =  - 1;
        event.info = info;
        event.data = &con2fb;
                            notifier_call_chain(&fb_notifier_list, FB_EVENT_GET_CONSOLE_MAP, &event);
         return copy_to_user(argp, &con2fb, sizeof(con2fb)) ?  -EFAULT: 0;
    case FBIOPUT_CON2FBMAP:
        if (copy_from_user(&con2fb, argp, sizeof(con2fb)))
          return  - EFAULT;
        if (con2fb.console < 0 || con2fb.console > MAX_NR_CONSOLES)
          return  - EINVAL;
       if (con2fb.framebuffer < 0 || con2fb.framebuffer >= FB_MAX)
          return  - EINVAL;
```

```
        #ifdef CONFIG_KMOD
           if (!registered_fb[con2fb.framebuffer])
              try_to_load(con2fb.framebuffer);
        #endif /* CONFIG_KMOD */
        if (!registered_fb[con2fb.framebuffer])
           return  - EINVAL;
        event.info = info;
        event.data = &con2fb;
        return notifier_call_chain(&fb_notifier_list,
           FB_EVENT_SET_CONSOLE_MAP, &event);
     case FBIOBLANK:
        acquire_console_sem();
        info->flags |= FBINFO_MISC_USEREVENT;
        i = fb_blank(info, arg);
        info->flags &= ~FBINFO_MISC_USEREVENT;
        release_console_sem();
        return i;
     default:
        if (fb->fb_ioctl == NULL)
           return  - EINVAL;
        return fb->fb_ioctl(info, cmd, arg);
  }
}
```

18.9 帧缓冲设备的用户空间访问

通过/dev/fbns，应用程序可进行的针对帧缓冲设备的操作主要有如下几种。

- 读/写 dev/fbn：相当于读/写屏幕缓冲区。例如用 cp /dev/fb0 tmp 命令可将当前屏幕的内容复制到一个文件中，而命令 cp tmp > /dev/fb0 则将图形文件 tmp 显示在屏幕上。
- 映射操作：对于帧缓冲设备，可通过 mmap()映射操作将屏幕缓冲区的物理地址映射到用户空间的一段虚拟地址中，之后用户就可以通过读/写这段虚拟地址访问屏幕缓冲区，在屏幕上绘图了。而且若干个进程可以映射到同一个显示缓冲区。实际上，使用帧缓冲设备的应用程序都是通过映射操作来显示图形的。
- I/O 控制：对于帧缓冲设备，对设备文件的 ioctl()操作可读取/设置显示设备及屏幕的参数，如分辨率、显示颜色数、屏幕大小等。

图 18.6 用户空间
帧缓冲设备流程

如图 18.6 所示，在应用程序中，操作/dev/fbn 的一般步骤如下。

（1）打开/dev/fbn 设备文件。

（2）用 ioctl()操作取得当前显示屏幕的参数，如屏幕分辨率、每个像素点的比特数和偏移。根据屏幕参数可计算屏幕缓冲区的大小。

（3）将屏幕缓冲区映射到用户空间。

（4）映射后就可以直接读/写屏幕缓冲区，进行绘图和图片显示了。

代码清单 18.15 所示为一段用户空间访问帧缓冲设备显示缓冲区的范例，包含打开和关闭帧缓冲设备、得到和设置可变参数、得到固定参数、生成与 BPP 对应的帧缓冲数据及填充显示缓冲区。

华清远见

代码清单 18.15　用户空间访问帧缓冲设备显示缓冲区范例

```
1   /*打开帧缓冲设备*/
2   int openFB(const char *name)
3   {
4      int fh;
5      char *dev;
6
7      if (name == NULL)
8      {
9         dev = getenv("FRAMEBUFFER");
10       if (dev)
11          name = dev;
12       else
13          name = DEFAULT_FRAMEBUFFER;
14     }
15
16     if ((fh = open(name, O_WRONLY)) ==  - 1)
17     {
18       fprintf(stderr, "open %s: %s\n", name, strerror(errno));
19       exit(1);
20     }
21     return fh;
22  }
23
24  /*关闭帧缓冲设备*/
25  void closeFB(int fh)
26  {
27     close(fh);
28  }
29
30  /*得到屏幕可变参数*/
31  void getVarScreenInfo(int fh, struct fb_var_screeninfo *var)
32  {
33    if (ioctl(fh, FBIOGET_VSCREENINFO, var))
34    {
35              fprintf(stderr,  "ioctl  FBIOGET_VSCREENINFO:  %s\n",
strerror(errno));
36       exit(1);
37    }
38  }
39
40  /*设置屏幕可变参数*/
41  void setVarScreenInfo(int fh, struct fb_var_screeninfo *var)
42  {
43     if (ioctl(fh, FBIOPUT_VSCREENINFO, var))
44     {
45              fprintf(stderr,  "ioctl  FBIOPUT_VSCREENINFO:  %s\n",
strerror(errno));
46        exit(1);
47    }
48  }
49
50  /*得到屏幕固定参数*/
51  void getFixScreenInfo(int fh, struct fb_fix_screeninfo *fix)
52  {
```

```
    if (ioctl(fh, FBIOGET_FSCREENINFO, fix))
    {
            fprintf(stderr, "ioctl FBIOGET_FSCREENINFO: %s\n", strerror(errno));
        exit(1);
    }
}

/*构造颜色表*/
void make332map(struct fb_cmap *map)
{
    int rs, gs, bs, i;
    int r = 8, g = 8, b = 4;

    map->red = red;
    map->green = green;
    map->blue = blue;

    rs = 256 / (r - 1);
    gs = 256 / (g - 1);
    bs = 256 / (b - 1);

    for (i = 0; i < 256; i++)
    {
        map->red[i] = (rs *((i / (g *b)) % r)) *255;
        map->green[i] = (gs *((i / b) % g)) *255;
        map->blue[i] = (bs *((i) % b)) *255;
    }
}

/*设置颜色表*/
void set8map(int fh, struct fb_cmap *map)
{
    if (ioctl(fh, FBIOPUTCMAP, map) < 0)
    {
        fprintf(stderr, "Error putting colormap");
        exit(1);
    }
}

/*构造并设置颜色表*/
void set332map(int fh)
{
    make332map(&map332);
    set8map(fh, &map332);
}

/*获得颜色表*/
void get8map(int fh, struct fb_cmap *map)
{
    if (ioctl(fh, FBIOGETCMAP, map) < 0)
    {
        fprintf(stderr, "Error getting colormap");
```

```
        exit(1);
  }
}
/*组成 BPP=16 时的缓冲数据*/
inline unsigned short make16color(unsigned char r, unsigned char g, unsigned char b)
{
        return ((((r >> 3) & 31) << 11) |
                        (((g >> 2) & 63) << 5)  |
                        ((b >> 3) & 31) );
}

/*把 RGB 数据按照屏幕的 BPP 调整*/
void *convertRGB2FB(int fh, unsigned char *rgbbuff, unsigned long count, int
    bpp, int *cpp)
{
    unsigned long i;
    void *fbbuff = NULL;
    unsigned char *c_fbbuff;
    unsigned short *s_fbbuff;
    unsigned int *i_fbbuff;

    switch (bpp)
    {
        case 8:
          //...
        case 15:
         //  ...
        case 16:
           *cpp = 2;
                        s_fbbuff  =  (unsigned  short*)malloc(count *sizeof(unsigned short));
            for (i = 0; i < count; i++)
                s_fbbuff[i] = make16color(rgbbuff[i *3], rgbbuff[i *3+1], rgbbuff
                [i *3+ 2]);
            fbbuff = (void*)s_fbbuff;
            break;
         case 24:
         case 32:
           //  ...
         default:
             fprintf(stderr, "Unsupported video mode! You've got: %dbpp\n", bpp);
            exit(1);
    }
    return fbbuff;
}

/*写显示缓冲区*/
void blit2FB(int fh, void *fbbuff, unsigned int pic_xs, unsigned int pic_ys,
     unsigned int scr_xs, unsigned int scr_ys, unsigned int xp, unsigned int yp,
     unsigned int xoffs, unsigned int yoffs, int cpp)
```

```
{
    int i, xc, yc;
    unsigned char *cp;
    unsigned short *sp;
    unsigned int *ip;
     cp = (unsigned char*)sp = (unsigned short*)ip = (unsigned int*)fbbuff;

    xc = (pic_xs > scr_xs) ? scr_xs : pic_xs;
    yc = (pic_ys > scr_ys) ? scr_ys : pic_ys;

    switch (cpp)
    {
      case 1:
        ...
      case 2:  //BPP=16
        for (i = 0; i < yc; i++)
        {
                lseek(fh, ((i + yoffs) *scr_xs + xoffs) *cpp, SEEK_SET);
            write(fh, sp + (i + yp) *pic_xs + xp, xc *cpp);
        }
        break;
      case 4:
        ...
  }
}


/*帧缓冲设备显示*/
void fb_display(unsigned char *rgbbuff, int x_size, int y_size, int x_pan, int
   y_pan, int x_offs, int y_offs)
{
   struct fb_var_screeninfo var;
   unsigned short *fbbuff = NULL;
   int fh =  - 1, bp = 0;

    /* 打开帧缓冲设备 */
    fh = openFB(NULL);

    /* 获得可变参数 */
    getVarScreenInfo(fh, &var);

    /*校正 panning */
    if (x_pan > x_size - var.xres)
        x_pan = 0;
    if (y_pan > y_size - var.yres)
        y_pan = 0;
    /* 校正 offset */
    if (x_offs + x_size > var.xres)
        x_offs = 0;
    if (y_offs + y_size > var.yres)
```

```
    204         y_offs = 0;
    205
    206     /* 将 RGB 数据转换为与 var 位域对应的数据并填充到显示缓冲区 */
    207         fbbuff = convertRGB2FB(fh, rgbbuff, x_size *y_size,
var.bits_per_pixel, &bp);
    208     blit2FB(fh, fbbuff, x_size, y_size, var.xres, var.yres, x_pan,
y_pan, x_offs,
    209         y_offs, bp);
    210     free(fbbuff);
    211
    212     /* 关闭帧缓冲设备 */
    213     closeFB(fh);
    214 }
```

18.10 Linux 图形用户界面

18.10.1 Qt-X11/QtEmbedded/Qtopia

Qt 是 Trolltech（奇趣）公司所开发的一个跨平台 FrameWork 环境，它采用类似 C++的语法，在 Microsoft Windows 95/98/2000、Microsoft Windows NT、MacOS X、Linux、Solaris、HP-UX、Tru64 (Digital UNIX)、Irix、FreeBSD、BSD/OS、SCO、AIX 等许多平台上都可执行。

Trolltech 也针对嵌入式环境推出了 Qt/Embedded 产品。与桌面版本不同，Qt/Embedded 已经直接取代掉 X Server 及 X Library 等角色，所有的功能全部整合在一起。Qt/Embedded 是一个专门为小型设备提供图形用户界面的应用框架和窗口系统，它提供了丰富的窗口小部件（Widgets），并且还支持窗口部件的定制，因此它可以为用户提供漂亮的图形界面，许多基于 Qt 的 X Window 程序可以非常方便地移植到 Qt/Embedded 版本上。

Qt/Embedded 以原始 Qt 为基础，做了许多针对嵌入式环境的调整。Qt/Embedded 通过 Qt 的 API 与 Linux 的 I/O 直接交互，同 Qt/X11 相比，Qt/Embedded 不需要一个 X 服务器或是 Xlib 库，它在底层抛弃了 Xlib，而是采用帧缓冲作为底层图形接口，Qt/Embedded 的应用程序可以直接写显示缓冲区，无须借助繁琐的 Xlib/Server 系统，如图 18.7 所示。

图 18.7 Qt/Embedded 与 Qt 的区

Qtopia 是建立在 Qt/Embedded 上的一种开放源代码窗口系统，它与实际的产品相似，专门针对 PDA、SmartPhone 这类运行嵌入式 Linux 的移动计算设备和手持设备而开发的。Trolltech 还发布了一款供应用开发人员使用的 Linux 手机“Qtopia Greenphone”。

在宿主机上可通过 qvfb（虚拟帧缓冲）来模拟帧缓冲。qvfb 是 X 窗口用来运行和测试 Qtopia 应用程序的系统程序，它使用了共享存储区域（虚拟的帧缓冲）来模拟帧缓冲并且在一个窗口中模拟一个应用来显示帧缓冲，允许我

嵌入式学院 华清远见旗下品牌

们在桌面及其上开发 Qt 嵌入式程序。显示的区域被周期性地改变和更新，通过指定显示设备的宽度和颜色深度，虚拟出来的缓冲帧和物理的显示设备在每个像素上保持一致，从而方便程序的调试。

信号（Signal）和插槽（Slot）是 Qt 中一种用于对象间通信的调用机制，不同于传统的函数回调方式，信号和插槽是 Qt 中非常有特色的地方，是 Qt 编程区别于其他编程的标志。信号和插槽不是标准 C++功能，C++编译器不能理解这些语句，必须经过特殊的工具对象编辑器 MOC（Meta Object Compiler）将源代码中创建信号和插槽的语句翻译成 C++编译器能够理解的代码。

Qt 的窗口在事件发生后会激发信号。例如，一个按钮被点击时会激发一个“clicked”信号。程序员通过建立一个函数（称做一个插槽），然后调用 connect()函数把这个插槽和一个信号连接起来，这样就完成了一个事件和响应代码的连接。信号与插槽机制并不要求类之间互相知道细节，这样就可以相对容易的开发出代码可高重用的类。

例如，如果一个退出按钮的 clicked()信号被连接到了一个应用的退出函数 quit()插槽。那么一个用户点击退出键将使应用程序终止运行，完成上述连接过程的代码如下。

```
connect( button, SIGNAL(clicked()), qApp, SLOT(quit()) );
```

代码清单 18.16 的应用程序创建一个 hello 窗口，该窗口显示一个动态字符串“Hello，World”，程序中添加了一个 Qtimer 定时器实例，以周期性刷新屏幕，从而得到动画的效果。

代码清单 18.16　Qt/Embedded 应用程序范例

```
/*****************************************************
 ** 以下是 hello.h 的代码
 *****************************************************/
#ifndef HELLO_H
   #define HELLO_H
   #include <qvariant.h>
   #include <qwidget.h>
   class QVBoxLayout;
   class QHBoxLayout;
   class QGridLayout;
   class Hello: public QWidget
   {
      Q_OBJECT public:
         Hello(QWidget *parent = 0, const char *name = 0, WFlags fl = 0);
         ~Hello();
         //以下是手动添加的代码
         signals: void clicked();
       protected:
          void mouseReleaseEvent(QMouseEvent*);
          void paintEvent(QPaintEvent*);
          private slots: void animate();
```

```
    22         private:
    23            QString t;
    24            int b;
    25      };
    26   #endif // HELLO_H
    27
    28  /*****************************************************************
    29   ** 以下是 hello.cpp 源代码
    30   *****************************************************************/
    31  #include "hello.h"
    32  #include <qlayout.h>
    33  #include <qvariant.h>
    34  #include <qtooltip.h>
    35  #include <qwhatsthis.h>
    36  #include <qpushbutton.h>
    37  #include <qtimer.h>
    38  #include <qpainter.h>
    39  #include <qpixmap.h>
    40  /* 构造一个Hello窗口 */
    41   Hello::Hello(QWidget  *parent,  const  char  *name,  WFlags  fl):
QWidget(parent,
    42    name, fl)
    43  {
    44     if (!name)
    45        setName("Hello");
    46     resize(240, 320);
    47     setMinimumSize(QSize(240, 320));
    48     setMaximumSize(QSize(240, 320));
    49     setSizeIncrement(QSize(240, 320));
    50     setBaseSize(QSize(240, 320));
    51     QPalette pal;
    52     QColorGroup cg;
    53     cg.setColor(QColorGroup::Foreground, black);
    54     cg.setColor(QColorGroup::Button, QColor(192, 192, 192));
    55     cg.setColor(QColorGroup::Light, white);
    56     cg.setColor(QColorGroup::Midlight, QColor(223, 223, 223));
    57     cg.setColor(QColorGroup::Dark, QColor(96, 96, 96));
    58     cg.setColor(QColorGroup::Mid, QColor(128, 128, 128));
    59     cg.setColor(QColorGroup::Text, black);
    60     cg.setColor(QColorGroup::BrightText, white);
    61     cg.setColor(QColorGroup::ButtonText, black);
    62     cg.setColor(QColorGroup::Base, white);
    63     cg.setColor(QColorGroup::Background, white);
    64     cg.setColor(QColorGroup::Shadow, black);
    65     cg.setColor(QColorGroup::Highlight, black);
    66     cg.setColor(QColorGroup::HighlightedText, white);
    67     pal.setActive(cg);
    68     cg.setColor(QColorGroup::Foreground, black);
    69     cg.setColor(QColorGroup::Button, QColor(192, 192, 192));
    70     cg.setColor(QColorGroup::Light, white);
    71     cg.setColor(QColorGroup::Midlight, QColor(220, 220, 220));
    72     cg.setColor(QColorGroup::Dark, QColor(96, 96, 96));
    73     cg.setColor(QColorGroup::Mid, QColor(128, 128, 128));
    74     cg.setColor(QColorGroup::Text, black);
    75     cg.setColor(QColorGroup::BrightText, white);
```

```
    cg.setColor(QColorGroup::ButtonText, black);
    cg.setColor(QColorGroup::Base, white);
    cg.setColor(QColorGroup::Background, white);
    cg.setColor(QColorGroup::Shadow, black);
    cg.setColor(QColorGroup::Highlight, black);
    cg.setColor(QColorGroup::HighlightedText, white);
    pal.setInactive(cg);
    cg.setColor(QColorGroup::Foreground, QColor(128, 128, 128));
    cg.setColor(QColorGroup::Button, QColor(192, 192, 192));
    cg.setColor(QColorGroup::Light, white);
    cg.setColor(QColorGroup::Midlight, QColor(220, 220, 220));
    cg.setColor(QColorGroup::Dark, QColor(96, 96, 96));
    cg.setColor(QColorGroup::Mid, QColor(128, 128, 128));
    cg.setColor(QColorGroup::Text, black);
    cg.setColor(QColorGroup::BrightText, white);
    cg.setColor(QColorGroup::ButtonText, QColor(128, 128, 128));
    cg.setColor(QColorGroup::Base, white);
    cg.setColor(QColorGroup::Background, white);
    cg.setColor(QColorGroup::Shadow, black);
    cg.setColor(QColorGroup::Highlight, black);
    cg.setColor(QColorGroup::HighlightedText, white);
    pal.setDisabled(cg);
    setPalette(pal);
    QFont f(font());
    f.setFamily("adobe-helvetica");
    f.setPointSize(29);
    f.setBold(TRUE);
    setFont(f);
    setCaption(tr(""));
    t = "Hello,World";
    b = 0;
    QTimer *timer = new QTimer(this);   //创建定时器
    connect(timer, SIGNAL(timeout()), SLOT(animate()));  //连接信号和插槽
    timer->start(40);
}

/* 销毁对象，释放任何被分配的资源 */
Hello::~Hello(){}

/* 每次定时器到期后调用插槽 */
void Hello::animate()
{
    b = (b + 1) &15;
    repaint(FALSE);   //重绘
}

/*    处理 hello 窗口的鼠标按钮释放事件 */
void Hello::mouseReleaseEvent(QMouseEvent *e)
{
    if (rect().contains(e->pos()))
        emit clicked(); //激活 clicked()信号
```

```
}

/* 处理 hello 窗口的重绘事件 */
void Hello::paintEvent(QPaintEvent*)
{
   static int sin_tbl[16] =
   {
      0, 38, 71, 92, 100, 92, 71, 38, 0, -38, -71, -92, -100, -92,
     -71, -38
   };
   if (t.isEmpty())
      return ;
   // 1: 计算尺寸、位置
   QFontMetrics fm = fontMetrics();
   int w = fm.width(t) + 20;
   int h = fm.height() *2;
   int pmx = width() / 2-w / 2;
   int pmy = height() / 2-h / 2;
   // 2: 创建 pixmap，用窗口背景填充它
   QPixmap pm(w, h);
   pm.fill(this, pmx, pmy);
   // 3: 绘制 pixmap
   QPainter p;
   int x = 10;
   int y = h / 2+fm.descent();
   int i = 0;
   p.begin(&pm);
   p.setFont(font());
   while (!t[i].isNull())
   {
     int i16 = (b + i) &15;
     p.setPen(QColor((15-i16) *16, 255, 255, QColor::Hsv));
     p.drawText(x, y - sin_tbl[i16] *h / 800, t.mid(i, 1), 1);
     x += fm.width(t[i]);
     i++;
  }
  p.end();
  // 4:复制 pixmap 到 Hello 窗口
  bitBlt(this, pmx, pmy, &pm);
}

/*****************************************************
 ** 以下是 main.cpp 的源代码
 *****************************************************/
#include "hello.h"
#include <qapplication.h>
/*
The program starts here. It parses the command line and builds a message
string to be displayed by the Hello widget.
 */
#define QT_NO_WIZARD
int main(int argc, char **argv)
{
    QApplication a(argc, argv);
```

```
    Hello dlg;
    QObject::connect(&dlg, SIGNAL(clicked()), &a, SLOT(quit()));
    a.setMainWidget(&dlg);
    dlg.show();
    return a.exec();
}
```

18.10.2 Microwindows/Nano-X

Microwindows 是嵌入式系统中广为使用的一种图形用户接口，其官方网站是 http://www. microwindows.org，Microwindows 完全支持 Linux 的帧缓冲技术。这个项目的早期目标是在嵌入式 Linux 平台上提供和普通个人电脑上类似的图形用户界面。作为 PC 上 X-Windows 的替代品，Microwindows 提供了和 X-Windows 类似的功能，但是占用的内存要少得多，根据用户的配置，Microwindows 占用的内存资源范围为 60KB～100KB。Microwindows 支持多种外部设备得输入，包括液晶显示器、鼠标和键盘等。

Microwindows 起源于 NanoGUI 项目，早期 Microwindows 有两个版本，一个版本包含了一组和微软的 WIN32 图形用户接口相似的 API，这个版本就是 Microwindows 版本；另外一个版本是基于 X-Windows 的一组 Xlib 风格的 API 函数库，这个版本允许 X11 的二进制代码直接在 Micro Windows 的 Nanx-X 服务器上运行，称之为 Nano-X。

Microwindows 的核心基于显示设备接口，因此可移植性很好，Microwindows 有自己的帧缓冲，并不局限于 Linux 开发平台，在 eCos、FreeBSD、RTEMS 等操作系统上都能很好地运行。此外，Microwindows 能在宿主机上仿真目标机，这一点与 Qt/Embedded 类似，可大大加快开发速度。Microwindows 的几个运行界面如图 18.8 所示。

图 18.8 Microwindows 运行界面

Microwindows 支持新的 Linux 内核帧缓冲结构，提供每像素 1、2、4、8、16、24 和 32 位的支持，支持彩色显示和灰度显示，其中彩色显示包括真彩色（每像素 15、16 和 32 位）和调色板（每像素 1、2、4 和 8 位）两种模式。在彩色显示模式下，所有的颜色用 RGB 格式给出，系统再将它转换成与之最相似的可显示颜色，而在单色模式下中则是转

换成不同的灰度级。Microwindows 支持窗口覆盖和子窗口概念、完全的窗口和客户区剪切、比例和固定字体，还提供了字体和位图文件处理工具。

Microwindows 采用分层设计方法。在最底层，屏幕、鼠标/触摸屏以及键盘驱动程序提供了对物理设备访问的能力。在中间层，实现了一个可移植的图形引擎，支持行绘制、区域填充、剪切以及颜色模型等。在上层，实现多种 API 以适应不同的应用环境。

代码清单 18.17 所示为一个简单的 Microwindows 应用程序，它基于 Nano-X API 编写，创建一个窗口并显示“Hello World”，如图 18.9 所示。

代码清单 18.17　Microwindows 应用程序范例

```
1  #include <stdio.h>
2  #include <microwin/nano-X.h>
3
4  GR_WINDOW_ID wid;
5  GR_GC_ID gc;
6
7  void event_handler(GR_EVENT *event);
8
9  int main(void)
10 {
11    if (GrOpen() < 0)
12    {
13       fprintf(stderr, "GrOpen failed");
14       exit(1);
15   }
16
17   gc = GrNewGC();
18   GrSetGCForeground(gc, 0xFF0000);
19   //创建窗口
20   wid = GrNewWindowEx(GR_WM_PROPS_APPFRAME |
21                          GR_WM_PROPS_CAPTION |
22                          GR_WM_PROPS_CLOSEBOX,
23                            "jollen.org", GR_ROOT_WINDOW_ID, 0, 0,
200,
24                          200,0xFFFFFF);
25   //选择事件
26   GrSelectEvents(wid, GR_EVENT_MASK_CLOSE_REQ|GR_EVENT_MASK_EXPOSURE);
27
28   GrMapWindow(wid);
29   GrMainLoop(event_handler); //挂接事件处理函数
30 }
31
32 void event_handler(GR_EVENT *event)
33 {
34    switch (event->type)
35    {
36       case GR_EVENT_TYPE_EXPOSURE: //显示文本
37          GrText(wid, gc, 50, 50, "Hello World",  - 1, GR_TFASCII);
38          break;
39       case GR_EVENT_TYPE_CLOSE_REQ:   //关闭
40          GrClose();
```

```
        exit(0);
      default:
        break;
  }
}
```

图 18.9　Microwindows 使用范例

18.10.3　MiniGUI

MiniGUI 是由北京飞漫软件技术有限公司开发的面向实时嵌入式系统的轻量级图形用户界面支持系统，1999 年年初遵循 GPL 条款发布第一个版本以来，已广泛应用于手持信息终端、机顶盒、工业控制系统及工业仪表、彩票机、金融终端等产品和领域。目前，MiniGUI 已成为跨操作系统的图形用户界面支持系统，可在 Linux/uClinux、eCos、uC/OS-II、VxWorks 等操作系统上运行，已验证的硬件平台包括 Intel x86、ARM（ARM7/AMR9/StrongARM/xScale）、PowerPC、MIPS 和 M68K（DragonBall/ColdFire）等。

作为操作系统和应用程序之间的中间件，MiniGUI 将底层操作系统及硬件平台的差别隐藏了起来，并对上层应用程序提供了一致的功能特性，这些功能特性如下。

- 完备的多窗口机制和消息传递机制。
- 常用的控件类，包括静态文本框、按钮、单行和多行编辑框、列表框、组合框、进度条、属性页、工具栏、拖动条、树型控件、月历控件等。
- 对话框和消息框支持以及其他 GUI 元素，包括菜单、加速键、插入符、定时器等。
- 界面皮肤支持。用户可通过皮肤支持获得外观非常华丽的图形界面。
- 通过两种不同的内部软件结构支持低端显示设备（比如单色 LCD）和高端显示设备（比如彩色显示器），前者小巧灵活，而后者在前者的基础上提供了更加强大的图形功能。
- Windows 的资源文件支持，如位图、图标、光标等。

- 各种流行图像文件的支持，包括 JPEG、GIF、PNG、TGA、BMP 等。
- 多字符集和多字体支持，目前支持 ISO8859-1～ISO8859-15、GB2312、GBK、GB18030、BIG5、EUC-JP、Shift-JIS、EUC-KR、UNICODE 等字符集，支持等宽点阵字体、变宽点阵字体、Qt/Embedded 使用的嵌入式字体 QPF、TrueType 以及 AdobeType1 等矢量字体。
- 多种键盘布局的支持。MiniGUI 除支持常见的美式 PC 键盘布局之外，还支持法语、德语等语种的键盘布局。
- 简体中文（GB2312）输入法支持，包括内码、全拼、智能拼音等。用户还可以从飞漫软件获得五笔、自然码等输入法支持。
- 针对嵌入式系统的特殊支持，包括一般性的 I/O 流操作，字节序相关函数等。

如图 18.10 所示，基于 MiniGUI 的应用程序一般通过 ANSI C 库以及 MiniGUI 自身提供的 API 来实现自己的功能；MiniGUI 中的可移植层可将特定操作系统及底层硬件的细节隐藏起来，而上层应用程序则无须关心底层的硬件平台输出和输入设备。

图 18.10　MiniGUI 和嵌入式操作系统的关系

因此，为了适合不同的操作系统环境，MiniGUI 可配置成以下 3 种运行模式。

1．MiniGUI-Threads

运行在 MiniGUI-Threads 上的程序可以在不同的线程中建立多个窗口，但所有的窗口在一个进程或者地址空间中运行。这种运行模式非常适合于大多数传统意义上的嵌入式操作系统，比如 uC/OS-II、eCos、VxWorks、pSOS 等。当然，在 Linux 和 uClinux 上，MiniGUI 也能以 MiniGUI-Threads 模式运行。

2．MiniGUI-Processes

和 MiniGUI-Threads 相反，MiniGUI-Processes 上的每个程序是单独的进程，每个进程也可以建立多个窗口，并且实现了多进程窗口系统。MiniGUI-Processes 适合于具有完整 UNIX 特性的嵌入式操作系统，比如嵌入式 Linux、VxWorks 等。

在 Linux 操作系统上运行的 MiniGUI 1.6.x 版本中，该运行模式称为“MiniGUI-Lite”。MiniGUI-Lite 为多进程环境的 Linux 操作系统提供了折中的解决方案，但没有解决进程间的窗口层叠问题。而 MiniGUI2.0.x 实现的 MiniGUI-Processes 模式为 Linux 等多进程操作系统提供了完整的图形界面解决方案。

3．MiniGUI-Standalone

在这种运行模式下，MiniGUI 可以以独立进程的方式运行，既不需要多线程也不

嵌入式学院
华清远见旗下品牌

需要多进程的支持，这种运行模式适合功能单一的应用场合。比如在一些使用 uClinux 的嵌入式产品中，因为各种原因而缺少线程支持，这时，就可以使用 MiniGUI-Standalone 来开发应用软件。

一般而言，MiniGUI-Standalone 模式的适应面最广，可以支持几乎所有的操作系统；MiniGUI-Threads 模式的适用面次之，可运行在支持多任务的实时嵌入式操作系统，或者具备完整 UNIX 特性的普通操作系统；MiniGUI-Processes 模式的适用面较小，它仅适合于具备完整 UNIX 特性的嵌入式操作系统，比如 Linux 和 VxWorks。

MiniGUI 下的通信是一种类似于 Win32 的消息机制，如果有 WIN32 图形用户界面程序的编程基础，编写 MiniGUI 程序将没有门槛。代码清单 18.18 所示为一个完整的 MiniGUI 应用程序，该程序在屏幕上创建一个大小为 240×180 的应用程序窗口，并在窗口客户区的中部显示“Hello world!”，如图 18.11 所示。

代码清单 18.18　MiniGUI 应用程序范例

```
1  #include <stdio.h>
2  #include <minigui/common.h>
3  #include <minigui/minigui.h>
4  #include <minigui/gdi.h>
5  #include <minigui/window.h>
6  static int HelloWinProc ( HWND hWnd, int message, WPARAM wParam, LPARAM lParam )
7  {
8     HDC hdc;
9     switch (message)
10    {
11       case MSG_PAINT:
12          hdc = BeginPaint(hWnd);
13          TextOut(hdc, 60, 60, "Hello world!");//输出文本
14          EndPaint(hWnd, hdc);
15          return 0;
16       case MSG_CLOSE:
17          DestroyMainWindow(hWnd);//破坏窗口
18          PostQuitMessage(hWnd); //释放退出消息
19          return 0;
20     }
21     return DefaultMainWinProc(hWnd, message, wParam, lParam);
22 }
23
24 int MiniGUIMain(int argc, const char *argv[])
25 {
26      MSG Msg;
27      HWND hMainWnd;//主窗口句柄
```

```
    MAINWINCREATE CreateInfo;
    #ifdef _MGRM_PROCESSES
       JoinLayer(NAME_DEF_LAYER, "helloworld", 0, 0);
    #endif
    CreateInfo.dwStyle = WS_VISIBLE | WS_BORDER | WS_CAPTION;
    CreateInfo.dwExStyle = WS_EX_NONE;
    CreateInfo.spCaption = "HelloWorld";
    CreateInfo.hMenu = 0;
    CreateInfo.hCursor = GetSystemCursor(0);
    CreateInfo.hIcon = 0;
    CreateInfo.MainWindowProc = HelloWinProc;//主窗口消息处理程序
    CreateInfo.lx = 0;
    CreateInfo.ty = 0;
    CreateInfo.rx = 240; //水平尺寸
    CreateInfo.by = 180; //垂直尺寸
    CreateInfo.iBkColor = COLOR_lightwhite;
    CreateInfo.dwAddData = 0;
    CreateInfo.hHosting = HWND_DESKTOP;
    hMainWnd = CreateMainWindow(&CreateInfo); //创建主窗口
    if (hMainWnd == HWND_INVALID)
        return  - 1;
    ShowWindow(hMainWnd, SW_SHOWNORMAL);
    while (GetMessage(&Msg, hMainWnd))
   {
        TranslateMessage(&Msg);//消息译码
        DispatchMessage(&Msg); //派送消息
  }
  MainWindowThreadCleanup(hMainWnd);
  return 0;
}
```

图 18.11　MiniGUI 使用范例

18.11 实例：S3C2410 LCD 设备驱动

18.11.1 S3C2410 LCD 控制器硬件描述

S3C2410 内部集成了 LCD 控制器，它支持 STN 和 TFT 屏，其特性如下。

1．STN 屏

支持 3 种扫描方式：4 位单扫描、4 位双扫描和 8 位单扫描。
支持单色、4 级灰度和 16 级灰度屏。
支持 256 色和 4096 色彩色 STN 屏（CSTN）。
支持分辨率为 640*480、320*240、160*160 以及其他规格的多种 LCD。

2．TFT 屏

支持单色、4 级灰度、256 色的调色板显示模式。
支持 64K 和 16M 色非调色板显示模式。
支持分辨率为 640*480，320*240 及其他多种规格的 LCD。

图 18.12 所示为 S3C2410 LCD 控制器内部的逻辑结构，REGBANK 是 LCD 控制器的寄存器组，用来对 LCD 控制器的各项参数进行设置。而 LCDCDMA 则是 LCD 控制器专用的 DMA 信道，负责将视频资料从系统总线（System Bus）上取来，通过 VIDPRCS 从 VD[23:0]发送给 LCD 屏。同时 TIMEGEN 和 LPC3600 负责产生 LCD 屏所需要的控制时序，例如 VSYNC、HSYNC、VCLK、VDEN，然后从 VIDEO MUX 送给 LCD 屏。

图 18.12　S3C2410 LCD 控制器逻辑结构

图 18.13 所示为 S3C2410 LCD 控制器外部的引脚连接，其中 VM 是 LCD 驱动器的 AC 信号。VM 信号被 LCD 驱动器用于改变行和列的电压极性，从而控制像素点的

显示或熄灭。VM 信号可以与每个帧同步，也可以与可变数量的 VLINE 信号同步。

S3C2410 LCD 控制器内部的寄存器可以控制 LCD 控制器接口的工作模式，如寄存器 1 记录屏幕类型（TFT、STN4、STN8、DSCAN4 等）和 BPP，寄存器 2 和寄存器 3 记录定时参数，代码清单 18.19 所示为 S3C2410 LCD 控制器的寄存器。

华清远见

U1B
S3C2410X
LCD接口
VD[23:0]
LCD_PWREN
LCDVF2
LCDVF1
LCDVF0
VM
VFRAME
VLINE
VCLK
LEND
DN0
DP0
DN1
DP1
XMON
nXPON
YMON
nYPON
EINT11
nSS_SPI
KBDSPICLK
SPICLK
KBDSPIMOSI
SPIMOSI
KBDSPIMISO
SPIMISO
I2SSDO
I2SSDI
CDCLK
I2SSCLK
I2SLRCK
IICSDA
IICSCL
LnWBE0
LnWBE1
LnWBE2
LnWBE3
LnGCB5
nGCB7
LnSCA9
LnSRA9
LsCKE
LsCLK0
LsCLK1
VDD33V
R3
4.7k
R2
4.7k
J3
TP3
PROEE UCLK
RnB
WP_SD
nCTS0
nRTS0
TXD0
RXD0
TXD1
RXD1
TXD2
RXD2
nTRST
TCK
TDI
TDD
TMB
ALE
CLE
nFCE
nFRE
nFWE
SDCLK
SDCMD
SDDATA0
SDDATA1
SDDATA2
SDDATA3
R174
R175
R172
R190
R172
R180
22
22
22
10
10

代码清单 18.19　S3C2410 LCD 控制器寄存器

```
#define S3C2410_LCDREG(x) ((x) + S3C24XX_VA_LCD)

/* LCD 控制寄存器 */
#define S3C2410_LCDCON1        S3C2410_LCDREG(0x00)
#define S3C2410_LCDCON2        S3C2410_LCDREG(0x04)
#define S3C2410_LCDCON3        S3C2410_LCDREG(0x08)
#define S3C2410_LCDCON4        S3C2410_LCDREG(0x0C)
#define S3C2410_LCDCON5        S3C2410_LCDREG(0x10)
/*LCD 控制寄存器 1 的位定义*/
#define S3C2410_LCDCON1_CLKVAL(x)  ((x) << 8)
#define S3C2410_LCDCON1_MMODE      (1<<7)
#define S3C2410_LCDCON1_DSCAN4     (0<<5)
#define S3C2410_LCDCON1_STN4       (1<<5)
#define S3C2410_LCDCON1_STN8       (2<<5)
#define S3C2410_LCDCON1_TFT        (3<<5)

#define S3C2410_LCDCON1_STN1BPP    (0<<1)
#define S3C2410_LCDCON1_STN2GREY   (1<<1)
#define S3C2410_LCDCON1_STN4GREY   (2<<1)
#define S3C2410_LCDCON1_STN8BPP    (3<<1)
#define S3C2410_LCDCON1_STN12BPP   (4<<1)

#define S3C2410_LCDCON1_TFT1BPP   (8<<1)
#define S3C2410_LCDCON1_TFT2BPP    (9<<1)
#define S3C2410_LCDCON1_TFT4BPP    (10<<1)
#define S3C2410_LCDCON1_TFT8BPP    (11<<1)
#define S3C2410_LCDCON1_TFT16BPP   (12<<1)
#define S3C2410_LCDCON1_TFT24BPP   (13<<1)

#define S3C2410_LCDCON1_ENVID     (1)
#define S3C2410_LCDCON1_MODEMASK   0x1E

/*LCD 控制寄存器 2 的位定义*/

#define S3C2410_LCDCON2_VBPD(x)     ((x) << 24)
#define S3C2410_LCDCON2_LINEVAL(x)  ((x) << 14)
#define S3C2410_LCDCON2_VFPD(x)     ((x) << 6)
#define S3C2410_LCDCON2_VSPW(x)     ((x) << 0)

#define S3C2410_LCDCON2_GET_VBPD(x) ( ((x) >> 24) & 0xFF)
```

```
41  #define S3C2410_LCDCON2_GET_VFPD(x) ( ((x) >>  6) & 0xFF)
42  #define S3C2410_LCDCON2_GET_VSPW(x) ( ((x) >>  0) & 0x3F)
43
44  /*LCD 控制寄存器 3 的位定义*/
45
46  #define S3C2410_LCDCON3_HBPD(x)       ((x) << 19)
47  #define S3C2410_LCDCON3_WDLY(x)       ((x) << 19)
48  #define S3C2410_LCDCON3_HOZVAL(x)     ((x) << 8)
49  #define S3C2410_LCDCON3_HFPD(x)       ((x) << 0)
50  #define S3C2410_LCDCON3_LINEBLANK(x)  ((x) << 0)
51
52  #define S3C2410_LCDCON3_GET_HBPD(x) ( ((x) >> 19) & 0x7F)
53  #define S3C2410_LCDCON3_GET_HFPD(x) ( ((x) >>  0) & 0xFF)
54
55  /*LCD 控制寄存器 4 的位定义*/
56
57  #define S3C2410_LCDCON4_MVAL(x)    ((x) << 8)
58  #define S3C2410_LCDCON4_HSPW(x)    ((x) << 0)
59  #define S3C2410_LCDCON4_WLH(x)     ((x) << 0)
60
61  #define S3C2410_LCDCON4_GET_HSPW(x) ( ((x) >>  0) & 0xFF)
62
63  /*LCD 控制寄存器 5 的位定义*/
64
65  #define S3C2410_LCDCON5_BPP24BL    (1<<12)
66  #define S3C2410_LCDCON5_FRM565     (1<<11)
67  #define S3C2410_LCDCON5_INVVCLK    (1<<10)
68  #define S3C2410_LCDCON5_INVVLINE    (1<<9)
69  #define S3C2410_LCDCON5_INVVFRAME   (1<<8)
70  #define S3C2410_LCDCON5_INVVD       (1<<7)
71  #define S3C2410_LCDCON5_INVVDEN     (1<<6)
72  #define S3C2410_LCDCON5_INVPWREN    (1<<5)
73  #define S3C2410_LCDCON5_INVLEND     (1<<4)
74  #define S3C2410_LCDCON5_PWREN       (1<<3)
75  #define S3C2410_LCDCON5_ENLEND      (1<<2)
76  #define S3C2410_LCDCON5_BSWP        (1<<1)
77  #define S3C2410_LCDCON5_HWSWP        (1<<0)
78
79  /* 显示缓冲区开始位置寄存器 */
80  #define S3C2410_LCDSADDR1   S3C2410_LCDREG(0x14)
81  #define S3C2410_LCDSADDR2   S3C2410_LCDREG(0x18)
82  #define S3C2410_LCDSADDR3   S3C2410_LCDREG(0x1C)
83
84  #define S3C2410_LCDBANK(x)   ((x) << 21)
85  #define S3C2410_LCDBASEU(x)    (x)
86
87  #define S3C2410_OFFSIZE(x)  ((x) << 11)
88  #define S3C2410_PAGEWIDTH(x)   (x)
89
```

```
90  /* 查找表 */
91
92  #define S3C2410_REDLUT    S3C2410_LCDREG(0x20)
93  #define S3C2410_GREENLUT   S3C2410_LCDREG(0x24)
94  #define S3C2410_BLUELUT     S3C2410_LCDREG(0x28)
95
96  #define S3C2410_DITHMODE   S3C2410_LCDREG(0x4C)
97  #define S3C2410_TPAL       S3C2410_LCDREG(0x50)
98
99  #define S3C2410_TPAL_EN        (1<<24)
```

18.11.2 S3C2410 LCD 驱动的模块加载与卸载函数

在 S3C2410 中，作为一个相对独立的硬件单元，LCD 控制器被认定为平台设备，因此，在驱动的模块加载和卸载函数中，分别注册和注销对应的 platform_driver 即可，如代码清单 18.20 所示。

代码清单 18.20　S3C2410 LCD 驱动的模块加载与卸载函数

```
1  static struct platform_driver s3c2410fb_driver = {
2  .probe     = s3c2410fb_probe,//平台驱动探测函数
3  .remove     = s3c2410fb_remove,  //平台驱动移除函数
4  .suspend   = s3c2410fb_suspend,
5  .resume    = s3c2410fb_resume,
6  .driver      = {
7       .name  = "s3c2410-lcd",  //驱动名
8  .owner      = THIS_MODULE,
9  },
10 };
11
12 int __devinit s3c2410fb_init(void)
13 {
14  return platform_driver_register(&s3c2410fb_driver);//注册平台设备
15 }
16
17 static void __exit s3c2410fb_cleanup(void)
18 {
19  platform_driver_unregister(&s3c2410fb_driver);//注销平台设备
20 }
21
22 module_init(s3c2410fb_init);
23 module_exit(s3c2410fb_cleanup);
```

S3C2410 LCD 控制器平台设备定义于 arch/arm/mach-s3c2410 对应的目录中，其对应的 platform_device 结构体如代码清单 18.21 所示。

代码清单 18.21　S3C2410 LCD 驱动的平台设备结构体

```
1  struct platform_device s3c_device_lcd = {
2  .name      = "s3c2410-lcd", //平台设备名
```

```
3  .id        = -1,
4  .num_resources   = ARRAY_SIZE(s3c_lcd_resource),//资源数
5  .resource    = s3c_lcd_resource,//资源
6  .dev            = {
7     .dma_mask     = &s3c_device_lcd_dmamask,
8       .coherent_dma_mask   = 0xffffffffUL
9   }
10 };
```

第 4 行和第 5 行访问的 s3c_lcd_resource 为平台设备所使用的资源，S3C2410 LCD 平台设备所使用的资源如代码清单 18.22 所示。

代码清单 18.22　S3C2410 LCD 驱动的平台设备资源

```
1  static struct resource s3c_lcd_resource[] = {
2  [0] = {  //I/O 内存资源
3     .start = S3C24XX_PA_LCD,
4     .end   = S3C24XX_PA_LCD + S3C24XX_SZ_LCD - 1,
5     .flags = IORESOURCE_MEM,
6   },
7   [1] = {  //IRQ 资源
8        .start = IRQ_LCD,
9        .end   = IRQ_LCD,
10       .flags = IORESOURCE_IRQ,
11  }
12 };
```

另外，内核还定义了一个 s3c2410fb_mach_info 结构体，记录 LCD 的“机器信息”（即硬件特征），并将这个结构体赋值给平台设备的平台数据（platform_data），如代码清单 18.23 所示。

代码清单 18.23　S3C2410 LCD 驱动平台设备的平台数据

```
1  void __init s3c24xx_fb_set_platdata(struct s3c2410fb_mach_info *pd)
2  {
3    struct s3c2410fb_mach_info *npd;
4
5    npd = kmalloc(sizeof(*npd), GFP_KERNEL); //申请 s3c2410fb_mach_info
内存
6    if (npd)
7    {
8      memcpy(npd, pd, sizeof(*npd));
9      //设置设备的 platform_data 为 s3c2410fb_mach_info
10     s3c_device_lcd.dev.platform_data = npd;
11   }
12   else
13   {
14     printk(KERN_ERR "no memory for LCD platform data\n");
15   }
```

```
16 }
17
18 struct s3c2410fb_mach_info
19 {
20     unsigned char fixed_syncs; /* 不更新同步和边界 */
21
22     /* 屏幕尺寸 */
23     int width;
24     int height;
25
26     /* 屏幕信息 */
27     struct s3c2410fb_val xres;
28     struct s3c2410fb_val yres;
29     struct s3c2410fb_val bpp;
30
31     /* LCD 配置寄存器 */
32     struct s3c2410fb_hw regs;
33     ...
34 };
35
36 struct s3c2410fb_val
37 {
38     unsigned int defval; //默认值
39     unsigned int min;    //最小值
40     unsigned int max;    //最大值
41 };
42
43 struct s3c2410fb_hw
44 {
45    /*5 个 LCD 配置寄存器*/
46    unsigned long lcdcon1;
47    unsigned long lcdcon2;
48    unsigned long lcdcon3;
49    unsigned long lcdcon4;
50    unsigned long lcdcon5;
51 };
```

18.11.3 S3C2410 LCD 驱动的探测与移除函数

代码清单 18.20 所示的模块加载函数中对平台驱动的注册会导致其中探测函数的执行，这个探测函数完成 LCD 硬件的初始化、FBI 参数的填充、申请显示缓冲区并注册帧缓冲设备，如代码清单 18.24 所示；模块卸载函数中对平台驱动的注销会导致其中移除函数的执行，这个函数释放显示缓冲区并注销帧缓冲设备，如代码清单 18.25 所示。

代码清单 18.24 S3C2410 LCD 驱动的探测函数

```
static int __init s3c2410fb_probe(struct platform_device *pdev)
{
    struct s3c2410fb_info *info;
    struct fb_info *fbinfo;
    struct s3c2410fb_hw *mregs;
    int ret;
    int irq;
    int i;

    mach_info = pdev->dev.platform_data; //获得 mach_info 结构体指针
    if (mach_info == NULL)
    {
        dev_err(&pdev->dev, "no platform data for lcd, cannot attach\n");
      return - EINVAL;
    }

    mregs = &mach_info->regs; //获得 mach_info 结构体中的寄存器指针

    irq = platform_get_irq(pdev, 0); //获得 IRQ 号
    if (irq < 0)
    {
       dev_err(&pdev->dev, "no irq for device\n");
       return - ENOENT;
    }

    //分配 FBI 结构体
      fbinfo = framebuffer_alloc(sizeof(struct s3c2410fb_info), &pdev->dev);
    if (!fbinfo)
    {
```

```
        return  - ENOMEM;
}

info = fbinfo->par;
info->fb = fbinfo;
platform_set_drvdata(pdev, fbinfo);//设置平台设备数据

dprintk("devinit\n");

strcpy(fbinfo->fix.id, driver_name);

memcpy(&info->regs, &mach_info->regs, sizeof(info->regs));

info->mach_info = pdev->dev.platform_data;

//初始化固定参数
fbinfo->fix.type = FB_TYPE_PACKED_PIXELS;
fbinfo->fix.type_aux = 0;
fbinfo->fix.xpanstep = 0;
fbinfo->fix.ypanstep = 0;
fbinfo->fix.ywrapstep = 0;
fbinfo->fix.accel = FB_ACCEL_NONE;

//初始化可变参数
fbinfo->var.nonstd = 0;
fbinfo->var.activate = FB_ACTIVATE_NOW;
fbinfo->var.height = mach_info->height;
fbinfo->var.width = mach_info->width;
fbinfo->var.accel_flags = 0;
fbinfo->var.vmode = FB_VMODE_NONINTERLACED;

fbinfo->fbops = &s3c2410fb_ops;   //fb_ops
fbinfo->flags = FBINFO_FLAG_DEFAULT;
fbinfo->pseudo_palette = &info->pseudo_pal;

//初始化可变参数中的分辨率和bpp
fbinfo->var.xres = mach_info->xres.defval;
fbinfo->var.xres_virtual = mach_info->xres.defval;
fbinfo->var.yres = mach_info->yres.defval;
fbinfo->var.yres_virtual = mach_info->yres.defval;
fbinfo->var.bits_per_pixel = mach_info->bpp.defval;
```

```
    //初始化可变参数中的上下边界和垂直同步
fbinfo->var.upper_margin=3C2410_LCDCON2_GET_VBPD(mregs->lcdcon2) + 1;
                                fbinfo->var.lower_margin        =
3C2410_LCDCON2_GET_VFPD(mregs->lcdcon2) + 1;
    fbinfo->var.vsync_len = S3C2410_LCDCON2_GET_VSPW(mregs->lcdcon2)
+ 1;
    //初始化可变参数中的左右边界和水平同步
                                 fbinfo->var.left_margin        =
S3C2410_LCDCON3_GET_HFPD(mregs->lcdcon3) + 1;
                                fbinfo->var.right_margin        =
S3C2410_LCDCON3_GET_HBPD(mregs->lcdcon3) + 1;
    fbinfo->var.hsync_len = S3C2410_LCDCON4_GET_HSPW(mregs->lcdcon4)
+ 1;
    //设置可变参数中的 R/G/B 位数和位置
    fbinfo->var.red.offset = 11;
    fbinfo->var.green.offset = 5;
    fbinfo->var.blue.offset = 0;
    fbinfo->var.transp.offset = 0;
    fbinfo->var.red.length = 5;
    fbinfo->var.green.length = 6;
    fbinfo->var.blue.length = 5;
    fbinfo->var.transp.length = 0;
    fbinfo->fix.smem_len = mach_info->xres.max *mach_info->yres.max
       *mach_info ->bpp.max / 8;    //buffer 长度

    for (i = 0; i < 256; i++)
        info->palette_buffer[i] = PALETTE_BUFF_CLEAR;

    //申请内存区域
     if (!request_mem_region((unsigned long)S3C24XX_VA_LCD, SZ_1M,
"s3c2410-lcd"))
    {
       ret =  - EBUSY;
       goto dealloc_fb;
    }
   dprintk("got LCD region\n");

```

```
102  //申请中断
103  ret = request_irq(irq, s3c2410fb_irq, SA_INTERRUPT, pdev->name,
info);
104  if (ret)
105  {
106     dev_err(&pdev->dev, "cannot get irq %d - err %d\n", irq, ret);
107     ret =  - EBUSY;
108     goto release_mem;
109  }
110
111  //获得时钟源并使能
112  info->clk = clk_get(NULL, "lcd");
113  if (!info->clk || IS_ERR(info->clk))
114  {
115      printk(KERN_ERR "failed to get lcd clock source\n");
116      ret =  - ENOENT;
117      goto release_irq;
118  }
119
120  clk_enable(info->clk);
121  dprintk("got and enabled clock\n");
122
123  msleep(1);
124
125  /* 初始化显示缓冲区 */
126  ret = s3c2410fb_map_video_memory(info);
127  if (ret)
128  {
129      printk(KERN_ERR "Failed to allocate video RAM: %d\n", ret);
130      ret =  - ENOMEM;
131      goto release_clock;
132  }
133  dprintk("got video memory\n");
134
135  ret = s3c2410fb_init_registers(info); //初始化 S3C2410 的 LCD 控制
器的寄存器
136
137  ret = s3c2410fb_check_var(&fbinfo->var, fbinfo);//检查可变参数
138
139  ret = register_framebuffer(fbinfo);//注册帧缓冲设备
```

```
140  if (ret < 0)
141  {
142      printk(KERN_ERR "Failed to register framebuffer device: %d\n",
ret);
143      goto free_video_memory;
144  }
145
146  /* 创建设备文件 */
147  device_create_file(&pdev->dev, &dev_attr_debug);
148
149  printk(KERN_INFO "fb%d: %s frame buffer device\n", fbinfo->node,
fbinfo
150    ->fix.id);
151
152  return 0;
153
154  free_video_memory: s3c2410fb_unmap_video_memory(info);
155  release_clock: clk_disable(info->clk);
156  clk_put(info->clk);
157  release_irq:
158  free_irq(irq, info);
159  release_mem:
160            release_mem_region((unsigned    long)S3C24XX_VA_LCD,
S3C24XX_SZ_LCD);
161  dealloc_fb:
162  framebuffer_release(fbinfo);
163  return ret;
164 }
```

代码清单 18.25　S3C2410 LCD 驱动的移除函数

```
1  static int s3c2410fb_remove(struct platform_device *pdev)
2  {
3    struct fb_info *fbinfo = platform_get_drvdata(pdev);
4    struct s3c2410fb_info *info = fbinfo->par;
5    int irq;
6
7    s3c2410fb_stop_lcd(); //停止 LCD
8    msleep(1);
```

```
9
10    s3c2410fb_unmap_video_memory(info); //释放显示缓冲区
11
12    if (info->clk)
13    {
14       //释放时钟源
15       clk_disable(info->clk);
16       clk_put(info->clk);
17       info->clk = NULL;
18    }
19
20    irq = platform_get_irq(pdev, 0);   //获得平台设备使用的 IRQ
21    free_irq(irq, info); //释放 IRQ
22    //释放内存区域
23              release_mem_region((unsigned    long)S3C24XX_VA_LCD,
S3C24XX_SZ_LCD);
24
25    unregister_framebuffer(fbinfo); //注销帧缓冲设备
26
27    return 0;
28 }
```

代码清单 18.24 第 126 行调用的 s3c2410fb_map_video_memory()用于申请显示缓冲区，代码清单 18.25 第 10 行调用的 s3c2410fb_unmap_video_memory()用户释放显示缓冲区，如代码清单 18.26 所示。

代码清单 18.26　S3C2410 LCD 驱动的显示缓冲区申请和释放

```
1  static int __init s3c2410fb_map_video_memory(struct s3c2410fb_info
*fbi)
2  {
3  fbi->map_size = PAGE_ALIGN(fbi->fb->fix.smem_len + PAGE_SIZE);
4  fbi->map_cpu  = dma_alloc_writecombine(fbi->dev, fbi->map_size,
5                     &fbi->map_dma, GFP_KERNEL);//分配内存
6
7  fbi->map_size = fbi->fb->fix.smem_len;//显示缓冲区长度
8
9  if (fbi->map_cpu) {
10     memset(fbi->map_cpu, 0xf0, fbi->map_size);
11
12     fbi->screen_dma       = fbi->map_dma;
13     fbi->fb->screen_base     = fbi->map_cpu;
14     fbi->fb->fix.smem_start  = fbi->screen_dma;//显示缓冲区开始地址
15
```

```
    16      dprintk("map_video_memory: dma=%08x cpu=%p size=%08x\n",
    17           fbi->map_dma, fbi->map_cpu, fbi->fb->fix.smem_len);
    18   }
    19
    20  return fbi->map_cpu ? 0 : -ENOMEM;
    21 }
    22
    23    static    inline    void    s3c2410fb_unmap_video_memory(struct
s3c2410fb_info *fbi)
    24 {
    25   //释放内存
    26        dma_free_writecombine(fbi->dev,fbi->map_size,fbi->map_cpu,
fbi->map_dma);
    27 }
```

18.11.4 S3C2410 LCD 驱动挂起与恢复函数

如果内核配置了能量管理（即定义了 CONFIG_PM 宏），平台驱动部分应该增加挂起和恢复函数，在挂起函数中关闭 LCD 和其时钟源以节省能量，在恢复函数中开启 LCD 和其时钟源，如代码清单 18.27 所示。

代码清单 18.27　S3C2410 LCD 驱动挂起与恢复函数

```
    1  #ifdef CONFIG_PM  //支持能量管理
    2     /* LCD 控制器的挂起和恢复支持 */
    3
    4       static  int  s3c2410fb_suspend(struct  platform_device  *dev,
pm_message_t state)
    5    {
    6       struct fb_info *fbinfo = platform_get_drvdata(dev);
    7       struct s3c2410fb_info *info = fbinfo->par;
    8
    9       s3c2410fb_stop_lcd();//停止 LCD
    10
    11      msleep(1);    //在停止时钟源前延迟
    12      clk_disable(info->clk); //停止时钟源
    13
    14      return 0;
    15   }
    16
    17   static int s3c2410fb_resume(struct platform_device *dev)
```

```
18  {
19      struct fb_info *fbinfo = platform_get_drvdata(dev);
20      struct s3c2410fb_info *info = fbinfo->par;
21
22      clk_enable(info->clk);   //启动时钟源
23      msleep(1);
24
25      s3c2410fb_init_registers(info); //初始化 LCD 控制器寄存器
26
27      return 0;
28  }
29
30 #else //不支持能量管理
31    #define s3c2410fb_suspend NULL
32    #define s3c2410fb_resume  NULL
33 #endif
```

18.11.5 S3C2410 LCD 驱动的 fb_ops 成员函数

1. fb_ops 结构体

S3C2410 LCD 驱动中定义了如代码清单 18.28 所示的 fb_ops，其中的 s3c2410fb_check_var()、s3c2410fb_set_par()、s3c2410fb_blank()和 s3c2410fb_setcolreg()成员函数特别针对 S3C2410 而实现。

代码清单 18.28 S3C2410 LCD 驱动的 fb_ops 结构体

```
1  static struct fb_ops s3c2410fb_ops = {
2  .owner          = THIS_MODULE,
3  .fb_check_var   = s3c2410fb_check_var,//检查可变参数
4  .fb_set_par     = s3c2410fb_set_par,  //设置参数
5  .fb_blank       = s3c2410fb_blank,  //显示空白
6  .fb_setcolreg   = s3c2410fb_setcolreg, //设置颜色表
7  .fb_fillrect    = cfb_fillrect, //矩形填充
8  .fb_copyarea    = cfb_copyarea,
9  .fb_imageblit   = cfb_imageblit,
10 };
```

2. 参数检查函数

s3c2410fb_check_var()函数用于检查用户设置的屏幕参数是否合理，譬如 xres、yres 和 BPP 都有一个范围 min～max，用户的设置如果在这个范围之外，则进行调整。另外，当 BPP 改变后，R、G、B 的位域也要进行相应调整，如代码清单 18.29 所示。

代码清单 18.29 S3C2410 LCD 驱动的参数检查函数

```
1  static int s3c2410fb_check_var(struct fb_var_screeninfo *var,
```

```
                    struct fb_info *info)
{
struct s3c2410fb_info *fbi = info->par;

dprintk("check_var(var=%p, info=%p)\n", var, info);

/* 验证 x/y 解析度 */

 if (var->yres > fbi->mach_info->yres.max)
     var->yres = fbi->mach_info->yres.max;
 else if (var->yres < fbi->mach_info->yres.min)
      var->yres = fbi->mach_info->yres.min;

 if (var->xres > fbi->mach_info->xres.max)
     var->yres = fbi->mach_info->xres.max;
 else if (var->xres < fbi->mach_info->xres.min)
     var->xres = fbi->mach_info->xres.min;

 /* 验证 bpp */

 if (var->bits_per_pixel > fbi->mach_info->bpp.max)
    var->bits_per_pixel = fbi->mach_info->bpp.max;
 else if (var->bits_per_pixel < fbi->mach_info->bpp.min)
    var->bits_per_pixel = fbi->mach_info->bpp.min;

 /* 设置 R、G、B 位置 */

 if (var->bits_per_pixel == 16) {
      var->red.offset     = 11;
      var->green.offset   = 5;
      var->blue.offset    = 0;
      var->red.length     = 5;
      var->green.length   = 6;
      var->blue.length    = 5;
      var->transp.length = 0;
 } else {
      var->red.length     = var->bits_per_pixel;
      var->red.offset     = 0;
      var->green.length   = var->bits_per_pixel;
```

```
    41        var->green.offset    = 0;
    42        var->blue.length     = var->bits_per_pixel;
    43        var->blue.offset     = 0;
    44        var->transp.length = 0;
    45   }
    46
    47   return 0;
    48 }
```

3. 参数设置函数

s3c2410fb_set_par()函数根据用户设置的可变参数 var 调整固定参数 fix，修改 fix 的 line_length，最终将相应的修改在硬件上激活，如代码清单 18.30 所示。

代码清单 18.30　参数设置函数

```
    1  static int s3c2410fb_set_par(struct fb_info *info)
    2  {
    3      struct s3c2410fb_info *fbi = info->par;
    4      struct fb_var_screeninfo *var = &info->var;
    5
    6      if (var->bits_per_pixel == 16)
    7        fbi->fb->fix.visual = FB_VISUAL_TRUECOLOR;
    8     else
    9        fbi->fb->fix.visual = FB_VISUAL_PSEUDOCOLOR;
    10
    11    fbi->fb->fix.line_length = (var->width *var->bits_per_pixel) /
8;
    12
    13   /* 激活新的配置 */
    14
    15   s3c2410fb_activate_var(fbi, var);
    16   return 0;
    17 }
    18
    19 /* 根据 var 和 fix 参数设置 LCD 控制器的硬件寄存器 */
    20  static  void  s3c2410fb_activate_var(struct  s3c2410fb_info  *fbi,
struct
    21     fb_var_screeninfo *var)
    22 {
    23     fbi->regs.lcdcon1 &= ~S3C2410_LCDCON1_MODEMASK;
    24
    25     dprintk("%s: var->xres  = %d\n", __FUNCTION__, var->xres);
    26     dprintk("%s: var->yres  = %d\n", __FUNCTION__, var->yres);
```

```
27            dprintk("%s:  var->bpp      =  %d\n",  __FUNCTION__,
var->bits_per_pixel);
28
29     switch (var->bits_per_pixel)
30    {
31      //设置bpp
32      case 1:
33         fbi->regs.lcdcon1 |= S3C2410_LCDCON1_TFT1BPP;
34         break;
35      case 2:
36         fbi->regs.lcdcon1 |= S3C2410_LCDCON1_TFT2BPP;
37         break;
38      case 4:
39        fbi->regs.lcdcon1 |= S3C2410_LCDCON1_TFT4BPP;
40        break;
41      case 8:
42        fbi->regs.lcdcon1 |= S3C2410_LCDCON1_TFT8BPP;
43        break;
44      case 16:
45        fbi->regs.lcdcon1 |= S3C2410_LCDCON1_TFT16BPP;
46        break;
47   }
48
49   /* 检查是否需要更新同步和边界 */
50
51   if (!fbi->mach_info->fixed_syncs)
52   {
53             dprintk("setting  vert:  up=%d,  low=%d,  sync=%d\n",
var->upper_margin,
54         var ->lower_margin, var->vsync_len);
55
56             dprintk("setting  horz:  lft=%d,  rt=%d,  sync=%d\n",
var->left_margin, var
57         ->right_margin, var->hsync_len);
58
59     fbi->regs.lcdcon2 = S3C2410_LCDCON2_VBPD(var->upper_margin - 1)
|
60        S3C2410_LCDCON2_VFPD(var->lower_margin - 1) |
```

```
        S3C2410_LCDCON2_VSPW(var ->vsync_len - 1);

    fbi->regs.lcdcon3 = S3C2410_LCDCON3_HBPD(var->right_margin - 1) |
        S3C2410_LCDCON3_HFPD(var->left_margin - 1);

    fbi->regs.lcdcon4 &= ~S3C2410_LCDCON4_HSPW(0xff);
    fbi->regs.lcdcon4 |= S3C2410_LCDCON4_HSPW(var->hsync_len - 1);
  }

  /* 更新 X/Y 信息 */

  fbi->regs.lcdcon2 &= ~S3C2410_LCDCON2_LINEVAL(0x3ff);
  fbi->regs.lcdcon2 |= S3C2410_LCDCON2_LINEVAL(var->yres - 1);

  fbi->regs.lcdcon3 &= ~S3C2410_LCDCON3_HOZVAL(0x7ff);
  fbi->regs.lcdcon3 |= S3C2410_LCDCON3_HOZVAL(var->xres - 1);

  if (var->pixclock > 0)
  {
     int clkdiv = s3c2410fb_calc_pixclk(fbi, var->pixclock);

     clkdiv = (clkdiv / 2) - 1;
     if (clkdiv < 0)
         clkdiv = 0;

     fbi->regs.lcdcon1 &= ~S3C2410_LCDCON1_CLKVAL(0x3ff);
     fbi->regs.lcdcon1 |= S3C2410_LCDCON1_CLKVAL(clkdiv);
  }

  /* 重写 LCD 控制器寄存器 */

  dprintk("new register set:\n");
  dprintk("lcdcon[1] = 0x%08lx\n", fbi->regs.lcdcon1);
  dprintk("lcdcon[2] = 0x%08lx\n", fbi->regs.lcdcon2);
  dprintk("lcdcon[3] = 0x%08lx\n", fbi->regs.lcdcon3);
  dprintk("lcdcon[4] = 0x%08lx\n", fbi->regs.lcdcon4);
  dprintk("lcdcon[5] = 0x%08lx\n", fbi->regs.lcdcon5);

              writel(fbi->regs.lcdcon1    &~S3C2410_LCDCON1_ENVID,
```

```
S3C2410_LCDCON1);
    100   writel(fbi->regs.lcdcon2, S3C2410_LCDCON2);
    101   writel(fbi->regs.lcdcon3, S3C2410_LCDCON3);
    102   writel(fbi->regs.lcdcon4, S3C2410_LCDCON4);
    103   writel(fbi->regs.lcdcon5, S3C2410_LCDCON5);
    104
    105   /* set lcd address pointers */
    106   s3c2410fb_set_lcdaddr(fbi);
    107
    108   writel(fbi->regs.lcdcon1, S3C2410_LCDCON1);
    109 }
```

18.12 总结

帧缓冲设备是一种典型的字符设备，它统一了显存，将显示缓冲区直接映射到用户空间。帧缓冲设备驱动 file_operations 中 VFS 接口函数由 fbmem.c 文件统一实现。这样，驱动工程师的工作重点将是实现针对特定设备 fb_info 中的 fb_ops 的成员函数，另外，理解并能灵活地修改 fb_info 中的 var 和 fix 参数非常关键。fb_info 中的 var 参数直接和 LCD 控制器的硬件设置以及 LCD 屏幕对应。

在用户空间，应用程序直接按照预先设置的 R、G、B 位数和偏移写经过 mmap() 映射后的显示缓冲区就可实现图形的显示，省去了内存从用户空间到内核空间的复制过程。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm

第 19 章　Flash 设备驱动

Flash 在嵌入式系统中是必不可少的，它是 BootLoader、Linux 内核和文件系统的最佳载体。在 Linux 内核中，引入了 MTD 层为 NOR Flash 和 NAND Flash 设备提供统一的接口，从而使得 Flash 驱动的设计工作大为简化。

19.1 节讲解了 Linux Flash 驱动的结构，主要讲解了 MTD 系统的层次结构和接口。

19.2 节和 19.3 节分别讲解了 NOR Flash 和 NAND Flash 驱动的设计方法，给出了设计模板。

19.4 节和 19.5 节分别以 S3C2410 外围 NOR Flash 和 NAND Flash 为实例进一步讲解了 NOR Flash 和 NAND Flash 驱动的设计。

19.6 节讲解了如何在 Flash 上建立 cramfs、jffs/jffs2 及 yaffs/yaffs2 文件系统。

华清远见

19.1 Linux Flash 驱动结构

19.1.1 Linux MTD 系统层次

在 Linux 系统中，提供了 MTD（Memory Technology Device，内存技术设备）系统来建立 Flash 针对 Linux 的统一、抽象的接口。MTD 将文件系统与底层的 Flash 存储器进行了隔离，使 Flash 驱动工程师无须关心 Flash 作为字符设备和块设备与 Linux 内核的 接口。

图 19.1 Linux MTD 系统

如图 19.1 所示，在引入 MTD 后，Linux 系统中的 Flash 设备驱动及接口可分为 4 层，从上到下依次是：设备节点、MTD 设备层、MTD 原始设备层和硬件驱动层，这 4 层的作用如下。

- 硬件驱动层：Flash 硬件驱动层负责 Flash 硬件设备的读、写、擦除，Linux MTD 设备的 NOR Flash 芯片驱动位于 drivers/mtd/chips 子目录下，NAND 型 Flash 的驱动程序则位于/drivers/mtd/nand 子目录下。
- MTD 原始设备层：MTD 原始设备层由两部分组成，一部分是 MTD 原始设备的通用代码，另一部分是各个特定的 Flash 的数据，例如分区。
- MTD 设备层：基于 MTD 原始设备，Linux 系统可以定义出 MTD 的块设备（主设备号 31）和字符设备（设备号 90），构成 MTD 设备层。MTD 字符设备的定义在 mtdchar.c 中实现，通过注册一系列 file_operation 函数（lseek、open、close、read、write、ioctl）可实现对 MTD 设备的读写和控制。MTD 块设备则是定义了一个描述 MTD 块设备的结构 mtdblk_dev，并声明了一个名为 mtdblks 的指针数组，这数组中的每一个 mtdblk_dev 和 mtd_table 中的每一个 mtd_info 一一对应。
- 设备节点：通过 mknod 在/dev 子目录下建立 MTD 字符设备节点（主设备号为 90）和 MTD 块设备节点（主设备号为 31），用户通过访问此设备节点即可访问 MTD 字符设备和块设备。

19.1.2 Linux MTD 系统接口

如图 19.2 所示，在引入 MTD 后，底层 Flash 驱动直接与 MTD 原始设备层交互，利用其提供的接口注册设备和分区。

用于描述 MTD 原始设备的数据结构是 mtd_info，这其中定义了大量关于 MTD 的

数据和操作函数，这个结构体的定义如代码清单 19.1 所示。mtd_info 是表示 MTD 原始设备的结构体，每个分区也被认为是一个 mtd_info，例如，如果有两个 MTD 原始设备，而每个上有 3 个分区，在系统中就将共有 6 个 mtd_info 结构体，这些 mtd_info 的指针被存放在名为 mtd_table 的数组里。

华清远见

图 19.2　底层 Flash 驱动

代码清单 19.1　mtd_info 结构体

```
struct mtd_info
{

  u_char type; // 内存技术的类型
  u_int32_t flags; //标志位
  u_int32_t size; //mtd 设备的大小
  u_int32_t erasesize; //主要的擦除块大小（同一个
                        //mtd 设备可能有数种不同的 erasesize）
  u_int32_t oobblock; // oob 块大小
  u_int32_t oobsize; // oob 数据大小
  u_int32_t ecctype; //ecc 类型
  u_int32_t eccsize; //ecc 工作的范围

  char *name;
  int index; //索引

  //不同的 erasesize 的区域
  int numeraseregions; //不同 erasesize 的区域的数目（通常是 1）
  struct mtd_erase_region_info *eraseregions;

  u_int32_t bank_size;
  struct module *module;
  int(*erase)(struct mtd_info *mtd, struct erase_info *instr);
  //此 routine 用于将一个 erase_info 加入 erase queue
```

```
25
26  /*针对 eXecute-In-Place */
27  int(*point)(struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen,
28    u_char **mtdbuf);
29
30  /* 如果 unpoint 为空，不允许 XIP */
31  void(*unpoint)(struct mtd_info *mtd, u_char *addr);
32
33  int(*read)(struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen,
34    u_char *buf);  //读 Flash
35  int(*write)(struct mtd_info *mtd, loff_t to, size_t len, size_t *retlen,
36    const u_char *buf); //写 Flash
37   int(*read_ecc)(struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen,
38    u_char *buf, u_char *eccbuf); //带 ecc 的读
39  int(*write_ecc)(struct mtd_info *mtd, loff_t to, size_t len, size_t *retlen,
40    const u_char *buf, u_char *eccbuf); //带 ecc 的写
41   int(*read_oob)(struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen,
42    u_char *buf); //读 out-of-band
43  int(*write_oob)(struct mtd_info *mtd, loff_t to, size_t len, size_t *retlen,
44    const u_char *buf); //写 out-of-band
45  /* iovec-based 读写函数，对于 NAND Flash 需要针对它定义  */
46  int(*readv)(struct mtd_info *mtd, struct iovec *vecs, unsigned long count,
47    loff_t from, size_t *retlen);
48   int(*writev)(struct mtd_info *mtd, const struct iovec *vecs, unsigned long
49    count, loff_t to, size_t *retlen);
50
51  /* Sync */
52  void(*sync)(struct mtd_info *mtd);
53
54  /* 设备锁 */
55  int(*lock)(struct mtd_info *mtd, loff_t ofs, size_t len);
56  int(*unlock)(struct mtd_info *mtd, loff_t ofs, size_t len);
57  /* 能量管理函数*/
58  int(*suspend)(struct mtd_info *mtd);
59  void(*resume)(struct mtd_info *mtd);
60
61  void *priv; //私有数据
62 }
```

mtd_info 的 type 字段给出底层物理设备的类型，包括 MTD_RAM、MTD_ROM、MTD_ NORFlash、MTD_NANDFlash、MTD_PEROM 等。

flags 字段包括 MTD_ERASEABLE（可擦除）、MTD_WRITEB_WRITEABLE（可编程）、MTD_XIP（可片内执行）、MTD_OOB（NAND 带外数据）、MTD_ECC（支持自动 ECC）等。某些内存技术支持带外数据（OOB），例如，NAND Flash 每 512 字节就会有 16 个字节的“额外数据”，用于存放纠错码或元数据。这是因为，所有 Flash 器件都受位交换现象的困扰，而 NAND 发生的概率比 NOR 大，因此 NAND 厂商推荐在使用 NAND 的时候最好要使用 ECC（Error Checking and Correcting），汉明码是最简单的 ECC。

ecctype 字段表明了 ECC 的类型，包括 MTD_ECC_NONE（不支持自动 ECC）、MTD_ECC_ RS_DiskOnChip（DiskOnChip 上自动 ECC）、MTD_ECC_SW（Toshiba & Samsung 设备 SW ECC）。

mtdcore.c 中定义了 MTD 设备数组：

```
static struct mtd_info *mtd_table[MAX_MTD_DEVICES];
```

最多可以有 MAX_MTD_DEVICES（默认定义为 16）个设备，每个 MTD 分区也算一个 MTD 设备。

mtd_info 中的 read()、write()、read_ecc()、write_ecc()、read_oob()、write_oob()是 MTD 设备驱动要实现的主要函数，后面我们将看到，在 NOR 和 NAND 的驱动代码中几乎看不到 mtd_info 的成员函数（也即这些成员函数对于 Flash 芯片驱动是透明的），这是因为 Linux 在 MTD 的下层实现了针对 NOR Hash 和 NAND Hash 的通用的 mtd_info 成员函数。

mtd_info 结构体在 Linux 2.6.18 内核发生了较大的变化，read_ecc()、write_ecc()成员函数被去掉，oobblock 等字段也被新的字段替代。

Flash 驱动中使用如下两个函数注册和注销 MTD 设备：

```
int add_mtd_device(struct mtd_info *mtd);
int del_mtd_device (struct mtd_info *mtd);
```

代码清单 19.2 所示的 mtd_part 结构体用于描述分区，其 mtd_info 结构体成员用于描述本分区，它会被加入到 mtd_table 中，其大部分成员由其主分区 mtd_part->master 决定，各种函数也指向主分区的相应函数，而主分区（其大小涵盖所有分区）则不作为一个 MTD 原始设备加入 mtd_table。

代码清单 19.2　mtd_part 结构体

```
struct mtd_part
{
  struct mtd_info mtd; //分区的信息（大部分由其 master 决定）
  struct mtd_info *master; //该分区的主分区
  u_int32_t offset; //该分区的偏移地址
  int index; //分区号
  struct list_head list;
};
```

mtd_partition 会在 MTD 原始设备层调用 add_mtd_partions()时传递分区信息用，这个结构体的定义如代码清单 19.3 所示。

代码清单 19.3　mtd_partition 结构体

```
struct mtd_partition
{
  char *name;             /* 标识字符串 */
  u_int32_t size;             /* 分区大小 */
  u_int32_t offset;       /* 主 MTD 空间内的偏移*/
  u_int32_t mask_flags;       /* 掩码标志 */
};
```

Flash 驱动中使用如下两个函数注册和注销分区：

```
int add_mtd_partitions(struct mtd_info *master, struct mtd_partition *parts, int nbparts);
int del_mtd_partitions(struct mtd_info *master);
```

add_mtd_partitions()会对每一个新建分区建立一个新的 mtd_part 结构体，将其加入 mtd_ partitions 中，并调用 add_mtd_device()将此分区作为 MTD 设备加入 mtd_table。成功时返回 0，如果分配 mtd_part 时内存不足，则返回-ENOMEM。

del_mtd_partitions()的作用是对于 mtd_partitions 上的每一个分区，如果它的主分区是 master（参数 master 是被删除分区的主分区），则将它从 mtd_partitions 和 mtd_table 中删除并释放掉，这个函数会调用 del_mtd_device()。

add_mtd_partitions()中新建的 mtd_part 需要依赖传入的 mtd_partition 参数对其进行初始化，如代码清单 19.4 所示。

代码清单 19.4　add_mtd_partitions()函数

```
int add_mtd_partitions(struct mtd_info *master, const struct mtd_partition
*parts, int nbparts)
{
  ...
  for (i = 0; i < nbparts; i++)
  {
    slave = kmalloc(sizeof(*slave), GFP_KERNEL);
    if (!slave)
    {
      printk("memory allocation error while creating partitions for \"%s\"\n",
          master->name);
        del_mtd_partitions(master);
        return  - ENOMEM;
      }

      memset(slave, 0, sizeof(*slave));
      list_add(&slave->list, &mtd_partitions);

      /* 设置该分区的 MTD 对象 */
      slave->mtd.type = master->type;
      slave->mtd.flags = master->flags &~parts[i].mask_flags;
      slave->mtd.size = parts[i].size;
      slave->mtd.oobblock = master->oobblock;
      slave->mtd.oobsize = master->oobsize;
```

```
25        slave->mtd.ecctype = master->ecctype;
26        slave->mtd.eccsize = master->eccsize;
27
28       slave->mtd.name = parts[i].name;
29       slave->offset = parts[i].offset;
30
31     }
32     ...
33 }
```

最后，为了使系统能支持 MTD 字符设备与块设备及 MTD 分区，在编译内核时应该包括相应的配置选项，如下所示。

```
Memory Technology Devices (MTD)  --->
 <*> Memory Technology Device (MTD) support
  [*]   MTD partitioning support
   ......
  --- User Modules And Translation Layers
  <*> Direct char device access to MTD devices
  <*> Caching block device access to MTD devices
  ......
```

19.1.3 MTD 用户空间编程

mtdchar.c 实现了字符设备接口，通过它，用户可以直接操作 Flash 设备。通过 read()、write()系统调用可以读写 Flash，通过一系列 IOCTL 命令可以获取 Flash 设备信息、擦除 Flash、读写 NAND 的 OOB、获取 OOB layout 及检查 NAND 坏块等。

代码清单 19.5 所示为 MEMGETINFO、MEMERASE、MEMREADOOB、MEMWRITEOOB、MEMGETBADBLOCK IOCRL 的例子。

代码清单 19.5 /dev/mtdX IOCTL 演示范例

```
1  mtd_oob_buf oob;
2  erase_info_user erase;
3  mtd_info_user meminfo;
4
5  //得到 MTD 设备信息
6  if (ioctl(fd, MEMGETINFO, &meminfo) != 0)
7  {
8    perror("ioctl(MEMGETINFO)");
9  }
10
11 //擦除块
12 if (ioctl(ofd, MEMERASE, &erase) != 0)
13 {
14   perror("ioctl(MEMERASE)");
```

```
  goto error;
}

//读 OOB
if (ioctl(fd, MEMREADOOB, &oob) != 0)
{
  perror("ioctl(MEMREADOOB)");
}

//写 OOB
if (ioctl(fd, MEMWRITEOOB, &oob) != 0)
{
  fprintf(stderr, "\n%s: %s: MTD writeoob failure: %s\n", exe_name, mtd_device,
      strerror(errno));
}

//检查坏块
if (blockstart != (ofs &(~meminfo.erasesize + 1)))
{
  blockstart = ofs &(~meminfo.erasesize + 1);
  if ((badblock = ioctl(fd, MEMGETBADBLOCK, &blockstart)) < 0)
  {
    perror("ioctl(MEMGETBADBLOCK)");
  }
  else if (badblock)//是坏块
  {
    ...
  }
  else//是好块
  ...
}
```

代码清单 19.6 所示的过程如下：它读取记录在一个映像文件中的针对 NAND 的数据信息，并把它写到 NAND Flash 中。代码中涉及大量 IOCTL 的调用及 NAND 的擦除和写入过程（含坏块检查），mtd-utils 中 nand_write、Flash_eraseall 等工具也是借助类似方法实现的。

代码清单 19.6　/dev/mtdX 用户空间编程综合范例

```
int main(int argc, char **argv)
{
  struct mtd_info_user meminfo;
  struct mtd_oob_buf oob;
  char  oobbuf[MAX_OOB_SIZE];
  ...

  memset(oobbuf, 0xff, sizeof(oobbuf));

  /* 打开/dev/mtdX */
  if ((fd = open(mtd_device, O_RDWR)) ==  - 1)
  {
```

```
    perror("open Flash");
    exit(1);
  }

  /* 填充 MTD 设备容量结构体 */
  if (ioctl(fd, MEMGETINFO, &meminfo) != 0)
  {
    perror("MEMGETINFO");
    close(fd);
    exit(1);
  }

  oob.length = meminfo.oobsize;
  oob.ptr = oobbuf;

  /* 打开输入文件 */
  if ((ifd = open(img, O_RDONLY)) ==  - 1)
  {
    perror("open input file");
    goto restoreoob;
  }


  imglen = lseek(ifd, 0, SEEK_END);   // 得到映像长度
  lseek(ifd, 0, SEEK_SET);

  pagelen = meminfo.oobblock + meminfo.oobsize; //一页的（数据+oob）长度
  ...

  /* 从输入文件读数据然后写入 MTD 设备 */
  while (imglen && (mtdoffset < meminfo.size))
  {
    // 在擦除块之前检查是否为坏块
    while (blockstart != (mtdoffset &(~meminfo.erasesize + 1)))
    {
      blockstart = mtdoffset &(~meminfo.erasesize + 1);
      offs = blockstart;
      baderaseblock = 0;
      if (!quiet)
        fprintf(stdout, "Writing data to block %x\n", blockstart);

      /* 检查坏块 */
      do
      {
        if ((ret = ioctl(fd, MEMGETBADBLOCK, &offs)) < 0)
        {
          perror("ioctl(MEMGETBADBLOCK)");
```

```
          goto closeall;
        }
        if (ret == 1)
        {
          baderaseblock = 1;
          if (!quiet)
            fprintf(stderr,
              "Bad block at %x, %u block(s) from %x will be skipped\n",
              (int)offs, blockalign, blockstart);
        }
        if (baderaseblock)
        {
          mtdoffset = blockstart + meminfo.erasesize;
        }
        offs += meminfo.erasesize / blockalign;
      }  while (offs < blockstart + meminfo.erasesize);
    }

    readlen = meminfo.oobblock;

    /* 从输入文件中读 page 数据 */
    if ((cnt = read(ifd, writebuf, readlen)) != readlen)
    {
      if (cnt == 0)   // EOF
        break;
      perror("File I/O error on input file");
      goto closeall;
    }

    /* 从输入文件读 OOB 数据 */
        if ((cnt = read(ifd, oobreadbuf, meminfo.oobsize)) != meminfo.oobsize)
    {
      perror("File I/O error on input file");
      goto closeall;
    }

    /* 将 OOB 数据写入设备 */
    oob.start = mtdoffset;
    if (ioctl(fd, MEMWRITEOOB, &oob) != 0)
    {
      perror("ioctl(MEMWRITEOOB)");
      goto closeall;
    }

    /* 写 page 数据 */
      if (pwrite(fd, writebuf, meminfo.oobblock, mtdoffset) !=
```

```
meminfo.oobblock)
    106     {
    107       perror("pwrite");
    108       goto closeall;
    109     }
    110     imglen -= readlen;
    111     mtdoffset += meminfo.oobblock;
    112   }
    113
    114   closeall: ...
    115   return 0;
    116 }
```

19.2 NOR Flash 驱动

图 19.3　MTD、通用 NOR Flash

在 Linux 系统中，实现了针对 cfi、jedec 等接口的通用 NOR 驱动，这一层的驱动直接面向 mtd_info 的成员函数，这使得 NOR 的芯片级驱动变得十分简单，只需要定义具体的内存映射情况结构体 map_info 并使用指定接口类型调用 do_map_ probe()。

NOR Flash 驱动的核心是定义 map_info 结构体，它指定了 NOR Flash 的基址、位宽、大小等信息以及 Flash 的读写函数，该结构体对于 NOR Flash 驱动而言至为关键，甚至 NOR Flash 驱动的代码本质上可以被认作是根据 map_info 探测芯片的过程，其定义如代码清单 19.7 所示。

代码清单 19.7　map_info 结构体

```
    1  struct map_info
    2  {
    3    char *name;
    4    unsigned long size;
    5    unsigned long phys;
    6    #define NO_XIP (-1UL)
    7
    8    void __iomem *virt; /* 虚拟地址 */
    9    void *cached;
    10
    11   int bankwidth; /* 总线宽度 */
    12
    13   #ifdef CONFIG_MTD_COMPLEX_MAPPINGS
```

```
    map_word(*read)(struct map_info *, unsigned long);
      void(*copy_from)(struct map_info *, void *, unsigned long, ssize_t);

    void(*write)(struct map_info *, const map_word, unsigned long);
    void(*copy_to)(struct map_info *, unsigned long, const void *, ssize_t);
  #endif
  /* 缓存的虚拟地址 */
  void(*inval_cache)(struct map_info *, unsigned long, ssize_t);

  void(*set_vpp)(struct map_info *, int);

  unsigned long map_priv_1;
  unsigned long map_priv_2;
  void *fldrv_priv;
  struct mtd_chip_driver *fldrv;
};
```

NOR Flash 驱动在 Linux 中的实现非常简单，如图 19.4 所示，主要的工作如下。

（1）定义 map_info 的实例，初始化其中的成员，根据目标板的情况为 name、size、bankwidth 和 phys 赋值。

19.4　NOR Flash 驱动

（2）如果 Flash 要分区，则定义 mtd_partition 数组，将实际电路板中 Flash 分区信息记录于其中。

（3）以 map_info 和探测的接口类型（如"cfi_probe"、"jedec_probe"等）为参数调用 do_map_ probe()，探测 Flash 得到 mtd_info。

do_map_probe()的函数原型为：

```
struct mtd_info *do_map_probe(const char *name, struct map_info *map);
```

第一个参数为探测的接口类型，常见的调用方法如下。

```
do_map_probe("cfi_probe",&xxx_map_info);
do_map_probe("jedec_probe",&xxx_map_info);
do_map_probe("map_rom",&xxx_map_info);
```

do_map_probe()会根据传入的参数 name 通过 get_mtd_chip_driver()得到具体的 MTD 驱动，调用与接口对应的 probe()函数探测设备，如代码清单 19.8 所示。

代码清单 19.8　do_map_probe()函数

```
1  struct mtd_info *do_map_probe(const char *name, struct map_info *map)
2  {
3   struct mtd_chip_driver *drv;
4   struct mtd_info *ret;
5
6   drv = get_mtd_chip_driver(name);/* 通过名称获得驱动 */
7
8   if (!drv && !request_module("%s", name))
9       drv = get_mtd_chip_driver(name);
10
11  if (!drv)
12      return NULL;
13
14  ret = drv->probe(map);/* 调用驱动的探测函数 */
15
16  module_put(drv->module);
17  if (ret)
18      return ret;
19
20  return NULL;
21 }
```

利用 map_info 中的配置，do_map_probe()可以自动识别支持 CFI 或 JEDEC（电子元件工业联合会）接口的 Flash 芯片，MTD 以后会自动采用适当的命令参数对 Flash 进行读写或擦除。

（4）在模块初始化时以 mtd_info 为参数调用 add_mtd_device()或以 mtd_info、mtd_partition 数组及分区数为参数调用 add_mtd_partitions()注册设备或分区。当然，在这之前可以调用 parse_mtd_partitions()查看 Flash 上是否已有分区信息，并将查看出的分区信息通过 add_mtd_partitions()注册。

（5）在模块卸载时调用第 4 行函数的“反函数”删除设备或分区。

代码清单 19.9 所示为一个最简单的 NOR Flash 驱动模板。

代码清单 19.9　NOR Flash 设备驱动模板

```
1  #define WINDOW_SIZE ...
2  #define WINDOW_ADDR ...
3  static struct map_info xxx_map = {  //map_info
```

```
 .name = "xxx Flash",
 .size = WINDOW_SIZE, //大小
 .bankwidth = 1,   //总线宽度
 .phys = WINDOW_ADDR //物理地址
};

static struct mtd_partition xxx_partitions[] = { // mtd_partition
 {
     .name = "Drive A",
     .offset = 0,   //分区的偏移地址
     .size = 0x0e0000 //分区大小
 },
 ...
};

#define NUM_PARTITIONS ARRAY_SIZE(xxx_partitions)

static struct mtd_info *mymtd;

static int __init init_xxx_map(void)
{
 int rc = 0;

 xxx_map.virt=ioremap_nocache(xxx_map.phys, xxx_map.size);//物理->虚拟地址

 if (!xxx_map.virt) {
     printk(KERN_ERR "Failed to ioremap_nocache\n");
     rc = -EIO;
     goto err2;
 }

 simple_map_init(&xxx_map);

 mymtd = do_map_probe("jedec_probe", &xxx_map);//探测 nor Flash
 if (!mymtd) {
     rc = -ENXIO;
     goto err1;
 }

 mymtd->owner = THIS_MODULE;
 add_mtd_partitions(mymtd, xxx_partitions, NUM_PARTITIONS);//添加分区信息

```

```
 return 0;

err1:
 map_destroy(mymtd);
 iounmap(xxx_map.virt);
err2:
 return rc;
}

static void __exit cleanup_xxx_map(void)
{
 if (mymtd) {
     del_mtd_partitions(mymtd);//删除分区
     map_destroy(mymtd);
 }

 if (xxx_map.virt) {
     iounmap(xxx_map.virt);
     xxx_map.virt = NULL;
 }
}
```

19.3 NAND Flash 驱动

和 NOR Flash 非常类似，如图 19.5 所示，Linux 内核在 MTD 的下层实现了通用的 NAND 驱动（主要通过 drivers/mtd/nand/nand_base.c 文件实现），因此芯片级的 NAND 驱动不再需要实现 mtd_info 中的 read()、write()、read_oob()、write_oob() 等成员函数，而主体转移到了 nand_chip 数据结构。

19.5 NAND Flash 驱动

MTD 使用 nand_chip 数据结构表示一个 NAND Flash 芯片，这个结构体中包含了关于 NAND Flash 的地址信息、读写方法、ECC 模式、硬件控制等一系列底层机制，其定义如代码清单 19.10 所示。

代码清单 19.10　nand_chip 结构体

```
struct nand_chip
{
  void __iomem *IO_ADDR_R; //读 8 根 I/O 线的地址
  void __iomem *IO_ADDR_W; //写 8 根 I/O 线的地址
```

```
5
u_char(*read_byte)(struct mtd_info *mtd); //从芯片读一个字节
void(*write_byte)(struct mtd_info *mtd, u_char byte); //向芯片写一个字节
u16(*read_word)(struct mtd_info *mtd); //从芯片读一个字
void(*write_word)(struct mtd_info *mtd, u16 word); //向芯片写一个字

void(*write_buf)(struct mtd_info *mtd, const u_char *buf, int len);
  //将缓冲区内容写入芯片
void(*read_buf)(struct mtd_info *mtd, u_char *buf, int len);
  //将芯片数据读到缓冲区
int(*verify_buf)(struct mtd_info *mtd, const u_char *buf, int len);
  //验证芯片和写入缓冲区中的数据
void(*select_chip)(struct mtd_info *mtd, int chip); //控制 CE 信号
int(*block_bad)(struct mtd_info *mtd, loff_t ofs, int getchip);
  //检查是否为坏块
int(*block_markbad)(struct mtd_info *mtd, loff_t ofs); //标志坏块
void(*hwcontrol)(struct mtd_info *mtd, int cmd); //板特定的硬件控制
int(*dev_ready)(struct mtd_info *mtd); //板特定的设备 ready/busy 信息
void(*cmdfunc)(struct mtd_info *mtd, unsigned command, int column, int
  page_addr); //命令处理函数
int(*waitfunc)(struct mtd_info *mtd, struct nand_chip *this, int state);
int(*calculate_ecc)(struct mtd_info *mtd, const u_char *dat, u_char *ecc_code)
  ; //计算 ecc
int(*correct_data)(struct mtd_info *mtd, u_char *dat, u_char *read_ecc,
  u_char *calc_ecc); //纠正数据
void(*enable_hwecc)(struct mtd_info *mtd,int mode);//板特定的硬件 ECC 使能
void(*erase_cmd)(struct mtd_info *mtd, int page); //擦除命令处理
int(*scan_bbt)(struct mtd_info *mtd); //扫描坏块
int eccmode; //板特定的 ECC 模式
int eccsize; //ECC 尺寸
int eccbytes; //ECC 字节，每 eccsize 算出 eccbytes 个校验码
```

```
int eccsteps;
int chip_delay; //芯片特定的延迟
spinlock_t chip_lock;
wait_queue_head_t wq;
nand_state_t state; //芯片状态，如输入/输出
int page_shift;
int phys_erase_shift;
int bbt_erase_shift;
int chip_shift;
u_char *data_buf; //数据 buffer
u_char *oob_buf; //oob buffer
int oobdirty;
u_char *data_poi;
unsigned int options;
int badblockpos;
int numchips;
unsigned long chipsize;
int pagemask;
int pagebuf;
struct nand_oobinfo *autooob;
uint8_t *bbt;
struct nand_bbt_descr *bbt_td;
struct nand_bbt_descr *bbt_md;
struct nand_bbt_descr *badblock_pattern;
struct nand_hw_control *controller;
void *priv;
int(*errstat)(struct mtd_info *mtd, struct nand_chip *this, int state, int
  status, int page);
};
```

与 NOR Flash 类似，由于有了 MTD 层，完成一个 NAND Flash 驱动在 Linux 中的工作量也很小，如图 19.6 所示，主要的工作如下。

（1）如果 Flash 要分区，则定义 mtd_partition 数组，将实际电路板中 Flash 分区信息记录于其中。

（2）在模块加载时分配和 nand_chip 的内存，根据目标板 NAND 控制器的特殊情况初始化 nand_chip 中的 hwcontrol()、dev_ready()、calculate_ecc()、correct_data()、read_byte()、write_byte() 等成员函数（如果不赋值会使用 nand_base.c 中的默认函数），注

意将 mtd_info 的 priv 置为 nand_chip。

如果使用软件 ECC，则不需要赋值 calculate_ecc()和 correct_data()成员，因为 NAND 核心层包含了相应的软件算法。如果使用 NAND 控制器的 ECC，应该自定义 calculate_ecc()函数，并在其中将硬件上产生的 ECC 字节返回到 ecc_code 参数。

（3）以 mtd_info 为参数调用 nand_scan()函数探测 NAND Flash 的存在。nand_scan()函数的原型为：

```
int nand_scan (struct mtd_info *mtd, int maxchips);
```

nand_scan()函数会读取 NAND 芯片 ID，并根据 mtd->priv 即 nand_chip 中的成员初始化 mtd_info。

（4）如果要分区，则以 mtd_info 和 mtd_partition 为参数调用 add_mtd_partitions()，添加分区信息。

代码清单 19.11 所示为一个简单的 NAND Flash 设备驱动模板。

代码清单 19.11　NAND Flash 设备驱动模板

```
1  #define CHIP_PHYSICAL_ADDRESS  ...
2  #define NUM_PARTITIONS 2
3  static struct mtd_partition partition_info[] =
4  {
5    {
6      .name = "Flash partition 1", .offset = 0, .size = 8 * 1024 * 1024
7    },
8    {
9      .name = "Flash partition 2", .offset = MTDPART_OFS_NEXT, .size
=
10       MTDPART_SIZ_FULL
11   } ,
12 };
13 int __init board_init(void)
14 {
15   struct nand_chip *this;
16   int err = 0;
17   /* 为 MTD 设备结构体和 nand_chip 分配内存 */
18     board_mtd = kmalloc(sizeof(struct mtd_info) + sizeof(struct
nand_chip),
19     GFP_KERNEL);
20   if (!board_mtd)
21   {
22     printk("Unable to allocate NAND MTD device structure.\n");
23     err = - ENOMEM;
24     goto out;
25   }
26   /* 初始化结构体 */
```

```
    27        memset((char*)board_mtd,  0,  sizeof(struct  mtd_info)  +
sizeof(struct nand_chip)
    28     );
    29   /* 映射物理地址 */
    30   baseaddr = (unsigned long)ioremap(CHIP_PHYSICAL_ADDRESS, 1024);
    31   if (!baseaddr)
    32   {
    33     printk("Ioremap to access NAND chip failed\n");
    34     err =  - EIO;
    35     goto out_mtd;
    36   }
    37   /* 获得私有数据（nand_chip）指针 */
    38   this = (struct nand_chip*)(&board_mtd[1]);
    39   /* 将 nand_chip 赋予 mtd_info 私有指针 */
    40   board_mtd->priv = this;
    41   /* 设置 NAND Flash 的 I/O 基地址 */
    42   this->IO_ADDR_R = baseaddr;
    43   this->IO_ADDR_W = baseaddr;
    44   /* 硬件控制函数 */
    45   this->hwcontrol = board_hwcontrol;
    46   /* 从数据手册获知命令延迟时间 */
    47   this->chip_delay = CHIP_DEPENDEND_COMMAND_DELAY;
    48   /* 初始化设备 ready 函数 */
    49   this->dev_ready = board_dev_ready;
    50   this->eccmode = NAND_ECC_SOFT;
    51   /* 扫描以确定设备的存在 */
    52   if (nand_scan(board_mtd, 1))
    53   {
    54     err =  - ENXIO;
    55     goto out_ior;
    56   }
    57   //添加分区
    58   add_mtd_partitions(board_mtd, partition_info, NUM_PARTITIONS);
    59   goto out;
    60   out_ior: iounmap((void*)baseaddr);
    61   out_mtd: kfree(board_mtd);
    62   out: return err;
    63 }
```

```
64
65 static void __exit board_cleanup(void)
66 {
67   /* 释放资源，注销设备 */
68   nand_release(board_mtd);
69   /* unmap 物理地址 */
70   iounmap((void*)baseaddr);
71   /* 释放 MTD 设备结构体 */
72   kfree(board_mtd);
73 }
74
75 /* GPIO 方式的硬件控制 */
76 static void board_hwcontrol(struct mtd_info *mtd, int cmd)
77 {
78   switch (cmd)
79   {
80     case NAND_CTL_SETCLE:
81       /* Set CLE pin high */
82       break;
83     case NAND_CTL_CLRCLE:
84       /* Set CLE pin low */
85       break;
86     case NAND_CTL_SETALE:
87       /* Set ALE pin high */
88       break;
89     case NAND_CTL_CLRALE:
90       /* Set ALE pin low */
91       break;
92     case NAND_CTL_SETNCE:
93       /* Set nCE pin low */
94       break;
95     case NAND_CTL_CLRNCE:
96       /* Set nCE pin high */
97       break;
98   }
99 }
100
101 /* 返回设备 ready 状态 */
102 static int board_dev_ready(struct mtd_info *mtd)
103 {
104 return xxx_read_ready_bit();
105 }
```

最后要强调的是，在 NAND 芯片级驱动中，如果在 nand_chip 中没有赋值，将使

用如代码清单 19.12 所示的默认分布，应该根据实际 NAND 控制器和 NAND 芯片的情况给 nand_chip 的 nand_oobinfo 成员赋值，定义 OOB 的分布。

代码清单 19.12　NAND 驱动默认的 OOB 分布

```
//页尺寸为 256，OOB 字节数为 8
static struct nand_oobinfo nand_oob_8 =
{
 .useecc = MTD_NANDECC_AUTOPLACE,
 .eccbytes = 3,
 .eccpos = {0, 1, 2},
 .oobfree = { {3, 2}, {6, 2} }
};

//页尺寸为 512，OOB 字节数为 16
static struct nand_oobinfo nand_oob_16 =
{
 .useecc = MTD_NANDECC_AUTOPLACE,
 .eccbytes = 6,
 .eccpos = {0, 1, 2, 3, 6, 7},
 .oobfree = { {8, 8} }
};

//页尺寸为 2048，OOB 字节数为 64
static struct nand_oobinfo nand_oob_64 =
{
 .useecc = MTD_NANDECC_AUTOPLACE,
 .eccbytes = 24,
 .eccpos = {
     40, 41, 42, 43, 44, 45, 46, 47,
     48, 49, 50, 51, 52, 53, 54, 55,
     56, 57, 58, 59, 60, 61, 62, 63},
 .oobfree = { {2, 38} }
};
```

nand_oobinfo 中的 useecc 定义 ECC 的放置模式，包括 MTD_NANDECC_OFF（不使用 ECC）、MTD_NANDECC_AUTOPLACE（自动放置）、MTD_NANDECC_PLACE（使用结构体内指定的放置方式）等；eccbytes 定义 ECC 字节数；eccpos 是 ECC 校验码的放置位置；oobfree 中记录还可被自由使用的 OOB 区域的开始位置和长度，它是一个数组，因此，可自由使用的 OOB 区域可以是不连续的多个小片段。

内核中包含了一个 NAND 模拟器 nandsim，使用一片内存区域模拟 NAND，在没有电路板的情况下，可以使用 nandsim 模拟 NAND 芯片。YAFFS 和 YAFFS2 中分别

包含了 nandemul 和 nandemul2k（用于模拟页大小为 2KB 的 NAND），也可以模拟 NAND。

19.4 NOR Flash 驱动实例：S3C2410 外围的 NOR Flash 驱动

其实这一节的实例有非常强的适应性，虽然节名为 S3C2410 外围的 NOR Flash 驱动，实际上由于 NOR Flash 的总线接口与 SRAM 类似，没有什么特殊的控制，因此呈现出一定的平台无关性，该 NOR Flash 驱动只需稍微修改一下就可以应用在别的平台上。

代码清单 19.13　S3C2410 外围的 NOR Flash 驱动

```
1   #define WINDOW_ADDR 0x01000000      /* NOR Flash 物理地址 */
2   #define WINDOW_SIZE 0x800000        /* NOR Flash 大小 */
3   #define BUSWIDTH    2
4   /* 探测的接口类型，可以是"cfi_probe", "jedec_probe", "map_rom", NULL
*/
5   #define PROBETYPES { "cfi_probe", NULL }
6
7   #define MSG_PREFIX "S3C2410-NOR:"       /* printk 的前缀*/
8   #define MTDID     "s3c2410-nor"     /* MTD 驱动*/
9
10  static struct mtd_info *mymtd;
11
12  struct map_info s3c2410nor_map =  // map_info
13  {
14    .name = "NOR Flash on S3C2410",
15    .size = WINDOW_SIZE,
16    .bankwidth = BUSWIDTH,
17    .phys = WINDOW_ADDR,
18  };
19
20  #ifdef CONFIG_MTD_PARTITIONS
21    /* MTD 分区信息  */
22    static struct mtd_partition static_partitions[] =
23    {
24      {
25            .name  =  "BootLoader",  .size  =  0x040000,  .offset  =  0x0
//bootloader 存放的区域
```

```
    26      } ,
    27      {
    28        .name = "Kernel", .size = 0x0100000, .offset = 0x40000 //内核映像存放的区域
    29      }
    30      ,
    31      {
    32          .name = "RamDisk", .size = 0x400000, .offset = 0x140000 //RamDisk 存放的区域
    33      }
    34      ,
    35      {
    36        .name = "cramfs(2MB)", .size = 0x200000, .offset = 0x540000 //只读的 cramfs 区域
    37      }
    38      ,
    39      {
    40        .name = "jffs2(0.75MB)", .size = 0xc0000, .offset = 0x740000 //可读写的 jffs2 区域
    41      }
    42      ,
    43    };
    44  #endif
    45
    46  static int mtd_parts_nb = 0;
    47  static struct mtd_partition *mtd_parts = 0;
    48
    49  int __init init_s3c2410nor(void)
    50  {
    51    static const char *rom_probe_types[] = PROBETYPES;
    52    const char **type;
    53    const char *part_type = 0;
    54
    55    printk(KERN_NOTICE MSG_PREFIX "0x%08x at 0x%08x\n", WINDOW_SIZE, WINDOW_ADDR);
    56    s3c2410nor_map.virt = ioremap(WINDOW_ADDR, WINDOW_SIZE);//物理->虚拟地址
    57
```

```
if (!s3c2410nor_map.virt)
{
  printk(MSG_PREFIX "failed to ioremap\n");
  return  - EIO;
}

simple_map_init(&s3c2410nor_map);

mymtd = 0;
type = rom_probe_types;
for (; !mymtd &&  *type; type++)
{
  mymtd = do_map_probe(*type, &s3c2410nor_map);//探测NOR Flash
}
if (mymtd)
{
  mymtd->owner = THIS_MODULE;

  #ifdef CONFIG_MTD_PARTITIONS
     mtd_parts_nb = parse_mtd_partitions(mymtd, NULL, &mtd_parts, MTDID);//探测分区信息
    if (mtd_parts_nb > 0)
      part_type = "detected";

    if (mtd_parts_nb == 0) //未探测到，使用数组定义的分区信息
    {
      mtd_parts = static_partitions;
      mtd_parts_nb = ARRAY_SIZE(static_partitions);
      part_type = "static";
    }
  #endif
  add_mtd_device(mymtd);
  if (mtd_parts_nb == 0)
          printk(KERN_NOTICE  MSG_PREFIX  "no  partition  info available\n");
  else
  {
         printk(KERN_NOTICE  MSG_PREFIX  "using  %s  partition definition\n",
      part_type);
```

```
95        add_mtd_partitions(mymtd, mtd_parts, mtd_parts_nb);//添加分区信息
96      }
97      return 0;
98    }
99
100   iounmap((void*)s3c2410nor_map.virt);
101   return  - ENXIO;
102 }
103
104 static void __exit cleanup_s3c2410nor(void)
105 {
106   if (mymtd)
107   {
108     del_mtd_partitions(mymtd);  //删除分区
109     del_mtd_device(mymtd);      //删除设备
110     map_destroy(mymtd);
111   }
112   if (s3c2410nor_map.virt)
113   {
114     iounmap((void*)s3c2410nor_map.virt);
115     s3c2410nor_map.virt = 0;
116   }
117 }
```

上述代码第 77 行调用的 parse_mtd_partitions()用于解析 MTD 分区信息，从第 76～95 行代码可以看出，当解析成功时，应该添加解析出来的分区表；否则，使用驱动中自定义的分区表。最常见的 MTD 分区解析器是命令行解析，即解析在 Linux 启动命令行中通过“mtdparts=”传入的 MTD 分区信息。

通过查看/proc/mtd 文件可以获知系统中包含的 MTD 设备（分区），如下所示：

```
dev :   size     erasesize  name
mtd0: 00040000  00020000  "bootloader"
mtd1: 002c0000  00020000  "kernel"
mtd2: 01000000  00020000  "rootfs"
...
```

19.5 NAND Flash 驱动实例：S3C2410 外围的 NAND Flash 驱动

19.5.1 S3C2410 NAND 控制器硬件描述

S3C2410 处理器集成了一个 NAND 控制器，它提供如下引脚。

- D[7:0]：数据/命令/地址 I/O 端口（数据、命令、地址复用）。
- CLE：命令锁存使能（数据线上 NAND Flash 命令有效，输出）。
- ALE：地址锁存使能（数据线上 NAND Flash 地址有效，输出）。
- nFCE：NAND Flash 片选（输出）。
- nFRE：NAND Flash 读使能（输出）。
- nFWE：NAND Flash 写使能（输出）。
- R/nB：NAND Flash 准备好/忙（输入）。
- NCON：输入，NAND Flash 内存地址步长选择，0：表示 3 步长地址，1：表示 4 步长地址（NAND Flash 中地址要通过 d[7:0]送多次，每送一次就为一步长）。

S3C2410 对 NAND Flash 的操作通过 NFCONF、NFCMD、NFADDR、NFDATA、NFSTAT 和 NFECC 这 6 个寄存器来完成。

- NFCONF：NAND Flash 配置寄存器，用于使能 NAND Flash 控制器、初始化 ECC、并设置 NAND Flash 片选信号 nFCE 为 1（即不选中）。
- NFCMD：NAND Flash 命令寄存器，对于 page 大小不一样的不同型号的 NAND，其操作命令可能不一样。本节的例子芯片为 K9F1208U0M，在其数据手册中该芯片命令集的有关描述。
- NFADDR：NAND Flash 地址寄存器。
- NFDATA：NAND Flash 数据寄存器，8 位。
- NFSTAT：NAND Flash 状态寄存器，位 0 如果为 0 表示设备忙，否则表示设备准备好。
- NFECC：NAND Flash 校验寄存器。S3C2410 NAND 控制器内置的 ECC 生成模块执行以下任务：当写入数据时，ECC 生成模块产生一个 ECC 码；当读数据时，ECC 生成模块产生一个 ECC 码，驱动中可以将该 ECC 码与原先存入 OOB 中的校验码进行比较。

通过 S3C2410 的 NAND 控制器访问 NAND Flash 的一般流程如下。

（1）设置 NAND Flash 配置寄存器 NFCONF。

（2）在 NFCMD 寄存器中写入 NAND Flash 命令。

（3）在 NFADDR 寄存器中写入地址。

（4）读/写数据，通过 NFSTAT 寄存器检查 NAND Flash 状态，在读/写操作后 R/nB 信号应该被检查。

19.5.2 nand_chip 初始化和成员函数

nand_chip 是 NAND Flash 驱动的核心数据结构，这个结构体中的成员直接对应着 NAND Flash 的底层操作，针对具体的 NAND 控制器情况，本驱动中初始化了 write_buf()、read_buf()、select_chip()、chip_delay()及几个 ECC 相关的成员函数。代码清单 19.14 所示为 S3C2410 外围 NAND Flash 驱动的 nand_chip 初始化及其成员函数的实现。

代码清单 19.14　S3C2410 NAND 驱动的 nand_chip 初始化

```
1   static void s3c2410_nand_init_chip(struct s3c2410_nand_info *info,
```

```
struct
    s3c2410_nand_mtd *nmtd, struct s3c2410_nand_set *set)
  {
    struct nand_chip *chip = &nmtd->chip;

    /* 初始化 nand_chip */
    chip->IO_ADDR_R = info->regs + S3C2410_NFDATA;
    chip->IO_ADDR_W = info->regs + S3C2410_NFDATA;
    chip->hwcontrol = s3c2410_nand_hwcontrol;
    chip->dev_ready = s3c2410_nand_devready;
    chip->write_buf = s3c2410_nand_write_buf;
    chip->read_buf = s3c2410_nand_read_buf;
    chip->select_chip = s3c2410_nand_select_chip;
    chip->chip_delay = 50;
    chip->priv = nmtd;
    chip->options = 0;
    chip->controller = &info->controller;

    nmtd->info = info;
    nmtd->mtd.priv = chip;
    nmtd->set = set;

    if (hardware_ecc)
    //如果采用硬件 ECC
    {
      chip->correct_data = s3c2410_nand_correct_data;
      chip->enable_hwecc = s3c2410_nand_enable_hwecc;
      chip->calculate_ecc = s3c2410_nand_calculate_ecc;
      chip->eccmode = NAND_ECC_HW3_512; //512 字节产生 3 字节的 ECC 码
      chip->autooob = &nand_hw_eccoob;
    }
    else
    //使用软件 ECC
    {
      chip->eccmode = NAND_ECC_SOFT;
    }
  }

```

```
/* 读 NAND 中的数据到缓冲区 */
static void s3c2410_nand_read_buf(struct mtd_info *mtd, u_char *buf, int len)
{
  struct nand_chip *this = mtd->priv;
  readsb(this->IO_ADDR_R, buf, len);
}

/* 写缓冲区的数据到 NAND Flash 中 */
 static void s3c2410_nand_write_buf(struct mtd_info *mtd, const u_char *buf, int
  len)
{
  struct nand_chip *this = mtd->priv;
  writesb(this->IO_ADDR_W, buf, len);
}

/* 读取 ECC */
static int s3c2410_nand_calculate_ecc(struct mtd_info *mtd, const u_char *dat,
  u_char *ecc_code)
{
  struct s3c2410_nand_info *info = s3c2410_nand_mtd_toinfo(mtd);

  ecc_code[0] = readb(info->regs + S3C2410_NFECC + 0);
  ecc_code[1] = readb(info->regs + S3C2410_NFECC + 1);
  ecc_code[2] = readb(info->regs + S3C2410_NFECC + 2);

      pr_debug("calculate_ecc:  returning  ecc  %02x,%02x,%02x\n", ecc_code[0],
    ecc_code[1], ecc_code[2]);

  return 0;
}

/* 数据纠正 */
static int s3c2410_nand_correct_data(struct mtd_info *mtd, u_char *dat,
  u_char*read_ecc, u_char *calc_ecc)
{
```

```
    74    pr_debug("s3c2410_nand_correct_data(%p,%p,%p,%p)\n", mtd, dat,
read_ecc,
    75      calc_ecc);
    76
    77    pr_debug("eccs: read %02x,%02x,%02x vs calc %02x,%02x,%02x\n",
    78    read_ecc[0],read_ecc[1], read_ecc[2], calc_ecc[0], calc_ecc[1],
calc_ecc[2]);
    79
    80    if (read_ecc[0]== calc_ecc[0] && read_ecc[1] == calc_ecc[1] &&
read_ecc[2]
    81      == calc_ecc[2]) //ECC 码一致
    82      return 0;
    83
    84
    85    return  - 1;
    86  }
    87
    88  /* NAND Flash 准备好 */
    89  static int s3c2410_nand_devready(struct mtd_info *mtd)
    90  {
    91    struct s3c2410_nand_info *info = s3c2410_nand_mtd_toinfo(mtd);
    92    return readb(info->regs + S3C2410_NFSTAT) &S3C2410_NFSTAT_BUSY;
    93  }
    94
    95  /* 选中芯片 */
    96   static void s3c2410_nand_select_chip(struct mtd_info *mtd, int
chip)
    97  {
    98    struct s3c2410_nand_info *info;
    99    struct s3c2410_nand_mtd *nmtd;
    100   struct nand_chip *this = mtd->priv;
    101   void __iomem *reg;
    102   unsigned long cur;
    103   unsigned long bit;
    104
    105   nmtd = this->priv;
    106   info = nmtd->info;
    107
```

```
  bit = S3C2410_NFCONF_nFCE;
  reg = info->regs + S3C2410_NFCONF;

  cur = readl(reg);

  if (chip ==  - 1)   //不选中 NAND 芯片
  {
    cur |= bit; //nFCE 位置 1
  }
  else   //选中 1 个 NAND
  {
    //NAND 序号越界
    if (nmtd->set != NULL && chip > nmtd->set->nr_chips)
    {
      printk(KERN_ERR PFX "chip %d out of range\n", chip);
      return ;
    }

    if (info->platform != NULL)
    {
      if (info->platform->select_chip != NULL)
        (info->platform->select_chip)(nmtd->set, chip);
    }

    cur &= ~bit; //nFCE 位置 0
  }

  writel(cur, reg); //写 NFCONF 寄存器
}

/* NAND 硬件控制 */
static void s3c2410_nand_hwcontrol(struct mtd_info *mtd, int cmd, unsigned int
  ctrl)
{
  struct s3c2410_nand_info *info = s3c2410_nand_mtd_toinfo(mtd);

  if (cmd == NAND_CMD_NONE) //不进行任何传输
    return ;

  if (ctrl &NAND_CLE)     //如果是发送命令
    writeb(cmd, info->regs + S3C2410_NFCMD);
  else    //传送地址
    writeb(cmd, info->regs + S3C2410_NFADDR);
}

/* 使能硬件 ECC 产生器 */
static void s3c2410_nand_enable_hwecc(struct mtd_info *mtd, int mode)
{
  struct s3c2410_nand_info *info = s3c2410_nand_mtd_toinfo(mtd);
  unsigned long ctrl;

```

```
159   ctrl = readl(info->regs + S3C2410_NFCONF);
160   ctrl |= S3C2410_NFCONF_INITECC; //ECC 使能为置 1
161   writel(ctrl, info->regs + S3C2410_NFCONF);
162 }
```

当使用硬件 ECC 的时候，从上述代码第 30 行可以看出，NAND Flash 驱动将使用如代码清单 19.15 所示的 OOB 分布，它将 3 个字节的 ECC 码放置在 OOB 的第 0、1、2 字节处。

代码清单 19.15　S3C2410 NAND 驱动采用硬件 ECC 时的 OOB 分布

```
1 static struct nand_oobinfo nand_hw_eccoob =
2 {
3   .useecc     = MTD_NANDECC_AUTOPLACE,
4   .eccbytes   = 3,//ECC 字节数为 3
5   .eccpos     = {0, 1, 2 },//ECC 放置位置在 0、1、2
6   .oobfree    = { {8, 8} } //可自由使用的 OOB 区域
7 };
```

19.5.3　NAND 设备驱动初始化与释放

在 Linux 2.6 内核中，S3C2410 的 NAND 被注册为一个平台设置，因此，当 S3C2410 NAND 驱动模块加载调用 platform_driver_register(&s3c2410_nand_driver)注册平台驱动时，其对应的 platform_driver 结构体的 s3c24xx_nand_probe()成员函数将被调用。在 s3c24xx_nand_probe() 函数中，将调用前述的 s3c2410_nand_init_chip() 初始化 nand_chip，通过 nand_scan() 扫描到 NAND Flash 存储器后，调用 s3c2410_nand_add_partition()添加 MTD 分区和设备信息。

同样地，当 NAND 驱动模块卸载调用 platform_driver_unregister(&s3c2410_nand_driver) 移除平台驱动时，其对应的 platform_driver 结构体的 s3c2410_nand_remove 成员函数将被调用，该函数中会删除 MTD 设备和分区信息。

代码清单 19.16 所示为 NAND 设备驱动的初始化和释放的代码。经过 s3c24xx_nand_probe()的初始化之后，用户即可访问对应的 NAND 设备，经过 s3c2410_nand_remove()的释放，NAND Flash 设备和分区信息将被移除。

代码清单 19.16　S3C2410 NAND 驱动的初始化和释放函数

```
1  static int s3c24xx_nand_probe(struct platform_device *pdev, enum
s3c_cpu_type
2    cpu_type)
3  {
4    struct s3c2410_platform_nand *plat = to_nand_plat(pdev);
5    struct s3c2410_nand_info *info;
6    struct s3c2410_nand_mtd *nmtd;
```

```
7    struct s3c2410_nand_set *sets;
8    struct resource *res;
9    int err = 0;
10   int size;
11   int nr_sets;
12   int setno;
13
14   pr_debug("s3c2410_nand_probe(%p)\n", pdev);
15
16   info = kmalloc(sizeof(*info), GFP_KERNEL);
17   ...
18   platform_set_drvdata(pdev, info);
19
20   spin_lock_init(&info->controller.lock);
21   init_waitqueue_head(&info->controller.wq);
22
23   /* 获得时钟源并使能 */
24   info->clk = clk_get(&pdev->dev, "nand");
25   ...
26   clk_enable(info->clk);
27
28   /* 分配并映射资源 */
29   res = pdev->resource;
30   size = res->end - res->start + 1; //I/O 内存大小
31   info->area= request_mem_region(res->start, size, pdev->name);//
申请 I/O 内存
32   ...
33   info->device = &pdev->dev;
34   info->platform = plat;
35   info->regs = ioremap(res->start, size); //寄存器地址 ioremap
36   info->cpu_type = cpu_type;
37   ...
38   dev_dbg(&pdev->dev, "mapped registers at %p\n", info->regs);
39
40   /* 初始化硬件 */
41   err = s3c2410_nand_inithw(info, pdev);
42   if (err != 0)
43     goto exit_error;
44
45   sets = (plat != NULL) ? plat->sets: NULL;
```

```
nr_sets = (plat != NULL) ? plat->nr_sets: 1;

info->mtd_count = nr_sets;

/* 分配 s3c2410 nand_info 中 s3c2410_nand_mtd 的内存*/
size = nr_sets * sizeof(*info->mtds);
info->mtds = kmalloc(size, GFP_KERNEL);
...

/* 初始化所有可能存在的芯片 */
nmtd = info->mtds;
for (setno = 0; setno < nr_sets; setno++, nmtd++)
{
  pr_debug("initialising set %d (%p, info %p)\n", setno, nmtd, info);

  s3c2410_nand_init_chip(info, nmtd, sets);
  nmtd->scan_res = nand_scan(&nmtd->mtd, (sets) ? sets->nr_chips: 1);//扫描Flash,初始化mtd_info
  if (nmtd->scan_res == 0)
  {
    s3c2410_nand_add_partition(info, nmtd, sets); //添加分区信息
  }
  if (sets != NULL)
    sets++;
}

if (allow_clk_stop(info))
{
  dev_info(&pdev->dev, "clock idle support enabled\n");
  clk_disable(info->clk);
}

pr_debug("initialised ok\n");
return 0;

exit_error: s3c2410_nand_remove(pdev);

```

```
  if (err == 0)
    err =  - EINVAL;
  return err;
}

static int s3c2410_nand_remove(struct platform_device *pdev)
{
  struct s3c2410_nand_info *info = to_nand_info(pdev);

  platform_set_drvdata(pdev, NULL);

  if (info == NULL)
    return 0;

  /* 释放所有的MTD设备和分区  */

  if (info->mtds != NULL)
  {
    struct s3c2410_nand_mtd *ptr = info->mtds;
    int mtdno;
    for (mtdno = 0; mtdno < info->mtd_count; mtdno++, ptr++)
    {
      pr_debug("releasing mtd %d (%p)\n", mtdno, ptr);
      nand_release(&ptr->mtd);
    }
    kfree(info->mtds);
  }

  /* 释放公用资源 */
  if (info->clk != NULL && !IS_ERR(info->clk))
  {
    if (!allow_clk_stop(info))
      clk_disable(info->clk);
    clk_put(info->clk);
  }

  if (info->regs != NULL)
  {
    iounmap(info->regs);
    info->regs = NULL;
```

```
  }
  if (info->area != NULL)
  {
    release_resource(info->area);
    kfree(info->area);
    info->area = NULL;
  }
  kfree(info);

  return 0;
}

#ifdef CONFIG_MTD_PARTITIONS //如果配置了 MTD 分区支持
static int s3c2410_nand_add_partition(struct s3c2410_nand_info *info,
                    struct s3c2410_nand_mtd *mtd,
                    struct s3c2410_nand_set *set)
{
  if (set == NULL)
     return add_mtd_device(&mtd->mtd); //添加 MTD 设备

  if (set->nr_partitions > 0 && set->partitions != NULL) {
     return add_mtd_partitions(&mtd->mtd, set->partitions,
              set->nr_partitions);  //添加分区
  }
  return add_mtd_device(&mtd->mtd);
}
#else //内核没有配置 MTD 分区支持
static int s3c2410_nand_add_partition(struct s3c2410_nand_info *info,
               struct s3c2410_nand_mtd *mtd,
               struct s3c2410_nand_set *set)
{
  return add_mtd_device(&mtd->mtd);//添加 MTD 设备
}
#endif
```

19.6 Flash 文件系统的建立

19.6.1 Flash 转换层

图 19.7 FTL 和 NFTL

由于无法重复地在 Flash 的同一块存储位置做写入操作（必须事先擦除该块后才能再写入），因此一般在硬盘上使用的文件系统，如 FAT16、FAT32、NTFS、Ext2 等将无法直接用在 Flash 上，为了沿用这些文件系统，则必须透过一层转换层（Translation Layer）来将逻辑块地址（Logical Block Address）对应到 Flash 存储器的物理位置，使系统能把 Flash 当作普通的硬盘一样处理，我们称这层为 FTL（Flash Translation Layer）。FTL 应用于 NOR Flash，而 NFTL 则应用于 NAND Flash，如图 19.7 所示。

一个闪存转换层的最简单的实现就是将模拟的块设备一对一地映射到闪存上。举例来说，当上层的文件系统要写一个块设备的扇区时，闪存转换层要做下面的操作来完成这个写请求。

（1）将这个扇区所在擦除块的数据读到内存中，放在缓存中。
（2）将缓存中与这个扇区对应的内容用新的内容替换。
（3）对该擦除块执行擦除操作。
（4）将缓冲中的数据写回该擦除块。

这种实现方式的缺点如下。

- 效率低，对一个扇区的更新要重写整个擦除块上的数据，造成数据带宽很大的浪费。可行的办法是只有当文件系统的写请求超过了一个擦除块的边界的时候，才去执行对闪存的擦除、写回操作（这种更新方式也叫 out-of-place）。
- 没有提供磨损平衡，那些被频繁更新的数据所在擦除块将首先变成坏块。
- 非常不安全，很容易引起数据的丢失。如果在上面的第（3）步和第（4）步之间发生了突然掉电，那么整个擦除块中的数据就全部丢失了。

为了解决上面这种实现方式的问题，闪存转换层不能只是简单地实现块设备与闪存的一一映射，它还需要将模拟块设备的扇区存储在闪存的不同位置，并且维持扇区到闪存的映射关系。而为了进行垃圾回收（Garbage Collection），闪存转换层必须能理解上层文件系统的语义。这样实现导致的最大问题就是效率不高，具体来说，闪存转换层为了能理解上层文件系统的语义，必须对文件系统的每个写请求进行解析，因此导致写操作的性能下降。另外，从软件的架构上来讲，要求文件系统下面的一层去理解文件系统的语义，也不太合理。因此，在 Flash 上，应尽可能地避免使用传统的依赖闪存转换层的文件系统，最好应采用专门的针对 Flash 的文件系统。

19.6.2 CramFS

在嵌入式 Linux 环境中，许多人会采用 RAMDISK 来储存文件系统的内容，RAMDISK 的含义是在启动时，把一部份内存虚拟成磁盘，并且把之前准备好的文件系统映像文件解压缩到该 RAMDISK 环境中。假设压缩后的文件系统映像为 8MB，存放于 Flash，解压缩后为 16MB，如果采用 RAMDISK，将需要 8MB 的 Flash 和 16MB

的 RAM 空间，而采用 CramFS 后，就不再需要消耗 16MB 的 RAM 空间。

CramFS 是 Linus Torvalds 参与开发的文件系统，在 linux/fs/cramfs 中可以找到 CramFS 的源代码。CramFS 是一种压缩的只读文件系统，当浏览 Flash 中的目录或读取文件时，CramFS 文件系统会动态地计算出压缩后的数据所储存的位置，并实时地解压缩到内存中，对于用户来说，使用 CramFS 与 RAMDISK 感觉不出使用上的差异性。

CramFS 工具的下载地址为 http://sourceforge.net/projects/cramfs/，通过如下命令可以创建 CramFS 文件系统映像：

```
mkcramfs my_cramfs/ cramfs.img （my_cramfs 是我们要创建映像的目录）
```

如下命令将生成的 cramfs.img 映像复制到 Flash 的第一个分区并 mount 到/mnt/nor 目录：

```
cp cramfs.img /dev/mtd1
mount -cramfs /dev/mtdblock1 /mnt/nor
```

很多时候，工程中需要基于已有的文件系统映像添加、删除一些文件后建立新的文件系统映像，这时候并不需要完全重新操作，可用如下的方法。

（1）将映像以 loop 方式挂载到某目录。

```
mkdir tmpdir
mount rootfs.cramfs tmpdir -o loop
cd tmpdir
```

（2）压缩被挂载的文件系统。

```
tar -cvf ../rootfs.tar ./     将 tmpdir 中的内容打包放在其父目录下
umount tmpdir
```

（3）解压缩文件系统到新目录。

```
mkdir rootfs
tar -xvf rootfs.tar -C rootfs
```

（4）修改新目录（这里是 rootfs）中的内容，以符合新的需要。

（5）重新创建映像文件。

```
mkcramfs rootfs rootfs.cramfs
```

19.6.3 JFFS/JFFS2

JFFS 是由瑞典 Axis Communications AB 公司开发的，于 1999 年年末基于 GNU GPL 发布的文件系统。最初的发布版本基于 Linux 2.0，后来 Red Hat 将它移植到 Linux 2.2，在使用的过程中，JFFS 设计中的局限被不断地暴露出来。于是在 2001 年年初，Red Hat 决定实现一个新的 JFFS2（http://www.infradead.org）。

JFFS2 是一个日志结构（log-structured）的文件系统，它在闪存上顺序地存储包含数据和原数据（meta-data）的节点。JFFS2 的日志结构存储方式使得它能对闪存进行 out-of-place 更新，而不是磁盘所采用的 in-place 更新方式。它提供的垃圾回收机制，使得我们不需要马上对擦写越界的块进行擦写，而只需要对其设置一个标志，标明为“脏”块。当可用的块数不足时，垃圾回收机制才开始回收这些节点。同时，由于 JFFS2 基于日志结构，在意外掉电后仍然可以保持数据的完整性，而不会丢失数据。因此，JFFS2 成为了目前 Flash 上应用最广泛的文件系统。

然而，JFFS2 挂载时需要扫描整块 Flash 以确定节点的合法性以及建立必要的数据结构，这使得 JFFS2 挂载时间比较长。又由于 JFFS2 将节点信息保存在内存中，使得它所占用的内存量和节点数目成正比。再者，由于 JFFS2 通过随机方式来实现磨损平衡，它不能保证磨损平衡的确定性。因此，人们提出了 JFFS3，它就是为解决 JFFS2 的这些缺陷而设计的。

和 CramFS 一样，也存在一个制作 JFFS2 文件系统的工具 mkfs.jffs2（包含在 mtd-utils 中），执行如下命令即可生成所要的映象：

```
./mkfs.jffs2 -d my_jffs2/  -o jffs2.img （my_jffs2是我们要制作映像的目录）
```

使用 mkfs.jffs2 制作映像的时候，要注意指定正确的擦除块打下和页面大小，对于 NAND Flash，应使用“-n”去掉生成映像中的 clean marker。

接下来将 jffs2.img 复制到 Flash 第 1 个分区（复制到 MTD 字符设备），如下所示：

```
cp jffs2.img /dev/mtd1
```

之后，就可以将对应的块设备 mount 到 Linux 的目录了，如下所示：

```
mount -t jffs2 /dev/mtdblock1 /mnt/nor
```

对于 NAND Flash，应使用 mtd-utils 中的 nandwrite 工具进行 jffs2 映像向 NAND Flash 的烧录，在烧录前可以使用 Flash_eraseall 擦除 Flash，并在 OOB 区域加上 JFFS2 需要的 clean marker。

mtd-utils 的下载地址为：

ftp://ftp.infradead.org/pub/mtd-utils/mtd-utils-1.0.0.tar.gz。

华清远见

19.6.4 YAFFS/YAFFS2

YAFFS（Yet Another Flash File System，http://www.yaffs.net）文件系统是专门针对 NAND 闪存设计的嵌入式文件系统，目前有 YAFFS 和 YAFFS2 两个版本，两个版本的主要区别之一在于 YAFFS2 能够更好地支持大容量的 NAND Flash 芯片，而前者只针对页大小为 512 字节的 NAND。

YAFFS 文件系统有些类似于 JFFS/JFFS2 文件系统，与之不同的是 JFFS1/2 文件系统最初是针对 NOR Flash 的应用场合设计的，而 NOR Flash 和 NAND Flash 本质上有较大的区别，所以尽管 JFFS1/2 文件系统也能应用于 NAND Flash，但由于它在内存占用和启动时间方面针对 NOR 的特性做了一些取舍，所以对 NAND 来说通常并不是最优的方案。NAND 上的每一页数据都有额外的空间用来存储附加信息，YAFFS 正好利用了该空间中一部分来存储文件系统相关的内容。

YAFFS 和 JFFS 都提供了写均衡、垃圾收集等底层操作，它们的不同之处如下。

- JFFS 是一种日志文件系统，通过日志机制保证文件系统的稳定性。YAFFS 仅仅借鉴了日志系统的思想，不提供日志机能，所以稳定性不如 JFFS，但是资源占用少。
- JFFS 中使用多级链表管理需要回收的脏块，并且使用系统生成伪随机变量决定要回收的块，通过这种方法能提供较好的写均衡，在 YAFFS 中是从头到尾对块搜索，所以在垃圾收集上 JFFS 的速度慢，但是能延长 NAND 的寿命。
- JFFS 支持文件压缩，适合存储容量较小的系统；YAFFS 不支持压缩，更适合存储容量较大的系统。
- YAFFS 还带有 NAND 芯片驱动，并为嵌入式系统提供了直接访问文件系统的 API，用户可以不使用 Linux 中的 MTD 和 VFS，直接对文件进行操作（如图 19.8）。尽管如此，NAND Flash 大多还是采用 MTD+YAFFS 的模式。

图 19.8 JFFS2 和 YAFFS

- YAFFS 使用 OOB 组织文件结构信息，而 JFFS 直接将节点信息保存在 NAND 数据区域里面，因此 YAFFS 在 mount 时只需读取 OOB，其 mount 时间远小于 JFFS。

在 Linux 2.6 内核下，YAFFS 文件系统的移植非常简单，主要包括以下工作。

（1）复制 YAFFS 源代码到 Linux 源码树。

只需要在内核中建立 YAFFS 目录 fs/yaffs，并把下载的 YAFFS 代码复制到该目录下面（YAFFS 代码包括 yaffs_ecc.c、yaffs_fileem.c、yaffs_fs.c、yaffs_guts.c、yaffs_mtdif.c、yaffs_ramem.c），下载的 YAFFS 源代码已包含了 Kconfig 和 Makefile，因此只需要修改 fs 目录下的 Kconfig 和 Makefile 并引用 fs/yaffs 中的对应文件即可，方法是：在 fs/Kconfig 中增加 source "fs/yaffs/Kconfig"；在 fs/Makefile 中增加 obj-$(CONFIG_YAFFS_FS) += yaffs/。

（2）配置内核编译选项。

在采用 make menuconfig 等方式配置内核编译选项时，除了应该选中 MTD 系统及目标板上 Flash 的驱动以外，也必须选中对 YAFFS 的支持，如下所示：

```
File systems  --->
 Miscellaneous filesystems  --->
  <*> Yet Another Flash Filing System(YAFFS) file system support
  [*]   NAND mtd support
  [*]   Use ECC functions of the generic MTD-NAND driver
  [*]   Use Linux file caching layer
  [*]   Turn off debug chunk erase check
  [*]   Cache short names in RAM
```

（3）挂载 YAFFS。

YAFFS 源代码包的 utils 目录下包含了 mkyaffs 工具，可以用它格式化 Flash，例如运行 mkyaffs /dev/mtd3 将用 YAFFS 文件系统格式化 NAND 的第 3 个分区，之后我们可以运行如下命令挂载 YAFFS：

```
mount –t yaffs /dev/mtdblock3 /mnt/nand
```

此后，对/mnt/nand 的操作就是对/dev/mtdblock3 的操作。

如果运行如下命令将根文件系统复制到/mnt/Flash0：

```
mount –t yaffs /dev/mtdblock3 /mnt/nand
cp (our_rootfs) /mnt/Flash0
umount /mnt/nand
```

则重新启动，并修改 Linux 启动参数中 root 为（仅仅是举例，具体系统的启动命令行很可能会有改变）：

```
param set linux_cmd_line "noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttyS0"
```

之后就可以直接以/dev/mtdblock3 中的根文件系统启动了。

YAFFS2 的移植方法与 YAFFS 类似，而且，目前我们已经没有必要再移植 YAFFS 了，因为 YAFFS2 的源码直接包含了对 YAFFS 的支持。

YAFFS 源代码的下载地址为：

http://www.aleph1.co.uk/cgi-bin/viewcvs.cgi/yaffs.tar.gz?view=tar

YAFFS2 源代码的下载地址为：

http://www.aleph1.co.uk/cgi-bin/viewcvs.cgi/yaffs2.tar.gz?view=tar

YAFFS2 源代码包的 utils 目录下包含了 mkyaffsimage、mkyaffs2image 工具的源代码，编译即可生成 mkyaffsimage、mkyaffs2image 工具。YAFFS 不支持大页的 NAND Flash，一般只用于页大小为 512Byte 的 NAND Flash，对于页大小为 2KB 的 NAND Flash 而言，只能使用 YAFFS2。

运行“mkyaffsimage dir imagename”可以制作出 YAFFS 文件系统的镜像。需要注意的是，使用 mkyaffsimage 制作出来的 YAFFS 映像文件与通常的文件系统的映像文件不同，因为在 image 文件里除了以 512 字节为单位的一个页的数据外，同时紧跟在后还包括了 16 字节为单位的 NAND OOB 数据，因此，YAFFS 映像的下载工具必须将映像中的额外数据写入到 NAND 的 OOB 中。

nandwrite 工具“名义上”可以支持 YAFFS 映像的烧录，但是考虑到实际上 NAND ECC 模式和分布的不确定性，而 mkyaffsimage 生成的映像采用固定的 OOB 区域分布，因此，为了完全支持自适应的 YAFFS 映像烧录，必须在烧录工具中先解析 NAND 驱动的 OOB layout，再将记录在 YAFFS 映像中的 OOB 数据转换为适合于在系统中 NAND 存放的形式。

对于各种文件系统而言，如果想在启动过程中将 Flash 的 xxxfs 文件系统分区挂载到某个目录，可以通过修改启动的 rc 脚本或/etc/fstab 来完成，需在启动 rc 脚本中增加“mount －t xxxfs /dev/mtdblock3 /mnt/Flash0”类似语句，或在 fstab 中增加“/dev/mtdblock3 /mnt/Flash0 xxxfs defaults 0 0”类似语句。

19.7 总结

本章主要讲解了 Linux 系统中 MTD 系统的层次和接口，NOR 和 NAND Flash 驱动的设计方法及如何在其上建立 Flash 文件系统。

由于引入了 MTD 系统以及 MTD 下层的通用 NOR 和 NAND 驱动，Linux 中 NOR 和 NAND Flash 芯片级驱动的设计难度甚至要低于一个普通的 GPIO 字符设备。

在串口驱动部分，本章讲解了 tty_driver 到 uart_driver 的角色转换，在 Flash 驱动中，本章讲解了 mtd_info 向 map_info/nand_chip 的转移，可以说，Linux 驱动的这种分层设计思想是贯穿各种 Linux 驱动框架始终的。

第 20 章　USB 主机与设备驱动

在 Linux 系统中，提供了主机侧和设备侧视角的 USB 驱动框架，本章主要讲解从主机侧角度看到的 USB 主机控制器驱动和设备驱动。

20.1 节给出了 Linux 系统中 USB 驱动的整体视图，讲解了 Linux 中主机侧和设备侧角度的 USB 驱动层次。

从主机侧的角度而言，需要编写的 USB 驱动程序包括主机控制器驱动和设备驱动两类，USB 主机控制器驱动程序控制插入其中的 USB 设备，而 USB 设备驱动程序控制该设备如何作为从设备与主机通信。本章 20.2 节分析了 USB 主机控制器驱动的结构并给出实例，20.3 节讲解了 USB 设备驱动的结构及其设备请求块处理过程，并分析了 USB 设备驱动的骨架程序，20.4 节则给出了 Linux 设备驱动的实例。

20.1 节与 20.2～20.4 节是整体与部分的关系，20.2 节与 20.3～20.4 节是并列关系。

20.1 Linux USB 驱动层次

20.1.1 主机侧与设备侧 USB 驱动

USB 采用树形拓扑结构，主机侧和设备侧的 USB 控制器分别称为主机控制器（Host Controller）和 USB 设备控制器（UDC），每条总线上只有一个主机控制器，负责协调主机和设备间的通信，而设备不能主动向主机发送任何消息。如图 20.1 所示，在 Linux 系统中，USB 驱动可以从两个角度去观察，一个角度是主机侧，一个角度是设备侧。

图 20.1 Linux USB 驱动总体结构

如图 20.1 的左侧所示，从主机侧的观念去看，在 Linux 驱动中，USB 驱动处于最底层的是 USB 主机控制器硬件，在其之上运行的是 USB 主机控制器驱动，主机控制器之上为 USB 核心层，再上层为 USB 设备驱动层（插入主机上的 U 盘、鼠标、USB 转串口等设备驱动）。因此，在主机侧的层次结构中，要实现的 USB 驱动包括两类：USB 主机控制器驱动和 USB 设备驱动，前者控制插入其中的 USB 设备，后者控制 USB 设备如何与主机通信。Linux 内核 USB 核心负责 USB 驱动管理和协议处理的主要工作。主机控制器驱动和设备驱动之间的 USB 核心非常重要，其功能包括：通过定义一些数据结构、宏和功能函数，向上为设备驱动提供编程接口，向下为 USB 主机控制器驱动提供编程接口；通过全局变量维护整个系统的 USB 设备信息；完成设备热插拔控制、总线数据传输控制等。

如图 20.1 的右侧所示，Linux 内核中 USB 设备侧驱动程序分为 3 个层次：UDC 驱动程序、Gadget API 和 Gadget 驱动程序。UDC 驱动程序直接访问硬件，控制 USB 设备和主机间的底层通信，向上层提供与硬件相关操作的回调函数。当前 Gadget API 是 UDC 驱动程序回调函数的简单包装。Gadget 驱动程序具体控制 USB 设备功能的实

现，使设备表现出“网络连接”、“打印机”或“USB Mass Storage”等特性，它使用 Gadget API 控制 UDC 实现上述功能。Gadget API 把下层的 UDC 驱动程序和上层的 Gadget 驱动程序隔离开，使得在 Linux 系统中编写 USB 设备侧驱动程序时能够把功能的实现和底层通信分离。

本章将重点讲解从主机侧角度看到的 USB 主机控制器驱动与 USB 设备驱动，关于设备侧的 Linux 驱动，将不会详细讲解。

20.1.2 设备、配置、接口、端点

在 USB 设备的逻辑组织中，包含设备、配置、接口和端点 4 个层次。

每个 USB 设备都提供了不同级别的配置信息，可以包含一个或多个配置，不同的配置使设备表现出不同的功能组合（在探测/连接期间需从其中选定一个），配置由多个接口组成。

在 USB 协议中，接口由多个端点组成，代表一个基本的功能，是 USB 设备驱动程序控制的对象，一个功能复杂的 USB 设备可以具有多个接口。每个配置中可以有多个接口，而设备接口是端点的汇集（collection）。例如 USB 扬声器可以包含一个音频接口以及对旋钮和按钮的接口。一个配置中的所有接口可以同时有效，并可被不同的驱动程序连接。每个接口可以有备用接口，以提供不同质量的服务参数。

端点是 USB 通信的最基本形式，每一个 USB 设备接口在主机看来就是一个端点的集合。主机只能通过端点与设备进行通信，以使用设备的功能。在 USB 系统中每一个端点都有惟一的地址，这是由设备地址和端点号给出的。每个端点都有一定的属性，其中包括传输方式、总线访问频率、带宽、端点号和数据包的最大容量等。一个 USB 端点只能在一个方向承载数据，或者从主机到设备（称为输出端点），或者从设备到主机（称为输入端点），因此端点可看作一个单向的管道。端点 0 通常为控制端点，用于设备初始化参数等。只要设备连接到 USB 上并且上电端点 0 就可以被访问。端点 1、2 等一般用作数据端点，存放主机与设备间往来的数据。

总体而言，USB 设备非常复杂，由许多不同的逻辑单元组成，如图 20.2 所示，这些单元之间的关系如下：

图 20.2 USB 设备、配置、接口和端点

- 设备通常有一个或多个配置；
- 配置通常有一个或多个接口；
- 接口通常有一个或多个设置；

- 接口有零或多个端点。

这种层次化配置信息在设备中通过一组标准的描述符来描述，如下所示。

- 设备描述符：关于设备的通用信息，如供应商 ID、产品 ID 和修订 ID，支持的设备类、子类和适用的协议以及默认端点的最大包大小等。在 Linux 内核中，USB 设备用 usb_device 结构体来描述，USB 设备描述符定义为 usb_device_descriptor 结构体，如代码清单 20.1 所示。

代码清单 20.1　usb_device_descriptor 结构体

```
struct usb_device_descriptor
{
  __u8  bLength; //描述符长度
  __u8  bDescriptorType; //描述符类型编号

  __le16 bcdUSB; //USB 版本号
  __u8  bDeviceClass; //USB 分配的设备类 code
  __u8  bDeviceSubClass;// USB 分配的子类 code
  __u8  bDeviceProtocol; //USB 分配的协议 code
  __u8  bMaxPacketSize0; //endpoint0 最大包大小
  __le16 idVendor; //厂商编号
  __le16 idProduct; //产品编号
  __le16 bcdDevice; //设备出厂编号
  __u8  iManufacturer; //描述厂商字符串的索引
  __u8  iProduct; //描述产品字符串的索引
  __u8  iSerialNumber; //描述设备序列号字符串的索引
  __u8  bNumConfigurations; //可能的配置数量
} __attribute__ ((packed));
```

- 配置描述符：此配置中的接口数、支持的挂起和恢复能力以及功率要求。USB 配置在内核中使用 usb_host_config 结构体描述，USB 配置描述符定义为结构体 usb_config_descriptor，如代码清单 20.2 所示。

代码清单 20.2　usb_config_descriptor 结构体

```
struct usb_config_descriptor
{
  __u8  bLength; //描述符长度
  __u8  bDescriptorType; //描述符类型编号

  __le16 wTotalLength; //配置所返回的所有数据的大小
  __u8  bNumInterfaces; // 配置所支持的接口数
  __u8  bConfigurationValue; //Set_Configuration 命令需要的参数值
```

```
__u8  iConfiguration; //描述该配置的字符串的索引值
__u8  bmAttributes; //供电模式的选择
__u8  bMaxPower; //设备从总线提取的最大电流
} __attribute__ ((packed));
```

- 接口描述符：接口类、子类和适用的协议，接口备用配置的数目和端点数目。USB 接口在内核中使用 usb_interface 结构体描述，USB 接口描述符定义为结构体 usb_interface_descriptor，如代码清单 20.3 所示。

代码清单 20.3　usb_interface_descriptor 结构体

```
struct usb_interface_descriptor
{
 __u8  bLength;          //描述符长度
 __u8  bDescriptorType; //描述符类型

 __u8  bInterfaceNumber;   // 接口的编号
 __u8  bAlternateSetting; //备用的接口描述符编号
 __u8  bNumEndpoints;      //该接口使用的端点数，不包括端点 0
 __u8  bInterfaceClass;    //接口类型
 __u8  bInterfaceSubClass; //接口子类型
 __u8  bInterfaceProtocol; //接口所遵循的协议
 __u8  iInterface; //描述该接口的字符串索引值
} __attribute__ ((packed));
```

- 端点描述符：端点地址、方向和类型，支持的最大包大小，如果是中断类型的端点则还包括轮询频率。在 Linux 内核中，USB 端点使用 usb_host_endpoint 结构体来描述，USB 端点描述符定义为 usb_endpoint_descriptor 结构体，如代码清单 20.4 所示。

代码清单 20.4　usb_endpoint_descriptor 结构体

```
struct usb_endpoint_descriptor
{
 __u8  bLength; //描述符长度
 __u8  bDescriptorType; //描述符类型
 __u8  bEndpointAddress; //端点地址: 0~3 位是端点号，第 7 位是方向(0-OUT,1-IN)
 __u8  bmAttributes; //端点属性: bit[0:1] 的值为 00 表示控制，为 01 表示同步，为 02 表示批量，为 03 表示中断
 __le16 wMaxPacketSize; //// 本端点接收或发送的最大信息包的大小
 __u8  bInterval;    //轮询数据传送端点的时间间隔
                     //对于批量传送的端点以及控制传送的端点，此域忽略
                     //对于同步传送的端点，此域必须为 1
                     //对于中断传送的端点，此域值的范围为 1~255
 __u8  bRefresh;
 __u8  bSynchAddress;
} __attribute__ ((packed));
```

- 字符串描述符：在其他描述符中会为某些字段提供字符串索引，它们可被用来检索描述性字符串，可以以多种语言形式提供。字符串描述符是可选的，有的设备有，有的设备没有，字符串描述符对应于 usb_string_descriptor 结构体，如代码清单 20.5 所示。

代码清单 20.5　usb_string_descriptor 结构体

```
struct usb_string_descriptor
{
  __u8  bLength; //描述符长度
  __u8  bDescriptorType; //描述符类型

  __le16 wData[1];       /* 以UTF-16LE编码 */
} __attribute__ ((packed));
```

例如，笔者在运行 Linux 2.6.15.5 的系统上插入一个 SanDisk U 盘后，通过 lsusb 命令得到这个 U 盘相关的描述符，从中可以显示这个 U 盘包含了一个设备描述符、一个配置描述符、一个接口描述符以及批量输入和批量输出两个端点描述符。呈现出来的信息内容直接对应于 usb_device_descriptor、usb_config_descriptor、usb_interface_descriptor、usb_endpoint_descriptor、usb_string_descriptor 结构体，如下所示：

```
Bus 001 Device 004: ID 0781:5151 SanDisk Corp.
Device Descriptor:
  bLength              18
  bDescriptorType       1
  bcdUSB             2.00
  bDeviceClass          0 Interface
  bDeviceSubClass       0
  bDeviceProtocol       0
  bMaxPacketSize0      64
  idVendor         0x0781 SanDisk Corp.
  idProduct        0x5151
  bcdDevice          0.10
  iManufacturer         1 SanDisk Corporation
  iProduct             2 Cruzer Micro
  iSerial              3 20060877500A1BE1FDE1
  bNumConfigurations     1
  Configuration Descriptor:
    bLength                9
    bDescriptorType        2
    wTotalLength          32
    bNumInterfaces         1
    bConfigurationValue    1
    iConfiguration         0
    bmAttributes        0x80
    MaxPower            200mA
    Interface Descriptor:
      bLength              9
      bDescriptorType      4
      bInterfaceNumber     0
      bAlternateSetting    0
      bNumEndpoints        2
      bInterfaceClass      8 Mass Storage
      bInterfaceSubClass   6 SCSI
      bInterfaceProtocol  80 Bulk (Zip)
      iInterface           0
```

```
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x81  EP 1 IN
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               none
        wMaxPacketSize        512
        bInterval               0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x01  EP 1 OUT
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               none
        wMaxPacketSize        512
        bInterval               1
    Language IDs: (length=4)
      0409 English(US)
```

20.2 USB 主机驱动

20.2.1 USB 主机驱动的整体结构

USB 主机控制器有 3 种规格：OHCI (Open Host Controller Interface)、UHCI (Universal Host Controller Interface) 和 EHCI (Enhanced Host Controller Interface)。OHCI 驱动程序用来为非 PC 系统上以及带有 SiS 和 ALi 芯片组的 PC 主板上的 USB 芯片提供支持。UHCI 驱动程序多用来为大多数其他 PC 主板（包括 Intel 和 Via）上的 USB 芯片提供支持。EHCI 由 USB 2.0 规范所提出，它兼容于 OHCI 和 UHCI。UHCI 的硬件线路比 OHCI 简单，所以成本较低，但需要较复杂的驱动程序，CPU 负荷稍重。本节将重点介绍嵌入式系统中常用的 OHCI 主机控制器驱动。

1. 主机控制器驱动

在 Linux 内核中，用 usb_hcd 结构体描述 USB 主机控制器驱动，它包含 USB 主机控制器的“家务”信息、硬件资源、状态描述和用于操作主机控制器的 hc_driver 等，其定义如代码清单 20.6 所示。

代码清单 20.6　usb_hcd 结构体

```
1  struct usb_hcd
2  {
3    /* 管理“家务” */
4    struct usb_bus self;
5    const char *product_desc; /* 产品/厂商字符串 */
```

```
char irq_descr[24]; /* 驱动 + 总线 # */

struct timer_list rh_timer; /* 根 Hub 轮询 */
struct urb *status_urb; /* 目前的状态 urb */

/* 硬件信息/状态 */
const struct hc_driver *driver; /* 硬件特定的钩子函数 */

/* 需要维护的标志 */
unsigned long flags;
#define HCD_FLAG_HW_ACCESSIBLE  0x00000001
#define HCD_FLAG_SAW_IRQ       0x00000002

unsigned rh_registered: 1; /* 根 Hub 注册？ */

/* 下一个标志的采用只是“权益之计”，当所有 HCDs 支持新的根 Hub 轮询机制后将移除 */
unsigned uses_new_polling: 1;
unsigned poll_rh: 1; /* 轮询根 Hub 状态? */
unsigned poll_pending: 1; /* 状态已经改变? */

int irq; /* 被分配的 irq */
void __iomem *regs; /* 设备内存和 I/O */
u64 rsrc_start; /* 内存和 I/O 资源开始位置 */
u64 rsrc_len; /* 内存和 I/O 资源长度 */
unsigned power_budget; /* mA, 0 = 无限制 */

#define HCD_BUFFER_POOLS       4
struct dma_pool *pool[HCD_BUFFER_POOLS];

int state;
#define __ACTIVE              0x01
#define __SUSPEND             0x04
#define __TRANSIENT           0x80

#define HC_STATE_HALT         0
#define HC_STATE_RUNNING       (__ACTIVE)
#define HC_STATE_QUIESCING   (__SUSPEND|__TRANSIENT|__ACTIVE)
#define HC_STATE_RESUMING      (__SUSPEND|__TRANSIENT)
#define HC_STATE_SUSPENDED     (__SUSPEND)

#define HC_IS_RUNNING(state) ((state) & __ACTIVE)
#define HC_IS_SUSPENDED(state) ((state) & __SUSPEND)
/* 主机控制器驱动的私有数据 */
```

```
    49   unsigned long hcd_priv[0]__attribute__((aligned(sizeof(unsigned
long))));
    50 };
```

usb_hcd 中的 hc_driver 成员非常重要，它包含具体的用于操作主机控制器的钩子函数，其定义如代码清单 20.7 所示。

代码清单 20.7　hc_driver 结构体

```
    1  struct hc_driver
    2  {
    3    const char *description; /* "ehci-hcd" 等 */
    4    const char *product_desc; /* 产品/厂商字符串 */
    5    size_t hcd_priv_size; /* 私有数据的大小 */
    6
    7    /* 中断处理函数 */
    8    irqreturn_t(*irq)(struct usb_hcd *hcd, struct pt_regs *regs);
    9
    10   int flags;
    11   #define HCD_MEMORY     0x0001     /* HC 寄存器使用的内存和 I/O */
    12   #define HCD_USB11      0x0010        /* USB 1.1 */
    13   #define HCD_USB2       0x0020        /* USB 2.0 */
    14
    15   /* 被调用以初始化 HCD 和根 Hub */
    16   int(*reset)(struct usb_hcd *hcd);
    17   int(*start)(struct usb_hcd *hcd);
    18
    19   /* 挂起 Hub 后，进入 D3(etc)前被调用 */
    20   int(*suspend)(struct usb_hcd *hcd, pm_message_t message);
    21
    22   /* 在进入 D0(etc)后，恢复 Hub 前调用 */
    23   int(*resume)(struct usb_hcd *hcd);
    24
    25   /* 使 HCD 停止写内存和进行 I/O 操作 */
    26   void(*stop)(struct usb_hcd *hcd);
    27
    28   /* 返回目前的帧数 */
    29   int(*get_frame_number)(struct usb_hcd *hcd);
    30
    31   /* 管理 I/O 请求和设备状态 */
    32   int(*urb_enqueue)(struct usb_hcd *hcd, struct usb_host_endpoint
*ep, struct
    33     urb *urb, gfp_t mem_flags);
    34   int(*urb_dequeue)(struct usb_hcd *hcd, struct urb *urb);
    35
    36   /* 释放 endpoint 资源 */
    37         void(*endpoint_disable)(struct   usb_hcd   *hcd,   struct
usb_host_endpoint *ep);
    38
    39   /* 根 Hub 支持 */
    40   int(*hub_status_data)(struct usb_hcd *hcd, char *buf);
    41   int(*hub_control)(struct usb_hcd *hcd, u16 typeReq, u16 wValue, u16
wIndex,
    42     char *buf, u16 wLength);
    43   int(*bus_suspend)(struct usb_hcd*);
    44   int(*bus_resume)(struct usb_hcd*);
    45   int(*start_port_reset)(struct usb_hcd *, unsigned port_num);
```

```
  void(*hub_irq_enable)(struct usb_hcd*);
};
```

在 Linux 内核中，使用如下函数来创建 HCD：

```
struct usb_hcd *usb_create_hcd (const struct hc_driver *driver,
     struct device *dev, char *bus_name);
```

如下函数被用来增加和移除 HCD：

```
int usb_add_hcd(struct usb_hcd *hcd,
         unsigned int irqnum, unsigned long irqflags);
void usb_remove_hcd(struct usb_hcd *hcd);
```

2．OHCI 主机控制器驱动

OHCI HCD 驱动属于 HCD 驱动的实例，它定义了一个 ohci_hcd 结构体，作为代码清单 20.6 给出的 usb_hcd 结构体的私有数据，这个结构体的定义如代码清单 20.8 所示。

代码清单 20.8　ohci_hcd 结构体

```
struct ohci_hcd
{
  spinlock_t lock;

  /* 与主机控制器通信的 I/O 内存(DMA 一致) */
  struct ohci_regs __iomem *regs;

  /* 与主机控制器通信的主存(DMA 一致) */
  struct ohci_hcca *hcca;
  dma_addr_t hcca_dma;

  struct ed *ed_rm_list; /* 将被移除 */
  struct ed *ed_bulktail; /* 批量队列尾 */
  struct ed *ed_controltail; /* 控制队列尾 */
  struct ed *periodic[NUM_INTS]; /* int_table “影子” */

  /* OTG 控制器和收发器需要软件交互，其他的外部收发器应该是软件透明的 */
  struct otg_transceiver *transceiver;

  /* 队列数据的内存管理 */
  struct dma_pool *td_cache;
  struct dma_pool *ed_cache;
  struct td *td_hash[TD_HASH_SIZE];
  struct list_head pending;

  /* driver 状态 */
  int num_ports;
  int load[NUM_INTS];
  u32 hc_control; /* 主机控制器控制寄存器的复制 */
  unsigned long next_statechange; /* 挂起/恢复 */
  u32 fminterval; /* 被保存的寄存器 */

  struct notifier_block reboot_notifier;
  unsigned long flags;
```

```
};
```

使用如下内联函数可实现 usb_hcd 和 ohci_hcd 的相互转换：

```
struct ohci_hcd *hcd_to_ohci (struct usb_hcd *hcd);
struct usb_hcd *ohci_to_hcd (const struct ohci_hcd *ohci);
```

从 usb_hcd 得到 ohci_hcd 只是取得“私有”数据，而从 ohci_hcd 得到 usb_hcd 则是通过 container_of()从结构体成员获得结构体指针。

使用如下函数可初始化 OHCI 主机控制器：

```
int ohci_init (struct ohci_hcd *ohci);
```

如下函数分别用于开启、停止及复位 OHCI 控制器：

```
int ohci_run (struct ohci_hcd *ohci);
void ohci_stop (struct usb_hcd *hcd);
void ohci_usb_reset (struct ohci_hcd *ohci);
```

OHCI 主机控制器驱动的主机工作仍然是实现代码清单 20.7 给出的 hc_driver 结构体中的成员函数。

20.2.2 实例：S3C2410 USB 主机驱动

S3C2410 内部集成了一个 USB 主机控制器，完全兼容 OCHI 1.0、USB 1.1 标准，支持低速和全速 USB 设备，从基地址 0x49000000 开始分别提供了 OHCI 的 HcRevision、HcControl、HcCommonStatus、HcInterruptStatus、HcInterruptEnable、HcInterruptDisable、HcHCCA、HcPeriodCuttentED、HcControlHeadED、HcControlCurrentED、HcBulkHeadED、HcBulk CurrentED、HcDoneHead、HcRmInterval、HcFmRemaining、HcFmNumber、HcPeriodicStart、HcLSThreshold、HcRhDescriptorA、HcRhDescriptorB、HcRhStatus、HcRhPortStatus1、HcRhPortStatus2 寄存器。

S3C2410 主机控制器驱动 hc_driver 结构体中的大多数成员函数都是通用的 ohci_xxx()函数，而 start()、hub_status_data()、hub_control()函数则针对 S3C2410 而编写的，如代码清单 20.9 所示。

代码清单 20.9 S3C2410 主机控制器驱动的 hc_driver 结构体

```
static const struct hc_driver ohci_s3c2410_hc_driver =
{
  .description =      hcd_name,
  .product_desc =     "S3C24XX OHCI",
  .hcd_priv_size =    sizeof(struct ohci_hcd),

  /* 通用硬件联接    */
  .irq =  ohci_irq,
  .flags =    HCD_USB11 | HCD_MEMORY, //USB 1.1 标准，hc 寄存器位于内存

 /* 基本的生命周期操作 */
 .start =        ohci_s3c2410_start,
 .stop =     ohci_stop,

 /* 管理 I/O 请求和相关的设备资源 */
 .urb_enqueue =      ohci_urb_enqueue,
 .urb_dequeue =      ohci_urb_dequeue,
 .endpoint_disable = ohci_endpoint_disable,

 /* 调度支持 */
 .get_frame_number = ohci_get_frame,
```

```

 /* 根 Hub 支持 */
 .hub_status_data = ohci_s3c2410_hub_status_data,
 .hub_control =     ohci_s3c2410_hub_control,
#ifdef  CONFIG_PM
 .bus_suspend =     ohci_bus_suspend,
 .bus_resume =      ohci_bus_resume,
#endif
 .start_port_reset = ohci_start_port_reset,
};
```

hc_driver 的 start()成员函数用于初始化 OHCI 并启动主机控制器，如代码清单 20.10 所示。

代码清单 20.10　S3C2410 主机控制器驱动的 start()函数

```
static int ohci_s3c2410_start (struct usb_hcd *hcd)
{
 struct ohci_hcd *ohci = hcd_to_ohci (hcd);
 int ret;

 if ((ret = ohci_init(ohci)) < 0) //初始化 ohci_hcd
     return ret;

 if ((ret = ohci_run (ohci)) < 0) { //启动 ohci_hcd
     err ("can't start %s", hcd->self.bus_name);
     ohci_stop (hcd);
     return ret;
 }

 return 0;
}
```

hc_driver 的 hub_control()成员函数 ohci_s3c2410_hub_control()中的主体是调用通用的 ohci_ hub_control()函数，hub_status_data()成员函数 ohci_s3c2410_hub_status_data()的主体是调用通用的 ohci_hub_status_data()函数。

20.3 USB 设备驱动

20.3.1　USB 设备驱动整体结构

Linux 系统实现了几类通用的 USB 设备驱动，划分为如下几个设备类。

- 音频设备类。
- 通信设备类。
- HID（人机接口）设备类。
- 显示设备类。
- 海量存储设备类。
- 电源设备类。

- 打印设备类。
- 集线器设备类。

一般的通用的 Linux 设备（如 U 盘、USB 鼠标、USB 键盘等）都不需要工程师再编写驱动，需要编写的是特定厂商、特定芯片的驱动，而且往往也可以参考内核中已提供的驱动的模板。

这里所说的 USB 设备驱动指的是从主机角度观察，怎样访问被插入的 USB 设备，而不是指 USB 设备内部本身运行的固件程序。USB 设备内的固件称为“设备用户固件”，“设备用户固件”完成设备内部的控制任务，并在 USB 传输中对接收到的设备请求做出解释并予以正确响应。

Linux 内核为各类 USB 设备分配了相应的设备号，如 ACM USB 调制解调器的主设备号为 166（默认设备名/dev/ttyACMn）、USB 打印机的主设备号为 180，次设备号为 0～15（默认设备名/dev/lpn）、USB 串口的主设备号为 188（默认设备名/dev/ttyUSBn）等。

内核中提供了 USB 设备文件系统（usbdevfs，Linux 2.6 改为 usbfs，即 USB 文件系统），它和/proc 类似，都是动态产生的。通过在/etc/fstab 文件中添加如下一行：

```
none /proc/bus/usb usbfs defaults
```

或者输入命令：

```
mount -t usbfs none /proc/bus/usb
```

可以实现 USB 设备文件系统的挂载。

一个典型的/proc/bus/usb/devices 文件的结构如下（笔者在 VmWare 上运行的 Linux 2.6.15.5 内核上的机器上插入了一个 SanDisk U 盘）：

```
T:  Bus=01 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=12  MxCh= 2
B:  Alloc=  0/900 us ( 0%), #Int=  0, #Iso=  0
D:  Ver= 1.10 Cls=09(hub  ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=0000 ProdID=0000 Rev= 2.06
S:  Manufacturer=Linux 2.6.15.5 uhci_hcd
S:  Product=UHCI Host Controller
S:  SerialNumber=0000:00:07.2
C:* #Ifs= 1 Cfg#= 1 Atr=c0 MxPwr=  0mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   2 Ivl=255ms

T:  Bus=01 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  2 Spd=12  MxCh= 0
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=0781 ProdID=5151 Rev= 0.10
S:  Manufacturer=SanDisk Corporation
S:  Product=Cruzer Micro
S:  SerialNumber=20060877500A1BE1FDE1
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=200mA
I:  If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=(none)
E:  Ad=81(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms
E:  Ad=01(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms
```

通过分析 usbfs 中记录的信息，可以得到系统中 USB 完整的信息，例如，usbview 可以以图形化的方式显示系统中的 USB 设备。

当然，在编译 Linux 内核时，应该包括“USB device filesystem”，如图 20.3 所示。usbfs 动态跟踪总线上插入和移除的设备，通过它可以查看系统中 USB 设备的信息，包括拓扑、带宽、设备描述符信息、产品 ID、字符串描述符、配置

描述符、接口描述符、端点描述　　符等。

```
Linux Kernel v2.6.15.5 Configuration
qqqq
```

```
        |   '-- usb1 -> ../../../../devices/pci0000:00/0000:00:07.2/usb1
        '-- usbfs
            |-- bind
            |-- module -> ../../../../module/usbcore
            '-- unbind
```

正如 tty_driver、pci_driver 等，在 Linux 内核中，使用 usb_driver 结构体描述一个 USB 设备驱动，usb_driver 结构体的定义如代码清单 20.11 所示。

代码清单 20.11　usb_driver 结构体

```
1  struct usb_driver {
2   const char *name; /* 驱动名称 */
4   int (*probe) (struct usb_interface *intf,
5                     const struct usb_device_id *id); /*探测函数*/
7   void (*disconnect) (struct usb_interface *intf); /*断开函数*/
9   int (*ioctl) (struct usb_interface *intf, unsigned int code,
10                     void *buf);    /* I/O控制函数 */
12    int  (*suspend)  (struct  usb_interface  *intf,  pm_message_t
message);/*挂起函数*/
13  int (*resume) (struct usb_interface *intf); /* 恢复函数 */
15  void (*pre_reset) (struct usb_interface *intf);
16  void (*post_reset) (struct usb_interface *intf);
18  const struct usb_device_id *id_table;/* usb_device_id表指针 */
20  struct usb_dynids dynids;
21  struct usbdrv_wrap drvwrap;
22  unsigned int no_dynamic_id:1;
23  unsigned int supports_autosuspend:1;
24 };
```

在编写新的 USB 设备驱动时，主要应该完成的工作是 probe()和 disconnect()函数，即探测和断开函数，它们分别在设备被插入和拔出的时候被调用，用于初始化和释放软硬件资源。对 usb_driver 的注册和注销通过这两个函数完成：

```
int usb_register(struct usb_driver *new_driver)
void usb_deregister(struct usb_driver *driver);
```

usb_driver 结构体中的 id_table 成员描述了这个 USB 驱动所支持的 USB 设备列表，它指向一个 usb_device_id 数组，usb_device_id 结构体用于包含 USB 设备的制造商 ID、产品 ID、产品版本、设备类、接口类等信息及其要匹配标志成员 match-Flash（标明要与哪些成员匹配）。可以借助下面一组宏来生成 usb_device_id 结构体的实例：

```
USB_DEVICE(vendor, product)
```

该宏根据制造商 ID 和产品 ID 生成一个 usb_device_id 结构体的实例，在数组中增加该元素将意味着该驱动可支持匹配制造商 ID、产品 ID 的设备。

```
USB_DEVICE_VER(vendor, product, lo, hi)
```

该宏根据制造商 ID、产品 ID、产品版本的最小值和最大值生成一个 usb_device_id 结构体的实例，在数组中增加该元素将意味着该驱动可支持匹配制造商 ID、产品 ID 和 lo～hi 范围内版本的设备。

```
USB_DEVICE_INFO(class, subclass, protocol)
```

该宏用于创建一个匹配设备指定类型的 usb_device_id 结构体实例。

```
USB_INTERFACE_INFO(class, subclass, protocol)
```

该宏用于创建一个匹配接口指定类型的 usb_device_id 结构体实例。

代码清单 20.12 所示为两个用于描述某 USB 驱动所支持的 USB 设备的 usb_device_id 结构体数组实例。

代码清单 20.12　usb_device_id 结构体数组实例

```
/* 本驱动支持的USB设备列表 */

/* 实例1 */
static struct usb_device_id id_table [] = {
  { USB_DEVICE(VENDOR_ID, PRODUCT_ID) },
  { },
};
MODULE_DEVICE_TABLE (usb, id_table);

/* 实例2 */
static struct usb_device_id id_table [] = {
  { .idVendor = 0x10D2, .match_flags = USB_DEVICE_ID_MATCH_VENDOR, },
  { },
};
MODULE_DEVICE_TABLE (usb, id_table);
```

上述 usb_driver 结构体中的函数是 USB 设备驱动中 USB 相关的部分，而 USB 只是一个总线，真正的 USB 设备驱动的主体工作仍然是 USB 设备本身所属类型的驱动，如字符设备、tty 设备、块设备、输入设备等。因此 USB 设备驱动包含其作为总线上挂在设备的驱动和本身所属设备类型的驱动两部分。

与 platform_driver 类似，usb_driver 起到了“牵线”的作用，即在 probe()里注册相应的字符、tty 等设备，在 disconnect()注销相应的字符、tty 等设备，而原先对设备的注册和注销一般直接发生在模块加载和卸载函数中。

尽管 USB 本身所属设备驱动的结构与其不挂在 USB 总线上时完全相同，但是在访问方式上却发生了很大的变化，例如，对于字符设备而言，尽管仍然是 write()、read()、ioctl()这些函数，但是在这些函数中，与 USB 设备通信时不再是 I/O 内存和 I/O 端口的访问，而贯穿始终的是称为 URB 的 USB 请求块。

如图 20.4 所示，在这棵树里，我们把树根比作主机控制器，树叶比作具体的 USB 设备，树干和树枝就是 USB 总线。树叶本身与树枝通过 usb_driver 连接，而树叶本身的驱动（读写、控制）则需要通过其树叶设备本身所属类设备驱动来完成。树根和树叶之间的“通信”依靠在树干和树枝里“流淌”的 URB 来完成。

由此可见，usb_driver 本身只是起到了找到 USB 设备、管理 USB 设备连接和断开的作用，也就是说，它是公司入口处的“打卡机”，可以获得员工（USB 设备）的上/下班情况。树叶和员工一样，可以是研发工程师也可以是销售工程师，而作为 USB 设备的树叶可以是字符树叶、网络树叶或块树叶，因此必须实现相应设备类的驱动。

图 20.4　USB 设备驱动结构

20.3.2　USB 请求块（URB）

1．urb 结构体

USB 请求块（USB request block，urb）是 USB 设备驱动中用来描述与 USB 设备通信所用的基本载体和核心数据结构，非常类似于网络设备驱动中的 sk_buff 结构体，是 USB 主机与设备通信的“电波”。

代码清单 20.13　urb 结构体

```
struct urb
{
  /* 私有的: 只能由 USB 核心和主机控制器访问的字段 */
  struct kref kref; /*urb 引用计数 */
  spinlock_t lock; /* urb 锁 */
  void *hcpriv; /* 主机控制器私有数据 */
  int bandwidth; /* INT/ISO 请求的带宽 */
  atomic_t use_count; /* 并发传输计数 */
  u8 reject; /* 传输将失败*/

  /* 公共的:  可以被驱动使用的字段 */
  struct list_head urb_list; /* 链表头*/
  struct usb_device *dev; /* 关联的 USB 设备 */
  unsigned int pipe; /* 管道信息 */
  int status; /* URB 的当前状态 */
```

```
    unsigned int transfer_flags; /* URB_SHORT_NOT_OK | ...*/
    void *transfer_buffer; /* 发送数据到设备或从设备接收数据的缓冲区 */
    dma_addr_t transfer_dma; /*用来以 DMA 方式向设备传输数据的缓冲区 */
    int transfer_buffer_length;/*transfer_buffer 或 transfer_dma 指向缓冲区的大小 */

    int actual_length; /* URB 结束后，发送或接收数据的实际长度 */
    unsigned char *setup_packet; /* 指向控制 URB 的设置数据包的指针*/
    dma_addr_t setup_dma; /*控制 URB 的设置数据包的 DMA 缓冲区*/
    int start_frame; /*等时传输中用于设置或返回初始帧*/
    int number_of_packets; /*等时传输中等时缓冲区数据 */
    int interval; /* URB 被轮询到的时间间隔（对中断和等时 urb 有效） */
    int error_count;  /* 等时传输错误数量 */
    void *context; /* completion 函数上下文 */
    usb_complete_t complete; /* 当 URB 被完全传输或发生错误时，被调用 */
    struct usb_iso_packet_descriptor iso_frame_desc[0];
    /*单个 URB 一次可定义多个等时传输时，描述各个等时传输 */
};
```

当 transfer_flags 标志中的 URB_NO_TRANSFER_DMA_MAP 被置位时，USB 核心将使用 transfer_dma 指向的缓冲区而非 transfer_buffer 指向的缓冲区，意味着即将传输 DMA 缓冲区。

当 transfer_flags 标志中的 URB_NO_SETUP_DMA_MAP 被置位时，对于有 DMA 缓冲区的控制 urb 而言，USB 核心将使用 setup_dma 指向的缓冲区而非 setup_packet 指向的缓冲区。

2．urb 处理流程

USB 设备中的每个端点都处理一个 urb 队列，在队列被清空之前，一个 urb 的典型生命周期如下：

（1）被一个 USB 设备驱动创建。

创建 urb 结构体的函数为：

```
struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
```

iso_packets 是这个 urb 应当包含的等时数据包的数目，若为 0 表示不创建等时数据包。mem_flags 参数是分配内存的标志，和 kmalloc()函数的分配标志参数含义相同。如果分配成功，该函数返回一个 urb 结构体指针，否则返回 0。

urb 结构体在驱动中不能静态创建，因为这可能破坏 USB 核心给 urb 使用的引用计数方法。

usb_alloc_urb()的“反函数”为：

```
void usb_free_urb(struct urb *urb);
```

该函数用于释放由 usb_alloc_urb()分配的 urb 结构体。

（2）初始化，被安排给一个特定 USB 设备的特定端点。

对于中断 urb，使用 usb_fill_int_urb()函数来初始化 urb，如下所示：

```
void usb_fill_int_urb(struct urb *urb, struct usb_device *dev,
 unsigned int pipe, void *transfer_buffer,
```

```
 int buffer_length, usb_complete_t complete,
 void *context, int interval);
```

urb 参数指向要被初始化的 urb 的指针；dev 指向这个 urb 要被发送到的 USB 设备；pipe 是这个 urb 要被发送到的 USB 设备的特定端点；transfer_buffer 是指向发送数据或接收数据的缓冲区的指针，和 urb 一样，它也不能是静态缓冲区，必须使用 kmalloc()来分配；buffer_length 是 transfer_buffer 指针所指向缓冲区的大小；complete 指针指向当这个 urb 完成时被调用的完成处理函数；context 是完成处理函数的“上下文”；interval 是这个 urb 应当被调度的间隔。

上述函数参数中的 pipe 使用 usb_sndintpipe()或 usb_rcvintpipe()创建。

对于批量 urb，使用 usb_fill_bulk_urb()函数来初始化 urb，如下所示：

```
void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev,
 unsigned int pipe, void *transfer_buffer,
 int buffer_length, usb_complete_t complete,
 void *context);
```

除了没有对应于调度间隔的 interval 参数以外，该函数的参数和 usb_fill_int_urb()函数的参数含义相同。

上述函数参数中的 pipe 使用 usb_sndbulkpipe()或者 usb_rcvbulkpipe()函数来创建。

对于控制 urb，使用 usb_fill_control_urb()函数来初始化 urb，如下所示：

```
void usb_fill_control_urb(struct urb *urb, struct usb_device *dev,
 unsigned int pipe, unsigned char *setup_packet,
 void *transfer_buffer, int buffer_length,
 usb_complete_t complete, void *context);
```

除了增加了新的 setup_packet 参数以外，该函数的参数和 usb_fill_bulk_urb()函数的参数含义相同。setup_packet 参数指向即将被发送到端点的设置数据包。

上述函数参数中的 pipe 使用 usb_sndctrlpipe()或 usb_rcvictrlpipe()函数来创建。

等时 urb 没有像中断、控制和批量 urb 的初始化函数，我们只能手动地初始化 urb，而后才能提交给 USB 核心。代码清单 20.14 给出了初始化等时 urb 的例子，它来自 drivers/usb/media/usbvideo.c 文件。

代码清单 20.14　初始化等时 urb

```
for (i = 0; i < USBVIDEO_NUMSBUF; i++)
{
  int j, k;
  struct urb *urb = uvd->sbuf[i].urb;
  urb->dev = dev;
  urb->context = uvd;
  urb->pipe = usb_rcvisocpipe(dev, uvd->video_endp);/*端口*/
  urb->interval = 1;
  urb->transfer_flags = URB_ISO_ASAP; /*urb 被调度*/
  urb->transfer_buffer = uvd->sbuf[i].data;/*传输 buffer*/
  urb->complete = usbvideo_IsocIrq; /* 完成函数 */
  urb->number_of_packets = FRAMES_PER_DESC; /*urb 中的等时传输数量*/
  urb->transfer_buffer_length = uvd->iso_packet_len *FRAMES_PER_DESC;
  for (j = k = 0; j < FRAMES_PER_DESC; j++, k += uvd->iso_packet_len)
  {
    urb->iso_frame_desc[j].offset = k;
    urb->iso_frame_desc[j].length = uvd->iso_packet_len;
```

```
  }
}
```

（3）被 USB 设备驱动提交给 USB 核心。

在完成第（1）、（2）步的创建和初始化 urb 后，urb 便可以提交给 USB 核心，通过 usb_submit_urb()函数来完成，如下所示：

```
int usb_submit_urb(struct urb *urb, int mem_flags);
```

urb 参数是指向 urb 的指针，mem_flags 参数与传递给 kmalloc()函数参数的意义相同，它用于告知 USB 核心如何在此时分配内存缓冲区。

在提交 urb 到 USB 核心后，直到完成函数被调用之前，不要访问 urb 中的任何成员。

usb_submit_urb()在原子上下文和进程上下文中都可以被调用，mem_flags 变量需根据调用环境进行相应的设置，如下所示。

- GFP_ATOMIC：在中断处理函数、底半部、tasklet、定时器处理函数以及 urb 完成函数中，在调用者持有自旋锁或者读写锁时以及当驱动将 current->state 修改为非 TASK_ RUNNING 时，应使用此标志。
- GFP_NOIO：在存储设备的块 I/O 和错误处理路径中，应使用此标志；
- GFP_KERNEL：如果没有任何理由使用 GFP_ATOMIC 和 GFP_NOIO，就使用 GFP_ KERNEL。

如果 usb_submit_urb()调用成功，即 urb 的控制权被移交给 USB 核心，该函数返回 0；否则，返回错误号。

（4）提交由 USB 核心指定的 USB 主机控制器驱动。

（5）被 USB 主机控制器处理，进行一次到 USB 设备的传送。

第（4）～（5）步由 USB 核心和主机控制器完成，不受 USB 设备驱动的控制。

（6）当 urb 完成，USB 主机控制器驱动通知 USB 设备驱动。

在如下 3 种情况下，urb 将结束，urb 完成函数将被调用。

- urb 被成功发送给设备，并且设备返回正确的确认。如果 urb->status 为 0，意味着对于一个输出 urb，数据被成功发送；对于一个输入 urb，请求的数据被成功收到。
- 如果发送数据到设备或从设备接收数据时发生了错误，urb->status 将记录错误值。
- urb 被从 USB 核心“去除连接”，这发生在驱动通过 usb_unlink_urb()或 usb_kill_urb()函数取消 urb，或 urb 虽已提交，而 USB 设备被拔出的情况下。

usb_unlink_urb()和 usb_kill_urb()这两个函数用于取消已提交的 urb，其参数为要被取消的 urb 指针。对 usb_unlink_urb()而言，如果 urb 结构体中的 URB_ASYNC_UNLINK（即异步 unlink）的标志被置位，则对该 urb 的 usb_unlink_urb()调用将立即返回，具体的 unlink 动作将在后台进行。否则，此函数一直等到 urb 被解开链接或结束时才返回。usb_kill_urb()会彻底终止 urb 的生命周期，它通常在设备的 disconnect()函数中被调用。

当 urb 生命结束时（处理完成或被解除链接），通过 urb 结构体的 status 成员可以获知其原因，如 0 表示传输成功，-ENOENT 表示被 usb_kill_urb()杀死，-ECONNRESET 表示被 usb_unlink_urb()杀死，-EPROTO 表示传输中发生了 bitstuff 错误或者硬件未能及时收到响应数据包，-ENODEV 表示 USB 设备已被移除，-EXDEV 表示等时传输仅完成了一部分等。

对以上 urb 的处理步骤进行一个总结，图 20.5 给出了一个 urb 的整个处理流程，虚线框的 usb_unlink_urb()和 usb_kill_urb()并非一定会发生，它只是在 urb 正在被 USB 核心和主机控制器处理时，被驱动程序取消的情况下才发生。

3．简单的批量与控制 URB

有时 USB 驱动程序只是从 USB 设备上接收或向 USB 设备发送一些简单的数据，这时候，没有必要将 urb 创建、初始化、提交、完成处理的整个流程走一遍，而可以使用两个更简单的函数，如下所示。

（1）usb_bulk_msg()。

usb_bulk_msg()函数创建一个 USB 批量 urb 并将它发送到特定设备，这个函数是同步的，它一直等待 urb 完成后才返回。usb_bulk_msg()函数的原型为：

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
                 void *data, int len, int *actual_length,
                 int timeout);
```

图 20.5　urb 处理流程

usb_dev 参数为批量消息要发送的 USB 设备的指针，pipe 为批量消息要发送到的 USB 设备的端点，data 参数为指向要发送或接收的数据缓冲区的指针，len 参数为 data 参数所指向的缓冲区的长度，actual_length 用于返回实际发送或接收的字节数，timeout 是发送超时，以 jiffies 为单位，0 意味着永远等待。

如果函数调用成功，返回 0；否则，返回 1 个负的错误值。

（2）usb_control_msg()函数。

usb_control_msg()函数与 usb_bulk_msg()函数类似，不过它提供驱动发送和结束 USB 控制信息而非批量信息的能力，该函数的原型为：

```
    int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request,
    __u8 requesttype, __u16 value, __u16 index, void *data, __u16 size, int timeout);
```

dev 指向控制消息发往的 USB 设备，pipe 是控制消息要发往的 USB 设备的端点，request 是这个控制消息的 USB 请求值，requesttype 是这个控制消息的 USB 请求类型，value 是这个控制消息的 USB 消息值，index 是这个控制消息的 USB 消息索引值，data 指向要发送或接收的数据缓冲区，size 是 data 参数所指向的缓冲区的大小，timeout 是发送超时，以 jiffies 为单位，0 意味着永远等待。

参数 request、requesttype、value 和 index 与 USB 规范中定义的 USB 控制消息直接对应。

如果函数调用成功，该函数返回发送到设备或从设备接收到的字节数；否则，返回一个负的错误值。

对 usb_bulk_msg()和 usb_control_msg()函数的使用要特别慎重，由于它们是同步的，因此不能在中断上下文和持有自旋锁的情况下使用。而且，该函数也不能被任何其他函数取消，因此，务必要使得驱动程序的 disconnect()函数掌握足够的信息，以判断和等待该调用的结束。

华清远见

20.3.3 探测和断开函数

在 USB 设备驱动 usb_driver 结构体的探测函数中，应该完成如下工作。

- 探测设备的端点地址、缓冲区大小，初始化任何可能用于控制 USB 设备的数据结构。
- 把已初始化数据结构的指针保存到接口设备中。

usb_set_intfdata()函数可以设置 usb_interface 的私有数据，这个函数的原型为：

```
void usb_set_intfdata (struct usb_interface *intf, void *data);
```

这个函数的“反函数”用于得到 usb_interface 的私有数据，其原型为：

```
void *usb_get_intfdata (struct usb_interface *intf);
```

- 注册 USB 设备。

如果是简单的字符设备，调用 usb_register_dev()，这个函数的原型为：

```
int usb_register_dev(struct usb_interface *intf,
          struct usb_class_driver *class_driver);
```

上述函数中第二个参数为 usb_class_driver 结构体，这个结构体的定义如代码清单 20.15 所示。

代码清单 20.15　usb_class_driver 结构体

```
struct usb_class_driver
{
  char *name;  /*sysfs 中用来描述设备名*/
  struct file_operations *fops;/*文件操作结构体指针*/
  int minor_base; /*开始次设备号*/
};
```

对于字符设备而言，usb_class_driver 结构体的 fops 成员中的 write()、read()、ioctl()等函数的地位完全等同于本书第 6 章中的 file_operations 成员函数。

如果是其他类型的设备，如 tty 设备，则调用对应设备的注册函数。

在 USB 设备驱动 usb_driver 结构体的探测函数中，应该完成如下工作。

- 释放所有为设备分配的资源。
- 设置接口设备的数据指针为 NULL。
- 注销 USB 设备。

对于字符设备，可以直接调用 usb_register_dev()函数的“反函数”，如下所示：

```
void usb_deregister_dev(struct usb_interface *intf,
      struct usb_class_driver *class_driver);
```

对于其他类型的设备，如 tty 设备，则调用对应设备的注销函数。

对探测函数的调用发生在 USB 设备被安装且 USB 核心认为该驱动程序与安装的 USB 设备对应时（usb_driver 的 id_table 成员在此时发挥作用），而对断开函数的调用则发生在驱动因为种种原因不再控制该设备的时候。对这两个函数的调用都是在内核线程中进行的，因此，可以在其中进行任何可能引起睡眠的操作。但是，由于 USB 核心对所有探测和断开函数的调用都发生在单一线程内，因此，太慢的 probe()或 disconnect()将拖延其他 USB 设备的探测时间。

20.3.4 USB 骨架程序

Linux 内核源代码中的 driver/usb/usb-skeleton.c 文件为我们提供了一个最基础的 USB 驱动程序，即 USB 骨架程序，可被看做一个最简单的 USB 设备驱动实例。尽管具体 USB 设备驱动千差万别，但其骨架则万变不离其宗。

首先看看 USB 骨架程序的 usb_driver 结构体定义，如代码清单 20.16 所示。

代码清单 20.16 USB 骨架程序的 usb_driver 结构体

```
1 static struct usb_driver skel_driver =
2 {
3   .name =     "skeleton",
4   .probe =    skel_probe,
5   .disconnect =   skel_disconnect,
6   .id_table = skel_table,
7 };
```

从上述代码第 6 行可以看出，它所支持的 USB 设备的列表数组为 skel_table[]，其定义如代码清单 20.17 所示。

代码清单 20.17 USB 骨架程序的 id_table

```
1  /* 本驱动支持的 USB 设备列表 */
2  static struct usb_device_id skel_table [] = {
3      { USB_DEVICE(USB_SKEL_VENDOR_ID,
4        USB_SKEL_PRODUCT_ID) },
5      { }
6  };
7  MODULE_DEVICE_TABLE (usb, skel_table);
```

对上述 usb_driver 的注册和注销发生在 USB 骨架程序的模块加载与卸载函数内，如代码清单 20.18 所示，其分别调用了 usb_register()和 usb_deregister()。

代码清单 20.18 USB 骨架程序的模块加载与卸载函数

```
1  static int __init usb_skel_init(void)
2  {
3   int result;
4
5   /* 注册 USB 驱动 */
6   result = usb_register(&skel_driver);
7   if (result)
8       err("usb_register failed. Error number %d", result);
9
10  return result;
11 }
```

```
12 static void __exit usb_skel_exit(void)
13 {
14   /* 注销 USB 驱动 */
15   usb_deregister(&skel_driver);
16 }
```

usb_driver 的 probe()成员函数中，会根据 usb_interface 的成员寻找第一个批量输入和输出端点，将端点地址、缓冲区等信息存入为 USB 骨架程序定义的 usb_skel 结构体，并将 usb_skel 实例的指针传入 usb_set_intfdata()作为 USB 接口的私有数据，最后，它会注册 USB 设备，如代码清单 20.19 所示。

代码清单 20.19　USB 骨架程序的探测函数

```
1  static int skel_probe(struct usb_interface *interface, const struct usb_device_id *id)
2  {
3   struct usb_skel *dev = NULL;
4   struct usb_host_interface *iface_desc;
5   struct usb_endpoint_descriptor *endpoint;
6   size_t buffer_size;
7   int i;
8   int retval = -ENOMEM;
9
10  /* 分配设备状态的内存并初始化 */
11  dev = kzalloc(sizeof(*dev), GFP_KERNEL);
12  if (dev == NULL) {
13      err("Out of memory");
14      goto error;
15  }
16  kref_init(&dev->kref);
17  sema_init(&dev->limit_sem, WRITES_IN_FLIGHT);
18
19  dev->udev = usb_get_dev(interface_to_usbdev(interface));
20  dev->interface = interface;
21
22  /* 设置端点信息 */
23  /* 仅使用第一个 bulk-in 和 bulk-out */
24  iface_desc = interface->cur_altsetting;
25  for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
26      endpoint = &iface_desc->endpoint[i].desc;
27
28      if (!dev->bulk_in_endpointAddr &&
29          ((endpoint->bEndpointAddress & USB_ENDPOINT_DIR_MASK)
```

```
                == USB_DIR_IN) &&
        ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
                == USB_ENDPOINT_XFER_BULK)) {
        /* 找到了一个批量 IN 端点 */
        buffer_size = le16_to_cpu(endpoint->wMaxPacketSize);
        dev->bulk_in_size = buffer_size;
        dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
        dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
        if (!dev->bulk_in_buffer) {
            err("Could not allocate bulk_in_buffer");
            goto error;
        }
    }

    if (!dev->bulk_out_endpointAddr &&
        ((endpoint->bEndpointAddress & USB_ENDPOINT_DIR_MASK)
                == USB_DIR_OUT) &&
        ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
                == USB_ENDPOINT_XFER_BULK)) {
        /* 找到了一个批量 OUT 端点 */
        dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
    }
}
if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
    err("Could not find both bulk-in and bulk-out endpoints");
    goto error;
}

/* 在设备结构中保存数据指针 */
usb_set_intfdata(interface, dev);

/* 注册 USB 设备 */
retval = usb_register_dev(interface, &skel_class);
if (retval) {
    /* something prevented us from registering this driver */
    err("Not able to get a minor for this device.");
    usb_set_intfdata(interface, NULL);
    goto error;
```

```
68  }
69
70  /* 告知用户设备依附于什么 node */
71  info("USB  Skeleton  device  now  attached  to  USBSkel-%d",
interface->minor);
72  return 0;
73
74 error:
75  if (dev)
76      kref_put(&dev->kref, skel_delete);
77  return retval;
78 }
```

usb_skel 结构体可以被看作一个私有数据结构体，其定义如代码清单 20.20 所示，应该根据具体的设备量身定制。

代码清单 20.20　USB 骨架程序的自定义数据结构 usb_skel

```
1  struct usb_skel
2  {
3   struct usb_device * udev;              /* 该设备的 usb_device 指针
*/
4   struct usb_interface *  interface;     /* 该设备的 usb_interface 指
针 */
5   struct semaphore    limit_sem;             /* 限制进程写的数量 */
6   unsigned char *     bulk_in_buffer;    /* 接收数据的缓冲区 */
7   size_t          bulk_in_size;              /* 接收缓冲区大小 */
8   __u8        bulk_in_endpointAddr;          /*批量 IN 端点的地址 */
9   __u8        bulk_out_endpointAddr;     /* 批量 OUT 端点的地址 */
10  struct kref     kref;
11 };
```

USB 骨架程序的断开函数会完成探测函数相反的工作，即设置接口数据为 NULL，注销 USB 设备，如代码清单 20.21 所示。

代码清单 20.21　USB 骨架程序的断开函数

```
1  static void skel_disconnect(struct usb_interface *interface)
2  {
3   struct usb_skel *dev;
4   int minor = interface->minor;
5
6   /* 阻止 skel_open()与 skel_disconnect()的竞争 */
7   lock_kernel();
8
```

```
9   dev = usb_get_intfdata(interface);
10  usb_set_intfdata(interface, NULL);
11
12  /* 注销 usb 设备，释放次设备号 */
13  usb_deregister_dev(interface, &skel_class);
14
15  unlock_kernel();
16
17  /* 减少引用计数 */
18  kref_put(&dev->kref, skel_delete);
19
20  info("USB Skeleton #%d now disconnected", minor);
21 }
```

代码清单 20.19 第 62 行的 usb_register_dev(interface, &skel_class)中第二个参数包含了字符设备的 file_operations 结构体指针，而这个结构体中的成员实现也是 USB 字符设备的另一个组成成分。代码清单 20.22 给出了 USB 骨架程序的字符设备文件操作 file_operations 结构体的定义。

代码清单 20.22　USB 骨架程序的字符设备文件操作结构体

```
1 static struct file_operations skel_fops =
2 {
3   .owner =    THIS_MODULE,
4   .read =     skel_read,  //读函数
5   .write =    skel_write,     //写函数
6   .open =     skel_open,      //打开函数
7   .release =  skel_release, //释放函数
8 };
```

由于只是一个象征性的骨架程序，open()成员函数的实现非常简单，它根据 usb_driver 和次设备号通过 usb_find_interface()获得 USB 接口，之后通过 usb_get_intfdata()获得接口的私有数据并赋予 file-> private_data，如代码清单 20.23 所示。

代码清单 20.23　USB 骨架程序的字符设备打开函数

```
1  static int skel_open(struct inode *inode, struct file *file)
2  {
3   struct usb_skel *dev;
4   struct usb_interface *interface;
5   int subminor;
6   int retval = 0;
7
```

```
    8   subminor = iminor(inode);
    9
    10  interface = usb_find_interface(&skel_driver, subminor);
    11  if (!interface) {
    12      err ("%s - error, can't find device for minor %d",
    13          __FUNCTION__, subminor);
    14      retval = -ENODEV;
    15      goto exit;
    16  }
    17
    18  dev = usb_get_intfdata(interface);/*获得接口数据*/
    19  if (!dev) {
    20      retval = -ENODEV;
    21      goto exit;
    22  }
    23
    24  /* 增加设备的引用计数 */
    25  kref_get(&dev->kref);
    26
    27  /* 将接口数据保存在 file->private_data 中 */
    28  file->private_data = dev;
    29
    30 exit:
    31  return retval;
    32 }
```

由于在 open()函数中并没有申请任何软件和硬件资源，因此与 open()函数对应的 release()函数不用进行资源的释放，它会减少在 open()中增加的引用计数，如代码清单 20.24 所示。

代码清单 20.24　USB 骨架程序的字符设备释放函数

```
    1  static int skel_release(struct inode *inode, struct file *file)
    2  {
    3   struct usb_skel *dev;
    4
    5   dev = (struct usb_skel *)file->private_data;
    6   if (dev == NULL)
    7       return -ENODEV;
    8
    9   /* 减少设备的引用计数 */
    10  kref_put(&dev->kref, skel_delete);
    11  return 0;
    12 }
```

接下来要分析的是读写函数，前面已经提到，在访问 USB 设备的时候，贯穿于其中的“中枢神经”是 urb 结构体。

在 skel_write()函数中进行的关于 urb 的操作与 20.3.2 小节的描述完全对应，即进行了 urb 的分配（调用 usb_alloc_urb()）、初始化（调用 usb_fill_bulk_urb()）和提交（调用 usb_submit_urb()）的操作，如代码清单 20.25 所示。

代码清单 20.25　USB 骨架程序的字符设备写函数

```
    1 static ssize_t skel_write(struct file *file, const char *user_buffer, size_t
count, loff_t *ppos)
    2  {
    3   struct usb_skel *dev;
    4   int retval = 0;
```

```
    5   struct urb *urb = NULL;
    6   char *buf = NULL;
    7   size_t writesize = min(count, (size_t)MAX_TRANSFER);
    8
    9   dev = (struct usb_skel *)file->private_data;
    10
    11  /* 验证实际上有数据要写入USB设备 */
    12  if (count == 0)
    13      goto exit;
    14
    15  /* 限制urb的数量，防止一个用户用完所有的RAM */
    16  if (down_interruptible(&dev->limit_sem)) {
    17      retval = -ERESTARTSYS;
    18      goto exit;
    19  }
    20
    21  /* 创建urb、urb的缓冲区，将数据复制给urb */
    22  urb = usb_alloc_urb(0, GFP_KERNEL);
    23  if (!urb) {
    24      retval = -ENOMEM;
    25      goto error;
    26  }
    27
    28  buf   =   usb_buffer_alloc(dev->udev,   writesize,   GFP_KERNEL,
&urb->transfer_dma);
    29  if (!buf) {
    30      retval = -ENOMEM;
    31      goto error;
    32  }
    33
    34  if (copy_from_user(buf, user_buffer, writesize)) {
    35      retval = -EFAULT;
    36      goto error;
    37  }
    38
    39  /* 恰当地初始化urb */
    40  usb_fill_bulk_urb(urb, dev->udev,
    41          usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
```

```
            buf, writesize, skel_write_bulk_callback, dev);
 urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

 /* 将数据发送到批量端口 */
 retval = usb_submit_urb(urb, GFP_KERNEL);
 if (retval) {
     err("%s - failed submitting write urb, error %d", __FUNCTION__, retval);
     goto error;
 }

 /* 释放对urb的引用 */
 usb_free_urb(urb);

exit:
 return writesize;

error:
 usb_buffer_free(dev->udev, writesize, buf, urb->transfer_dma);
 usb_free_urb(urb);
 up(&dev->limit_sem);
 return retval;
}
```

写函数中发起的 urb 结束后，其完成函数 skel_write_bulk_callback()将被调用，它会进行 urb->status 的判断，如代码清单 20.26 所示。

代码清单 20.26　USB 骨架程序的字符设备写操作完成函数

```
static void skel_write_bulk_callback(struct urb *urb, struct pt_regs *regs)
{
 struct usb_skel *dev;

 dev = (struct usb_skel *)urb->context;

 /* sync/async去除链路故障，不是错误 */
 if (urb->status &&
     !(urb->status == -ENOENT ||
       urb->status == -ECONNRESET ||
       urb->status == -ESHUTDOWN)) {
     dbg("%s - nonzero write bulk status received: %d",
         __FUNCTION__, urb->status);
 }

 /* 释放被分配的内存 */
```

```
usb_buffer_free(urb->dev, urb->transfer_buffer_length,
        urb->transfer_buffer, urb->transfer_dma);
up(&dev->limit_sem);
}
```

USB 骨架程序的字符设备读函数并没有进行类似写函数的一系列针对 urb 的操作，而是简单地调用 usb_bulk_msg()发起一次同步 urb 传输操作，如代码清单 20.27 所示。

代码清单 20.27　USB 骨架程序的字符设备读函数

```
static ssize_t skel_read(struct file *file, char *buffer, size_t count, loff_t *ppos)
{
 struct usb_skel *dev;
 int retval = 0;
 int bytes_read;

 dev = (struct usb_skel *)file->private_data;

 /* 从设备进行一次阻塞的批量读 */
 retval = usb_bulk_msg(dev->udev,
             usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr),
             dev->bulk_in_buffer,
             min(dev->bulk_in_size, count),
             &bytes_read, 10000);

 /* 如果读成功，将数据复制到用户空间 */
 if (!retval) {
     if (copy_to_user(buffer, dev->bulk_in_buffer, bytes_read))
         retval = -EFAULT;
     else
         retval = bytes_read;
 }

 return retval;
}
```

20.4 USB 设备驱动实例

20.4.1　USB 串口驱动

在 Linux 内核中，串口属于 tty 设备，对于一个 USB 串口设备而言，其驱动主要由两部分组成：usb_driver 的成员函数和 tty 设备的 tty_operations 结构体成员函数。

在 USB 串口设备驱动的模块加载函数中，将注册对应于 USB 串口的 usb_driver，

并初始化和注册 tty 驱动，如代码清单 20.28 所示。

代码清单 20.28　USB 串口设备驱动的模块加载函数

```
static int __init usb_serial_init(void)
{
  int i;
  int result;

  /* 分配 tty_driver */
  usb_serial_tty_driver = alloc_tty_driver(SERIAL_TTY_MINORS);
  if (!usb_serial_tty_driver)
    return  - ENOMEM;

  /* 初始化全局数据 */
  for (i = 0; i < SERIAL_TTY_MINORS; ++i)
  {
    serial_table[i] = NULL;
  }

  /* 注册总线 */
  result = bus_register(&usb_serial_bus_type);
  if (result)
  {
    err("%s - registering bus driver failed", __FUNCTION__);
    goto exit_bus;
  }

  /* 初始化 tty_driver */
  usb_serial_tty_driver->owner = THIS_MODULE;
  usb_serial_tty_driver->driver_name = "usbserial";
  usb_serial_tty_driver->devfs_name = "usb/tts/";
  usb_serial_tty_driver->name = "ttyUSB";
  usb_serial_tty_driver->major = SERIAL_TTY_MAJOR;
  usb_serial_tty_driver->minor_start = 0;
  usb_serial_tty_driver->type = TTY_DRIVER_TYPE_SERIAL;
  usb_serial_tty_driver->subtype = SERIAL_TYPE_NORMAL;
  usb_serial_tty_driver->flags = TTY_DRIVER_REAL_RAW | TTY_DRIVER_NO_DEVFS;
  usb_serial_tty_driver->init_termios = tty_std_termios;
  usb_serial_tty_driver->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL |
```

```
    CLOCAL;
  tty_set_operations(usb_serial_tty_driver, &serial_ops);
  /* 注册 tty_driver */
  result = tty_register_driver(usb_serial_tty_driver);
  if (result)
  {
    err("%s - tty_register_driver failed", __FUNCTION__);
    goto exit_reg_driver;
  }

  /* 注册 USB 驱动 */
  result = usb_register(&usb_serial_driver);
  if (result < 0)
  {
    err("%s - usb_register failed", __FUNCTION__);
    goto exit_tty;
  }

  result = usb_serial_generic_register(debug);
  if (result < 0)
  {
    err("%s - registering generic driver failed", __FUNCTION__);
    goto exit_generic;
  }

  info(DRIVER_DESC);

  return result;

  exit_generic: usb_deregister(&usb_serial_driver);
  exit_tty: tty_unregister_driver(usb_serial_tty_driver);
  exit_reg_driver: bus_unregister(&usb_serial_bus_type);
   exit_bus: err("%s - returning with error %d", __FUNCTION__, result);
  put_tty_driver(usb_serial_tty_driver);
  return result;
}
```

在 USB 串口设备驱动的模块卸载函数中，将注销对应于 USB 串口的 usb_driver，并注销 tty 驱动，如代码清单 20.29 所示。

代码清单 20.29　USB 串口设备驱动的模块卸载函数

```
static void __exit usb_serial_exit(void)
{
  usb_serial_console_exit();
  usb_serial_generic_deregister();
  usb_deregister(&usb_serial_driver);//注销 usb_driver
  tty_unregister_driver(usb_serial_tty_driver);//注销 tty_driver
  put_tty_driver(usb_serial_tty_driver);//减少引用计数
```

```
8   bus_unregister(&usb_serial_bus_type);//注销 bus
9 }
```

在 usb_driver 的探测成员函数 usb_serial_probe()中，将初始化 USB 端点等信息，并通过 usb_set_intfdata()设置接口私有数据，它也将初始化 urb。相反地，在断开成员函数 usb_serial_disconnect()中将设置接口私有数据为 NULL，并释放引用计数。

USB 串口驱动的 tty_operations 结构体实例 serial_ops 定义如代码清单 20.30 所示，它封装了 USB 串口设备驱动中的串口驱动成分。

代码清单 20.30　USB 串口驱动的 tty_operations 结构体

```
1  static struct tty_operations serial_ops =
2  {
3   .open =         serial_open,
4   .close =        serial_close,
5   .write =        serial_write,
6   .write_room =       serial_write_room,
7   .ioctl =        serial_ioctl,
8   .set_termios =      serial_set_termios,
9   .throttle =     serial_throttle,
10  .unthrottle =       serial_unthrottle,
11  .break_ctl =        serial_break,
12  .chars_in_buffer =  serial_chars_in_buffer,
13  .read_proc =        serial_read_proc,
14  .tiocmget =     serial_tiocmget,
15  .tiocmset =     serial_tiocmset,
16 };
```

在 tty_operations 的各 write()、read()等成员函数中，将调用 usb_serial_driver 结构体中的相应函数，usb_serial_driver 结构体中封装了串口的各函数（读写、读写中断端点完成函数、读写批量端点完成函数等），其定义如代码清单 20.31 所示。

代码清单 20.31　usb_serial_driver 结构体

```
1  struct usb_serial_driver
2  {
3    const char *description; //用于描述该驱动的字符串
4    const struct usb_device_id *id_table; //usb_device_id 数组
5    char num_interrupt_in; //中断输入端点数量
6    char num_interrupt_out; //中断输出端点数量
7    char num_bulk_in; //批量输入端点数量
8    char num_bulk_out; //批量输出端点数量
9    char num_ports; //设备包含的端口数量
10
11   struct list_head driver_list;
```

```
struct device_driver driver;

int(*probe)(struct usb_serial *serial, const struct usb_device_id *id);
int(*attach)(struct usb_serial *serial);
int(*calc_num_ports)(struct usb_serial *serial);

void(*shutdown)(struct usb_serial *serial);

int(*port_probe)(struct usb_serial_port *port);
int(*port_remove)(struct usb_serial_port *port);

/* 串口函数 */
int(*open)(struct usb_serial_port *port, struct file *filp);
void(*close)(struct usb_serial_port *port, struct file *filp);
int(*write)(struct usb_serial_port *port, const unsigned char *buf, int count)
    ;
int(*write_room)(struct usb_serial_port *port);
 int(*ioctl)(struct usb_serial_port *port, struct file *file, unsigned int cmd,
  unsigned long arg);
void(*set_termios)(struct usb_serial_port *port, struct termios *old);
void(*break_ctl)(struct usb_serial_port *port, int break_state);
int(*chars_in_buffer)(struct usb_serial_port *port);
void(*throttle)(struct usb_serial_port *port);
void(*unthrottle)(struct usb_serial_port *port);
int(*tiocmget)(struct usb_serial_port *port, struct file *file);
int(*tiocmset)(struct usb_serial_port *port, struct file *file, unsigned int
  set, unsigned int clear);
/* urb 完成回调函数 */
void(*read_int_callback)(struct urb *urb, struct pt_regs *regs);
void(*write_int_callback)(struct urb *urb, struct pt_regs *regs);
void(*read_bulk_callback)(struct urb *urb, struct pt_regs *regs);
void(*write_bulk_callback)(struct urb *urb, struct pt_regs *regs);
};
```

文件 drivers/usb/serial/Generic.c 文件中提供了 USB 串口驱动的通用打开/关闭、写函数、批量 urb 完成函数，如 usb_serial_generic_write()、usb_serial_generic_write_bulk_callback()、usb_serial_generic_read_bulk_callback()等。代码清单 20.32 列举了通用写函数及输出批量 urb 完成回调函数。

代码清单 20.32　USB 串口通用写函数及批量写 urb 完成函数

```
int usb_serial_generic_write(struct usb_serial_port *port, const unsigned char
  *buf, int count)
{
  struct usb_serial *serial = port->serial;
  int result;
  unsigned char *data;

  if (count == 0)
  {
    dbg("%s - write request of 0 bytes", __FUNCTION__);
    return (0);
  }

  /* 如果有批量输出端点 */
  if (serial->num_bulk_out)
  {
    spin_lock(&port->lock);
    if (port->write_urb_busy)
    {
      spin_unlock(&port->lock);
      dbg("%s - already writing", __FUNCTION__);
      return 0;
    }
    port->write_urb_busy = 1;
    spin_unlock(&port->lock);

     count = (count > port->bulk_out_size) ? port->bulk_out_size: count;

    memcpy(port->write_urb->transfer_buffer, buf, count);
    data = port->write_urb->transfer_buffer;
    usb_serial_debug_data(debug, &port->dev, __FUNCTION__, count, data);

    /* 设置urb */
                usb_fill_bulk_urb(port->write_urb,   serial->dev, usb_sndbulkpipe(serial->dev,
                                  port->bulk_out_endpointAddress), port->write_urb->transfer_buffer, count,
                        ((serial->type->write_bulk_callback)   ?
```

```
serial->type->write_bulk_callback:
    37      usb_serial_generic_write_bulk_callback), port);
    38
    39    /* 将数据送出给批量端口 */
    40    port->write_urb_busy = 1;
    41    result = usb_submit_urb(port->write_urb, GFP_ATOMIC);
    42    if (result)
    43    {
    44      port->write_urb_busy = 0;
    45    }
    46    else
    47      result = count;
    48
    49    return result;
    50  }
    51
    52  return 0;
    53 }
    54
    55 void usb_serial_generic_write_bulk_callback(struct urb *urb, struct
pt_regs
    56  *regs)
    57 {
    58              struct     usb_serial_port     *port     =     (struct
usb_serial_port*)urb->context;
    59
    60  port->write_urb_busy = 0;
    61  if (urb->status)   //不成功
    62  {
    63    dbg("%s - nonzero write bulk status received: %d", __FUNCTION_
_, urb
    64      ->status);
    65    return ;
    66  }
    67
    68  usb_serial_port_softint((void*)port);//tty_wakeup
    69  schedule_work(&port->work);//调度底半部
    70 }
```

20.4.2 USB 键盘驱动

在 Linux 系统中，键盘被认定为标准输入设备，对于一个 USB 键盘而言，其驱动主要由两部分组成：usb_driver 的成员函数和输入设备的打开、关闭、中断处理等函数。

在 USB 键盘设备驱动的模块加载和卸载函数中，将分别注册和注销对应于 USB 键盘的 usb_driver 结构体 usb_kbd_driver，代码清单 20.33 所示为模块加载与卸载函数以及 usb_kbd_driver 结构体的定义。

代码清单 20.33　USB 键盘设备驱动的模块加载与卸载函数及 usb_driver 结构体

```
1  static int __init usb_kbd_init(void)
2  {
3    int result = usb_register(&usb_kbd_driver); //注册 USB 设备驱动
4    if (result == 0)
5      info(DRIVER_VERSION ":" DRIVER_DESC);
6    return result;
7  }
8
9  static void __exit usb_kbd_exit(void)
10 {
11   usb_deregister(&usb_kbd_driver); //注销 USB 设备驱动
12 }
13
14 //usb_driver 结构体
15 static struct usb_driver usb_kbd_driver =
16 {
17   .name = "usbkbd",
18   .probe = usb_kbd_probe,
19   .disconnect = usb_kbd_disconnect,
20   .id_table = usb_kbd_id_table,
21 };
22
23 //支持的设备列表
24 static struct usb_device_id usb_kbd_id_table [] = {
25  { USB_INTERFACE_INFO(3, 1, 1) },
26  { }
27 };
28 MODULE_DEVICE_TABLE (usb, usb_kbd_id_table);
```

在 usb_driver 的探测函数中，将进行 input 设备的初始化和注册，USB 键盘要使用的中断 urb 和控制 urb 的初始化，并设置接口的私有数据，如代码清单 20.34 所示。

代码清单 20.34　USB 键盘设备驱动的探测函数

```
1  static int usb_kbd_probe(struct usb_interface *iface, const struct
2    usb_device_id *id)
3  {
4    struct usb_device *dev = interface_to_usbdev(iface);
5    struct usb_host_interface *interface;
6    struct usb_endpoint_descriptor *endpoint;
7    struct usb_kbd *kbd;
8    struct input_dev *input_dev;
9    int i, pipe, maxp;
```

```

interface = iface->cur_altsetting;
/* 设备是否适合本驱动? */
if (interface->desc.bNumEndpoints != 1)
  return  - ENODEV;

endpoint = &interface->endpoint[0].desc;
if (!(endpoint->bEndpointAddress &USB_DIR_IN))
  return  - ENODEV;
if ((endpoint->bmAttributes &USB_ENDPOINT_XFERTYPE_MASK) !=
  USB_ENDPOINT_XFER_INT)
  return  - ENODEV;

pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);//创建端点的管道
maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));

kbd = kzalloc(sizeof(struct usb_kbd), GFP_KERNEL);
input_dev = input_allocate_device();//分配 input_dev 结构体
if (!kbd || !input_dev)             //分配
  goto fail1;

if (usb_kbd_alloc_mem(dev, kbd))
  goto fail2;

kbd->usbdev = dev;
kbd->dev = input_dev;

if (dev->manufacturer)//制造商名非空
  strlcpy(kbd->name, dev->manufacturer, sizeof(kbd->name));

if (dev->product)//产品名非空
{
  if (dev->manufacturer)
    strlcat(kbd->name, " ", sizeof(kbd->name));
  strlcat(kbd->name, dev->product, sizeof(kbd->name));
}

if (!strlen(kbd->name))
  snprintf(kbd->name, sizeof(kbd->name), "USB HIDBP Keyboard
                %04x:%04x",le16_to_cpu(dev->descriptor.idVendor), le16_to_cpu(dev
    ->descriptor.idProduct));

usb_make_path(dev, kbd->phys, sizeof(kbd->phys));
strlcpy(kbd->phys, "/input0", sizeof(kbd->phys));
/* 输入设备初始化 */
input_dev->name = kbd->name;
```

```
    input_dev->phys = kbd->phys;
    usb_to_input_id(dev, &input_dev->id);
    input_dev->cdev.dev = &iface->dev;
     input_dev->private = kbd;

    input_dev->evbit[0] = BIT(EV_KEY) | BIT(EV_LED) | BIT(EV_REP);
    input_dev->ledbit[0] = BIT(LED_NUML) | BIT(LED_CAPSL) |
     BIT(LED_SCROLLL) |BIT(LED_COMPOSE) | BIT(LED_KANA);

    for (i = 0; i < 255; i++)
     set_bit(usb_kbd_keycode[i], input_dev->keybit);
    clear_bit(0, input_dev->keybit);

    input_dev->event = usb_kbd_event;
    input_dev->open = usb_kbd_open;
    input_dev->close = usb_kbd_close;
    /* 初始化中断 urb */
    usb_fill_int_urb(kbd->irq, dev, pipe, kbd->new, (maxp > 8 ? 8 :
maxp),
     usb_kbd_irq, kbd, endpoint->bInterval);
    kbd->irq->transfer_dma = kbd->new_dma;
    kbd->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

    kbd->cr->bRequestType = USB_TYPE_CLASS | USB_RECIP_INTERFACE;
    kbd->cr->bRequest = 0x09;
    kbd->cr->wValue = cpu_to_le16(0x200);
    kbd->cr->wIndex = cpu_to_le16(interface->desc.bInterfaceNumber);
    kbd->cr->wLength = cpu_to_le16(1);
    /* 初始化控制 urb */
     usb_fill_control_urb(kbd->led, dev, usb_sndctrlpipe(dev, 0),
(void*)kbd->cr,
     kbd->leds, 1, usb_kbd_led, kbd);
    kbd->led->setup_dma = kbd->cr_dma;
    kbd->led->transfer_dma = kbd->leds_dma;
    kbd->led->transfer_flags |= (URB_NO_TRANSFER_DMA_MAP |
     URB_NO_SETUP_DMA_MAP);
    input_register_device(kbd->dev); //注册输入设备

    usb_set_intfdata(iface, kbd);  //设置接口私有数据
    return 0;

    fail2: usb_kbd_free_mem(dev, kbd);
    fail1: input_free_device(input_dev);
    kfree(kbd);
    return  - ENOMEM;
}
```

在 usb_driver 的断开函数中，将设置接口私有数据为 NULL、终止已提交的 urb 并注销输入设备，如代码清单 20.35 所示。

代码清单 20.35　USB 键盘设备驱动的断开函数

```
static void usb_kbd_disconnect(struct usb_interface *intf)
```

```
{
  struct usb_kbd *kbd = usb_get_intfdata(intf);

  usb_set_intfdata(intf, NULL);//设置接口私有数据为 NULL
  if (kbd)
  {
    usb_kill_urb(kbd->irq);//终止 urb
    input_unregister_device(kbd->dev); //注销输入设备
    usb_kbd_free_mem(interface_to_usbdev(intf), kbd);
    kfree(kbd);
  }
}
```

在键盘中断处理函数，也就是中断 urb 的完成函数中，将会通过 input_report_key()报告按键事件，通过 input_sync()报告同步事件，并发起一次新的控制 urb 传输，如代码清单 20.36 所示。

代码清单 20.36　USB 键盘设备驱动的中断 urb 完成函数

```
static void usb_kbd_irq(struct urb *urb, struct pt_regs *regs)
{
  struct usb_kbd *kbd = urb->context;
  int i;

  switch (urb->status)
  {
    case 0:  /* 成功 */
      break;
      case  - ECONNRESET:  /* unlink */
      case  - ENOENT:
      case  - ESHUTDOWN:
      return ;
    default: /* 错误 */
      goto resubmit;
  }
/*获得键盘扫描码并报告按键事件*/
  input_regs(kbd->dev, regs);

  for (i = 0; i < 8; i++)
          input_report_key(kbd->dev,  usb_kbd_keycode[i  +  224], (kbd->new[0] >> i) &1);

  for (i = 2; i < 8; i++)
  {
    if (kbd->old[i] > 3 && memscan(kbd->new + 2, kbd->old[i], 6) ==
      kbd->new +8)
    {
```

```
    28      if (usb_kbd_keycode[kbd->old[i]])
    29          input_report_key(kbd->dev, usb_kbd_keycode[kbd->old[i]],
0);
    30      else
    31        info("Unknown key (scancode %#x) released.", kbd->old[i]);
    32    }
    33
    34    if (kbd->new[i] > 3 && memscan(kbd->old + 2, kbd->new[i], 6) ==
    35      kbd->old +8)
    36    {
    37      if (usb_kbd_keycode[kbd->new[i]])
    38          input_report_key(kbd->dev, usb_kbd_keycode[kbd->new[i]],
1);
    39      else
    40        info("Unknown key (scancode %#x) pressed.", kbd->new[i]);
    41    }
    42  }
    43
    44  input_sync(kbd->dev);报告同步事件
    45
    46  memcpy(kbd->old, kbd->new, 8);
    47
    48  resubmit: i = usb_submit_urb(urb, SLAB_ATOMIC);
    49  if (i)
    50          err("can't  resubmit  intr,  %s-%s/input0,  status  %d",
kbd->usbdev->bus
    51      ->bus_name, kbd->usbdev->devpath, i);
    52 }
```

20.5 总结

USB 驱动分为 USB 主机驱动和 USB 设备驱动，如果系统的 USB 主机控制器符合 OHCI 等标准，这主机驱动的绝大部分工作都可以沿用通用的代码。

对于一个 USB 设备而言，它至少具备两重身份：首先它是“USB”的，其次它是“自己”的。USB 设备是“USB”的，指它挂接在 USB 总线上，其必须完成 usb_driver 的初始化和注册；USB 设备是“自己”的，意味着本身可能是一个字符设备、tty 设备、网络设备等，因此，USB 设备驱动中也必须实现符合相应框架的代码。

USB 设备驱动的自身设备驱动部分的读写等操作流程有其特殊性，即以 URB 来贯穿始终，一个 URB 的生命周期通常包含创建、初始化、提交，和被 USB 核心及 USB 主机传递及完成后回调函数被调用的过程，当然，在 URB 被驱动提交后，也可以被取消。此外，简单的控制及批量消息传递可以用同步的 usb_bulk_msg()、usb_control_msg()函数完成。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm
- 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程： 华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：
  http://www.farsight.com.cn/courses/TS-LinuxApp.htm
- 嵌入式 Linux 系统开发班：
  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm
- 嵌入式 Linux 驱动开发班：
  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm

华清远见

第 21 章　PCI 设备驱动

PCI（及 cPCI）总线在一般的小型手持设备中不太可能用到，但是在工控和通信设备及其 PC 中却引领着潮流。在 Linux 系统中，PCI 设备驱动和 USB 设备驱动有共性，那就是其驱动都由总线相关部分和自身设备类型驱动两部分组成。

21.1 节讲解了 PCI 总线及其配置空间，给出了 PCI 总线在 Linux 内核中的数据结构。

PCI 设备驱动的 PCI 相关部分围绕着 pci_driver 结构体的成员函数展开，21.2 节讲解了 pci_driver 结构体及其成员函数的含义，并分析了 PCI 设备驱动的框架结构。

21.3 节以 Intel 810 主板集成声卡为实例进一步讲解 PCI 设备驱动的编写方法。

华清远见

21.1 PCI 总线与配置空间

21.1.1 PCI 总线的 Linux 描述

从本书第 2 章的图 2.16 可以看出，PCI 总线体系结构是一种层次式的体系结构。在这种层次式体系结构中，PCI 桥设备占据着重要的地位，它将父总线与子总线连接在一起，从而使整个系统看起来像一颗倒置的树型结构。树的顶端是系统的 CPU，它通过一个较为特殊的 PCI 桥设备——Host/PCI 桥设备与根 PCI 总线连接起来。

作为一种特殊的 PCI 设备，PCI 桥包括以下几种。

- Host/PCI 桥：用于连接 CPU 与 PCI 根总线，第 1 个根总线的编号为 0。在 PC 中，内存控制器也通常被集成到 Host/PCI 桥设备芯片中，因此，Host/PCI 桥通常也被称为“北桥芯片组（North Bridge Chipset）”。
- PCI/ISA 桥：用于连接旧的 ISA 总线。通常，PCI 中的类似 i8359A 中断控制器这样的设备也会被集成到 PCI/ISA 桥设备中，因此，PCI/ISA 桥通常也被称为“南桥芯片组（South Bridge Chipset）”。
- PCI-to-PCI 桥：用于连接 PCI 主总线（primary bus）与次总线（secondary bus）。

PCI 桥所处的 PCI 总线称为“主总线”（即次总线的父总线），桥设备所连接的 PCI 总线称为“次总线”（即主总线的子总线）。

在 Linux 系统中，PCI 总线用 pci_bus 来描述，这个结构体记录了本 PCI 总线的信息以及本 PCI 总线的父总线、子总线、桥设备信息，这个结构体的定义如代码清单 21.1 所示。

代码清单 21.1　pci_bus 结构体

```
struct pci_bus
{
  struct list_head node; /* 链表元素 node */
  struct pci_bus *parent; /*指向该 PCI 总线的父总线，即 PCI 桥所在的总线 */
  struct list_head children; /* 描述了这条 PCI 总线的子总线链表的表头 */
  struct list_head devices; /* 描述了这条 PCI 总线的逻辑设备链表的表头 */
  struct pci_dev *self; /* 指向引出这条 PCI 总线的桥设备的 pci_dev 结构 */
  struct resource *resource[PCI_BUS_NUM_RESOURCES];
      /* 指向应路由到这条 PCI 总线的地址空间资源 */

  struct pci_ops *ops; /* 这条 PCI 总线所使用的配置空间访问函数 */
  void *sysdata; /* 指向系统特定的扩展数据 */
  struct proc_dir_entry *procdir;/*该 PCI 总线在/proc/bus/pci 中对应的目录项*/
```

```
14
15  unsigned char number; /* 这条 PCI 总线的总线编号 */
16  unsigned char primary; /* 桥设备的主总线 */
17  unsigned char secondary; /* PCI 总线的桥设备的次总线号 */
18  unsigned char subordinate;/*PCI 总线的下属 PCI 总线的总线编号最大值*/
19
20  char name[48];
21
22  unsigned short bridge_ctl;
23  unsigned short pad2;
24  struct device *bridge;
25  struct class_device class_dev;
26  struct bin_attribute *legacy_io;
27  struct bin_attribute *legacy_mem;
28 };
```

假定一个如图 21.1 所示的 PCI 总线系统，根总线 0 上有一个 PCI 桥，它引出子总线 Bus 1，Bus 1 上又有一个 PCI 桥引出 Bus 2。

图 21.1　示例 PCI 总线系统

在上图中，Bus 0 总线的 pci_bus 结构体中的 number、primary、secondary 都应该为 0，因为它是通过 Host/PCI 桥引出的根总线；Bus 1 总线的 pci_bus 结构体中的 number 和 secondary 都为 1，但是它的 primary 应该为 0；Bus 2 总线的 pci_bus 结构体中的 number 和 secondary 都应该为 2，而其 primary 则应该等于 1。这 3 条总线的 subordinate 值都应该等于 2。

系统中当前存在的所有根总线都通过其 pci_bus 结构体中的 node 成员链接成一条全局的根总线链表，其表头由 list 类型的全局变量 pci_root_buses 来描述。而根总线下面的所有下级总线则都通过其 pci_bus 结构体中的 node 成员链接到其父总线的 children 链表中。这样，通过这两种 PCI 总线链表，Linux 内核就将所有的 pci_bus 结构体以一种倒置树的方式组织起来。假定对于如图 21.2 所示的多根 PCI 总线体系结构，它所对应的总线链表结构将如图 21.3 所示。

图 21.2　多根 PCI 总线体系结构

图 21.3　PCI 总线链表

21.1.2　PCI 设备的 Linux 描述

在 Linux 系统中，所有种类的 PCI 设备都可以用 pci_dev 结构体来描述，由于一个 PCI 接口卡上可能包含多个功能模块，每个功能被当作一个独立的逻辑设备，因此，每一个 PCI 功能，即 PCI 逻辑设备都唯一地对应一个 pci_dev 设备描述符，该结构体

的定义如代码清单 21.2 所示。

代码清单 21.2　pci_dev 结构体

```
struct pci_dev
{
  struct list_head global_list; /* 全局链表元素 */
  struct list_head bus_list; /* 总线设备链表元素 */
  struct pci_bus *bus; /* 这个 PCI 设备所在的 PCI 总线的 pci_bus 结构 */
  struct pci_bus *subordinate; /* 指向这个 PCI 设备所桥接的下级总线 */

  void *sysdata; /* 指向一片特定于系统的扩展数据 */
  struct proc_dir_entry *procent; /* 该 PCI 设备在/proc/bus/pci 中对应的目录项 */

  unsigned int devfn; /* 这个 PCI 设备的设备功能号 */
  unsigned short vendor; /* PCI 设备的厂商 ID*/
  unsigned short device; /* PCI 设备的设备 ID */
  unsigned short subsystem_vendor; /* PCI 设备的子系统厂商 ID */
  unsigned short subsystem_device; /* PCI 设备的子系统设备 ID */
  unsigned int class ; /* 32 位的无符号整数，表示该 PCI 设备的类别，
           bit［7：0］为编程接口，bit［15：8］为子类别代码，bit［23：16］
          为基类别代码，bit［31：24］无意义 */
  u8 hdr_type; /* PCI 配置空间头部的类型 */
  u8 rom_base_reg; /* 表示 PCI 配置空间中的 ROM 基地址寄存器在 PCI 配置空间中的位置 */

  struct pci_driver *driver; /* 指向这个 PCI 设备所对应的驱动 pci_driver 结构 */
  u64 dma_mask; /* 该设备支持的总线地址位掩码，通常是 0xffffffff */

  pci_power_t current_state; /* 当前的操作状态 */

  struct device dev; /* 通用的设备接口 */

  /* 定义这个 PCI 设备与哪些设备相兼容 */
  unsigned short vendor_compatible[DEVICE_COUNT_COMPATIBLE];
  unsigned short device_compatible[DEVICE_COUNT_COMPATIBLE];

  int cfg_size; /* 配置空间大小 */

  unsigned int irq;
  struct resource resource[DEVICE_COUNT_RESOURCE];
            /*表示该设备可能用到的资源，包括:
            I/O 端口区域、设备内存地址区域以及扩展 ROM 地址区域 */

  unsigned int transparent: 1; /* 透明 PCI 桥 */
  unsigned int multifunction: 1; /* 多功能设备 */
  /* keep track of device state */
  unsigned int is_enabled: 1; /* pci_enable_device 已经被调用？ */
  unsigned int is_busmaster: 1; /* 设备是主设备？ */
  unsigned int no_msi: 1; /* 设备可不使用 msi？ */

  u32 saved_config_space[16]; /* 挂起事保存的配置空间 */
  struct bin_attribute *rom_attr; /* sysfs ROM 入口的属性描述 */
  int rom_attr_enabled;
   struct bin_attribute *res_attr[DEVICE_COUNT_RESOURCE];/*资源的 sysfs 文件*/
};
```

在 Linux 系统中，所有的 PCI 设备都通过其 pci_dev 结构体中的 global_list 成员链

接一条全局 PCI 设备链表 pci_devices。另外，同属一条 PCI 总线上的所有 PCI 设备也通过其 pci_dev 结构体中的 bus_list 成员链接成一个属于这条 PCI 总线的总线设备链表，表头则由该 PCI 总线的 pci_bus 结构中的 devices 成员所定义。图 21.4 所示为 PCI 设备链表的典型例子。

图 21.4　PCI 设备链表

21.1.3　PCI 配置空间访问

PCI 有 3 种地址空间：PCI I/O 空间、PCI 内存地址空间和 PCI 配置空间。CPU 可以访问所有的地址空间，其中 PCI I/O 空间和 PCI 内存地址空间由设备驱动程序使用，而 PCI 配置空间由 Linux 内核中的 PCI 初始化代码使用，这些代码用于配置 PCI 设备，比如中断号以及 I/O 或内存基地址。

PCI 规范定义了 3 种类型的 PCI 配置空间头部，其中 type 0 用于标准的 PCI 设备，type 1 用于 PCI 桥，type 2 用于 PCI CardBus 桥。如图 2.17 所示，不管是哪一种类型的配置空间头部，其前 16 个字节的格式都是相同的，/include/linux/pci_regs.h 文件中定义了 PCI 配置空间头部，如代码清单 21.3 所示。

代码清单 21.3　PCI 配置空间头部寄存器定义

```
#define PCI_VENDOR_ID        0x00    /* 16位厂商ID */
#define PCI_DEVICE_ID        0x02    /* 16位设备ID */

/* PCI命令寄存器 */
#define PCI_COMMAND      0x04        /* 16位 */
#define PCI_COMMAND_IO   0x1         /* 使能设备响应对 I/O 空间的访问 */
#define PCI_COMMAND_MEMORY   0x2         /* 使能设备响应对存储空间的访问 */
#define PCI_COMMAND_MASTER   0x4         /* 使能总线主模式 */
#define PCI_COMMAND_SPECIAL  0x8     /* 使能设备响应特殊周期 */
#define PCI_COMMAND_INVALIDATE 0x10 /*使用PCI内存写无效事务 */
#define PCI_COMMAND_VGA_PALETTE 0x20   /* 使能VGA调色板侦测 */
```

```
#define PCI_COMMAND_PARITY    0x40    /* 使能奇偶校验 */
#define PCI_COMMAND_WAIT     0x80  /* 使能地址/数据步进 */
#define PCI_COMMAND_SERR      0x100   /* 使能 SERR */
#define PCI_COMMAND_FAST_BACK  0x200  /* 使能背靠背写 */
#define PCI_COMMAND_INTX_DISABLE 0x400 /* 禁止中断竞争*/

/* PCI 状态寄存器 */
#define PCI_STATUS            0x06   /* 16 位 */
#define PCI_STATUS_CAP_LIST    0x10    /* 支持的能力列表 */
#define PCI_STATUS_66MHZ      0x20    /* 支持 PCI 2.1 66MHz */
#define PCI_STATUS_UDF        0x40    /* 支持用户定义的特征 */
#define PCI_STATUS_FAST_BACK   0x80    /* 快速背靠背操作 */
#define PCI_STATUS_PARITY     0x100   /* 侦测到奇偶校验错 */
#define PCI_STATUS_DEVSEL_MASK 0x600   /* DEVSEL 定时 */
#define PCI_STATUS_DEVSEL_FAST        0x000
#define PCI_STATUS_DEVSEL_MEDIUM      0x200
#define PCI_STATUS_DEVSEL_SLOW        0x400
#define PCI_STATUS_SIG_TARGET_ABORT    0x800 /* 目标设备异常 */
#define PCI_STATUS_REC_TARGET_ABORT    0x1000 /* 主设备确认 */
#define PCI_STATUS_REC_MASTER_ABORT 0x2000 /* 主设备异常 */
#define PCI_STATUS_SIG_SYSTEM_ERROR 0x4000 /* 驱动了 SERR */
#define PCI_STATUS_DETECTED_PARITY  0x8000 /* 奇偶校验错 */

/* 类代码寄存器和修订版本寄存器 */
#define PCI_CLASS_REVISION   0x08  /* 高 24 位为类码，低 8 位为修订版本 */
#define PCI_REVISION_ID  0x08    /* 修订号 */
#define PCI_CLASS_PROG  0x09    /* 编程接口 */
#define PCI_CLASS_DEVICE       0x0a    /* 设备类 */
#define PCI_CACHE_LINE_SIZE    0x0c    /* 8 位 */
#define PCI_LATENCY_TIMER     0x0d    /* 8 位 */

/* PCI 头类型 */
#define PCI_HEADER_TYPE        0x0e    /* 8 位头类型 */
#define PCI_HEADER_TYPE_NORMAL        0
#define PCI_HEADER_TYPE_BRIDGE        1
#define PCI_HEADER_TYPE_CARDBUS       2

/* 表示配置空间头部中的 Built-In Self-Test 寄存器在配置空间中的字节地址索引
*/
#define PCI_BIST              0x0f    /* 8 位 */
```

```
#define PCI_BIST_CODE_MASK     0x0f    /* 完成代码 */
#define PCI_BIST_START         0x40    /* 用于启动 BIST*/
#define PCI_BIST_CAPABLE       0x80    /* 设备是否支持 BIST?  */
```

紧接着前 16 个字节的寄存器为基地址寄存器 0～基地址寄存器 5，其中，PCI_BASE_ADDRESS_ 2～5 仅对标准 PCI 设备的 0 类型配置空间头部有意义，而 PCI_BASE_ADDRESS_0～1 则适用于 0 类型和 1 类型配置空间头部。

基地址寄存器中的 bit［0］的值决定了这个基地址寄存器所指定的地址范围是在 I/O 空间还是在内存映射空间内进行译码。当基地址寄存器所指定的地址范围位于内存映射空间中时，其 bit［2：1］表示内存地址的类型，bit［3］表示内存范围是否为可预取（Prefetchable）的内存。

1 类型配置空间头部适用于 PCI-PCI 桥设备，其基地址寄存器 0 与基地址寄存器 1 可以用来指定桥设备本身可能要用到的地址范围，而后 40 个字节（0x18～0x39）则被用来配置桥设备的主、次编号以及地址过滤窗口等信息。

pci_bus 结构体中的 pci_ops 类型成员指针 ops 指向该 PCI 总线所使用的配置空间访问操作的具体实现，pci_ops 结构体的定义如代码清单 21.4 所示。

代码清单 21.4　pci_ops 结构体

```
struct pci_ops
{
  int(*read)(struct pci_bus *bus, unsigned int devfn, int where, int size, u32
    *val);//读配置空间
  int(*write)(struct pci_bus *bus, unsigned int devfn, int where, int size, u32
    val); //写配置空间
};
```

read()和 write()成员函数中的 size 表示访问的是字节、2 字节还是 4 字节，对于 write()而言，val 是要写入的值；对于 read()而言，val 是要返回的读取到的值的指针。通过 bus 参数的成员以及 devfn 可以定位相应 PCI 总线上相应 PCI 逻辑设备的配置空间。在 Linux 设备驱动中，可用如下一组函数来访问配置空间：

```
inline int pci_read_config_byte(struct pci_dev *dev, int where, u8 *val);
inline int pci_read_config_word(struct pci_dev *dev, int where, u16 *val);
inline int pci_read_config_dword(struct pci_dev *dev, int where, u32 *val);
inline int pci_write_config_byte(struct pci_dev *dev, int where, u8 val);
inline int pci_write_config_word(struct pci_dev *dev, int where, u16 val);
inline int pci_write_config_dword(struct pci_dev *dev, int where, u32 val);
```

上述函数只是对如下函数进行调用：

```
int pci_bus_read_config_byte (struct pci_bus *bus, unsigned int devfn, int where, u8 *val); //读字节
```

```
    int pci_bus_read_config_word (struct pci_bus *bus, unsigned int devfn,
int where, u16 *val); //读字
    int pci_bus_read_config_dword (struct pci_bus *bus, unsigned int devfn,
int where, u32 *val); //读双字
    int pci_bus_write_config_byte (struct pci_bus *bus, unsigned int devfn,
int where, u8 val); //写字节
    int pci_bus_write_config_word (struct pci_bus *bus, unsigned int devfn,
int where, u16 val); //写字
    int pci_bus_write_config_dword (struct pci_bus *bus, unsigned int devfn,
int where, u32 val); //写双字
```

最后，我们来看一下 PCI 总线、设备与驱动在/proc 和/sysfs 文件系统中的描述。首先，通过查看/proc/bus/pci 中的文件，可以获得系统连接的 PCI 设备的基本信息描述。在 VmWare 虚拟机 Linux 上的/proc/bus/pci 目录下的树型结构如下：

```
    |-- 00
    |   |-- 00.0
    |   |-- 01.0
    |   |-- 07.0
    |   |-- 07.1
    |   |-- 07.2
    |   |-- 07.3
    |   |-- 0f.0
    |   |-- 10.0
    |   |-- 11.0
    |   '-- 12.0
    '-- devices
```

sysfs 文件系统/sys/bus/pci 目录中也给出了系统中总线上挂接的设备及驱动信息，该目录下的树型结构如下：

```
    |-- devices
    |   |-- 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
    |   |-- 0000:00:01.0 -> ../../../devices/pci0000:00/0000:00:01.0
    |   |-- 0000:00:07.0 -> ../../../devices/pci0000:00/0000:00:07.0
    |   |-- 0000:00:07.1 -> ../../../devices/pci0000:00/0000:00:07.1
    |   |-- 0000:00:07.2 -> ../../../devices/pci0000:00/0000:00:07.2
    |   |-- 0000:00:07.3 -> ../../../devices/pci0000:00/0000:00:07.3
    |   |-- 0000:00:0f.0 -> ../../../devices/pci0000:00/0000:00:0f.0
    |   |-- 0000:00:10.0 -> ../../../devices/pci0000:00/0000:00:10.0
    |   |-- 0000:00:11.0 -> ../../../devices/pci0000:00/0000:00:11.0
    |   '-- 0000:00:12.0 -> ../../../devices/pci0000:00/0000:00:12.0
    '-- drivers
        |   ...
        |-- PIIX_IDE
        |                                   |--            0000:00:07.1
```

```
-> ../../../../devices/pci0000:00/0000:00:07.1
        |   |-- bind
        |   |-- new_id
        |   '-- unbind
        |   ...
        '-- pcnet32
            |--                                                  0000:00:11.0
-> ../../../../devices/pci0000:00/0000:00:11.0
            |-- bind
            |-- new_id
            '-- unbind
```

此外，pciutils（PCI 工具）中的 lspci 工具会分析/proc/bus/pci 中的文件，从而可被用户用于查看系统中 PCI 设备的描述信息，例如笔者在 VmWare 虚拟机 Linux 上运行 lspci 的结果为：

```
    00:00.0 Host bridge: Intel Corp. 440BX/ZX/DX - 82443BX/ZX/DX Host bridge (rev 01)
    00:01.0 PCI bridge: Intel Corp. 440BX/ZX/DX - 82443BX/ZX/DX AGP bridge (rev 01)
    00:07.0 ISA bridge: Intel Corp. 82371AB/EB/MB PIIX4 ISA (rev 08)
    00:07.1 IDE interface: Intel Corp. 82371AB/EB/MB PIIX4 IDE (rev 01)
    00:07.2 USB Controller: Intel Corp. 82371AB/EB/MB PIIX4 USB
    00:07.3 Bridge: Intel Corp. 82371AB/EB/MB PIIX4 ACPI (rev 08)
    00:0f.0 VGA compatible controller: VMWare Inc: Unknown device 0405
    00:10.0 SCSI storage controller: BusLogic BT-946C (BA80C30) [MultiMaster 10] (rev 01)
    00:11.0 Ethernet controller: Advanced Micro Devices [AMD] 79c970 [PCnet32 LANCE] (rev 10)
    00:12.0 Multimedia audio controller: Ensoniq ES1371 [AudioPCI-97] (rev 02)
```

21.2 PCI 设备驱动结构

21.2.1 pci_driver 结构体

从本质上讲 PCI 只是一种总线，具体的 PCI 设备可以是字符设备、网络设备、USB 主机控制器等，因此，一个通过 PCI 总线与系统连接的设备的驱动至少包含以下两部

分内容。

- PCI 设备驱动。
- 设备本身的驱动。

PCI 驱动只是为了辅助设备本身的驱动，它不是目的，只是手段，PCI 设备本身含有双重以上的身份。例如对于通过 PCI 总线与系统连接的字符设备，则驱动中除了要实现 PCI 驱动部分外，其主体仍然是设备作为字符设备本身的驱动，即实现 file_operations 成员函数并注册 cdev。分析 Linux 内核可知，在/drivers/block/、/drivers/atm/、/drivers/char/、/drivers/i2c/、/drivers/ieee1394/、/drivers/media/、/drivers/mtd/、/drivers/net/、/drivers/serial/、/drivers/video/、/sound/等目录中均广泛分布着 PCI 设备驱动。

在 Linux 内核中，用 pci_driver 结构体来定义 PCI 驱动，该结构体中包含了 PCI 设备的探测/移除、挂起/恢复等函数，其定义如代码清单 21.5 所示。

代码清单 21.5　pci_driver 结构体

```
struct pci_driver
{
  struct list_head node;
  char *name;
  struct module *owner;
  const struct pci_device_id *id_table; /*不能为NULL，以便probe函数调用*/
  /* 新设备添加 */
  int(*probe)(struct pci_dev *dev, const struct pci_device_id *id);
  void(*remove)(struct pci_dev *dev); /* 设备移出 */
  int(*suspend)(struct pci_dev *dev, pm_message_t state); /* 设备挂起 */
  int(*resume)(struct pci_dev *dev); /* 设备唤醒 */
  /* 使能唤醒事件 */
  int(*enable_wake)(struct pci_dev *dev, pci_power_t state, int enable);
  void(*shutdown)(struct pci_dev *dev);

  struct device_driver driver;
  struct pci_dynids dynids;
};
```

对 pci_driver 的注册和注销通过如下函数来实现：

```
int pci_register_driver(struct pci_driver *driver);
void pci_unregister_driver(struct pci_driver *drv);
```

有一个对应于 pci_register_driver()的已经过时的宏定义：

```
#define pci_module_init pci_register_driver
```

在 PCI 设备驱动中其他常用的函数（或宏）如下所示。

- 获取 I/O 或内存资源。

```
#define pci_resource_start(dev,bar)   ((dev)->resource[(bar)].start)
#define pci_resource_end(dev,bar)     ((dev)->resource[(bar)].end)
#define pci_resource_flags(dev,bar)   ((dev)->resource[(bar)].flags)
#define pci_resource_len(dev,bar) \
        ((pci_resource_start((dev),(bar)) == 0 &&  \
         pci_resource_end((dev),(bar)) ==          \
          pci_resource_start((dev),(bar))) ? 0 :   \
                                                    \
         (pci_resource_end((dev),(bar)) -          \
          pci_resource_start((dev),(bar)) + 1))
```

- 申请/释放 I/O 或内存资源。

```
int pci_request_regions(struct pci_dev *pdev, const char *res_name);
void pci_release_regions(struct pci_dev *pdev);
```

- 获取/设置驱动私有数据。

```
void *pci_get_drvdata (struct pci_dev *pdev);
void pci_set_drvdata (struct pci_dev *pdev, void *data);
```

- 使能/禁止 PCI 设备。

```
int pci_enable_device(struct pci_dev *dev);
void pci_disable_device(struct pci_dev *dev);
```

- 设置为总线主 DMA。

```
void pci_set_master(struct pci_dev *dev);
```

- 寻找指定总线指定槽位的 PCI 设备。

```
struct pci_dev *pci_find_slot (unsigned int bus, unsigned int devfn);
```

- 设置 PCI 能量管理状态（0=D0 ... 3=D3）。

```
int pci_set_power_state(struct pci_dev *dev, pci_power_t state);
```

- 在设备的能力表中找出指定的能力。

```
int pci_find_capability (struct pci_dev *dev, int cap);
```

- 启用设备内存写无效事务。

```
int pci_set_mwi(struct pci_dev *dev);
```

- 禁用设备内存写无效事务。

```
void pci_clear_mwi(struct pci_dev *dev);
```

pci_driver 的 probe()函数要完成 PCI 设备的初始化及其设备本身身份（字符、TTY、网络等）驱动的注册。当 Linux 内核启动并完成对所有 PCI 设备进行扫描、登录和分配资源等初始化操作的同时，会建立起系统中所有 PCI 设备的拓扑结构，probe()函数将负责硬件的探测工作并保存配置信息。

drivers/net/pci-skeleton.c 给出了一个 PCI 接口网络设备驱动程序的“骨架”，其 pci_driver 中 probe()成员函数 netdrv_init_one()及其调用的 netdrv_init_board()完成了

PCI 设备初始化及对应的网络设备注册工作，代码清单 21.6 所示为这两个函数的实现。

代码清单 21.6　PCI 设备驱动的 probe()函数

```
static int __devinit netdrv_init_one (struct pci_dev *pdev,
                          const struct pci_device_id *ent)
{
struct net_device *dev = NULL;
struct netdrv_private *tp;
int i, addr_len, option;
void *ioaddr = NULL;
static int board_idx = -1;

  ...

board_idx++;
i = netdrv_init_board (pdev, &dev, &ioaddr);
if (i < 0) {
    return i;
}

...
/* 赋值 net_device 的成员函数 */
dev->open = netdrv_open;
dev->hard_start_xmit = netdrv_start_xmit;
dev->stop = netdrv_close;
dev->get_stats = netdrv_get_stats;
dev->set_multicast_list = netdrv_set_rx_mode;
dev->do_ioctl = netdrv_ioctl;
dev->tx_timeout = netdrv_tx_timeout;
dev->watchdog_timeo = TX_TIMEOUT;

dev->irq = pdev->irq;
dev->base_addr = (unsigned long) ioaddr;
...
pci_set_drvdata(pdev, dev);//设置 PCI 私有数据
...

return 0;
}

static int __devinit netdrv_init_board(struct pci_dev *pdev, struct
```

```
net_device
    **dev_out, void **ioaddr_out)
 {
   void *ioaddr = NULL;
   struct net_device *dev;
   struct netdrv_private *tp;
   int rc, i;
   u32 pio_start, pio_end, pio_flags, pio_len;
   unsigned long mmio_start, mmio_end, mmio_flags, mmio_len;
   u32 tmp;

   *ioaddr_out = NULL;
   *dev_out = NULL;

   /* 分配ethernet设备 */
   dev = alloc_etherdev(sizeof(*tp));
   if (dev == NULL)
   {
     printk(KERN_ERR PFX "unable to alloc new ethernet\n");
     DPRINTK("EXIT, returning -ENOMEM\n");
     return  - ENOMEM;
   }
   SET_MODULE_OWNER(dev);
   SET_NETDEV_DEV(dev, &pdev->dev);
   tp = dev->priv;

   /* 使能设备 */
   rc = pci_enable_device(pdev);
   if (rc)
     goto err_out;

   /* 获取I/O和内存基地址 */
   pio_start = pci_resource_start(pdev, 0);
   pio_end = pci_resource_end(pdev, 0);
   pio_flags = pci_resource_flags(pdev, 0);
   pio_len = pci_resource_len(pdev, 0);

   mmio_start = pci_resource_start(pdev, 1);
```

```
mmio_end = pci_resource_end(pdev, 1);
mmio_flags = pci_resource_flags(pdev, 1);
mmio_len = pci_resource_len(pdev, 1);

DPRINTK("PIO region size == 0x%02X\n", pio_len);
DPRINTK("MMIO region size == 0x%02lX\n", mmio_len);

/* 确保 PCI 基地址寄存器 0 是 PIO */
if (!(pio_flags &IORESOURCE_IO))
{
  printk(KERN_ERR PFX "region #0 not a PIO resource, aborting\n");
  rc = - ENODEV;
  goto err_out;
}

/* 确保 PCI 基地址寄存器 1 是 MMIO */
if (!(mmio_flags &IORESOURCE_MEM))
{
  printk(KERN_ERR PFX "region #1 not an MMIO resource, aborting\n");
  rc = - ENODEV;
  goto err_out;
}

/* 检查 weird/broken PCI 区域报告 */
if ((pio_len < NETDRV_MIN_IO_SIZE) || (mmio_len < NETDRV_MIN_IO_SIZE))
{
  printk(KERN_ERR PFX "Invalid PCI region size(s), aborting\n");
  rc = - ENODEV;
  goto err_out;
}

rc = pci_request_regions(pdev, "pci-skeleton");
if (rc)
  goto err_out;

pci_set_master(pdev);

#ifdef USE_IO_OPS
```

```
    ioaddr = (void*)pio_start;
  #else
    /* ioremap MMIO 区域 */
    ioaddr = ioremap(mmio_start, mmio_len);
    if (ioaddr == NULL)
    {
      printk(KERN_ERR PFX "cannot remap MMIO, aborting\n");
      rc = - EIO;
      goto err_out_free_res;
    }
  #endif /* USE_IO_OPS */

  /* 软重启芯片 */
      NETDRV_W8(ChipCmd, (NETDRV_R8(ChipCmd) &ChipCmdClear) |
CmdReset);

  /* 检查芯片已完成复位，让芯片进入低能耗模式 */
  /* <在此插入设备特定的代码> */
  ...

  /* 注册网络设备 */
  i = register_netdev(dev);
  if (i)
    goto err_out_unmap;

  DPRINTK("EXIT, returning 0\n");
  *ioaddr_out = ioaddr;
  *dev_out = dev;
  return 0;

  err_out_unmap:
  ...
}
```

从上述代码可以看出，probe()函数中的主体分为两个部分，即 PCI 配置部分和网络设备初始化和注册部分。pci_driver 的 remove()成员函数完成相反的工作，即释放 PCI 设备和网络设备，如代码清单 21.7 所示。

代码清单 21.7　PCI 设备驱动的 remove()函数

```
1  static void __devexit netdrv_remove_one (struct pci_dev *pdev)
2  {
3   struct net_device *dev = pci_get_drvdata (pdev);
4   struct netdrv_private *np;
5
6   np = dev->priv;
7
8   unregister_netdev (dev);//注销网络设备
9  #ifndef USE_IO_OPS
10  iounmap (np->mmio_addr);//iounmap
11 #endif /* !USE_IO_OPS */
12
13  pci_release_regions (pdev);//释放 pci 区域
14  free_netdev (dev);//释放 ethernet
15  pci_set_drvdata (pdev, NULL);
16  pci_disable_device (pdev);//禁止 PCI 设备
17 }
```

如同在 USB 设备驱动中定义 usb_device_id 结构体数组一样，在 PCI 设备驱动中，也需要定义一个 pci_device_id 结构体数组并导出到用户空间，使热插拔和模块装载系统知道驱动模块所针对的硬件设备。pci_device_id 结构体的定义如代码清单 21.8 所示。

代码清单 21.8　pci_device_id 结构体

```
1 struct pci_device_id {
2          __u32 vendor, device;         /* 厂商和设备 ID 或 PCI_ANY_ID*/
3          __u32 subvendor, subdevice;   /* 子系统 ID 或 PCI_ANY_ID */
4          __u32 class, class_mask;      /* (类、子类、prog-if) 三元组 */
5          kernel_ulong_t driver_data;   /* 驱动私有数据 */
6 };
```

其中的 PCI_ANY_ID 定义为~0，即对任意 ID 都适用。pci_device_id 结构体数组使用宏 MODULE_DEVICE_TABLE 导出到用户空间。/drivers/net/pci-skeleton.c 中定义的 pci_device_id 结构体数组及 MODULE_DEVICE_TABLE 导出代码如清单 21.9 所示。

代码清单 21.9　PCI 设备驱动的 pci_device_id 数组及 MODULE_DEVICE_TABLE

```
1  static struct pci_device_id netdrv_pci_tbl[] = {
2   {0x10ec, 0x8139, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
3   {0x10ec, 0x8138, PCI_ANY_ID, PCI_ANY_ID, 0, 0, NETDRV_CB },
4   {0x1113, 0x1211, PCI_ANY_ID, PCI_ANY_ID, 0, 0, SMC1211TX },
5  /*   {0x1113, 0x1211, PCI_ANY_ID, PCI_ANY_ID, 0, 0, MPX5030 },*/
6   {0x1500, 0x1360, PCI_ANY_ID, PCI_ANY_ID, 0, 0, DELTA8139 },
7   {0x4033, 0x1360, PCI_ANY_ID, PCI_ANY_ID, 0, 0, ADDTRON8139 },
8   {0,}
```

```
};
MODULE_DEVICE_TABLE (pci, netdrv_pci_tbl);
```

21.2.2 PCI 设备驱动的组成

下面以一个 PCI 接口字符设备为例，给出 PCI 设备驱动的完整模板。其一部分代码实现 pci_driver 成员函数，一部分代码实现字符设备的 file_operations 成员函数，如代码清单 21.10 所示。

代码清单 21.10　PCI 设备驱动的程序模板

```
/* 指明该驱动程序适用于哪一些PCI设备 */
static struct pci_device_id xxx_pci_tbl [] __initdata = {
    {PCI_VENDOR_ID_DEMO, PCI_DEVICE_ID_DEMO,
     PCI_ANY_ID, PCI_ANY_ID, 0, 0, DEMO},
    {0,}
};
MODULE_DEVICE_TABLE(pci, xxx_pci_tbl);

/* 中断处理函数 */
static void xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    ...
}

/* 字符设备file_operations open成员函数 */
static int xxx_open(struct inode *inode, struct file *file)
{
    /* 申请中断，注册中断处理程序 */
    request_irq(xxx_irq, &xxx_interrupt, ...));
    ...
}

/* 字符设备file_operations ioctl成员函数 */
static int xxx_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
   ...
}

```

```
... /* 字符设备 file_operations read、write、mmap 等成员函数 */

/* 设备文件操作接口 */
static struct file_operations xxx_fops = {
        owner:      THIS_MODULE,        /* xxx_fops 所属的设备模块 */
        read:       xxx_read,       /* 读设备操作*/
        write:      xxx_write,          /* 写设备操作*/
        ioctl:      xxx_ioctl,          /* 控制设备操作*/
        mmap:       xxx_mmap,           /* 内存重映射操作*/
        open:       xxx_open,           /* 打开设备操作*/
        release:    xxx_release         /* 释放设备操作*/
       /* ... */
};

/* pci_driver 的 probe 成员函数 */
static int __init xxx_probe(struct pci_dev *pci_dev, const struct pci_device_id *pci_id)
{
          pci_enable_device(pci_dev); //启动 PCI 设备

           /* 读取 PCI 配置信息 */
          iobase = pci_resource_start (pci_dev,1);
          ...

          pci_set_master(pci_dev);//设置成总线主 DMA 模式

          pci_request_regions(pci_dev);//申请 I/O 资源

          /* 注册字符设备 */
          cdev_init(xxx_cdev,&xxx_fops);
          register_chrdev_region(xxx_dev_no, 1, ...);
          cdev_add(xxx_cdev);

          return 0;
}

/* pci_driver 的 remove 成员函数 */
static int __init xxx_release(struct pci_dev *pdev)
{
        pci_release_regions(pdev);//释放 I/O 资源
```

```
68        pci_disable_device (pdev);//禁止 PCI 设备
69        unregister_chrdev_region(xxx_dev_no, 1); //释放占用的设备号
70        cdev_del(&xxx_dev.cdev); //注销字符设备
71        ...
72        return 0;
73 }
74
75 /* 设备模块信息 */
76 static struct pci_driver xxx_pci_driver = {
77        name:      xxx_MODULE_NAME,          /* 设备模块名称 */
78        id_table:   xxx_pci_tbl,             /* 能够驱动的设备列表 */
79        probe:     xxx_probe,                /* 查找并初始化设备 */
80        remove:    xxx_remove            /* 卸载设备模块 */
81 };
82
83 static int __init xxx_init_module (void)
84 {
85        pci_register_driver(&xxx_pci_driver);
86 }
87 static void __exit xxx_cleanup_module (void)
88 {
89        pci_unregister_driver(&xxx_pci_driver);
90 }
91 /*驱动模块加载函数 */
92 module_init(xxx_init_module);
93 /* 驱动模块卸载函数 */
94 module_exit(xxx_cleanup_module);
```

将代码清单 21.10 中的各个函数进行归类，可得出该驱动的组成，如图 21.5 所示。

图 21.5　PCI 字符设备驱动的组成

图 21.6 所示的树中，树根是主机/PCI 桥，树叶是具体的 PCI 设备，树叶本身与树枝通过 pci_driver 连接，而树叶本身的驱动，读写、控制树叶则需要通过其树叶设备本身所属类设备驱动来完成。

由此我们看出，对于 USB、PCI 设备这种挂接在总线上的设备而言，USB、PCI 只是它们的“工作单位”，它们需要向“工作单位”注册（注册 usb_driver、pci_driver），

并接收“工作单位”的管理（被调入 probe()、调出 disconnect()/remove()、放假 shutdown()、继续上班 resume()等），但是其本身可能是一个工程师、一个前台或一个经理，因此，做好工程师、前台或经理是其主体工作，这部分对应于字符设备驱动、tty 设备驱动、网络设备驱动等。

图 21.6　PCI 设备驱动结构

21.2.3　旧版内核的 PCI 设备探测

21.1.1~21.1.2 节给出的 PCI 设备驱动通过 pci_register_driver()注册 pci_driver，从而导致其 probe()成员函数被调用。但是，在 Linux 2.6 之前的内核中，PCI 设备的探测主要依靠驱动程序在模块加载函数中调用如下函数来完成：

```
struct pci_dev *pci_find_device (unsigned int vendor, unsigned int device, const struct pci_dev *from);
struct pci_dev *pci_find_device_reverse (unsigned int vendor, unsigned int device, const struct pci_dev *from);
```

pci_find_device()、pci_find_device_reverse()函数在设备列表中查找匹配厂商及设备 ID 的 PCI 设备，若是第 1 次搜索，则 from 置 NULL，若是第 *n* 次搜索，则使用第 *n*−1 次搜索获得的 pci_dev。

```
struct pci_dev * pci_find_subsys (unsigned int vendor, unsigned int device, unsigned int ss_vendor, unsigned int ss_device, const struct pci_dev * from);
```

该函数用于查找匹配厂商及设备 ID、子系统厂商及设备 ID 的 PCI 设备。

```
struct pci_dev * pci_find_class (unsigned int class, const struct pci_dev * from);
```

该函数用于查找参数 class 类 PCI 设备。

旧的 PCI 设备驱动模块加载函数通常采用如图 21.7 所示的流程，代码清单 21.11 则给出了相应的函数模板。

代码清单 21.11　旧版内核下 PCI 探测函数范例

```
1  int xxx_init_module()
2  {
3    int ret;
4    struct pci_dev *pdev = NULL;
5    struct XXX_DEV *dev;
6    //判断是否有总线
7    if (!pci_present())
8      return  - ENODEV;
9    // PCI
10   //查找该类卡
11     while  ((pdev  =  (struct  pci_dev*)pci_find_device(VENDOR_ID,
DEVICE_ID, pdev)))
12     dev = &my_device[dev_num];
13   //得到中断号和基址
14   dev->irq = pdev->irq;
15   dev->iobase = pdev->base_address[0] &PCI_BASE_ ADDRESS_IO_MASK;
16   //检查地址空间是否可用
17   if (check_region(dev->iobase, DEVICE_IO_EXTENT) < 0)
18     continue;
19   //注册中断处理函数
20   if (request_irq(dev->irq, my_interrupt, SA_SHIRQ | SA_INTERRUPT,
dev
21     ->dev_name, dev))
22     return  - EAGAIN;
23   //申请空间
24   request_region(pci_ioaddr, DEVICE_IO_EXTENT, dev->dev _name);
25   ...
26   //注册该设备
27   ret = register_xxxdev(...);//字符、USB、网络等
28   ...
29 }
```

图 21.7　旧的 PCI 设备驱动的模块加载函数流程

在 Linux 2.6 内核中，不宜再采用这些函数搜寻系统中的 PCI 设备，如果驱动程序中确实需要查找特定的 PCI 设备，则可以使用如下函数：

```
struct pci_dev *pci_get_device (unsigned int vendor, unsigned int device,
struct pci_dev *from);
```

这个函数从系统中当前存在的 PCI 设备列表中寻找与 vendor 和 device 参数匹配的 PCI 设备，如果找到，就增加对应 pci_dev 的引用计数并返回该结构体。使用 pci_get_device()函数后，记住要调用 pci_put_dev()函数减小使用计数。

类似于 pci_get_device()的函数还包括：

```
struct pci_dev *pci_get_subsys (unsigned int vendor, unsigned int device,
unsigned int ss_vendor, unsigned int ss_device, struct pci_dev *from);
```

ss_vendor 参数为子系统厂商 ID，ss_device 参数为子系统设备 ID。因此这个函数用于从系统中当前存在的 PCI 设备列表中寻找与 vendor、device、ss_vendor 和 ss_device 参数匹配的 PCI 设备。

```
struct pci_dev *pci_get_slot (struct pci_bus *bus, unsigned int devfn);
```

这个函数在指定的 PCI 总线上查找指定功能编号的 PCI 设备。

pci_get_device()、pci_get_subsys()和 pci_get_slot()函数只是查找的限制条件不一样，它们都会增加 PCI 设备的引用计数，因此，在调用者结束使用 pci_dev 之前，它必须调用 pci_dev_put()函数减少引用计数，如代码清单 21.12 所示。

代码清单 21.12　pci_get_device 等函数的使用

```
1 struct pci_dev *dev;
2 dev = pci_get_device(PCI_VENDOR_FOO, PCI_DEVICE_FOO, NULL);
3 if (dev)
4 {
```

```
5   /* 使用PCI设备 */
6   ...
7   pci_dev_put(dev); /* 减小引用计数 */
8 }
```

21.3 实例：Intel 810 主板声卡驱动

Intel 810 主板的集成声卡符合 AC97 标准，其驱动的 PCI 相关部分包含 pci_driver 的定义、支持设备列表的 pci_device_id 数组定义，对 pci_driver 的注册和注销发生在模块加载函数与卸载函数中，代码清单 21.13 所示为这一部分的实现。

代码清单 21.13　Intel 810 主板声卡驱动的 PCI 数据结构定义及模块加载与卸载函数

```
1  /* pci_driver结构体 */
2  static struct pci_driver i810_pci_driver =
3  {
4   .name       = I810_MODULE_NAME,
5   .id_table   = i810_pci_tbl,
6   .probe      = i810_probe,
7   .remove     = __devexit_p(i810_remove),
8  #ifdef CONFIG_PM
9   .suspend    = i810_pm_suspend,
10  .resume     = i810_pm_resume,
11 #endif /* CONFIG_PM */
12 };
13
14 /* 支持的设备列表 */
15 static struct pci_device_id i810_pci_tbl [] = {
16  {PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82801AA_5,
17   PCI_ANY_ID, PCI_ANY_ID, 0, 0, ICH82801AA},
18  {PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82801AB_5,
19   PCI_ANY_ID, PCI_ANY_ID, 0, 0, ICH82901AB},
20  {PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_440MX,
21   PCI_ANY_ID, PCI_ANY_ID, 0, 0, INTEL440MX},
22  {PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82801BA_4,
23   PCI_ANY_ID, PCI_ANY_ID, 0, 0, INTELICH2},
24  {PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82801CA_5,
```

```
 PCI_ANY_ID, PCI_ANY_ID, 0, 0, INTELICH3},
{PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82801DB_5,
 PCI_ANY_ID, PCI_ANY_ID, 0, 0, INTELICH4},
{PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82801EB_5,
 PCI_ANY_ID, PCI_ANY_ID, 0, 0, INTELICH5},
{PCI_VENDOR_ID_SI, PCI_DEVICE_ID_SI_7012,
 PCI_ANY_ID, PCI_ANY_ID, 0, 0, SI7012},
{PCI_VENDOR_ID_NVIDIA, PCI_DEVICE_ID_NVIDIA_MCP1_AUDIO,
 PCI_ANY_ID, PCI_ANY_ID, 0, 0, NVIDIA_NFORCE},
{PCI_VENDOR_ID_NVIDIA, PCI_DEVICE_ID_NVIDIA_MCP2_AUDIO,
 PCI_ANY_ID, PCI_ANY_ID, 0, 0, NVIDIA_NFORCE},
{PCI_VENDOR_ID_NVIDIA, PCI_DEVICE_ID_NVIDIA_MCP3_AUDIO,
 PCI_ANY_ID, PCI_ANY_ID, 0, 0, NVIDIA_NFORCE},
{PCI_VENDOR_ID_AMD, PCI_DEVICE_ID_AMD_OPUS_7445,
 PCI_ANY_ID, PCI_ANY_ID, 0, 0, AMD768},
{PCI_VENDOR_ID_AMD, PCI_DEVICE_ID_AMD_8111_AUDIO,
 PCI_ANY_ID, PCI_ANY_ID, 0, 0, AMD8111},
{PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_ESB_5,
 PCI_ANY_ID, PCI_ANY_ID, 0, 0, INTELICH4},
{PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_ICH6_18,
 PCI_ANY_ID, PCI_ANY_ID, 0, 0, INTELICH4},
{PCI_VENDOR_ID_NVIDIA,  PCI_DEVICE_ID_NVIDIA_CK804_AUDIO,
 PCI_ANY_ID, PCI_ANY_ID, 0, 0, NVIDIA_NFORCE},
{0,}
};

/* 模块加载，注册 pci_driver */
static int __init i810_init_module (void)
{
 int retval;

 printk(KERN_INFO "Intel 810 + AC97 Audio, version "
         DRIVER_VERSION ", " __TIME__ " " __DATE__ "\n");

 retval = pci_register_driver(&i810_pci_driver);
 if (retval)
     return retval;

 if(ftsodell != 0) {
     printk("i810_audio: ftsodell is now a deprecated option.\n");
```

```
}
if(spdif_locked > 0 ) {
    if(spdif_locked == 32000 || spdif_locked == 44100 || spdif_locked == 48000) {
        printk("i810_audio: Enabling S/PDIF at sample rate %dHz.\n", spdif_locked);
    } else {
        printk("i810_audio: S/PDIF can only be locked to 32000, 44100, or 48000Hz.\n");
        spdif_locked = 0;
    }
}

return 0;
}

/* 模块卸载，注销 pci_driver */
static void __exit i810_cleanup_module (void)
{
 pci_unregister_driver(&i810_pci_driver);
}

module_init(i810_init_module);
module_exit(i810_cleanup_module);
```

在 pci_driver 的 probe()成员函数中，将分析声卡占用的 I/O 资源、I/O 内存资源及其 IRQ，并申请这些资源，之后会注册 OSS 声卡驱动的/dev/dsp 接口和/dev/mixer 接口，如代码清单 20.14 所示。

代码清单 20.14　Intel 810 声卡驱动的探测函数

```
static int __devinit i810_probe(struct pci_dev *pci_dev, const struct
  pci_device_id *pci_id)
{
  struct i810_card *card;

  if (pci_enable_device(pci_dev))  //使能 PCI 设备
    return  - EIO;

```

```
    /* 设置设备的DMA掩码 */
    if (pci_set_dma_mask(pci_dev, I810_DMA_MASK))
    {
      printk(KERN_ERR "i810_audio: architecture does not support"
        " 32bit PCI busmaster DMA\n");
      return  - ENODEV;
    }

    /* 分配全局数据结构i810_card的内存 */
     if ((card = kmalloc(sizeof(struct i810_card), GFP_KERNEL)) ==
NULL)
    {
      printk(KERN_ERR "i810_audio: out of memory\n");
      return  - ENOMEM;
    } memset(card, 0, sizeof(*card));

    card->initializing = 1;
    card->pci_dev = pci_dev;
    card->pci_id = pci_id->device;
    /* 寄存器地址 */
    card->ac97base = pci_resource_start(pci_dev, 0);
    card->iobase = pci_resource_start(pci_dev, 1);
    if (!(card->ac97base) || !(card->iobase))
    {
      card->ac97base = 0;
      card->iobase = 0;
    }

    /* 如果芯片有mmio能力，检查它 */
    if (card_cap[pci_id->driver_data].flags &CAP_MMIO)
    {
      card->ac97base_mmio_phys = pci_resource_start(pci_dev, 2);
      card->iobase_mmio_phys = pci_resource_start(pci_dev, 3);

      if ((card->ac97base_mmio_phys) && (card->iobase_mmio_phys))
      {
        card->use_mmio = 1;
      }
      else
      {
```

```
    48        card->ac97base_mmio_phys = 0;
    49        card->iobase_mmio_phys = 0;
    50      }
    51    }
    52
    53    /*I/O 资源不可获得 */
    54    if (!(card->use_mmio) && (!(card->iobase) || !(card->ac97base)))
    55    {
    56      printk(KERN_ERR "i810_audio: No I/O resources available.\n");
    57      goto out_mem;
    58    }
    59
    60    card->irq = pci_dev->irq;
    61    card->next = devs;
    62    card->magic = I810_CARD_MAGIC;
    63    #ifdef CONFIG_PM
    64      card->pm_suspended = 0;
    65    #endif
    66    spin_lock_init(&card->lock);
    67    spin_lock_init(&card->ac97_lock);
    68    devs = card;
    69
    70    pci_set_master(pci_dev); //设置为总线主 DMA
    71
    72    printk(KERN_INFO "i810: %s found at IO 0x%04lx and 0x%04lx, "
    73                    "MEM   0x%04lx   and   0x%04lx,   IRQ   %d\n",
card_names[pci_id->driver_data], card
    74      ->iobase, card->ac97base, card->ac97base_mmio_phys,
    75      card->iobase_mmio_phys,card->irq);
    76
    77    card->alloc_pcm_channel = i810_alloc_pcm_channel;
    78    card->alloc_rec_pcm_channel = i810_alloc_rec_pcm_channel;
    79    card->alloc_rec_mic_channel = i810_alloc_rec_mic_channel;
    80    card->free_pcm_channel = i810_free_pcm_channel;
    81
    82    if ((card->channel = pci_alloc_consistent(pci_dev, sizeof(struct
    83      i810_channel)*NR_HW_CH, &card->chandma)) == NULL)
    84    {
```

```
    85      printk(KERN_ERR "i810: cannot allocate channel DMA memory\n");
    86      goto out_mem;
    87    }
    88
    89    ...
    90    /* 申请 I/O 资源 */
    91                    if     (!request_region(card->iobase,     64,
card_names[pci_id->driver_data]))
    92    {
    93      printk(KERN_ERR "i810_audio: unable to allocate region %lx\n",
    94        card->iobase);
    95      goto out_region1;
    96    }
    97                  if     (!request_region(card->ac97base,     256,
card_names[pci_id->driver_data]))
    98    {
    99      printk(KERN_ERR "i810_audio: unable to allocate region %lx\n",
card
    100       ->ac97base);
    101     goto out_region2;
    102   }
    103
    104   if (card->use_mmio)
    105   {
    106     /* 申请 I/O 内存资源 */
    107          if  (request_mem_region(card->ac97base_mmio_phys,  512,
"ich_audio MMBAR"))
    108     {
    109       ...
    110     }
    111     else
    112     {
    113       card->use_mmio = 0;
    114     }
    115   }
    116
    117   /* 初始化 AC97 codec 并注册/dev/mixer 接口 */
    118   if (i810_ac97_init(card) <= 0)
    119     goto out_iospace;
    120   pci_set_drvdata(pci_dev, card); //设置 PCI 设备私有数据
```

```
121
122  if (clocking == 0)
123  {
124    clocking = 48000;
125    i810_configure_clocking();
126  }
127
128  /* 注册/dev/dsp */
129  if ((card->dev_audio = register_sound_dsp(&i810_audio_fops,  -
1)) < 0)
130  {
131    int i;
132    printk(KERN_ERR "i810_audio: couldn't register DSP device!\n");
133    for (i = 0; i < NR_AC97; i++)
134    if (card->ac97_codec[i] != NULL)
135    {
136      unregister_sound_mixer(card->ac97_codec[i]->dev_mixer);
137      ac97_release_codec(card->ac97_codec[i]);
138    }
139    goto out_iospace;
140  }
141
142  /* 申请 irq，注册中断处理程序 */
143      if  (request_irq(card->irq,  &i810_interrupt,  SA_SHIRQ,
card_names[pci_id
144    ->driver_data], card))
145  {
146     printk(KERN_ERR "i810_audio: unable to allocate irq %d\n",
card->irq);
147    goto out_iospace;
148  }
149
150  card->initializing = 0;
151  return 0;
152  ...
153 }
```

在 pci_driver 的 remove()成员函数中，将释放声卡所占用的 I/O 资源、I/O 内存资源及其 IRQ，之后会注销 OSS 声卡驱动的/dev/dsp 接口和/dev/mixer 接口，如代码清单 20.15 所示。

代码清单 20.15　Intel 810 声卡驱动的移除函数

```
1  static void __devexit i810_remove(struct pci_dev *pci_dev)
2  {
3    int i;
4    struct i810_card *card = pci_get_drvdata(pci_dev);
5    /* 释放硬件资源 */
6    free_irq(card->irq, devs);
7    release_region(card->iobase, 64);
8    release_region(card->ac97base, 256);
9        pci_free_consistent(pci_dev,  sizeof(struct  i810_channel)
*NR_HW_CH, card
10     ->channel, card->chandma);
11   if (card->use_mmio)
12   {
13     iounmap(card->ac97base_mmio);
14     iounmap(card->iobase_mmio);
15     release_mem_region(card->ac97base_mmio_phys, 512);
16     release_mem_region(card->iobase_mmio_phys, 256);
17   }
18
19   /* 注销/dev/mixer 和/dev/dsp */
20   for (i = 0; i < NR_AC97; i++)
21   if (card->ac97_codec[i] != NULL)
22   {
23     unregister_sound_mixer(card->ac97_codec[i]->dev_mixer);
24     ac97_release_codec(card->ac97_codec[i]);
25     card->ac97_codec[i] = NULL;
26   }
27   unregister_sound_dsp(card->dev_audio);
28   kfree(card);
29 }
```

代码清单 20.14 第 129 行注册 dsp 接口时，给出的参数 i810_audio_fops 是 dsp 接口文件操作结构体，内部包含了 dsp 接口的 i810_open()、i810_release()、i810_read()、i810_write()、i810_poll()、i810_ioctl()等成员函数。118 行调用的 i810_ac97_init()函数会通过 register_sound_mixer(&i810_mixer_fops, -1))注册/dev/mixer 接口，其内部包含了 i810_open_mixdev()、i810_ioctl_mixdev()函数。这些函数的实现与第 17 章所讲解的结构非常相似，由于本章的重点是讲解 PCI 设备驱动的 PCI 相关部分，因此，对于 OSS 驱动部分不再赘述。

由以上分析可知，Intel 810 主板集成声卡的驱动主要由两部分组成，在驱动的模块加载和卸载函数中，会分别注册和注销 pci_driver，pci_driver 的探测和移除函数会分别申请/释放资源、注册/注销/dev/dsp 和/dev/mixer 接口。Intel 810 主板集成声卡的 OSS 驱动部分主要包含/dev/dsp、/dev/mixer 接口的 file_operations 成员实现，如图 21.8 所示。

图 21.8　Intel 810 主板集成声卡驱动组成

21.4 总结

PCI 设备驱动只是字符设备、tty 设备、网络设备、音频设备等与系统的一个接口，因此，驱动将由两部分组成，一部分是 PCI 相关部分，另一部分是设备本身所属类驱动。PCI 驱动的核心数据结构是 pci_driver，在 probe()成员函数中将申请资源并注册对应的字符设备、tty 设备、网络设备、音频设备等，而 remove()成员函数中将释放资源并注销对应的字符设备、tty 设备、网络设备、音频设备等。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm

第 22 章　Linux 设备驱动的调试

“工欲善其事，必先利其器”，为了方便进行 Linux 设备驱动的开发和调试，建立良好的开发环境很重要，包括实验室环境建设、使用必要的工具软件以及掌握常用的调试技巧等。

22.1 节介绍 Linux 开发环境的建设，包括实验室配置、工具链、串口工具等。

22.2 节讲解了 Linux 下调试器 gdb 的基本用法和技巧。

22.3 节讲解了 Linux 内核的调试方法，22.4～22.9 节对 22.3 节的概述展开讲解，分别讲解了 Linux 内核调试用到的 printk()、/proc、oops、监视工具，kcore、kdb 和 kgdb，以及使用仿真器进行调试的方法。

22.10 节讲解了 Linux 应用程序的调试方法，驱动工程师往往需要编写用户空间的应用程序对自身编写的驱动进行验证和测试，因此，掌握应用程序调试方法对驱动工程师而言也是必须的。

22.1 Linux 开发环境建设

22.1.1 实验室建设

在公司或学校的实验室中，PC 的性能一般来说不会太高，用 PC 来编译 Linux 内核和模块的速度总是受限。相反地，服务器的资源相对比较充分，CPU 以及磁盘性能都较高，因此在服务器上进行内核、驱动及应用程序的编译开发都将更加快捷，而且使用服务器更便于统一管理实验室内的所有开发者。图 22.1 所示为一种常见的小型 Linux 实验室环境。

图 22.1　Linux 实验室环境

Linux 服务器上启动了 Samba 和 sshd 进程，各工程师在自己的 Windows 或 Linux 客户机上通过 SSH 用自己的用户名和密码登录服务器便可以使用服务器上的 GCC、GDB 等软件（Windows 下 SSH Secure Shell 界面如图 22.2 所示）。同时，SSH 软件提供了类似于 FTP 的文件共享功能（Windows 下 SSH Secure File Transfer 界面如图 22.3 所示），方便在客户端和服务器端复制文件。

目标板、服务器和客户端全部通过交换机连接，同时客户端连接目标板的串口作为控制台。在调试 Linux 应用程序时，目标板 erver 与调试用的 GDB，目标板与服务器的 NFS 挂接都借助网络通信解决。编写完成的应用程序或内核模块可直接存放在服务器的 NFS 服务目录内，而该目录可被目标板上的 Linux 系统 mount 到本身的一个目

录内。

华清远见

图 22.2　SSH Secure Shell

图 22.3　SSH Secure File Transfer

22.1.2　工具链

为了编译、连接并调试 Linux 应用程序和内核，我们首先需要建立针对目标板处

理器的 GNU 工具链。

GNU 工具链有力地支撑了 Linux 系统的发展，由于它可被看作许多嵌入式处理器的一个交叉编译器，所以在嵌入式软件开发中相当流行，其支持包括 ARM、StrongARM、XScale、PowerPC 、MIPS、68K/ColdFire、Intel x86/IA-32、Intel i960、Hitachi SH 在内的多种体系结构。GNU 工具链中大多数有用的工具主要集中于以下几个源代码包中。

- GCC 包，主要包括 gcc（C 编译器）、g++（C++编译器）、cpp（C 预处理器）。
- Binutils (binary utilities) 包，主要包括 as（汇编程序）、ld（连接器）、objcopy（目标文件翻译器，用于从连接器输出中创建一个 ROM 映像）、objdump（目标文件阅读器，用于反汇编目标文件）。
- glibc/uclibc/newlib，提供系统调用和基本函数的 C 库，比如 open()、malloc()、printf()等。
- Make，主要包括 make 工具。
- Debugger，主要包括 gdb（源代码调试器）。

上述源代码包都可以从 gnu FTP 站点上直接下载，如为了建立 ARM Linux 的 GNU 工具链，我们需下载 newlib、binutils、gcc 和 gdb 代码包。在解压缩、针对特定处理器配置、编译并安装这样软件包后，便可以使用它们进行交叉编译与开发。配置、编译与安装的所需执行的命令步骤如下：

```
    (1)cd [binutils-build]
    (2)[binutils-source]/configure                        --target=arm-elf
--prefix=[toolchain-prefix] --enable-
interwork --enable-multilib
    (3)make all install
    (4)export PATH="$PATH:[toolchain-prefix]/bin"
    (5)cd [gcc-build]
    (6)[gcc-source]/configure  --target=arm-elf  --prefix=[toolchain-prefix]
--enable-interwork --
enable-multilib           --enable-languages="c,c++"           --with-newlib
--with-headers=[newlib-source]/newlib/
libc/include
    (7)make all-gcc install-gcc
    (8)cd [newlib-build]
    (9)[newlib-source]/configure                        --target=arm-elf
--prefix=[toolchain-prefix] --enable-
interwork--enable-multilib
    (10)make all install
    (11)cd [gcc-build]
    (12)make all install
    (13)cd [gdb-build]
    (14)[gdb-source]/configure --target=arm-elf --prefix=[toolchain-prefix]
```

```
--enable-inter
work --enable-multilib
    (15)make all install
```

当使用交叉编译器的时候，程序通常用前缀来指示目标的体系结构和连接器的输出格式，如 arm-linux-gcc、arm-linux-gdb、powerpc-linux-gcc 等。

建立交叉工具链的过程相当繁琐，我们不必亲自操作，可以直接下载第三方编译好了的、开放的、针对目标处理器的交叉工具链，如在 http://www.codesourcery.com/gnu_toolchains/上可以下载针对 ARM、ColdFire 和 Power 的工具链。

22.1.3 串口工具

在嵌入式 Linux 的调试过程中，目标机往往会提供给主机一个串口控制台，驱动工程师在 80%以上的情况下都是通过串口与目标机通信。因此，好用的串口工具将大大提高工程师的生产效率。

在 Windows 环境下，其附件内自带了超级终端，超级终端包括了对 VT100、ANSI 等终端仿真功能以及对 xmodem、ymodem、zmodem 等协议的支持，如图 22.4 所示。

图 22.4 超级终端

在调试过程中，经常需要保存串口打印信息的历史记录，这时候可以使用“传送”菜单下的“捕获文字”功能来实现。

SecureCRT 是比超级终端更强大且更方便的工具，它将 SSH 的安全登录、数据传送性能和 Windows 终端仿真提供的可靠性、可用性和可配置性结合在一起，其界面如图 22.5 所示。鉴于 SecureCRT 具备比超级终端更强大且好用的功能，建议直接用 SecureCRT 替代超级终端。

图 22.5　SecureCRT

在开发过程中，为执行自动化的串口发送操作，可以使用 SecureCRT 的 VBScript 脚本功能，让其运行一段脚本，自动捕获接收到的串口信息并向串口上发送指定的数据或文件。代码清单 22.1 所示的脚本等待接收到“CCC”字符串后通过 xmodem 协议发送 file.bin 文件，之后，当接收到“y/n”时，选择“y”。

代码清单 22.1　SecureCRT VBScript 脚本范例

```
#$language = "VBScript"
#$interface = "1.0"

Sub main
  Dir = "d:\baohua\"
  ' turn on synchronous mode so we don't miss any data
  crt.Screen.Synchronous = True
  'wait "CCC" string then send file
  crt.Screen.WaitForString "CCC"
  crt.FileTransfer.SendXmodem Dir & "file.bin"
  'wait "y/n" string then send "y"
  crt.Screen.WaitForString "y/n"
  crt.Screen.Send "y" & VbCr
End Sub
```

Minicom 是 Linux 系统下常用的类似于 Windows 下超级终端的工具，当要发送文件或设置串口时，需先按下“CTRL+A”，紧接着按下“Z”键激活菜单，如图 22.6 所示。

```
Welcome to minicom 2.00.0
                          Minicom Command Summary
OPTIO
Compi          Commands can be called by CTRL-A <key>

Press                Main Functions                  Other Functions

      Dialing directory..D  run script (Go)....G ¦ Clear Screen.......C
      Send files.........S  Receive files......R ¦ cOnfigure Minicom..O
      comm Parameters....P  Add linefeed.......A ¦ Suspend minicom....J
      Capture on/off.....L  Hangup.............H ¦ eXit and reset.....X
      send break.........F  initialize Modem...M ¦ Quit with no reset.Q
      Terminal settings..T  run Kermit.........K ¦ Cursor key mode....I
      lineWrap on/off....W  local Echo on/off..E ¦ Help screen........Z
                                                 ¦ scroll Back........B

           Select function or press Enter for none. _

                  Written by Miquel van Smoorenburg 1991-1995
                  Some additions by Jukka Lahtinen 1997-2000
                  i18n by Arnaldo Carvalho de Melo 1998

 CTRL-A Z for help ¦ 38400 8N1 ¦ NOR ¦ Minicom 2.00.0 ¦ VT102 ¦ Online 00:00
```

图 22.6　Minicom

除了 Minicom 以外，在 Linux 系统下，也可以直接使用 C-Kermit。运行 kermit 命令即可启动 C-Kermit。在使用 C-Kermit 连接目标板之前，需先进行串口设置，如下所示：

```
set line /dev/ttyS0
set speed 115200
set carrier-watch off
set handshake none
set flow-control none
robust
set file type bin
set file name lit
set rec pack 1000
set send pack 1000
set window 5
```

之后，使用以下命令就可以连接到目标板：

```
connect
```

在 kermit 的使用过程中，会涉及串口控制台和 kermit 功能模式之间的切换，从串口控制台切换到 kermit 的方法是按下“Ctrl + \”，然后再按下“C”。

假设我们在串口控制台上敲入命令使得目标板进入文件接收等待状态，此后可按下“Ctrl + \ ”，再按“C”，切换到 kermit，运行“send /file_name”命令传输文件。文件传输结束后，再运行“c”命令将进入串口控制台。

22.2 GDB 调试器用法

22.2.1 GDB 基本用法

GDB 是 GNU 开源组织发布的一个强大的 UNIX 下的程序调试工具，GDB 主要可帮助工程师完成下面 4 个方面的功能。

- 启动程序，可以按照工程师自定义的要求运行程序。
- 让被调试的程序在工程师指定的断点处停住，断点可以是条件表达式。
- 当程序被停住时，可以检查此时程序中所发生的事，并追踪上文。
- 动态地改变程序的执行环境。

不管是调试 Linux 内核空间的驱动还是调试用户空间的应用程序，掌握 GDB 的用法都是必须。而且，调试内核和调试应用程序时使用的 GDB 命令是完全相同的，下面以代码清单 22.2 的应用程序为例演示 GDB 调试器的用法。

代码清单 22.2　GDB 调试演示程序

```
int add(int a, int b)
{
  return a + b;
}

main()
{
  int sum[10] =
  {
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0
  }  ;
  int i;

  int array1[10] =
  {
    48, 56, 77, 33, 33, 11, 226, 544, 78, 90
  };
  int array2[10] =
  {
    85, 99, 66, 0x199, 393, 11, 1, 2, 3, 4
  };

  for (i = 0; i < 10; i++)
  {
    sum[i] = add(array1[i], array2[i]);
```

```
    26   }
    27 }
```

使用命令“gcc –g gdb_example.c –o gdb_example”编译上述程序，得到包含调试信息的二进制文件 example，执行“gdb gdb_example”命令进入调试状态，如下所示：

```
    [root@localhost driver_study]# gdb gdb_example
    GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
    Copyright 2003 Free Software Foundation, Inc.
    GDB is free software, covered by the GNU General Public License, and
you are
    welcome to change it and/or distribute copies of it under certain
conditions.
    Type "show copying" to see the conditions.
    There is absolutely no warranty for GDB.  Type "show warranty" for
details.
    This GDB was configured as "i386-redhat-linux-gnu"...
    (gdb)
```

1. list 命令

在 GDB 中运行 list 命令（缩写 l）可以列出代码，list 的具体形式如下。

- list <linenum>，显示程序第 linenum 行周围的源程序，如下所示：

```
    (gdb) list 15
    10
    11      int array1[10] =
    12      {
    13        48, 56, 77, 33, 33, 11, 226, 544, 78, 90
    14      };
    15      int array2[10] =
    16      {
    17        85, 99, 66, 0x199, 393, 11, 1, 2, 3, 4
    18      };
    19
```

- list <function> ，显示函数名为 function 的函数的源程序，如下所示：

```
    (gdb) list main
    2      {
    3        return a + b;
    4      }
    5
    6      main()
    7      {
    8        int sum[10];
    9        int i;
    10
    11       int array1[10] =
```

- list，显示当前行后面的源程序。
- list -，显示当前行前面的源程序。

下面演示了使用 GDB 中的 run（缩写为 r）、break（缩写为 b）、next（缩写为 n）

命令控制程序的运行，并使用 print（缩写为 p）命令打印程序中的变量 sum 的过程：

```
(gdb) break add
Breakpoint 1 at 0x80482f7: file gdb_example.c, line 3.
(gdb) run
Starting program: /driver_study/gdb_example

Breakpoint 1, add (a=48, b=85) at gdb_example.c:3
warning: Source file is more recent than executable.

3        return a + b;
(gdb) next
4      }
(gdb) next
main () at gdb_example.c:23
23       for (i = 0; i < 10; i++)
(gdb) next
25         sum[i] = add(array1[i], array2[i]);
(gdb) print sum
$1 = {133, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

2．run 命令

在 GDB 中，运行程序使用 run 命令。在程序运行前，我们可以设置如下 4 方面的工作环境。

（1）程序运行参数。

“set args”可指定运行时参数，如：“set args 10 20 30 40 50”；“show args”命令可以查看设置好的运行参数。

（2）运行环境。

“path <dir>”可设定程序的运行路径；“how paths”可查看程序的运行路径；“set environment varname [=value]”用于设置环境变量，如“set env USER=baohua”；“show environment [varname]”则用于查看环境变量。

（3）工作目录。

“cd <dir>”相当于 shell 的 cd 命令，pwd 显示当前所在的目录。

（4）程序的输入输出。

“info terminal”用于显示程序用到的终端的模式；GDB 中也可以使用重定向控制程序输出，如“run > outfile”；tty 命令可以指定输入输出的终端设备，如：tty /dev/ttyS1。

3．break 命令

在 GDB 中用 break 命令来设置断点，设置断点的方法如下。

（1）break <function>。

在进入指定函数时停住，C++中可以使用“class::function”或“function(type, type)”格式来指定函数名。

（2）break <linenum>。

在指定行号停住。

（3）break +offset / break – offset。

华清远见

在当前行号的前面或后面的 offset 行停住，offiset 为自然数。

（4）break filename:linenum。

在源文件 filename 的 linenum 行处停住。

（5）break filename:function。

在源文件 filename 的 function 函数的入口处停住。

（6）break *address。

在程序运行的内存地址处停住。

（7）break。

break 命令没有参数时，表示在下一条指令处停住。

（8）break…if <condition>。

“…”可以是上述的“break <linenum>”、“break +offset / break –offset”中的参数，condition 表示条件，在条件成立时停住。比如在循环体中，可以设置“break if i=100”，表示当 i 为 100 时停住程序。

查看断点时，可使用 info 命令，如“info breakpoints [n]”、“info break [n]”（n 表示断点号）。

4．单步命令

在调试过程中，next 命令用于单步执行，类似于 VC++中的 step over。next 的单步不会进入函数的内部，与 next 对应的 step（缩写为 s）命令则在单步执行一个函数时，会进入其内部，类似于 VC++中的 step into。下面演示了 step 命令的执行情况，在第 23 行的 add()函数调用处执行 step 会进入其内部的“return a+b;”语句：

```
(gdb) break 25
Breakpoint 1 at 0x8048362: file gdb_example.c, line 25.
(gdb) run
Starting program: /driver_study/gdb_example

Breakpoint 1, main () at gdb_example.c:25
25        sum[i] = add(array1[i], array2[i]);
(gdb) step
add (a=48, b=85) at gdb_example.c:3
3       return a + b;
```

单步执行的更复杂用法如下。

（1）step <count>。

单步跟踪，如果有函数调用，则进入该函数（进入函数的前提是，此函数被编译有 debug 信息）。step 后面不加 count 表示一条条地执行，加 count 表示执行后面的 count 条指令，然后再停住。

（2）next <count>。

单步跟踪，如果有函数调用，它不会进入该函数。同样地，next 后面不加 count

表示一条条地执行，加 count 表示执行后面的 count 条指令，然后再停住。

（3）set step-mode。

“set step-mode on”用于打开 step-mode 模式，这样，在进行单步跟踪时，程序不会因为没有 debug 信息而不停住，这个参数的设置可便于查看机器码。“set step-mod off”用于关闭 step-mode 模式。

（4）finish。

运行程序，直到当前函数完成返回，并打印函数返回时的堆栈地址和返回值及参数值等信息。

（5）until（缩写为 u）。

一直在循环体内执行单步，退不出来是一件令人烦恼的事情，until 命令可以运行程序直到退出循环体。

（6）stepi（缩写为 si）和 nexti（缩写为 ni）。

stepi 和 nexti 用于单步跟踪一条机器指令，一条程序代码有可能由数条机器指令完成，stepi 和 nexti 可以单步执行机器指令。

另外，运行“display/i $pc”命令后，单步跟踪会在打出程序代码的同时打出机器指令，即汇编代码。

5．continue 命令

当程序被停住后，可以使用 continue 命令（缩写为 c，fg 命令同 continue 命令）恢复程序的运行直到程序结束，或到达下一个断点，命令格式为：

```
continue [ignore-count]
c [ignore-count]
fg [ignore-count]
```

ignore-count 表示忽略其后多少次断点。

假设我们设置了函数断点 add()，并 watch i，则在 continue 过程中，每次遇到 add() 函数或 i 发生变化，程序就会停住，如下所示：

```
(gdb) continue
Continuing.
Hardware watchpoint 3: i

Old value = 2
New value = 3
0x0804838d in main () at gdb_example.c:23
23        for (i = 0; i < 10; i++)
(gdb) continue
Continuing.

Breakpoint 1, main () at gdb_example.c:25
25          sum[i] = add(array1[i], array2[i]);
(gdb) continue
```

```
Continuing.
Hardware watchpoint 3: i

Old value = 3
New value = 4
0x0804838d in main () at gdb_example.c:23
23        for (i = 0; i < 10; i++)
```

6. print 命令

在调试程序时，当程序被停住时，可以使用 print 命令（缩写为 p），或是同义命令 inspect 来查看当前程序的运行数据。print 命令的格式如下：

```
print <expr>
print /<f> <expr>
```

<expr>是表达式，是被调试的程序中的表达式，<f>是输出的格式，比如，如果要把表达式按十六进制的格式输出，那么就是/x。在表达式中，有几种 GDB 所支持的操作符，它们可以用在任何一种语言中，“@”是一个和数组有关的操作符，“::”指定一个在文件或是函数中的变量，“{<type>} <addr>”表示一个指向内存地址<addr>的类型为 type 的一个对象。

下面演示了查看 sum[]数组的值的过程：

```
(gdb) print sum
$2 = {133, 155, 0, 0, 0, 0, 0, 0, 0, 0}
(gdb) next

Breakpoint 1, main () at gdb_example.c:25
25        sum[i] = add(array1[i], array2[i]);
(gdb) next
23        for (i = 0; i < 10; i++)
(gdb) print sum
$3 = {133, 155, 143, 0, 0, 0, 0, 0, 0, 0}
```

当需要查看一段连续内存空间的值的时间，可以使用 GDB 的“@”操作符，“@”的左边是第一个内存地址，“@”的右边则是想查看内存的长度。例如如下动态申请的内存：

```
int *array = (int *) malloc (len * sizeof (int));
```

在 GDB 调试过程中这样显示出这个动态数组的值：

```
p *array@len
```

print 的输出格式包括：

- x：按十六进制格式显示变量。
- d：按十进制格式显示变量。
- u：按十六进制格式显示无符号整型。
- o：按八进制格式显示变量。

- t：按二进制格式显示变量。
- a：按十六进制格式显示变量。
- c：按字符格式显示变量。
- f：按浮点数格式显示变量。

我们可用 display 命令设置一些自动显示的变量，当程序停住时，或是单步跟踪时，这些变量会自动显示。

如果要修改变量，如 x 的值，可使用如下命令：

```
print x=4
```

当用 GDB 的 print 查看程序运行时的数据时，每一个 print 都会被 GDB 记录下来。GDB 会以$1，$2，$3 …这样的方式为每一个 print 命令编号。我们可以使用这个编号访问以前的表达式，如$1。

7．watch 命令

watch 一般来观察某个表达式（变量也是一种表达式）的值是否有变化了，如果有变化，马上停止程序运行。我们有如下几种方法来设置观察点。

watch <expr>：为表达式（变量）expr 设置一个观察点。一量表达式值有变化时，马上停止程序运行。

rwatch <expr>：当表达式（变量）expr 被读时，停止程序运行。

awatch <expr>：当表达式（变量）的值被读或被写时，停止程序运行。

info watchpoints：列出当前所设置了的所有观察点。

下面演示了观察 i 并在连续运行 next 时一旦发现 i 变化，i 值就会显示出来的过程：

```
(gdb) watch i
Hardware watchpoint 3: i
(gdb) next
23        for (i = 0; i < 10; i++)
(gdb) next
Hardware watchpoint 3: i

Old value = 0
New value = 1
0x0804838d in main () at gdb_example.c:23
23        for (i = 0; i < 10; i++)
(gdb) next

Breakpoint 1, main () at gdb_example.c:25
25          sum[i] = add(array1[i], array2[i]);
(gdb) next
23        for (i = 0; i < 10; i++)
(gdb) next
Hardware watchpoint 3: i

Old value = 1
New value = 2
0x0804838d in main () at gdb_example.c:23
23        for (i = 0; i < 10; i++)
```

8．examine 命令

我们可以使用 examine 命令（缩写为 x）来查看内存地址中的值。examine 命令的语法如下所示：

```
x/<n/f/u> <addr>
```

<addr>表示一个内存地址。“x/”后的 n、f、u 都是可选的参数，n 是一个正整数，表示显示内存的长度，也就是说从当前地址向后显示几个地址的内容；f 表示显示的格式，如果地址所指的是字符串，那么格式可以是 s，如果地址是指令地址，那么格式可以是 i；u 表示从当前地址往后请求的字节数，如果不指定的话，GDB 默认是 4 字节。u 参数可以被一些字符代替：b 表示单字节，h 表示双字节，w 表示四字节，g 表示八字节。当我们指定了字节长度后，GDB 会从指定的内存地址开始，读写指定字节，并把其当作一个值取出来。n、f、u 这 3 个参数可以一起使用，例如命令“x/3uh 0x54320”表示从内存地址 0x54320 开始以双字节为 1 个单位（h）、16 进制方式（u）显示 3 个单位（3）的内存。

9．jump 命令

一般来说，被调试程序会按照程序代码的运行顺序依次执行，但是 GDB 也提供了乱序执行的功能，也就是说，GDB 可以修改程序的执行顺序，从而让程序随意跳跃。这个功能可以由 GDB 的 jump 命令“jump <linespec>”来指定下一条语句的运行点。<linespec>可以是文件的行号，可以是“file:line”格式，也可以是“+num”这种偏移量格式，表示下一条运行语句从哪里开始。

```
jump <address>
```

这里的<address>是代码行的内存地址。

注意，jump 命令不会改变当前的程序栈中的内容，所以，如果使用 jump 从一个函数跳转到另一个函数，当跳转到的函数运行完返回，进行出栈操作时必然会发生错误，这可能导致意想不到的结果，所以最好只用 jump 在同一个函数中进行跳转。

10．signal 命令

使用 singal 命令，可以产生一个信号量给被调试的程序，如中断信号“Ctrl+C”。这非常方便于程序的调试，可以在程序运行的任意位置设置断点，并在该断点用 GDB 产生一个信号量，这种精确地在某处产生信号的方法非常有利于程序的调试。

signal 命令的语法是“signal <signal>”，UNIX 的系统信号量通常为 1～15，所以<signal>取值也在这个范围。

11．return 命令

如果在函数中设置了调试断点，在断点后还有语句没有执行完，这时候我们可以使用 return 命令强制函数忽略还没有执行的语句并返回。

```
return
```

```
return <expression>
```

上述 return 命令用于取消当前函数的执行，并立即返回，如果指定了<expression>，那么该表达式的值会被作为函数的返回值。

12．call 命令

call 命令用于强制调用某函数：

```
call <expr>
```

表达式可以是函数，以此达到强制调用函数的目的，它会显示函数的返回值（如果函数返回值不是 void）。

其实，前面介绍的 print 命令也可以完成强制调用函数的功能。

13．info 命令

info 命令可以在调试时用来查看寄存器、断点、观察点和信号等信息。要查看寄存器的值，可以使用如下命令：

```
info registers （查看除了浮点寄存器以外的寄存器）
info all-registers （查看所有寄存器，包括浮点寄存器）
info registers <regname ...>  （查看所指定的寄存器）
```

要查看断点信息，可以使用如下命令：

```
info break
```

列出当前所设置的所有观察点，使用如下命令：

```
info watchpoints
```

查看有哪些信号正在被 gdb 检测，使用如下命令：

```
info signals
info handle
```

也可以使用 info line 命令来查看源代码在内存中的地址。info line 后面可以跟行号、函数名、文件名：行号、文件名:函数名等多种形式，例如下面的命令会打印出所指定的源码在运行时的内存地址：

```
info line tst.c:func
```

14．disassemble

disassemble 命令用于反汇编，它可被用来查看当前执行时的源代码的机器码，实际上只是把目前内存中的指令 dump 出来。下面的示例用于查看函数 func 的汇编代码：

```
(gdb) disassemble func
Dump of assembler code for function func:
0x8048450 <func>:      push   %ebp
0x8048451 <func+1>:    mov    %esp,%ebp
0x8048453 <func+3>:    sub    $0x18,%esp
0x8048456 <func+6>:    movl   $0x0,0xfffffffc(%ebp)
...
End of assembler dump.
```

22.2.2　DDD 图形界面调试工具

GDB 本身是一种命令行调试工具，但是通过 DDD（Data Display Debugger，

http://www.gnu.org/software/ddd/），可以被图形界面化。DDD 可以作为 GDB、DBX、WDB、Ladebug、JDB、XDB、Perl Debugger 或 Python Debugger 的可视化图形前端，其特有的图形数据显示功能（Graphical Data Display）可以把数据结构按照图形的方式显示出来。

DDD 最初源于 1990 年 Andreas Zeller 编写的 VSL 结构化语言，后来经过一些程序员的努力，演化成今天的模样。DDD 的功能非常强大，可以调试用 C\C++、Ada、Fortran、Pascal、Modula-2 和 Modula-3 编写的程序；可以超文本方式浏览源代码；能够进行断点设置、回溯调试和历史记录编辑；具有程序在终端运行的仿真窗口，具备在远程主机上进行调试的能力；能够显示各种数据结构之间的关系，并将数据结构以图形化形式显示；具有 GDB/DBX/XDB 的命令行界面，包括完全的文本编辑、历史纪录、搜寻引擎等。

DDD 的主界面如图 22.7 所示，和 Visual Studio 等集成开发环境非常相近，而且 DDD 包含了 Visual Studio 所不包含的部分功能。

图 22.7　DDD 的主界面

在设计 DDD 的时候，设计人员决定把它与 GDB 之间的耦合度尽可能降低。因为像 GDB 这样的开源软件，更新比商业软件快。所以为了使 GDB 的变化不会影响到 DDD，在 DDD 中，GDB 是作为独立的进程运行的，通过命令行接口与 DDD 进行交互。

图 22.8 显示了用户、DDD、GDB 和被调试进程之间的关系，DDD 和 GDB 之间

的所有通信都是异步进行的。在 DDD 中发出的 GDB 命令都会与一个回调例程相连，放入命令队列中。这个回调例程在合适的时间会处理 GDB 的输出。例如，如果用户手动输入一条 GDB 的命令，DDD 就会把这条命令与显示 GDB 输出的一个回调例程连起来。一旦 GDB 命令完成，就会触发回调例程，GDB 的输出就会显示在 DDD 的命令窗口中。

图 22.8　DDD 运行机理

DDD 在事件循环时等待用户输入和 GDB 输出，同时等着 GDB 进入等待输入状态。当 GDB 可用时，下一条命令就会从命令队列中取出，送给 GDB。GDB 到达的输出由上次命令的回调过程来处理。这种异步机制避免了 DDD 在等待 GDB 输出时发生阻塞现象，到达的事件可以在任何时间得到处理。

不可否认的是，DDD 和 GDB 的分离使得 DDD 运行速度相对来说比较慢，但是这种方法带来了灵活性和兼容性的好处。例如，用户可以把 GDB 调试器换成其他调试器，如 DBX 等。另外，GDB 和 DDD 的分离使得用户可以在不同的机器上分别运行 GDB 和 DDD。

在 DDD 中，可以直接在底部的控制台中输入 GDB 命令，也可以通过菜单和鼠标来以图形方式触发 GDB 命令的运行，使用方法甚为简单，因此这里不再赘述。除了基本的 GDB 命令外，DDD 中的 Plot 工具可以用于将数组以二维或三维坐标系中点、曲线或曲面的方式显示出来，如图 22.9 所示，这在某些场合下会非常有用。dsp 工程师应该不会陌生，因为在 dsp 程序调试中，在集成开发环境中绘制数组曲线是十分常见的用法。

图 22.9　DDD 中的 Plot 绘制数组

22.3 Linux 内核调试

在嵌入式系统中，由于目标机资源有限，因此往往在主机上编译好程序，再在目标机上运行。用户所有的开发工作在主机开发环境下完成，包括编码、编译、连接、下载和调试等。目标机和主机通过串口、以太网、仿真器或其他通信手段通信，主机用这些接口控制目标机，调试目标机上的程序。

调试嵌入式 Linux 内核的方法如下。

- 目标机“插桩”，如打上 KGDB 补丁，这样主机上的 GDB 可与目标机的 KGDB 通过串口或网口通信。
- 使用仿真器，仿真器可直接连接目标机的 JTAG/BDM，这样主机的 GDB 就可以通过与仿真器的通信来控制目标机。
- 在目标板上通过 printk()、oops、strace 等软件方法进行“观察”调试，这些方法不具备查看和修改数据结构、断点、单步等功能。

第 22.3～22.7 节将对这些调试方法进行一一讲解。

不管是目标机“插桩”还是使用仿真器连接目标机 JTAG/BDM，在主机上，调试

工具一般都采用 GDB。尽管采用“插桩”和仿真器的方式可以进行查看和修改数据结构、断点、单步等，而 printk()这种最原始的方法却更广泛地被应用。

22.4 内核打印信息——printk()

在 Linux 中，内核打印语句 printk()会将内核信息输出到内核信息缓冲区中。内核信息缓冲区是一个环形缓冲区（ring buffer），因此，如果塞入的消息过多，就会将之前的消息冲刷掉。

Linux 的 klogd 进程（一个系统守护进程，它截获并且记录下 Linux 内核日志信息）会通过/proc/kmsg 文件读取缓冲区，一旦读取完成，内核信息便从缓冲区中被删除。之后，klogd 守护进程会将读取的内核信息派发给 syslogd 守护进程（syslogd 记录下系统里所有提供日志记录的程序给出的日志和信息内容，每一个被记录的消息至少包含时间戳和主机名），syslogd 这个守护进程会根据/etc/syslog.conf 将不同的服务产生的日志记录到不同的文件中。例如在/etc/syslog.conf 中增加“kern.* /tmp/kernel_debug.txt”一行，则内核信息也会被放置到/tmp/kernel_debug.txt 文件中。执行“insmod hello.ko”，我们看到/tmp/kernel_debug.txt 文件中多出如下一行：

```
Jul  6 00:40:51 localhost kernel:  Hello World enter
```

用户也可以直接使用“cat /proc/kmsg”命令来显示内核信息，但是，由于/proc/kmsg 是一个“永无休止的文件”，因此，“cat /proc/kmsg”的进程只能通过“Ctrl+C”或 kill 终止。另外，使用 dmesg 命令也可以直接读取 ring buffer 中的信息。

printk()定义了 8 个消息级别，分为级别 0～7，越低级别（数值越大）的消息越不重要，第 0 级是紧急事件级，第 7 级是调试级，代码清单 22.3 所示为 printk()的级别定义。

代码清单 22.3　printk()的级别定义

```
1  #define KERN_EMERG "<0>" /*紧急事件，一般是系统崩溃之前提示的消息 */
2  #define KERN_ALERT "<1>" /* 必须立即采取行动 */
3  #define KERN_CRIT "<2>"/*临界状态，通常涉及严重的硬件或软件操作失败 */
4  #define KERN_ERR  "<3>"   /*用于报告错误状态，设备驱动程序会
5                              经常使用 KERN_ERR 来报告来自硬件的问题 */
6  #define KERN_WARNING  "<4>"   /*对可能出现问题的情况进行警告，
7                                 这类情况通常不会对系统造成严重问题 */
8  #define KERN_NOTICE   "<5>"   /*有必要进行提示的正常情形，
9                                 许多与安全相关的状况用这个级别进行汇报*/
10 #define KERN_INFO     "<6>"     /*内核提示性信息，很多驱动程序
11                               在启动的时候，以这个级别打印出它们找到的硬件
信息*/
12 #define KERN_DEBUG    "<7>"   /* 用于调试信息 */
```

通过/proc/sys/kernel/printk 文件可以调节 printk 的输出等级，该文件有 4 个数字值，如下所示。

- 控制台日志级别：优先级高于该值的消息将被打印至控制台。
- 默认的消息日志级别：将用该优先级来打印没有优先级的消息。
- 最低的控制台日志级别：控制台日志级别可被设置的最小值（最高优先级）。
- 默认的控制台日志级别：控制台日志级别的默认值。

上述 4 个值的默认设置为 6、4、1、7。

通过如下命令可以使得 Linux 内核的任何 printk 都被输出：

```
# echo 8 > /proc/sys/kernel/printk
```

在设备驱动中，我们经常需要输出调试或系统信息，尽管可以直接采用 printk("<7>debug info…\n")方式的 printk()语句输出，但是通常可以使用封装了 printk()的更高级的宏，如 pr_debug()、dev_debug()等。代码清单 22.4 所示 pr_debug()和 pr_info()的定义，代码清单 22.5 所示为 dev_dbg()、dev_err()、dev_info()等的定义，前一组的输出中不包含设备信息，后一组包含。

代码清单 22.4　替代 printk()的宏

```
1  #ifdef DEBUG
2  #define pr_debug(fmt,arg...) \
3    printk(KERN_DEBUG fmt,##arg)
4  #else
5  static inline int __attribute__ ((format (printf, 1, 2))) pr_debug(const char * fmt, ...)
6  {
7    return 0;
8  }
9  #endif
10
11 #define pr_info(fmt,arg...) \
12    printk(KERN_INFO fmt,##arg)
```

代码清单 22.5　包含设备信息的替代 printk()的宏

```
1  #define dev_printk(level, dev, format, arg...)  \
2      printk(level "%s %s: " format , dev_driver_string(dev) , (dev)->bus_id , ## arg)
3
4  #ifdef DEBUG
5  #define dev_dbg(dev, format, arg...)          \
6        dev_printk(KERN_DEBUG , dev , format , ## arg)
7  #else
8  #define dev_dbg(dev, format, arg...) do { (void)(dev); } while (0)
9  #endif
```

```

#define dev_err(dev, format, arg...)          \
        dev_printk(KERN_ERR , dev , format , ## arg)
#define dev_info(dev, format, arg...)         \
        dev_printk(KERN_INFO , dev , format , ## arg)
#define dev_warn(dev, format, arg...)         \
        dev_printk(KERN_WARNING , dev , format , ## arg)
#define dev_notice(dev, format, arg...)       \
        dev_printk(KERN_NOTICE , dev , format , ## arg)
```

在打印信息时，如果想输出其所在的函数，可以使用__FUNCTION__，如：

```
printk("%s: Incorrect IRQ %d from %s\n", __FUNCTION__, irq, devname);
```

C99 标准已经提供了__func__来指定函数名，因此目前__FUNCTION__实际定义为：

```
#define __FUNCTION__ (__func__)
```

22.5 使用/proc

在 Linux 系统中，/proc 文件系统十分有用，它被用于内核向用户导出信息。/proc 文件系统是一个虚拟文件系统，通过它可以使用一种新的方法在 Linux 内核空间和用户空间之间进行通信。在/proc 文件系统中，我们可以将对虚拟文件的读写作为与内核中实体进行通信的一种手段，与普通文件不同的是，这些虚拟文件的内容都是动态创建的。/proc 下的文件并非完全是只读的，若节点可写，还可用于一定的控制或配置目的，例如前面介绍的写/proc/sys/kernel/printk 可以改变 printk 的打印级别。

Linux 系统的许多命令本身都是通过分析/proc 下的文件来完成，如 ps、top、uptime 和 free 等。例如，free 命令通过分析/proc/meminfo 文件得到可用内存信息，下面显示了对应的 meminfo 文件和 free 命令的结果。

- meminfo 文件：

```
[root@localhost proc]# cat meminfo
MemTotal:       29516 kB
MemFree:         1472 kB
Buffers:         4096 kB
Cached:         12648 kB
SwapCached:         0 kB
Active:         14208 kB
Inactive:        8844 kB
HighTotal:          0 kB
HighFree:           0 kB
LowTotal:       29516 kB
```

```
LowFree:          1472 kB
SwapTotal:      265064 kB
SwapFree:       265064 kB
Dirty:             20 kB
Writeback:          0 kB
Mapped:         10052 kB
Slab:            3864 kB
CommitLimit:    279820 kB
Committed_AS:    13760 kB
PageTables:        444 kB
VmallocTotal:   999416 kB
VmallocUsed:       560 kB
VmallocChunk:   998580 kB
```

- free 命令：

```
[root@localhost proc]# free
          total       used       free     shared    buffers     cached
Mem:      29516      28104       1412          0       4100      12700
-/+ buffers/cache:  11304      18212
Swap:     265064          0     265064
```

Linux 的 USB、PCI 等内核代码本身都会创建/proc 节点导出内核信息，虽然不值得鼓励，但在 Linux 设备驱动程序中，驱动工程师自定义/proc 节点以向外界传递信息的方法仍然是可行的。

在 Linux 系统中，可用如下函数创建/proc 节点：

```
struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode,
                struct proc_dir_entry *parent);

struct proc_dir_entry *create_proc_read_entry(const char *name, mode_t mode,
                    struct proc_dir_entry *base, read_proc_t *read_proc,
void * data);
```

create_proc_entry()函数用于创建/proc 节点，而 create_proc_read_entry()调用 create_proc_entry()创建只读的/proc 节点。参数 name 为/proc 节点的名称，parent/base 为父目录的节点，如果为 NULL，则指/proc 目录，read_proc 是/proc 节点的读函数指针。当 read()系统调用在/proc 文件系统中执行时，它映像到一个数据产生函数，而不是一个数据获取函数。

下列函数用于创建/proc 目录：

```
struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent);
```

结合 create_proc_entry()和 proc_mkdir()，代码清单 22.6 中的程序可用于先在/proc 下创建一个目录，而后在该目录下创建一个文件。

代码清单 22.6　proc_mkdir()和 create_proc_entry()函数使用范例

```
/* 创建/proc 下的目录 */
example_dir = proc_mkdir("procfs_example", NULL);
if (example_dir == NULL)
//创建失败
{
  rv =  - ENOMEM;
  goto out;
}

example_dir->owner = THIS_MODULE;

/* 创建一个例子/proc 文件 */
  example_file   =   create_proc_entry("example_file",   0666, example_dir);
if (example_file == NULL)
//创建失败
{
  rv =  - ENOMEM;
  goto out;
}

example_file->owner = THIS_MODULE;
example_file->read_proc = example_file_read;
example_file->write_proc = example_file_write;

out: ...
```

作为上述函数各返回值的 proc_dir_entry 结构体中包含了/proc 节点的读函数指针（read_proc_t *read_proc）、写函数指针（write_proc_t *write_proc）以及父节点、子节点信息等。

/proc 节点的读写函数的类型分别为：

```
typedef int (read_proc_t)(char *page, char **start, off_t off,
                                  int count, int *eof, void *data);
typedef int (write_proc_t)(struct file *file, const char __user *buffer,
                                  unsigned long count, void *data);
```

读函数中 page 指针指向用于写入数据的缓冲区，start 用于返回实际的数据写到内存页的位置，eof 是用于返回读结束标志，offset 是读的偏移，count 是要读的数据长度。

start 参数比较复杂，对于/proc 只包含简单数据的情况，通常不需要在读函数中设置*start，意味着内核将认为数据保存在内存页偏移 0 的地方。如果将*start 设置为非 0 值，意味着内核将认为*start 指向的数据是 offset 偏移处的数据。

写函数与 file_operations 中的 write()成员类似，需要一次从用户缓冲区到内存空间的复制过程。

Linux 系统中可用如下函数删除/proc 节点：

```
void remove_proc_entry(const char *name, struct proc_dir_entry *parent);
```

Linux 系统中已经定义好的可使用的/proc 节点宏包括：proc_root_fs（/proc）、proc_net （/proc/net）、proc_bus（/proc/bus）、proc_root_driver（/proc/driver）等，proc_root_fs 实际就是 NULL。

代码清单 22.7 所示为一个简单的/proc 使用范例，这段代码在模块加载函数中创建/proc 文件节点，在模块卸载函数中撤销/proc 节点，而文件中只保存了一个 32 位的无符号整形值。

代码清单 22.7 /proc 文件系统使用模板

```
1  #include ...
2
3  static struct proc_dir_entry *proc_entry;
4  static unsigned long val = 0x12345678;
5
6  /* 读/proc 文件接口 */
7  ssize_t simple_proc_read(char *page, char **start, off_t off, int count,
8    int*eof, void *data)
9  {
10   int len;
11   if (off > 0) //不能偏移访问
12   {
13     *eof = 1;
14     return 0;
15   }
16
17   len = sprintf(page, "%08x\n", val);
18
19   return len;
20 }
21
22 /* 写/proc 文件接口 */
23 ssize_t simple_proc_write(struct file *filp, const char __user *buff, unsigned
24   long len, void *data)
25 {
26   #define MAX_UL_LEN 8
27   char k_buf[MAX_UL_LEN];
28   char *endp;
29   unsigned long new;
30   int count = min(MAX_UL_LEN, len);
31   int ret;
32
33   if (copy_from_user(k_buf, buff, count))
34   //用户空间->内核空间
35   {
```

```
36    ret =  - EFAULT;
37    goto err;
38  }
39  else
40  {
41    new = simple_strtoul(k_buf, &endp, 16); //字符串转化为整数
42    if (endp == k_buf)
43    //无效的输入参数
44    {
45      ret =  - EINVAL;
46      goto err;
47    }
48    val = new;
49    return count;
50  }
51  err:
52  return ret;
53 }
54
55 int __init simple_proc_init(void)
56 {
57   proc_entry = create_proc_entry("sim_proc", 0666, NULL); //创建
/proc
58  if (proc_entry == NULL)
59  {
60    printk(KERN_INFO "Couldn't create proc entry\n");
61    goto err;
62  }
63  else
64  {
65    proc_entry->read_proc = simple_proc_read;
66    proc_entry->write_proc = simple_proc_write;
67    proc_entry->owner = THIS_MODULE;
68  }
69  return 0;
70  err:
71  return  - ENOMEM;
72 }
73
74 void __exit simple_proc_exit(void)
75 {
76  remove_proc_entry("sim_proc", &proc_root); //撤销/proc
77 }
78
79 module_init(simple_proc_init);
80 module_exit(simple_proc_exit);
81
82 MODULE_AUTHOR("Song Baohua, author@linuxdriver.cn");
83 MODULE_DESCRIPTION("A simple Module for showing proc");
84 MODULE_VERSION("V1.0");
```

上述代码第 41 行用于转换用户输入的字符串无符号长整数，第 3 个参数 16 意味

着转化方式是十六进制。

编译上述简单的“sim_proc.c”为“sim_proc.ko”，运行“insmod sim_proc.ko”加载该模块后，/proc 目录下将多出一个文件 sim_proc，“ls –l”的结果如下：

```
[root@localhost proc]# ls -1 sim_proc
-rw-rw-rw-   1 root    root          0 Sep  4 20:31 sim_proc
```

权限与创建/proc/sim_proc 时给出的 0666 参数是一致的，现在读取 sim_proc，如下所示：

```
[root@localhost proc]# cat sim_proc
12345678
```

读出来的正好是我们赋的初值 0x12345678。

测试写/proc/sim_proc 文件，使用 echo 命令修改它为 0x88888888：

```
[root@localhost driver_study]# echo 88888888 > /proc/sim_proc
```

再查看新值：

```
[root@localhost driver_study]# cat /proc/sim_proc
88888888
```

说明我们上一步执行的写操作是正确的。

22.6 Oops

当内核出现 Segmentation Fault 时（例如内核访问一个并不存在的虚拟地址），Oops 会被打印到控制台和写入系统 ring buffer。

我们编写一个字符设备驱动，使让它产生 Oops，在其读写函数中都访问 0 地址，如代码清单 22.8 所示。

代码清单 22.8　产生 Oops 设备驱动的读写函数

```
1  static ssize_t oopsexam_read(struct file *filp, char *buf, size_t len, loff_t *off)
2  {
3    int *p=0;
4    *p  = 1;  //故意访问 0 地址
5    return len;
6  }
7
8  static ssize_t oopsexam_write(struct file *filp, const char *buf, size_t len, loff_t
9    *off)
10 {
11   int *p=0;
```

```
12   *p  = 1;  //故意访问 0 地址
13   return len;
14 }
```

假设这个字符设备对应的设备节点是/dev/oops_example，通过“echo 1 > /dev/oops_example”命令写设备文件，将得到如下 Oops 信息：

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
 printing eip:
c381a013
*pde = 00000000
Oops: 0002 [#1]
PREEMPT SMP
Modules linked in: oops_example
CPU:   0
EIP:   0060:[<c381a013>]   Not tainted VLI
EFLAGS: 00010286  (2.6.15.5)
EIP is at oopsexam_write+0x4/0x11 [oops_example]
eax: 00000002  ebx: c2b35480  ecx: 00000000  edx: c381a00f
esi: 00000002  edi: 080e9408  ebp: c2007fa4  esp: c2007f68
ds: 007b  es: 007b  ss: 0068
Process bash (pid: 2453, threadinfo=c2006000 task=c2021570)
Stack: c015e036 c2b35480 080e9408 00000002 c2007fa4 00000000 c2b35480 fffffff7
       080e9408 c2006000 c015e1d1 c2b35480 080e9408 00000002 c2007fa4 00000000
       00000000 00000000 00000001 00000002 c0102f9f 00000001 080e9408 00000002
Call Trace:
 [<c015e036>] vfs_write+0xc5/0x18f
 [<c015e1d1>] sys_write+0x51/0x80
 [<c0102f9f>] sysenter_past_esp+0x54/0x75
Code:  Bad EIP value.
```

上述 Oops 的第一行给出了“原因”，即访问了“NULL pointer”。Oops 中的“EIP is at oopsexam_write+0x4/0x11 [oops_example]”这一行也比较关键，给出了“事发现场”，即 oopsexam_
write()函数偏移 4 字节的指令处。

通过反汇编可以知道偏移 4 字节的指令对应的 C 代码，如下所示：

```
1  00000000 <oopsexam_read>:
2    0:   8b 44 24 0c          mov    0xc(%esp,1),%eax
3    4:   c7 05 00 00 00 00 01    movl   $0x1,0x0
```

```
    4     b:   00 00 00
    5     e:   c3                  ret
```

第 3 行的“movl　$0x1,0x0”对应“*p　= 1;”。这里仅仅给出了一个例子，实际的“事发现场”并不这么容易被找到，但方法都是类似的。

同样地，我们通过“cat /dev/oops_example”命令去读设备文件，将得到如下 Oops 信息：

```
    Unable to handle kernel NULL pointer dereference at virtual address
00000000
     printing eip:
    c381a004
    *pde = 00000000
    Oops: 0002 [#2]
    PREEMPT SMP
    Modules linked in: oops_example
    CPU:    0
    EIP:    0060:[<c381a004>]    Not tainted VLI
    EFLAGS: 00010286   (2.6.15.5)
    EIP is at oopsexam_read+0x4/0xf [oops_example]
    eax: 00001000   ebx: c0d80180   ecx: 00000000   edx: c381a000
    esi: 00001000   edi: 0804d8d0   ebp: c1df5fa4   esp: c1df5f68
    ds: 007b   es: 007b   ss: 0068
    Process cat (pid: 2969, threadinfo=c1df4000 task=c2021570)
    Stack: c015dd9e c0d80180 0804d8d0 00001000 c1df5fa4 00000000 c0d80180
fffffff7
          0804d8d0 c1df4000 c015e151 c0d80180 0804d8d0 00001000 c1df5fa4
00000000
          00000000 00000000 00000003 00001000 c0102f9f 00000003 0804d8d0
00001000
    Call Trace:
     [<c015dd9e>] vfs_read+0xc5/0x18f
     [<c015e151>] sys_read+0x51/0x80
     [<c0102f9f>] sysenter_past_esp+0x54/0x75
    Code:  Bad EIP value.
```

现在给出的“原因”与写时完全相同，但是“事发地”变成了 oopsexam_read()函数偏移 4 字节的指令处。

在驱动中如果发现硬件或软件的运行情况与预期的不一致，完全可以通过下面的语句故意抛出一个 Oops，以便于提供 bug 的上下文信息：

```
    (*(int *)0 = 0);
```

内核中有许多地方调用的“BUG();”语句中的 BUG()宏通常就被定义为该语句。

22.7 监视工具

在 Linux 系统中，strace 是一种相当有效的跟踪工具，它的主要特点是可以被用来监视系统调用。我们不仅可以用 strace 调试一个新开始的程序，也可以调试一个已经在运行的程序（这意味着把 strace 绑定到一个已有的 PID 上）。对于第 6 章的 globalmem 字符设备文件，以 strace 方式运行如代码清单 22.9 所示的用户空间应用程序 globalmem_test，运行的结果如下：

```
    execve("./globalmem_test", ["./globalmem_test"], [/* 24 vars */]) = 0
    ...
    open("/dev/globalmem",  O_RDWR)              =  3            //打开的
/dev/globalmem的fd是3
    ioctl(3, FIBMAP, 0)                  = 0
    read(3, 0xbff17920, 200)           = -1 ENXIO (No such device or address)
//读取失败
    fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
    mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0xb7f04000
    write(1, "-1 bytes read from globalmem\n", 29-1 bytes read from globalmem
    ) = 29                                  //向标准输出设备(fd为1)写入printf
中的字符串
    write(3, "This is a test of globalmem", 27) = 27
    write(1, "27 bytes written into globalmem\n", 3227 bytes written into
globalmem
    ) = 32
    ...
```

输出的每一行对应一次 Linux 系统调用，其格式为“左边=右边”，等号左边是系统调用的函数名及其参数，右边是该调用的返回值。

代码清单 22.9　用户空间应用程序 globalmem_test

```
1  #include ...
2
3  #define MEM_CLEAR 0x1
4  main()
5  {
6    int fd, num, pos;
7    char wr_ch[200] = "This is a test of globalmem";
```

```
8   char rd_ch[200];
9   //打开/dev/globalmem
10  fd = open("/dev/globalmem", O_RDWR, S_IRUSR | S_IWUSR);
11  if (fd != -1 )
12  {
13      //清除 globalmem
14      if(ioctl(fd, MEM_CLEAR, 0) < 0)
15      {
16      printf("ioctl command failed\n");
17      }
18      //读 globalmem
19      num = read(fd, rd_ch, 200);
20      printf("%d bytes read from globalmem\n",num);
21
22   //写 globalmem
23    num = write(fd, wr_ch, strlen(wr_ch));
24    printf("%d bytes written into globalmem\n",num);
25    ...
26    close(fd);
27  }
28 }
```

使用 strace 虽然无法直接追踪到设备驱动中的函数，但是足够可以帮助工程师推演，如从“open("/dev/globalmem", O_RDWR) = 3”的返回结果知道/dev/globalmem 的 fd 为 3，之后对 fd 为 3 的文件进行的 read()、write()和 ioctl()系统调用最终都会引起 globalmem 中 file_operations 中的相应函数被调用，通过系统调用的结果就可以知道驱动中 globalmem_read()、globalmem_write()和 globalmem_ioctl()的运行结果。

22.8 内核调试器

22.8.1 kcore

GDB 调试器可以把内核作为一个应用程序来调试，在这种方式中，需要给 GDB 指定未压缩的内核映像的文件名和“core 文件”的名字。对于一个正在运行的内核，“core 文件”就是运行时的内存映像/proc/kcore（kcore 代表整个内核地址空间，对应于所有的物理内存）。因此，使用 GDB 和 kcore 调试内核的典型命令如下：

```
gdb /usr/src/linux/vmlinux /proc/kcore
```

第一个参数是非压缩的 ELF 核心可执行文件的名字，不能是 zImage、bzImage。第二个参数是内核档案的名称，如同其他/proc 中的文件，/proc/kcore 是在被读的时候才产生的。

在“gdb <path>/vmlinux /proc/kcore”这种调试方式中，可用 print 命令打印变量，

如“print jiffies”。当从 GDB 打印数据时间，GDB 会缓存已经读取的数据，但是由于内核正在执行，各种数据项在不同时间有不同的值，GDB 的缓存可能导致连续多次读取同一变量得到相同的值。例如，多次显示 jiffies：

```
(gdb) print jiffies
$3 = 153729
(gdb) print jiffies
$4 = 153729
(gdb) print jiffies
$5 = 153729
```

虽然 jiffies 已经变更了，但每次 print 出来的都是第 1 次的值 153729。为避免此问题，我们可以在需要刷新 GDB 缓存时发出“core-file /proc/kcore”命令，这将导致调试器使用新的“core 文件”并且丢弃旧信息。当执行“core-file /proc/kcore”命令后，再运行“print jiffies”，值会发生变化，如下所示：

```
(gdb) core-file /proc/kcore
Core was generated by 'ro root=/dev/sda1 hdc=ide-scsi'.
#0  0x00000000 in globalmem_fops ()
(gdb)  print jiffies
$6 = 178683
```

$6 比$3、$4、$5 要大，这说明新的值被 print 出来了。

为了使 Linux 系统中包含/proc/kcore 文件，必须在编译时包含“/proc/kcore support”（如图 22.10 所示），而为了给 GDB 提供 symbol 信息，必须设定 CONFIG_DEBUG_INFO 选项来编译内核（如图 22.11 所示）。

在“gdb <path>/vmlinux /proc/kcore”这种调试方式中，GDB 的绝大多数功能都不能使用，如修改内核变量的值、设置断点、单步执行等，而 22.9.2 和 22.9.3 小节将要介绍的 KDB 和 KGDB 方式则可支持这些功能。

值得一提的是，可加载模块的 symbol 并未包含在 vmlinux 中，必须使用一些辅助方法才能调试模块。Linux 可加载模块是 ELF 格式的可执行映像，它们被分成几个段，有 3 个典型的与模块调试相关的段。

```
Linux Kernel v2.6.15.5 Configuration
qqqq
```

```
..  .data  .rodata                    .strtab        .text
```

通过 cat 其中的.text、.data、.bss 可以得到我们感兴趣的 3 个段的地址，如下所示：

```
[root@localhost sections]# cat .text
0xc3816000
[root@localhost sections]# cat .bss
0xc3816b88
[root@localhost sections]# cat .data
0xc3816a94
```

之后就可以借用 GDB 的 add-symbol-file 来添加模块的符号信息，这样之后便可以查看模块中的变量了，如下所示：

```
(gdb)  add-symbol-file globalmem.ko 0xc3816000\
       -s .bss 0xc3816b88\
       -s .data 0xc3816a94
add symbol table from file "globalmem.ko" at
       .text_addr = 0xc3816000
       .bss_addr = 0xc3816b88
       .data_addr = 0xc3816a94
(y or n) y
Reading symbols from globalmem.ko...done.
(gdb) p globalmem_major   （查看 globalmem.ko 中的变量）
$7 = 254
```

22.8.2 KDB

KDB 项目由 Silicon Graphics 维护，为使内核内嵌 KDB，需要从 Silicon Graphics 的 FTP 站点下载与内核版本有关的补丁，一个是公共补丁（包含了对通用内核代码的更改），另一个是特定于体系结构的补丁，下载地址为 ftp://oss.sgi.com/www/projects/kdb/download。给内核打上 KDB 补丁之后，内核“Kernel hacking”编译选项中就会包含与 KDB 相关的内容，如图 22.12 所示，选中“Built-in Kernel Debugger support”，并根据实际需求选择默认的 KDB 状态为 OFF 或 ON。

```
Linux Kernel v2.6.15.5 Configuration
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
 lqqqqqqqqqqqqqqqqqqqqqqqqqqqq Kernel hacking qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqk
 x  Arrow keys navigate the menu.  <Enter> selects submenus --->.          x
 x  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, x
 x  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> x
 x  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >       x
 x lqqqqqqq^(-)qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqk x
 x x      < >   torture tests for RCU                                     x x
 x x      [ ] KGDB: kernel debugging with remote gdb                      x x
 x x      [*] Check for stack overflows                                   x x
 x x      [ ] Stack utilization instrumentation                           x x
 x x      [ ] Page alloc debugging                                        x x
 x x      [ ] Use 4Kb for kernel stacks instead of 8Kb                    x x
 x x      [*] Built-in Kernel Debugger support                            x x
 x x      < >   KDB modules (NEW)                                         x x
 x x      [ ]   KDB off by default (NEW)                                  x x
 x x      (0)   KDB continues after catastrophic errors (NEW)             x x
 x mqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj x
 tqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqu
 x                    <Select>    < Exit >    < Help >                    x
 mqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj
```

图 22.12　KDB 编译选项

如果编译期间没有选中 CONFIG_KDB_OFF（即选择了“KDB off by default”），那么在默认情况下 KDB 是活动的。否则，需要显式地激活它，通过在引导期间将 KDB=on 标志传递给内核或者通过执行如下命令可激活 KDB：

```
#echo "1" >/proc/sys/kernel/kdb
```

反过来，如果默认情况下 KDB 是打开的，那么将 KDB = off 标志传递给内核或者执行如下命令将会取消激活 KDB：

```
#echo "0" >/proc/sys/kernel/kdb
```

在引导期间还可以将另一个标志传递给内核，即 KDB = early。early 标志将导致在引导过程的初始阶段就把控制权传递给 KDB，这将非常有利于工程师在引导过程初始阶段进行内核调试。

在两种情况下 KDB 会被调用：当 KDB 处于打开状态时，只要内核中有紧急情况就会自动调用它；其次，按下键盘上的“Pause”键也可手工调用 KDB。使用 KDB 可进行内存和寄存器修改、设置断点和跟踪堆栈等。

KDB 中进行内存显示和修改的常用命令是 md 和 mm。md 命令以一个地址/符号和行计数 line-count 为参数，显示从该地址开始的 line-count 行的内存。如果没有指定 line-count，那么就使用环境变量所指定的默认值。如果没有指定地址，那么 md 就从上一次打印的地址继续。mm 命令用于修改内存内容，它以地址/符号和新内容作为参数，用新的内容替换指定地址处的内容。

例如，如下命令可显示从 0xc000000 开始的 15 行内存：

```
kdb> md 0xc000000 15
```

如下命令可将内存位置为 0xc000000 上的内容更改为 0x10：

```
kdb> mm 0xc000000 0x10
```

rd 命令用于显示处理器寄存器的内容，rm 命令用于修改寄存器的内容。它以寄存器名称和新的内容作为参数，用新的内容修改寄存器。寄存器名称与特定的体系结构

有关。目前，不能修改控制寄存器。

KDB 中常用的断点命令有 bp、bc、bd、be 和 bl。bp 命令以一个地址/符号作为参数，当遇到该断点时则系统停止执行并将控制权交予 KDB；bl 命令列出当前的断点集，它包含了启用和禁用的断点；be 命令用于启用断点，该命令的参数是断点号；bc 命令用于从断点表中去除断点，它以断点号或“*”作为参数，为“*”意味着去除所有断点。

例如，执行如下命令将对函数 sys_write()设置断点：

```
kdb> bp sys_write
```

执行如下命令可列出断点表中的所有断点：

```
kdb> bl
```

执行如下命令可清除断点号为 1 的断点：

```
kdb> bc 1
```

KDB 中主要的堆栈跟踪命令有 bt、btp、btc 和 bta。bt 命令提供有关当前线程的堆栈的信息；btp 命令以进程标识作为参数，并对这个特定进程进行堆栈回溯；btc 命令对每个活动 CPU 上正在运行的进程执行堆栈回溯，它从第一个活动 CPU 开始执行 bt，然后切换到下一个活动 CPU，依次类推；bta 命令对处于某种特定状态的所有进程执行回溯，可以有选择性地将各种参数传递给该命令，回溯选项包括 D（不可中断状态）、R（正运行）、S（可中断休眠）、T（已跟踪或已停止）、Z（僵死）和 U（不可运行）。bta 命令若不带任何参数，会对所有进程执行回溯。

除此之外，在内核调试过程中其他的常用 KDB 命令如下。

- id 命令：以地址/符号作为参数，它对从该地址开始的指令进行反汇编，环境变量 IDCOUNT 确定要显示输出的行数。
- ss 命令：单步执行指令然后将控制返回给 KDB。ssb 是该指令的一个变体，它执行从当前指令指针地址开始的指令（在屏幕上打印指令），直到它遇到将引起分支转移的指令为止。
- go 命令：让系统继续正常执行，直到遇到断点为止。
- reboot 命令：立刻重新引导系统。
- ll 命令：以地址、偏移量和另一个 KDB 命令作为参数，它对链表中的每个元素反复执行作为参数的这个命令。

例如，如下命令将反汇编从 schedule()开始的指令（所显示的行数取决于环境变量 IDCOUNT）：

```
kdb> id schedule
```

22.8.3 KGDB

gdb <path>/vmlinux /proc/kcore 方式在调试模块时缺少一些至关重要的功能，KDB 尽管克服了部分缺陷，但是它只能在汇编代码级进行调试，而本小节要介绍的 KGDB 则能很方便地在源码级对内核进行调试。KGDB 采用的正是嵌入式系统中远程调试的思路，主机和目标机之间通过串口或网口进行通信。

MontaVista Linux 直接提供了对 KGDB 的支持，而开源社区的内核中必须打上相应版本的 KGDB 补丁（即 kgdb stub，这种方式俗称“插桩”），如 X86 PC 上需要打的补丁包括：core-lite.patch、i386-lite.patch、8250.patch、eth.patch、i386.patch 和 core.patch。

KGDB 内核补丁的下载地址为：http://kgdb.linsyssoft.com/downloads.htm。

打上 KGDB 补丁后，运行 make menuconfig 时需选择关于 KGDB 的编译项目，包括选择“KGDB: kernel debugging with remote gdb”、“KGDB: Console messages through gdb”并设置串口通信方式，如图 22.13 所示。

```
Linux Kernel v2.6.15.5 Configuration
qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
 lqqqqqqqqqqqqqqqqqqqqqqqqqqqq Kernel hacking qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqk
 x  Arrow keys navigate the menu.  <Enter> selects submenus --->.          x
 x  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, x
 x  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> x
 x  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >       x
 x l^(-)qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqk x
 x x---   Compile the kernel with frame pointers                          x x
 x x< >   torture tests for RCU                                           x x
 x x[*] KGDB: kernel debugging with remote gdb                            x x
 x x[*]   KGDB: Console messages through gdb                              x x
 x x      Method for KGDB communication (KGDB: On generic serial port (82x x
 x x< >   KGDB: On ethernet (NEW)                                         x x
 x x[*] Simple selection of KGDB serial port (NEW)                        x x
 x x(115200) Debug serial port baud rate (NEW)                            x x
 x x(1) Serial port number for KGDB (NEW)                                 x x
 x x[ ] Check for stack overflows                                         x x
 x mv(+)qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj x
 tqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqu
 x                    <Select>    < Exit >    < Help >                    x
 mqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj
```

图 22.13　KGDB 编译选项配置

在 VmWare 中同时启动两个虚拟机就可以模拟 KGDB 的使用环境，在主机上运行如下命令设置串口波特率：

```
stty ispeed 115200 ospeed 115200 -F /dev/ttyS0
```

修改目标机的/etc/grub.conf，增加如下项目：

```
title Linux-2.6.15.5-kgdb
root (hd0,0)
kernel /boot/vmlinuz-2.6.15.5-kgdb ro root=/dev/sda1 hdc=ide-scsi kgdbwait
```

kgdbwait 的含义是启动时就等待主机的 GDB 连接。

依次运行如下命令就可以启动调试并连接至目标机：

```
<root#> gdb ./vmlinux
GNU gdb Red Hat Linux (6.0post-0.20040223.17rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".
```

```
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS0                          //连接目标机
Remote debugging using /dev/ttyS0
breakpoint () at kernel/kgdb.c:1212
1212 atomic_set(&kgdb_setting_breakpoint, 0);
warning: shared library handler failed to enable breakpoint
(gdb)
```

之后，在主机上，我们可以使用 GDB 就像调试应用程序一样调试加载了 KGDB 的目标机上的内核。

为进行可加载模块的调试，需要使用 gdbmod 并借助一些技巧来在模块加载的时候获取 symbol 信息。首先需要设置 solib-search-path 变量的路径，如运行“set solib-search-path /driver_study”命令后，再加载 globalmem.ko，接着运行“info sharedlibrary”命令，如果看到相应的模块信息，就可以在主机上调试加载后的 globalmem.ko 模块中的 C 代码了。

最后，需要注意到 KGDB 的工作是以目标系统的串口或网口正常工作为前提的，作为一种软件“插桩”的调试方式，在调试过程中如果出现死机问题，主机上将无法定位。

22.9 使用仿真器调试内核

本节以 BDI2000 为例来讲解如何使用仿真器调试 Linux 内核。BDI2000 是一种最常见的功能强大的仿真器，使用它可以直接调试 Linux 内核。在典型的调试环境中，BDI2000、主机、目标机这 3 者的关系如图 22.14 所示。

除了支持免费的 GDB 以外，BDI2000 还支持商业级的 MontaVista DevRocket、LinuxScope（Eclipse GDB）调试器。

为使用 BDI2000 调试目标板内核，在 BDI2000 和目标板启动加电后，BDI2000 端需完成如下工作。

（1）配置 BDI2000，修改.cfg 文件，确保目标机的正常初始化。

（2）主机通过 telnet 登录 BDI2000，例如，若主机的/etc/hosts 中包含了“bdi”节点，则运行“telnet bdi”即可登录到 BDI2000。

图 22.14　BDI2000、主机与目标机

（3）设置内核运行时的第一个断点，通常在函数 start_kernel()，如“[BDI2000] bi XXXXXXXX”。

通过 grep start_kernel System.map 可以获得 start_kernel()的具体位置，代替上面的 XXXXXXXX。

（4）执行内核代码 “[BDI2000] go”。

以上第（3）、（4）步的运行是以目标板已加载 Linux 内核到 RAM 为前提的，如果没有加载，则需借助仿真器和主机端加载。

相应地，主机端需完成如下工作。

（1）通过 GDB 启动内核调试，如运行：

```
arm-linux-gdb vmlinux
```

（2）连接仿真器，使用 GDB 的“target remote”命令：

```
target remote bdi:2001
```

之后，就可以在 GDB 中像调试应用程序一样调试目标板上运行的 Linux 内核了。

华清远见

22.10 应用程序调试

在嵌入式系统中，为调试 Linux 应用程序，可在目标板上先运行 GDBServer，再让主机上的 GDB 与目标板上的 GDBServer 通过网口或串口通信。

1．目标板

需要运行如下命令启动 GDBSerber：

```
gdbserver <host_ip>:<port> <app>
```

<host_ip>:<port>为主机的 IP 地址和端口，app 是可执行的应用程序名。

当然，也可以用系统中空闲的串口作为 GDB 调试器和 GDBServer 的底层通信手段，如：

```
gdbserver/dev/ttyS0./tdemo
```

2. 主机

需要先运行如下命令启动 GDB：

```
arm-linux-gdb <app>
```

app 与 GDBServer 的 app 参数对应，arm-linux-gdb 是专门为 ARM 处理器编译出的 GDB 调试器。

之后，运行如下命令就可以连接目标板：

```
target remote <target_ip>:<port>
```

<target_ip>:<port>为目标机的 IP 地址和端口。

如果目标板上的 GDBServer 使用串口，则在宿主机上 GDB 也应该使用串口，如：

```
(gdb)target remote/dev/ttyS1
```

之后，便可以使用 GDB 像调试本机上的程序一样调试目标机上的程序。

22.11 总结

Linux 程序的调试尤其是内核的调试看起来比较复杂，没有类似于 VC++、Tornado 的 IDE 开发环境，最常用的调试手段依然是文本方式的 GDB。文本方式的 GDB 调试器功能异常强大，当我们使用习惯后，就会用得非常自然了。

Linux 内核驱动的调试方法包括“插桩”、使用仿真器和借助 printk()、oops、strace 等，在大多数情况下，原始的 printk()仍然是最有效的手段。

除了本章介绍的方法外，在驱动的调试中很可能还会借助其他的硬件或软件调试工具，如调试 USB 驱动最好借助 USB 分析仪，USB 分析仪将可捕获 USB 通信中的包，如同网络中的 sniffer 软件一样。

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm
- 实验设备：http://www.embedu.org/courses/course5.htm

嵌入式学院 华清远见旗下品牌

第 23 章　Linux 设备驱动的移植

在编写 Linux 设备驱动的时候，驱动程序所服务的硬件芯片可能会在公司的多个采用不同处理器的系统中用到，因此，在编写驱动时，应该尽量考虑其可移植性，23.1 节从数据类型、结构体对界、大小端模式、内存页面大小等多个角度阐述了编写可移植驱动程序的注意事项。

“他山之石，可以攻玉”，为了高效地推出新的设备驱动，借用 demo 板、类似芯片和厂商范例程序是几种很有效的手段，23.2 节讲解了这些快速编写设备驱动的方法。

23.3 节讲解了 Linux 2.4 和 Linux 2.6 内核在设备驱动方面的差异，通过对两者差异的分析，可以得出移植 Linux 2.4 内核驱动到 Linux2.6 内核的方法。

23.4 节给出了将其他操作系统内的驱动移植到 Linux 中的方法，主要分析了实时操作系统 VxWorks 设备驱动和 Linux 设备驱动的异同点。

23.1 编写可移植的设备驱动

23.1.1 可移植的数据类型

C 语言中的标准数据类型 int、long 的长度直接与平台相关，在驱动中，关键部分代码直接使用这些类型时需要特别小心。表 23.1 给出了在几个不同的平台下 char、short、int、long、ptr、long long 的长度。

表 23.1 不同平台下 char、short、int、long、ptr、long long 的长度

arch	char	short	int	long	ptr	long long
i386	1	2	4	4	4	8
Alpha	1	2	4	8	8	8
armv4l	1	2	4	4	4	8
ia64	1	2	4	8	8	8
M68K	1	2	4	4	4	8
MIPS	1	2	4	4	4	8
PaverPC	1	2	4	4	4	8
Sparc	1	2	4	4	4	8
Sparc64	1	2	4	4	4	8
x86_64	1	2	4	8	8	8

因此，在 Linux 系统中，针对不同的体系结构重新 typedef 出了 u8、u16、u32、u64、s8、s16、s32、s64 等类型。例如，在 i386/arm 下这些类型的定义如代码清单 23.1 所示，而在 ppc64 下这些类型的定义则如代码清单 23.2 所示，可见影响 C 语言基本数据类型大小的主要因素是 CPU 字长。

代码清单 23.1 i386/arm 平台下 u8、u16、u32、u64、s8、s16、s32、s64 的定义

```
typedef signed char s8;
typedef unsigned char u8;

typedef signed short s16;
typedef unsigned short u16;

typedef signed int s32;
typedef unsigned int u32;

typedef signed long long s64;
typedef unsigned long long u64;
```

华清远见

代码清单 23.2　ppc64 平台下 u8、u16、u32、u64、s8、s16、s32、s64 的定义

```
typedef signed char s8;
typedef unsigned char u8;

typedef signed short s16;
typedef unsigned short u16;

typedef signed int s32;
typedef unsigned int u32;

typedef signed long s64;
typedef unsigned long u64;
```

由于 u8、u16、u32、u64、s8、s16、s32、s64 是被针对不同的体系结构单独定义的，因此，在任何平台下对其进行 sizeof 运算的结果都是不变的，是确定长度的数据类型。但是，这些类型都应该只在内核空间使用。在 Linux 用户空间中，如果要使用确定长度的数据类型，应该使用上述定义加“__”的版本，如__u8、__u16、__u32 等。鉴于此，设备驱动中如果要向用户空间导出头文件，在头文件中也应该定义__sxx、__uxx 等数据类型。

上面给出的 uxx、sxx、__uxx、__sxx 类型定义都是 Linux 系统所特有的，为了更好地向其他平台移植，驱动中最好使用 int8_t、int16_t、int32_t、uint8_t、uint16_t、uint32_t、int64_t、uint64_t 这些 C99 标准确定长度类型。

Linux 系统中定义了许多以_t 为后缀的数据类型，这些类型用在内核的一些常用功能的实现中，如 dma_addr_t、uid_t、gid_t、size_t、ssize_t、pid_t、loff_t 等，这些类型的使用将使内核屏蔽实际数据类型间存在的差异。例如，file_operations 中的 read()、write()成员函数返回值为 ssize_t 类型，llseek()返回的是 loff_t 类型。此外，ssize_t、pid_t 这些类型也被赋予了一些含义，这一点从名字就可以看出来，如 pid_t 是进程 ID 类型。

对于硬件的寄存器定义或者驱动中会轮询访问的内存地址、变量等，一般会将对应类型前面加上 volatile 关键字以告知编译器不要对其进行优化，这一点非常关键，例如如下的一段代码：

```
void my_delay(uint32_t t)
{
  uint32_t i;
  i = t;
  while (i--);
}
```

很有可能达不到延迟的目的，如果这样修改：

```
void my_delay(uint32_t t)
{
   volatile uint32_t i;
```

```
    i = t;
    while (i--);
}
```

则 while()循环的代码不会被编译器优化掉。

23.1.2 结构体对界

在 C 语言中使用结构体时有一个需要特别注意的事项，那就是结构体的对界。struct 是一种复合数据类型，其构成元素既可以是基本数据类型的变量，也可以是一些复合数据类型（如数据、结构体、联合体等）的数据单元。对于结构体，编译器很可能会自动进行成员变量的对齐，以提高存取效率。默认情况下，编译器为结构体的每个成员按其自然对界（natural alignment）条件分配空间。各个成员按照它们被声明的顺序在内存中顺序存储，第一个成员的地址和整个结构的地址相同。

自然对界指按结构体的成员中 sizeof 最大的成员对齐（如果 sizeof 大于 CPU 的字长，仍然按照 CPU 字长对齐），例如对于 32 位系统：

```
struct naturalalign
{
    char a;
    short b;
    char c;
};
```

在上述结构体中，size 最大的是 short，其长度为两个字节，因而结构体中的 char 成员 a、c 都以 2 为单位对齐，sizeof(naturalalign)的结果等于 6。

如果改为：

```
struct naturalalign
{
     char a;
     int b;
     char c;
};
```

其结果为 12。

在 Linux 内核编程中，为了防止编译器自动在结构体的数据间插入空隙，可以使用__attribute__
((packed))定义结构体，如：

```
struct
{
  u16 id;
  u64 lun;
  u16 reserved1;
```

```
    u32 reserved2;
} __attribute__((packed))scsi;  //不要在数据间插入空隙
```

23.1.3 Little Endian 与 Big Endian

采用 Little Endian 模式的 CPU 对操作数的存放方式是从低字节到高字节，而 Big Endian 模式对操作数的存放方式是从高字节到低字节。例如，16bit 宽的数 0x1234 在 Little Endian 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）为：

内存地址	0x4000	0x4001
存放内容	0x34	0x12

而在 Big Endian 模式，CPU 内存中的存放方式则为：

内存地址	0x4000	0x4001
存放内容	0x12	0x34

32bit 宽的数 0x12345678 在 Little Endian 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）为：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x78	0x56	0x34	0x12

而在 Big Endian 模式 CPU 内存中的存放方式则为：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x12	0x34	0x56	0x78

内核中定义如下多个宏来进行 Big Endian 模式与 Little Endian 模式的互换，包括 cpu_to_le64、le64_to_cpu、cpu_to_le32、le32_to_cpu、cpu_to_le16、le16_to_cpu。

内核中定义如下多个宏来进行 Big Endian 模式与 Big Endian 模式的互换：

`cpu_to_be64`、`be64_to_cpu`、`cpu_to_be32`、`be32_to_cpu`、`cpu_to_be16`、`be16_to_cpu`。

在 Linux/drivers 目录的 ATM、IEEE1394、SCSI、NET、USB 等源码中，都大量存在对上述这些宏的使用。

23.1.4 内存页面大小

一般情况下，内存页面的大小是 4KB（即 PAGE_SIZE 定义为 4KB），但是这并非是一定的，实际上，页面大小在一个 4～64KB 的范围内是可变的，即使在相同的平台下也可以定义不同的 PAGE_SIZE 和 PAGE_SHIFT。

鉴于此，当在内核空间中通过 get_free_pages()函数申请内存时，如果它的第二个参数为 order，意味着申请 PAGE_SIZE * 2^{order} 的内存，同样是申请 64KB 内存，如果 PAGE_SIZE 为 4KB，应该传入的 order 是 4，如果 PAGE_SIZE 是 16KB，应该传入的参数是 2。为了保证在申请 64KB 内存时，在任何 PAGE_SIZE 的情况下都成立，可以使用如代码清单 23.3 所示的方法。

代码清单 23.3　通过 get_order()获得要申请内存的 order

```
1 #include <asm/page.h>
```

中搜索 request_region()、request_irq()、request_dma()等出现 request_xxx()函数的语句，根据其使用的参数反推到定义资源的宏或变量。

23.2.2 巧用类似芯片的驱动程序

任何驱动工程师都没有必要在面对新设备驱动编写需求的时候一切从头开始，因为内核源代码 drivers 目录（音频设备的驱动在 sound 目录）中已经包含了大量现成的类似芯片驱动的源代码，是极好的参考模板，我们不需要“re-invent the wheel”。实际上，在内核源代码许多后期编写的驱动程序中，就直接参考了之前的驱动源码，所以同类设备的驱动往往呈现出非常相似的架构和数据结构定义。我们来看看 sound/oss/au1550_ac97.c 文件最开始的一段注释：

```
/*
 * au1550_ac97.c  --  Sound driver for Alchemy Au1550 MIPS Internet Edge
 *                    Processor.
 *
 * Copyright 2004 Embedded Edge, LLC
 *  dan@embeddededge.com
 *
 * Mostly copied from the au1000.c driver and some from the
 * PowerMac dbdma driver.
 * We assume the processor can do memory coherent DMA.
...
*/
```

这段注释很清楚地说明其绝大多数代码都来自 au1000.c 驱动，还有一些来自 PowerMac dbdma 驱动。

打开 sound/oss 目录下的 au1550_ac97.c（Alchemy Au1550 MIPS 处理器的音频驱动）、es1370.c（Ensoniq ES1370/Asahi Kasei AK4531 声卡驱动）、es1371.c（reative Ensoniq ES1371 声卡驱动）、cs46xx.c（Crystal SoundFusion CS46xx 声卡驱动），发现如下相似之处。

- 它们全都自定义了全局的 xxx_state 结构体实例用于封装音频设备的锁、信号量、缓冲区、ID 等信息，这几个结构体分别是：au1550_state、es1370_state、es1371_state、cs_state。
- 它们的核心函数都使用了完全相同的实现方法，au1550_ac97.c 的 au1550_read()和 es1370.c 的 es1370_read()的处理流程是一致的，下面列表的左右两列对等地给出了 au1550_read()和 es1370_read()函数的源代码（为了进行横向比较，适当地增加了源代码的换行，以达到类似 WinMerge 等源码比较软件的效果）。

static ssize_t u1550_read(struct file *file, char *buffer, size_t count, loff_t*ppos) { struct au1550_state *s =	Static ssize_t es1370_read(struct file *file, char __user *buffer, size_t count, loff_t *ppos) { struct es1370_state *s =

```
(struct

au1550_state*)file->private_data;
      struct    dmabuf    *db    =
&s->dma_adc;
      DECLARE_WAITQUEUE(wait,
current);
      ssize_t ret;
      unsigned long flags;
      int cnt, usercnt, avail;



      if (db->mapped)
        return  - ENXIO;
      if  (!access_ok(VERIFY_WRITE,
buffer,
        count))
        return  - EFAULT;
      ret = 0;

      count *= db->cnt_factor;

      down(&s->sem);





      add_wait_queue(&db->wait,
&wait);

      while (count > 0)
      {
        /* wait for samples in ADC
dma buffer
         */
        do
        {
```

```
(struct

es1370_state*)file->private_data;

    DECLARE_WAITQUEUE(wait,
current);
      ssize_t ret = 0;
      unsigned long flags;
      unsigned swptr;
      int cnt;

      VALIDATE_STATE(s);
      if (s->dma_adc.mapped)
        return  - ENXIO;
      if   (!access_ok(VERIFY_WRITE,
buffer,
        count))
        return  - EFAULT;




      down(&s->sem);
      if (!s->dma_adc.ready && (ret =
        prog_dmabuf_adc(s)))
        goto out;

      add_wait_queue(&s->dma_adc.wait,
&wait);

      while (count > 0)
      {




        spin_lock_irqsave(&s->lock,
flags);
```

```
spin_lock_irqsave(&s->lock, flags) ;
        if (db->stopped)
          start_adc(s);
        avail = db->count;

        if (avail <= 0)
          __set_current_state
            (TASK_INTERRUPTIBLE);

spin_unlock_irqrestore(&s->lock,
          flags);



        if (avail <= 0)
        {



          if      (file->f_flags
&O_NONBLOCK)
          {
            if (!ret)
              ret =  - EAGAIN;
            goto out;
          }
          up(&s->sem);
          schedule();
          if
(signal_pending(current))
          {
            if (!ret)
              ret     =          -
ERESTARTSYS;
            goto out2;
          }
          down(&s->sem);
        }
      }
      while (avail <= 0);
```

```
      swptr = s->dma_adc.swptr;
      cnt  =  s->dma_adc.dmasize  -
swptr;
      if (s->dma_adc.count < cnt)
        cnt = s->dma_adc.count;
      if (cnt <= 0)
        __set_current_state
          (TASK_INTERRUPTIBLE);

spin_unlock_irqrestore(&s->lock,
        flags);
      if (cnt > count)
        cnt = count;
      if (cnt <= 0)
      {
        if (s->dma_adc.enabled)
          start_adc(s);
        if          (file->f_flags
&O_NONBLOCK)
        {
          if (!ret)
            ret =  - EAGAIN;
          goto out;
        }
        up(&s->sem);
        schedule();
        if
(signal_pending(current))
        {
          if (!ret)
            ret =  - ERESTARTSYS;
          goto out;
        }
        down(&s->sem);
        if (s->dma_adc.mapped)
        {
          ret =  - ENXIO;
          goto out;
        }
        continue;
      }
      if (copy_to_user(buffer, s
        ->dma_adc.rawbuf  +  swptr,
cnt))
      {
        if (!ret)
          ret =  - EFAULT;
        goto out;
      }
```

```
        /* copy from nextOut to user
         */

        if          ((cnt          =
copy_dmabuf_user(db,
          buffer, count > avail ?
avail :
          count, 1)) < 0)
        {
          if (!ret)
            ret =  - EFAULT;
          goto out;
        }

        spin_lock_irqsave(&s->lock,
flags);
        db->count -= cnt;
        db->nextOut += cnt;
        if     (db->nextOut     >=
db->rawbuf + db
          ->dmasize)
          db->nextOut            -=
db->dmasize;

spin_unlock_irqrestore(&s->lock,
          flags);

        count -= cnt;
        usercnt     =     cnt     /
db->cnt_factor;
        buffer += usercnt;
        ret += usercnt;
      } /* while (count > 0) */


      out: up(&s->sem);
      out2:
remove_wait_queue(&db->wait,
        &wait);

set_current_state(TASK_RUNNING);
      return ret;
    }
```

```
        swptr = (swptr + cnt) % s
          ->dma_adc.dmasize;

        spin_lock_irqsave(&s->lock,
flags);
        s->dma_adc.swptr = swptr;
        s->dma_adc.count -= cnt;




spin_unlock_irqrestore(&s->lock,
            flags);

        count -= cnt;
        buffer += cnt;
        ret += cnt;
        if (s->dma_adc.enabled)
          start_adc(s);
      }

      out: up(&s->sem);

remove_wait_queue(&s->dma_adc.wait,
        &wait);

set_current_state(TASK_RUNNING);
      return ret;
    }
```

可以看出，内核中看似神秘的、庞大的设备驱动源码也是互相学习、互相借鉴的

结果。

华清远见

23.2.3 借用芯片厂商的范例程序

在外围芯片上市之前，芯片厂商往往进行了严格的验证，在他们的验证过程中，必然会编写代码去访问和控制这些芯片。很多时候，这些代码稍经整理就被芯片厂商随同 datasheet 一起在网站上作为参考代码发布。

范例程序往往停留在无操作系统的层次上，只是最底层的硬件操作代码，这一部分代码对驱动工程师的意义如下。

- 帮助工程师进一步理解芯片与 CPU 的接口原理、芯片的访问和控制方法。
- 直接加以改进后搬到 Linux 设备驱动中。

Linux 设备驱动的硬件操作方法会与无操作系统时的硬件操作方法有如下差异。

- 无操作系统的硬件访问方法中往往没有物理地址到虚拟地址的映射过程，因此，在搬到 Linux 系统中的时候，要注意以静态映射或 ioremap()等方式映射到虚拟地址。
- 硬件访问中往往夹杂着延时，因此，在无操作系统的源码中，经常会出现 xxx_delay()这样的 for 循环延迟，这些代码应该被内核中的 ndelay()或 udelay()替换。如果延迟时间达到数十 ms，应该使用 msleep()或 msleep_interruptible()等函数。
- 芯片范例程序只是对芯片的操作方法进行示范，它并不会考虑真实应用场景中对 CPU 的资源占用以及代码的时间性能。例如，如果在写寄存器 REGA 后，要判断寄存器 REGB 的第 0 位为 1 后才能进行下一次写，则无操作系统中的代码呈现为：

```
write_rega(int value)
{
  rega = value;
  while (!(regb &0x1));
}
```

第 2 句的 while (!(regb &0x1))是比较致命的，如果系统中用的 Linux 不支持抢占调度，而 REGB 的第 0 位变成 1 需要相当长的时间（如数十 ms），这种忙等待会导致其他的进程全部得不到机会执行。即使 Linux 支持抢占调度，进行这样的忙等待也毫无意义，Linux 中理想的做法是进行在这种情况下调度其他进程执行。

使用 while (!(regb &0x1))这样的判断还有一个更严重的陷阱，如果硬件出现了故障，REGB 的第 0 位总是变不成 1 的话，在系统不支持抢占调度的情况下，就“死机”了。

23.3 从 Linux 2.4 移植设备驱动到 Linux 2.6

从 Linux 2.4 内核到 Linux 2.6 内核，Linux 在可装载模块机制、设备模型、一些核

心 API 等方面发生了较大改变，随着公司产品的过渡，驱动工程师会面临着将驱动从 Linux 2.4 内核移植到 Linux 2.6 内核，或是让驱动能同时支持 Linux 2.4 内核与 Linux 2.6 内核的任务。

下面分析 Linux 2.4 内核和 Linux 2.6 内核在设备驱动方面的几个主要的不同点。

1．内核模块的 Makefile

Linux 2.4 内核中，模块的编译只需内核源码头文件，并在包括 linux/modules.h 头文件之前定义 MODULE，且其编译、连接后生成的内核模块后缀为.o。而在 Linux 2.6 内核中，模块的编译需要依赖配置过的内核源码，编译过程首先会到内核源码目录下，读取顶层的 Makefile 文件，然后再返回模块源码所在目录，且编译、连接后生成的内核模块后缀为.ko。

Linux 2.4 中内核模块的 Makefile 模板如代码清单 23.4 所示。

代码清单 23.4　Linux 2.4 中内核模块的 Makefile 模板

```
1  #Makefile2.4
2  KVER=$(shell uname -r)
3  KDIR=/lib/modules/$(KVER)/build
4  OBJS=mymodule.o
5      CFLAGS=-D_  _KERNEL_  _   -I$(KDIR)/include   -DMODULE   -D_
_KERNEL_SYSCALLS_ _
6   -DEXPORT_SYMTAB -O2 -fomit-frame-pointer  -Wall  -DMODVERSIONS
7   -include $(KDIR)/include/linux/modversions.h
8  all: $(OBJS)
9   mymodule.o: file1.o file2.o
10  ld -r -o $@ $^
11 clean:
12  rm -f *.o
```

而 Linux 2.6 中内核模块的 Makefile 模板如代码清单 23.5 所示。

代码清单 23.5　Linux 2.6 中内核模块的 Makefile 模板

```
1  # Mcakefile2.6
2  ifneq ($(KERNELRELEASE),)
3  #dependency relationshsip of files and target modules
4  #mymodule-objs := file1.o file2.o
5  #obj-m := mymodule.o
6  obj-m := second.o
7  else
8  PWD  := $(shell pwd)
9  KVER ?= $(shell uname -r)
10 KDIR := /lib/modules/$(KVER)/build
11 all:
12  $(MAKE) -C $(KDIR) M=$(PWD)
13 clean:
14  rm -rf .*.cmd *.o *.mod.c *.ko .tmp_versions
15 endif
```

Linux 2.6 内核模板 Makefile 中的 KERNELRELEASE 是在内核源码的顶层 Makefile 中定义的一个变量，在第一次读取执行此 Makefile 时，KERNELRELEASE 没有被定义，所以 make 将读取执行 else 之后的内容。如果 make 的目标是 clean，将直接执行 clean 操作，然后结束；当 make 的目标为 all 时，-C $(KDIR)指明跳转到内核源码目录下读取那里的 Makefile，M=$(PWD)表明之后要返回到当前目录继续读入、执行当前的 Makefile。当从内核源码目录返回时，KERNELRELEASE 已被定义，kbuild 也被启动去解析 kbuild 语法的语句，make 将继续读取 else 之前的内容。else 之前的内容为 kbuild 语法的语句，指明模块源码中各文件的依赖关系，以及要生成的目标模块名。"mymodule-objs := file1.o file2.o"表示 mymoudule.o 由 file1.o 与 file2.o 连接生成，"obj-m := mymodule.o"表示编译连接后将生成 mymodule 模块。

“$(MAKE) -C $(KDIR) M=$(PWD)”与“$(MAKE) -C $(KDIR) SUBDIRS =$(PWD)”的作用是等效的，后者是较老的使用方法。

通过以上比较可以看到，从 Makefile 编写角度来看，在 Linux 2.6 内核下，内核模块编译不必定义复杂的 CFLAGS，而且模块中各文件依赖关系的表示更加简洁清晰。

在分析清楚 Linux 2.4 和 Linux 2.6 的内核模块 Makefile 的差异之后，可以给出同时支持 Linux 2.4 内核和 Linux 2.6 内核的内核模块 Makefile 文件，如代码清单 23.6 所示。这个模板中实际上根据内核版本，去读取不同的 Makefile。

代码清单 23.6　同时支持 Linux 2.4/2.6 的内核模块的 Makefile 模板

```
#Makefile for 2.4 & 2.6
VERS26=$(findstring 2.6,$(shell uname -r))
MAKEDIR?=$(shell pwd)
ifeq ($(VERS26),2.6)
include $(MAKEDIR)/Makefile2.6
else
 include $(MAKEDIR)/Makefile2.4
 endif
```

2．内核模块加载时的版本检查

Linux 2.4 内核下，执行“cat /proc/ksyms”，将会看到内核符号，而且在名字后还会跟随着一串校验字符串，此校验字符串与内核版本有关。在内核源码头文件 linux/modules 目录下存在许多*.ver 文件，这些文件起着为内核符号添加校验后缀的作用，如 ksyms.ver 文件里有一行"#define printk _set_ver(printk)"，linux/modversions.h 文件会包含所有的.ver 文件。

所以当模块包含 linux/modversions.h 文件后，编译时，模块里使用的内核符号实质上成为带有校验后缀的内核符号。在加载模块时，如果模块使用的内核符号的校验字符串与当前运行内核所导出的相应的内核符号的校验字符串不一致，即当前内核空间并不存在模块所使用的内核符号，就会出现“Invalid module format”的错误。

Linux 内核所采用的在内核符号添加校验字符串来验证模块的版本与内核的版本

是否匹配的方法很复杂且会浪费内核空间，而且随着 SMP、PREEMPT 等机制在 Linux 2.6 内核的引入和完善，模块运行时对内核的依赖不再仅仅取决于内核版本，还取决于内核的配置，此时内核符号的校验码是否一致不能成为判断模块可否被加载的充分条件。

在 Linux 2.6 内核的 linux/vermagic.h 头文件中定义了“版本魔术字符串”——VERMAGIC_ STRING（如代码清单 23.7 所示），VERMAGIC_STRING 不仅包含内核版本号，还包含内核编译所使用的 GCC 版本、SMP 与 PREEMPT 等配置信息。在编译模块时，我们可以看到屏幕上会显示“MODPOST”（模块后续处理），在内核源码目录下 scripts/mod/modpost.c 文件中可以看到模块后续处理部分的代码。

就是在这个阶段，VERMAGIC_STRING 会被添加到模块的 modinfo 段中，模块编译生成后，通过“modinfo mymodule.ko”命令可以查看此模块的 vermagic 等信息。

Linux 2.6 内核下的模块装载器里保存有内核的版本信息，在装载模块时，装载器会比较所保存的内核 vermagic 与此模块的 modinfo 段里保存的 vermagic 信息是否一致，两者一致时，模块才能被装载。

代码清单 23.7　VERMAGIC_STRING 的定义

```
#ifdef CONFIG_SMP   //配置了 SMP
#define MODULE_VERMAGIC_SMP "SMP "
#else
#define MODULE_VERMAGIC_SMP ""
#endif

#ifdef CONFIG_PREEMPT    //配置了 PREEMPT
#define MODULE_VERMAGIC_PREEMPT "preempt "
#else
#define MODULE_VERMAGIC_PREEMPT ""
#endif

#ifdef CONFIG_MODULE_UNLOAD  //支持 module 卸载
#define MODULE_VERMAGIC_MODULE_UNLOAD "mod_unload "
#else
#define MODULE_VERMAGIC_MODULE_UNLOAD ""
#endif

#ifndef MODULE_ARCH_VERMAGIC  //体系结构 VERMAGIC
#define MODULE_ARCH_VERMAGIC ""
#endif

/* 拼接内核版本、上述 VERMAGIC 以及 GCC 版本 */
#define VERMAGIC_STRING              \
 UTS_RELEASE " "                     \
```

```
MODULE_VERMAGIC_SMP MODULE_VERMAGIC_PREEMPT  \
MODULE_VERMAGIC_MODULE_UNLOAD MODULE_ARCH_VERMAGIC \
"gcc-" __stringify(__GNUC__) "." __stringify(__GNUC_MINOR__)
```

在通过 make menuconfig 对内核进行新的配置后，再基于 Linux 2.6.15.5 内核编译生成的 hello.ko 模块（见第 4 章），这个模块的 modinfo 结果如下：

```
[root@localhost driver_study]# modinfo hello.ko
filename:       hello.ko
license:        Dual BSD/GPL
author:         Song Baohua
description:     A simple Hello World Module
alias:          a simplest module
vermagic:       2.6.15.5 SMP preempt PENTIUM4 gcc-3.2
depends:
```

从中可以看出，其 vermagic 为“2.6.15.5 SMP preempt PENTIUM4 gcc-3.2”，运行“insmod hello.ko”命令，得到如下错误：

```
insmod: error inserting 'hello.ko': -1 Invalid module format
hello: version magic '2.6.15.5 SMP preempt PENTIUM4 gcc-3.2' should be
'2.6.15.5 686 gcc-3.2'
```

原因在于加载该 hello.ko 时候所使用的内核虽然还是 Linux 2.6.15.5，但是和编译 hello.ko 时的内核的关键部分配置不一样，导致 vermagic 不一致，发生冲突，从而加载失败。

3．内核模块的加载与卸载函数

在 Linux 2.6 内核中，内核模块必须调用宏 module_init 与 module_exit() 去注册初始化与退出函数。在 Linux 2.4 内核中，如果加载函数命名为 init_module()，卸载函数命名为 cleanup_module()，可以不必使用 module_init 与 module_exit 宏。因此，若使用 module_init 与 module_exit 宏，代码在 Linux 2.4 内核与 Linux 2.6 内核中都能工作，如代码清单 23.8 所示。

代码清单 23.8　同时支持 Linux 2.4/2.6 的内核模块加载/卸载函数

```
static int mod_init_func(void)
{
  ...
  return 0;
}

static void mod_exit_func(void)
{
  ...
}
```

```
11
12 module_init(mod_init_func);
13 module_exit(mod_exit_func);
```

4．内核模块使用计数

不管是在 Linux 2.4 内核还是在 Linux 2.6 内核中，当内核模块正在被使用时，是不允许被卸载的，内核模板使用计数用来反映模块的使用情况。Linux 2.4 内核中，模块自身会通过 MOD_INC_USE_ COUNT、MOD_DEC_USE_COUNT 宏来管理自己被使用的计数。Linux 2.6 内核提供了更健壮、灵活的模块计数管理接口 try_module_get(&module)及 module_put (&module)取代 Linux 2.4 中的模块使用计数管理宏。而且，Linux 2.6 内核下，对于为具体设备写驱动的开发人员而言，基本无须使用 try_module_get()与 module_put()，设备驱动框架结构中的驱动核心往往已经承担了此项工作。

5．内核模块导出符号

在 Linux 2.4 内核下，默认情况下模块中的非静态全局变量及函数在模块加载后会输出到内核空间。而在 Linux 2.6 内核下，默认情况时模块中的非静态全局变量及函数在模块加载后不会输出到内核空间，需要显式调用宏 EXPORT_SYMBOL 才能输出。所以在 Linux 2.6 内核的模块下，EXPORT_NO_SYMBOLS 宏的调用没有意义，是空操作。在同时支持 Linux 2.4 内核与 Linux 2.6 内核的设备驱动中，可以通过代码清单 23.9 来导出模块的内核符号。

代码清单 23.9　同时支持 Linux 2.4/2.6 内核的导出内核符号代码段

```
1 #include <linux/module.h>
2 #ifndef LINUX26
3   EXPORT_NO_SYMBOLS;
4 #endif
5 EXPORT_SYMBOL(var);
6 EXPORT_SYMBOL(func);
```

另外，如果需要在 Linux 2.4 内核下使用 EXPORT_SYMBOL，必须在 CFLAGS 中定义 EXPORT_SYMTAB，否则编译将会失败。

从良好的代码风格角度出发，模块中不需要输出到内核空间且不需为模块中其他文件所用的全局变量及函数最好显式申明为 static 类型，需要输出的内核符号最好以模块名为前缀。模块加载后，Linux 2.4 内核下可通过/proc/ksyms，Linux 2.6 内核下可通过/proc/kallsyms 查看模块输出的内核符号。

6．内核模块输入参数

在 Linux 2.4 内核下，通过 MODULE_PARM(var,type)宏来向模块传递命令行参数。var 为接受参数值的变量名，type 为采取如下格式的字符串[min[-max]]{b,h,i,l,s}。min 及 max 用于表示当参数为数组类型时，允许输入的数组元素的个数范围；b 指 byte，h 指 short，i 指 int，l 指 long，s 指 string。

在 Linux 2.6 内核下，宏 MODULE_PARM(var,type)不再被支持，而是使用 module_param(name, type, perm)和 module_param_array(name, type, nump, perm)宏。

同样地，为了使驱动能根据内核的版本分别调用不同的宏导出内核符号，可以使用类似代码清单 23.10 所示的方法。

代码清单 23.10　同时支持 Linux 2.4/2.6 的模块输入参数范例

```
1  #include <linux/module.h>
2  #ifdef LINUX26
3    #include <linux/moduleparam.h>
4  #endif
5  int int_param  = 0;
6  char *string_param = "I love Linux";
7  int array_param[4] =
8  {
9    1, 1, 1, 1
10 };
11 #ifdef LINUX26
12   int len = 1;
13 #endif
14 #ifdef LINUX26
15   MODULE_PARM(int_param, "i");
16   MODULE_PARM(string_param, "s");
17   MODULE_PARM(array_param, "1-4i");
18 #else
19   module_param(int_param, int, 0644);
20   module_param(string_param, charp, 0644);
21   #if LINUX_VERSION_CODE >=
22     KERNEL_VERSION(2, 6, 10)
23     module_param_array(array_param, int,
24       &len, 0644);
25   #else
26     module_param_array(array_param, int, len, 0644);
27   #endif
28 #endif
```

7. 内核模块别名、加载接口

Linux 2.6 内核在 linux/module.h 中提供了 MODULE_ALIAS(alias)宏，模块可以通过调用此宏为自己定义一个或若干个别名。而在 Linux 2.4 内核下，用户只能在 /etc/modules.conf 中为模块定义别名。

加载内核模块的接口 request_module()在 Linux 2.4 内核下为 request_module(const

char * module_name)，在 Linux 2.6 内核下则为 request_module(const char *fmt, ...)。在 Linux 2.6 内核下，驱动开发人员可以通过调用以下的方法来加载内核模块。

```
request_module("xxx");
request_module("char-major-%d-%d", MAJOR(dev), MINOR(dev));
```

8．结构体初始化

在 Linux 2.4 内核中，习惯以代码清单 23.11 所示的方法来初始化结构体，即“成员:值”的方式。

代码清单 23.11　Linux 2.4 内核中结构体初始化习惯

```
1 static struct file_operations lp_fops =
2 {
3    owner: THIS_MODULE,
4    write: lp_write,
5    ioctl: lp_ioctl,
6    open: lp_open,
7    release: lp_release,
8 };
```

但是，在 Linux 2.6 内核中，为了尽量向标准 C 靠拢，习惯使用如代码清单 23.12 所示的方法来初始化结构体，即“.成员=值”的方式。

代码清单 23.12　Linux 2.6 内核中结构体初始化习惯

```
1 static struct file_operations lp_fops =
2 {
3    .owner         = THIS_MODULE,
4    .write         = lp_write,
5    .ioctl         = lp_ioctl,
6    .open          = lp_open,
7    .release       = lp_release,
8 };
```

9．字符设备驱动

在 Linux 2.6 内核中，将 Linux 2.4 内核中都为 8 位的主次设备号分别扩展为 12 位和 20 位。鉴于此，Linux 2.4 内核中的 kdev_t 被废除，Linux 2.6 内核中新增的 dev_t 拓展到了 32 位。在 Linux 2.4 内核中，通过 inode->i_rdev 即可得到设备号，而在 Linux 2.6 内核中，为了增强代码的可移植性，内核中新增了 iminor()和 imajor()这两个函数来从 inode 获得设备号。

在 Linux 2.6 内核中，对于字符设备驱动，提供了专门用于申请/动态分配设备号的 register_ chrdev_region()函数和 alloc_chrdev_region()函数，而在 Linux 2.4 内核中，对设备号的申请和注册字符设备的行为都是在 register_chrdev()函数中进行的，没有单独的 cdev 结构体，因此也不存在 cdev_init()、cdev_add()、cdev_del()这些函数。要注意的是，register_chrdev()在 Linux 2.6 内核中仍然被支持，但是不能访问超过 256 的设备号。

其次，devfs 设备文件系统在 Linux 2.6 内核中被取消了，因此，最新的驱动中也

不宜再调用 devfs_register()、devfs_unregister()这样的函数。

- proc 操作。

以前的/proc 中只能给出字符串，而新增的 seq_file 操作使得/proc 中的文件能导出如 long 等多种数据，为了支持这一新的特性，需要实现 seq_operations 结构体中的 seq_printf()、seq_putc()、seq_puts()、seq_escape()、seq_path()、seq_open()等成员函数。

- 内存分配。

Linux 2.4 和 Linux 2.6 在内存分配方面发生了一些细微的变化，这些变化主要包括：

＜linux/malloc.h＞头文件被改为＜linux/slab.h＞；

分配标志 GFP_BUFFER 被 GFP_NOIO 和 GFP_NOFS 取代；

新增了__GFP_REPEAT、__GFP_NOFAIL 和__GFP_NORETRY 分配标志；

页面分配函数 alloc_pages()、get_free_page()被包含在＜linux/gfp.h＞中；

对 NUMA 系统新增了 alloc_pages_node()、free_hot_page()、free_cold_page()函数；

新增了内存池；

针对 r-cpu 变量的 DEFINE_PER_CPU()、EXPORT_PER_CPU_SYMBOL()、EXPORT_PER_CPU_SYMBOL_GPL()、DECLARE_PER_CPU()、DEFINE_PER_CPU()等宏因为抢占调度的出现而变得不安全，被 get_cpu_var()、put_cpu_var()、alloc_percpu()、free_percpu()、per_cpu_ptr()、get_cpu_ptr()、put_cpu_ptr()等函数替换。

- 内核时间变化。

在 Linux 2.6 中，一些平台的节拍（HZ）发生了变化，因此引入了新的 64 位计数器 jiffies_64，新的时间结构体 timespec 增加了 ns 成员变量，新增了 add_timer_on()定时器函数，新增了 ns 级延时函数 ndelay()。

- 并发/同步。

任务队列（task queue）接口函数都被取消，新增了 work queue 接口函数。

- 音频设备驱动。

Linux 2.4 内核中音频设备驱动的默认框架是 OSS，而 Linux2.6 内核中音频设备驱动的默认框架则是 ALSA，这显示 ALSA 是一种未来的趋势。

在内核的更新过程中，大部分驱动源代码也随着内核中的 API 变更而修改了，如下面的列表分别摘录了 linux-2.4.18 和 linux-2.6.15.5 中的并口打印机字符设备驱动 drivers/char/lp.c 的源代码：

static struct file_operations lp_fops = {
owner: THIS_MODULE,
write: lp_write,
ioctl: lp_ioctl,
open: lp_open,
release: lp_release,
#ifdef CONFIG_PARPORT_1284
	static struct file_operations lp_fops = {
	.owner = THIS_MODULE,
	.write = lp_write,
	.ioctl = lp_ioctl,
	.open = lp_open,
	.release = lp_release,
	#ifdef CONFIG_PARPORT_1284
	.read = lp_read,

```
        read:       lp_read,
    #endif
    };

    MODULE_PARM(parport,"1-"_
_MODULE_
STRING(LP_NO) "s");
    MODULE_PARM(reset, "i");


    static   int   lp_release(struct
inode
      *inode, struct file *file)
    {
      unsigned    int    minor    =
MINOR(inode
        ->i_rdev);


      lp_claim_parport_or_block
        (&lp_table[minor]);

parport_negotiate(lp_table[minor].dev
        ->port,
IEEE1284_MODE_COMPAT);
      lp_table[minor].current_mode
=
        IEEE1284_MODE_COMPAT;

lp_release_parport(&lp_table[minor]);
      lock_kernel();


kfree(lp_table[minor].lp_buffer);
      lp_table[minor].lp_buffer    =
NULL;
      LP_F(minor) &= ~LP_BUSY;
      unlock_kernel();
      return 0;
    }
```

```
    #endif
    };

    module_param_array(parport,  charp,
NULL, 0);
    module_param(reset, bool, 0);


    static   int   lp_release(struct
inode
      *inode, struct file *file)
    {
      unsigned    int    minor    =
iminor(inode);


      lp_claim_parport_or_block
        (&lp_table[minor]);

parport_negotiate(lp_table[minor].dev
        ->port,
IEEE1284_MODE_COMPAT);
      lp_table[minor].current_mode
=
        IEEE1284_MODE_COMPAT;

lp_release_parport(&lp_table[minor]);


kfree(lp_table[minor].lp_buffer);
      lp_table[minor].lp_buffer    =
NULL;
      LP_F(minor) &= ~LP_BUSY;

      return 0;
    }
```

如果驱动源代码要同时支持 Linux 2.4 和 Linux 2.6 内核，其实也非常简单，因为通过 linux/version.h 中的 LINUX_VERSION_CODE 可以获知内核版本，之后便可以针对不同的宏定义实现不同的驱动源代码，如代码清单 23.13 所示。

代码清单 23.13　同时支持 Linux 2.4 和 Linux 2.6 内核的驱动编写方法

```
#include <linux/version.h>
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2, 6, 0)
 #define LINUX26
#endif
#ifdef LINUX26
```

```
6   /*Linux 2.6 内核中的代码*/
7 #else
8   /*Linux 2.4 内核中的代码 */
9 #endif
```

23.4 Linux 与其他操作系统之间的驱动移植

在公司的项目更替过程中，可能会出现操作系统的更换，譬如类似的产品中，以前用 VxWorks 或 Windows CE，现在想改用 Linux。以前同类产品的 VxWorks/WinCE 运行稳定，且设备驱动也被经过严格测试，现在要换成 Linux 了，是不是 VxWorks/WinCE 下的工作就完全作废了呢？

在采用原操作系统的系统中，底层的硬件操作代码已经经过验证，这一部分可以被 Linux 利用；VxWorks、WindowsCE 等操作系统驱动的架构和 Linux 的驱动架构有一定的相似性，经过适当的修改，就可以被移植到 Linux 中。

本节将以 VxWorks 为例讲解 VxWorks 驱动向 Linux 驱动的移植方法。代码清单 23.14 所示为 VxWorks 下的 LPT 并口这一典型的字符设备驱动的骨干。

代码清单 23.14 VxWorks 下的 LPT 驱动

```
1   LOCAL int lptOpen(LPT_DEV *pDev, char *name, int mode);
2   LOCAL int lptRead(LPT_DEV *pDev, char *pBuf, int size);
3   LOCAL int lptWrite(LPT_DEV *pDev, char *pBuf, int size);
4   LOCAL STATUS lptIoctl(LPT_DEV *pDev, int function, int arg);
5   LOCAL void lptIntr(LPT_DEV *pDev);
6   LOCAL void lptInit(LPT_DEV *pDev);
7
8   /*初始化设备驱动*/
9   STATUS lptDrv(int channels,  /* LPT 通道 */
10  LPT_RESOURCE *pResource /* LPT 资源 */
11  )
12  {
13    int ix;
14    LPT_DEV *pDev;
15
16    /* 检查驱动是否已被安装 */
17    if (lptDrvNum > 0)
18      return (OK);
19
20    if (channels > N_LPT_CHANNELS)
21      return (ERROR);
22
```

```
    for (ix = 0; ix < channels; ix++, pResource++)
    {
      pDev = &lptDev[ix];

      pDev->created = FALSE;
      pDev->autofeed = pResource->autofeed;
      pDev->inservice = FALSE;

      if (pResource->regDelta == 0)
        pResource->regDelta = 1;

      pDev->data = LPT_DATA_RES(pResource);
      pDev->stat = LPT_STAT_RES(pResource);
      pDev->ctrl = LPT_CTRL_RES(pResource);
      pDev->intCnt = 0;
      pDev->retryCnt = pResource->retryCnt;
      pDev->busyWait = pResource->busyWait;
      pDev->strobeWait = pResource->strobeWait;
      pDev->timeout = pResource->timeout;
      pDev->intLevel = pResource->intLevel;

      //创建二进制信号量
      pDev->syncSem = semBCreate(SEM_Q_FIFO, SEM_EMPTY);
      //创建互斥信号量
      pDev->muteSem = semMCreate(SEM_Q_PRIORITY | SEM_DELETE_SAFE
       | SEM_INVERSION_SAFE);
      //连接中断
(void)intConnect((VOIDFUNCPTR*)INUM_TO_IVEC(pResource->intVector),
        (VOIDFUNCPTR)lptIntr, (int)pDev);

      sysIntEnablePIC(pDev->intLevel); /* 开中断 */

      lptInit(&lptDev[ix]);
    }

    //注册驱动
    lptDrvNum = iosDrvInstall(lptOpen, (FUNCPTR)NULL, lptOpen,
      (FUNCPTR)NULL,lptRead, lptWrite, lptIoctl);

```

```
  return (lptDrvNum == ERROR ? ERROR :  OK);
}


/* lptOpen - 打开 LPT  */
LOCAL int lptOpen(LPT_DEV *pDev, char *name, int mode)
{
  return ((int)pDev);
}

/* lptRead - 读并口 */
LOCAL int lptRead(LPT_DEV *pDev, char *pBuf, int size)
{
  return (ERROR);
}

/* lptWrite - 写并口
 * 返回值:  被写入的字节数, 或者 ERROR
 */
LOCAL int lptWrite(LPT_DEV *pDev, char *pBuf, int size)
{
  int byteCnt = 0;
  BOOL giveup = FALSE;
  int retry;
  int wait;

  if (size == 0)
    return (size);

  semTake(pDev->muteSem, WAIT_FOREVER);//获取互斥

  retry = 0;
  while ((sysInByte(pDev->stat) &S_MASK) != (S_SELECT | S_NODERR | S_NOBUSY))
  {
    if (retry++ > pDev->retryCnt)
    {
      if (giveup)
```

```
    {
      errnoSet(S_ioLib_DEVICE_ERROR);
      semGive(pDev->muteSem);
      return (ERROR);
    }
    else
    {
      lptInit(pDev);
      giveup = TRUE;
    }
  }
  wait = 0;
  while (wait != pDev->busyWait)
    wait++;
}

retry = 0;
do
{
  wait = 0;
  sysOutByte(pDev->data,  *pBuf);
  while (wait != pDev->strobeWait)wait++;
    sysOutByte(pDev->ctrl,  sysInByte(pDev->ctrl)  |  C_STROBE  |
C_ENABLE);
  while (wait)
    wait--;
  sysOutByte(pDev->ctrl, sysInByte(pDev->ctrl) &~C_STROBE);

  if (semTake(pDev->syncSem, pDev->timeout *sysClkRateGet()) ==
ERROR)
  {
    if (retry++ > pDev->retryCnt)
    {
      errnoSet(S_ioLib_DEVICE_ERROR);
      semGive(pDev->muteSem);//释放互斥
      return (ERROR);
    }
  }
  else
  {
```

```
    pBuf++;
    byteCnt++;
  }
}while (byteCnt < size) ;

semGive(pDev->muteSem);//释放互斥

return (size);
}

/* lptIoctl - 设备特定的控制 */
LOCAL STATUS lptIoctl(LPT_DEV *pDev,  /* 要控制的设备 */
int function,  /* 命令 */
int arg /* 参数 */
)
{
  int status = OK;

  semTake(pDev->muteSem, WAIT_FOREVER);
  switch (function)
  {
    case LPT_SETCONTROL:
      sysOutByte(pDev->ctrl, arg);
      break;

    case LPT_GETSTATUS:
      *(int*)arg = sysInByte(pDev->stat);
      break;

    default:
      (void)errnoSet(S_ioLib_UNKNOWN_REQUEST);
      status = ERROR;
      break;
  }
  semGive(pDev->muteSem);

  return (status);
}
```

```
175
176 /* lptIntr - 中断处理函数 */
177 LOCAL void lptIntr(LPT_DEV *pDev)
178 {
179   pDev->inservice = TRUE;
180   pDev->intCnt++;
181   semGive(pDev->syncSem);  //释放同步信号量
182   pDev->inservice = FALSE;
183   sysOutByte(pDev->ctrl, sysInByte(pDev->ctrl) &~C_ENABLE);
184 }
185
186 /* lptInit - 初始化 LPT 端口 */
187 LOCAL void lptInit(LPT_DEV *pDev)
188 {
189   sysOutByte(pDev->ctrl, 0); /* init */
190   taskDelay(sysClkRateGet() >> 3); /* hold min 50 mili sec */
191   if (pDev->autofeed)
192     sysOutByte(pDev->ctrl, C_NOINIT | C_SELECT | C_AUTOFEED);
193   else
194     sysOutByte(pDev->ctrl, C_NOINIT | C_SELECT);
195 }
```

上述驱动和 Linux 下的字符设备驱动很多相似之处，有如下体现。

- 同样包含了 open()、read()、write()、ioctl()等函数，而且同样要注册驱动（使用 iosDrvInstall()）；
- 同样包含中断号和中断处理函数的绑定过程，只是用 intConnect()，而不是 request_irq()；
- 为了避免并发和竞争或进行同步，同样定义了信号量和互斥，只是信号量和互斥的名字发生了变化，而且用 semBCreate()、semMCreate()这样的函数来创建。在访问临界资源时，也一样用互斥加以保护。

两者的差异如下。

- 内核对象、API 的名称和参数有很大不同，VxWorks 与 Linux 对等地位的函数的返回值也不相同。表 23.2 所示为 VxWorks 和 Linux 在同步、互斥和中断方面的对比。

表 23.2　　VxWorks 和 Linux 同步/互斥/中断对比

事　　项	VxWorks	Linux
临界资源保护	互斥（二进制信号量）	自旋锁、信号量
中断	没有顶底半部的概念，但是中断服务程序也通过 semGive()和 MsgQSend()等方式唤醒任务去完成中断引发的事务	有顶底半部的概念，顶半部通过软中断、tasklet、workqueue 触发底半部，也可以只是 wake_up 一个进程

同步	同步 semGive()、MsgQSend()进行	通过等待队列、完成量进行

- VxWorks 中的命名习惯和 Linux 系统不同，VxWorks 常用“xxxYyyZzz”命名方式，而 Linux 常用“xxx_yyy_zzz”命名规则。
- VxWorks 对外设的访问中不会出现类似 ioremap()这样的物理地址到虚拟地址的映射过程。尽管 VxWorks 也能支持带 MMU 的处理器，但是对系统中的所有任务而言，其进行的是完全相同的物理地址到虚拟地址映射。

因此，在移植 VxWorks 驱动到 Linux 驱动的过程中，可以借鉴其流程，但是必须要替换 API、函数和变量命名等。

除了简单的字符设备外，VxWorks 下的 TTY 设备、MTD 设备、块设备等驱动的架构都和 Linux 下的架构有一定相似性，但是，由于 VxWorks 是一种完全定位于嵌入式系统的轻量级操作系统，因此，总体而言，VxWorks 设备驱动的架构要比 Linux 架构简单。由于 VxWorks 中并不存在内核空间和用户空间的界限，很多时候，驱动甚至可以不遵循架构，直接给应用程序提供接口。

接下来分析几个典型的 VxWorks 设备驱动的框架结构。

1．串行设备

VxWorks 下串行设备驱动的结构如图 23.1 所示，I/O 系统（VxWorks 的 I/O 系统类似于 Linux 的 VFS，都是向用户提供统一的文件操作接口 open()、write()、read()、ioctl()、close()等）不直接与串行设备驱动打交道，而是通过 ttyDrv 进行。

ttyDrv 是一个虚拟的设备驱动，它与 tylib 一起，用于处理 I/O 系统与底层实际设备之间的通信。它们完成的工作包括在驱动表中添加相应的驱动条目、创建设备标识符，实现与上层标准 I/O 函数及实际驱动程序的连接，ttyDrv 完成 open()和 ioctl()两项功能（对应函数 ttyopen()和 ttyioctl()），tylib 完成 read()和 write()两项功能（对应函数 tyRead()和 tyWrite()）并管理输入/输出数据缓冲区。

图 23.1 VxWorks 串行设备驱动的结构

在上下层数据传递方面，VxWorks 使用如图 23.2 所示的结构（假设串口 UART 为 8250）。

当用户调用 write()函数进行写操作时，系统根据相应的文件描述符 fd 调用驱动表中注册的 tyWrite()函数，此函数会将用户缓冲区的内容写入相应的输出 ring buffer。当发现缓冲区内有内容时，函数 tyITx()被调用，从 ring buffer 读取字符，将字符发往指定的串口。

图 23.2　VxWorks 串行设备驱动数据流

当串口接收到数据时会调用输入中断服务程序，将输入的字符写入指定的缓冲区。然后由回调函数 tyIRd()将缓冲区的内容读入 ring buffer，当用户调用 read()函数进行写操作时，会根据文件描述符 fd 调用在驱动表中注册的函数 tyRead()，此函数会将 ring buffer 中的内容读入用户缓冲区。

2．块设备

如图 23.3 所示，与 Linux 类似，VxWorks 中的块设备驱动程序也不与 I/O 系统直接作用，而是通过磁盘、Flash 文件系统与 I/O 系统交互。文件系统把自己作为一个驱动程序装载到 I/O 系统中，并把请求转发给实际的设备驱动程序。块设备的驱动程序不使用 iosDrvlnstsll()来安装驱动程序，而是通过初始化块设备描述结构 BLK_DEV 或顺序设备描述结构 SEQ_DEV，来实现驱动程序提供给文件系统的功能。块设备也不调用非块设备的安装函数 iosDevAdd()，而是调用文件系统的设备初始化函数，如 dosFsDevlnit()等。

3．网络设备

在 VxWorks 中，网卡驱动程序分为 END（Enhanced Network Driver，增强型网络驱动）和 BSD 两种。

如图 23.4 所示，END 驱动程序基于 MUX 模式，网络驱动程序被划分为协议组件和硬件组件。MUX 是数据链路层和网络层之间的接口，它管理网络协议接口和底层硬件接口之间的交互。

图 23.3　VxWorks 块设备驱动与 I/O 系统

图 23.4　VxWorks 下网络设备 END 驱动

如图 23.5 所示，MUX 数据包采用 mBlk-clBlk-cluster 数据结构处理网络协议栈传输的数据。其中，cluster 保存的是实际的数据，mBlk 和 clBlk 中保存的信息用来管理 cluster 中保存的数据。

图 23.5　VxWorks 下网络设备驱动的 mBlk-clBlk-cluster 结构

在数据接收过程中，设备会直接将接收到的数据包放入内存池预先分配的 cluster 中并产生一个中断。如果设备不能完成上述功能，END 驱动函数应该完成将数据从 buffer 到 cluster 中的拷贝。数据被放到 cluster 以后，驱动程序将通过对 netBuflib()中函数的调用来完成 mBlk-clBlk-cluster 链的创建，从而为数据在网络协议各层之间的传递做好准备，创建此结构链一般需要以下 4 步。

（1）调用函数 netClblkGet()，从内存池中取 cluster 结构。

（2）调用函数 netClblkJoin()，将 clBlk 与存有数据的 cluster 连接起来。

（3）调用函数 netMblkGet()，从内存池中取 mBlk 结构。

（4）调用函数 netMblkClJoin()，将 mBlk 与 clBlk-cluster 结构连接起来。

END 设备驱动的数据包接收过程的整个流程如图 23.6 所示，经历了“设备中断服务程序→调用 netjobAdd()添加网络任务（类似于 Linux 中的底半部，引发“底半部”xxReceive 被执行）→到达 MUX 层→协议层→用户 read()函数”的过程。

数据包的发送是数据包接收的反过程，应用程序通过 write()函数调用将要发送的数据放入应用程序数据缓冲区中后，网络协议负责将 bufer 中的数据放入为其分配的内存池中，并以 mBlk-clBlk-cluster 链的形式来存储，这样实现了向下层协议传递数据报时，传递的只是指向此数据链结构的指针，而代替了数据在各层协议之间的拷贝。

当有数据要发送时，网络协议层通过其与 MUX 层的接口调用 muxSend()函数，而 muxSend()函数又调用 xxSend()函数将传递来的数据包送到发送 FIFO 队列中，然后起动网卡设备的发送功能，发送完后将随之产生中断信号，调用中断服务程序，清除

设备缓冲区。

与 VxWorks 相比，Linux 网络设备驱动中没有 mBlk-clBlk-cluster，贯穿始终的是 sk_buff，作为数据包的“容器”，其地位与 mBlk-clBlk-cluster 相当。

图 23.6　VxWorks 网络设备驱动的数据包接收流程　　图 23.7　VxWorks 网络设备驱动的数据包发送流程

4．PCI 设备

在 VxWorks 中，PCI 设备驱动和 Linux 系统类似，PCI 部分只是一个“外壳”，设备本身所属类型的驱动才是工作的主体。与 Linux 系统类似，在 VxWorks 中，初始化 PCI 设备需要如下几个步骤。

（1）利用供应商和设备标识确定设备的总线号、设备号和功能号，在系统中找到设备，要用的 API 是包括 pciFindDevice()、pciFindClass()，前者根据 ID、总线编号、设备编号、功能编号找到一个特定的 PCI 设备，后者寻找与参数中类匹配的 PCI 设备。

（2）进行 PCI 设备的配置，和 Linux 系统非常相似，使用的 API 包括 pciConfigInByte()、pciConfigInWord()、pciConfigInLong()、pciConfigOutByte()、pciConfigOutWord()、pciConfigOutLong()、pciConfigModifyLong()、pciConfigModifyWord() 和 pciConfigModifyByte()等。

（3）确定映射到系统中的设备的基地址，即分析 PCI 设备的 I/O 内存/端口资源。

（4）挂接 PCI 设备中断。

5．USB 设备

如图 23.8 所示，与 Linux 系统相似，VxWorks 中的 USB 驱动也分成了主机控制器驱动（HCD）、USB 核心驱动层（USBD）和 USB 设备驱动（Client Driver）几个层次。

Client Driver 负责管理连接到 USB 上的不同设备，它通过 IRP（I/O 请求包，与 Linux 系统中的 URB 类似）向 USBD 层发出数据接收或发送报文，它向应用层提供 API 函数，屏蔽 USB 实现的细节，实现数据的透明传输。

USBD 通过 IRP 得到此设备的属性和本次数据通信的要求，将 IRP 转换成 USB 所能辨识的一系列事务处理，交给 HCD 层或者直接交给 HCD。此外，USBD 还负责新设备的配置、被拔掉设备资源的释放和 Client Driver 的装载/卸载。

图 23.8　VxWorks 下的 USB 驱动层次

HCD 完成对 Host 控制器的管理、带宽分配、链表管理及根 Hub 功能等。它将数据按传输类型组成不同的链表，并定义不同类型传输在一帧中所占带宽的比例，交给 Host 控制器处理，控制器根据规则从链表上摘下数据块，为其创建一个或多个事务，完成与设备的数据传输。当事务处理完成时，HCD 将结果交给 USBD 层，由它通知对 Client Driver 进行处理。

从以上分析可知，对于块设备、网络设备、PCI 设备、USB 设备等复杂设备而言，VxWorks 和 Linux 驱动的框架呈现出了较大的差异，但其中也不乏共同的设计思想。Linux 驱动工程师应该在了解这些差异的情况下，尽可能地利用 VxWorks 中现成的稳定的代码以减少 Linux 驱动的工作量，至少硬作控制这一部分是可以通用的。

23.5 总结

在编写 Linux 设备驱动的程序，要特别注意代码的可移植性，要留意数据类型的长度、结构体的对界、CPU 大小端模式以及内存页面的大小。

为了加速驱动的开发过程，在拿到一个驱动开发任务的时候，务必搜集足够的“情报”，找到可模拟的芯片或可利用的代码，这样可以事半功倍。一般而言，demo 板的驱动、类似芯片的驱动以及无操作系统时的硬件操作代码都是可以参考的代码。

Linux 2.4 到 Linux 2.6 的改进导致驱动中发生了一些细微的变化，了解这些变化后可进行驱动的更新。除了不同版本的 Linux 以外，其他操作系统中的驱动源代码经过适当的修改也可被移植到 Linux 中

推荐课程： 嵌入式学院-嵌入式 Linux 长期就业班

· 招生简章：http://www.embedu.org/courses/index.htm