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

附源代码光盘

人民邮电出版社
POSTS & TELECOM PRESS

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

```
1  typedef signed char s8;
2  typedef unsigned char u8;
3
4  typedef signed short s16;
5  typedef unsigned short u16;
6
7  typedef signed int s32;
8  typedef unsigned int u32;
9
10 typedef signed long long s64;
11 typedef unsigned long long u64;
```

代码清单 23.2 ppc64 平台下 u8、u16、u32、u64、s8、s16、s32、s64 的定义

```
1  typedef signed char s8;
2  typedef unsigned char u8;
3
4  typedef signed short s16;
5  typedef unsigned short u16;
6
7  typedef signed int s32;
8  typedef unsigned int u32;
9
10 typedef signed long s64;
11 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>
```

```
2 int order = get_order(64*1024);
3 buf = get_free_pages(GFP_KERNEL, order);
```

23.2 巧用同类设备驱动

23.2.1 巧用 demo 板驱动

对于推出的重要芯片，芯片厂商往往会同时提供一套 demo 板。这样的 demo 板不仅在硬件设计中被硬件工程师充分利用并进行参考，而且其提供的驱动程序往往在新设计的硬件系统中被参考。

借用 demo 板驱动的方法主要是寻找共性中的差异，例如共性是芯片相同，差异则可能体现在所使用的 I/O 内存（片选）、中断和 DMA 资源不同，在这种情况下，简单地修改 I/O 内存基地址、中断号以及 DMA 通道，demo 板的驱动就可以用在目标电路板上。而如果除了芯片相同以外，外围芯片与 CPU 连接所用的内存、中断和 DMA 资源都相同的话，则 demo 板驱动基本上可以不加任何修改地搬到目标电路板上。

如果 demo 板和目标电路板所用资源不同，而 demo 板对应设备被定义为 pltaform_device，且其资源并定义在 resource 结构体数组中，则直接修改 resource 结构体即可，如下所示：

```
static struct resource xxx_resource[] =
{
     [0] =
     {
          .start = XXX_MEM_START, //修改这里替换 I/O 内存基地址
          .end   = XXX_MEM_START + XXX_MEM_SIZE,
          .flags = IORESOURCE_MEM,
     },

     [1] =
     {
          .start = XXX_INT_START, //修改这里换中断
          .end   = XXX_INT_END,
          .flags = IORESOURCE_IRQ,
     }
};
```

许多情况下，驱动中的资源被单独用宏、变量或结构体定义，这时候要学会怎样快速地找到驱动中所使用的资源。其实方法很简单，那就是在 demo 板驱动的源代码

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。

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

```
1  #Makefile for 2.4 & 2.6
2  VERS26=$(findstring 2.6,$(shell uname -r))
3  MAKEDIR?=$(shell pwd)
4  ifeq ($(VERS26),2.6)
5  include $(MAKEDIR)/Makefile2.6
6  else
7   include $(MAKEDIR)/Makefile2.4
8   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

```
1  #ifdef CONFIG_SMP   //配置了 SMP
2  #define MODULE_VERMAGIC_SMP "SMP "
3  #else
4  #define MODULE_VERMAGIC_SMP ""
5  #endif
6
7  #ifdef CONFIG_PREEMPT    //配置了 PREEMPT
8  #define MODULE_VERMAGIC_PREEMPT "preempt "
9  #else
10 #define MODULE_VERMAGIC_PREEMPT ""
11 #endif
12
13 #ifdef CONFIG_MODULE_UNLOAD  //支持 module 卸载
14 #define MODULE_VERMAGIC_MODULE_UNLOAD "mod_unload "
15 #else
16 #define MODULE_VERMAGIC_MODULE_UNLOAD ""
17 #endif
18
19 #ifndef MODULE_ARCH_VERMAGIC  //体系结构 VERMAGIC
20 #define MODULE_ARCH_VERMAGIC ""
21 #endif
22
23 /* 拼接内核版本、上述 VERMAGIC 以及 GCC 版本 */
24 #define VERMAGIC_STRING              \
25  UTS_RELEASE " "                     \
```

```
26  MODULE_VERMAGIC_SMP MODULE_VERMAGIC_PREEMPT  \
27  MODULE_VERMAGIC_MODULE_UNLOAD MODULE_ARCH_VERMAGIC \
28  "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

```
1  static int mod_init_func(void)
2  {
3    ...
4    return 0;
5  }
6
7  static void mod_exit_func(void)
8  {
9    ...
10 }
```

```
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

```
1 #include <linux/version.h>
2 #if LINUX_VERSION_CODE >= KERNEL_VERSION(2, 6, 0)
3   #define LINUX26
4 #endif
5 #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
```

```
23    for (ix = 0; ix < channels; ix++, pResource++)
24    {
25      pDev = &lptDev[ix];
26
27      pDev->created = FALSE;
28      pDev->autofeed = pResource->autofeed;
29      pDev->inservice = FALSE;
30
31      if (pResource->regDelta == 0)
32        pResource->regDelta = 1;
33
34      pDev->data = LPT_DATA_RES(pResource);
35      pDev->stat = LPT_STAT_RES(pResource);
36      pDev->ctrl = LPT_CTRL_RES(pResource);
37      pDev->intCnt = 0;
38      pDev->retryCnt = pResource->retryCnt;
39      pDev->busyWait = pResource->busyWait;
40      pDev->strobeWait = pResource->strobeWait;
41      pDev->timeout = pResource->timeout;
42      pDev->intLevel = pResource->intLevel;
43
44      //创建二进制信号量
45      pDev->syncSem = semBCreate(SEM_Q_FIFO, SEM_EMPTY);
46      //创建互斥信号量
47      pDev->muteSem = semMCreate(SEM_Q_PRIORITY | SEM_DELETE_SAFE
48       | SEM_INVERSION_SAFE);
49      //连接中断
50
(void)intConnect((VOIDFUNCPTR*)INUM_TO_IVEC(pResource->intVector),
51        (VOIDFUNCPTR)lptIntr, (int)pDev);
52
53      sysIntEnablePIC(pDev->intLevel); /* 开中断 */
54
55      lptInit(&lptDev[ix]);
56    }
57
58    //注册驱动
59    lptDrvNum = iosDrvInstall(lptOpen, (FUNCPTR)NULL, lptOpen,
60      (FUNCPTR)NULL,lptRead, lptWrite, lptIoctl);
61
```

```
    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)
```

```
    99      {
    100       errnoSet(S_ioLib_DEVICE_ERROR);
    101       semGive(pDev->muteSem);
    102       return (ERROR);
    103     }
    104     else
    105     {
    106       lptInit(pDev);
    107       giveup = TRUE;
    108     }
    109   }
    110   wait = 0;
    111   while (wait != pDev->busyWait)
    112     wait++;
    113  }
    114
    115  retry = 0;
    116  do
    117  {
    118    wait = 0;
    119    sysOutByte(pDev->data,  *pBuf);
    120    while (wait != pDev->strobeWait)wait++;
    121      sysOutByte(pDev->ctrl,  sysInByte(pDev->ctrl)  |  C_STROBE  |
C_ENABLE);
    122    while (wait)
    123      wait--;
    124    sysOutByte(pDev->ctrl, sysInByte(pDev->ctrl) &~C_STROBE);
    125
    126    if (semTake(pDev->syncSem, pDev->timeout *sysClkRateGet()) ==
ERROR)
    127    {
    128      if (retry++ > pDev->retryCnt)
    129      {
    130        errnoSet(S_ioLib_DEVICE_ERROR);
    131        semGive(pDev->muteSem);//释放互斥
    132        return (ERROR);
    133      }
    134    }
    135    else
    136    {
```

```
137      pBuf++;
138      byteCnt++;
139    }
140  }while (byteCnt < size) ;
141
142  semGive(pDev->muteSem);//释放互斥
143
144  return (size);
145 }
146
147 /* lptIoctl - 设备特定的控制 */
148 LOCAL STATUS lptIoctl(LPT_DEV *pDev,  /* 要控制的设备 */
149 int function,  /* 命令 */
150 int arg /* 参数 */
151 )
152 {
153  int status = OK;
154
155  semTake(pDev->muteSem, WAIT_FOREVER);
156  switch (function)
157  {
158    case LPT_SETCONTROL:
159      sysOutByte(pDev->ctrl, arg);
160      break;
161
162    case LPT_GETSTATUS:
163      *(int*)arg = sysInByte(pDev->stat);
164      break;
165
166    default:
167      (void)errnoSet(S_ioLib_UNKNOWN_REQUEST);
168      status = ERROR;
169      break;
170  }
171  semGive(pDev->muteSem);
172
173  return (status);
174 }
```

```
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。

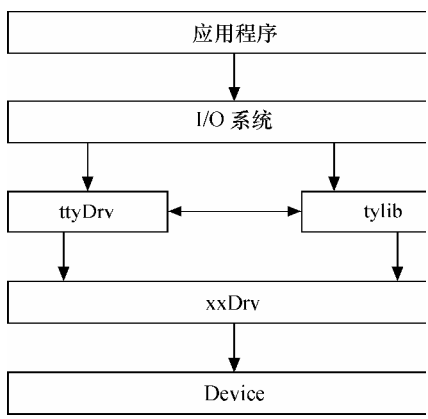

图 23.1　VxWorks 串行设备驱动的结构

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。

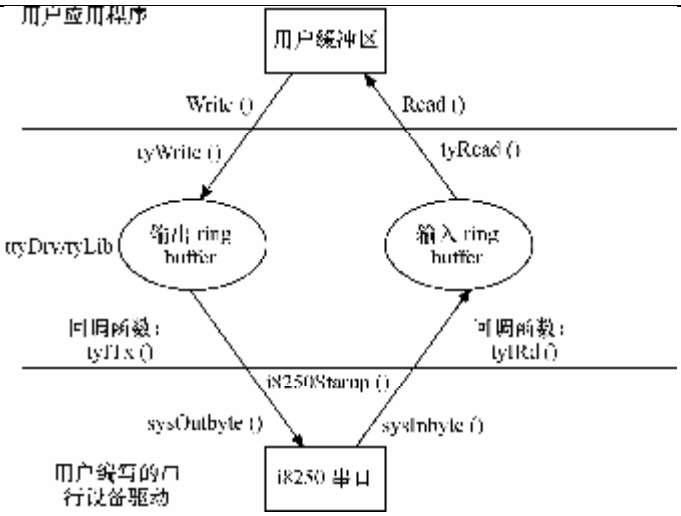

图 23.2　VxWorks 串行设备驱动数据流

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。

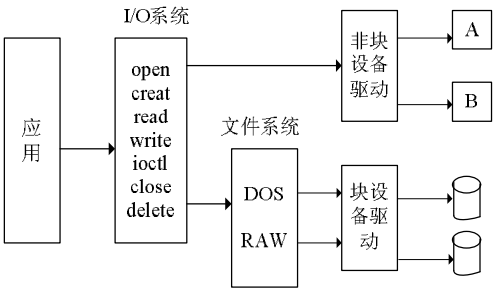

图 23.3　VxWorks 块设备驱动与 I/O 系统

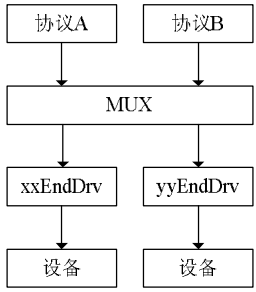

图 23.4　VxWorks 下网络设备 END 驱动

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。

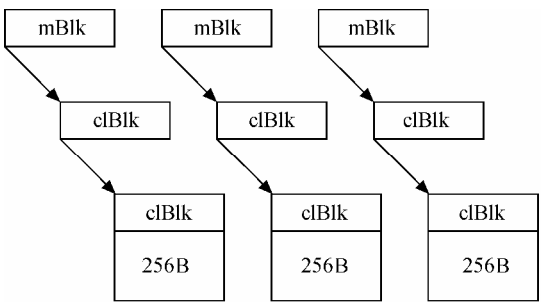

图 23.5　VxWorks 下网络设备驱动的 mBlk-clBlk-cluster 结构

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。

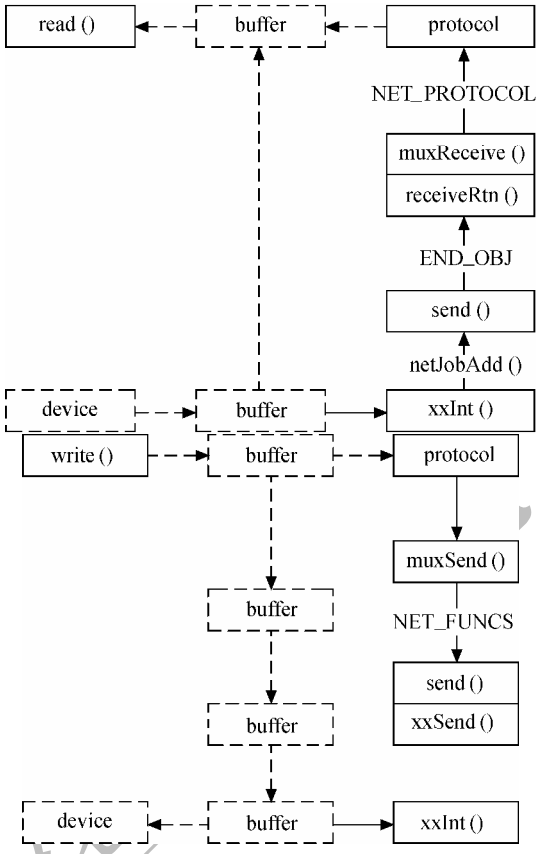

图 23.6　VxWorks 网络设备驱动的数据包接收流程　　图 23.7　VxWorks 网络设备驱动的数据包发送流程

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。

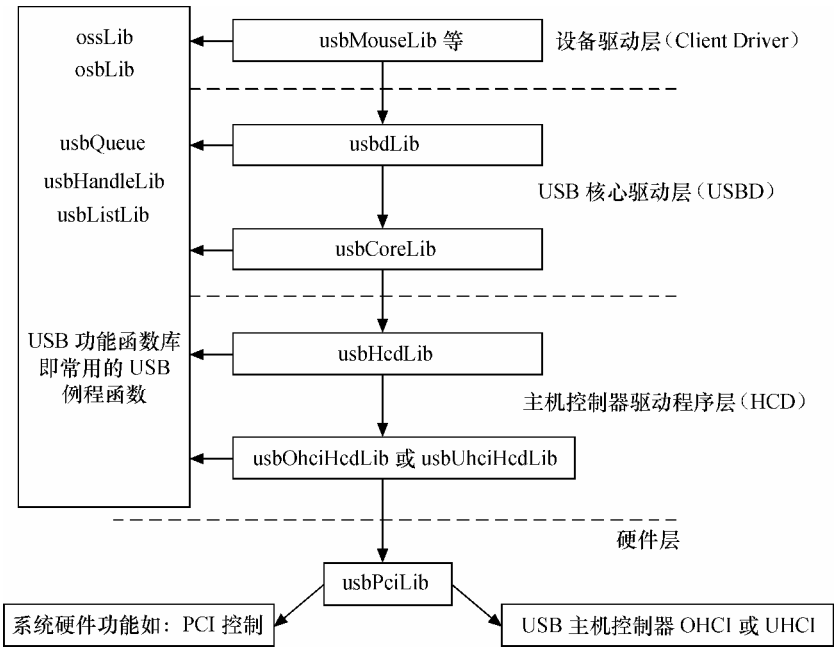

图 23.8　VxWorks 下的 USB 驱动层次

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