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 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。

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。

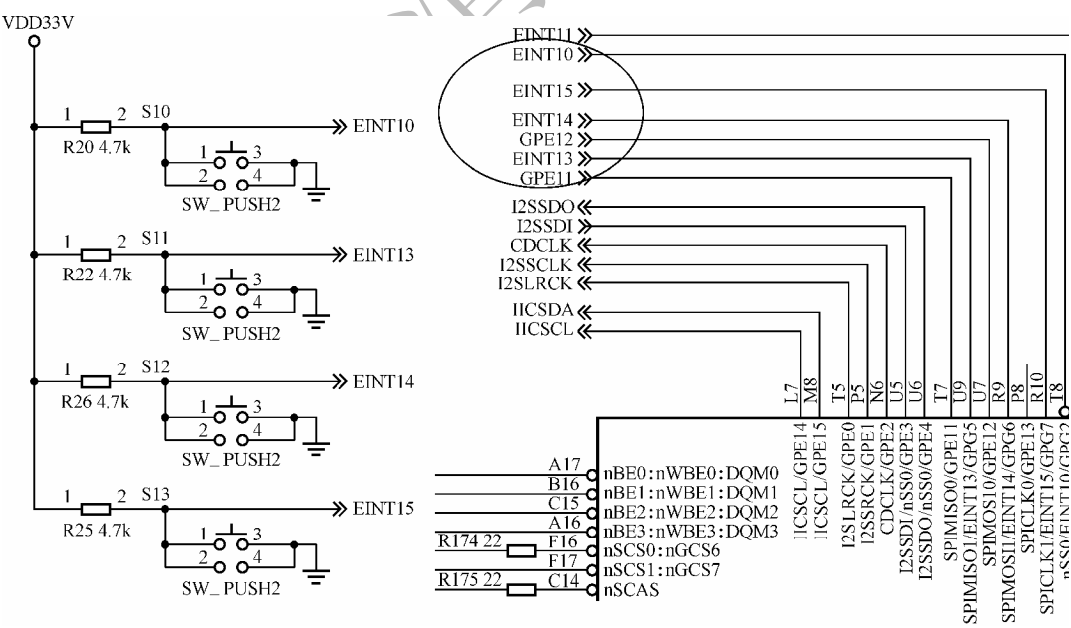

图 12.1 按键的硬件原理

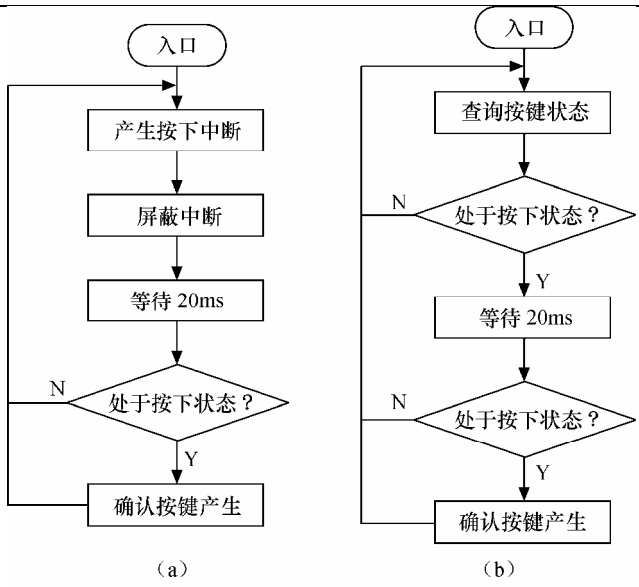

图 12.2　确认按键的流程

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

```
static void __exit s3c2410_key_exit(void)
{
  free_irqs(); //注销中断
  ...//释放设备号，删除 cdev
}
```

代码清单 12.6　按键设备驱动的中断申请函数

```
/*申请系统中断，中断方式为下降沿触发*/
static int request_irqs(void)
{
  struct key_info *k;
  int i;
  for (i = 0; i < sizeof(key_info_tab) / sizeof(key_info_tab[1]); i++)
  {
    k = key_info_tab + i;
    set_external_irq(k->irq_no, EXT_LOWLEVEL, GPIO_PULLUP_DIS);
                //设置低电平触发
        if (request_irq(k->irq_no, &buttons_irq, SA_INTERRUPT,
DEVICE_NAME,
      i)) //申请中断，将按键序号作为参数传入中断服务程序
    {
      return - 1;
    }
  }
  return 0;
}
```

代码清单 12.7　按键设备驱动的中断释放函数

```
/*释放中断*/
static void free_irqs(void)
{
  struct key_info *k;
  int i;
  for (i = 0; i < sizeof(key_info_tab) / sizeof(key_info_tab[1]); i++)
  {
    k = key_info_tab + i;
    free_irq(k->irq_no, buttons_irq); //释放中断
  }
}
```

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

```
static void s3c2410_eint_key(int irq, void *dev_id, struct pt_regs
*reg)
{
  int key = dev_id;
  disable_irq(key_info_tab[key].irq_no); //关中断，转入查询模式

  keydev.keyStatus[key] = KEYSTATUS_DOWNX;//状态为按下
  key_timer[key].expires == jiffies + KEY_TIMER_DELAY1;//延迟
  add_timer(&key_timer[key]); //启动定时器
}
```

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

```
1  static void key_timer_handler(unsigned long data)
2  {
3    int key = data;
4    if (ISKEY_DOWN(key))
5    {
6      if (keydev.keyStatus[key] == KEYSTATUS_DOWNX)
7      //从中断进入
8      {
9        keydev.keyStatus[key] = KEYSTATUS_DOWN;
10       key_timer[key].expires == jiffies + KEY_TIMER_DELAY; //延迟
11       keyEvent();   //记录键值，唤醒等待队列
12       add_timer(&key_timer[key]);
13     }
14     else
15     {
16       key_timer[key].expires == jiffies + KEY_TIMER_DELAY; //延迟
17       add_timer(&key_timer[key]);
18     }
19   }
20   else      //键已抬起
21   {
22     keydev.keyStatus[key] = KEYSTATUS_UP;
23     enable_irq(key_info_tab[key].irq_no);
24   }
25 }
```

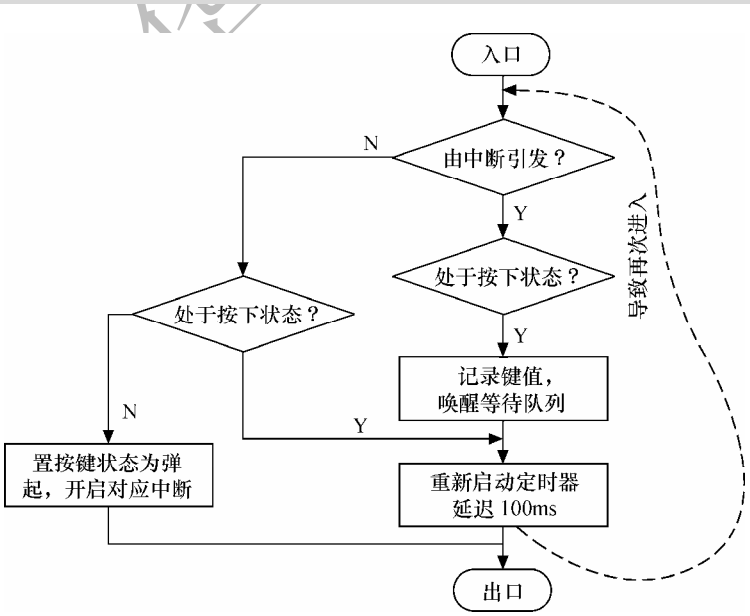

图 12.3　定时器处理函数流程

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

```
1  static int s3c2410_key_open(struct inode *inode, struct file *filp)
2  {
3    keydev.head = keydev.tail = 0; //清空按键动作缓冲区
4    keyEvent = keyEvent_raw; //函数指针指向按键处理函数 keyEvent_raw
5    return 0;
6  }
7
8  static int s3c2410_key_release(struct inode *inode, struct file
*filp)
9  {
10   keyEvent = keyEvent_dummy; //函数指针指向空函数
11   return 0;
12 }
```

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

```
1  static ssize_t s3c2410_key_read(struct file *filp, char *buf, ssize_t
count,
2    loff_t*ppos)
3  {
4    retry: if (keydev.head != keydev.tail)
5    //当前循环队列中有数据
6    {
7      key_ret = keyRead(); //读取按键
8      copy_to_user(..); //把数据从内核空间传送到用户空间
9    }
10   else
11   {
12     if (filp->f_flags &O_NONBLOCK)
13     //若用户采用非阻塞方式读取
14     {
15       return  - EAGAIN;
16     }
17     interruptible_sleep_on(&(keydev.wq));
18       //用户采用阻塞方式读取，调用该函数使进程睡眠
19     goto retry;
20   }
21   return 0;
22 }
```

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。

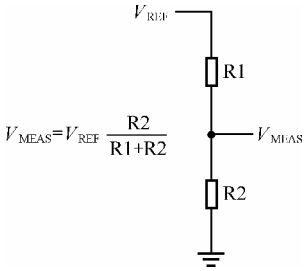

图 12.4　电阻触摸屏分压

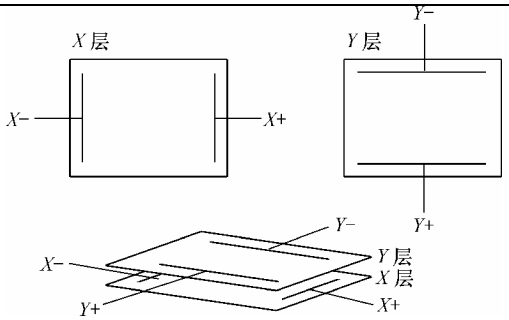

图 12.5　四线电阻式触摸屏

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。

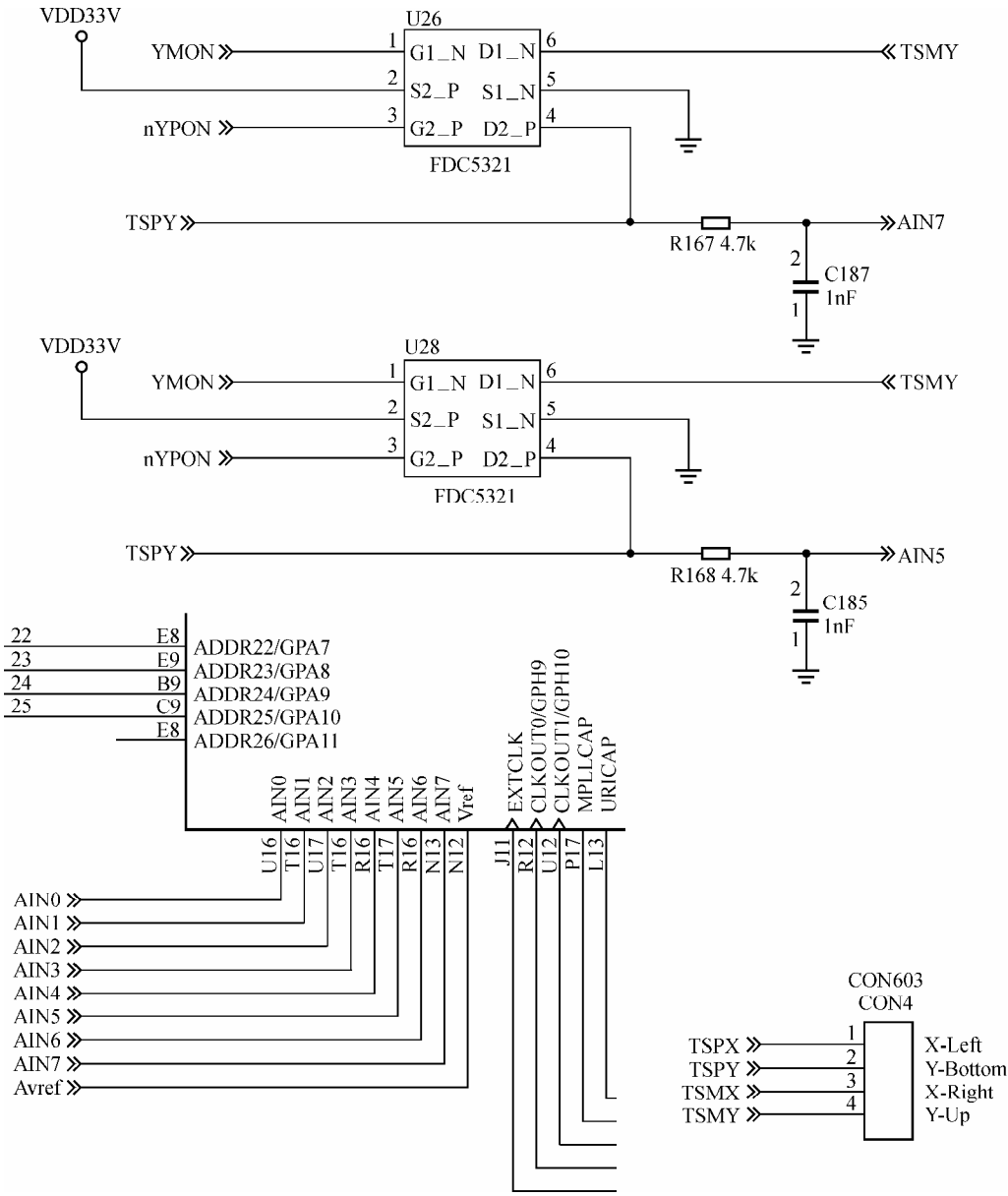

图 12.6　S3C2410 连接 4 线电阻式触摸屏

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

```
1  typedef struct
2  {
3    unsigned int penStatus; /* PEN_UP, PEN_DOWN, PEN_SAMPLE */
4    TS_RET buf[MAX_TS_BUF]; /* 缓冲区 */
5    unsigned int head, tail; /* 缓冲区头和尾 */
6    wait_queue_head_t wq; /*等待队列*/
7    spinlock_t lock;
8    #ifdef USE_ASYNC
9      struct fasync_struct *aq;
10   #endif
11   struct cdev cdev;
12 } 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

```
1 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

```
static void s3c2410_isr_tc(int irq, void *dev_id, struct pt_regs *reg)
{
  spin_lock_irq(&(tsdev.lock));
  if (tsdev.penStatus == PEN_UP)
  {
    start_ts_adc(); //开始 X/Y 位置转换
  }
  else
  {
    tsdev.penStatus = PEN_UP;
    DPRINTK("PEN UP: x: %08d, y: %08d\n", x, y);
    wait_down_int();//置于等待触点中断模式
    tsEvent();
  }
  spin_unlock_irq(&(tsdev.lock));
}
```

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

```
static void s3c2410_isr_adc(int irq, void *dev_id, struct pt_regs *reg)
{
  spin_lock_irq(&(tsdev.lock));
  if (tsdev.penStatus == PEN_UP)
    s3c2410_get_XY(); //读取坐标
  #ifdef HOOK_FOR_DRAG
    else
      s3c2410_get_XY();
  #endif
  spin_unlock_irq(&(tsdev.lock));
}
```

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

```
static inline void s3c2410_get_XY(void)
{
```

```
3   if (adc_state == 0)
4   {
5     adc_state = 1;
6     disable_ts_adc();   //禁止 INT-ADC
7     y = (ADCDAT0 &0x3ff); //读取坐标值
8     mode_y_axis();
9     start_adc_y();   //开始 y 位置转换
10  }
11  else if (adc_state == 1)
12  {
13    adc_state = 0;
14    disable_ts_adc(); //禁止 INT-ADC
15    x = (ADCDAT1 &0x3ff);  //读取坐标值
16    tsdev.penStatus = PEN_DOWN;
17    DPRINTK("PEN DOWN: x: %08d, y: %08d\n", x, y);
18    wait_up_int();   //置于等待抬起中断模式
19    tsEvent();
20  }
21 }
```

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

```
1  static void tsEvent_raw(void)
2  {
3    if (tsdev.penStatus == PEN_DOWN)
4    {
5      /*填充缓冲区*/
6      BUF_HEAD.x = x;
7      BUF_HEAD.y = y;
8      BUF_HEAD.pressure = PEN_DOWN;
9
10     #ifdef HOOK_FOR_DRAG
11       ts_timer.expires = jiffies + TS_TIMER_DELAY;
12       add_timer(&ts_timer);//启动定时器
13     #endif
14   }
15   else
16   {
17     #ifdef HOOK_FOR_DRAG
18       del_timer(&ts_timer);
19     #endif
20
21     /*填充缓冲区*/
22     BUF_HEAD.x = 0;
23     BUF_HEAD.y = 0;
24     BUF_HEAD.pressure = PEN_UP;
25   }
26
27   tsdev.head = INCBUF(tsdev.head, MAX_TS_BUF);
28   wake_up_interruptible(&(tsdev.wq));  //唤醒等待队列
29
```

```
30   #ifdef USE_ASYNC
31     if (tsdev.aq)
32       kill_fasync(&(tsdev.aq), SIGIO, POLL_IN);//异步通知
33   #endif
34 }
```

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。

代码清单 12.22　触摸屏设备驱动的定时器处理函数

```
#ifdef HOOK_FOR_DRAG
  static void ts_timer_handler(unsigned long data)
  {
    spin_lock_irq(&(tsdev.lock));
    if (tsdev.penStatus == PEN_DOWN)
    {
      start_ts_adc();  //开始 X/Y 位置转换
    }
    spin_unlock_irq(&(tsdev.lock));
  }
#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

```
static int s3c2410_ts_open(struct inode *inode, struct file *filp)
{
  tsdev.head = tsdev.tail = 0;
  tsdev.penStatus = PEN_UP;//初始化触摸屏状态为 PEN_UP
#ifdef HOOK_FOR_DRAG //如果定义了拖动钩子函数
  init_timer(&ts_timer);//初始化定时器
  ts_timer.function = ts_timer_handler;
#endif
  tsEvent = tsEvent_raw;
  init_waitqueue_head(&(tsdev.wq));//初始化等待队列

  return 0;
}
```

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

```
static int s3c2410_ts_release(struct inode *inode, struct file *filp)
{
  #ifdef HOOK_FOR_DRAG
    del_timer(&ts_timer);//删除定时器
  #endif
  return 0;
}
```

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)
```

```
3 {
4   return fasync_helper(fd, filp, mode, &(tsdev.aq));
5 }
6 #endif
```

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

```
1  /*在按键中断中报告事件*/
2  static void button_interrupt(int irq, void *dummy, struct pt_regs
*fp)
3  {
4    input_report_key(&button_dev, BTN_1, inb(BUTTON_PORT) &1);
5    input_sync(&button_dev);
6  }
7
8  static int __init button_init(void)
9  {
10   /*申请中断*/
11   if (request_irq(BUTTON_IRQ, button_interrupt, 0, "button", NULL))
12   {
13     printk(KERN_ERR "button.c: Can't allocate irq %d\n", button_irq);
14     return  - EBUSY;
15   }
16
17   button_dev.evbit[0] = BIT(EV_KEY);    //支持 EV_KEY 事件
18   button_dev.keybit[LONG(BTN_0)] = BIT(BTN_0);
19
20   input_register_device(&button_dev);   //注册 input 设备
21 }
22
23 static void __exit button_exit(void)
24 {
25   input_unregister_device(&button_dev);   //注销 input 设备
26   free_irq(BUTTON_IRQ, button_interrupt); //释放中断
27 }
```

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。

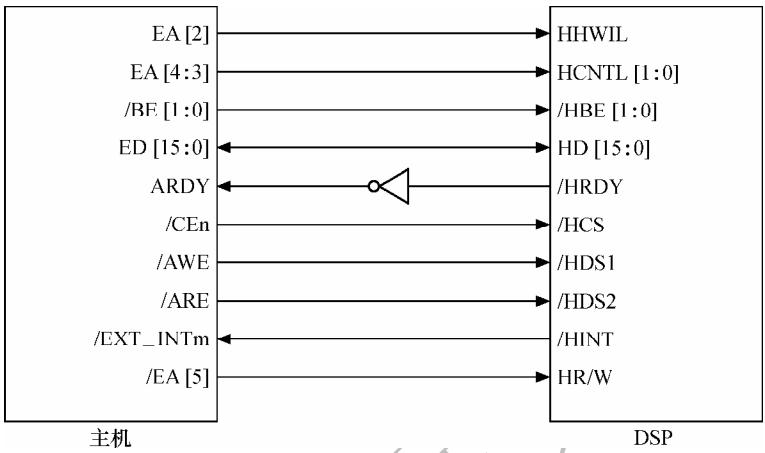

图 12.7　host 与 DSP HPI 连接

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

```
1 static struct miscdevice nvram_dev =
2 {
3  NVRAM_MINOR, //次设备号
4  "nvram",   //设备名
5  &nvram_fops   //文件操作结构体
6 };
```

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

```
1  struct file_operations nvram_fops =
2  {
3    .owner = THIS_MODULE,
4    .llseek = nvram_llseek,  //seek
5    .read = read_nvram,  //读
6    .write = write_nvram,//写
7    .ioctl = nvram_ioctl,//IO 控制
8  };
```

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

```
1  int __init nvram_init(void)
2  {
3   printk(KERN_INFO "Macintosh non-volatile memory driver v%s\n",
4       NVRAM_VERSION);
5   return misc_register(&nvram_dev);//注册 miscdevice
6  }
7
8  void __exit nvram_cleanup(void)
9  {
10   misc_deregister( &nvram_dev );//注销 miscdevice
11 }
```

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

```
1  /*读函数*/
2  static ssize_t read_nvram(struct file *file, char __user *buf, size_t
count,
3    loff_t *ppos)
4  {
5    unsigned int i;
6    char __user *p = buf;
7
8    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   {
```

```
15    unlock_kernel();
16    return  - EINVAL;
17  }
18  file->f_pos = offset;
19  unlock_kernel();
20  return file->f_pos;
21 }
```

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

```
1 void enable_watchdog()
2 {
3   rWTCON = WTCON_DIV64 | WTCON_RSTEN;//64 分频、开启复位信号
4   rWTDAT = 0x8000;//计数目标
5   rWTCON |= WTCON_ENABLE;//开启看门狗
6 }
```

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”

```
1 void feed_dog()
2 {
```

```
3   rWTCNT=0x8000;
4 }
```

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

```
1 void main()
2 {
3   init_system();
4   ...
5   enable_watchdog();//启动看门狗
6   ...
7   while(1)
8   {
9     ...
10    feed_dog(); //喂狗
11  }
12}
```

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

```
1 struct platform_device
2 {
3   const char  * name;//设备名
4   u32     id;
5   struct device dev;
6   u32     num_resources;//设备所使用各类资源数量
7   struct resource * resource;//资源
8 };
```

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

```
1  struct platform_device *s3c24xx_uart_devs[];
```

```
2
3  struct platform_device s3c_device_usb; //USB 控制器
4  struct platform_device s3c_device_lcd; //LCD 控制器
5  struct platform_device s3c_device_wdt; //看门狗
6  struct platform_device s3c_device_i2c; //I2C 控制器
7  struct platform_device s3c_device_iis; //IIS
8  struct platform_device s3c_device_rtc; //实时钟
9   ...
10 /*SMDK2410 开发板使用的平台设备*/
11 static struct platform_device *smdk2410_devices[]__initdata =
12 {
13   &s3c_device_usb, //USB
14   &s3c_device_lcd, //LCD
15   &s3c_device_wdt, //看门狗
16   &s3c_device_i2c, //I2C
17   &s3c_device_iis,  //IIS
18 };
```

代码清单 12.45　S3C2410 看门狗的 platform_device 结构体

```
1 struct platform_device s3c_device_wdt =
2 {
3   .name = "s3c2410-wdt",  //设备名
4   .id =  - 1, .
5   num_resources = ARRAY_SIZE(s3c_wdt_resource),  //资源数量
6   .resource = s3c_wdt_resource,  //看门狗所使用资源
7 };
```

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

```
1  int platform_add_devices(struct platform_device **devs, int num)
2  {
3    int i, ret = 0;
4
5    for (i = 0; i < num; i++)
6    {
```

```
    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 =
```

```
    2 {
    3   .owner      = THIS_MODULE,
    4   .llseek     = no_llseek,   //seek
    5   .write      = s3c2410wdt_write,   //写函数
    6   .ioctl      = s3c2410wdt_ioctl, //ioctl 函数
    7   .open       = s3c2410wdt_open,  //打开函数
    8   .release    = s3c2410wdt_release,//释放函数
    9 };
```

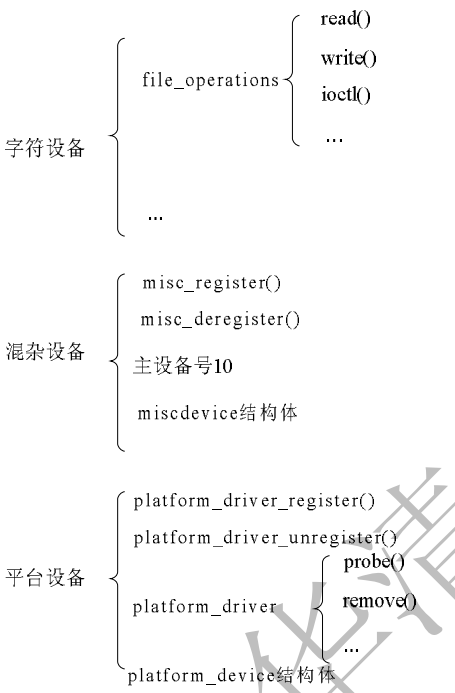

图 12.8　看门狗与字符设备、混杂设备和平台设备

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

```
    1  static int __init watchdog_init(void)
    2  {
    3   printk(banner);
    4   return   platform_driver_register(&s3c2410wdt_driver);//  注  册
platform_driver
    5  }
    6
    7  static void __exit watchdog_exit(void)
    8  {
```

```
    9   platform_driver_unregister(&s3c2410wdt_driver);//      注      销
platform_driver
    10 }
```

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

```
    1  static int s3c2410wdt_probe(struct platform_device *pdev)
    2  {
    3    struct resource *res;
    4    int started = 0;
    5    int ret;
    6    int size;
    7
    8    DBG("%s: probe=%p\n", __FUNCTION__, pdev);
    9
    10   /* 获得看门狗的内存区域 */
    11
    12   res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    13   if (res == NULL)
    14   {
    15     printk(KERN_INFO PFX "failed to get memory region resouce\n");
    16     return  - ENOENT;
    17   }
    18
    19   size = (res->end - res->start) + 1;
    20   //申请 I/O 内存
    21   wdt_mem = request_mem_region(res->start, size, pdev->name);
    22   if (wdt_mem == NULL)
    23   {
    24     printk(KERN_INFO PFX "failed to get memory region\n");
    25     return  - ENOENT;
    26   }
    27
    28   wdt_base = ioremap(res->start, size); //设备内存->虚拟地址
    29   if (wdt_base == 0)
    30   {
    31     printk(KERN_INFO PFX "failed to ioremap() region\n");
    32     return  - EINVAL;
    33   }
    34
    35   DBG("probe: mapped wdt_base=%p\n", wdt_base);
    36
    37   /* 获得看门狗的中断 */
    38
    39   res = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
    40   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

```
1  /*停止看门狗*/
2  static int s3c2410wdt_stop(void)
3  {
4    unsigned long wtcon;
5
6    wtcon = readl(wdt_base + S3C2410_WTCON);
7    //停止看门狗，禁止复位
8    wtcon &= ~(S3C2410_WTCON_ENABLE | S3C2410_WTCON_RSTEN);
9    writel(wtcon, wdt_base + S3C2410_WTCON);
10
11   return 0;
12 }
13
14 /*开启看门狗*/
15 static int s3c2410wdt_start(void)
16 {
17   unsigned long wtcon;
18
19   s3c2410wdt_stop();
20
21   wtcon = readl(wdt_base + S3C2410_WTCON);
22   //使能看门狗，128 分频
23   wtcon |= S3C2410_WTCON_ENABLE | S3C2410_WTCON_DIV128;
24
25   if (soft_noboot)
26   {
27     wtcon |= S3C2410_WTCON_INTEN; //使能中断
28     wtcon &= ~S3C2410_WTCON_RSTEN; //禁止复位
29   }
30   else
31   {
32     wtcon &= ~S3C2410_WTCON_INTEN; //禁止中断
33     wtcon |= S3C2410_WTCON_RSTEN; //使能复位
34   }
35
36   DBG("%s: wdt_count=0x%08x, wtcon=%08lx\n", __FUNCTION__,
37      wdt_count, wtcon);
38   writel(wdt_count, wdt_base + S3C2410_WTDAT);
39   writel(wdt_count, wdt_base + S3C2410_WTCNT);
40   writel(wtcon, wdt_base + S3C2410_WTCON);
41
42   return 0;
43 }
```

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

```
1  static ssize_t s3c2410wdt_write(struct file *file, const char __user
*data,
2    size_t len, loff_t *ppos)
3  {
4    /* 刷新看门狗 */
5    if (len)
6    {
7      if (!nowayout)
```

```
8     {
9       size_t i;
10      allow_close = CLOSE_STATE_NOT;
11      for (i = 0; i != len; i++)
12      {
13        char c;
14        if (get_user(c, data + i))//用户空间->内核空间
15          return  - EFAULT;
16        if (c == 'V')  //如果写入了'V' ，允许关闭
17          allow_close = CLOSE_STATE_ALLOW;
18      }
19    }
20    s3c2410wdt_keepalive();
21  }
22  return len;
23 }
```

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。

- 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