## 1.Linux内核模块

设备驱动程序是操作系统和机器硬件之间的接口，它为应用程序屏蔽了硬件的细节，这样在应用程序看来，硬件设备只是一个设备文件，应用程序可以和操作普通文件一样，只需要进行文件的打开，读写和关闭等操作，就能控制底层的具体硬件设备。

Linux内核的一个模块可以以两种方式被编译和加载：

（1）直接编译进Linux内核，随同Linux启动时加载；

（2）编译成一个可加载和删除的模块，使用insmod加载（modprobe和insmod命令类似，但依赖于相关的配置文件），rmmod删除。这种方式控制了内核的大小，而模块一旦被插入内核，它就和内核其他部分一样。

下面我们给出一个内核模块的例子：

#include <linux/module.h> //所有模块都需要的头文件

#include <linux/init.h> // init&exit相关宏

MODULE\_LICENSE("GPL"); // 用于声明模块的许可证。

static int \_\_init hello\_init (void)

{

printk("Hello module init\n");

return 0;

}

static void \_\_exit hello\_exit (void)

{

printk("Hello module exit\n");

}

module\_init(hello\_init);

module\_exit(hello\_exit);

分析上述程序，发现一个Linux内核模块需包含模块初始化和模块卸载函数，前者在insmod的时候运行，后者在rmmod的时候运行。初始化与卸载函数必须在宏module\_init和module\_exit使用前定义，否则会出现编译错误。

如果要把上述程序编译为一个运行时加载和删除的模块，则编译命令为：

gcc –D\_\_KERNEL\_\_ -DMODULE –DLINUX –I /usr/local/src/linux2.4/include -c –o hello.o hello.c

由此可见，Linux内核模块的编译需要给gcc指示–D\_\_KERNEL\_\_ -DMODULE –DLINUX参数。-I选项跟着Linux内核源代码中Include目录的路径。

下列命令将可加载hello模块：

insmod ./hello.o

下列命令完成相反过程：

rmmod hello

如果要将其直接编译入Linux内核，则需要将源代码文件拷贝入Linux内核源代码的相应路径里，并修改Makefile。

我们有必要补充一下Linux内核编程的一些基本知识：

内存

在Linux内核模式下，我们不能使用用户态的malloc()和free()函数申请和释放内存。进行内核编程时，最常用的内存申请和释放函数为在include/linux/kernel.h文件中声明的kmalloc()和kfree()，其原型为：

void \*kmalloc(unsigned int len, int priority);

void kfree(void \*\_\_ptr);

kmalloc的priority参数通常设置为GFP\_KERNEL，如果在中断服务程序里申请内存则要用GFP\_ATOMIC参数，因为使用GFP\_KERNEL参数可能会引起睡眠，不能用于非进程上下文中（在中断中是不允许睡眠的）。

由于内核态和用户态使用不同的内存定义，所以二者之间不能直接访问对方的内存。而应该使用Linux中的用户和内核态内存交互函数（这些函数在include/asm/uaccess.h中被声明）：

unsigned long copy\_from\_user(void \*to, const void \*from, unsigned long n);

unsigned long copy\_to\_user (void \* to, void \* from, unsigned long len);

copy\_from\_user、copy\_to\_user函数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。

include/asm/uaccess.h中定义的put\_user和get\_user用于内核空间和用户空间的单值交互（如char、int、long）。

输出

在内核编程中，我们不能使用用户态C库函数中的printf()函数输出信息，而只能使用printk()。但是，内核中printk()函数的设计目的并不是为了和用户交流，它实际上是内核的一种日志机制，用来记录下日志信息或者给出警告提示。

每个printk都会有个优先级，内核一共有8个优先级，它们都有对应的宏定义。如果未指定优先级，内核会选择默认的优先级DEFAULT\_MESSAGE\_LOGLEVEL。如果优先级数字比int console\_loglevel变量小的话，消息就会打印到控制台上。如果syslogd和klogd守护进程在运行的话，则不管是否向控制台输出，消息都会被追加进/var/log/messages文件。klogd 只处理内核消息，syslogd 处理其他系统消息，比如应用程序。

模块参数

2.6内核下，include/linux/module.h中定义的宏MODULE\_PARM(var,type) 用于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：byte；h：short；i：int；l：long；s：string。

在装载内核模块时，用户可以向模块传递一些参数：

insmod modname var=value

如果用户未指定参数，var将使用模块内定义的缺省值。

## 1.x Linux内核同步机制——信号量

在操作系统中，信号量用来限制多个进程在同一时间访问某个资源。一个信号量实质上是一个整数值，它和一对函数联合使用，这一对函数通常称为P和V。每个线程中访问临界资源的那段代码被称为临界区(critical section)。即将进入临界区的进程将在相关信号量上调用P；如果信号量的值大于零，则该值会减小1，而进程可以继续。相反，如果信号量的值为0（或更小），进程必须等待直到其他进程释放该信号量。对信号量的解锁通过调用V完成；该函数增加信号量的值，并在必要时唤醒等待的进程。

在Linux中要使用信号量，内核代码中必须包括*<asm/semaphore.h>*。相关的数据结构为：

struct semaphore {

atomic\_t count;

int sleepers;

wait\_queue\_head\_t wait;

};

在Linux中，P函数被称为down，指的是该函数减小了信号量的值，它也许会将调用者置于休眠状态，然后等待信号量变得可用，之后授予调用者对被保护资源的访问。

static inline void down(struct semaphore \* sem)

当一个线程成功调用down后，就称为该线程“拥有”了该信号量。这样，该线程就被赋予了访问由该信号量保护的临界区的权利。当互斥操作完成后，必须返回该信号量。Linux下等价于V的函数是up：

static inline void up(struct semaphore \* sem)

调用up后，调用者不再拥有该信号量。任何拿到信号量的线程都必须通过一次且只有一次对up的调用而释放该信号量。在出现错误的情况下要特别注意，如果在拥有一个信号量时发生错误，必须在出现错误状态返回给调用者之前释放该信号量。我们很容易犯忘记释放信号量的错误，而其结果（进程在某些无关位置处被挂起）是很难复现和跟踪的。

在我们的SPI驱动中，使用全局变量来定义信号量：

static struct semaphore spi\_lock; /\* protect access to SPI bus \*/

信号量必须在设备对系统其他部分可用前被初始化，否则会引起竟态的发生，即在信号量准备好之前，代码有可能访问它们。然后确保代码在不拥有该信号量的时候不会访问SPI相关数据结构。具体实现为，在调用spi\_access\_bus的时候包含如下代码：

down(&spi\_lock);

在释放总线前，应释放信号量，即在spi\_release\_bus中释放信号量：

up(&spi\_lock);

## 2.字符设备驱动程序

Linux下的设备驱动程序被组织为一组完成不同任务的函数的集合，通过这些函数使得Windows的设备操作犹如文件一般。在应用程序看来，硬件设备只是一个设备文件，应用程序可以像操作普通文件一样对硬件设备进行操作，如open()、close()、read()、write() 等。

Linux主要将设备分为三类：字符设备，块设备和网络接口[[1]](#endnote-1)。字符设备是指设备发送和接收数据以字符的形式进行；而块设备则以整个数据缓冲区的形式进行。字符设备的驱动相对比较简单。网络接口是一个能够与其他主机交换数据的设备，通常一个接口是一个硬件设备，但是它也可能是一个纯粹的网络设备。

下面我们来假设一个非常简单的虚拟字符设备：这个设备中只有一个4个字节的全局变量int global\_var，而这个设备的名字叫做“gobalvar”。对“gobalvar”设备的读写等操作即是对其中全局变量global\_var的操作。

file\_operations将每个打开的文件和一组函数关联起来，这些函数主要是用来实现系统调用。随着内核不断增加新的功能，file\_operations结构体已逐渐变得越来越大，但是大多数的驱动程序只是利用了其中的一部分。对于字符设备来说，要提供的主要入口有：open()、release()、read()、write()、ioctl()、llseek()、poll()等。

**open()函数**　对设备特殊文件进行open()系统调用时，将调用驱动程序的open() 函数：

int (\*open)(struct inode \*, struct file \*);

其中参数inode为设备特殊文件的inode (索引结点) 结构的指针，参数file是指向这一设备的文件结构的指针。open()的主要任务是确定硬件处在就绪状态、验证次设备号的合法性(次设备号可以用MINOR(inode-> i - rdev) 取得)、控制使用设备的进程数、根据执行情况返回状态码(0表示成功，负数表示存在错误) 等；

**release()函数**　当最后一个打开设备的用户进程执行close()系统调用时，内核将调用驱动程序的release()函数：

void (\*release) (struct inode \* ,struct file \*);

release 函数的主要任务是清理未结束的输入/输出操作、释放资源、用户自定义排他标志的复位等。

**read()函数**　当对设备特殊文件进行read()系统调用时，将调用驱动程序read()函数：

ssize\_t (\*read) (struct file \*, char \*, size\_t, loff\_t \*);

用来从设备中读取数据。当该函数指针被赋为NULL 值时，将导致read 系统调用出错并返回-EINVAL（“Invalid argument，非法参数”）。函数返回非负值表示成功读取的字节数（返回值为“signed size”数据类型，通常就是目标平台上的固有整数类型）。

指针参数 file为进行读取信息的目标文件；指针参数char为对应放置信息的缓冲区（即用户空间内存地址），参数size\_t为要读取的信息长度，参数loff\_t 为读的位置相对于文件开头的偏移，在读取信息后，这个指针一般都会移动，移动的值为要读取信息的长度值。

**write( )函数**　当设备特殊文件进行write()系统调用时，将调用驱动程序的write()函数：

ssize\_t (\*write) (struct file \*, const char \*, size\_t, loff\_t \*);

向设备发送数据。如果没有这个函数，write 系统调用会向调用程序返回一个-EINVAL。如果返回值非负，则表示成功写入的字节数。

## 3.SPI工作原理

SPI的全称是“Serial Peripheral Interface（串行外围接口）”，用来在微控制器和外围设备芯片直接提供一个低成本，易使用的接口。

与标准的串行接口不同，SPI是一个同步协议接口，所有的传输都参照一个共同的时钟，这个同步时钟信号由主机（处理器）产生，接收数据的外设（从设备）使用时钟对串行比特流的接收进行同步化。有时会有许多芯片连到主机的同一个SPI接口上，这时主机通过触发从设备的片选输入引脚来选择接收数据的从设备，没有被选中的外设将不会参与SPI传输。

SPI接口是在CPU和外围低速器件之间进行同步串行数据传输，在主器件的移位脉冲下，数据按位传输，高位在前，低位在后，为全双工通信，数据传输速度总体来说比I2C总线快。SPI通信与串口通信相比，它不需要固定的波特率，其工作时序均与时钟信号同步，所以比串口更灵活。

SPI接口以主从方式进行工作，其接口包括4种信号：

（1）MOSI———主器件数据输出，从器件数据输入；

（2）MISO———主器件数据输入，从器件数据输出；

（3）SCLK———时钟信号，由主器件产生；

（4）SS———从器件使能信号，由主器件控制。

要特别注意的是，SCLK信号线只由主设备控制，从设备不能控制信号线。同样，在一个基于SPI的设备中，至少有一个主控设备。此传输方式与普通的串行通讯不同，普通的串行通讯一次连续传送至少8位数据，而SPI允许1位数据传送，甚至允许暂停，因为SCK时钟线由主控设备控制，当没有时钟跳变时，从设备不采集或传送数据。也就是说，主设备通过对

SCK时钟线的控制可以完成对通讯的控制。同时，SPI也是一个数据交换协议： 因为SPI的数据输入和输出线独立，所以允许同时完成数据的输入和输出。

不同的SPI设备的实现方式不尽相同，主要是数据改变和采集的时间不同， 在时钟信号上沿或下沿采集有不同定义。[[2]](#endnote-2)

## 4. 基于AT91RM9200的SPI分析

### 4.1 AT91RM9200接口的内部电路

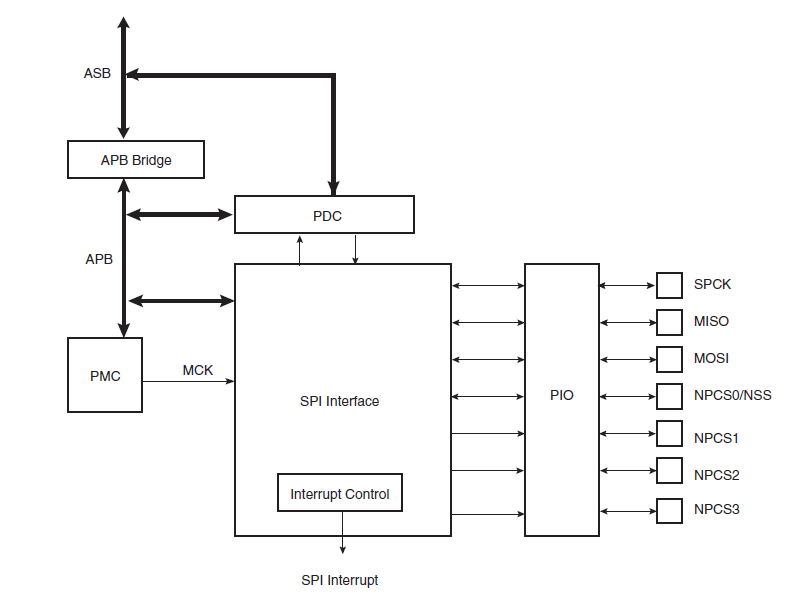


图4-1 AT91RM9200 SPI方框图

详见AT91RM9200数据手册

## 5. SPI设备驱动程序详细分析

Linux下设备驱动的开发就是编写各种函数实现设备的注册，初始化，I/O操作和卸载等功能。

### 5.1 代码结构概述

本驱动主要由at91\_spi.c和spi\_dev.c组成。

at91\_spi.c实现了SPI设备驱动的加载和卸载，并向spi\_dev.c文件导出3个函数：spi\_access\_bus()函数用于使能SPI总线，完成SPI传输前的准备工作；spi\_transfer()函数实现SPI传输；spi\_release\_bus()函数释放SPI总线。

EXPORT\_SYMBOL(spi\_access\_bus);

EXPORT\_SYMBOL(spi\_release\_bus);

EXPORT\_SYMBOL(spi\_transfer);

spi\_dev.c文件实现了SPI设备的注册和释放，以及SPI设备文件接口函数。[[3]](#endnote-3)

### 5.2 at91\_spi.c文件详解

#### 5.2.1 设备初始化函数

static int \_\_init at91\_spi\_init(void)是设备初始化函数，用于完成对PIO口进行初始化设置，对SPI控制器进行软件重置，设置SPI的工作方式、各控制寄存器和模式寄存器，设置中断，最后使能SPI。

初始化工作首先要对AT91RM9200的PIO控制器进行编程，将PA0~PA3设置成SPI控制模式分配给外设，其实现函数为include/asm-arm/arch-at91rm9200/pio.h/static inline void AT91\_CfgPIO\_SPI(void),函数原型为：

static inline void AT91\_CfgPIO\_SPI(void) {

AT91\_SYS->PIOA\_PDR = AT91C\_PA0\_MISO | AT91C\_PA1\_MOSI |AT91C\_PA2\_SPCK;

}

然后执行SPI接口的软件触发硬件复位：

controller->SPI\_CR = AT91C\_SPI\_SWRST;

设置SPI的工作方式主要包括对控制寄存器(SPI\_CR)，模式寄存器(SPI\_MR)和片选寄存器(SPI\_CSR)的配置，需要定义一个指向SPI寄存器的结构体变量controller来调用这些寄存器。由于在Linux下是不允许对设备的物理地址进行直接访问的，必须转换为虚拟地址后才能访问，则定义变量static AT91PS\_SPI controller = (AT91PS\_SPI) AT91C\_VA\_BASE\_SPI;其中AT91PS\_SPI是在内核/include/asm-arm/arch-at91rm9200/AT91RM9200\_SPI.h中定义的，用于指向SPI操作的所有寄存器。而AT91C\_VA\_BASE\_SPI是在/include/asm-arm/arch-at91rm9200/hardware.h中定义的SPI寄存器的虚拟地址，它通过同一文件下定义的AT91\_IO\_P2V宏来完成物理地址到虚拟地址的映射，代码如下：

#define AT91C\_VA\_BASE\_SPI AT91\_IO\_P2V(AT91C\_BASE\_SPI)

/\* Convert a physical IO address to virtual IO address \*/

#define AT91\_IO\_P2V(x) ((x) - AT91C\_IO\_PHYS\_BASE + AT91C\_IO\_VIRT\_BASE)

利用(AT91PS\_SPI) AT91C\_VA\_BASE\_SPI进行强制类型转换引用该虚拟地址，再定义controller即可访问AT91PS\_SPI中的成员了。初始化配置如下：

controller->SPI\_CR = AT91C\_SPI\_SWRST;// 执行SPI接口的软件触发硬件复位

controller->SPI\_CSR0 = AT91C\_SPI\_BITS\_8 | (1 << 16) | (60 << 8);

//设置SPI的片选寄存器，根据cs5532的数据手册，将此寄存器设置为SPCK无效状态值为逻辑0，数据在SPCK起始边沿改变，在SPCK下一个边沿捕获，每次传输8位数据。设置波特率，SPCK前延时和连续传输间延迟。

controller->SPI\_PTCR = AT91C\_PDC\_RXTDIS | AT91C\_PDC\_TXTDIS;

//设置PDC传输控制寄存器，接收器传输禁用，发送器传输禁用

memset(&spi\_dev, 0, **sizeof**(spi\_dev));

spi\_dev[0].pcs = 0xE;

spi\_dev[1].pcs = 0xD;

spi\_dev[2].pcs = 0xB;

spi\_dev[3].pcs = 0x7;

//初始化结构体spi\_dev,其类型为struct spi\_local,在/include/asm-arm/arch-at91rm9200/at91\_spi.h中定义，具体如下：

struct spi\_local {

unsigned int pcs; /\* Peripheral Chip Select value \*/

short pio\_enabled; /\* has PIO been enabled? \*/

struct spi\_transfer\_list \*xfers; /\* current transfer list \*/

dma\_addr\_t tx, rx; /\* DMA address for current transfer \*/

dma\_addr\_t txnext, rxnext; /\* DMA address for next transfer \*/

};

if (request\_irq(AT91C\_ID\_SPI, spi\_interrupt, 0, "dudu\_spi", NULL)){

printk(KERN\_INFO "I am Busy.\n");

return -EBUSY;

}

//request\_irq()函数用来注册中断函数，在2.6的内核中，需要包含的头文件是#include <linux/interrupt.h>,函数原型为

request\_irq(unsigned int irq, irq\_handler\_t handler, unsigned long flags, const char \*name, void \*dev);

在发生对应于第 1个参数 irq 的中断时，则调用第 2 个参数 handler 指定的中断服务函数(也就是把 handler() 中断服务函数注册到内核中 )。第 3 个参数 flags 指定了快速中断或中断共享等中断处理属性，这是一个与中断管理有关的位掩码选项。第4个参数\*name是传递给request\_irq的字符串，用来在/proc/interrupts中显示中断的拥有者，或内核发生中断错误时使用。第 5 个参数 dev\_id 可作为共享中断时的中断区别参数，也可以用来指定中断服务函数需要参考的数据地址。函数运行正常时返回 0 ，否则返回对应错误的负值。

AT91C\_ID\_SPI是要申请的中断号，定义在/include/asm-arm/arch-at91rm9200/AT91RM9200\_SPI.h中：

#define AT91C\_ID\_SPI (13) // Serial Peripheral Interface

函数spi\_interrupt则在本文件中定义：

static irqreturn\_t spi\_interrupt(int irq, void \*dev\_id, struct pt\_regs \*regs)；

最后，通过设置SPI控制寄存器来使能SPI：

controller->SPI\_CR = AT91C\_SPI\_SPIEN;

#### 5.2.2 设备卸载函数

static void \_\_exit at91\_spi\_exit(void)是设备卸载函数，用于释放中断，禁止SPI总线传输。

static void at91\_spi\_exit(void)

{

controller->SPI\_CR = AT91C\_SPI\_SPIDIS; /\* Disable SPI \*/

free\_irq(AT91C\_ID\_SPI, 0);

}

#### 5.2.3 中断处理函数

/\*

\* Handle interrupts from the SPI controller.

\*/

**static** irqreturn\_t **spi\_interrupt**(**int** irq, **void** \*dev\_id, **struct** pt\_regs \*regs)

{

**unsigned** **int** status;

**struct** spi\_local \*device = (**struct** spi\_local \*) &spi\_dev[current\_device];

**struct** spi\_transfer\_list \*list = device->xfers;

**#ifdef** DEBUG\_SPI

printk(KERN\_INFO "[DEBUG INFO] : SPI interrupt %i\n", current\_device);

**#endif**

**if** (!list)

panic("at91\_spi: spi\_interrupt with a NULL transfer list");

status = controller->SPI\_SR & controller->SPI\_IMR; /\* read status \*/

pci\_unmap\_single(NULL, device->tx, list->txlen[list->curr], PCI\_DMA\_TODEVICE);

pci\_unmap\_single(NULL, device->rx, list->rxlen[list->curr], PCI\_DMA\_FROMDEVICE);

device->tx = device->txnext; /\* move next transfer to current transfer \*/

device->rx = device->rxnext;

list->curr = list->curr + 1;

**if** (list->curr == list->nr\_transfers) { /\* all transfers complete \*/

controller->SPI\_IDR = AT91C\_SPI\_SPENDRX; /\* disable interrupt \*/

/\* Disable transmitter and receiver \*/

controller->SPI\_PTCR = AT91C\_PDC\_RXTDIS | AT91C\_PDC\_TXTDIS;

device->xfers = NULL;

complete(&transfer\_complete);

}

**else** **if** (list->curr+1 == list->nr\_transfers) { /\* no more next transfers \*/

device->txnext = 0;

device->rxnext = 0;

controller->SPI\_TNCR = 0;

controller->SPI\_RNCR = 0;

}

**else** {

**int** i = (list->curr)+1;

device->txnext = pci\_map\_single(NULL, list->tx[i], list->txlen[i], PCI\_DMA\_TODEVICE);

device->rxnext = pci\_map\_single(NULL, list->rx[i], list->rxlen[i], PCI\_DMA\_FROMDEVICE);

controller->SPI\_TNPR = device->txnext;

controller->SPI\_RNPR = device->rxnext;

controller->SPI\_TNCR = list->txlen[i];

controller->SPI\_RNCR = list->rxlen[i];

}

**return** IRQ\_HANDLED;

}

#### 5.2.4 模块的加载与卸载

设备初始化和卸载的时候分别调用module\_init(at91\_spi\_init)和module(at91\_spi\_exit)，前者是供insmod在加载此模块的时候自动调用，负责进行设备驱动程序的初始化工作。module\_init返回0表示初始化成功，返回负数表示失败。后者则是模块被卸载时调用，负责进行设备驱动程序的清除工作。

#### 5.2.5 at91\_spi.c的导出函数

这份文件的导出函数有三个，分别用于获取和释放SPI总线，实现SPI的传输。这三个函数须在/include/asm-arm/arch-at91rm9200/at91\_spi.h文件中声明出来：

/\* Exported functions \*/

extern void spi\_access\_bus(short device);

extern void spi\_release\_bus(short device);

extern int spi\_transfer(struct spi\_transfer\_list\* list);

并且在spi\_dev.c中要把这个头文件at91\_spi.h包含进去，这样这三个函数就可以被调用了。

在spi\_dev.c这份文件开头处首先定义几个全局变量：

**static** **struct** spi\_local spi\_dev[NR\_SPI\_DEVICES]; /\* SPI设备状态相关信息 \*/

**static** **int** spi\_enabled = 0; /\* 对当前正在使用的所有SPI设备进行计数 \*/

**static** **struct** semaphore spi\_lock; /\* 用于保护对SPI的访问 \*/

**static** **int** current\_device = -1; /\* 当前所选择的SPI编号 \*/

##### 5.2.6.1 获取SPI总线

必须在SPI传输开始之前获取SPI资源并配置相关寄存器。相关函数为

void spi\_access\_bus(short device)

{

if ((device < 0) || (device >= NR\_SPI\_DEVICES))

panic("at91\_spi: spi\_access\_bus called with invalid device");

//确保该设备号有效，其设备号应该大于等于0，小于NR\_SPI\_DEVICES，由于at91rm9200共有4根片选线，故NR\_SPI\_DEVICES定义为4，该宏定义在内核文件/include/asm-arm/arch-at91rm9200/at91\_spi.h中：

#define NR\_SPI\_DEVICES 4 /\* number of devices on SPI bus \*/

if (spi\_enabled == 0) {

AT91\_SYS->PMC\_PCER = 1 << AT91C\_ID\_SPI; /\* Enable Peripheral clock \*/

controller->SPI\_CR = AT91C\_SPI\_SPIEN; /\* Enable SPI \*/

controller->SPI\_MR = AT91C\_SPI\_DIV32; /\* The SPI operates at MCK/32 \*/

#ifdef DEBUG\_SPI

printk(KERN\_INFO "[DEBUG INFO] : SPI On\n");

#endif /\* DEBUG\_SPI \*/

}

spi\_enabled++;

// 若是第一次使用SPI总线，则需要对其进行初始化，包括初始化时钟，使能SPI和设置时钟分频。

// 对PMC\_PCER的定义是在内核文件/include/asm-at91rm9200/AT91RM9200\_SYS.h中，其全称为Peripheral Clock Enable Register，外设时钟使能寄存器，用于使能其对应的外设时钟，详见AT91RM9200技术手册280页。

// 在Linux启动时，可以看到其Master clock为60,204,503Hz，而CS5532所支持的最大串口时钟频率为2MHz（CS5532技术手册第10页），因此，需将SPI的时钟设置为对主时钟进行1/32分频。

/\* Lock the SPI bus \*/

down(&spi\_lock); //获得信号量spi\_lock

current\_device = device;//为当前设备号赋值

/\* Enable PIO \*/

if (!spi\_dev[device].pio\_enabled) {

switch (device) {

case 0: AT91\_CfgPIO\_SPI\_CS0(); break;

case 1: AT91\_CfgPIO\_SPI\_CS1(); break;

case 2: AT91\_CfgPIO\_SPI\_CS2(); break;

case 3: AT91\_CfgPIO\_SPI\_CS3(); break;

}

spi\_dev[device].pio\_enabled = 1;

}

//检测该设备编号device是否曾被激活过。若是第一次使用，则需根据此device取值来配置当前设备所对应的片选管脚。通过调用AT91\_CfgPIO\_SPI\_CS#()来实现。例如，当device = 0时，调用AT91\_CfgPIO\_SPI\_CS0()，其原型定义在/include/asm-arm/arch-at91rm9200/pio.h中：

**static** **inline** **void** **AT91\_CfgPIO\_SPI\_CS0**(**void**) {

AT91\_SYS->PIOA\_PDR = AT91C\_PA3\_NPCS0;

}

controller->SPI\_MR = AT91C\_SPI\_MSTR | AT91C\_SPI\_MODFDIS | (spi\_dev[device].pcs << 16);

//配置SPI模式寄存器为主机模式，模式错误检测禁用及外设片选。

至此，SPI设备已准备好进行数据传输。

##### 5.2.6.2 释放SPI总线

当SPI传输结束后，需要释放SPI总线及其相关资源。相关函数如下：

void spi\_release\_bus(short device)

{

if (device != current\_device)

panic("at91\_spi: spi\_release called with invalid device");

//首先检验传入的SPI编号所对应的设备是否为当前正在被调用的设备

current\_device = -1; //释放设备

up(&spi\_lock); //释放信号量

spi\_enabled--; //打开的设备总数减一

if (spi\_enabled == 0) {

controller->SPI\_CR = AT91C\_SPI\_SPIDIS; /\* Disable SPI \*/

AT91\_SYS->PMC\_PCER = 1 << AT91C\_ID\_SPI; /\* Disable Peripheral clock \*/

}

// 若当前已经没有打开的SPI设备，则关闭SPI总线，关闭SPI外设时钟

}

##### 5.2.6.3 执行传输

/\*

\* Perform a data transfer over the SPI bus

\*/

int spi\_transfer(struct spi\_transfer\_list\* list)

{

struct spi\_local \*device = (struct spi\_local \*) &spi\_dev[current\_device];

if (!list)

panic("at91\_spi: spi\_transfer called with NULL transfer list");

if (current\_device == -1)

panic("at91\_spi: spi\_transfer called without acquiring bus");

#ifdef DEBUG\_SPI

printk(KERN\_INFO "[DEBUG INFO] : SPI transfer start [%i]\n", list->nr\_transfers);

#endif

/\* Store transfer list \*/

device->xfers = list;

list->curr = 0;

/\* Assume there must be at least one transfer \*/

device->tx = pci\_map\_single(NULL, list->tx[0], list->txlen[0], PCI\_DMA\_TODEVICE);

device->rx = pci\_map\_single(NULL, list->rx[0], list->rxlen[0], PCI\_DMA\_FROMDEVICE);

/\* Program PDC registers \*/

controller->SPI\_TPR = device->tx;

controller->SPI\_RPR = device->rx;

controller->SPI\_TCR = list->txlen[0];

controller->SPI\_RCR = list->rxlen[0];

/\* Is there a second transfer? \*/

if (list->nr\_transfers > 1) {

device->txnext = pci\_map\_single(NULL, list->tx[1], list->txlen[1], PCI\_DMA\_TODEVICE);

device->rxnext = pci\_map\_single(NULL, list->rx[1], list->rxlen[1], PCI\_DMA\_FROMDEVICE);

/\* Program Next PDC registers \*/

controller->SPI\_TNPR = device->txnext;

controller->SPI\_RNPR = device->rxnext;

controller->SPI\_TNCR = list->txlen[1];

controller->SPI\_RNCR = list->rxlen[1];

}

else {

device->txnext = 0;

device->rxnext = 0;

controller->SPI\_TNCR = 0;

controller->SPI\_RNCR = 0;

}

// TODO: If we are doing consecutive transfers (at high speed, or

// small buffers), then it might be worth modifying the 'Delay between

// Consecutive Transfers' in the CSR registers.

// This is an issue if we cannot chain the next buffer fast enough

// in the interrupt handler.

/\* Enable transmitter and receiver \*/

controller->SPI\_PTCR = AT91C\_PDC\_RXTEN | AT91C\_PDC\_TXTEN;

controller->SPI\_IER = AT91C\_SPI\_SPENDRX; /\* enable buffer complete interrupt \*/

wait\_for\_completion(&transfer\_complete);

#ifdef DEBUG\_SPI

printk("[DEBUG INFO] : SPI ");

printk(KERN\_INFO "[DEBUG INFO] : SPI transfer end\n");

#endif

return 0;

}

### 5.3 spi\_dev.c文件详解

SPI设备拥有一个有限的用户空间API，用以支持基本的对SPI从设备的读写操作。

### 5.3.1 驱动与内核接口层

驱动与内核接口层主要完成驱动模块在Linux内核的注册加载、卸载清除工作。这部分工作分别由初始化和退出函数完成。

#### 5.3.1.1 初始化函数

SPI设备模块初始化函数主要用于注册一个字符型设备并创建一个文件系统节点。

**static** **int** \_\_init spi\_dev\_init(**void**)

{

**#ifdef** CONFIG\_DEVFS\_FS

**int** i;

**#endif**

**if** (register\_chrdev(SPI\_MAJOR, "spi", &spidev\_fops)) {

printk(KERN\_ERR "at91\_spidev: Unable to get major %d for SPI bus\n", SPI\_MAJOR);

**return** -EIO;

}

/\*

\* 注册字符型设备，其原型为

\* int register\_chrdev(unsigned int, const char \*, struct file\_operations \*);

\* 参数依次为主设备号，名称和file\_operations结构体指针。它的定义在内核文件

\* /fs/char\_dev.c中。register\_chrdev()是一个老式分配设备编号范围的函数。它分配一

\* 个单独主设备号和0~255的次设备号范围。如果申请的主设备号为0则动态分配一个。该函数

\* 还需传入一个file\_operations结构的指针，函数内部自动分配了一个新的cdev结构并对其\* 进行初始化，然后把它添加到char\_device\_struct结构体中。最后把主设备号返回出来。

\* 内核中所有已分配的字符设备编号都记录在一个名为chrdevs散列表里。该散列表中的每一

\* 个元素是一个char\_device\_struct结构，它的定义如下：

\* static struct char\_device\_struct {

\* struct char\_device\_struct \*next; // 指向散列冲突链表中的下一个元素的指针

\* unsigned int major; // 主设备号

\* unsigned int baseminor;   // 起始次设备号 ­

\* int minorct;   // 设备编号的范围大小 ­

\* const char \*name;    // 处理该设备编号范围内的设备驱动的名称 ­

\* struct file\_operations \*fops;

\* struct cdev \*cdev;    // 指向字符设备驱动程序描述符的指针 ­

\* } \*chrdevs[MAX\_PROBE\_HASH];

\* register\_chrdev()调用\_\_register\_chrdev\_region()函数来注册一组设备编号范围，

\* 即一个char\_device\_struct结构。函数\_\_register\_chrdev\_region()主要执行以下步骤：

\* 1.分配一个新的char\_device\_struct结构，并用0填充。 ­

\* 2.如果申请的设备编号范围的主设备号为0，那么表示设备驱动程序请求动态分配一个主设

\* 备号。动态分配主设备号的原则是从散列表的最后一个桶向前寻找，那个桶是空的，主设备

\* 号就是相应散列桶的序号。所以动态分配的主设备号总是小于 256，如果每个桶都有字符设\* 备编号了，那动态分配就会失败。 ­

\* 3.根据参数设置char\_device\_struct结构中的初始设备号，范围大小及设备驱动名称。

\* 4. 计算出主设备号所对应的散列桶，为新的 char\_device\_struct 结构寻找正确的位置。

\* 同时，如果设备编号范围有重复的话，则出错返回。 ­

\* 5. 将新的char\_device\_struct结构插入散列表中，并返回char\_device\_struct结构的

\* 地址。 ­

\*\*/

**#ifdef** CONFIG\_DEVFS\_FS

devfs\_mk\_dir("spi");

/\*

\* 在devfs的命名空间中创建一个名为spi的文件夹，这个函数定义在内核文件

\* /fs/devfs/base.c中。

\*\*/

**for** (i = 0; i < NR\_SPI\_DEVICES; i++) {

devfs\_mk\_cdev(MKDEV(SPI\_MAJOR, i), S\_IFCHR | S\_IRUSR | S\_IWUSR, "spi/%d",i);

}

/\* 创建设备节点，定义在内核文件/fs/devfs/base.c中，否则需要调用mknod手工创建节点

\*\*/

**#endif**

printk(KERN\_INFO "AT91 SPI driver loaded\n");

**return** 0;

}

#### 5.3.1.2 卸载函数

卸载函数负责将驱动模块从内核中移除。

**static** **void** \_\_exit spi\_dev\_exit(**void**)

{

**#ifdef** CONFIG\_DEVFS\_FS

**int** i;

**for** (i = 0; i < NR\_SPI\_DEVICES; i++) {

devfs\_remove("spi/%d", i); /\* 移除设备文件 \*/

}

devfs\_remove("spi");/\* 移除设备在文件系统中对应的节点 \*/

**#endif**

**if** (unregister\_chrdev(SPI\_MAJOR, "spi")) {

printk(KERN\_ERR "at91\_spidev: Unable to release major %d for SPI bus\n", SPI\_MAJOR);

**return**;

/\* 移除该设备所对应的字符型驱动\*/

}

### 5.3.2 定义file\_operations结构

在写底层驱动程序的时候struct file\_operations是一个很重要的结构体，这个结构是字符设备驱动程序的核心，当应用程序操作设备文件时所调用的open、read、write等函数，最终会调用这个结构中指定的对应函数。struct file\_operations是一个字符设备把驱动的操作和设备号联系在一起的纽带，是一系列指针的集合，每个被打开的文件都对应于一系列的操作，这就是file\_operations用来执行一系列的系统调用。

文件接口spidev\_fops代码如下：

static struct file\_operations spidev\_fops = {

.owner = THIS\_MODULE,

.read = spidev\_read,

.write = spidev\_write,

.open = spidev\_open,

.release = spidev\_close,

};

设备驱动程序所提供的这组入口点由一个结构对系统进行说明，在设备驱动程序初始化的时候对系统进行登记，以便系统在适当的时候调用。

在这个项目中，只对cs5532进行读写操作，因此除了打开和关闭文件所需的open()和release()之外，仅仅实现了read()和write()这两个基本方法。

#### 5.3.2.1 struct module \*owner

第一个 file\_operations 成员根本不是一个操作; 它是一个指向拥有这个结构的模块的指针. 这个成员用来在它的操作还在被使用时阻止模块被卸载. 几乎所有时间中, 它被简单初始化为THIS\_MODULE,即一个在 <linux/module.h> 中定义的宏。

**#ifdef** MODULE

**extern** **struct** module \_\_this\_module;

**#define** THIS\_MODULE (&\_\_this\_module)

**#else**

**#define** THIS\_MODULE ((**struct** module \*)0)

**#endif**

#### 5.3.2.2 文件的打开与关闭

对设备文件进行的第一个操作是open，若不对其进行定义，则总是打开文件成功。

**static** **int** **spidev\_open**(**struct** inode\* inode, **struct** file \*file) {

**unsigned** **int** spi\_device = MINOR(inode->i\_rdev);

**if** (spi\_device >= NR\_SPI\_DEVICES)

**return** -ENODEV;

file->private\_data = (**void** \*) spi\_device;

**return** 0;

}

#### 5.3.2.3 读写函数指针

在本项目中，代码实现如下：

**static** ssize\_t spidev\_read(**struct** file \* file, **char** \_\_user \* userbuf, size\_t count, loff\_t \* off)

{

**char** buffer[100];

memset(buffer,0, 100);

spi\_access\_bus(0);

send\_data(buffer, count);

spi\_release\_bus(0);

**if** (copy\_to\_user(userbuf, buffer, count))

**return** -EFAULT;

**return** 0;

}

首先创建一个字符型数组buffer，并初始化为0，用于存放读取回来的数据。然后通过调用at91\_spi.c中的spi\_access\_bus(0);send\_data(buffer, count);和spi\_release\_bus(0);来完成数据交换，读取的数据放在buffer中，copy\_to\_user()负责把buffer中的数据复制到用户空间。

（重点介绍copy\_to\_user ）

**static** ssize\_t spidev\_write(**struct** file \* file, **const** **char** \_\_user \* userbuf, size\_t count, loff\_t \* off)

{

**char** buffer[100];

**if** (copy\_from\_user(buffer, userbuf, count))

**return** -EFAULT;

spi\_access\_bus(0);

send\_data(buffer,count);

spi\_release\_bus(0);

**return** 0;

}

同样，第一步创建一个字符型数组，然后把将要发送的数据从用户空间复制到内核空间的这个数组中来，再使用和上例相同的方法进行数据交换。

1. 参考文献：

   Jonathan Corbet，Alessandro Rubini & Greg Kroah-Hartman， 《Linux设备驱动程序（第三版）》，2006.01，中国电力出版社；

   宋宝华，《Linux设备开发详解（第二版）》，2010，人民邮电出版社； [↑](#endnote-ref-1)
2. 崔微，秦征宇，《基于ARM Linux的SPI实现》，2009.08，重庆科技学院学报（自然科学版）第11卷 第4期； [↑](#endnote-ref-2)
3. 仇洁婷，陈儒军，《SPI串行外围设备驱动移植》，2010.06 [↑](#endnote-ref-3)