基于 SPI 总线协议的字符设备驱动程序①

付兴武, 张 军, 王 洋

(辽宁工程技术大学 电气与控制工程学院, 葫芦岛 125105)

摘 要: 介绍了嵌入式 Linux 字符设备驱动程序的概念、分类及关键技术点, 总结分析了 SPI 总线设备驱动程序的详细细节. 同时本文采用 mini2440-ARM9 开发板作为硬件平台, 成功实验测试本文列出的驱动程序.

关键词: Linux; 字符设备驱动; SPI 总线; 嵌入式系统

Char Drive Procedures Based on SPI Bus Protocol

FU Xing-Wu, ZHANG Jun, WANG Yang

(Faculty of Electrical and Control Engineering, Liaoning Technical University, Huludao 125105, China)

Abstract: The paper describes the conceptions,classification and key_points of the char drive procedures.The most important is that the summary and analysis the details of SPI bus device driver. At the same time, this paper uses the mini2440 ARM9 development board as the realization platform.And the program listed in the paper has been tested successfully.

Key words: Linux; char drive procedures; SPI Bus; embedded system

SPI(Serial Peripheral Interface), 是一种常用的同步串行通信协议, 可以实现高速的串行数据收发. SPI 总线可直接与具有标准 SPI 接口的不同厂家生产的设备进行通信, 兼容性比较好, 这使得 SPI 通信接口越来越广泛, 在嵌入式开发领域也愈发重要, 所以 Linux 系统下 SPI 驱动程序开发尤为关键.

在嵌入式 Linux 开发过程中, 驱动程序开发是一个至关重要环节. 设备驱动程序就是隐藏了所操作硬件的具体细节, 给用户在应用程序中提供了一套统一的编程接口. 所有硬件设备都可以抽象为一个个设备文件(网络设备除外), 使得用户不用去考虑复杂的硬件连接, 从而对硬件的操作转化为对简单文件的操作[1,2].

嵌入式 Linux 中设备驱动大体上分为三类: 字符设备驱动、块设备驱动、网络设备驱动. 其中字符设备以字节为单位顺序读写的设备, 如键盘、串口等. SPI 总线属于字符设备, 所以我们这里着重介绍一下字符设备驱动程序的编写.

1 关于字符设备驱动程序

嵌入式 Linux 系统中应用程序不能直接控制相应硬件, 在系统当中应用程序与硬件之间有许多相互联系的桥梁. 式 Linux 应用程序通过设备文件(或设备结点)来链接驱动程序从而实现对字符设备的操作. 字符设备文件是连接应用程序和字符设备驱动的桥梁, 同样字符设备文件与字符设备驱动之间通信的桥梁是主设备号.

(1) 设备号及其申请

设备号有主设备号和次设备号之分. 主设备号用来标识与设备文件相连的驱动程序, 次设备号被驱动程序辨认操作的具体操作的硬件设备[2]. 主设备号系统不会自动提供, 需要向系统申请. 主设备号申请一般有两种方式: 动态分配和静态申请.

本文采用的是静态申请方式, 使用静态申请的前提是需要确定一个没有被使用的主设备号. (注意: 无论采用哪种方式分配主设备都应该在驱动程序卸载的模块中将主设备号注销).

① 收稿时间:2011-11-24;收到修改稿时间:2012-11-15

静态申请的一般函数表达式:

Int register_chrdev_region(dev_t from,unsigned count,const char*name)

功能: 注册从 from 开始的 count 个设备号(主设备号不变, 次设备号增加, 如果次设备号溢出, 主设备号加 1)

(2) 设备文件的创建

设备文件也指设备结点, 链接设备文件与设备驱动程序的就是设备号. 设备文件的创建一般也有两种方式: 使用 mknod 手工创建或者自动创建.

本文中驱动程序是采用模块方式动态添加, 所以设备文件的创建采用的是手工创建的方式. 在创建设备文件方面, 为了满足系统性要求一般在 dev 目录下即可,当然也可以在其他目录下[3].

如果你想查看可以在 Linux 超级终端中输入: #cd /dev //先 cd 到 dev 目录下, 而后执行#ls -l 就会显示一下内容:

```
crw-rw----  1 vcsa tty      7,   3 09-13 09:47 vcs3
```

上图所示中 crw-rw 中的 C 代表字符设备, 7 代表主设备号, 3 代表次设备号

手工创建使用的是 mknod 指令, 且后面为设备文件名、设备类型(字符设备为 c、块设备为 b)、主设备号及次设备号. 就本实验来说具体如下:

#mknod mini2440_spi c 55

```
[root@localhost dev]# mknod mini2440_spi c 55 0
[root@localhost dev]# ls mini2440_spi -l
crw-r--r-- 1 root root 55, 0 10-29 21:20 mini2440_spi
```

(3) 设备注册

注册字符设备驱动程序是指注册 file_ operations 结构体的变量, 需要对字符设备进行详细描述而后添加进内核.

设备注册主要分两种方式: 1.使用 register_chrdev()函数; 2.使用 cdev_add()函数注册. 本文中用的第二种方式进行设备注册. 其功能实现主要靠 struct cdev 结构来实现; 实现设备注册一般分三个步骤: 1 分配 cdev; 2 初始化 cdev; 3 添加 cdev.

(4) 与字符设备驱动相关的数据结构

字符设备驱动程序中有三种比较重要的数据结构, 分别是: 1)truct file, 每次打开设备就会创建一个 struct file 结构体, 结构体内主要包含了文件指针位置及文件结构体指针. 2)truct inode,每次打开同一个文件都会创建一个 struct file 结构体, 但是不管打开几次只能创建一个 struct inode 结构体. 结构体内主要存储设备的主主设备号信息. 3)truct file_operations,对于字符设备驱动程序最核心的就是 struct file_operations 结构, 它是嵌入式 Linux 内核识别设备的桥梁. 这个结构实际上是 VFS(虚拟文件系统)的文件接口, 它的每个成员函数一般都对应一个系统调用[4]. 整个驱动程序就是围绕着 struct file_operations 结构体展开的, 且驱动程序的主体就是完成对 struct file_operations 结构体成员函数指针(即设备操作函数)的完善.

2 SPI总线设备驱动硬件实现平台

本实验所用的开发平台为 Mini2440—ARM9 开发板. 采用 S3C2440 为处理器. 其中 S3C2440 处理器包含 2 通道 SPI 接口(SPI0、SPI1). 本实验中采用的 SPI0 口.

Mini2440 开发板带有一个 34 引脚的 GPIO 接口, 其中大部分引脚都是多功能引脚, 可通过相应的寄存器更改其用途以实现特定功能. 将其设置成 SPI 功能引脚, 具体对应关系如图 2 所示:

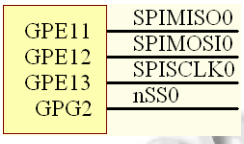

图 2 ARM9 关于 SPI 接口的硬件接口示意图

3 关于SPI总线设备驱动程序软件解析

3.1 头文件以及宏定义部分

#include<linux/gpio.h>

……

#define spi_major 55 //主设备定为 55, 故选择静态申请模式

#define spi_name “mini2440_spi” //定义 SPI 总线设备名称

/*用来检验 SPI 数据寄存器是否准备好发送或接收,利用宏定义设置成一个标志位*/

#define SPI_TXRX_READY

(((*spi_spsta0)&0X1)==0X1)

Struct cdev spiCdev //定义一个 cdev 结构体变量, 用于设备注册

int loopChar=0x88;

3.2 SPI 总线驱动模块的加载及卸载函数

module_init()函数主要包括主设备号申请、设备注册及 SPI 相关寄存器映射等操作. 驱动程序不能直接通过硬件物理地址去访问 I/O 内存资源, 必须先将相应的物理地址映射到内核的虚拟地址空间中后, 这样才能根据映射所得到的虚拟地址范围, 通过访内指令访问相应的 I/O 内存资源, 这就需要映射函数 ioremap().

其中 dev_t 数据类型实际上就是 32 位无符号整型, 其中高 12 位为主设备号, 低 20 位为次设备号; 宏 MKDEV()主要作用是将主、次设备号构造成 32 位设备号.

在命令行中执行:$insmod+模块名.ko 时执行该函数, 然后完成字符设备驱动的注册.

```
static int __init spi_init(void) {
  int result;
  dev_t devno = MKDEV(spi_major, 0);
  if(spi_major)
  result = register_chrdev_region(devno, 1, spi_name);          //静态申请执行函数
  else{
  result = alloc_chrdev_region(&devno, 0, 1, spi_name);         //动态申请执行函数
  spi_major = MAJOR(devno); }
  if(result < 0)
  return result;                 /*初始换 cdev 结构*/
  cdev_init(&spiCdev, &spi_fops);  //初始化 cdev 主要是使 cdev 结构与文件结构体相关联
  spiCdev.owner = THIS_MODULE;
  spiCdev.ops = &spi_fops;      /*注册字符设备*/
  if(cdev_add(&spiCdev, devno, 1))
  /*下面就是 S3C2440 与 SPI 相关寄存器地址映射的操作*/
  s3c2440_clkcon  =(int*)ioremap(0x4C00000c,3);
 spi_gpgcon = (int*)ioremap (0x56000060,4);
  ……
  spi_gpecon = (int*)ioremap (0x56000040,4);
  ……
  spi_spcon0 = (int*)ioremap(0x59000000,1);
  ……
  return result;
  }
```

```
/*模块卸载函数, 主要包括 设备注销、主设备号注销等操作*/
```

在命令行中输入: $rmmod+模块名 时执行该函数且此时模块名后面是不加.ko 后缀的.

```
static void __exit spi_exit(void) {
cdev_del(&spiCdev);
unregister_chrdev_region(MKDEV(spi_major,0),1);
}
```

3.3 文件结构体 file_operations

由上面所述驱动程序中最重要的数据结构是 file_operations, 它提供了驱动程序执行系统调用时所有功能函数的入口指针.

本文中的 file_operations 数据结构体的实现如下:

```
Static const struct file_operations spi_fops={
.owner=THIS_MODULE,  //驱动程序模块对应的模块名
.open=spi_open,   //执行打开设备文件操作的函数
.read=spi_read,   //执行对设备进行读操作的函数
.write=spi_write, //执行对设备进行写操作的函数
.release=spi_release, //关闭设备文件时执行的函数
};
```

3.4 file_operations 结构体成员函数的实现

在用户应用程序中对设备进行操作前必须先打开设备文件找到相应的设备驱动程序, 用户程序中的 fopen()与 fclose()函数对应于驱动程序中的 spi_open()与 spi_release();二者的具体实现方法如下:

Spi_open()函数中主要执行了 S3C2440 中 SPI 功能引脚的功能初始化.

```
static int spi_open(struct inode *inode, struct file *filp) {
filp->private_data =&spiCdev;      //每打开一个文件都自动创立一个 struct file 结构指针, 可以将设备注册有关的 cdev 结构体指针赋予文件指针变量, 使文件与抽象的 cdev 结构体关联起来
/*下面为对相应 GPIO 口进行设置, 将 GPE11 设置成 SPIMISO0 功能; 将 GPE12 设置成 SPIMOSI0 功能; GPE13 设置成 SPISCLK0; GPG2 设置成 nSS0. 寄存器设置比较简单在此不再赘述*/
…….
return 0;
}
```

对设备进行操作时需要打开相应的设备文件, 同样对设备执行完相应操作后需要及时关闭文件, spi_release()函数具体实现方法如下:

```
static int spi_release(struct inode *inode, struct file *filp) {
return 0;}
```

设备文件打开后, 就可以对 SPI 总线接口进行读写操作了, 具体读写操作功能实现方法如下: 由于 SPI 对应的发送/接收数据寄存器都是单字节大小, 首先我们定义单字节读写子函数, 实现过程如下:

单字节写操作子函数实现如下:

```
static void writeByte(const char c)
{
  int j = 0;
  *spi_sptdat0 = c;          //将要发送的数据存入发送数据寄存器
  for(j=0;j<0xFF;j++);
  while(!SPI_TXRX_READY)    //等待发送允许标志位  置位.
  for(j=0;j<0xFF;j++);
}
```

单字节读操作子函数实现如下:

在执行读/ 写操作时, 都是要先发送一字节进行读或者写操作的寄存器地址; 再发送任意一字节数据以获得时钟信号, 与此同时从机才能同一时间返回主机想要读取的数据. 故下面程序中: *spi_sptdat0=(char)loopChar;语句便起此作用.

```
static char readByte(void)
{
  int j = 0;
  char ch = 0;
  *spi_sptdat0 = (char)loopChar;
  for(j=0;j<0xFF;j++);
  while(!SPI_TXRX_READY)     //等待接收数据允许标志位置位.
      for(j=0;j<0xFF;j++);
  ch=*spi_sprdat0 ;              //读取接收寄存器
  return ch;
}
```

下面是对设备进行读操作的函数 spi_read()其中参数 flip 代表打开的设备文件指针, 参数 buf 代表读取的数据所存储的缓冲区, 参数 count 代表所读取的字节数, 参数 f_ops 代表文件当前访问位置.

其中第二个和第三个参数是系统调用用户赋值的, 另外两个是内核自动匹配的. 执行读操作成功返回 1. 调用读函数时参数 buf 为用户空间指针变量, 内核空间代码不能直接使用 buf 指针. 为了使用 buf 使用内核空间提供的专门函数访问用户空间指针, 比如: copy_to_user();copy_from_user().

```
static ssize_t spi_read(struct file *filp, char __user *buf, size_t count,loff_t *f_ops)
{
char ch;
ch=readByte();
copy_to_user(buf,&ch,1);
return 1;
}
```

执行写操作时函数参数的含义跟读操作相同. 此函数实现对设备的写操作, 且同样涉及到用户空间与内核空间的数据拷贝. 其中的 kmalloc 函数实现内核空间内动态申请内存, 同样需要注意的是使用 kmalloc()函数时, 在申请的内存使用完毕后需要及时释放, 否则会导致内存泄漏, 出现意想不到的错误. 进行数据拷贝时, 拷贝成功返回 0, 失败返回未拷贝成功的字节数.

```
static ssize_t spi_write(struct file *filp, const char __user *buf, size_t count,loff_t *f_ops)
{
int i;
char *kbuf;
char *pt;
kbuf=kmalloc(count,GFP_KERNEL);          // 参数 count 代表申请空间的大小以字节为单位, 参数 GFP_KERNEL 代表分配内存的类型.
If(NULL!=kbuf)
  {
   pt=kbuf;
   }
If(copy_from_user(kbuf,buf,count))
   {
    printk( “no enough memory!\n”);
    return -1;
```

```
    }
for(i=0;i<count;i++)
    {
      writeByte(*pt);
      pt++;
}
If(kbuf)
kfree(kbuf);
return count;
}
```

4　系统测试

(1) 生成驱动模块

生成模块化的驱动程序除了在内核编译时将其配置成<M>外还可以有另外一种方法. 可以独立于内核代码树. 这样就需要为可加载的模块编写 Makefile 文件[5]. 关键是如何设置 Makefile 文件最重要的几个变量, 针对与本实验来说:

```
CC          =arm-linux-gcc
obj-m :=smodule.o
KERNELDIR ?=/mnt/v/linux-2.6.32.2
```

CC 是编译器, obj-m 为需要编译的目标模块, KERNELDIR 为内核路径. 注意在编写可加载模块前先要有一个内核代码目录树. KERNELDIR 的内核版本必须与运行版本一致, 否则编译出的模块往往无法加载.

(2) 系统测试代码

```
#include <stdio.h>
…….
Int spi_fd; //SPI 文件描述符
Int main(void)
{
unsigned char s=4;
spi_fd=open("/dev/mini2440_spi",O_RDWR);
while(1)
{
   write(spi_fd,&s,1);
}
   close(spi_fd);
   return 0;
}
```

源文件在此命名为: spi_test.c 文件. 在交叉编译器中将测试程序编译成可执行代码文件命名为: spi_test. 最后在超级终端中执行即可, 执行结果如图 3 所示: (实验测试的是 SPIMOSI 及 SCLK 引脚输出).

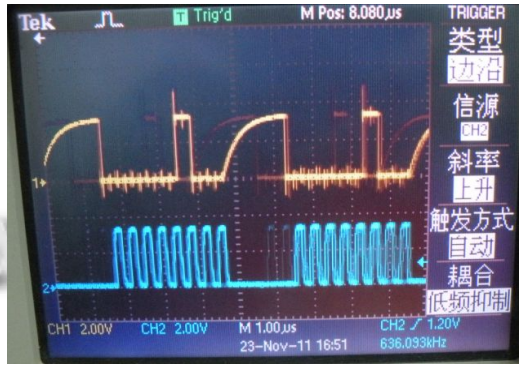

图 3　实验测试波形

由测试波形可以表明, 本驱动程序基本实现了对 SPI 总线设备的控制.

5　结语

本文详细介绍了嵌入式 Linux 下字符设备驱动程序的编写过程及需要注意的细节, 并以 SPI 总线设备为例详细介绍了 SPI 总线设备驱动程序的实现. 最后以 Mini2440-ARM9 为硬件实施平台, 编写了简单的测试程序实现了 SPI 数据的发送. 对以 SPI 总线协议进行通信的设备提供一定的参考价值.

参考文献

1 博韦.深入理解 Linux 内核.北京:中国电力出版社,2008.
2 宋宝华,何昭然,史海滨译.精通 Linux 设备驱动程序开发. 北京:人民邮电出版社,2010.
3 冯国进.Linux 驱动程序开发实例.北京:机械工业出版社, 2011.
4 李红姬,李明吉译.Linux 设备驱动开发技术及应用.北京:人民邮电出版社,2008.
5 孙纪坤,张小全.嵌入式 Linux 系统开发技术详解基于 ARM.北京:人民邮电出版社,2007.