# 字符设备基础代码

## 前言

通常我们设计到的设备只有块设备和字符设备两种，目前工作中大量接触的设备为字符设备，工作占比90+%。所以把各类字符设备的共同部分整理出来，各种字符设备的差异性均再此基础上扩展出来。目的是在用户以后工作过程中，可以对关键代码进行*复制/粘贴*，以提高工作效率。

因差异化的字符设备的驱动由器件厂家提供，BSP工作本身不涉及器件驱动的编码。BSP的工作是将厂家提供的驱动程序整合其中。或向厂家提供接口描述本机型的特性，由厂家实现该接口。BSP层面通过编译机制查找厂家器件驱动中里符号作为调用入口并接入其中。

作为Hal层和器件驱动的衔接带，向上暴露操作接口，向下直接和器件驱动交互。

经过Linux开发人员的长期的实践，将上下层之间的调用规则标准化，形成了一套非常成熟的接口或方法。

为确保代码的正确性，以下代码均在Ubuntu环境下编译及测试通过：

代码下载地址：

git clone <https://hub.fastgit.org/yaolihui/test_driver.git> -b main

cd test\_driver/1\_dev\_attr/

该目录主要内容为体现如何注册字符设备，和创建设备节点。主要如下6个文件：

./char\_dev.c #注册字符设备

./sysfs\_create\_file.c #创建/dev文件节点

./device\_create\_file.c #创建设备属性

./device\_add\_group.c #创建设备属性组

./device\_add\_groups.c #创建设备属性组 集合

./Makefile #编译入口

检测驱动是否已经挂接成功的重要检查点，即查看设备节点是否创建成功，但不限于此。比如查看sensor\_id是否已经正确设置等。

char\_dev.c

#include <linux/fs.h> // struct file\_operations

#include <linux/cdev.h> // struct cdev

#include <linux/uaccess.h>

char mchar[10] = "test";

int test\_open(struct inode \* i, struct file \* f)

{

printk("%s \n", \_\_func\_\_);

return 0;

}

ssize\_t test\_read(struct file \* f, char \_\_user \* buf, size\_t count , loff\_t \*p)

{

printk("%s, mchar=%s, count=%d, loff\_t=%d, strlen=%d \n", \_\_func\_\_, mchar, count, \*p, strlen(mchar));

if (copy\_to\_user(buf, mchar, strlen(mchar))) printk("%s: copy\_to\_user fail \n", \_\_func\_\_);

return 0; // 0:表示成功；失败返回没有读取成功的字符数量

}

ssize\_t test\_write(struct file \* f, const char \_\_user \* buf, size\_t count , loff\_t \*p)

{

printk("%s, count=%d, loff\_t=%d \n", \_\_func\_\_, count, \*p);

if (copy\_from\_user(mchar, buf, count)) printk("%s: copy\_from\_user fail \n", \_\_func\_\_);

printk("%s: mchar=%s \n", \_\_func\_\_, mchar);

return count; //返回成功写入的字节数

}

//向Hal层暴露操作的操作接口

struct file\_operations fops = {

.owner = THIS\_MODULE, //固定写法

.open = test\_open, //Hal层打开设备节点时的Kernel入口

.read = test\_read, //Hal层读操作

.write = test\_write, //Hal层写操作

.release = NULL, //Hal层关闭节点操作

.unlocked\_ioctl = NULL, //Hal层ioctl调用

};

static struct cdev \*cdev; //字符设备本身

static dev\_t devno; //唯一设备号，设备的唯一标识

static int major; //主设备号

static const char\* DEV\_NAME = "1\_test\_char\_dev"; // proc/devices中的设备名称

int reg\_cdev(int mjr)

{

//register\_chrdev(mjr, DEV\_NAME, &fops); //注册字符设备的简易方法。可取代以下的所有代码操作。

devno = MKDEV(mjr, 0); //主设备转化为系统可认识的唯一设备标识。次设备号最多允许254个

int ret;

if(mjr) {

ret = register\_chrdev\_region(devno , 1, DEV\_NAME); //静态方式指定设备号。不能和已有设备号相同。不常用。

major = mjr;

} else {

ret = alloc\_chrdev\_region(&devno , 0, 1, DEV\_NAME); //动态方式，由kernel查找第一个没有使用的设备号，由&devo带出。

major = MAJOR(devno);

}

printk("%s: ret=%d, major=%d, devno=%d\n", \_\_func\_\_, ret, major, devno);

if(ret < 0) return ret;

// cdev\_init(cdev, &fops) // 静态方式创建字符设备，cdev的内存须自行kmalloc

cdev= cdev\_alloc(); // 动态方式为struct cdev分配及内存，及清零初始化

cdev->ops = &fops; // 关联fops 对象

cdev->owner = THIS\_MODULE;

printk("%s: cdev=%p\n", \_\_func\_\_, cdev);

if(!cdev) {

ret = -EFAULT;

unregister\_chrdev\_region(devno, 1); //内存分配失败后，把已申请的设备号还给系统。

}

ret = cdev\_add(cdev, devno, 1); //添加字符设备到kobject树上。 //kobject树中某节点的存在性，充要表示了该设备的注册成功

printk("%s: ret=%d, cdev=%p, devno=%d, major=%d\n", \_\_func\_\_, ret, cdev, devno, major);

return devno;

}

void unreg\_cdev(void)

{

cdev\_del(cdev); //从kobject树中移除该设备对象

printk("%s: cdev\_del:cdev=%p\n", \_\_func\_\_, cdev);

unregister\_chrdev\_region(devno, 1); //回收设备号

printk("%s: unregister\_chrdev\_region:devno=%d, major=%d\n", \_\_func\_\_, devno, major);

}

以上代码不能在本例中单独运行，其仅仅描述字符设备的注册过程，只作为其他文件的*子调用*存在。依赖关系可参考Makefile文件中(**char\_dev.o**)：

sysfs\_create-objs := sysfs\_create\_file.o **char\_dev.o**

device\_create-objs := device\_create\_file.o **char\_dev.o**

add\_group-objs := device\_add\_group.o **char\_dev.o**

add\_groups-objs := device\_add\_groups.o **char\_dev.o**

若要使上述代码单独运行可参照sysfs\_create\_file.c 自行添加module\_init和module\_exit，并生成char\_dev.ko。

module\_init(reg\_cdev);

module\_exit(unreg\_cdev);

**强调一点**：若char\_dev.ko能正常运行，但这种写法属于2.6之前的一种古老写法，现在已经不常用了。

可通过如下命令查看设备是否已经注册成功：

cat /proc/devices

回显中包含DEV\_NAME中定义字符串且设备号和major相同，如下：



*附注：此种情况仅仅说明了字符设备注册成功，但不表示字符节点创建成功。*

须手动创建字符节点（或由Hal层代码生成），命令如下：

Usage: mknod [OPTION]... NAME TYPE [MAJOR MINOR]

sudo mknod 1\_test\_char\_dev c 300 0

其中1\_test\_char\_dev，为节点名称，为任意字符。可与/proc/devices中的设备名称不同。用于给Hal层代码中 open(“/dev/1\_test\_char\_dev”, flag)使用。 c 标识字符设备；300为主设备号，0为此设备号，主次设备号必须和注册时代码里提供的主次设备号相同，并因此使设备节点和已注册的设备设备发生关联。

当下不常使用上述写法，为历史遗留。Linux 2.6之后引入“总线、设备、驱动”模型（后文简称“模型”），基于模型的代码有很好的向后兼容性。模型的概念不在此文介绍。

本例程代码基于模型接口实现。

sysfs\_create\_file.c

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/init.h>

#include <linux/device.h>

#include <linux/cdev.h>

#define MY\_MAJOR 300

extern int reg\_cdev(int mjr); // 声明外部引用，调用上述Char\_dev.c中的reg\_cdev方法

extern void unreg\_cdev(void); // 声明外部引用，调用上述char\_dev.c中的 unreg\_cdev方法

dev\_t dno = 0; // 暂存char\_dev.c中的reg\_cdev方法生成的设备号

struct class \*cls; // 对应/sys/class目录中的文件目录

struct device \*dev; // 对应上述/sys/class中cls下的目录

char mm[10]; // 内核临时存储，测试Hal层传来的字符串

ssize\_t test\_show(struct device \*dev, struct device\_attribute \*attr,char \*buf)

{

printk("mm=%s\n", mm);

strcpy(buf, mm); // 将Kernel中处理过的字符串返给Hal层

printk("test\_show dev=%p, buf=%s\n", dev, buf);

return 10;

}

ssize\_t test\_store(struct device \*dev, struct device\_attribute \*attr,const char \*buf, size\_t count)

{

strcpy(mm, buf); //Kernel记录Hal层传下来的字符串

printk("buf=%s\n", buf);

mm[5]='$'; //对已存字符做任意加工

mm[9]='\0';

printk("test\_store dev=%p, mm=%s, count=%d", dev, mm, (int)count);

return 10;

}

**DEVICE\_ATTR(test, 0660, test\_show, test\_store);** //宏：将上述两个方法展开，并生成dev\_attr\_test.attr 引用。

//其中第1个参数xxx被替换为dev\_attr\_xxx.attr

//第2个参数为对应文件系统中的chmod值，但最大为0660。大于此值编译报错。

//第3个参数为read方法，自定义。同理第4个参数为write方法

// ko安装成功后会在/sys/class/1\_class/1\_device/中生成test节点，读写时自动调用上述read/write方法

int enter(void)

{

printk("\n%s:%s:%d\n", \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_);

**dno** = reg\_cdev(MY\_MAJOR); // 注册字符设备（调用上述Char\_dev.c中的reg\_cdev方法）

printk("%s:reg\_cdev:dno=%d\n", \_\_func\_\_, dno);

cls = class\_create(THIS\_MODULE, "1\_class"); // 创建/sys/class/1\_class目录

printk("class\_creat cls=%p\n", cls);

dev = device\_create(cls, NULL, **dno**, NULL, "1\_device"); // 创建/sys/class/1\_class/1\_device目录，其中uevent节点记录本设备相关信息

// /dev/1\_device 设备节点

printk("device\_create dev=%p\n", dev);

**sysfs\_create\_file**(&dev->kobj, &dev\_attr\_test.attr); // 在/sys/class/1\_class/1\_device目录中生成test属性节点

printk("sysfs\_create\_file\n");

printk("\nend of enter\n");

return 0;

}

void quit(void)

{

device\_destroy(cls, dno); // 回收设备，删除目录/sys/class/1\_class/1\_device和 /dev/1\_device 目录

printk("device\_destory\n");

class\_destroy(cls); // 删除目录/sys/class/1\_class/目录

printk("class\_destory\n");

unreg\_cdev(); // 回收字符设备

printk("%s:unreg\_cdev dno=%d\n", \_\_func\_\_, dno);

printk("\n%s:%s:%d\n", \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_);

}

module\_init(enter); // ko文件入口

module\_exit(quit); // ko 出口

MODULE\_LICENSE("GPL"); //签名固定写法，必须存在

本例解读：当对/sys/class/1\_class/1\_device/test节点read|write时，将自动调用例程中的test\_show|test\_store方法。在test\_show|test\_store方法里对与用户态传入的数据，不需要经过copy\_to\_user和copy\_from\_user的转换。但在reg\_cdev方法中绑定的fops操作依然需要坚持调用copy\_to\_user和copy\_from\_user以实现用户态和内核态的转化。

对比旧式的字符设备和节点的创建方法，发现/dev下的字符节点由内核创建，而不再交由Hal层创建。但旧式字符设备写法依然是不可替代的，至少ioctl调用非常有用，模型的写法只是对旧式写法进行了有利的补充。

device\_create\_file.c

#include <linux/kernel.h>

.......

#define MY\_MAJOR 301

......

ssize\_t test\_show(struct device \*dev, struct device\_attribute \*attr,char \*buf)

{

.......

return 10;

}

ssize\_t test\_store(struct device \*dev, struct device\_attribute \*attr,const char \*buf, size\_t count)

{

.......

return 10;

}

DEVICE\_ATTR(test, 0660, test\_show, test\_store);

int enter(void)

{

cls = class\_create(THIS\_MODULE,"2\_class");

dev = device\_create(cls, NULL, dno, NULL, "2\_device");

**device\_create\_file**(dev, &dev\_attr\_test); // 其它代码和上述一样，仅本句不同。执行成功后会在/sys/class/2\_class/2\_device中生成test属性

return 0;

}

void quit(void)

{

.......

}

module\_init(enter);

module\_exit(quit);

MODULE\_LICENSE("GPL");

解读：对比方法sysfs\_create\_file(&dev->kobj, &dev\_attr\_test.attr);和device\_create\_file(dev, &dev\_attr\_test);，仅仅2个参数分装级别不同，device\_create\_file使用的device对象的封装度高一个级别，使用起来会更加方便。其实质是对sysfs\_create\_file进行内部调用，逻辑上几乎没有差别。两种方法根据需要自行选择

device\_add\_group.c

#include <linux/kernel.h>

......

ssize\_t test\_show(struct device \*dev, struct device\_attribute \*attr,char \*buf)

{

return 10;

}

ssize\_t test\_store(struct device \*dev, struct device\_attribute \*attr,const char \*buf, size\_t count)

{

return 1;

}

ssize\_t test2\_show(struct device \*dev, struct device\_attribute \*attr,char \*buf)

{

...... // 和上面方法一样，增加了test2

return 10;

}

ssize\_t test2\_store(struct device \*dev, struct device\_attribute \*attr,const char \*buf, size\_t count)

{

...... // 和上面方法一样，增加了test2

return 10;

}

DEVICE\_ATTR(test, 0660, test\_show, test\_store);

DEVICE\_ATTR(test2, 0660, test2\_show, test2\_store); // 和上面方法一样，增加了test2

struct attribute \*attrs[] = { // 此处封装成了数组

&dev\_attr\_test.attr, / /test展开的属性

&dev\_attr\_test2.attr, // test2展开的属性

NULL, //此句必须有

};

struct attribute\_group attr\_grp = { //再将上述两个属性封装成group

.attrs = attrs,

};

int enter(void)

{

cls = class\_create(THIS\_MODULE,"3\_class");

dev = device\_create(cls, NULL, dno, NULL, "3\_device");

device\_add\_group(dev, &attr\_grp); //和前例对比此处变成了group，执行完成后会在/sys/class/3\_class/3\_device中同时生成两个属性test, test2

printk("device\_add\_group\n");

......

}

......

device\_add\_groups.c

#include <linux/kernel.h>

......

DEVICE\_ATTR(test1, 0660, test\_show, test\_store);

DEVICE\_ATTR(test2, 0660, test2\_show, test2\_store);

DEVICE\_ATTR(test3, 0660, test\_show, test\_store);

DEVICE\_ATTR(test4, 0660, test2\_show, test2\_store);

struct attribute \*attrs[] = { //属性数组1

&dev\_attr\_test1.attr,

&dev\_attr\_test2.attr,

NULL,

};

struct attribute\_group attr\_grp = { //封装成group1

.attrs = attrs,

};

struct attribute \*attrs2[] = { //属性数组2

&dev\_attr\_test3.attr,

&dev\_attr\_test4.attr,

NULL,

};

struct attribute\_group attr\_grp2 = { //封装成group2

.attrs = attrs2,

};

const struct attribute\_group \*attr\_grps[] = { //再将group1， group2封装成数组

&attr\_grp,

&attr\_grp2,

NULL,

};

int enter(void)

{

cls = class\_create(THIS\_MODULE,"4\_class");

dev = device\_create(cls, NULL, dno, NULL, "4\_device");

device\_add\_groups(dev, attr\_grps); // 添加group数组。在/sys/class/4\_class/4\_device中一次生成4个设备属性节点test/test1/test2/test3

return 0;

}

......

小结：

sysfs\_create\_file和device\_create\_file可相互替换用于生成单个设备属性节点。

device\_add\_group和device\_add\_groups可一次生成多个设备属性节点，后者可对组进行再次分组。

属性节点的出现可在一定程度上替代file\_operation的.read和.write方法，其中的数据传递不需要从用户态到核心态之间再次转化，因此使用起来更加方便。