2 字符设备的新写法

2.1 设备号管理

在上一章中的实验中，设备号直接在驱动代码中写死了。这样做会带来很多麻烦：

①编译驱动代码前，必须要先查看目标系统中设备号的占用情况；

②更换设备后原先驱动中写死的设备号就可能已被占用；

③原先的驱动注册函数 register_chrdev()输入参数中仅有主设备号而没有次设备号，这意味着一个设备就会占用所有的次设备号，十分浪费。

针对这些问题，Linux 内核提出了新的字符设备注册方法，并由内核来管理设备号。新增两个设备号注册函数。

1.当驱动程序需要给定主设备号时，使用函数来注册设备号：

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

输入参数说明：

from：需要申请的起始设备号，dev_t 类型，它取代了原先的主设备号和次设备号，在需要指定主次设备号的情况下，可以通过方法 from=MKDEV(major, minor);来获取他的值。

count：需要申请设备号的个数，一般只要申请一个。

name：设备名。

示例：

```
1.  int   major = 200;    //主设备号指定为 200
2.  int   minor = 0;      //次设备号为 0
3.  dev_t devid = MKDEV(major, minor);    //通过主次设备号获得设备号
4.  /* 向内核注册设备号 */
5.  register_chrdev_region(devid, 1, "xxx-dev");
```

2.驱动程序不需要指定主设备号时，使用函数：

int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name)

输入参数说明：

dev：设备号指针，不指定主次设备号的情况下，设备号由内核分配，因此传入指针来获取设备号，注册成功后可以通过方法 major = MAJOR(*dev); 和 minor = MINOR(*dev);分别获取主次设备号，如果不需要用到主次设备号，不获取也可以。

baseminor：次设备号起始地址。

count：需要申请设备号的个数，一般只要申请一个。

name：设备名。

示例：

```
1.  dev_t devid;    //设备号
2.  /* 申请设备号 */
3.  alloc_chrdev_region(&devid, 0, 1, "xxx-dev");
```

3.注销设备号只要使用同一个函数：
void unregister_chrdev_region(dev_t from, unsigned count)
输入参数含义以注册函数一致。
示例：

```
1.  dev_t devid;    //设备号
2.  /* 申请设备号 */
3.  alloc_chrdev_region(&devid, 0, 1, " xxx-dev");
4.  /* 注销设备号 */
5.  unregister_chrdev_region(devid, 1);
```

2.2 新的注册方法

上面提到过注册函数 register_chrdev()存在问题。和新的设备号管理方式相对应，现在摒弃 register_chrdev()，使用一套新的方法来来注册字符设备。

2.2.1 字符设备数据结构

使用 cdev 结构体来定来定义一个字符设备，他定义在 include/linux/cdev.h 中，具体如下：

```
1.  struct cdev {
2.      struct kobject kobj;
3.      struct module *owner;
4.      const struct file_operations *ops;
5.      struct list_head list;
6.      dev_t dev;
7.      unsigned int count;
8.  };
```

重要成员变量：
owner：一般设置为 THIS_MODULE；
ops：设备操作函数指针；
dev：设备号。

2.2.2 cdev 结构体初始化

cdev 结构体变量定义后需要用 cdev_init()函数初始化，函数原型为：
void cdev_init(struct cdev *cdev, const struct file_operations *fops)
cdev：字符设备结构体指针；
fops：设备操作函数集合结构体指针。
使用示例：

```
1.    /* 字符设备 */
2.   struct cdev ax_cdev = {
3.       .owner = THIS_MODULE,
4.   };
5.
6.   /* 设备操作函数 */
7.   static struct file_operations ax_fops = {
8.       .owner = THIS_MODULE,
9.       .open……
10.  };
11.
12.  /* ax_cdev 变量初始化 */
13.  cdev_init(&ax_cdev, &ax_fops);
```

2.2.3 新的注册和注销函数

初始化字符结构体变量后，便可以使用这个变量来向 Linux 系统注册字符设备。使用新的注册函数 cdev_add，原型为：

int cdev_add(struct cdev *p, dev_t dev, unsigned count)

p：上面初始化后的字符设备结构体变量；

dev：设备号；

count：需要添加的设备数量。

注册函数变了，注销卸载函数也一样，不再使用 unregister_chrdev()函数，改用 cdev_del()函数，原型：

void cdev_del(struct cdev *p)

输入参数即为字符设备结构体变量。

结合上设备号，补充一下 2.2.2 中的示例：

```
1.   /* 字符设备 */
2.   struct cdev ax_cdev = {
3.       .owner = THIS_MODULE,
4.   };
5.
6.   /* 设备操作函数 */
7.   static struct file_operations ax_fops = {
8.       .owner = THIS_MODULE,
9.       .open……
10.  };
11.
12.  /* 设备号 */
13.  cdev_t devid;
14.  /* 申请设备号 */
15.  alloc_chrdev_region(&devid, 0, 1, " xxx-dev");
16.
```

```
/* ax_cdev 变量初始化 */
cdev_init(&ax_cdev, &ax_fops);
/* 注册字符设备 */
cdev_add(&ax_cdev, devid, 1);
……
/* 卸载字符设备 */
cdev_del(&ax_cdev);
```

2.3 自动创建设备文件

2.3.1 mdev

medv 是一个用户程序，是 udev 的简化版。它可以检测并根据系统中硬件设备状态来创建或者删除设备文件。在加载驱动模块后，会自动在/dev 目录下创建设备节点文件，卸载驱动模块后设备节点也会自动删除。接下来看看如何实现。

2.3.2 类的创建和删除

创建设备前需要先创建类，设备是在类下面创建的。类的结构体 struct class 结构体定义在 include/linux/device.h 中，需要使用函数 class_create()来创建。class_create()是个宏定义：

```
#define class_create(owner, name) \
({ \
    static struct lock_class_key __key; \
    __class_create(owner, name, &__key); \
})

struct class *__class_create(struct module *owner, const char *name, struct lock_class_key *key)
```

展开后可已看出：

输入参数：

owner：至今出现的 owner 值都是 THIS_MODULE，这里任然不例外；

name：类的名字。

返回值：struct class 类型的结构体指针。

卸载驱动程序时需要删除类，使用函数 class_destroy()，原型如下：

void class_destroy(struct class *cls);

cls 为需要删除的类。

2.2.3 设备节点的创建和删除

创建类后，使用 device_create()函数在类下面创建设备，原型为：

```
struct device *device_create(struct class *class,
                             struct device *parent,
                             dev_t devt,
                             void * drvdata,
                             const char *fmt, ...)
```

参数说明：

class：上节介绍的类，设备会在这个类下创建；

parent：父设备，没有父设备的话填 NULL；

devt：设备号；

drvdata：设备可能会用到的数据，没有的话填 NULL；

fmt：设备名，比如当 fmt=axled 时，创建设备后就会生成/dev/axled 文件。

删除设备函数为：

void device_destroy(struct class *class, dev_t devt)

输入参数含义和上面一样。

自动创建设备节点的实现一般放在驱动入口函数中，结合上一节类的创建以及设备号，自动创建设备节点的实现示例如下：

```
struct class *class;      /* 类 */
struct device *device;    /* 设备 */
dev_t devid;              /* 设备号 */

/* 驱动入口函数 */
static int __init xxx_init(void)
{
    ……
    /* 申请设备号 */
    alloc_chrdev_region(&devid, 0, 1, "xxx-dev");
    ……
    /* 创建类 */
    class = class_create(THIS_MODULE, "xxx");
    /* 创建设备 */
    device = device_create(class, NULL, devid, NULL, "xxx");

    return 0;
}

/* 驱动出口函数 */
static void __exit led_exit(void)
{
    /* 删除设备 */
    device_destroy(class, devid);
    /* 删除类 */
    class_destroy(class);
    /* 注销字符设备 */
```

```
28.     unregister_chrdev_region(devid, 1);
29. }
30.
31. module_init(led_init);
32. module_exit(led_exit);
```

2.4 字符设备新驱动实验

现在了解字符设备驱动的新写法相关的要点，接下来就动手尝试，这章我们要达成的目标和上一章相同，编写开发板上”PS LED1”这个 led 的设备驱动，通过驱动程序能点亮、熄灭这个 led。

2.4.1 查看硬件原理图以及数据手册

和上一章 1.3.1 节的内容相同。

2.4.2 编写字符设备驱动程序

使用 petalinux 创建新的驱动，方法也和上一章相同，之后重复的步骤就略过了。这里我新建了名为 ax-newled-drv 的驱动。打开文件 ax-newled-drv.c，输入以下内容：

```
1.  #include <linux/module.h>
2.  #include <linux/kernel.h>
3.  #include <linux/fs.h>
4.  #include <linux/init.h>
5.  #include <linux/ide.h>
6.  #include <linux/types.h>
7.  #include <linux/errno.h>
8.  #include <linux/cdev.h>
9.  #include <linux/device.h>
10. #include <asm/uaccess.h>
11.
12. /* 设备节点名称 */
13. #define DEVICE_NAME       "gpio_leds"
14. /* 设备号个数 */
15. #define DEVID_COUNT       1
16. /* 设备个数 */
17. #define DEVICE_COUNT      1
18. /* 主设备号 */
19. #define MAJOR
20. /* 次设备号 */
21. #define MINOR             0
22.
```

```
/* gpio 寄存器虚拟地址 */
static unsigned int gpio_add_minor;
/* gpio 寄存器物理基地址 */
#define GPIO_BASE         0xE000A000
/* gpio 寄存器所占空间大小 */
#define GPIO_SIZE         0x1000
/* gpio 方向寄存器 */
#define GPIO_DIRM_0       (unsigned int *)(0xE000A204 - GPIO_BASE + gpio_add_minor)
/* gpio 使能寄存器 */
#define GPIO_OEN_0        (unsigned int *)(0xE000A208 - GPIO_BASE + gpio_add_minor)
/* gpio 控制寄存器 */
#define GPIO_DATA_0       (unsigned int *)(0xE000A040 - GPIO_BASE + gpio_add_minor)

/* 时钟使能寄存器虚拟地址 */
static unsigned int clk_add_minor;
/* 时钟使能寄存器物理基地址 */
#define CLK_BASE          0xF8000000
/* 时钟使能寄存器所占空间大小 */
#define CLK_SIZE          0x1000
/* AMBA 外设时钟使能寄存器 */
#define APER_CLK_CTRL     (unsigned int *)(0xF800012C - CLK_BASE + clk_add_minor)

/* 把驱动代码中会用到的数据打包进设备结构体 */
struct alinx_char_dev{
    dev_t            devid;      //设备号
    struct cdev      cdev;       //字符设备
    struct class     *class;     //类
    struct device    *device;    //设备节点
};
/* 声明设备结构体 */
static struct alinx_char_dev alinx_char = {
    .cdev = {
        .owner = THIS_MODULE,
    },
};

/* open 函数实现，对应到 Linux 系统调用函数的 open 函数 */
static int gpio_leds_open(struct inode *inode_p, struct file *file_p)
{
    /* 把需要修改的物理地址映射到虚拟地址 */
    gpio_add_minor = (unsigned int)ioremap(GPIO_BASE, GPIO_SIZE);
    clk_add_minor = (unsigned int)ioremap(CLK_BASE, CLK_SIZE);

    /* MIO_0 时钟使能 */
```

```
67.     *APER_CLK_CTRL |= 0x00400000;
68.     /* MIO_0 设置成输出 */
69.     *GPIO_DIRM_0 |= 0x00000001;
70.     /* MIO_0 使能 */
71.     *GPIO_OEN_0 |= 0x00000001;
72.
73.     printk("gpio_test module open\n");
74.
75.     return 0;
76. }
77.
78.
79. /* write 函数实现，对应到 Linux 系统调用函数的 write 函数 */
80. static ssize_t gpio_leds_write(struct file *file_p, const char __user *buf, size_t len, loff_t *lo
    ff_t_p)
81. {
82.     int rst;
83.     char writeBuf[5] = {0};
84.
85.     printk("gpio_test module write\n");
86.
87.     rst = copy_from_user(writeBuf, buf, len);
88.     if(0 != rst)
89.     {
90.         return -1;
91.     }
92.
93.     if(1 != len)
94.     {
95.         printk("gpio_test len err\n");
96.         return -2;
97.     }
98.     if(1 == writeBuf[0])
99.     {
100.        *GPIO_DATA_0 &= 0xFFFFFFFE;
101.        printk("gpio_test ON\n");
102.    }
103.    else if(0 == writeBuf[0])
104.    {
105.        *GPIO_DATA_0 |= 0x00000001;
106.        printk("gpio_test OFF\n");
107.    }
108.    else
109.    {
```

```
110.         printk("gpio_test para err\n");
111.         return -3;
112.     }
113.
114.     return 0;
115. }
116.
117. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
118. static int gpio_leds_release(struct inode *inode_p, struct file *file_p)
119. {
120.     /* 释放对虚拟地址的占用 */
121.     iounmap((unsigned int *)gpio_add_minor);
122.     iounmap((unsigned int *)clk_add_minor);
123.
124.     printk("gpio_test module release\n");
125.     return 0;
126. }
127.
128. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
129. static struct file_operations ax_char_fops = {
130.     .owner   = THIS_MODULE,
131.     .open    = gpio_leds_open,
132.     .write   = gpio_leds_write,
133.     .release = gpio_leds_release,
134. };
135.
136. /* 模块加载时会调用的函数 */
137. static int __init gpio_led_init(void)
138. {
139.     /* 注册设备号 */
140.     alloc_chrdev_region(&alinx_char.devid, MINOR, DEVID_COUNT, DEVICE_NAME);
141.
142.     /* 初始化字符设备结构体 */
143.     cdev_init(&alinx_char.cdev, &ax_char_fops);
144.
145.     /* 注册字符设备 */
146.     cdev_add(&alinx_char.cdev, alinx_char.devid, DEVICE_COUNT);
147.
148.     /* 创建类 */
149.     alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
150.     if(IS_ERR(alinx_char.class))
151.     {
152.         return PTR_ERR(alinx_char.class);
153.     }
```

```
154.
155.     /* 创建设备节点 */
156.     alinx_char.device = device_create(alinx_char.class, NULL,
157.                                       alinx_char.devid, NULL,
158.                                       DEVICE_NAME);
159.     if (IS_ERR(alinx_char.device))
160.     {
161.         return PTR_ERR(alinx_char.device);
162.     }
163.
164.     return 0;
165. }
166.
167. /* 卸载模块 */
168. static void __exit gpio_led_exit(void)
169. {
170.     /* 注销字符设备 */
171.     cdev_del(&alinx_char.cdev);
172.
173.     /* 注销设备号 */
174.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
175.
176.     /* 删除设备节点 */
177.     device_destroy(alinx_char.class, alinx_char.devid);
178.
179.     /* 删除类 */
180.     class_destroy(alinx_char.class);
181.
182.     printk("gpio_led_dev_exit_ok\n");
183. }
184.
185. /* 标记加载、卸载函数 */
186. module_init(gpio_led_init);
187. module_exit(gpio_led_exit);
188.
189. /* 驱动描述信息 */
190. MODULE_AUTHOR("Alinx");
191. MODULE_ALIAS("gpio_led");
192. MODULE_DESCRIPTION("NEW GPIO LED driver");
193. MODULE_VERSION("v1.0");
194. MODULE_LICENSE("GPL");
```

与上一章相比，改动的部分加粗了，除了新增了一些宏定义和参数声明之外，改动的只有入口函数和出口函数两个地方。

13~21 行，新增了几个宏定义，DEVICE_NAME 表示设备名，也是设备节点名，最终我们要到/dev 目录中去寻找这个名字的设备文件。

DEVID_COUNT 和 DEVICE_COUNT 分别表示设备号数和设备节点数。一般一个设备对应一个驱动，所以在一个驱动代码中就只有一个设备，所以这里都为 1。

MAJOR 主设备号，这里我们通过函数获取设备号，主设备号不需要自己设置。MAJOR 次设备号一般从 0 开始。

46~57 行，新建了一个结构体类型，并用这个类型声明了一个变量。把之后会用到的设备号、字符设备、类、设备节点等数据类型都打包进这个结构体中。这样在使用变量时会方便很多，也增加了可读性。并且在后面我们用到私有数据时，这个结构体也会带来遍历，具体在用到的时候再细讲。

140~162 行是关键的地方，在驱动入口函数中，把本章先后讲到的知识点申请设备号、初始化并注册字符设备、创建类和设备节点全部结合。

171~180 行出口函数中就是做与入口函数中注册创建相对应的注销和删除。

结合上面几节的内容，这个新驱动代码不难理解。完成后在 ubuntu 虚拟机重编译得到驱动模块文件 ax-newled-dev.ko。

2.4.3 编写测试 APP

测试 APP 与上一章 1.3.4 节内容一致，可以直接使用上一章的测试程序。

2.4.4 运行测试

测试方法也与之前一样，给开发板上电，并挂在虚拟机的工作目录到开发板/mnt 路径。

```
root@ax_peta:~# mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
```

1.先加载驱动，执行命令：

insmod ax-newled-dev.ko

```
ax_newled_dev: loading out-of-tree module taints kernel.
root@ax_peta:/mnt#
```

2.驱动加载成功，再看看设备文件有没有创建成功，执行命令：

ls /dev

```
root@ax_peta:/mnt# ls /dev
block               loop-control        mtd2ro              ram13
bus                 loop0               mtd3                ram14
char                loop1               mtd3ro              ram15
console             loop2               mtdblock0           ram2
cpu_dma_latency     loop3               mtdblock1           ram3
disk                loop4               mtdblock2           ram4
fd                  loop5               mtdblock3           ram5
full                loop6               network_latency     ram6
gpio_leds           loop7               network_throughput  ram7
gpiochip0           mem                 null                ram8
gpiochip1           memory_bandwidth    port                ram9
gpiochip2           mmcblk0             psaux               random
i2c-0               mmcblk0p1           ptmx                shm
i2c-1               mtab                pts                 snd
iio:device0         mtd0                ram0                stderr
initctl             mtd0ro              ram1                stdin
input               mtd1                ram10               stdout
kmsg                mtd1ro              ram11               tty
log                 mtd2                ram12               tty0
root@ax_peta:/mnt#
```

3.设备节点文件已经存在了，可以使用测试 APP 来试试驱动程序了，执行下列命令：

cd ./build-axleddev_test-ZYNQ-Debug/

./axleddev_test /dev/gpio_leds on

```
root@ax_peta:/mnt# cd ./build-axleddev_test-ZYNQ-Debug/
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# ./axleddev_test /dev/gpio_leds on
gpio_test module open
ps_led1 on
gpio_test module write
gpio_test ON
gpio_test module release
```

4.led 被点亮，最后在测试卸载驱动，执行命令：

rmmod ax-newled-dev

如果不确定驱动名称，可以先执行 lsmod 命令查看。

```
root@ax_peta:/mnt# rmmod ax-newled-dev
gpio_led_dev_exit_ok
root@ax_peta:/mnt#
```

5.删除设备后，再确认设备节点文件有没有被删除：

ls /dev/gpio_led

```
root@ax_peta:/mnt# ls /dev/gpio_led
ls: /dev/gpio_led: No such file or directory
root@ax_peta:/mnt#
```

没有问题，试验成功。