# 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. };
6. /\* 设备操作函数 \*/
7. **static** **struct** file\_operations ax\_fops = {
8. .owner = THIS\_MODULE,
9. .open……
10. };
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. };
6. /\* 设备操作函数 \*/
7. **static** **struct** file\_operations ax\_fops = {
8. .owner = THIS\_MODULE,
9. .open……
10. };
12. /\* 设备号 \*/
13. cdev\_t devid;
14. /\* 申请设备号 \*/
15. alloc\_chrdev\_region(&devid, 0, 1, " xxx-dev");
17. /\* ax\_cdev变量初始化 \*/
18. cdev\_init(&ax\_cdev, &ax\_fops);
19. /\* 注册字符设备 \*/
20. cdev\_add(&ax\_cdev, devid, 1);
21. ……
22. /\* 卸载字符设备 \*/
23. 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：

1. #define class\_create(owner, name) \
2. ({ \
3. **static** **struct** lock\_class\_key \_\_key; \
4. \_\_class\_create(owner, name, &\_\_key); \
5. })
7. **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：

1. **struct** device \*device\_create(**struct** **class** \***class**,
2. **struct** device \*parent,
3. dev\_t devt,
4. **void** \* drvdata,
5. **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：

1. **struct** **class** \***class**;      /\* 类 \*/
2. **struct** device \*device;    /\* 设备 \*/
3. dev\_t devid;              /\* 设备号 \*/
5. /\* 驱动入口函数 \*/
6. **static** **int** \_\_init xxx\_init(**void**)
7. {
8. ……
9. /\* 申请设备号 \*/
10. alloc\_chrdev\_region(&devid, 0, 1, "xxx-dev");
11. ……
12. /\* 创建类 \*/
13. **class** = class\_create(THIS\_MODULE, "xxx");
14. /\* 创建设备 \*/
15. device = device\_create(**class**, NULL, devid, NULL, "xxx");
17. **return** 0;
18. }
20. /\* 驱动出口函数 \*/
21. **static** **void** \_\_exit led\_exit(**void**)
22. {
23. /\* 删除设备 \*/
24. device\_destroy(**class**, devid);
25. /\* 删除类 \*/
26. class\_destroy(**class**);
27. /\* 注销字符设备 \*/
28. unregister\_chrdev\_region(devid, 1);
29. }
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>
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**
23. /\* gpio寄存器虚拟地址 \*/
24. **static** unsigned **int** gpio\_add\_minor;
25. /\* gpio寄存器物理基地址 \*/
26. #define GPIO\_BASE         0xE000A000
27. /\* gpio寄存器所占空间大小 \*/
28. #define GPIO\_SIZE         0x1000
29. /\* gpio方向寄存器 \*/
30. #define GPIO\_DIRM\_0       (unsigned int \*)(0xE000A204 - GPIO\_BASE + gpio\_add\_minor)
31. /\* gpio使能寄存器 \*/
32. #define GPIO\_OEN\_0        (unsigned int \*)(0xE000A208 - GPIO\_BASE + gpio\_add\_minor)
33. /\* gpio控制寄存器 \*/
34. #define GPIO\_DATA\_0       (unsigned int \*)(0xE000A040 - GPIO\_BASE + gpio\_add\_minor)
36. /\* 时钟使能寄存器虚拟地址 \*/
37. **static** unsigned **int** clk\_add\_minor;
38. /\* 时钟使能寄存器物理基地址 \*/
39. #define CLK\_BASE          0xF8000000
40. /\* 时钟使能寄存器所占空间大小 \*/
41. #define CLK\_SIZE          0x1000
42. /\* AMBA外设时钟使能寄存器 \*/
43. #define APER\_CLK\_CTRL     (unsigned int \*)(0xF800012C - CLK\_BASE + clk\_add\_minor)
45. **/\* 把驱动代码中会用到的数据打包进设备结构体 \*/**
46. **struct alinx\_char\_dev{**
47. **dev\_t            devid;      //设备号**
48. **struct cdev      cdev;       //字符设备**
49. **struct class     \*class;     //类**
50. **struct device    \*device;    //设备节点**
51. **};**
52. **/\* 声明设备结构体 \*/**
53. **static struct alinx\_char\_dev alinx\_char = {**
54. **.cdev = {**
55. **.owner = THIS\_MODULE,**
56. **},**
57. **};**
59. /\* open函数实现, 对应到Linux系统调用函数的open函数 \*/
60. **static** **int** gpio\_leds\_open(**struct** inode \*inode\_p, **struct** file \*file\_p)
61. {
62. /\* 把需要修改的物理地址映射到虚拟地址 \*/
63. gpio\_add\_minor = (unsigned **int**)ioremap(GPIO\_BASE, GPIO\_SIZE);
64. clk\_add\_minor = (unsigned **int**)ioremap(CLK\_BASE, CLK\_SIZE);
66. /\* 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;
73. printk("gpio\_test module open\n");
75. **return** 0;
76. }

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 \*loff\_t\_p)
81. {
82. **int** rst;
83. **char** writeBuf[5] = {0};
85. printk("gpio\_test module write\n");
87. rst = copy\_from\_user(writeBuf, buf, len);
88. **if**(0 != rst)
89. {
90. **return** -1;
91. }
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. }
114. **return** 0;
115. }
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);
124. printk("gpio\_test module release\n");
125. **return** 0;
126. }
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. };
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);**
142. **/\* 初始化字符设备结构体 \*/**
143. **cdev\_init(&alinx\_char.cdev, &ax\_char\_fops);**
145. **/\* 注册字符设备 \*/**
146. **cdev\_add(&alinx\_char.cdev, alinx\_char.devid,** **DEVICE\_COUNT);**
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. **}**
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. **}**
164. **return** 0;
165. }
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);**
173. **/\* 注销设备号 \*/**
174. **unregister\_chrdev\_region(alinx\_char.devid, DEVID\_COUNT);**
176. **/\* 删除设备节点 \*/**
177. **device\_destroy(alinx\_char.class, alinx\_char.devid);**
179. **/\* 删除类 \*/**
180. **class\_destroy(alinx\_char.class);**
182. printk("gpio\_led\_dev\_exit\_ok\n");
183. }
185. /\* 标记加载、卸载函数 \*/
186. module\_init(gpio\_led\_init);
187. module\_exit(gpio\_led\_exit);
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。

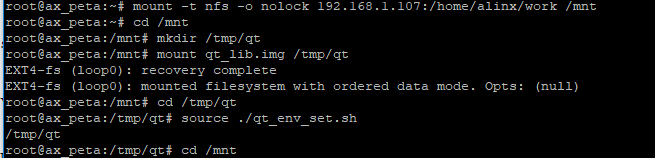
结合上面几节的内容，这个新驱动代码不难理解。完成后在ubuntu虚拟机重编译得到驱动模块文件ax-newled-dev.ko。

### 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。



1.先加载驱动，执行命令：

insmod ax-newled-dev.ko



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：

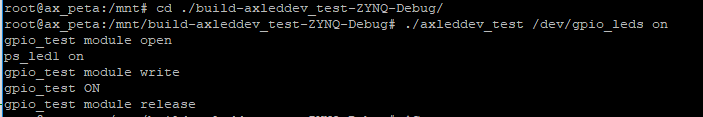
ls /dev



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



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。



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



没有问题，试验成功。