# 4 pinctrl和gpio子系统

在前面的实验中，我们操作设备都是直接操作寄存器的。虽然上一章使用了设备树，但也只是把寄存器信息放到设备树中，本质上都是一样的。直接操作寄存器来控制硬件资源对于编码的人来说很方便，然而对于阅读代码的人来说想必是很崩溃的。而且在设备树中写入大量的寄存器地址，不仅不易于阅读，在使用时也很容易出错。比如上一章中，仅是控制led就用了4个寄存器，而且寄存器地址在设备树中的顺序也没有规则，全是自定的，在驱动程序中读取地址也无法对应和确认，最终我们在设置寄存器时就很容易错位。

pinctrl子系统来可以应对这个问题，同时针对我们实验中用到的led也就是gpio操作来说还需要结合gpio子系统。

## 4.1 pinctrl子系统

Linux内核提供pinctrl子系统来统一芯片厂商的pin脚管理方法，他会帮我们完成以下工作：

1. 从设备树中获取pin脚信息；

2. 设置pin脚的复用功能；

3. 配置pin脚的电器特性。

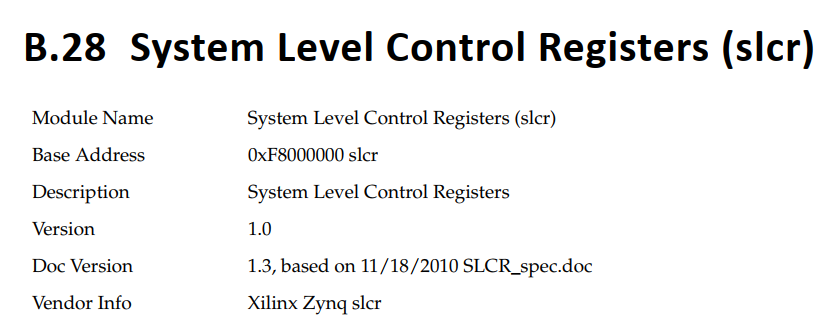
使用pinctrl子系统，只需要在设备树中配置好相关节点信息，剩余的工作全由pinctrl子系统完成，我们只要做甩手掌柜就行了。

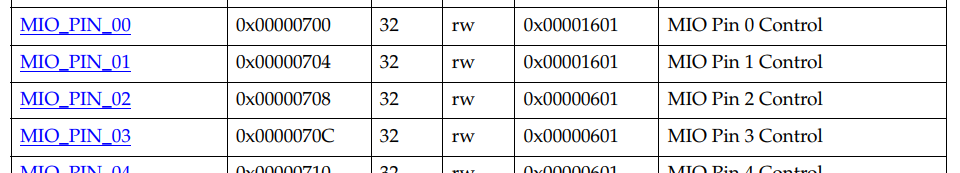
### 4.1.1 zynq平台pinctrl子系统的设备树节点和属性

不同的芯片pin脚配置方法有差别，相应的芯片厂家的pinctrl子系统实现也会有差别，但我们需要关心的仍然只有设备树的写法。直接来看一段示例：

1. pinctrl0: pinctrl@700 {
2. compatible = "xlnx,pinctrl-zynq";
3. reg = <0x700 0x200>;
4. syscon = <&slcr>;
5. pinctrl\_uart1\_default: uart1-default {
6. mux {
7. groups = "uart1\_10\_grp";
8. function = "uart1";
9. };
10. conf {
11. groups = "uart1\_10\_grp";
12. slew-rate = <0>;
13. io-standard = <1>;
14. };
15. conf-rx {
16. pins = "MIO49";
17. bias-high-impedance;
18. };
19. conf-tx {
20. pins = "MIO48";
21. bias-disable;
22. };
23. };
24. };

第一行的pinctrl0: pinctrl@700节点，就是ZYNQ中pinctrl子系统的节点。pinctrl0的父节点为slcr@f8000000，700是相对f8000000的偏移，也就说pin的寄存器起始地址为0xf8000700。可以在数据手册中找到对应的寄存器说明。





不过寄存相关的内容不是这里我们需要关系的对象，示例的2、3、4行是pinctrl0的三个属性，都是xilinx设置好的，也不用去关心。重点是第5行开始，在pinctrl0节点中添加的子设备节点uart1-default，我们要通过学习这个子设备节点的配置方法，添加我们自己需要的设备节点。这个节点下面没有属性，而是有4个子节点：mux、conf、conf-rx、conf-

tx。其中mux是配置pin脚的复用功能，其他三个是配置pin脚的电气特性，conf中设置通用的特性，conf-rx中设置输入引脚和相关特性，conf-tx中设置输出引脚和相关特性。

**mux**节点中，有两个必要属性：

**groups**：选择这个复用功能的pin脚分组列表。可选的值有：

*ethernet0\_0\_grp, ethernet1\_0\_grp, mdio0\_0\_grp, mdio1\_0\_grp, qspi0\_0\_grp, qspi1\_0\_grp, qspi\_fbclk, qspi\_cs1\_grp, spi0\_0\_grp - spi0\_2\_grp, spi0\_X\_ssY (X=0..2, Y=0..2), spi1\_0\_grp - spi1\_3\_grp, spi1\_X\_ssY (X=0..3, Y=0..2), sdio0\_0\_grp - sdio0\_2\_grp, sdio1\_0\_grp - sdio1\_3\_grp, sdio0\_emio\_wp, sdio0\_emio\_cd, sdio1\_emio\_wp, sdio1\_emio\_cd, smc0\_nor, smc0\_nor\_cs1\_grp, smc0\_nor\_addr25\_grp, smc0\_nand, can0\_0\_grp - can0\_10\_grp, can1\_0\_grp - can1\_11\_grp, uart0\_0\_grp - uart0\_10\_grp, uart1\_0\_grp - uart1\_11\_grp, i2c0\_0\_grp - i2c0\_10\_grp, i2c1\_0\_grp - i2c1\_10\_grp, ttc0\_0\_grp - ttc0\_2\_grp, ttc1\_0\_grp - ttc1\_2\_grp, swdt0\_0\_grp - swdt0\_4\_grp, gpio0\_0\_grp - gpio0\_53\_grp, usb0\_0\_grp, usb1\_0\_grp。*

这些分组包含哪些pin脚，可以在drivers/pinctrl/pinctrl-zynq.c文件中找到相应的数组，比如示例中的group”uart1\_10\_grp”，对应的数组为static const unsigned int uart1\_10\_pins[] = {48, 49}，这个group包含的pin脚即为pin48和pin49。

**function**：选择pin脚分组的复用功能。可选的值有：

*ethernet0, ethernet1, mdio0, mdio1, qspi0, qspi1, qspi\_fbclk, qspi\_cs1, spi0, spi0\_ss, spi1, spi1\_ss, sdio0, sdio0\_pc, sdio0\_cd, sdio0\_wp, sdio1, sdio1\_pc, sdio1\_cd, sdio1\_wp, smc0\_nor, smc0\_nor\_cs1, smc0\_nor\_addr25, smc0\_nand, can0, can1, uart0, uart1, i2c0, i2c1, ttc0, ttc1, swdt0, gpio0, usb0, usb1。*

示例中的设备节点名为uart1-default，用作uart1，所以function的值是uart1。

需要什么功能，可以用什么引脚，可以从数据手册中查看，也可以直接在drivers/pinctrl/pinctrl-zynq.c文件中查看，很容易看出对应关系。

**conf**节点可以分为多组，因为相同复用功能的pin组里每个pin的电器特性可能会不同，比如示例中的uart1，有一个pin脚用作输出不需要设置偏置，另一个用做输入需要设置成高阻。conf节点的名字必须要包含”conf”这个关键词，其他的关键词仅用于区分分组含义，比如示例中的conf-rx是指这个conf节点适用于配置输入pin脚的，命名为conf-in也是同样的效果，甚至把示例中rx和tx交换，对使用也没用影响，仅用于含以上的区分。

**conf**节点的必要属性为**pin**和**group**二选一。设置pin组时使用属性group，如示例中第10行的conf节点，设置单个pin或多个不在一个group中的pin时使用属性**pin**，如15行的conf-rx和19行的conf-tx。

group属性设上面mux节点中的一样。

**pin**：使用这个conf的pin脚列表。可选的值为：MIO0 - MIO53。

conf节点还有一些可选属性：

**io-standard**：设置io电平标准，可选值为：

1: LVCMOS18

2: LVCMOS25

3: LVCMOS33

4: HSTL

**bias-disable、bias-high-impedance、bias-pull-up**：设置pin脚偏置，不需要赋值，只需要在对应的conf节点中包含需要的偏置属性即可。

**slew-rate**：设置转换速率，等于0时为slow，等于1时为fast。

**low-power-disable、low-power-enable**：使能低功耗模式，在对应的conf节点中包含需要的偏置属性即可。

还有其他的可选属性不就一一介绍了，可以在内核源码目录Documentation/devicetree/bindings/pinctrl/pinctrl-binding.txt文件中，搜索”== Generic pin configuration node content ==”词条查看。

### 4.1.2 pinctrl子系统的使用

在pinctrl节点中添加了设备子节点后，系统还不回去初始化这个设备，我们需要在客户设备(client device)节点中去调用他。举个例子：

1. device {
2. pinctrl-names = "active", "idle";
3. pinctrl-0 = <&state\_0\_node\_a>;
4. pinctrl-1 = <&state\_1\_node\_a &state\_1\_node\_b>;
5. };

**device**是客户设备节点，是调用pinctrl的设备。

**pinctrl-0**是必选属性，是这个设备对应pin的pinctrl节点应用列表，如果这个设备对应了多个pin，那就增加属性**pinctrl-0~n**，如果一个pin对应了多个pinctrl，就如第4行的pinctrl-1属性一般附上多个pinctrl节点。

**pinctrl-names**是下面pinctrl0~n的名称列表，命名是自定的但要尽量易读并贴合实际。

## 4.2 gpio子系统

pinctrl子系统中，如果把功能配置成gpio的话，就需要用到gpio子系统了。gpio子系统帮我们实现了gpio初始化，并提供了一些接口函数给我们操作gpio。我们要做的同样只是在设备树中配置好信息，然后了解一下接口函数的用法即可。

### 4.2.1 设备树中的gpio

在Documentation/devicetree/bindings/gpio/gpio-zynq.txt文件中，介绍了zynq系列芯片gpio在设备数中的描述方式，我们在现有的zynq设备数中，也能找到gpio的默认配置，如下：

1. gpio0:gpio@e000a000 {
2. #gpio-cells = <2>;
3. compatible = "xlnx,zynq-gpio-1.0";
4. clocks = <&clkc 42>;
5. gpio-controller;
6. interrupt-parent = <&intc>;
7. interrupts = <0 20 4>;
8. interrupt-controller;
9. #interrupt-cells = <2>;
10. reg = <0xe000a000 0x1000>;
11. };

这些属性都是必要属性。

interrupt-parent、interrupts、interrupt-controller、#interrupt-cells：这四个属性是中断相关的，等之后讲中断的时候再说。

clocks属性时时钟设置相关，暂时使用默认配置即可。

**#gpio-cells**：指代引用这个gpio节点时需要两个cell，他的值必须是2，即引用节点时需要有两个cell，第一个cell为gpio编号，第二个cell用于指顶可选参数，第二个cell在zynq中没有用到填0即可。举个例子：alinxled-gpios = <&gpio0 0 0>;，即客户设备的使用的gpio0\_mio0这个io。

**compatible**：兼容新，必须为”xlnx, zynq-gpio-1.0”或”xlnx, zynqmp-gpio-1.0”。

**gpio-controller**：指定该节点为gpio控制器。

**reg**：这组gpio的寄存器首地址和范围。

这里我们不需要修改什么，保持现有配置即可。

### 4.2.2 gpio接口函数

gpio子系统提供接口函数来代替我们直接对寄存器的操作把驱动程序再次分离分层。这节我们介绍一些常用的gpio接口函数。

#### 4.2.2.1 of函数扩展

在介绍gpio接口函数之前，先扩展一个gpio相关的of函数of\_get\_named\_gpio，用于通过设备树中节点属性如”gpios = <&gpio0 0 0 &gpio0 1 0>;”获取gpio编号，函数原型如下：

**int of\_get\_named\_gpio(struct device\_node \*np, const char \*propname, int index)**

参数说明：

**np**：设备节点。

**propname**：属性名称，从这个属性中获取gpio编号。

**index**：需要获取的gpio下标。

**返回值**：gpio编号；返回负值则获取失败。

#### 4.2.2.2 gpio\_request()

在使用gpio做其他操作之前，需要先用gpio\_request()函数申请一个io。至今为止接触到的Linux内核提供的接口，都是这种面向对象的思想，先声明一个对象，再操作这个对象。

函数原型：

**int gpio\_request(unsigned int gpio, const char \*label)**

参数说明：

**gpio**：需要申请的gpio标号，通过上面介绍的of\_get\_named\_gpio函数获取。

**label**：申请到的gpio的标签。

**返回值**：0表示成功，其他表示失败。

#### 4.2.2.3 gpio\_free()

释放gpio，与gpio\_request()相对，原型：

**void gpio\_free(unsigned int gpio)**

**gpio**：需要释放的gpio标号。

#### 4.2.2.4 gpio\_direction\_input()

设置gpio为输入，函数原型：

**int gpio\_direction\_input(unsigned gpio)**

**gpio**：需要设置的gpio标号。

**返回值**：0表示成功，负值表示失败。

#### 4.2.2.5 gpio\_direction\_output()

设置gpio为输出，函数原型：

**int gpio\_direction\_output(unsigned gpio, int value)**

**gpio**：需要设置的gpio标号。

**value**：输出的默认电平。

**返回值**：0表示成功，负值表示失败。

#### 4.2.2.6 gpio\_get\_value()

读取gpio的电平，原型：

**int \_\_gpio\_get\_value(unsigned gpio)**

**gpio**：需要读取的gpio标号。

**返回值**：读取到的电平，负值表示失败。

#### 4.2.3.6 gpio\_set\_value()

设置gpio的电平，原型：

**void \_\_gpio\_set\_value(unsigned gpio, int value)**

**gpio**：需要操作的gpio标号。

**value**：需要写入的电平。

## 4.3实验

必要的知识都了解了，接下来通过实验来验证。仍然使用led来测试。

### 4.3.1 原理图

和章节1.3.1的内容相同。

### 4.3.2 设备树

打开文件”system-conf.dtsi”，在根节点下添加以下节点内容：

1. amba {
2. gpio@e000a000 {
3. compatible = "xlnx,zynq-gpio-1.0";
4. #gpio-cells = <0x2>;
5. clocks = <0x1 0x2a>;
6. gpio-controller;
7. interrupt-controller;
8. #interrupt-cells = <0x2>;
9. interrupt-parent = <0x4>;
10. interrupts = <0x0 0x14 0x4>;
11. reg = <0xe000a000 0x1000>;
12. };
14. slcr@f8000000 {
15. pinctrl@700 {
16. pinctrl\_led\_default: led-**default** {
17. mux {
18. groups = "gpio0\_0\_grp";
19. function = "gpio0";
20. };
21. conf {
22. pins = "MIO0";
23. io-standard = <1>;
24. bias-disable;
25. slew-rate = <0>;
26. };
27. };
28. };
29. };
30. };
32. alinxled {
33. compatible = "alinx-led";
34. pinctrl-names = "default";
35. pinctrl-0 = <&pinctrl\_led\_default>;
36. alinxled-gpios = <&gpio0 0 0>;
37. };

第**2~12**行是gpio节点，这个内容其实是zynq设备树中已有的，内容4.2.1节讲过了。

第**16~28**行我们新增的pin设备节点，用于配置led用到的pin脚。led链接的io是MIO0，所在分组为"gpio0\_0\_grp"，功能就是gpio0。因为只有一个引脚，所以只要一个conf节点就行了，转换率设置为slow，电平标准为1.8，偏置失能，配置目标为”MIO0”。

第**33**行则是我们的led设备节点，对于gpio子系统和pinctrl子系统来说，就是客户设备(client device)，通过属性pinctrl-0 = <&pinctrl\_led\_default>调用pin节点，通过属性alinxled-gpios = <&gpio0 0 0>来绑定gpio。这个属性的名字alinxled-gpios需要记号，稍后我们要通过这个名字来获取gpio标号。

### 4.3.3 驱动代码

使用petalinux新建名为”ax-pinioled-dev”的驱动程序，并执行petalinux-config -c rootfs命令选上新增的驱动程序。

在ax- pinioled -dev.c文件中输入下面的代码：

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/of.h>
10. #include <linux/of\_address.h>
11. #include <linux/of\_gpio.h>
12. #include <linux/device.h>
13. #include <linux/delay.h>
14. #include <linux/init.h>
15. #include <linux/gpio.h>
16. #include <asm/uaccess.h>
17. #include <asm/mach/map.h>
18. #include <asm/io.h>
20. /\* 设备节点名称 \*/
21. #define DEVICE\_NAME       "gpio\_leds"
22. /\* 设备号个数 \*/
23. #define DEVID\_COUNT       1
24. /\* 驱动个数 \*/
25. #define DRIVE\_COUNT       1
26. /\* 主设备号 \*/
27. #define MAJOR1
28. /\* 次设备号 \*/
29. #define MINOR1            0
30. /\* LED点亮时输入的值 \*/
31. #define ALINX\_LED\_ON      1
32. /\* LED熄灭时输入的值 \*/
33. #define ALINX\_LED\_OFF     0

36. /\* 把驱动代码中会用到的数据打包进设备结构体 \*/
37. **struct** alinx\_char\_dev{
38. dev\_t              devid;             //设备号
39. **struct** cdev        cdev;              //字符设备
40. **struct** **class**       \***class**;            //类
41. **struct** device      \*device;           //设备
42. **struct** device\_node \*nd;               //设备树的设备节点
43. **int                alinx\_led\_gpio;    //gpio号**
44. };
45. /\* 声明设备结构体 \*/
46. **static** **struct** alinx\_char\_dev alinx\_char = {
47. .cdev = {
48. .owner = THIS\_MODULE,
49. },
50. };
52. /\* open函数实现, 对应到Linux系统调用函数的open函数 \*/
53. **static** **int** gpio\_leds\_open(**struct** inode \*inode\_p, **struct** file \*file\_p)
54. {
55. /\* 设置私有数据 \*/
56. file\_p->private\_data = &alinx\_char;
57. **printk("gpio\_test module open\n");**
58. **return** 0;
59. }

62. /\* write函数实现, 对应到Linux系统调用函数的write函数 \*/
63. **static** ssize\_t gpio\_leds\_write(**struct** file \*file\_p, **const** **char** \_\_user \*buf, **size\_t** len, loff\_t \*loff\_t\_p)
64. {
65. **int** retvalue;
66. unsigned **char** databuf[1];
67. /\* 获取私有数据 \*/
68. **struct** alinx\_char\_dev \*dev = file\_p->private\_data;
69. retvalue = copy\_from\_user(databuf, buf, len);
70. **if**(retvalue < 0)
71. {
72. printk("alinx led write failed\r\n");
73. **return** -EFAULT;
74. }
76. **if**(databuf[0] == ALINX\_LED\_ON)
77. {
78. **gpio\_set\_value(&alinx\_char.alinx\_led\_gpio, !!0);**
79. }
80. **else** **if**(databuf[0] == ALINX\_LED\_OFF)
81. {
82. **gpio\_set\_value(&alinx\_char.alinx\_led\_gpio, !!1);**
83. }
84. **else**
85. {
86. printk("gpio\_test para err\n");
87. }
89. **return** 0;
90. }
92. /\* release函数实现, 对应到Linux系统调用函数的close函数 \*/
93. **static** **int** gpio\_leds\_release(**struct** inode \*inode\_p, **struct** file \*file\_p)
94. {
95. printk("gpio\_test module release\n");
96. **return** 0;
97. }
99. /\* 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 \*/
100. **static** **struct** file\_operations ax\_char\_fops = {
101. .owner   = THIS\_MODULE,
102. .open    = gpio\_leds\_open,
103. .write   = gpio\_leds\_write,
104. .release = gpio\_leds\_release,
105. };
107. /\* 模块加载时会调用的函数 \*/
108. **static** **int** \_\_init gpio\_led\_init(**void**)
109. {
110. /\* 用于接受返回值 \*/
111. u32 ret = 0;
113. **/\* 获取设备节点 \*/**
114. **alinx\_char.nd = of\_find\_node\_by\_path("/alinxled");**
115. **if(alinx\_char.nd == NULL)**
116. **{**
117. **printk("alinx\_char node not find\r\n");**
118. **return -EINVAL;**
119. **}**
120. **else**
121. **{**
122. **printk("alinx\_char node find\r\n");**
123. **}**
125. **/\* 获取节点中gpio标号 \*/**
126. **alinx\_char.alinx\_led\_gpio = of\_get\_named\_gpio(alinx\_char.nd, "alinxled-gpios", 0);**
127. **if(alinx\_char.alinx\_led\_gpio < 0)**
128. **{**
129. **printk("can not get alinxled-gpios");**
130. **return -EINVAL;**
131. **}**
132. **printk("alinxled-gpio num = %d\r\n", alinx\_char.alinx\_led\_gpio);**
134. **/\* 申请gpio标号对应的引脚 \*/**
135. **gpio\_request(alinx\_char.alinx\_led\_gpio, "alinxled");**
136. **if(ret != 0)**
137. **{**
138. **printk("can not request gpio\r\n");**
139. **}**
141. **/\* 把这个io设置为输出 \*/**
142. **ret = gpio\_direction\_output(alinx\_char.alinx\_led\_gpio, 1);**
143. **if(ret < 0)**
144. **{**
145. **printk("can not set gpio\r\n");**
146. **}**
148. /\* 注册设备号 \*/
149. alloc\_chrdev\_region(&alinx\_char.devid, MINOR1, DEVID\_COUNT, DEVICE\_NAME);
151. /\* 初始化字符设备结构体 \*/
152. cdev\_init(&alinx\_char.cdev, &ax\_char\_fops);
154. /\* 注册字符设备 \*/
155. cdev\_add(&alinx\_char.cdev, alinx\_char.devid, DRIVE\_COUNT);
157. /\* 创建类 \*/
158. alinx\_char.**class** = class\_create(THIS\_MODULE, DEVICE\_NAME);
159. **if**(IS\_ERR(alinx\_char.**class**))
160. {
161. **return** PTR\_ERR(alinx\_char.**class**);
162. }
164. /\* 创建设备节点 \*/
165. alinx\_char.device = device\_create(alinx\_char.**class**, NULL,
166. alinx\_char.devid, NULL,
167. DEVICE\_NAME);
168. **if** (IS\_ERR(alinx\_char.device))
169. {
170. **return** PTR\_ERR(alinx\_char.device);
171. }
173. **return** 0;
174. }
176. /\* 卸载模块 \*/
177. **static** **void** \_\_exit gpio\_led\_exit(**void**)
178. {
179. **/\* 释放gpio \*/**
180. **gpio\_free(alinx\_char.alinx\_led\_gpio);**
182. /\* 注销字符设备 \*/
183. cdev\_del(&alinx\_char.cdev);
185. /\* 注销设备号 \*/
186. unregister\_chrdev\_region(alinx\_char.devid, DEVID\_COUNT);
188. /\* 删除设备节点 \*/
189. device\_destroy(alinx\_char.**class**, alinx\_char.devid);
191. /\* 删除类 \*/
192. class\_destroy(alinx\_char.**class**);
194. printk("gpio\_led\_dev\_exit\_ok\n");
195. }
197. /\* 标记加载、卸载函数 \*/
198. module\_init(gpio\_led\_init);
199. module\_exit(gpio\_led\_exit);
201. /\* 驱动描述信息 \*/
202. MODULE\_AUTHOR("Alinx");
203. MODULE\_ALIAS("gpio\_led");
204. MODULE\_DESCRIPTION("PINCTRL AND GPIO LED driver");
205. MODULE\_VERSION("v1.0");
206. MODULE\_LICENSE("GPL");

与上一章驱动代码相比有却别的部分加粗了，主要是在入口函数中用gpio子系统获取gpio并初始化替换了原先获取寄存器再初始化的操作。

**43**行在设备结构体中定义了gpio标号便于后续的获取操作。

**55**行原先通过寄存器初始化io的操作删除了。

**56**行在open函数中设置私有数据，之后在write函数中可以获取使用。

**68**行获取私有数据。

**79**和**83**行使用gpio\_set\_value函数来设置gpio的电平，用获取的私有数据应用gpio标号，第二个输入参数用!!二值化。

**115**行获取/alinxled节点。

**127**行使用of\_get\_named\_gpio函数，通过属性alinxled-gpios获取gpio标号。

**136**行使用gpio\_request函数申请标号对应的引脚。

**143**行使用gpio\_direction\_output函数把gpio设置为输出。

**181**行在卸载模块时，增加gpio释放操作。

### 4.3.4 测试代码

测试APP与章节1.3.4内容一致，可以直接使用。

### 4.3.1 运行测试

因为APP相同，所以测试方法任然相同，只要能成功点亮led就成功了。

