# 3 设备树和of函数

## 3.1 设备树

ARM Linux3.0以前是没有应用设备树的，那时ARM Linux使用者们用c语言的数据结构来描述板子上的设备，每一个设备就对应到一个描述文件。ARM的板子种类不计其数，而Linux社区为了保证内核源码的通用性，把这些描述文件全部塞了进去，导致Linux内核充满了垃圾代码，震怒了Linux之父Linus。之后ARM社区就学习PowerPC引入了设备树来替代以前的方法。

设备树除了没有用于代码之外还有其他优势。在驱动对应的硬件有变动时，不需要重新编译内核或驱动程序，只需要单独编译设备树文件，在启动板卡时把设备树便宜结果传给内核即可。（遗憾的是，这个优势在petalinux中没有得到体现，petalinux编译出来的结果中，u-boot、内核和设备树被一并打包进了image.ub里，而不是单独提供的。所以变更设备树时，往往会使用petalinux-build命令把内核整体重新编译。其实通过前面两章可以发现，在更改或者新增驱动时，也是整体编译内核的。实际上单独编译驱动是可以的，但是，为了统一操作步骤就都用petalinux-build命令了。）

设备树即树状的设备结构，“树”是个形象的比喻，以系统总线为主干，其他挂在在系统总线上的如I2C、SPI、GPIO控制器等设备为分支，而这些挂载在主干上的分支又有他们自身挂载的设备，如I2C上挂载了EEPROM、RTC等设备，这样的枝干分叉结构就如同树一般。

### 3.1.1 DTS、DTB和DTSI

通常我们编辑的设备树文件扩展名为.dtc(device tree source)，而内核读取的设备树文件为编译之后得到的二进制文件.dtb(device tree binary large object)。DTB文件我们也可以直接修改，但是二进制文件修改很不方便，也不是设备树设计的初衷，这里就重点说DTS文件。

DTS文件内容乍一看有点像JSON数据结构，大括号里大括号。当然肯定是有差别的，DTS有自己的一套语法规则，详情可参考官方文档”devicetree-specification-v0.2.pdf”。

先来看一段DTS文件内容，文件arch\arm\boot\dts\zynq-zc702.dts的节选：

1. #include "zynq-7000.dtsi"
3. / {
4. model = "Zynq ZC702 Development Board";
5. compatible = "xlnx,zynq-zc702", "xlnx,zynq-7000";
7. ……
9. memory@0 {
10. device\_type = "memory";
11. reg = <0x0 0x40000000>;
12. };
14. ……
16. gpio-keys {
17. compatible = "gpio-keys";
18. #address-cells = <1>;
19. #size-cells = <0>;
20. autorepeat;
21. sw14 {
22. label = "sw14";
23. gpios = <&gpio0 12 0>;
24. linux,code = <108>; /\* down \*/
25. wakeup-source;
26. autorepeat;
27. };
29. ……
30. };
31. };

结合这段代码，来大致了解一下DTS的语法格式。

第**1**行是包含头文件的语句。和c语言一样，DTS文件使用#include来引用头文件，可以引用.h、.dts以及.dtsi文件。”.dtsi”文件是专门用于编写DTS头文件的文件，语法与DTS文件相同，是编写DTS头文件的首选。

### 3.1.2 节点

#### 3.1.2.1 通用节点格式

DTS文件中，设备用节点来表示，格式为：

1. [label:] node-name[@unit-address] {
2. [properties definitions]
3. [child nodes]
4. };

**[]**中的部分是非必要项。

**node-name**是设备节点名称，根节点用”/”表示。第三行的”/”即是设备"Zynq ZC702 Development Board"的根节点。第9行的memory、第11行的gpio-keys、第21行的sw14都是设备节点名称。

**unit-address**一般是指代设备地址或寄存器首地址，如代码中第**9**行的 memory@0。如果设备没有设备地址或寄存器就不用写。

**label**是设备别称，用于便捷访问节点。如一个名称为slcr@f8000000的节点，正常要访问的话，需要用名称slcr@f8000000去访问，如果给这个节点加上标签slcr1: slcr@f8000000，访问节点只需要使用&slcr1即可。

**{}**里的是节点的内容，节点内容有两类，[properties definitions]是节点属性，后面再讲解。[child nodes]是这个挂在在这个设备节点上的子节点。子节点的格式和上面介绍的一样。

#### 3.1.2.2 特殊节点aliases

aliases节点用于定义别名，作用于label标签相似，格式如下：

1. aliases {
2. ethernet0 = "&gem0";
3. serial0 = "&uart1";
4. };

之后便可以使用&gem来访问节点ethernet。

### 3.1.3 属性

节点属性[properties definitions]有4种形式：

**1.** [label:]property-name;

属性为空值。如示例代码中**20**行和**26**行的autorepeat。

**2.** [label:]property-name = <arrays of cells>;

用<>括起来的值内容是32位数据的合集。如示例代码11行的reg = <0x0 0x40000000>。

**3.** [label:]property-name = “string”;

用””包含的表示字符串，如第五行的compatible = "xlnx,zynq-zc702", "xlnx,zynq-7000"。

**4.** [label:]property-name = [bytestring];

用[]括起来的表示字符序列，这个比较少见，手动举个例子，假设有属性表示为memory-addr = [0011223344];，这个就等同于memory-addr = [00 11 22 33 44];他的值是5个byte型的数组成的序列，并且这个这个byte型的数是两位16进制数组成的。

属性可以用户自定义，也有很多标准属性，下面介绍几种常见的标准属性。

#### 3.1.3.1 compatible属性

compatible属性也叫兼容性，他的值是字符串列表，是设备和驱动关联的关键，在驱动代码中定义一个OF匹配表来和compatible属性对应。如示例代码17行，节点gpio-keys的 compatible = "gpio-keys"，那么在对应的驱动代码中，就会有以下定义来与之对应：

1. **static** **const** **struct** of\_device\_id gpiokeys\_ids[] = {
2. { .compatible = "gpio-keys", },
3. { /\* sentinel \*/ }
4. };

**struct of\_device\_id**是OF匹配表的数据类型。当驱动程序OF匹配表中的compatible 值与设备树节点中的compatible 对应时，这个节点就会使用这个驱动程序。不过在目前我们只用到了字符设备的框架，这个框架还用不上OF匹配表，等后面我们用到总线设备模型时，会再讲到OF匹配表。

根节点中的compatible 属性表示这个板子兼容哪些平台。一般有两个值，前者表示板子的型号，后者表示使用的芯片。

#### 3.1.3.2 model属性

model属性的值也是字符串，一般用来表示板子的名称，与根节点中的compatible属性类似。

#### 3.1.3.3 #address-cells、#size-cells和reg属性

#address-cells属性表示当前节点**子节点**的reg属性中，使用多少个u32整数来描述地址address。

#size-cells属性表示当前节点**子节点**的reg属性中, 使用多少个u32整数来描述大小length。

reg属性一般用于描述某个外设的寄存器地址范围信息，格式为reg = <address1 length1 address2 length2 address3 length3……>。**父节点**中的#address-cells属性即指代reg属性中address的大小，#size-cells即指代reg属性中length的大小。

这个在示例代码中也没体现，为了便于理解，举个极端一点的例子：

1. ax-parent {
2. #address-cells = <2>;
3. #size-cells = <1>;
4. ……
6. ax-son {
7. reg = <0x00000001 0x00000002 0x00000003
8. 0x00000004 0x00000005 0x00000006>;
9. ……
10. }
11. }

子节点ax-son的reg属性的有6个u32的数据，父节点ax-parent中#address-cells等于2，因此子节点ax-son的reg属性的值中表示address的有两个u32的数，即0x00000001、0x00000002这两个数据都是address的值。同理#size-cells等于1，所以length1的值仅有一个u32的数据等于0x00000003。而后面的三个u32数据则是address2和length2的值。

#### 3.1.3.4 device\_type属性

这个属性现在只能用于 cpu 节点或者 memory 节点。在cpu节点中device\_type = “cpu”，在memory节点中device\_type = “memory”。

#### 3.1.3.5 phandle属性

phandle属性的取值必须是唯一的，他的作用与label标签相似，用来引用节点。

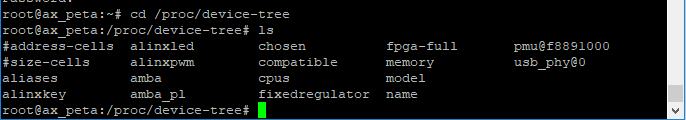
1. ax-node-1 {
2. phandle = <1>;
3. interrupt-controller;
4. }
6. ax-node-2 {
7. interrupt-parent = <1>;
8. }

在节点ax-node-1中有phandle属性为<1>，在ax-node-2中interrupt-parent属性需要指定父节点，赋值为<1>即可。

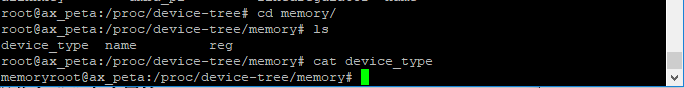
标准属性先介绍这些，还有一些特殊的以后用到在说。

### 3.1.4 在文件系统中查看设备树

内核启动时会解析DTB文件中的节点信息，并在根文件系统的/proc/devicetree目录下创建个节点对应的文件夹。



进入到对应的节点中，能查看到节点的各个属性。

****

### 3.1.5 修改设备树

设备树本身有一定的标准，不同的芯片厂家对于设备树有一些不同的自定义标准，我们在修改设备树时，有些需要遵循这些标准，但我们却不知道标准时，可以在内核源码目录/Documentation/devicetree/bindings中查看说明和指导。如果找不到，那就只能咨询芯片厂家了。

## 3.2 of函数

Linux内核提供了of函数来让我们获取设备树中的信息，of之名来自这些函数的前缀”of\_”。of函数的原型定义在内核目录include/linux/of.h中。这节我们介绍一些常用的of函数，没有介绍的等用到的时候再去了解也不迟。

### 3.2.1 查找节点的of函数

#### 3.2.1.1 device\_node结构体

device\_node结构体也定义在include/linux/of.h中，作为查找节点的of函数的返回值，用于给内核描述设备节点。

#### 3.2.1.2 of\_find\_node\_by\_name函数

of\_find\_node\_by\_name ()通过节点名查找节点，函数原型为：

struct device\_node \*of\_find\_node\_by\_name(struct device\_node \*from, const char \*name);

参数说明：

**from**：从这个节点开始查找，输入NULL时从根节点开始查找。

**name**：目标节点名称。

**返回值**：找到目标节点返回device\_node结构体。没有找到时返回NULL。

#### 3.2.1.3 of\_find\_node\_by\_type函数

of\_find\_node\_by\_type ()通过device\_type属性查找节点，函数原型为：

struct device\_node \*of\_find\_node\_by\_type(struct device\_node \*from, const char \*type);

参数说明：

**from**：从这个节点开始查找，输入NULL时从根节点开始查找。

**type**：device\_type 属性值。

**返回值**：找到目标节点返回device\_node结构体。没有找到时返回NULL。

#### 3.2.1.4 of\_find\_compatible\_node函数

of\_find\_compatible\_node ()通过device\_type和compatible属性查找节点，函数原型为：

struct device\_node \*of\_find\_compatible\_node(struct device\_node \*from, const char \*type, const char \*compatible);

参数说明：

**from**：从这个节点开始查找，输入NULL时从根节点开始查找。

**type**：device\_type 属性值，输入NULL时忽略。

**compatible**：compatible 属性值。

**返回值**：找到目标节点返回device\_node结构体。没有找到时返回NULL。

#### 3.2.1.5 of\_find\_node\_by\_path函数

of\_find\_node\_by\_path ()通过节点路径查找节点，函数原型为：

struct device\_node \*of\_find\_node\_by\_path(const char \*path);

参数说明：

**path**：节点的完整路径，以3.1.1节中的示例代码为例，21行的sw14设备的完整路径为/gpio-keys/sw14。

**返回值**：找到目标节点返回device\_node结构体。没有找到时返回NULL。

### 3.2.2 提取属性的of函数

#### 3.2.2.1 property结构体

property结构体同样也定义在include/linux/of.h中，作为提取属性的of函数的返回值，用于给内核描述节点属性。

#### 3.2.2.2 of\_find\_property函数

of\_find\_property 函数通过设备节点、属性名、属性值的大小查找属性，函数原型：

property \*of\_find\_property(const struct device\_node \*np, const char \*name, int \*len);

参数说明：

**np**：设备节点。

**name**：目标属性名。

**len**：目标属性值的长度。

**返回值**：目标属性。

#### 3.2.2.3 of\_property\_read\_u32\_array函数

of\_property\_read\_u32\_array()用于获取有多个值的属性的多个数据，函数名以及输入函数中的u32代表目标属性单个值的大小，可替换为u8、u16、u32。函数原型：

int of\_property\_read\_u32\_array(const struct device\_node \*np,const char \*propname, u32 \*out\_values, size\_t size);

参数说明：

**np**：设备节点。

**propname**：目标属性名。

**out\_values**：读取到的数据指针，读取到的数据会保存到这个地址中。

**size**：要读取数据数量。

**返回值**：

0：读取成功；

-EINVAL：属性不存在；

-ENODATA：属性无数据；

-EOVERFLOW：属性值数据数量小于size。

#### 3.2.2.4 of\_property\_read\_u32 函数

of\_property\_read\_u32()用于获取只有单个值的属性数据，函数名以及输入函数中的u32代表目标属性单个值的大小，可替换为u8、u16、u32。函数原型：

int of\_property\_read\_u32(const struct device\_node \*np,const char \*propname, u32 out\_value);

参数说明：

**np**：设备节点。

**propname**：目标属性名。

**out\_values**：目标数据指针，读取到的数据会保存到这个地址中。

**返回值**：

0：读取成功；

-EINVAL：属性不存在；

-ENODATA：属性无数据。

#### 3.2.2.5 of\_property\_read\_string 函数

of\_property\_read\_string 函数用于读取属性中字符串值，函数原型如下：

int of\_property\_read\_string(struct device\_node \*np, const char \*propname, const char \*\*out\_string);

参数说明：

**np**：设备节点。

**proname**：目标属性名。

**out\_string**：目标字符串指针，读取到的字符串会保存到该地址。

**返回值**：返回0读取成功。

## 3.3 设备树下的led驱动实验

通过上面两节大概了解设备树和of函数之后，光看书面上的东西很难理解深刻，还是得通过实验来深入理解。这节还是使用简单的led设备来测试，不会涵盖上面讲的所有，但是在以后的实验中，会一直用到设备树，所以不用着急，在之后的实验中慢慢去掌握就行了。

### 3.3.1 原理图

和章节1.3.1的内容相同。

### 3.3.2 修改设备树

petaliunx的工程文件中提供了让我们修改的设备树文件，在工程目录”ax\_peta/project-spec/meta-user/recipes-bsp/device-tree/files”中，ax\_peta是我的petalinux工程命，需要根据自身实际情况修改。

打开文件”system-conf.dtsi”，在根节点下添加以下节点内容：

1. alinxled {
2. compatible = "alinxled";
3. reg = <
4. 0xE000A204 0x04    /\* gpio方向寄存器 \*/
5. 0xE000A208 0x04    /\* gpio使能寄存器 \*/
6. 0xE000A040 0x04    /\* gpio控制寄存器 \*/
7. 0xF800012C 0x04    /\* AMBA外设时钟使能寄存器 \*/
8. >;
9. };

节点名称为alinxled，兼容性属性值为”alinxled”，reg中的值即为在前两次实验中用到的led相关的寄存器。

如果你的”system-conf.dtsi”文件是空的，那就自己写一个根目录，再把上面的内容放进根目录即可。

### 3.3.3 驱动程序

使用petalinux新建名为”ax-dtled-dev”的驱动程序，别忘了用petalinux-config -c rootfs命令选上新增的驱动程序。

在ax-dtled-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/device.h>
11. #include <asm/uaccess.h>
13. /\* 设备节点名称 \*/
14. #define DEVICE\_NAME       "gpio\_leds"
15. /\* 设备号个数 \*/
16. #define DEVID\_COUNT       1
17. /\* 驱动个数 \*/
18. #define DRIVE\_COUNT       1
19. /\* 主设备号 \*/
20. #define MAJOR
21. /\* 次设备号 \*/
22. #define MINOR             0
24. **/\* gpio寄存器虚拟地址 \*/**
25. **static u32 \*GPIO\_DIRM\_0;**
26. **/\* gpio使能寄存器 \*/**
27. **static u32 \*GPIO\_OEN\_0;**
28. **/\* gpio控制寄存器 \*/**
29. **static u32 \*GPIO\_DATA\_0;**
30. **/\* AMBA外设时钟使能寄存器 \*/**
31. **static u32 \*APER\_CLK\_CTRL;**
33. /\* 把驱动代码中会用到的数据打包进设备结构体 \*/
34. **struct** alinx\_char\_dev{
35. dev\_t              devid;      //设备号
36. **struct** cdev        cdev;       //字符设备
37. **struct** **class**       \***class**;     //类
38. **struct** device      \*device;    //设备
39. **struct device\_node \*nd;        //设备树的设备节点**
40. };
41. /\* 声明设备结构体 \*/
42. **static** **struct** alinx\_char\_dev alinx\_char = {
43. .cdev = {
44. .owner = THIS\_MODULE,
45. },
46. };
48. /\* open函数实现, 对应到Linux系统调用函数的open函数 \*/
49. **static** **int** gpio\_leds\_open(**struct** inode \*inode\_p, **struct** file \*file\_p)
50. {
51. /\* MIO\_0时钟使能 \*/
52. \*APER\_CLK\_CTRL |= 0x00400000;
53. /\* MIO\_0设置成输出 \*/
54. \*GPIO\_DIRM\_0 |= 0x00000001;
55. /\* MIO\_0使能 \*/
56. \*GPIO\_OEN\_0 |= 0x00000001;
58. printk("gpio\_test module open\n");
60. **return** 0;
61. }

64. /\* write函数实现, 对应到Linux系统调用函数的write函数 \*/
65. **static** ssize\_t gpio\_leds\_write(**struct** file \*file\_p, **const** **char** \_\_user \*buf, **size\_t** len, loff\_t \*loff\_t\_p)
66. {
67. **int** rst;
68. **char** writeBuf[5] = {0};
70. printk("gpio\_test module write\n");
72. rst = copy\_from\_user(writeBuf, buf, len);
73. **if**(0 != rst)
74. {
75. **return** -1;
76. }
78. **if**(1 != len)
79. {
80. printk("gpio\_test len err\n");
81. **return** -2;
82. }
83. **if**(1 == writeBuf[0])
84. {
85. \*GPIO\_DATA\_0 &= 0xFFFFFFFE;
86. printk("gpio\_test ON\n");
87. }
88. **else** **if**(0 == writeBuf[0])
89. {
90. \*GPIO\_DATA\_0 |= 0x00000001;
91. printk("gpio\_test OFF\n");
92. }
93. **else**
94. {
95. printk("gpio\_test para err\n");
96. **return** -3;
97. }
99. **return** 0;
100. }
102. /\* release函数实现, 对应到Linux系统调用函数的close函数 \*/
103. **static** **int** gpio\_leds\_release(**struct** inode \*inode\_p, **struct** file \*file\_p)
104. {
105. printk("gpio\_test module release\n");
106. **return** 0;
107. }
109. /\* 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 \*/
110. **static** **struct** file\_operations ax\_char\_fops = {
111. .owner   = THIS\_MODULE,
112. .open    = gpio\_leds\_open,
113. .write   = gpio\_leds\_write,
114. .release = gpio\_leds\_release,
115. };
117. /\* 模块加载时会调用的函数 \*/
118. **static** **int** \_\_init gpio\_led\_init(**void**)
119. {
120. **/\* 用于接受返回值 \*/**
121. **u32 ret = 0;**
122. **/\* 存放reg数据的数组 \*/**
123. **u32 reg\_data[10];**
125. **/\* 通过节点名称获取节点 \*/**
126. **alinx\_char.nd = of\_find\_node\_by\_name(NULL, "alinxled");**
127. **/\* 4、获取 reg 属性内容 \*/**
128. **ret = of\_property\_read\_u32\_array(alinx\_char.nd, "reg", reg\_data, 8);**
129. **if(ret < 0)**
130. **{**
131. **printk("get reg failed!\r\n");**
132. **return -1;**
133. **}**
134. **else**
135. **{**
136. **/\* do nothing \*/**
137. **}**
139. **/\* 把需要修改的物理地址映射到虚拟地址 \*/**
140. **GPIO\_DIRM\_0   = ioremap(reg\_data[0], reg\_data[1]);**
141. **GPIO\_OEN\_0    = ioremap(reg\_data[2], reg\_data[3]);**
142. **GPIO\_DATA\_0   = ioremap(reg\_data[4], reg\_data[5]);**
143. **APER\_CLK\_CTRL = ioremap(reg\_data[6], reg\_data[7]);**
145. /\* 注册设备号 \*/
146. alloc\_chrdev\_region(&alinx\_char.devid, MINOR, DEVID\_COUNT, DEVICE\_NAME);
148. /\* 初始化字符设备结构体 \*/
149. cdev\_init(&alinx\_char.cdev, &ax\_char\_fops);
151. /\* 注册字符设备 \*/
152. cdev\_add(&alinx\_char.cdev, alinx\_char.devid, DRIVE\_COUNT);
154. /\* 创建类 \*/
155. alinx\_char.**class** = class\_create(THIS\_MODULE, DEVICE\_NAME);
156. **if**(IS\_ERR(alinx\_char.**class**))
157. {
158. **return** PTR\_ERR(alinx\_char.**class**);
159. }
161. /\* 创建设备节点 \*/
162. alinx\_char.device = device\_create(alinx\_char.**class**, NULL,
163. alinx\_char.devid, NULL,
164. DEVICE\_NAME);
165. **if** (IS\_ERR(alinx\_char.device))
166. {
167. **return** PTR\_ERR(alinx\_char.device);
168. }
170. **return** 0;
171. }
173. /\* 卸载模块 \*/
174. **static** **void** \_\_exit gpio\_led\_exit(**void**)
175. {
176. /\* 注销字符设备 \*/
177. cdev\_del(&alinx\_char.cdev);
179. /\* 注销设备号 \*/
180. unregister\_chrdev\_region(alinx\_char.devid, DEVID\_COUNT);
182. /\* 删除设备节点 \*/
183. device\_destroy(alinx\_char.**class**, alinx\_char.devid);
185. /\* 删除类 \*/
186. class\_destroy(alinx\_char.**class**);
188. **/\* 释放对虚拟地址的占用 \*/**
189. **iounmap(GPIO\_DIRM\_0);**
190. **iounmap(GPIO\_OEN\_0);**
191. **iounmap(GPIO\_DATA\_0);**
192. **iounmap(APER\_CLK\_CTRL);**
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("DEVICE TREE GPIO LED driver");
205. MODULE\_VERSION("v1.0");
206. MODULE\_LICENSE("GPL");

和上一章有区别的地方加粗了。

主要的改动集中在入口函数的**120~137**行。

**126**行使用of\_find\_node\_by\_name函数通过节点名称获取节点，因为alinxled节点挂在在根目录下，所以第一个参数输入NULL。

**128**行，在获取到节点后，再获取节点中的reg属性的数据，因为reg属性中存放着我们需要的寄存器地址和大小。总共4个地址4个size，因此数据总数为8。

其他的操作与上一章基本相同。

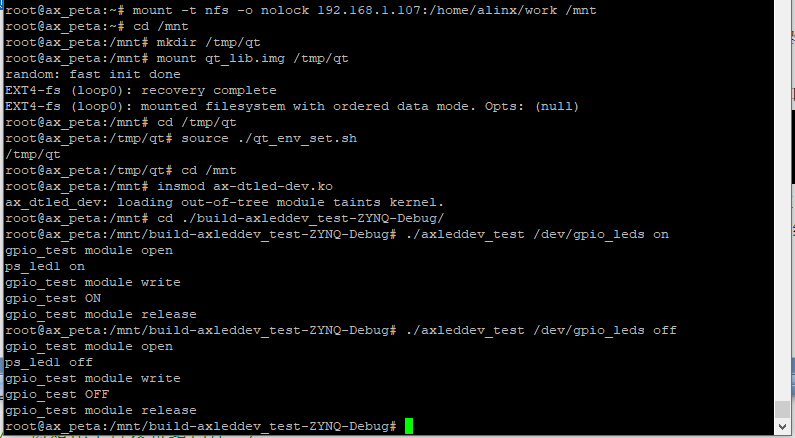
因为修改了设备树，在3.1节我们就说过，petalinux在修改设备树后，会编译出新的BOOT.bin和image.ub。所以别忘了在编译完成后，把新的BOOT.bin和image.ub拷贝到SD中，并重启开发板。

### 3.3.4 测试程序

测试APP与章节1.3.4内容一致，可以直接使用。

### 3.3.5 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。



另外在系统运行之后，查看以下/proc/device-tree路径中有没有我们添加的alinxled节点，并核对内容。

