# 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"
2.
3.  / {
4.      model = "Zynq ZC702 Development Board";
5.      compatible = "xlnx,zynq-zc702", "xlnx,zynq-7000";
6.
7.      ……
8.
9.      memory@0 {
10.         device_type = "memory";
11.         reg = <0x0 0x40000000>;
12.     };
```

```
13.
14.     ……
15.
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.         };
28.
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.      ……
5.
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.  }
5.
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。

```
root@ax_peta:~# cd /proc/device-tree
root@ax_peta:/proc/device-tree# ls
#address-cells  alinxled        chosen          fpga-full       pmu@f8891000
#size-cells     alinxpwm        compatible      memory          usb_phy@0
aliases         amba            cpus            model
alinxkey        amba_pl         fixedregulator  name
root@ax_peta:/proc/device-tree#
```

进入到对应的节点中，能查看到节点的各个属性。

```
root@ax_peta:/proc/device-tree# cd memory/
root@ax_peta:/proc/device-tree/memory# ls
device_type  name         reg
root@ax_peta:/proc/device-tree/memory# cat device_type
memoryroot@ax_peta:/proc/device-tree/memory#
```

## 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：

```
alinxled {
    compatible = "alinxled";
    reg = <
            0xE000A204 0x04    /* gpio 方向寄存器 */
            0xE000A208 0x04    /* gpio 使能寄存器 */
            0xE000A040 0x04    /* gpio 控制寄存器 */
            0xF800012C 0x04    /* AMBA 外设时钟使能寄存器 */
     >;
};
```

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：

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/ide.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <linux/cdev.h>
#include <linux/of.h>
#include <linux/device.h>
#include <asm/uaccess.h>
```

```
12.
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
23.
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;
32.
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. };
47.
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 */
```

```
    *GPIO_OEN_0 |= 0x00000001;

    printk("gpio_test module open\n");

    return 0;
}


/* write 函数实现, 对应到 Linux 系统调用函数的 write 函数 */
static ssize_t gpio_leds_write(struct file *file_p, const char __user *buf, size_t len, lof
f_t *loff_t_p)
{
    int rst;
    char writeBuf[5] = {0};

    printk("gpio_test module write\n");

    rst = copy_from_user(writeBuf, buf, len);
    if(0 != rst)
    {
        return -1;
    }

    if(1 != len)
    {
        printk("gpio_test len err\n");
        return -2;
    }
    if(1 == writeBuf[0])
    {
        *GPIO_DATA_0 &= 0xFFFFFFFE;
        printk("gpio_test ON\n");
    }
    else if(0 == writeBuf[0])
    {
        *GPIO_DATA_0 |= 0x00000001;
        printk("gpio_test OFF\n");
    }
    else
    {
        printk("gpio_test para err\n");
        return -3;
    }

```

```
99.     return 0;
100. }
101.
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. }
108.
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. };
116.
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];
124.
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.     }
138.
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]);
144. 
145.     /* 注册设备号 */
146.     alloc_chrdev_region(&alinx_char.devid, MINOR, DEVID_COUNT, DEVICE_NAME);
147. 
148.     /* 初始化字符设备结构体 */
149.     cdev_init(&alinx_char.cdev, &ax_char_fops);
150. 
151.     /* 注册字符设备 */
152.     cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
153. 
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.     }
160. 
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.     }
169. 
170.     return 0;
171. }
172. 
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);
178. 
179.     /* 注销设备号 */
180.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
181. 
182.     /* 删除设备节点 */
183.     device_destroy(alinx_char.class, alinx_char.devid);
184. 
185.     /* 删除类 */
186.     class_destroy(alinx_char.class);
```

```
187.
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);
193.
194.     printk("gpio_led_dev_exit_ok\n");
195. }
196.
197. /* 标记加载、卸载函数 */
198. module_init(gpio_led_init);
199. module_exit(gpio_led_exit);
200.
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。

```
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
random: fast init done
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
root@ax_peta:/mnt# insmod ax-dtled-dev.ko
ax_dtled_dev: loading out-of-tree module taints kernel.
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
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# ./axleddev_test /dev/gpio_leds off
gpio_test module open
ps_led1 off
gpio_test module write
gpio_test OFF
gpio_test module release
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug#
```

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。

```
root@ax_peta:~# cd /proc/device-tree
root@ax_peta:/proc/device-tree# ls
#address-cells  amba            cpus            model
#size-cells     amba_pl         fixedregulator  name
aliases         chosen          fpga-full       pmu@f8891000
alinxled        compatible      memory          usb_phy@0
root@ax_peta:/proc/device-tree# cd ./alinxled/
root@ax_peta:/proc/device-tree/alinxled# ls
compatible  name        reg
root@ax_peta:/proc/device-tree/alinxled# cat ./compatible
alinxledroot@ax_peta:/proc/device-tree/alinxled# cat ./name
alinxledroot@ax_peta:/proc/device-tree/alinxled#
```