## 第八章 驱动设计的思想：面向对象/分层/分离

### 8.1 面向对象

字符设备驱动程序抽象出一个file\_operations结构体；

我们写的程序针对硬件部分抽象出led\_operations结构体。

### 8.2 分层

上下分层，比如我们前面写的LED驱动程序就分为2层：

① 上层实现硬件无关的操作，比如注册字符设备驱动：leddrv.c

② 下层实现硬件相关的操作，比如board\_A.c实现单板A的LED操作



### 8.3 分离

还能不能改进？分离。

在board\_A.c中，实现了一个led\_operations，为LED引脚实现了初始化函数、控制函数：

static struct led\_operations board\_demo\_led\_opr = {

.num = 1,

.init = board\_demo\_led\_init,

.ctl = board\_demo\_led\_ctl,

};

如果硬件上更换一个引脚来控制LED怎么办？你要去修改上面结构体中的init、ctl函数。

实际情况是，每一款芯片它的GPIO操作都是类似的。比如：GPIO1\_3、GPIO5\_4这2个引脚接到LED：

① GPIO1\_3属于第1组，即GPIO1。

有方向寄存器DIR、数据寄存器DR等，基础地址是addr\_base\_addr\_gpio1。

设置为output引脚：修改GPIO1的DIR寄存器的bit3。

设置输出电平：修改GPIO1的DR寄存器的bit3。

② GPIO5\_4属于第5组，即GPIO5。

有方向寄存器DIR、数据寄存器DR等，基础地址是addr\_base\_addr\_gpio5。

设置为output引脚：修改GPIO5的DIR寄存器的bit4。

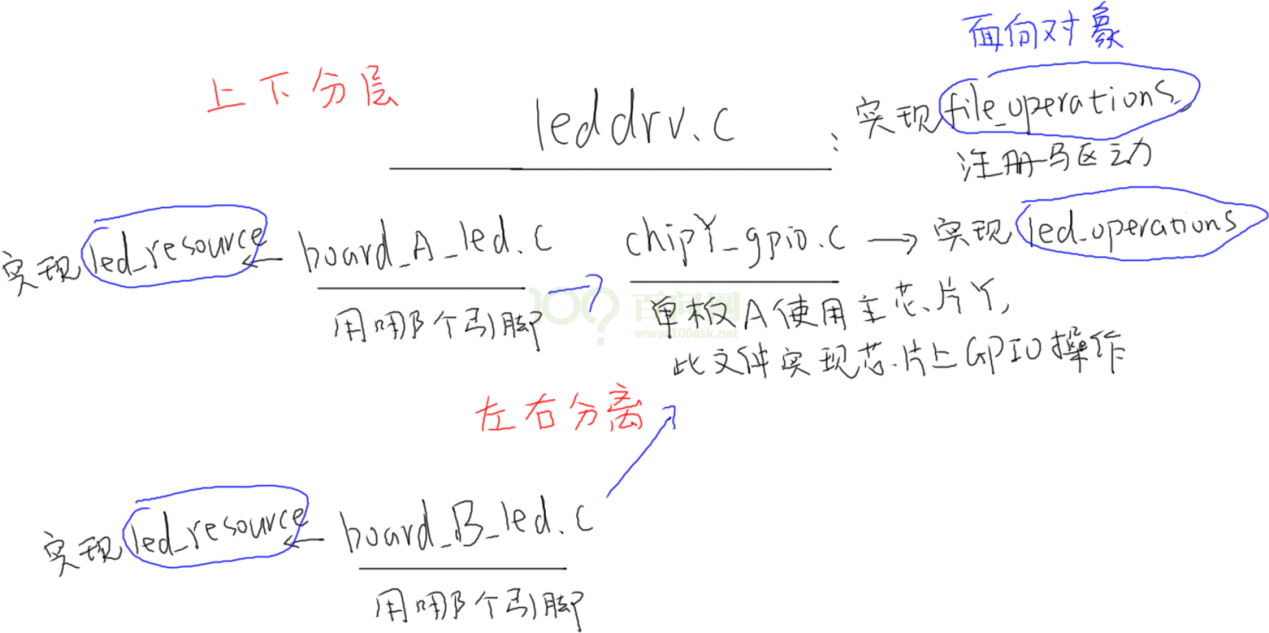
设置输出电平：修改GPIO5的DR寄存器的bit4。

既然引脚操作那么有规律，并且这是跟主芯片相关的，那可以针对该芯片写出比较通用的硬件操作代码。

比如board\_A.c使用芯片chipY，那就可以写出：chipY\_gpio.c，它实现芯片Y的GPIO操作，适用于芯片Y的所有GPIO引脚。

使用时，我们只需要在board\_A\_led.c中指定使用哪一个引脚即可。

程序结构如下：



以面向对象的思想，在board\_A\_led.c中实现led\_resouce结构体，它定义“资源”──要用哪一个引脚。

在chipY\_gpio.c中仍是实现led\_operations结构体，它要写得更完善，支持所有GPIO。

### 8.4 写示例代码

使用GIT下载所有源码后，本节源码位于如下目录：

01\_all\_series\_quickstart\

05\_嵌入式Linux驱动开发基础知识\source\02\_led\_drv\03\_led\_drv\_template\_seperate

程序仍分为上下结构：上层leddrv.c向内核注册file\_operations结构体；下层chip\_demo\_gpio.c提供led\_operations结构体来操作硬件。

下层的代码分为2个：chip\_demo\_gpio.c实现通用的GPIO操作，board\_A\_led.c指定使用哪个GPIO，即“资源”。

led\_resource.h中定义了led\_resource结构体，用来描述GPIO：

04 /\* GPIO3\_0 \*/

05 /\* bit[31:16] = group \*/

06 /\* bit[15:0] = which pin \*/

07 #define GROUP(x) (x>>16)

08 #define PIN(x) (x&0xFFFF)

09 #define GROUP\_PIN(g,p) ((g<<16) | (p))

10

11 struct led\_resource {

12 int pin;

13 };

14

15 struct led\_resource \*get\_led\_resouce(void);

16

board\_A\_led.c指定使用哪个GPIO，它实现一个led\_resource结构体，并提供访问函数：

02 #include "led\_resource.h"

03

04 static struct led\_resource board\_A\_led = {

05 .pin = GROUP\_PIN(3,1),

06 };

07

08 struct led\_resource \*get\_led\_resouce(void)

09 {

10 return &board\_A\_led;

11 }

12

chip\_demo\_gpio.c中，首先获得board\_A\_led.c实现的led\_resource结构体，然后再进行其他操作，请看下面第26行：

20 static struct led\_resource \*led\_rsc;

21 static int board\_demo\_led\_init (int which) /\* 初始化LED, which-哪个LED \*/

22 {

23 //printk("%s %s line %d, led %d\n", \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_, which);

24 if (!led\_rsc)

25 {

26 led\_rsc = get\_led\_resouce();

27 }

28

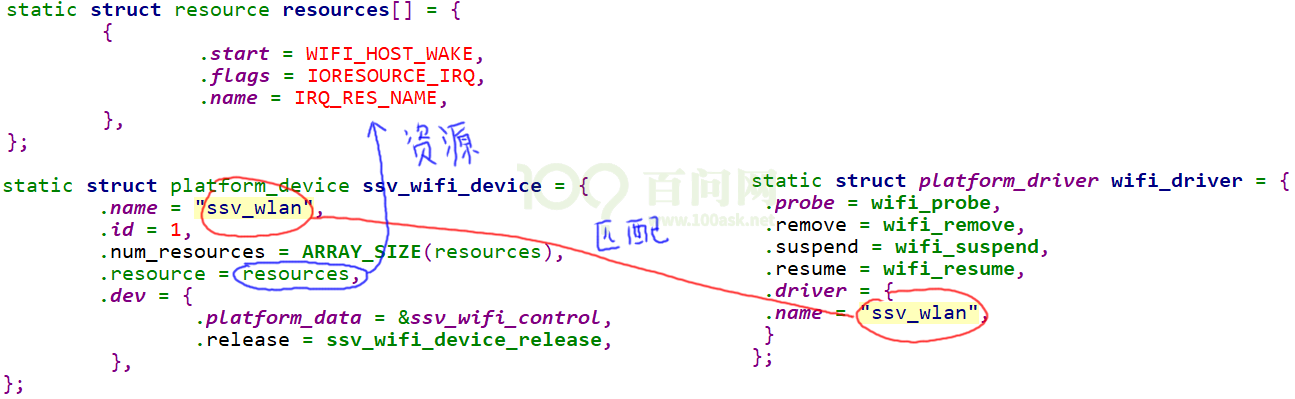
### 8.5 课后作业

使用“分离”的思想，去改造前面写的LED驱动程序：实现led\_resouce，在里面可以指定要使用哪一个LED；改造led\_operations，让它能支持更多GPIO。

注意：作为练习，led\_operations结构体不需要写得很完善，不需要支持所有GPIO，你可以只支持若干个GPIO即可。

## 第九章 驱动进化之路：总线设备驱动模型

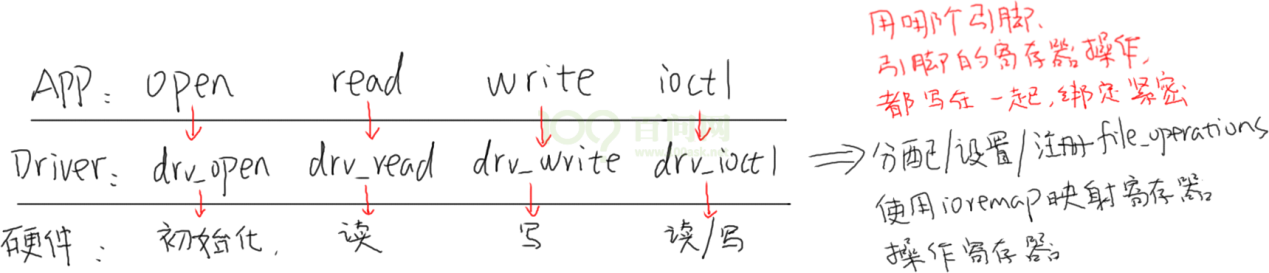
示例：



### 9.1 驱动编写的3种方法

以LED驱动为例：

#### 9.1.1 传统写法

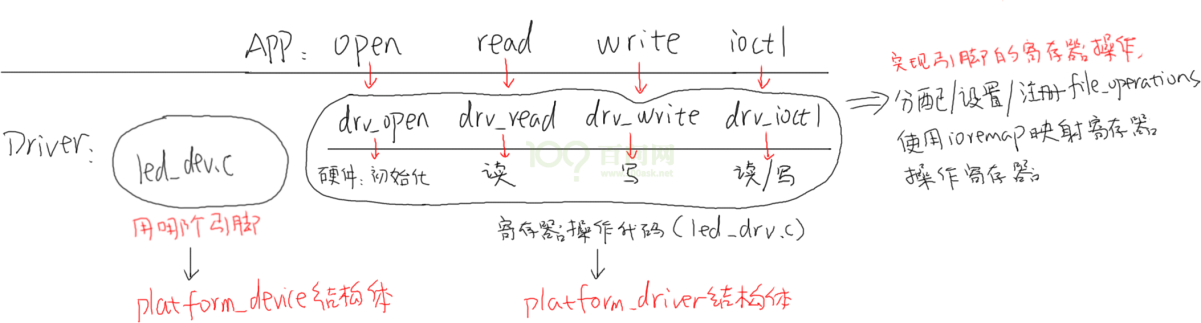


使用哪个引脚，怎么操作引脚，都写死在代码中。

最简单，不考虑扩展性，可以快速实现功能。

修改引脚时，需要重新编译。

#### 9.1.2 总线设备驱动模型



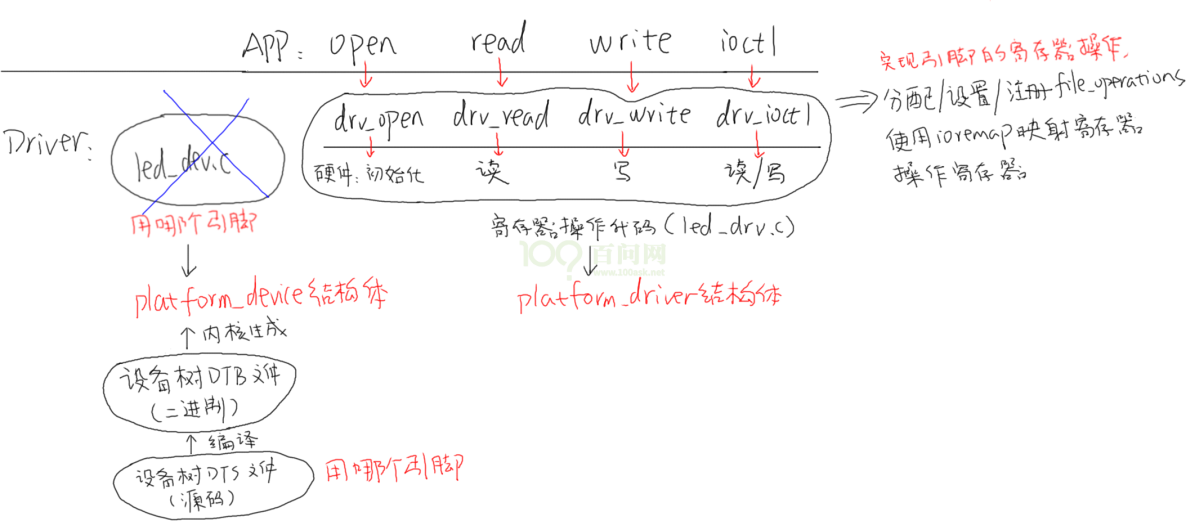
引入platform\_device/platform\_driver，将“资源”与“驱动”分离开来。

代码稍微复杂，但是易于扩展。

冗余代码太多，修改引脚时设备端的代码需要重新编译。

更换引脚时，上图中的led\_drv.c基本不用改，但是需要修改led\_dev.c

#### 9.1.3 设备树



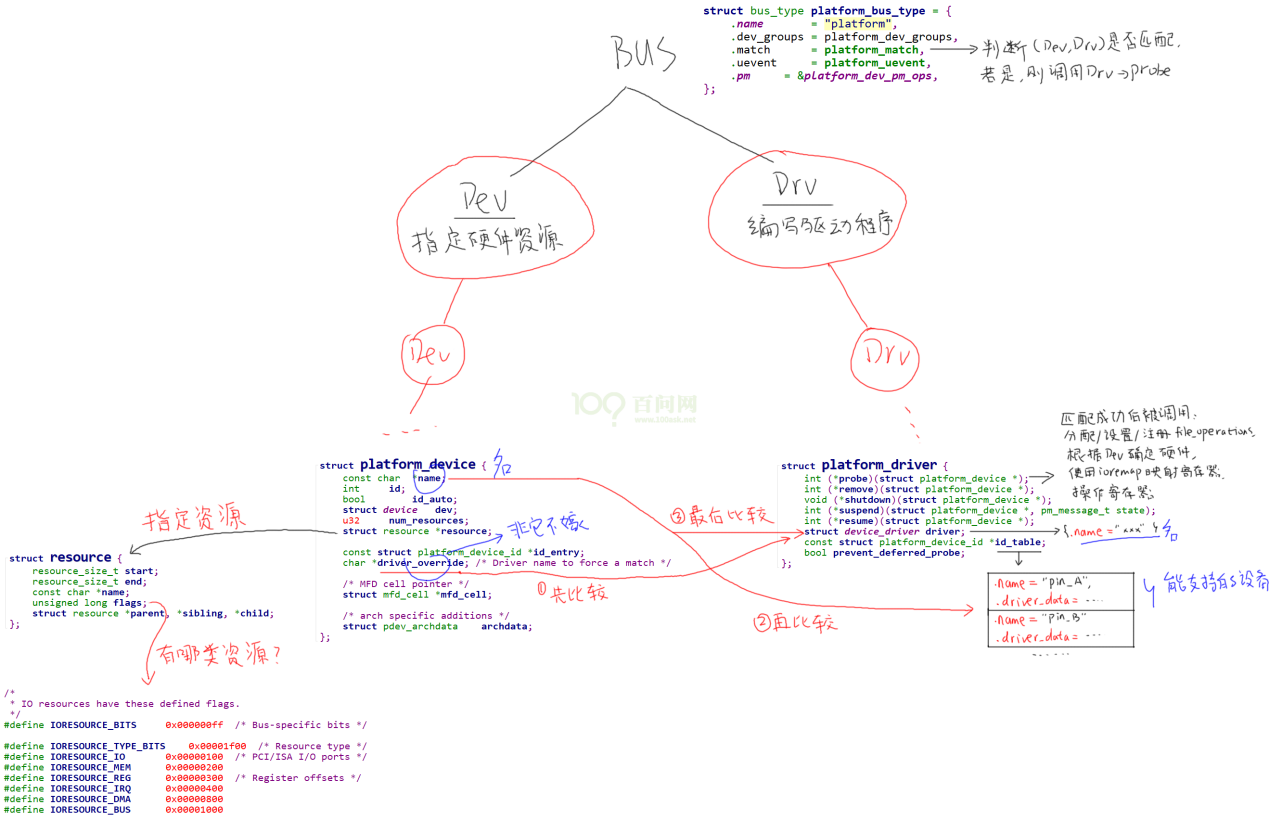
通过配置文件──设备树来定义“资源”。

代码稍微复杂，但是易于扩展。

无冗余代码，修改引脚时只需要修改dts文件并编译得到dtb文件，把它传给内核。

无需重新编译内核/驱动。

### 9.2 在Linux中实现“分离”：Bus/Dev/Drv模型



### 9.3 匹配规则

#### 9.3.1 最先比较：platform\_device. driver\_override和platform\_driver.driver.name

可以设置platform\_device的driver\_override，强制选择某个platform\_driver。

#### 9.3.2 然后比较：platform\_device. name和platform\_driver.id\_table[i].name

Platform\_driver.id\_table是“platform\_device\_id”指针，表示该drv支持若干个device，它里面列出了各个device的{.name, .driver\_data}，其中的“name”表示该drv支持的设备的名字，driver\_data是些提供给该device的私有数据。

#### 9.3.3 最后比较：platform\_device.name和platform\_driver.driver.name

platform\_driver.id\_table可能为空，

这时可以根据platform\_driver.driver.name来寻找同名的platform\_device。

#### 9.3.4 函数调用关系

platform\_device\_register

platform\_device\_add

device\_add

bus\_add\_device // 放入链表

bus\_probe\_device // probe枚举设备，即找到匹配的(dev, drv)

device\_initial\_probe

\_\_device\_attach

bus\_for\_each\_drv(...,\_\_device\_attach\_driver,...)

\_\_device\_attach\_driver

driver\_match\_device(drv, dev) // 是否匹配

driver\_probe\_device // 调用drv的probe

platform\_driver\_register

\_\_platform\_driver\_register

driver\_register

bus\_add\_driver // 放入链表

driver\_attach(drv)

bus\_for\_each\_dev(drv->bus, NULL, drv, \_\_driver\_attach);

\_\_driver\_attach

driver\_match\_device(drv, dev) // 是否匹配

driver\_probe\_device // 调用drv的probe

### 9.4 常用函数

这些函数可查看内核源码：drivers/base/platform.c，根据函数名即可知道其含义。

下面摘取常用的几个函数。

#### 9.4.1 注册/反注册

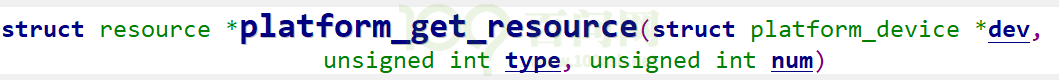
platform\_device\_register/ platform\_device\_unregister

platform\_driver\_register/ platform\_driver\_unregister

platform\_add\_devices // 注册多个device

#### 9.4.2 获得资源

返回该dev中某类型(type)资源中的第几个(num)：



返回该dev所用的第几个(num)中断：



通过名字(name)返回该dev的某类型(type)资源：



通过名字(name)返回该dev的中断号：



### 9.5 怎么写程序

#### 9.5.1 分配/设置/注册platform\_device结构体

在里面定义所用资源，指定设备名字。

#### 9.5.2 分配/设置/注册platform\_driver结构体

在其中的probe函数里，分配/设置/注册file\_operations结构体，

并从platform\_device中确实所用硬件资源。

指定platform\_driver的名字。

### 9.6 课后作业

在内核源码中搜索platform\_device\_register可以得到很多驱动，选择一个作为例子：

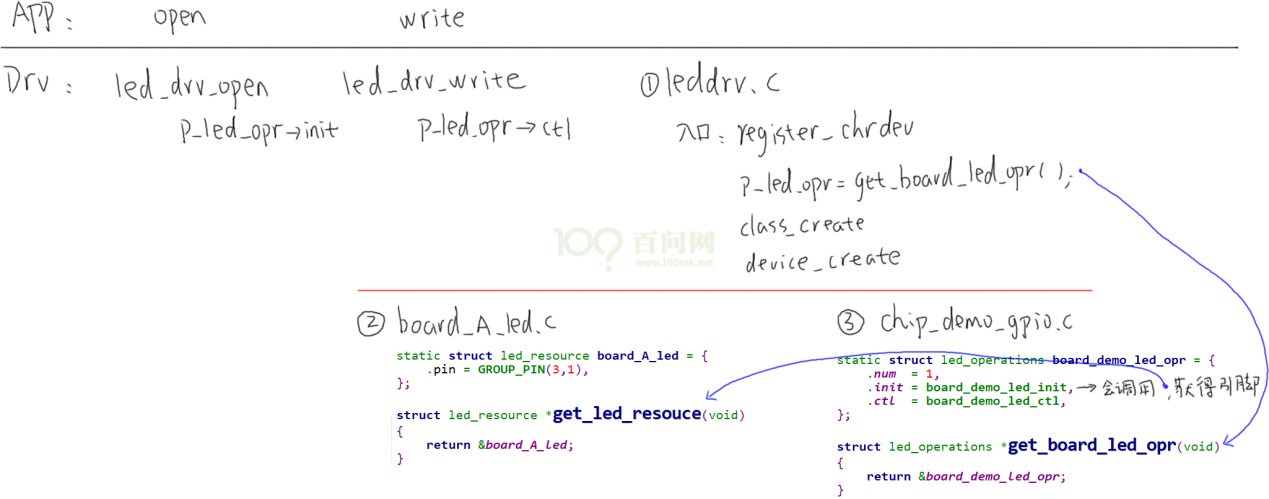
① 确定它的名字

② 根据它的名字找到对应的platform\_driver

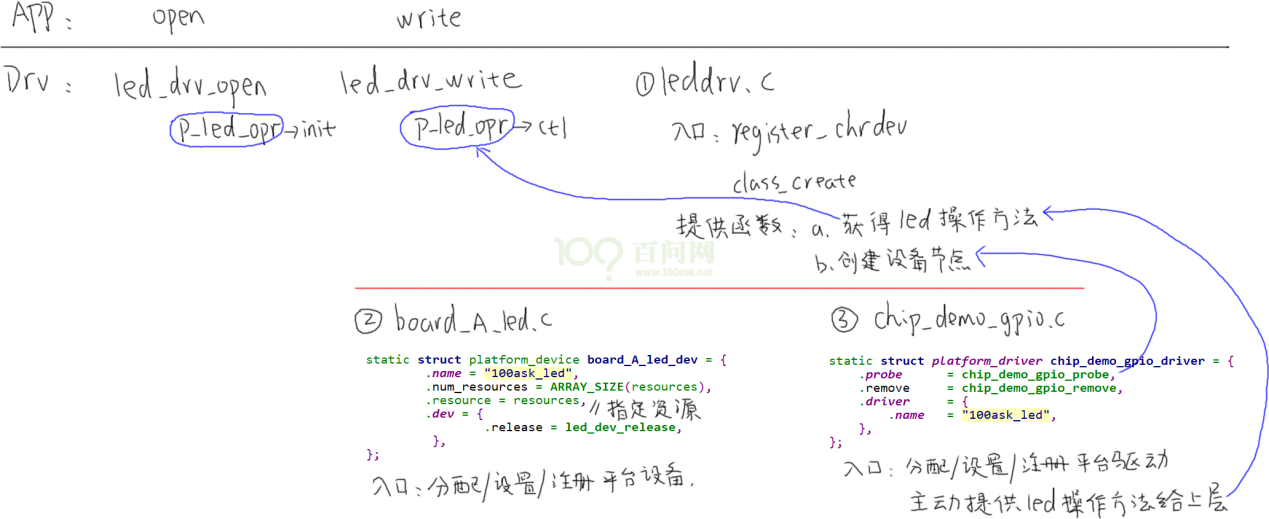
③ 进入platform\_device\_register/platform\_driver\_register内部，分析dev和drv的匹配过程

## 第十章 LED模板驱动程序的改造：总线设备驱动模型

### 10.1 原来的框架



### 10.2 要实现的框架



### 10.3 写代码

使用GIT下载所有源码后，本节源码位于如下目录：

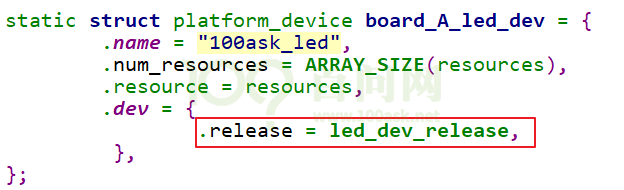
01\_all\_series\_quickstart\

05\_嵌入式Linux驱动开发基础知识\source\

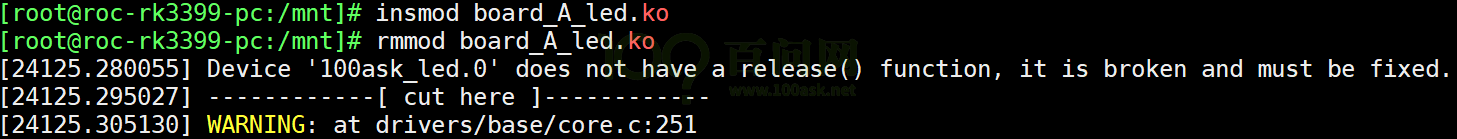
02\_led\_drv\04\_led\_drv\_template\_bus\_dev\_drv

#### 10.3.1 注意事项

① 如果platform\_device中不提供release函数，如下图所示不提供红框部分的函数：



则在调用platform\_device\_unregister时会出现警告，如下图所示：



你可以提供一个release函数，如果实在无事可做，把这函数写为空。

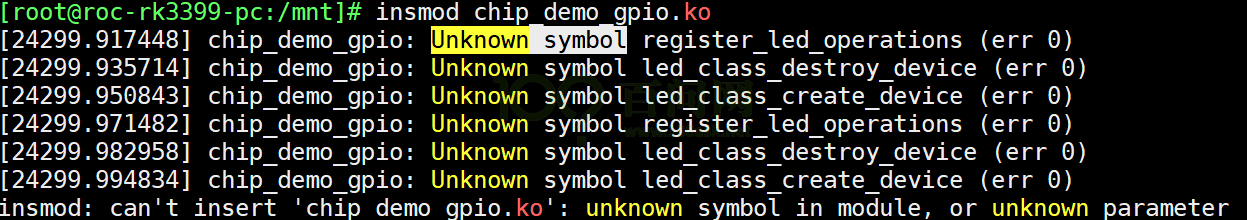
② EXPORT\_SYMBOL

a.c编译为a.ko，里面定义了func\_a；如果它想让b.ko使用该函数，那么a.c里需要导出此函数(如果a.c, b.c都编进内核，则无需导出)：

EXPORT\_SYMBOL(led\_device\_create);

并且，使用时要先加载a.ko。

如果先加载b.ko，会有类似如下“Unknown symbol”的提示：



#### 10.3.2 实现platform\_device结构体

board\_A.c作为一个可加载模块，里面也有入口函数、出口函数。在入口函数中注册platform\_device结构体，在platform\_device结构体中指定使用哪个GPIO引脚。

首先看入口函数，它调用platform\_device\_register函数，向内核注册board\_A\_led\_dev结构体：

50 static int \_\_init led\_dev\_init(void)

51 {

52 int err;

53

54 err = platform\_device\_register(&board\_A\_led\_dev);

55

56 return 0;

57 }

58

board\_A\_led\_dev结构体定义如下。

在resouces数组中指定了2个引脚(第27～38行)；

我们还提供了一个空函数led\_dev\_release(第23～25行)，它被赋给board\_A\_led\_dev结构体(第46行)，这个函数在卸载platform\_device时会被调用，如果不提供的话内核会打印警告信息。

23 static void led\_dev\_release(struct device \*dev)

24 {

25 }

26

27 static struct resource resources[] = {

28 {

29 .start = GROUP\_PIN(3,1),

30 .flags = IORESOURCE\_IRQ,

31 .name = "100ask\_led\_pin",

32 },

33 {

34 .start = GROUP\_PIN(5,8),

35 .flags = IORESOURCE\_IRQ,

36 .name = "100ask\_led\_pin",

37 },

38 };

39

40

41 static struct platform\_device board\_A\_led\_dev = {

42 .name = "100ask\_led",

43 .num\_resources = ARRAY\_SIZE(resources),

44 .resource = resources,

45 .dev = {

46 .release = led\_dev\_release,

47 },

48 };

49

#### 10.3.3 实现platform\_driver结构体

chip\_demo\_gpio.c中注册platform\_driver结构体，它使用Bus/Dev/Drv模型，当有匹配的platform\_device时，它的probe函数就会被调用。

在probe函数中所做的事情跟之前的代码没有差别。

先看入口函数。

第150行向内核注册一个platform\_driver结构体；

这个结构体的核心在于第140行的chip\_demo\_gpio\_probe函数。

138 static struct platform\_driver chip\_demo\_gpio\_driver = {

139 .probe = chip\_demo\_gpio\_probe,

140 .remove = chip\_demo\_gpio\_remove,

141 .driver = {

142 .name = "100ask\_led",

143 },

144 };

145

146 static int \_\_init chip\_demo\_gpio\_drv\_init(void)

147 {

148 int err;

149

150 err = platform\_driver\_register(&chip\_demo\_gpio\_driver);

151 register\_led\_operations(&board\_demo\_led\_opr);

152

153 return 0;

154 }

155

chip\_demo\_gpio\_probe函数代码如下。

第107行：从匹配的platform\_device中获取资源，确定GPIO引脚。

第111行：把引脚记录下来，在操作硬件时要用。

第112行：新发现了一个GPIO引脚，就调用上层驱动的代码创建设备节点。

100 static int chip\_demo\_gpio\_probe(struct platform\_device \*pdev)

101 {

102 struct resource \*res;

103 int i = 0;

104

105 while (1)

106 {

107 res = platform\_get\_resource(pdev, IORESOURCE\_IRQ, i++);

108 if (!res)

109 break;

110

111 g\_ledpins[g\_ledcnt] = res->start;

112 led\_class\_create\_device(g\_ledcnt);

113 g\_ledcnt++;

114 }

115 return 0;

116

117 }

118

操作硬件的代码如下，第31、63行的代码里用到了数组g\_ledpins，里面的值来自platform\_device，在probe函数中根据platform\_device的资源确定了引脚：

23 static int g\_ledpins[100];

24 static int g\_ledcnt = 0;

25

26 static int board\_demo\_led\_init (int which) /\* 初始化LED, which-哪个LED \*/

27 {

28 //printk("%s %s line %d, led %d\n", \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_, which);

29

30 printk("init gpio: group %d, pin %d\n", GROUP(g\_ledpins[which]), PIN(g\_ledpins[which]));

31 switch(GROUP(g\_ledpins[which]))

32 {

33 case 0:

34 {

35 printk("init pin of group 0 ...\n");

36 break;

37 }

38 case 1:

39 {

40 printk("init pin of group 1 ...\n");

41 break;

42 }

43 case 2:

44 {

45 printk("init pin of group 2 ...\n");

46 break;

47 }

48 case 3:

49 {

50 printk("init pin of group 3 ...\n");

51 break;

52 }

53 }

54

55 return 0;

56 }

57

58 static int board\_demo\_led\_ctl (int which, char status) /\* 控制LED, which-哪个LED, status:1-亮,0-灭 \*/

59 {

60 //printk("%s %s line %d, led %d, %s\n", \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_, which, status ? "on" : "off");

61 printk("set led %s: group %d, pin %d\n", status ? "on" : "off", GROUP(g\_ledpins[which]), PIN(g\_ledpins[which]));

62

63 switch(GROUP(g\_ledpins[which]))

64 {

65 case 0:

66 {

67 printk("set pin of group 0 ...\n");

68 break;

69 }

70 case 1:

71 {

72 printk("set pin of group 1 ...\n");

73 break;

74 }

75 case 2:

76 {

77 printk("set pin of group 2 ...\n");

78 break;

79 }

80 case 3:

81 {

82 printk("set pin of group 3 ...\n");

83 break;

84 }

85 }

86

87 return 0;

88 }

89

90 static struct led\_operations board\_demo\_led\_opr = {

91 .init = board\_demo\_led\_init,

92 .ctl = board\_demo\_led\_ctl,

93 };

94

95 struct led\_operations \*get\_board\_led\_opr(void)

96 {

97 return &board\_demo\_led\_opr;

98 }

99

### 10.4 课后作业

完善半成品程序：04\_led\_drv\_template\_bus\_dev\_drv\_unfinished。

请仿照本节提供的程序(位于04\_led\_drv\_template\_bus\_dev\_drv目录)，改造你所用的单板的LED驱动程序。

## 第十一章 驱动进化之路：设备树的引入及简明教程

官方文档(可以下载到devicetree-specification-v0.2.pdf):

<https://www.devicetree.org/specifications/>

内核文档:

Documentation/devicetree/booting-without-of.txt

我录制“设备树视频”时写的文档：设备树详细分析.txt

这个txt文件也同步上传到wiki了：<http://wiki.100ask.org/Linux_devicetree>

我录制的设备树视频，它是基于s3c2440的，用的是linux 4.19；需要深入研究的可以看该视频(收费)。

注意，如果只是想入门，看本文档及视频即可。

### 11.1 设备树的引入与作用

以LED驱动为例，如果你要更换LED所用的GPIO引脚，需要修改驱动程序源码、重新编译驱动、重新加载驱动。

在内核中，使用同一个芯片的板子，它们所用的外设资源不一样，比如A板用GPIO A，B板用GPIO B。而GPIO的驱动程序既支持GPIO A也支持GPIO B，你需要指定使用哪一个引脚，怎么指定？在c代码中指定。

随着ARM芯片的流行，内核中针对这些ARM板保存有大量的、没有技术含量的文件。

Linus大发雷霆："this whole ARM thing is a f\*cking pain in the ass"。

于是，Linux内核开始引入设备树。

设备树并不是重新发明出来的，在Linux内核中其他平台如PowerPC，早就使用设备树来描述硬件了。

Linus发火之后，内核开始全面使用设备树来改造，神人就神人。

有一种错误的观点，说“新驱动都是用设备树来写了”。

设备树不可能用来写驱动。

请想想，要操作硬件就需要去操作复杂的寄存器，如果设备树可以操作寄存器，那么它就是“驱动”，它就一样很复杂。

设备树只是用来给内核里的驱动程序，指定硬件的信息。比如LED驱动，在内核的驱动程序里去操作寄存器，但是操作哪一个引脚？这由设备树指定。

你可以事先体验一下设备树，板子启动后执行下面的命令：

# ls /sys/firmware/

devicetree fdt

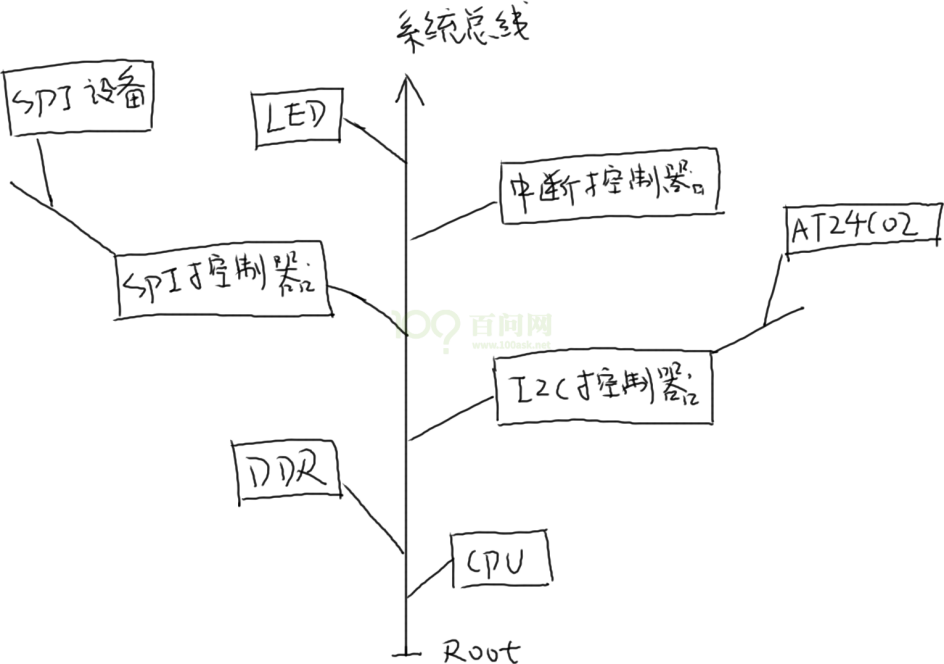
/sys/firmware/devicetree目录下是以目录结构程现的dtb文件, 根节点对应base目录, 每一个节点对应一个目录, 每一个属性对应一个文件。

这些属性的值如果是字符串，可以使用cat命令把它打印出来；对于数值，可以用hexdump把它打印出来。

一个单板启动时，u-boot先运行，它的作用是启动内核。U-boot会把内核和设备树文件都读入内存，然后启动内核。在启动内核时会把设备树在内存中的地址告诉内核。

### 11.2 设备树的语法

为什么叫“树”？

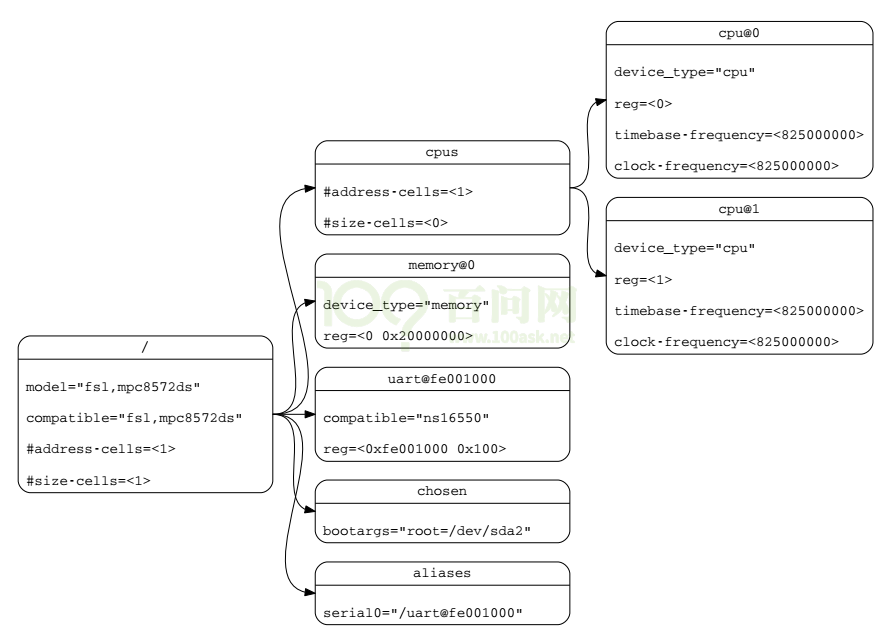


怎么描述这棵树？

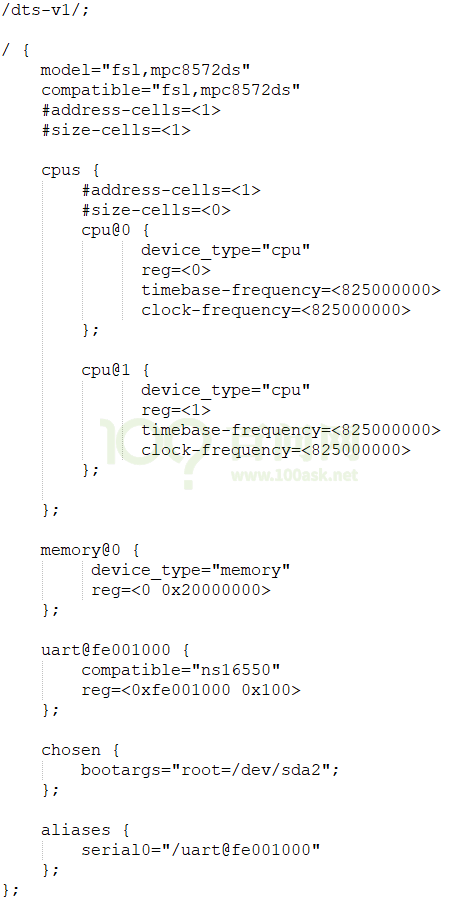
我们需要编写设备树文件(dts: device tree source)，它需要编译为dtb(device tree blob)文件，内核使用的是dtb文件。

dts文件是根本，它的语法很简单。

下面是一个设备树示例：



它对应的dts文件如下：



#### 11.2.1 1Devicetree格式

##### 11.2.1.1 1DTS文件的格式

DTS文件布局(layout):

/dts-v1/; // 表示版本

[memory reservations] // 格式为: /memreserve/ <address> <length>;

/ {

[property definitions]

[child nodes]

};

##### 11.2.1.2 node的格式

设备树中的基本单元，被称为“node”，其格式为：

[label:] node-name[@unit-address] {

[properties definitions]

[child nodes]

};

label是标号，可以省略。label的作用是为了方便地引用node，比如：

/dts-v1/;

/ {

uart0: uart@fe001000 {

compatible="ns16550";

reg=<0xfe001000 0x100>;

};

};

可以使用下面2种方法来修改uart@fe001000这个node：

// 在根节点之外使用label引用node：

&uart0 {

status = “disabled”;

};

或在根节点之外使用全路径：

&{/uart@fe001000} {

status = “disabled”;

};

##### 11.2.1.3 properties的格式

简单地说，properties就是“name=value”，value有多种取值方式。

**Property格式1:**

[label:] property-name = value;

**Property格式2(没有值):**

[label:] property-name;

**Property取值只有3种:**

arrays of cells(1个或多个32位数据, 64位数据使用2个32位数据表示),

string(字符串),

bytestring(1个或多个字节)

**示例:**

a. Arrays of cells : cell就是一个32位的数据，用尖括号包围起来

interrupts = <17 0xc>;

b. 64bit数据使用2个cell来表示，用尖括号包围起来:

clock-frequency = <0x00000001 0x00000000>;

c. A null-terminated string (有结束符的字符串)，用双引号包围起来:

compatible = "simple-bus";

d. A bytestring(字节序列) ，用中括号包围起来:

local-mac-address = [00 00 12 34 56 78]; // 每个byte使用2个16进制数来表示

local-mac-address = [000012345678]; // 每个byte使用2个16进制数来表示

e. 可以是各种值的组合, 用逗号隔开:

compatible = "ns16550", "ns8250";

example = <0xf00f0000 19>, "a strange property format";

#### 11.2.2 dts文件包含dtsi文件

设备树文件不需要我们从零写出来，内核支持了某款芯片比如imx6ull，在内核的arch/arm/boot/dts目录下就有了能用的设备树模板，一般命名为xxxx.dtsi。“i”表示“include”，被别的文件引用的。

我们使用某款芯片制作出了自己的单板，所用资源跟xxxx.dtsi是大部分相同，小部分不同，所以需要引脚xxxx.dtsi并修改。

dtsi文件跟dts文件的语法是完全一样的。

dts中可以包含.h头文件，也可以包含dtsi文件，在.h头文件中可以定义一些宏。

示例：

/dts-v1/;

#include <dt-bindings/input/input.h>

#include "imx6ull.dtsi"

/ {

……

};

#### 11.2.3 常用的属性

##### 11.2.3.1 #address-cells、#size-cells

cell指一个32位的数值，

address-cells：address要用多少个32位数来表示；

size-cells：size要用多少个32位数来表示。

比如一段内存，怎么描述它的起始地址和大小？

下例中，address-cells为1，所以reg中用1个数来表示地址，即用0x80000000来表示地址；size-cells为1，所以reg中用1个数来表示大小，即用0x20000000表示大小：

/ {

#address-cells = <1>;

#size-cells = <1>;

memory {

reg = <0x80000000 0x20000000>;

};

};

##### 11.2.3.2 compatible

“compatible”表示“兼容”，对于某个LED，内核中可能有A、B、C三个驱动都支持它，那可以这样写：

led {

compatible = “A”, “B”, “C”;

};

内核启动时，就会为这个LED按这样的优先顺序为它找到驱动程序：A、B、C。

根节点下也有compatible属性，用来选择哪一个“machine desc”：一个内核可以支持machine A，也支持machine B，内核启动后会根据根节点的compatible属性找到对应的machine desc结构体，执行其中的初始化函数。

compatible的值，建议取这样的形式："manufacturer,model"，即“厂家名,模块名”。

注意：machine desc的意思就是“机器描述”，学到内核启动流程时才涉及。

##### 11.2.3.3 model

model属性与compatible属性有些类似，但是有差别。

compatible属性是一个字符串列表，表示可以你的硬件兼容A、B、C等驱动；

model用来准确地定义这个硬件是什么。

比如根节点中可以这样写：

/ {

compatible = "samsung,smdk2440", "samsung,mini2440";

model = "jz2440\_v3";

};

它表示这个单板，可以兼容内核中的“smdk2440”，也兼容“mini2440”。

从compatible属性中可以知道它兼容哪些板，但是它到底是什么板？用model属性来明确。

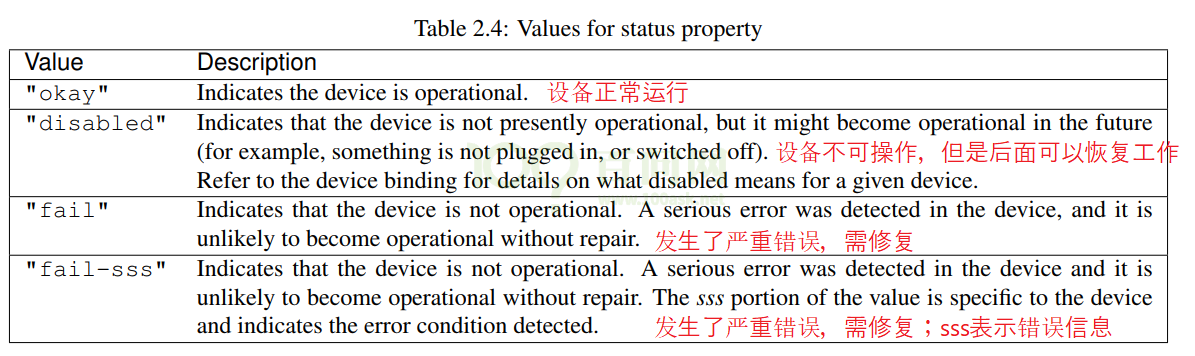
##### 11.2.3.4 status

dtsi文件中定义了很多设备，但是在你的板子上某些设备是没有的。这时你可以给这个设备节点添加一个status属性，设置为“disabled”：

&uart1 {

status = "disabled";

};



##### 11.2.3.5 reg

reg的本意是register，用来表示寄存器地址。

但是在设备树里，它可以用来描述一段空间。反正对于ARM系统，寄存器和内存是统一编址的，即访问寄存器时用某块地址，访问内存时用某块地址，在访问方法上没有区别。

reg属性的值，是一系列的“address size”，用多少个32位的数来表示address和size，由其父节点的#address-cells、#size-cells决定。

示例：

/dts-v1/;

/ {

#address-cells = <1>;

#size-cells = <1>;

memory {

reg = <0x80000000 0x20000000>;

};

};

##### 11.2.3.6 name(过时了，建议不用)

它的值是字符串，用来表示节点的名字。在跟platform\_driver匹配时，优先级最低。

compatible属性在匹配过程中，优先级最高。

##### 11.2.3.7 1device\_type(过时了，建议不用)

它的值是字符串，用来表示节点的类型。在跟platform\_driver匹配时，优先级为中。

compatible属性在匹配过程中，优先级最高。

#### 11.2.4 常用的节点(node)

##### 11.2.4.1 根节点

dts文件中必须有一个根节点：

/dts-v1/;

/ {

model = "SMDK24440";

compatible = "samsung,smdk2440";

#address-cells = <1>;

#size-cells = <1>;

};

根节点中必须有这些属性：

#address-cells // 在它的子节点的reg属性中, 使用多少个u32整数来描述地址(address)

#size-cells // 在它的子节点的reg属性中, 使用多少个u32整数来描述大小(size)

compatible // 定义一系列的字符串, 用来指定内核中哪个machine\_desc可以支持本设备

// 即这个板子兼容哪些平台

// uImage : smdk2410 smdk2440 mini2440 ==> machine\_desc

model // 咱这个板子是什么

// 比如有2款板子配置基本一致, 它们的compatible是一样的

// 那么就通过model来分辨这2款板子

##### 11.2.4.2 CPU节点

一般不需要我们设置，在dtsi文件中都定义好了：

cpus {

#address-cells = <1>;

#size-cells = <0>;

cpu0: cpu@0 {

.......

}

};

##### 11.2.4.3 memory节点

芯片厂家不可能事先确定你的板子使用多大的内存，所以memory节点需要板厂设置，比如：

memory {

reg = <0x80000000 0x20000000>;

};

##### 11.2.4.4 chosen节点

我们可以通过设备树文件给内核传入一些参数，这要在chosen节点中设置bootargs属性：

chosen {

bootargs = "noinitrd root=/dev/mtdblock4 rw init=/linuxrc console=ttySAC0,115200";

};

### 11.3 编译、更换设备树

我们一般不会从零写dts文件，而是修改。程序员水平有高有低，改得对不对？需要编译一下。并且内核直接使用dts文件的话，就太低效了，它也需要使用二进制格式的dtb文件。

#### 11.3.1 在内核中直接make

设置ARCH、CROSS\_COMPILE、PATH这三个环境变量后，进入ubuntu上板子内核源码的目录，执行如下命令即可编译dtb文件：

make dtbs V=1

这些操作步骤在各个开发板的高级用户使用手册，或是<http://wiki.100ask.net>中各个板子的页面里，都有说明。

以野火的IMX6UL为例，可以看到如下输出：

mkdir -p arch/arm/boot/dts/ ;

arm-linux-gnueabihf-gcc -E

-Wp,-MD,arch/arm/boot/dts/.imx6ull-14x14-ebf-mini.dtb.d.pre.tmp

-nostdinc

-I./arch/arm/boot/dts

-I./arch/arm/boot/dts/include

-I./drivers/of/testcase-data

-undef -D\_\_DTS\_\_ -x assembler-with-cpp

-o arch/arm/boot/dts/.imx6ull-14x14-ebf-mini.dtb.dts.tmp

arch/arm/boot/dts/imx6ull-14x14-ebf-mini.dts ;

./scripts/dtc/dtc -O dtb

-o arch/arm/boot/dts/imx6ull-14x14-ebf-mini.dtb

-b 0 -i arch/arm/boot/dts/ -Wno-unit\_address\_vs\_reg

-d arch/arm/boot/dts/.imx6ull-14x14-ebf-mini.dtb.d.dtc.tmp

arch/arm/boot/dts/.imx6ull-14x14-ebf-mini.dtb.dts.tmp ;

它首先用arm-linux-gnueabihf-gcc预处理dts文件，把其中的.h头文件包含进来，把宏展开。

然后使用scripts/dtc/dtc生成dtb文件。

可见，dts文件之所以支持“#include”语法，是因为arm-linux-gnueabihf-gcc帮忙。

如果只用dtc工具，它是不支持”#include”语法的，只支持“/include”语法。

#### 11.3.2 手工编译

除非你对设备树比较了解，否则不建议手工使用dtc工具直接编译。

内核目录下scripts/dtc/dtc是设备树的编译工具，直接使用它的话，包含其他文件时不能使用“#include”，而必须使用“/incldue”。

编译、反编译的示例命令如下，“-I”指定输入格式，“-O”指定输出格式，“-o”指定输出文件：

./scripts/dtc/dtc -I dts -O dtb -o tmp.dtb arch/arm/boot/dts/xxx.dts // 编译dts为dtb

./scripts/dtc/dtc -I dtb -O dts -o tmp.dts arch/arm/boot/dts/xxx.dtb // 反编译dtb为dts

#### 11.3.3 给开发板更换设备树文件

怎么给各个单板编译出设备树文件，它们的设备树文件是哪一个？

这些操作步骤在各个开发板的高级用户使用手册，或是<http://wiki.100ask.net>中各个板子的页面里，都有说明。

基本方法都是：设置ARCH、CROSS\_COMPILE、PATH这三个环境变量后，在内核源码目录中执行：

make dtbs

##### 11.3.3.4 对于百问网使用QEMU模拟的IMX6ULL板子

设备树文件是：内核源码目录中arch/arm/boot/dts/100ask\_imx6ul\_qemu.dtb

它是执行qemu时直接在命令行中指定设备树文件的，你可以打开脚本文件qemu-imx6ul-gui.sh找到dtb文件的位置，然后使用新编译出来的dtb去覆盖老文件。

##### 11.3.3.5 对于野火imx6ull-pro

设备树文件是：内核源码目录中arch/arm/boot/dts/imx6ull-14x14-ebf.dtb

对于这款板子，本教程中我们使用SD卡上的系统。

要更换板上的设备树文件，你可以使用SD卡启动开发板后，更换这个文件：/boot/imx6ull-14x14-ebf.dtb

#### 11.3.4 板子启动后查看设备树

板子启动后执行下面的命令：

# ls /sys/firmware/

devicetree fdt

/sys/firmware/devicetree目录下是以目录结构程现的dtb文件, 根节点对应base目录, 每一个节点对应一个目录, 每一个属性对应一个文件。

这些属性的值如果是字符串，可以使用cat命令把它打印出来；对于数值，可以用hexdump把它打印出来。

还可以看到/sys/firmware/fdt文件，它就是dtb格式的设备树文件，可以把它复制出来放到ubuntu上，执行下面的命令反编译出来(-I dtb：输入格式是dtb，-O dts：输出格式是dts)：

cd 板子所用的内核源码目录

./scripts/dtc/dtc -I dtb -O dts /从板子上/复制出来的/fdt -o tmp.dts

### 11.4 内核对设备树的处理

从源代码文件dts文件开始，设备树的处理过程为：



① dts在PC机上被编译为dtb文件；

② u-boot把dtb文件传给内核；

③ 内核解析dtb文件，把每一个节点都转换为device\_node结构体；

④ 对于某些device\_node结构体，会被转换为platform\_device结构体。

#### 11.4.1 dtb中每一个节点都被转换为device\_node结构体



根节点被保存在全局变量of\_root中，从of\_root开始可以访问到任意节点。

#### 11.4.2 哪些设备树节点会被转换为platform\_device

A. 根节点下含有compatile属性的子节点

B. 含有特定compatile属性的节点的子节点

如果一个节点的compatile属性，它的值是这4者之一："simple-bus","simple-mfd","isa","arm,amba-bus",

那么它的子结点(需含compatile属性)也可以转换为platform\_device。

C. 总线I2C、SPI节点下的子节点：不转换为platform\_device

某个总线下到子节点，应该交给对应的总线驱动程序来处理, 它们不应该被转换为platform\_device。

比如以下的节点中：

/mytest会被转换为platform\_device, 因为它兼容"simple-bus";

它的子节点/mytest/mytest@0 也会被转换为platform\_device

/i2c节点一般表示i2c控制器, 它会被转换为platform\_device, 在内核中有对应的platform\_driver;

/i2c/at24c02节点不会被转换为platform\_device, 它被如何处理完全由父节点的platform\_driver决定, 一般是被创建为一个i2c\_client。

类似的也有/spi节点, 它一般也是用来表示SPI控制器, 它会被转换为platform\_device, 在内核中有对应的platform\_driver;

/spi/flash@0节点不会被转换为platform\_device, 它被如何处理完全由父节点的platform\_driver决定, 一般是被创建为一个spi\_device。

/ {

mytest {

compatile = "mytest", "simple-bus";

mytest@0 {

compatile = "mytest\_0";

};

};

i2c {

compatile = "samsung,i2c";

at24c02 {

compatile = "at24c02";

};

};

spi {

compatile = "samsung,spi";

flash@0 {

compatible = "winbond,w25q32dw";

spi-max-frequency = <25000000>;

reg = <0>;

};

};

};

#### 11.4.3 怎么转换为platform\_device

内核处理设备树的函数调用过程，这里不去分析；我们只需要得到如下结论：

A. platform\_device中含有resource数组, 它来自device\_node的reg, interrupts属性;

B. platform\_device.dev.of\_node指向device\_node, 可以通过它获得其他属性

### 11.5 platform\_device如何与platform\_driver配对

从设备树转换得来的platform\_device会被注册进内核里，以后当我们每注册一个platform\_driver时，它们就会两两确定能否配对，如果能配对成功就调用platform\_driver的probe函数。

套路是一样的。

我们需要将前面讲过的“匹配规则”再完善一下：

先贴源码：



#### 11.5.1 最先比较：是否强制选择某个driver

比较platform\_device. driver\_override和platform\_driver.driver.name

可以设置platform\_device的driver\_override，强制选择某个platform\_driver。

#### 11.5.2 然后比较：设备树信息

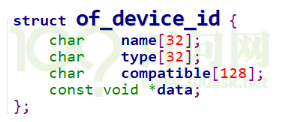
比较：platform\_device. dev.of\_node和platform\_driver.driver.of\_match\_table。

由设备树节点转换得来的platform\_device中，含有一个结构体：of\_node。

它的类型如下：



如果一个platform\_driver支持设备树，它的platform\_driver.driver.of\_match\_table是一个数组，类型如下：



使用设备树信息来判断dev和drv是否配对时，

首先，如果of\_match\_table中含有compatible值，就跟dev的compatile属性比较，若一致则成功，否则返回失败；

其次，如果of\_match\_table中含有type值，就跟dev的device\_type属性比较，若一致则成功，否则返回失败；

最后，如果of\_match\_table中含有name值，就跟dev的name属性比较，若一致则成功，否则返回失败。

而设备树中建议不再使用devcie\_type和name属性，所以基本上只使用设备节点的compatible属性来寻找匹配的platform\_driver。

#### 11.5.3 接下来比较：platform\_device\_id

比较platform\_device. name和platform\_driver.id\_table[i].name，id\_table中可能有多项。

platform\_driver.id\_table是“platform\_device\_id”指针，表示该drv支持若干个device，它里面列出了各个device的{.name, .driver\_data}，其中的“name”表示该drv支持的设备的名字，driver\_data是些提供给该device的私有数据。

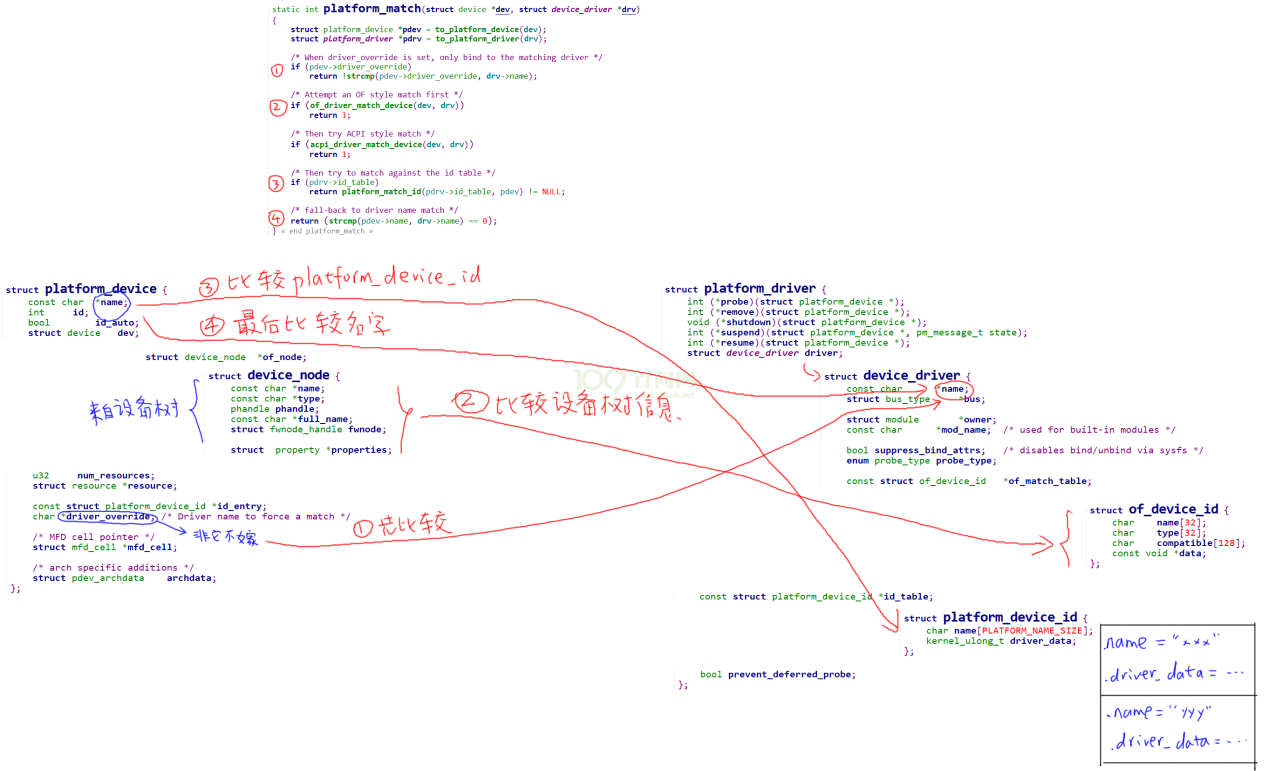
#### 11.5.4 最后比较：platform\_device.name和platform\_driver.driver.name

platform\_driver.id\_table可能为空，

这时可以根据platform\_driver.driver.name来寻找同名的platform\_device。

#### 11.5.5 一个图概括所有的配对过程

概括出了这个图：



### 11.6 没有转换为platform\_device的节点，如何使用

任意驱动程序里，都可以直接访问设备树。

你可以使用“11.7”节中介绍的函数找到节点，读出里面的值。

### 11.7 内核里操作设备树的常用函数

内核源码中include/linux/目录下有很多of开头的头文件，of表示“open firmware”即开放固件。

#### 11.7.1 内核中设备树相关的头文件介绍

设备树的处理过程是：dtb -> device\_node -> platform\_device。

##### 11.7.1.1 处理DTB

of\_fdt.h // dtb文件的相关操作函数, 我们一般用不到,

// 因为dtb文件在内核中已经被转换为device\_node树(它更易于使用)

##### 11.7.1.2 处理device\_node

of.h // 提供设备树的一般处理函数,

// 比如 of\_property\_read\_u32(读取某个属性的u32值),

// of\_get\_child\_count(获取某个device\_node的子节点数)

of\_address.h // 地址相关的函数,

// 比如 of\_get\_address(获得reg属性中的addr, size值)

// of\_match\_device (从matches数组中取出与当前设备最匹配的一项)

of\_dma.h // 设备树中DMA相关属性的函数

of\_gpio.h // GPIO相关的函数

of\_graph.h // GPU相关驱动中用到的函数, 从设备树中获得GPU信息

of\_iommu.h // 很少用到

of\_irq.h // 中断相关的函数

of\_mdio.h // MDIO (Ethernet PHY) API

of\_net.h // OF helpers for network devices.

of\_pci.h // PCI相关函数

of\_pdt.h // 很少用到

of\_reserved\_mem.h // reserved\_mem的相关函数

##### 11.7.1.3 处理 platform\_device

of\_platform.h // 把device\_node转换为platform\_device时用到的函数,

// 比如of\_device\_alloc(根据device\_node分配设置platform\_device),

// of\_find\_device\_by\_node (根据device\_node查找到platform\_device),

// of\_platform\_bus\_probe (处理device\_node及它的子节点)

of\_device.h // 设备相关的函数, 比如 of\_match\_device

#### 11.7.2 platform\_device相关的函数

of\_platform.h中声明了很多函数，但是作为驱动开发者，我们只使用其中的1、2个。其他的都是给内核自己使用的，内核使用它们来处理设备树，转换得到platform\_device。

##### 11.7.2.1 of\_find\_device\_by\_node

函数原型为：

extern struct platform\_device \*of\_find\_device\_by\_node(struct device\_node \*np);

设备树中的每一个节点，在内核里都有一个device\_node；你可以使用device\_node去找到对应的platform\_device。

##### 11.7.2.2 platform\_get\_resource

这个函数跟设备树没什么关系，但是设备树中的节点被转换为platform\_device后，设备树中的reg属性、interrupts属性也会被转换为“resource”。

这时，你可以使用这个函数取出这些资源。

函数原型为：

/\*\*

\* platform\_get\_resource - get a resource for a device

\* @dev: platform device

\* @type: resource type // 取哪类资源？IORESOURCE\_MEM、IORESOURCE\_REG

\* // IORESOURCE\_IRQ等

\* @num: resource index // 这类资源中的哪一个？

\*/

struct resource \*platform\_get\_resource(struct platform\_device \*dev,

unsigned int type, unsigned int num);

对于设备树节点中的reg属性，它对应IORESOURCE\_MEM类型的资源；

对于设备树节点中的interrupts属性，它对应IORESOURCE\_IRQ类型的资源。

#### 11.7.3 有些节点不会生成platform\_device，怎么访问它们

内核会把dtb文件解析出一系列的device\_node结构体，我们可以直接访问这些device\_node。

内核源码incldue/linux/of.h中声明了device\_node和属性property的操作函数，device\_node和property的结构体定义如下：



##### 11.7.3.1 找到节点

**a. of\_find\_node\_by\_path**

根据路径找到节点，比如“/”就对应根节点，“/memory”对应memory节点。

函数原型：

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

**b. of\_find\_node\_by\_name**

根据名字找到节点，节点如果定义了name属性，那我们可以根据名字找到它。

函数原型：

extern struct device\_node \*of\_find\_node\_by\_name(struct device\_node \*from,

const char \*name);

参数from表示从哪一个节点开始寻找，传入NULL表示从根节点开始寻找。

但是在设备树的官方规范中不建议使用“name”属性，所以这函数也不建议使用。

**c. of\_find\_node\_by\_type**

根据类型找到节点，节点如果定义了device\_type属性，那我们可以根据类型找到它。

函数原型：

extern struct device\_node \*of\_find\_node\_by\_type(struct device\_node \*from,

const char \*type);

参数from表示从哪一个节点开始寻找，传入NULL表示从根节点开始寻找。

但是在设备树的官方规范中不建议使用“device\_type”属性，所以这函数也不建议使用。

**d. of\_find\_compatible\_node**

根据compatible找到节点，节点如果定义了compatible属性，那我们可以根据compatible属性找到它。

函数原型：

extern struct device\_node \*of\_find\_compatible\_node(struct device\_node \*from,

const char \*type, const char \*compat);

参数from表示从哪一个节点开始寻找，传入NULL表示从根节点开始寻找。

参数compat是一个字符串，用来指定compatible属性的值；

参数type是一个字符串，用来指定device\_type属性的值，可以传入NULL。

**e. of\_find\_node\_by\_phandle**

根据phandle找到节点。

dts文件被编译为dtb文件时，每一个节点都有一个数字ID，这些数字ID彼此不同。可以使用数字ID来找到device\_node。这些数字ID就是phandle。

函数原型：

extern struct device\_node \*of\_find\_node\_by\_phandle(phandle handle);

参数from表示从哪一个节点开始寻找，传入NULL表示从根节点开始寻找。

**f. of\_get\_parent**

找到device\_node的父节点。

函数原型：

extern struct device\_node \*of\_get\_parent(const struct device\_node \*node);

参数from表示从哪一个节点开始寻找，传入NULL表示从根节点开始寻找。

**g. of\_get\_next\_parent**

这个函数名比较奇怪，怎么可能有“next parent”？

它实际上也是找到device\_node的父节点，跟of\_get\_parent的返回结果是一样的。

差别在于它多调用下列函数，把node节点的引用计数减少了1。这意味着调用of\_get\_next\_parent之后，你不再需要调用of\_node\_put释放node节点。

of\_node\_put(node);

函数原型：

extern struct device\_node \*of\_get\_next\_parent(struct device\_node \*node);

参数from表示从哪一个节点开始寻找，传入NULL表示从根节点开始寻找。

**h. of\_get\_next\_child**

取出下一个子节点。

函数原型：

extern struct device\_node \*of\_get\_next\_child(const struct device\_node \*node,

struct device\_node \*prev);

参数node表示父节点；

prev表示上一个子节点，设为NULL时表示想找到第1个子节点。

不断调用of\_get\_next\_child时，不断更新pre参数，就可以得到所有的子节点。

**i. of\_get\_next\_available\_child**

取出下一个“可用”的子节点，有些节点的status是“disabled”，那就会跳过这些节点。

函数原型：

struct device\_node \*of\_get\_next\_available\_child(const struct device\_node \*node,

struct device\_node \*prev);

参数node表示父节点；

prev表示上一个子节点，设为NULL时表示想找到第1个子节点。

**j. of\_get\_child\_by\_name**

根据名字取出子节点。

函数原型：

extern struct device\_node \*of\_get\_child\_by\_name(const struct device\_node \*node,

const char \*name);

参数node表示父节点；

name表示子节点的名字。

##### 11.7.3.2 找到属性

内核源码incldue/linux/of.h中声明了device\_node的操作函数，当然也包括属性的操作函数。

**a. of\_find\_property**

找到节点中的属性。

函数原型：

extern struct property \*of\_find\_property(const struct device\_node \*np,

const char \*name,

int \*lenp);

参数np表示节点，我们要在这个节点中找到名为name的属性。

lenp用来保存这个属性的长度，即它的值的长度。

在设备树中，节点大概是这样：

xxx\_node {

xxx\_pp\_name = “hello”;

};

上述节点中，“xxx\_pp\_name”就是属性的名字，值的长度是6。

##### 11.7.3.3 获取属性的值

**a. of\_get\_property**

根据名字找到节点的属性，并且返回它的值。

函数原型：

/\*

\* Find a property with a given name for a given node

\* and return the value.

\*/

const void \*of\_get\_property(const struct device\_node \*np, const char \*name,

int \*lenp)

参数np表示节点，我们要在这个节点中找到名为name的属性，然后返回它的值。

lenp用来保存这个属性的长度，即它的值的长度。

**b. of\_property\_count\_elems\_of\_size**

根据名字找到节点的属性，确定它的值有多少个元素(elem)。

函数原型：

\* of\_property\_count\_elems\_of\_size - Count the number of elements in a property

\*

\* @np: device node from which the property value is to be read.

\* @propname: name of the property to be searched.

\* @elem\_size: size of the individual element

\*

\* Search for a property in a device node and count the number of elements of

\* size elem\_size in it. Returns number of elements on sucess, -EINVAL if the

\* property does not exist or its length does not match a multiple of elem\_size

\* and -ENODATA if the property does not have a value.

\*/

int of\_property\_count\_elems\_of\_size(const struct device\_node \*np,

const char \*propname, int elem\_size)

参数np表示节点，我们要在这个节点中找到名为propname的属性，然后返回下列结果：

return prop->length / elem\_size;

在设备树中，节点大概是这样：

xxx\_node {

xxx\_pp\_name = <0x50000000 1024> <0x60000000 2048>;

};

调用of\_property\_count\_elems\_of\_size(np, “xxx\_pp\_name”, 8)时，返回值是2；

调用of\_property\_count\_elems\_of\_size(np, “xxx\_pp\_name”, 4)时，返回值是4。

**c. 读整数u32/u64**

函数原型为：

static inline int of\_property\_read\_u32(const struct device\_node \*np,

const char \*propname,

u32 \*out\_value);

extern int of\_property\_read\_u64(const struct device\_node \*np,

const char \*propname, u64 \*out\_value);

在设备树中，节点大概是这样：

xxx\_node {

name1 = <0x50000000>;

name2 = <0x50000000 0x60000000>;

};

调用of\_property\_read\_u32 (np, “name1”, &val)时，val将得到值0x50000000；

调用of\_property\_read\_u64 (np, “name2”, &val)时，val将得到值0x0x6000000050000000。

**d. 读某个整数u32/u64**

函数原型为：

extern int of\_property\_read\_u32\_index(const struct device\_node \*np,

const char \*propname,

u32 index, u32 \*out\_value);

在设备树中，节点大概是这样：

xxx\_node {

name2 = <0x50000000 0x60000000>;

};

调用of\_property\_read\_u32 (np, “name2”, 1, &val)时，val将得到值0x0x60000000。

**e. 读数组**

函数原型为：

int of\_property\_read\_variable\_u8\_array(const struct device\_node \*np,

const char \*propname, u8 \*out\_values,

size\_t sz\_min, size\_t sz\_max);

int of\_property\_read\_variable\_u16\_array(const struct device\_node \*np,

const char \*propname, u16 \*out\_values,

size\_t sz\_min, size\_t sz\_max);

int of\_property\_read\_variable\_u32\_array(const struct device\_node \*np,

const char \*propname, u32 \*out\_values,

size\_t sz\_min, size\_t sz\_max);

int of\_property\_read\_variable\_u64\_array(const struct device\_node \*np,

const char \*propname, u64 \*out\_values,

size\_t sz\_min, size\_t sz\_max);

在设备树中，节点大概是这样：

xxx\_node {

name2 = <0x50000012 0x60000034>;

};

上述例子中属性name2的值，长度为8。

调用of\_property\_read\_variable\_u8\_array (np, “name2”, out\_values, 1, 10)时，out\_values中将会保存这8个字节： 0x12,0x00,0x00,0x50,0x34,0x00,0x00,0x60。

调用of\_property\_read\_variable\_u16\_array (np, “name2”, out\_values, 1, 10)时，out\_values中将会保存这4个16位数值： 0x0012, 0x5000,0x0034,0x6000。

总之，这些函数要么能取到全部的数值，要么一个数值都取不到；

如果值的长度在sz\_min和sz\_max之间，就返回全部的数值；否则一个数值都不返回。

**f. 读字符串**

函数原型为：

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

const char \*\*out\_string);

返回节点np的属性(名为propname)的值，(\*out\_string)指向这个值，把它当作字符串。

### 11.8 怎么修改设备树文件

一个写得好的驱动程序, 它会尽量确定所用资源。

只把不能确定的资源留给设备树, 让设备树来指定。

根据原理图确定"驱动程序无法确定的硬件资源", 再在设备树文件中填写对应内容。

那么, 所填写内容的格式是什么?

#### 11.8.1 使用芯片厂家提供的工具

有些芯片，厂家提供了对应的设备树生成工具，可以选择某个引脚用于某些功能，就可以自动生成设备树节点。

你再把这些节点复制到内核的设备树文件里即可。

#### 11.8.2 看绑定文档

内核文档 Documentation/devicetree/bindings/

做得好的厂家也会提供设备树的说明文档

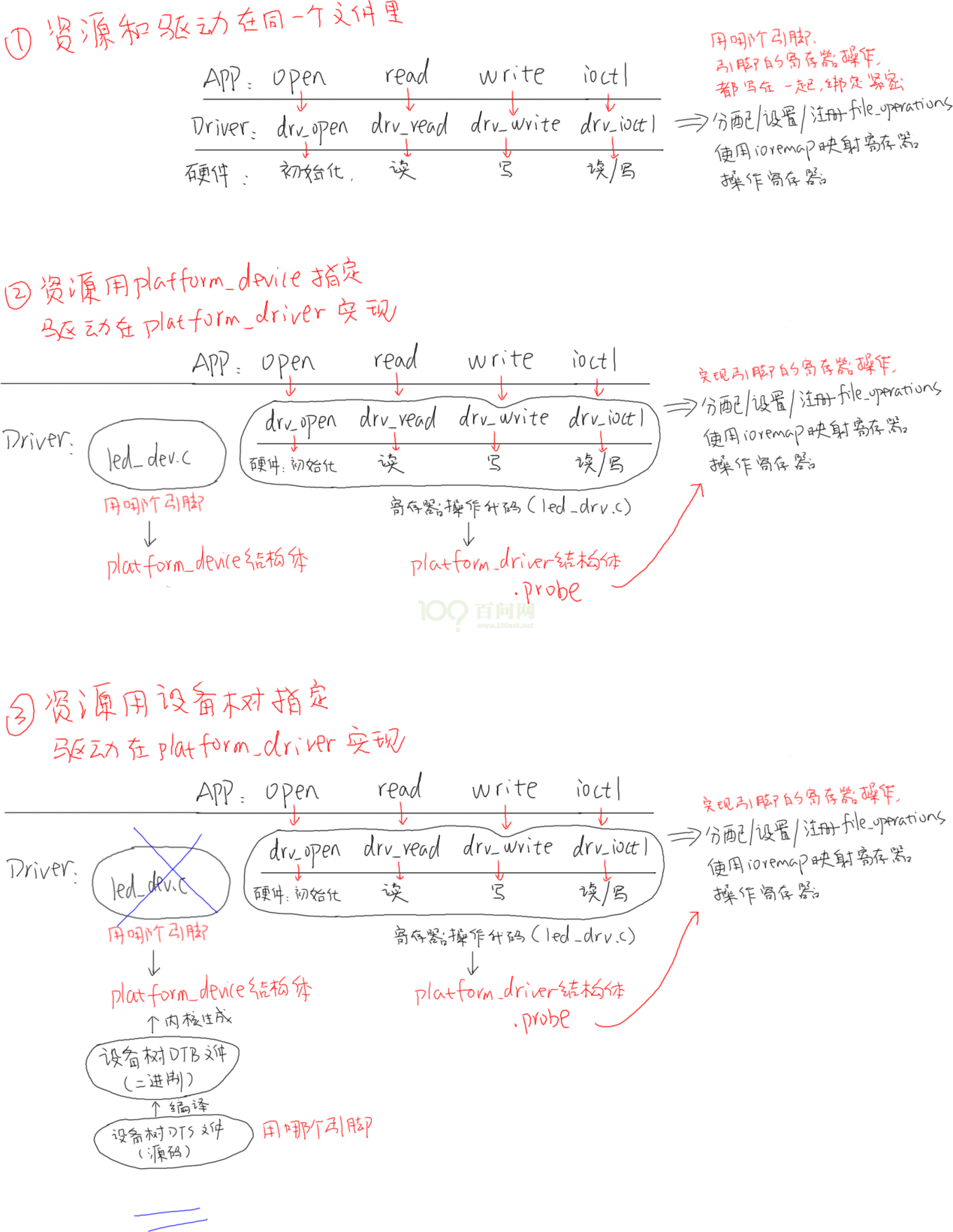
#### 11.8.3 参考同类型单板的设备树文件

#### 11.8.4 网上搜索

#### 11.8.5 实在没办法时, 只能去研究驱动源码

## 第十二章 LED模板驱动程序的改造：设备树

### 12.1 总结3种写驱动程序的方法



核心永远是file\_operations结构体。

上述三种方法，只是指定“硬件资源”的方式不一样。

从上图可以知道，platform\_device/platform\_driver只是编程的技巧，不涉及驱动的核心。

### 12.2 怎么使用设备树写驱动程序

#### 12.2.1 设备树节点要与platform\_driver能匹配

在我们的工作中，驱动要求设备树节点提供什么，我们就得按这要求去编写设备树。

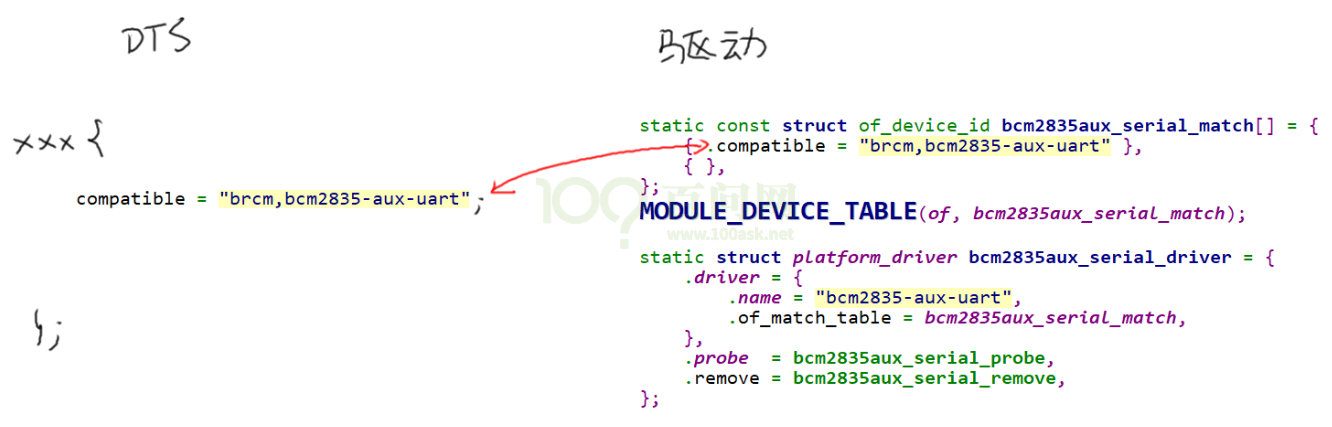
但是，匹配过程所要求的东西是固定的：

① 设备树要有compatible属性，它的值是一个字符串

② platform\_driver中要有of\_match\_table，其中一项的.compatible成员设置为一个字符串

③ 上述2个字符串要一致。

示例如下：



#### 12.2.2 设备树节点指定资源，platform\_driver获得资源

如果在设备树节点里使用reg属性，那么内核生成对应的platform\_device时会用reg属性来设置IORESOURCE\_MEM类型的资源。

如果在设备树节点里使用interrupts属性，那么内核生成对应的platform\_device时会用reg属性来设置IORESOURCE\_IRQ类型的资源。对于interrupts属性，内核会检查它的有效性，所以不建议在设备树里使用该属性来表示其他资源。

在我们的工作中，驱动要求设备树节点提供什么，我们就得按这要求去编写设备树。驱动程序中根据pin属性来确定引脚，那么我们就在设备树节点中添加pin属性。

设备树节点中：

#define GROUP\_PIN(g,p) ((g<<16) | (p))

100ask\_led0 {

compatible = “100ask,led”;

pin = <GROUP\_PIN(5, 3)>;

};

驱动程序中，可以从platform\_device中得到device\_node，再用of\_property\_read\_u32得到属性的值：

struct device\_node\* np = pdev->dev. of\_node;

int led\_pin;

int err = of\_property\_read\_u32(np, “pin”, &led\_pin);

### 12.3 开始编程

#### 12.3.1 修改设备树添加led设备节点

在本实验中，需要添加的设备节点代码是一样的，你需要找到你的单板所用的设备树文件，在它的根节点下添加如下内容：

#define GROUP\_PIN(g,p) ((g<<16) | (p))

100ask\_led@0 {

compatible = "100as,leddrv";

pin = <GROUP\_PIN(3, 1)>;

};

100ask\_led@1 {

compatible = "100as,leddrv";

pin = <GROUP\_PIN(5, 8)>;

};

##### 12.3.1.1 对百问网imx6ull Pro板

设备树文件是：内核源码目录中arch/arm/boot/dts/100ask\_imx6ull-14x14.dts

修改、编译后得到arch/arm/boot/dts/100ask\_imx6ull-14x14.dtb文件。

对于这款板子，本教程中我们使用SD卡上的系统。

要更换板上的设备树文件，你可以使用SD卡启动开发板后，更换这个文件：/boot/100ask\_imx6ull-14x14.dtb

##### 12.3.1.2 对百问网imx6ull MINI

设备树文件是：内核源码目录中arch/arm/boot/dts/100ask\_imx6ull\_mini.dts

修改、编译后得到arch/arm/boot/dts/100ask\_imx6ull\_mini.dtb文件。

对于这款板子，本教程中我们使用SD卡上的系统。

要更换板上的设备树文件，你可以使用SD卡启动开发板后，更换这个文件：/boot/100ask\_imx6ull\_mini.dtb

##### 12.3.1.3 对于百问网使用QEMU模拟的IMX6ULL板子

设备树文件是：内核源码目录中arch/arm/boot/dts/100ask\_imx6ul\_qemu.dts

修改、编译后得到arch/arm/boot/dts/100ask\_imx6ul\_qemu.dtb文件。

它是执行qemu时直接在命令行中指定设备树文件的，你可以打开脚本文件qemu-imx6ul-gui.sh找到dtb文件的位置，然后使用新编译出来的dtb去覆盖老文件。

#### 12.3.2 修改platform\_driver的源码

使用GIT下载所有源码后，本节源码位于如下目录：

01\_all\_series\_quickstart\

05\_嵌入式Linux驱动开发基础知识\source\

02\_led\_drv\05\_led\_drv\_template\_device\_tree

关键代码在chip\_demo\_gpio.c，主要看里面的platform\_driver，代码如下。

第166行指定了of\_match\_table，它是用来跟设备树节点匹配的，如果设备树节点中有compatile属性，并且其值等于第157行的“100as,leddrv”，就会导致第162行的probe函数被调用。

156 static const struct of\_device\_id ask100\_leds[] = {

157 { .compatible = "100as,leddrv" },

158 { },

159 };

160

161 static struct platform\_driver chip\_demo\_gpio\_driver = {

162 .probe = chip\_demo\_gpio\_probe,

163 .remove = chip\_demo\_gpio\_remove,

164 .driver = {

165 .name = "100ask\_led",

166 .of\_match\_table = ask100\_leds,

167 },

168 };

169

170 static int \_\_init chip\_demo\_gpio\_drv\_init(void)

171 {

172 int err;

173

174 err = platform\_driver\_register(&chip\_demo\_gpio\_driver);

175 register\_led\_operations(&board\_demo\_led\_opr);

176

177 return 0;

178 }

179

### 12.4 上机实验

1．使用新的设备树dtb文件启动单板，查看/sys/firmware/devicetree/base下有无节点

2. 查看/sys/devices/platform目录下有无对应的platform\_device

3. 加载驱动：

# insmod leddrv.ko

# insmod chip\_demo\_gpio.ko

4. 测试驱动：

# ./ledtest /dev/100ask\_led0 on

# ./ledtest /dev/100ask\_led0 off

### 12.5 调试技巧

/sys目录下有很多内核、驱动的信息：

#### 12.5.1 设备树的信息

以下目录对应设备树的根节点，可以从此进去找到自己定义的节点。

cd /sys/firmware/devicetree/base/

节点是目录，属性是文件。

属性值是字符串时，用cat命令可以打印出来；属性值是数值时，用hexdump命令可以打印出来。

#### 12.5.2 platform\_device的信息

以下目录含有注册进内核的所有platform\_device：

/sys/devices/platform

一个设备对应一个目录，进入某个目录后，如果它有“driver”子目录，就表示这个platform\_device跟某个platform\_driver配对了。

比如下面的结果中，平台设备“ff8a0000.i2s”已经跟平台驱动“rockchip-i2s”配对了：

/sys/devices/platform/ff8a0000.i2s]# ls driver -ld

lrwxrwxrwx 1 root root 0 Jan 18 16:28 driver -> ../../../bus/platform/drivers/rockchip-i2s

#### 12.5.3 platform\_driver的信息

以下目录含有注册进内核的所有platform\_driver：

/sys/bus/platform/drivers

一个driver对应一个目录，进入某个目录后，如果它有配对的设备，可以直接看到。

比如下面的结果中，平台驱动“rockchip-i2s”跟2个平台设备“平台设备“ff890000.i2s”、“ff8a0000.i2s”配对了：



注意：一个平台设备只能配对一个平台驱动，一个平台驱动可以配对多个平台设备。

### 12.6 课后作业

请仿照本节提供的程序(位于05\_led\_drv\_template\_device\_tree目录)，改造你所用的单板的LED驱动程序。