

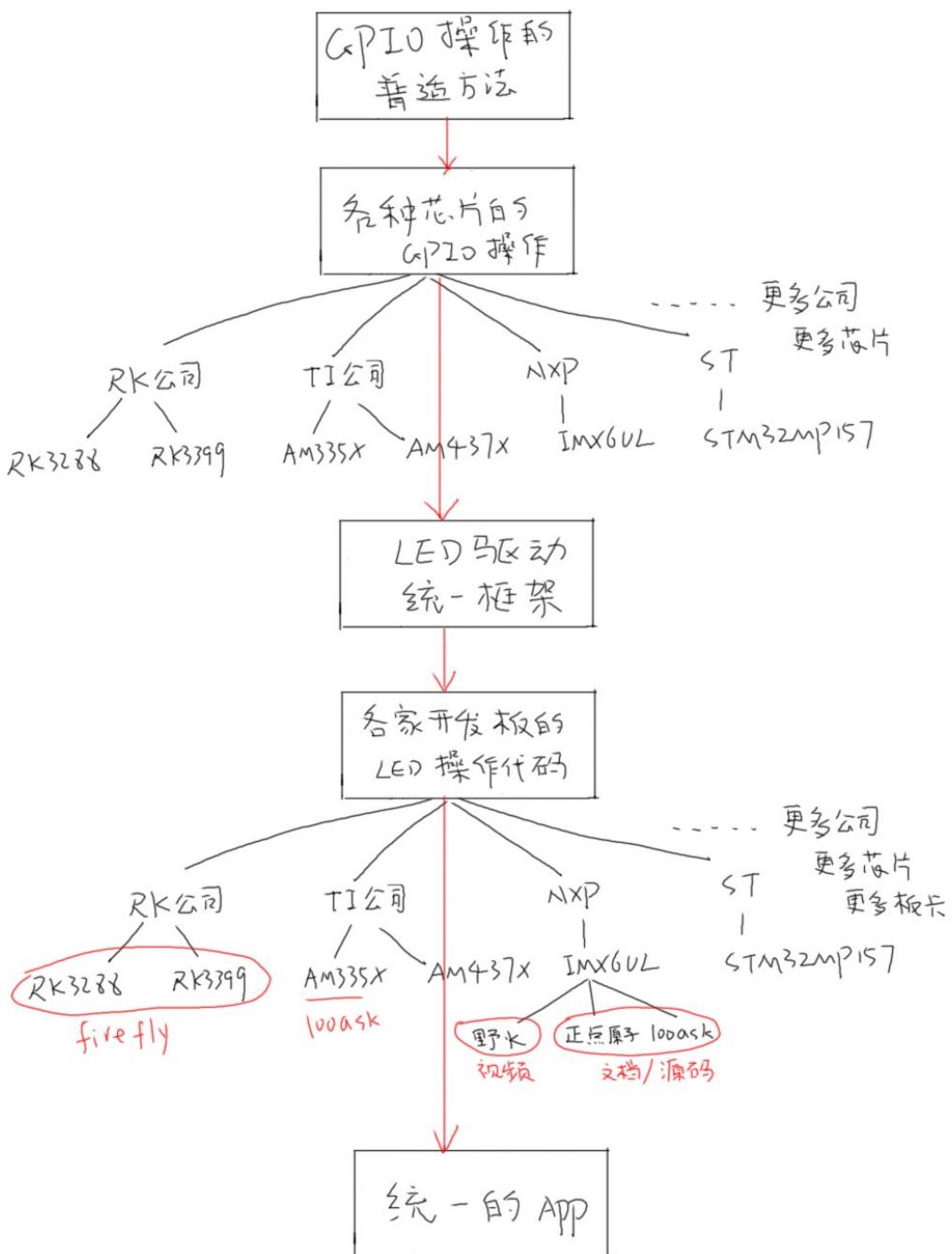
1. 嵌入式后 Linux 驱动开发基础知识的引导与说明

1.1 打算讲什么、怎么讲？

以几个简单的驱动程序，讲解嵌入式 Linux 驱动的框架，了解驱动开发的流程、方法，掌握从 APP 到驱动的调用流程。

会涉及很多种开发板，让你明白“Linux 驱动 = 软件框架 + 硬件操作”，让你“一通百通”，掌握了普适性的原理之后，在工作中很容易在其他板子使用这些知识。

以 LED 驱动为例，会如下讲解：

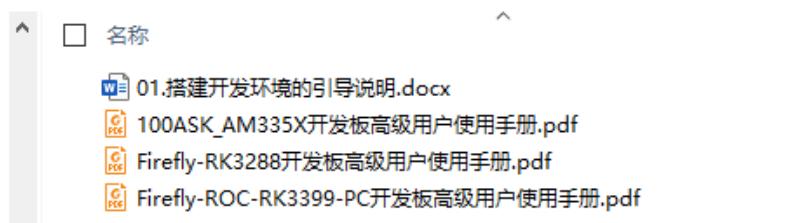


1.2 需要做什么准备工作

驱动程序依赖于 Linux 内核，你为开发板 A 开发驱动，那就先在 Ubuntu 中得到、配置、编译开发板 A 所使用的 Linux 内核。

请使用 git 下载本教程的文档、源码，查看如下目录中你所用开发板的高级用户使用手册(有些开发板的手册我们还没编写完，持续更新)：

01_all_series_quickstart > 03_高级手册对应的操作(搭环境等)



根据手册完成下面操作：

硬件部分：

- ① 开发板接线：串口线、电源线、网线
- ② 开发板烧写系统

软件部分：

- ① 下载 Linux 内核，Windows 和 Ubuntu 下各放一份
- ② Windows 下：使用 Source Insight 创建内核源码的工程，这是用来浏览内核、编辑驱动
- ③ Ubuntu 下：安装[工具链](#)，配置、编译[Linux 内核](#)

注意：git 的使用方法请参考 <http://wiki.100ask.net> 中的“初学者学习路线”：



2. Hello 驱动(不涉及硬件操作)

我们选用的内核都是 4.x 版本，操作都是类似的：

```
rk3399  linux 4.4.154
rk3288  linux 4.4.154
imx6ul   linux 4.9.88
am3358   linux 4.9.168
```

2.1 APP 打开的文件在内核中如何表示

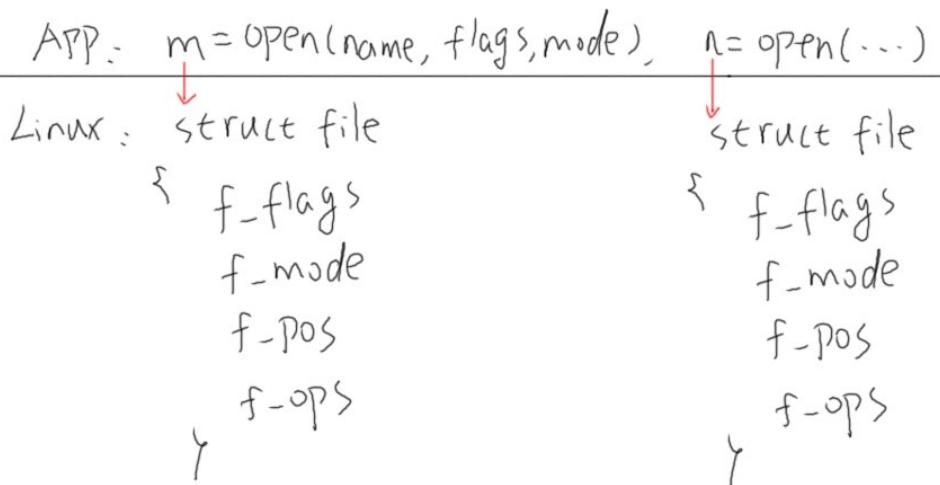
APP 打开文件时，可以得到一个整数，这个整数被称为文件句柄。对于 APP 的每一个文件句柄，在内核里面都有一个“struct file”与之对应。

```
894: struct file {
895:     union {
896:         struct llist_node    fu_llist;
897:         struct rcu_head      fu_rcuhead;
898:     } f_u;
899:     struct path        f_path;
900:     struct inode       *f_inode;    /* cached value */
901:     const struct file_operations  *f_op; ←
902:
903:     /*
904:      * Protects f_ep_links, f_flags.
905:      * Must not be taken from IRQ context.
906:      */
907:     spinlock_t      f_lock;
908:     atomic_long_t    f_count;
909:     unsigned int     f_flags; ←
910:     fmode_t         f_mode; ←
911:     struct mutex     f_pos_lock;
912:     loff_t          f_pos; ←
913:     struct fown_struct f_owner;
914:     const struct cred  *f_cred;
915:     struct file_ra_state   f_ra;
916:
```

可以猜测，我们使用 open 打开文件时，传入的 flags、mode 等参数会被记录在内核中对应的 struct file 结构体里(f_flags、f_mode)：

```
int open(const char *pathname, int flags, mode_t mode);
```

去读写文件时，文件的当前偏移地址也会保存在 struct file 结构体的 f_pos 成员里。



2.2 打开字符设备节点时，内核中也有对应的 struct file

注意这个结构体中的结构体：struct file_operations *f_op, 这是由驱动程序提供的。

```

894: struct file {
895:     union {
896:         struct llist_node    fu_llist;
897:         struct rcu_head      fu_rcuhead;
898:     } f_u;
899:     struct path          f_path;
900:     struct inode        *f_inode;    /* cached value */
901:     const struct file_operations *f_op; ←
902:

```



结构体 struct file_operations 的定义如下：

```

1674: struct file_operations {
1675:     struct module *owner;
1676:     loff_t (*llseek) (struct file *, loff_t, int);
1677:     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
1678:     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
1679:     ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
1680:     ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
1681:     int (*iterate) (struct file *, struct dir_context *);
1682:     unsigned int (*poll) (struct file *, struct poll_table_struct *);
1683:     long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
1684:     long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
1685:     int (* mmap) (struct file *, struct vm_area_struct *);
1686:     int (*open) (struct inode *, struct file *);
1687:     int (*flush) (struct file *, fl_owner_t id);
1688:     int (*release) (struct inode *, struct file *);
1689:     int (*fsync) (struct file *, loff_t, loff_t, int datasync);

```

2.3 请猜猜：

怎么编写驱动程序？

- ① 确定主设备号，也可以让内核分配
- ② 定义自己的 file_operations 结构体
- ③ 实现对应的 drv_open/drv_read/drv_write 等函数，填入 file_operations 结构体
- ④ 把 file_operations 结构体告诉内核：register_chrdev
- ⑤ 谁来注册驱动程序啊？得有一个入口函数：安装驱动程序时，就会去调用这个入口函数
- ⑥ 有入口函数就应该有出口函数：卸载驱动程序时，出口函数调用 unregister_chrdev
- ⑦ 其他完善：提供设备信息，自动创建设备节点：class_create, device_create

2.4 请不要啰嗦，表演你的代码吧

① 写驱动程序

参考 driver/char 中的程序，包含头文件，写框架，传输数据：

- A. 驱动中实现 open, read, write, release，APP 调用这些函数时，都打印内核信息
- B. APP 调用 write 函数时，传入的数据保存在驱动中
- C. APP 调用 read 函数时，把驱动中保存的数据返回给 APP

② 写测试程序

测试程序要实现写、读功能：

- A. ./hello_drv_test -w wiki.100ask.net // 把字符串“wiki.100ask.net”发给驱动程序
- B. ./hello_drv_test -r // 把驱动中保存的字符串读回来

③ 测试

- A. 编写驱动程序的 Makefile
- B. 上机实验

注意：如果安装驱动时提示 version magic 不匹配，请看本文档最后的“常见问题”。

2.5 Hello 驱动中的一些补充知识

- ① module_init/module_exit 的实现
- ② register_chrdev 的内部实现
- ③ class_destroy/device_create 浅析

3. 硬件知识_LED 原理图

当我们学习 C 语言的时候，我们会写个 Hello 程序。

那当我们写 ARM 程序，也该有一个简单的程序引领我们入门，这个程序就是点亮 LED。

我们怎样去点亮一个 LED 呢？

分为三步：

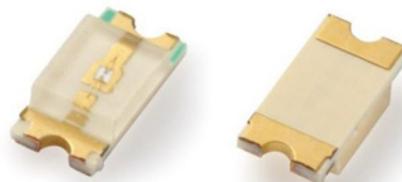
- 1.看原理图，确定控制 LED 的引脚；
- 2.看主芯片的芯片手册，确定如何设置控制这个引脚；
- 3.写程序；

3.1 先来讲讲怎么看原理图

LED 样子有很多种，像插脚的，贴片的。



插脚封装LED

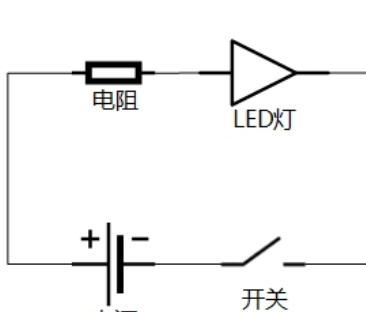


贴片封装LED

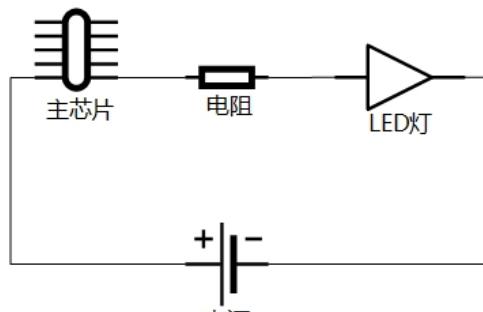
它们长得完全不一样，因此我们在原理图中将它抽象出来。

点亮 LED 需要通电源，同时为了保护 LED，加个电阻减小电流。

控制 LED 灯的亮灭，可以手动开关 LED，但在电子系统中，不可能让人来控制开关，通过编程，利用芯片的引脚去控制开关。



手动控制



编程自动控制

LED 的驱动方式，常见的有四种。

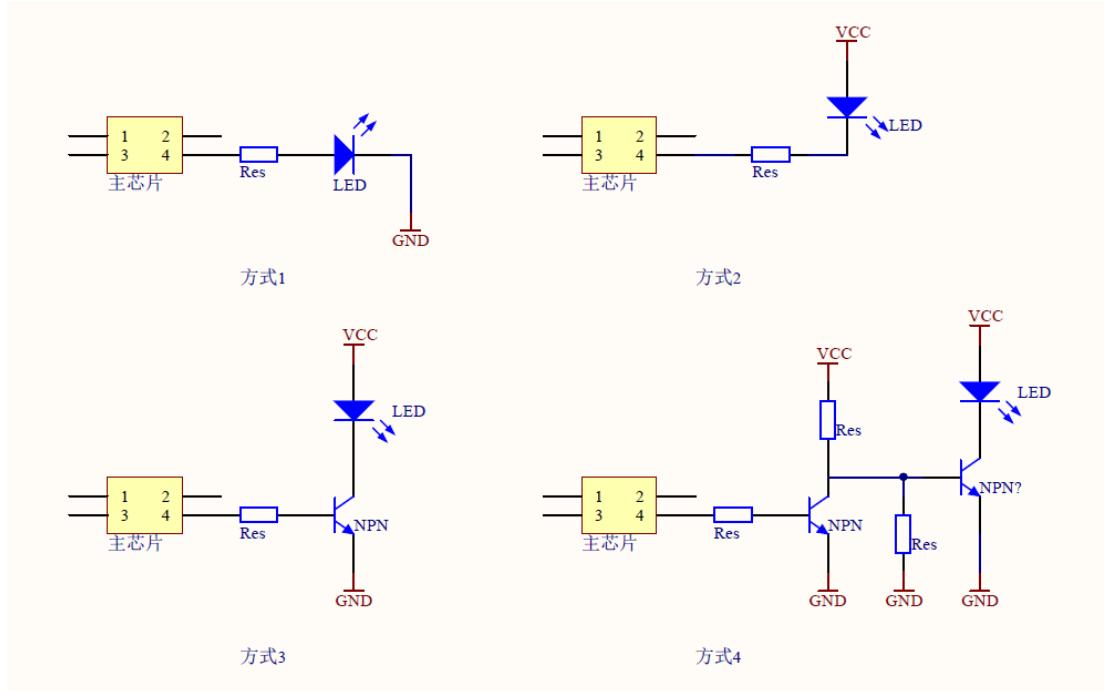
方式 1：使用引脚输出 3.3V 点亮 LED，输出 0V 熄灭 LED。

方式 2：使用引脚拉低到 0V 点亮 LED，输出 3.3V 熄灭 LED。

有的芯片为了省电等原因，其引脚驱动能力不足，这时可以使用三极管驱动。

方式 3：使用引脚输出 1.2V 点亮 LED，输出 0V 熄灭 LED。

方式 4：使用引脚输出 0V 点亮 LED，输出 1.2V 熄灭 LED。



由此，主芯片引脚输出高电平/低电平，即可改变 LED 状态，而无需关注 GPIO 引脚输出的是 3.3V 还是 1.2V。

所以简称输出 1 或 0：

逻辑 1-->高电平

逻辑 0-->低电平

4. 普通的 GPIO 引脚操作方法

GPIO: General-purpose input/output, 通用的输入输出口

4.1 GPIO 模块一般结构

- a. 有多组 GPIO, 每组有多个 GPIO
- b. 使能: 电源/时钟
- c. 模式(Mode): 引脚可用于 GPIO 或其他功能
- d. 方向: 引脚 Mode 设置为 GPIO 时, 可以继续设置它是输出引脚, 还是输入引脚
- e. 数值: 对于输出引脚, 可以设置寄存器让它输出高、低电平
对于输入引脚, 可以读取寄存器得到引脚的当前电平

4.2 GPIO 寄存器操作

- a. 芯片手册一般有相关章节, 用来介绍: power/clock
可以设置对应寄存器使能某个 GPIO 模块(Module)
有些芯片的 GPIO 是没有使能开关的, 即它总是使能的
- b. 一个引脚可以用于 GPIO、串口、USB 或其他功能,
有对应的寄存器来选择引脚的功能
- c. 对于已经设置为 GPIO 功能的引脚, 有方向寄存器用来设置它的方向: 输出、输入
- d. 对于已经设置为 GPIO 功能的引脚, 有数据寄存器用来写、读引脚电平状态

GPIO 寄存器的 2 种操作方法:

原则: 不能影响到其他位

- a. 直接读写: 读出、修改对应位、写入
要设置 bit n:

```
val = data_reg;  
val = val | (1<<n);  
data_reg = val;
```

要清除 bit n:

```
val = data_reg;  
val = val & ~(1<<n);  
data_reg = val;
```

- b. set-and-clear protocol:

set_reg, clr_reg, data_reg 三个寄存器对应的是同一个物理寄存器,
要设置 bit n: set_reg = (1<<n);
要清除 bit n: clr_reg = (1<<n);

4.3 GPIO 的其他功能: 防抖动、中断、唤醒:

后续章节再介绍

5. 具体单板的 GPIO 操作方法

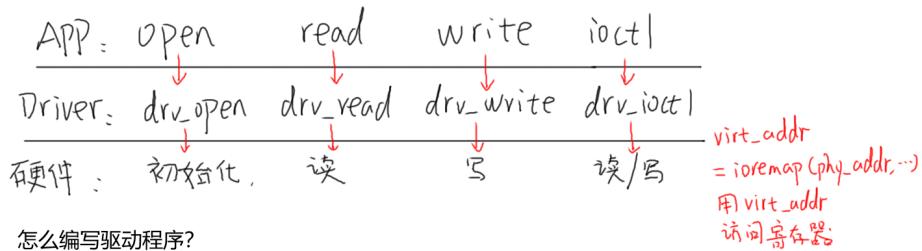
请使用 GIT 下载文档后，看下图红框所示目录中各板子对应的文档。
网盘中相同名字的目录下也有对应的视频。

- ▽ 01_all_series_quickstart
 - > .git
 - 00_视频体系介绍及引导
 - 01_使用Arduino操作体验简单开发
 - > 02_Linux基本操作与开发工具使用
 - 03_高级手册对应的操作(搭环境等)
- ▽ 04_快速入门(正式开始)
 - 00_快速入门总体介绍_讲什么_怎么讲
 - ▽ 01_嵌入式Linux应用开发基础知识
 - doc_pic
 - > source
 - ▽ 02_嵌入式Linux驱动开发基础知识
 - ▽ doc_pic
 - 05.具体单板的GPIO操作方法
 - > source

6. LED 驱动程序框架

注意：如果做实验安装驱动时提示 version magic 不匹配，请看本文档最后的“常见问题”。

6.1 回顾字符设备驱动程序框架



- ① 确定主设备号，也可以让内核分配
- ② 定义自己的 `file_operations` 结构体

它是核心
- ③ 实现对应的 `drv_open/drv_read/drv_write` 等函数，填入 `file_operations` 结构体
- ④ 把 `file_operations` 结构体告诉内核： `register_chrdev`
- ⑤ 谁来注册驱动程序啊？得有一个入口函数：安装驱动程序时，就会去调用这个入口函数
- ⑥ 有入口函数就应该有出口函数：卸载驱动程序时，出口函数调用 `unregister_chrdev`
- ⑦ 其他完善：提供设备信息，自动创建设备节点：`class_create, device_create`

6.2 对于 LED 驱动，我们想要什么样的接口？

用法： ledtest /dev/led0/1/2 on/off

APP: open("/dev/led0/1/2", ...), write(fd, val, 1)

Driver: led_open:

根据子设备号确定哪个LED
使能 GPIO 模块
而已置引脚为 output

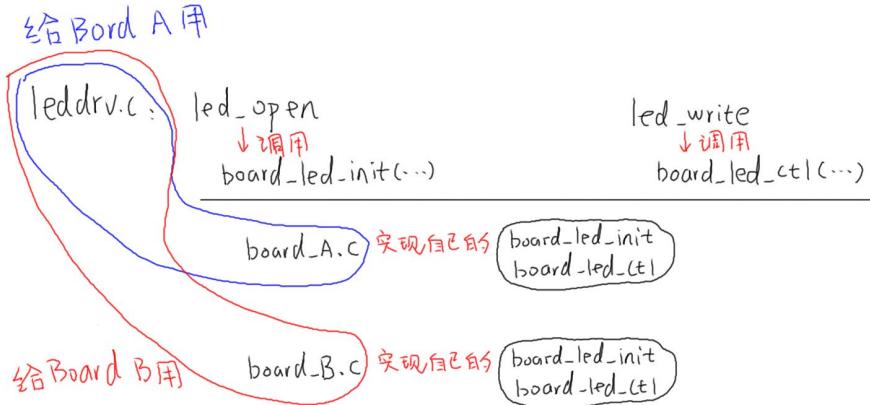
led_write

根据子设备号确定哪个LED
根据 val 设置引脚的电平

硬件：

6.3 LED 驱动要怎么写，才能支持多个板子？分层！

1. 把驱动拆分为通用的框架(leddrv.c)、具体的硬件操作(board_X.c):

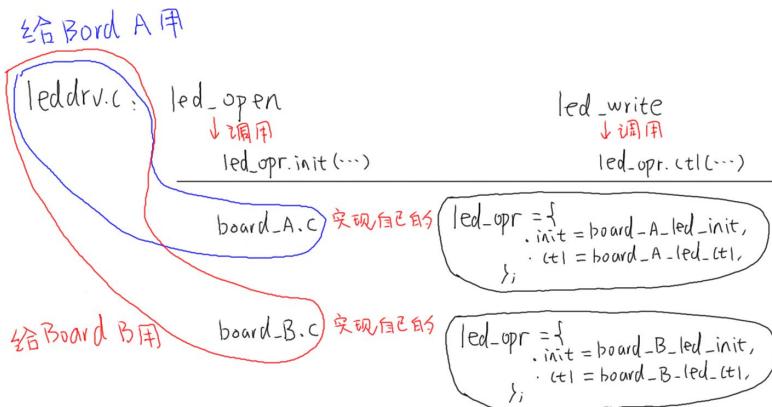


2. 以面向对象的思想，改进代码：

抽象出一个结构体：

```
struct led_operations {
    int (*init) (int which); /* 初始化LED, which-哪个LED */
    int (*ctl) (int which, int status); /* 控制LED, which-哪个LED, status:1-亮,0-灭 */
};
```

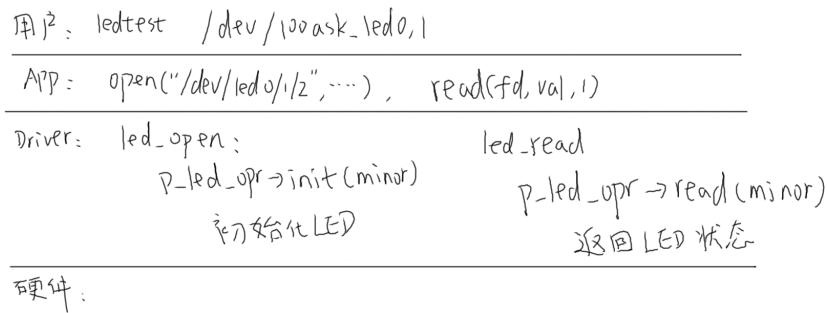
每个单板相关的 board_X.c 实现自己的 led_operations 结构体，供上层的 leddrv.c 调用：



6.4 写代码

6.5 课后作业

实现读 LED 状态的功能：涉及 APP 和驱动。



7. 具体单板的 LED 驱动程序

我们选用的内核都是 4.x 版本，操作都是类似的：

```
rk3399    linux 4.4.154
rk3288    linux 4.4.154
imx6ul    linux 4.9.88
am3358    linux 4.9.168
```

录制视频时，我的 source insight 里总是使用某个版本的内核。这没有关系，驱动程序中调用的内核函数，在这些 4.x 版本的内核里都是一样的。

7.0 怎么写 LED 驱动程序？

详细步骤如下：

- ① 看原理图确定引脚，确定引脚输出什么电平才能点亮/熄灭 LED
- ② 看主芯片手册，确定寄存器操作方法：哪些寄存器？哪些位？地址是？
- ③ 编写驱动：先写框架，再写硬件操作的代码

注意：在芯片手册中确定的寄存器地址被称为**物理地址**，在 Linux 内核中无法直接使用。

需要使用内核提供的 ioremap 把物理地址映射为**虚拟地址**，使用虚拟地址。

ioremap 函数的使用：

- ① 函数原型：

```
void __iomem *ioremap(resource_size_t res_cookie, size_t size)
```

使用时，要包含头文件：

```
#include <asm/io.h>
```

- ② 它的作用：

把物理地址 phys_addr 开始的一段空间(大小为 size)，映射为虚拟地址；返回值是该段虚拟地址的首地址。

```
virt_addr = ioremap(phsys_addr, size);
```

实际上，它是按页(4096 字节)进行映射的，是整页整页地映射的。

假设 phys_addr = 0x10002，size=4，ioremap 的内部实现是：

- a. phys_addr 按页取整，得到地址 0x10000
- b. size 按页取整，得到 4096
- c. 把起始地址 0x10000，大小为 4096 的这一块物理地址空间，映射到虚拟地址空间，
假设得到的虚拟空间起始地址为 0xf0010000
- d. 那么 phys_addr = 0x10002 对应的 virt_addr = 0xf0010002

- ③ 不再使用该段虚拟地址时，要 iounmap(virt_addr)：

```
void iounmap(volatile void __iomem *cookie)
```

volatile 的使用：

① 编译器很聪明，会帮我们做些优化，比如：

```
int a;  
a = 0; // 这句话可以优化掉，不影响 a 的结果  
a = 1;
```

② 有时候编译器会自作聪明，比如：

```
int *p = ioremap(xxxx, 4); // GPIO 寄存器的地址  
*p = 0; // 点灯，但是这句话被优化掉了  
*p = 1; // 灭灯
```

③ 对于上面的情况，为了避免编译器自动优化，需要加上 volatile，告诉它“这是容易出错的，别乱优化”：

```
volatile int *p = ioremap(xxxx, 4); // GPIO 寄存器的地址  
*p = 0; // 点灯，这句话不会被优化掉  
*p = 1; // 灭灯
```

7.1 AM335X 的 LED 驱动程序

7.1.1 原理图

100ask_AM335X 开发板结构为：底板+核心板，其中一个 LED 原理图如下：



7.1.2 所涉及的寄存器操作

a. 使能 GPIO1

```
/* set PRCM to enable GPIO1
 * set CM_PER_GPIO1_CLKCTRL (0x44E00000 + 0xAC)
 * val: (1<<18) | 0x2
 */
```

b. 设置 GPIO1_16 的功能，让它工作于 GPIO 模式

```
/* set Control Module to set GPIO1_16 (R13) used as GPIO
 * conf_gpmc_ad0 as mode 7
 * addr : 0x44E10000 + 0x800
 * val : 7
 */
```

c. 设置 GPIO1_16 的方向，让它作为输出引脚

```
/* set GPIO1's registers , to set GPIO1_16'S dir (output)
 * GPIO_OE
 * addr : 0x4804C000 + 0x134
 * clear bit 16
 */
```

d. 设置 GPIO1_16 的数据，让它输出高电平

AM335X 芯片支持 set-and-clear protocol，设置 GPIO_SETDATAOUT 的 bit 16 为 1 即可让引脚输出 1：

```
/* set GPIO1_16's registers , to output 1
 * GPIO_SETDATAOUT
 * addr : 0x4804C000 + 0x194
 */
```

e. 清除 GPIO1_16 的数据，让它输出低电平

AM335X 芯片支持 set-and-clear protocol，设置 GPIO_CLEARDATAOUT 的 bit 16 为 1 即可让引脚输出 0：

```
/* set GPIO1_16's registers , to output 0
 * GPIO_CLEARDATAOUT
 * addr : 0x4804C000 + 0x190
 */
```

7.1.3 写程序

7.1.4 为避免内核原来的 LED 驱动干扰实验，怎么配置内核去掉原有驱动？

7.1.5 课后作业

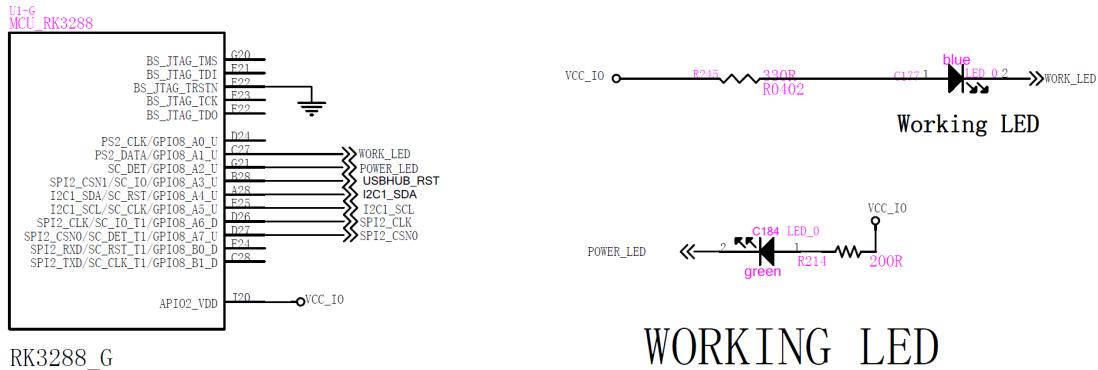
- 在 board_am335x.c 里有 ioremap，什么时候执行 iounmap？请完善程序
- 视频里我们只实现了点一个 LED，请修改代码实现操作 4 个 LED

7.2 RK3288 和 RK3399 的 LED 驱动程序

7.2.1 原理图

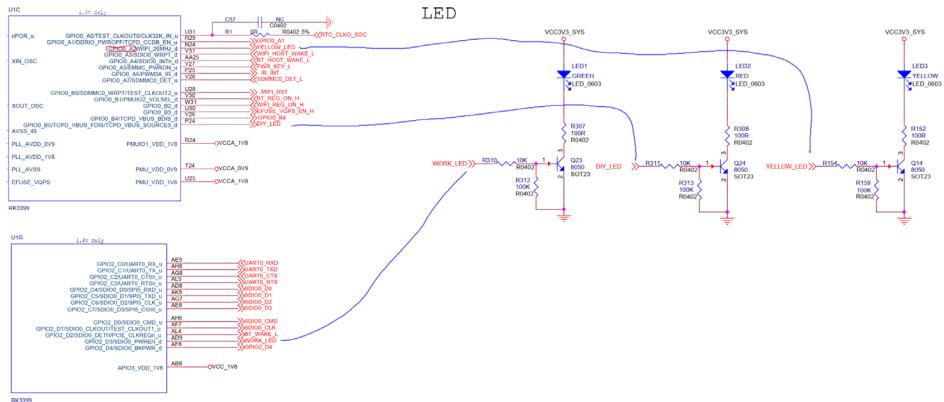
a) firefly RK3288 的 LED 原理图:

RK3288 开发板上有 2 个 LED，原理图如下，其中的 WORK_LED 使用引脚 GPIO8_A1：



b) firefly RK3399 的 LED 原理图:

RK3399 开发板上有 3 个 LED，原理图如下，其中的 WORK_LED 使用引脚 GPIO2_D3：



7.2.2 所涉及的寄存器操作

截图便于对比，后面有文字便于复制：

```

/* rk3288 GPIO8_A1 */
/* a. 使能GPIO8
 * set CRU to enable GPIO8
 * CRU_CLKGATE14_CON 0xFF760000 + 0x198
 * (1<<(8+16)) | (0<<8)
 */

/* b. 设置GPIO8_A1用于GPIO
 * set PMU/GRF to configure GPIO8_A1 as GPIO
 * GRF_GPIO8A_IOMUX 0xFF770000 + 0x0080
 * bit[3:2] = 0b00
 * (3<<(2+16)) | (0<<2)
 */

/* c. 设置GPIO8_A1作为output引脚
 * set GPIO_SWPORTA_DDR to configure GPIO8_A1 as output
 * GPIO_SWPORTA_DR 0xFF7F0000 + 0x0004
 * bit[1] = 0b1
 */

/* d. 设置GPIO8_A1输出高电平
 * set GPIO_SWPORTA_DR to configure GPIO8_A1 output 1
 * GPIO_SWPORTA_DR 0xFF7F0000 + 0x0000
 * bit[1] = 0b1
 */

/* e. 设置GPIO8_A1输出低电平
 * set GPIO_SWPORTA_DR to configure GPIO8_A1 output 0
 * GPIO_SWPORTA_DR 0xFF7F0000 + 0x0000
 * bit[1] = 0b0
 */

/* f. 设置GPIO2_D3 */
/* a. 使能GPIO2_D3
 * set CRU to enable GPIO2_D3
 * CRU_CLKGATE14_CON 0xFF760000 + 0x037c
 * (1<<(3+16)) | (0<<6)
 */

/* b. 设置GPIO2_D3用于GPIO
 * set PMU/GRF to configure GPIO2_D3 as GPIO
 * GRF_GPIO2D_IOMUX 0xFF770000 + 0x0e00c
 * bit[7:6] = 0b00
 * (3<<(6+16)) | (0<<6)
 */

/* c. 设置GPIO2_D3作为output引脚
 * set GPIO_SWPORTA_DDR to configure GPIO2_D3 as output
 * GPIO_SWPORTA_DR 0xFF780000 + 0x0004
 * bit[1] = 0b1
 */

/* d. 设置GPIO2_D3输出高电平
 * set GPIO_SWPORTA_DR to configure GPIO2_D3 output 1
 * GPIO_SWPORTA_DR 0xFF780000 + 0x0000
 * bit[1] = 0b1
 */

/* e. 设置GPIO2_D3输出低电平
 * set GPIO_SWPORTA_DR to configure GPIO2_D3 output 0
 * GPIO_SWPORTA_DR 0xFF780000 + 0x0000
 * bit[1] = 0b0
 */

```

a. 对于 RK3288 的 GPIO8_A1 引脚：

```

/* rk3288 GPIO8_A1 */
/* a. 使能 GPIO8
 * set CRU to enable GPIO8
 * CRU_CLKGATE14_CON 0xFF760000 + 0x198
 * (1<<(8+16)) | (0<<8)
 */

/* b. 设置 GPIO8_A1 用于 GPIO
 * set PMU/GRF to configure GPIO8_A1 as GPIO
 * GRF_GPIO8A_IOMUX 0xFF770000 + 0x0080
 * bit[3:2] = 0b00
 * (3<<(2+16)) | (0<<2)
 */

/* c. 设置 GPIO8_A1 作为 output 引脚
 * set GPIO_SWPORTA_DDR to configure GPIO8_A1 as output
 * GPIO_SWPORTA_DR 0xFF7F0000 + 0x0004
 * bit[1] = 0b1
 */

/* d. 设置 GPIO8_A1 输出高电平
 * set GPIO_SWPORTA_DR to configure GPIO8_A1 output 1
 * GPIO_SWPORTA_DR 0xFF7F0000 + 0x0000
 * bit[1] = 0b1
 */

/* e. 设置 GPIO8_A1 输出低电平
 * set GPIO_SWPORTA_DR to configure GPIO8_A1 output 0
 * GPIO_SWPORTA_DR 0xFF7F0000 + 0x0000
 * bit[1] = 0b0
 */

/* f. 设置 GPIO2_D3 */
/* a. 使能 GPIO2_D3
 * set CRU to enable GPIO2_D3
 * CRU_CLKGATE14_CON 0xFF760000 + 0x037c
 * (1<<(3+16)) | (0<<6)
 */

/* b. 设置 GPIO2_D3 用于 GPIO
 * set PMU/GRF to configure GPIO2_D3 as GPIO
 * GRF_GPIO2D_IOMUX 0xFF770000 + 0x0e00c
 * bit[7:6] = 0b00
 * (3<<(6+16)) | (0<<6)
 */

/* c. 设置 GPIO2_D3 作为 output 引脚
 * set GPIO_SWPORTA_DDR to configure GPIO2_D3 as output
 * GPIO_SWPORTA_DR 0xFF780000 + 0x0004
 * bit[1] = 0b1
 */

/* d. 设置 GPIO2_D3 输出高电平
 * set GPIO_SWPORTA_DR to configure GPIO2_D3 output 1
 * GPIO_SWPORTA_DR 0xFF780000 + 0x0000
 * bit[1] = 0b1
 */

/* e. 设置 GPIO2_D3 输出低电平
 * set GPIO_SWPORTA_DR to configure GPIO2_D3 output 0
 * GPIO_SWPORTA_DR 0xFF780000 + 0x0000
 * bit[1] = 0b0
 */

```

b. 对于 RK3399 的 GPIO2_D3 引脚:

```
/* rk3399 GPIO2_D3 */
/* a. 使能 GPIO2
 * set CRU to enable GPIO2
 * CRU_CLKGATE_CON31 0xFF760000 + 0x037c
 * (1<<(3+16)) | (0<<3)
 */

/* b. 设置 GPIO2_D3 用于 GPIO
 * set PMU/GRF to configure GPIO2_D3 as GPIO
 * GRF_GPIO2D_IOMUX 0xFF770000 + 0x0e00c
 * bit[7:6] = 0b00
 * (3<<(6+16)) | (0<<6)
 */

/* c. 设置 GPIO2_D3 作为 output 引脚
 * set GPIO_SWPORTA_DDR to configure GPIO2_D3 as output
 * GPIO_SWPORTA_DDR 0xFF780000 + 0x0004
 * bit[27] = 0b1
 */

/* d. 设置 GPIO2_D3 输出高电平
 * set GPIO_SWPORTA_DR to configure GPIO2_D3 output 1
 * GPIO_SWPORTA_DR 0xFF780000 + 0x0000
 * bit[27] = 0b1
 */

/* e. 设置 GPIO2_D3 输出低电平
 * set GPIO_SWPORTA_DR to configure GPIO2_D3 output 0
 * GPIO_SWPORTA_DR 0xFF780000 + 0x0000
 * bit[27] = 0b0
 */
```

7.2.3 写程序

7.2.4 上机实验

rk3288:

```
insmod 100ask_led.ko
./ledtest /dev/100ask_led0 on
./ledtest /dev/100ask_led0 off
```

rk3399:

要先禁止内核中原来的 LED 驱动，把“heartbeat”功能关闭，执行以下命令即可：

```
echo none > /sys/class/leds/firefly\:yellow\:heartbeat/trigger
echo none > /sys/class/leds/firefly\:yellow\:user/trigger
echo none > /sys/class/leds/firefly\:red\:power/trigger
```

这样就可以使用我们的驱动程序做实验了：

```
insmod 100ask_led.ko
./ledtest /dev/100ask_led0 on
./ledtest /dev/100ask_led0 off
```

如果想恢复原来的心跳功能，可以执行：

```
echo heartbeat > /sys/class/leds/firefly\:yellow\:heartbeat/trigger
echo heartbeat > /sys/class/leds/firefly\:yellow\:user/trigger
echo heartbeat > /sys/class/leds/firefly\:red\:power/trigger
```

7.2.5 课后作业

- 在驱动里有 ioremap，什么时候执行 iounmap？请完善程序
- 视频里我们只实现了点一个 LED，修改代码实现操作所有 LED

7.3 野火/正点原子 IMX6UL/6ULL 的 LED 驱动程序

野火、正点原子用的内核版本是 4.1.15，

我们用的内核版本是 linux 4.9.88，

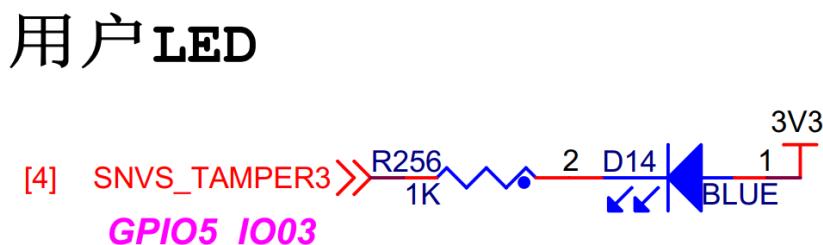
都是 4.x 版本，在学习上没有任何差别。

你拿到板子后，可以使用他们出厂的系统，

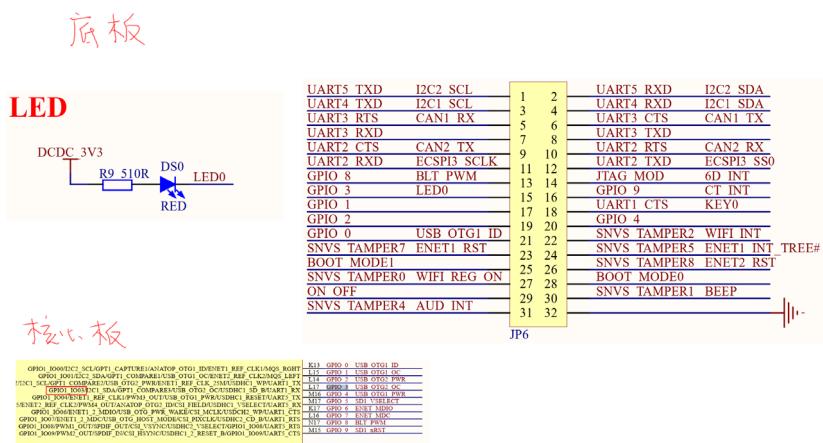
也可以根据我们提供的高级用户手册更改为我们的系统。

7.3.1 原理图

a) 野火 fire_imx6ull-pro 开发板的 LED 原理图，它使用 GPIO5_IO03:



b) 正点原子 Atk_imx6ull-alpha 开发板的 LED 原理图，它使用 GPIO1_IO03:



7.3.2 所涉及的寄存器操作

截图便于对比，后面有文字便于复制：

```

/* GPIO5 IO03 */
/* a. 使能GPIO5
 * set CCM to enable GPIO5
 * CCM_CGCR1[CG15] 0x20C406C
 * bit[31:30] = 0b11
 */

/* b. 设置GPIO5_IO03用于GPIO
 * set IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3
 * to configure GPIO5_IO03 as GPIO
 * IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 0x2290014 ①
 * bit[3:0] = 0b0101 alt5
 */

/* b. 设置GPIO5_IO03作为output引脚
 * set GPIO5_GDIR to configure GPIO5_IO03 as output ②
 * GPIO5_GDIR 0x020AC000 + 0x4
 * bit[3] = 0b1
 */

/* d. 设置GPIO5_DR输出低电平
 * set GPIO5_DR to configure GPIO5_IO03 output 0
 * GPIO5_DR 0x020AC000 + 0
 * bit[3] = 0b0
 */

/* e. 设置GPIO5_IO03输出高电平
 * set GPIO5_DR to configure GPIO5_IO03 output 1
 * GPIO5_DR 0x020AC000 + 0
 * bit[3] = 0b1
 */

/* a. 使能GPIO1
 * set CCM to enable GPIO1
 * CCM_CGCR1[CG13] 0x20C406C
 * bit[27:26] = 0b11
 */

/* b. 设置GPIO1_IO03用于GPIO
 * set IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03
 * to configure GPIO1_IO03 as GPIO
 * IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03 0x20E0068
 * bit[3:0] = 0b0101 alt5
 */

/* c. 设置GPIO1_IO03作为output引脚
 * set GPIO1_GDIR to configure GPIO1_IO03 as output
 * GPIO1_GDIR 0x0209C000 + 0x4
 * bit[3] = 0b1
 */

/* d. 设置GPIO1_DR输出低电平
 * set GPIO1_DR to configure GPIO1_IO03 output 0
 * GPIO1_DR 0x0209C000 + 0
 * bit[3] = 0b0
 */

/* e. 设置GPIO1_IO03输出高电平
 * set GPIO1_DR to configure GPIO1_IO03 output 1
 * GPIO1_DR 0x0209C000 + 0
 * bit[3] = 0b1
 */

```

只有3个寄存器地址不同
这个寄存器涉及的位不同

a. 对于野火 fire_imx6ull-pro 开发板的使用的 GPIO5_IO3 引脚：

```

/* GPIO5_IO03 */
/* a. 使能 GPIO5
 * set CCM to enable GPIO5
 * CCM_CGCR1[CG15] 0x20C406C
 * bit[31:30] = 0b11
 */

/* b. 设置 GPIO5_IO03 用于 GPIO
 * set IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3
 * to configure GPIO5_IO03 as GPIO
 * IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 0x2290014
 * bit[3:0] = 0b0101 alt5
 */

/* b. 设置 GPIO5_IO03 作为 output 引脚
 * set GPIO5_GDIR to configure GPIO5_IO03 as output
 * GPIO5_GDIR 0x020AC000 + 0x4
 * bit[3] = 0b1
 */

/* d. 设置 GPIO5_DR 输出低电平
 * set GPIO5_DR to configure GPIO5_IO03 output 0
 * GPIO5_DR 0x020AC000 + 0
 * bit[3] = 0b0
 */

/* e. 设置 GPIO5_IO03 输出高电平
 * set GPIO5_DR to configure GPIO5_IO03 output 1
 * GPIO5_DR 0x020AC000 + 0
 * bit[3] = 0b1
 */

```

```
* set GPIO5_DR to configure GPIO5_IO03 output 1
* GPIO5_DR 0x020AC000 + 0
* bit[3] = 0b1
*/
```

b. 对于正点原子 Atk_imx6ull-alpha 开发板的使用的 GPIO1_IO03 引脚:

```
/* GPIO1_IO03 */
/* a. 使能 GPIO1
 * set CCM to enable GPIO1
 * CCM_CGCR1[CG13] 0x20C406C
 * bit[27:26] = 0b11
*/
/* b. 设置 GPIO1_IO03 用于 GPIO
 * set IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03
 *      to configure GPIO1_IO03 as GPIO
 * IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03 0x20E0068
 * bit[3:0] = 0b0101 alt5
*/
/* c. 设置 GPIO1_IO03 作为 output 引脚
 * set GPIO1_GDIR to configure GPIO1_IO03 as output
 * GPIO1_GDIR 0x0209C000 + 0x4
 * bit[3] = 0b1
*/
/* d. 设置 GPIO1_DR 输出低电平
 * set GPIO1_DR to configure GPIO1_IO03 output 0
 * GPIO1_DR 0x0209C000 + 0
 * bit[3] = 0b0
*/
/* e. 设置 GPIO1_IO03 输出高电平
 * set GPIO1_DR to configure GPIO1_IO03 output 1
 * GPIO1_DR 0x0209C000 + 0
 * bit[3] = 0b1
*/
```

7.3.3 写程序

7.3.4 上机实验

a. 对于野火 fire_imx6ull-pro 开发板：

使用我们的系统时，按如下操作。

注意：如果要使用板子自带的系统，怎么关闭原有的驱动，以后我再来更新文档。

要先禁止内核中原来的 LED 驱动，把“heartbeat”功能关闭，执行以下命令即可：

```
echo none > /sys/class/leds/cpu/trigger
```

这样就可以使用我们的驱动程序做实验了：

```
insmod 100ask_led.ko
./ledtest /dev/100ask_led0 on
./ledtest /dev/100ask_led0 off
```

如果想恢复原来的心跳功能，可以执行：

```
echo heartbeat > /sys/class/leds/cpu/trigger
```

b. 对于正点原子 Atk_imx6ull-alpha 开发板：

使用我们的系统时，按如下操作。

注意：如果要使用板子自带的系统，怎么关闭原有的驱动，以后我再来更新文档。

要先禁止内核中原来的 LED 驱动，把“heartbeat”功能关闭，执行以下命令即可：

```
echo none > /sys/class/leds/sys-led/trigger
```

这样就可以使用我们的驱动程序做实验了：

```
insmod 100ask_led.ko
./ledtest /dev/100ask_led0 on
./ledtest /dev/100ask_led0 off
```

如果想恢复原来的心跳功能，可以执行：

```
echo heartbeat > /sys/class/leds/sys-led/trigger
```

7.3.5 课后作业

a. 在驱动里有 ioremap，什么时候执行 iounmap？请完善程序

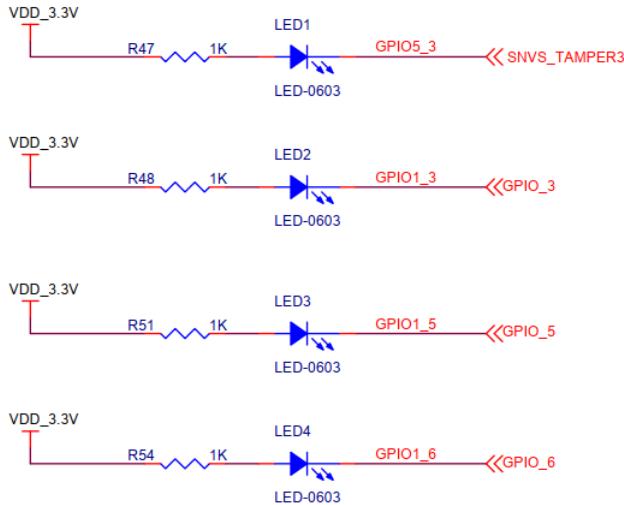
b. 视频里我们只实现了点一个 LED，开发板上也只有一个 LED，
所以，请修改代码操作蜂鸣器

7.4 百问网 IMX6ULL-QEMU 的 LED 驱动程序

使用 QEMU 模拟的硬件，它的硬件资源可以随意扩展。

在 IMX6ULL QEMU 虚拟开发板上，我们为它设计了 4 个 LED。

7.4.1 看原理图确定引脚及操作方法



从上图可知，这 4 个 LED 用到了 GPIO5_3、GPIO1_3、GPIO1_5、GPIO1_6 共 4 个引脚。

在芯片手册里，这些引脚的名字是：GPIO5_IO03、GPIO1_IO03、GPIO1_IO05、GPIO1_IO06。可以根据名字搜到对应的寄存器。

当这些引脚输出低电平时，对应的 LED 被点亮；输出高电平时，LED 熄灭。

7.4.2 所涉及的寄存器操作

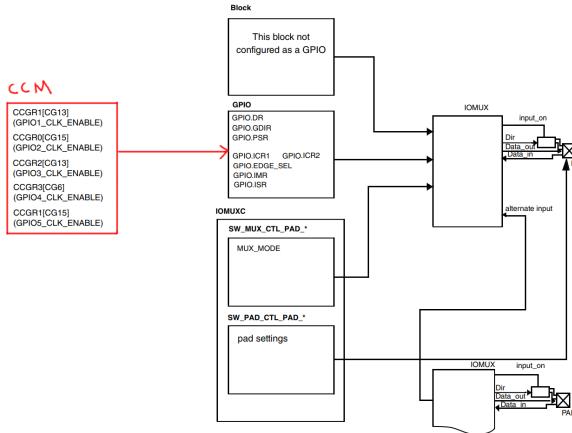


Figure 26-1. Chip IOMUX Scheme

步骤 1：使能 GPIO1、GPIO5

| Address: 20C_4000h base + 6Ch offset = 20C_406Ch | | | | | | | | | | | | | | | | |
|--|------|------|------|------|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Bit | 31 | 30 | 29 | 28 | 27 | 28 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| R | CG15 | CG14 | CG13 | CG12 | CG11 | CG10 | CG9 | CG8 | CG7 | CG6 | CG5 | CG4 | CG3 | CG2 | CG1 | CG0 |
| W | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Reset 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Bit 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Reset 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

CCM_CCGR1 field descriptions

| Field | Description |
|---------------|--|
| 31–30 CG15 | gpio5 clock (gpio5_clk_enable) |
| 29–28 CG14 | csu clock (csu_clk_enable) |
| 27–26 CG13 | gpio1 clock (gpio1_clk_enable) |
| 25–24 CG12 | uart4 clock (uart4_clk_enable) |
| 23–22 CG11 | gpt serial clock (gpt_serial_clk_enable) |
| 21–20 CG10 | gpt bus clock (gpt_clk_enable) |
| 19–18 CG9 | sim_s clock (sim_s_clk_enable) |
| 17–16 CG8 | adc1 clock (adc1_clk_enable) |
| 15–14 CG7 | epit2 clocks (epit2_clk_enable) |
| 13–12 CG6 | epit1 clocks (epit1_clk_enable) |
| 11–10 CG5 | uart3 clock (uart3_clk_enable) |
| 9–8 CG4 | adc2 clock (adc2_clk_enable) |
| 7–6 CG3 | ecspi4 clocks (ecspi4_clk_enable) |
| 5–4 CG2 | ecspi3 clocks (ecspi3_clk_enable) |
| 3–2 CG1 | ecspi2 clocks (ecspi2_clk_enable) |
| CG0 | ecspi1 clocks (ecspi1_clk_enable) |

设置 b[31:30]、b[27:26]就可以使能 GPIO5、GPIO1，设置为什么值呢？

看下图，设置为 0b11：

| CGR value | Clock Activity Description |
|-----------|---|
| 00 | Clock is off during all modes. Stop enter hardware handshake is disabled. |
| 01 | Clock is on in run mode, but off in WAIT and STOP modes |
| 10 | Not applicable (Reserved). |
| 11 | Clock is on during all modes, except STOP mode. |

- ① 00：该 GPIO 模块全程被关闭
- ② 01：该 GPIO 模块在 CPU run mode 情况下是使能的；在 WAIT 或 STOP 模式下，关闭
- ③ 10：保留
- ④ 11：该 GPIO 模块全程使能

步骤 2：设置 GPIO5_IO03、GPIO1_IO03、GPIO1_IO05、GPIO1_IO06 为 GPIO 模式

① 对于 GPIO5_IO03，设置如下寄存器：

Address: 229_0000h base + 14h offset = 229_0014h

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|-----------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Reserved | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reserved | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SION MUX_MODE | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER3 field descriptions

| Field | Description |
|----------|---|
| 31-5 | This field is reserved. |
| - | Reserved |
| 4 SION | Software Input On Field. Force the selected mux mode Input path no matter of MUX_MODE functionality. 1 ENABLED — Force input path of pad SNVS_TAMPER3 0 DISABLED — Input Path is determined by functionality |
| MUX_MODE | Mux Mode Select Field NOTE: ALT5 mode is only valid when TAMPER PIN is used as GPIO. This depends on FUSE setting "TAMPER_PIN_DISABLE[1:0]". Following is the mux information when TAMPER PIN is used as GPIO: SNVS_TAMPER3 ==> GPIO5_03 101 ALT5 — Select mux mode: ALT5 mux port: GPIO5_IO03 of instance - gpio Other Reserved |

② 对于 GPIO1_IO03，设置如下寄存器：

Address: 20E_0000h base + 68h offset = 20E_0068h

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Reserved | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reserved | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| SION MUX_MODE | | | | | | | | | | | | | | | | |
| IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03 field descriptions | | | | | | | | | | | | | | | | |
| Field | Description | | | | | | | | | | | | | | | |
| 31-5 | This field is reserved. | | | | | | | | | | | | | | | |
| - | Reserved | | | | | | | | | | | | | | | |
| 4 SION | Software Input On Field. Force the selected mux mode Input path no matter of MUX_MODE functionality. 1 ENABLED — Force input path of pad GPIO1_IO03 0 DISABLED — Input Path is determined by functionality | | | | | | | | | | | | | | | |
| MUX_MODE | Mux Mode Select Field. Select 1 of 9 iomux modes to be used for pad: GPIO1_IO03. 0000 ALT0 — Select mux mode: ALT0 mux port: I2C1_SDA of instance: i2c1 0001 ALT1 — Select mux mode: ALT1 mux port: GPT1_COMPARE3 of instance: gpt1 0010 ALT2 — Select mux mode: ALT2 mux port: USB_OTG2_OC of instance: usb 0100 ALT4 — Select mux mode: ALT4 mux port: USDHC1_CD_B of instance: usdhc1 0101 ALT5 — Select mux mode: ALT5 mux port: GPIO1_IO03 of instance: gpio1 0110 ALT6 — Select mux mode: ALT6 mux port: CCM_DIO_EXT_CLK of instance: ccm 0111 ALT7 — Select mux mode: ALT7 mux port: SRC_TESTER_ACK of instance: src NOTE: ALT7 mode will be automatically active when system reset. The PAD setting will be 100 K pull down and input enable during reset period. Once system reset is completed, the state of GPIO1_IO03 will be output keeper and input enable. 1000 ALT8 — Select mux mode: ALT8 mux port: UART1_RX of instance: uart1 | | | | | | | | | | | | | | | |

③ 对于 GPIO1_IO05, 设置如下寄存器:

Address: 20E_0000h base + 70h offset = 20E_0070h

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Reserved | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reserved | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO05 field descriptions

| Field | Description |
|-----------|---|
| 31–5 - | This field is reserved. Reserved |
| 4 SION | Software Input On Field. Force the selected mux mode Input path no matter of MUX_MODE functionality. 1 ENABLED — Force input path of pad GPIO1_IO05 0 DISABLED — Input Path is determined by functionality |
| MUX_MODE | MUX Mode Select Field. Select 1 of 9 iomux modes to be used for pad: GPIO1_IO05. 0000 ALT0 — Select mux mode: ALT0 mux port: ENET2_REF_CLK2 of instance: enet2 0001 ALT1 — Select mux mode: ALT1 mux port: PWM4_OUT of instance: pwm4 0010 ALT2 — Select mux mode: ALT2 mux port: ANATOP_OTG2_ID of instance: anatop 0011 ALT3 — Select mux mode: ALT3 mux port: CSL_FIELD of instance: csi 0100 ALT4 — Select mux mode: ALT4 mux port: USDHCI1_VSELECT of instance: usdhc1 0101 ALT5 — Select mux mode: ALT5 mux port: GPIO1_IO05 of instance: gpio1 0110 ALT6 — Select mux mode: ALT6 mux port: ENET2_1588_EVENTO_OUT of instance: enet2 1000 ALT8 — Select mux mode: ALT8 mux port: UART5_RX of instance: uart5 |

④ 对于 GPIO1_IO06, 设置如下寄存器:

Address: 20E_0000h base + 74h offset = 20E_0074h

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Reserved | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reserved | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO06 field descriptions

| Field | Description |
|-----------|---|
| 31–5 - | This field is reserved. Reserved |
| 4 SION | Software Input On Field. Force the selected mux mode Input path no matter of MUX_MODE functionality. 1 ENABLED — Force input path of pad GPIO1_IO06 0 DISABLED — Input Path is determined by functionality |
| MUX_MODE | MUX Mode Select Field. Select 1 of 9 iomux modes to be used for pad: GPIO1_IO06. 0000 ALT0 — Select mux mode: ALT0 mux port: ENET1_MDIO of instance: enet1 0001 ALT1 — Select mux mode: ALT1 mux port: ENET2_MDIO of instance: enet2 0010 ALT2 — Select mux mode: ALT2 mux port: USB_OTG_PWR_WAKE of instance: usb 0011 ALT3 — Select mux mode: ALT3 mux port: CSI_MCLK of instance: csi 0100 ALT4 — Select mux mode: ALT4 mux port: USDHCI2_WP of instance: usdhc2 0101 ALT5 — Select mux mode: ALT5 mux port: GPIO1_IO06 of instance: gpio1 0110 ALT6 — Select mux mode: ALT6 mux port: CCM_WAIT of instance: ccm 0111 ALT7 — Select mux mode: ALT7 mux port: CCM_REF_EN_B of instance: ccm 1000 ALT8 — Select mux mode: ALT8 mux port: UART1_CTS_B of instance: uart1 |

步骤3：设置 GPIO5_IO03、GPIO1_IO03、GPIO1_IO05、GPIO1_IO06 为输出引脚，设置其输出电平

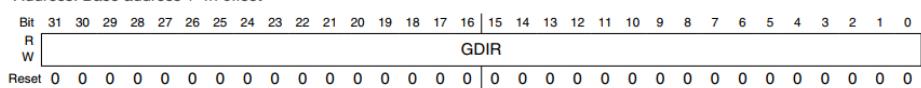
寄存器地址为：

GPIO memory map

| Absolute address (hex) | Register name | Width (in bits) | Access | Reset value | Section/ page |
|------------------------|--------------------------------------|-----------------|--------|-------------|-----------------------------|
| 209_C000 | GPIO data register (GPIO1_DR) | 32 | R/W | 0000_0000h | 28.5.1/1358 |
| 209_C004 | GPIO direction register (GPIO1_GDIR) | 32 | R/W | 0000_0000h | 28.5.2/1359 |
| 20A_C000 | GPIO data register (GPIO5_DR) | 32 | R/W | 0000_0000h | 28.5.1/1358 |
| 20A_C004 | GPIO direction register (GPIO5_GDIR) | 32 | R/W | 0000_0000h | 28.5.2/1359 |

设置方向寄存器，把引脚设置为输出引脚：

Address: Base address + 4h offset

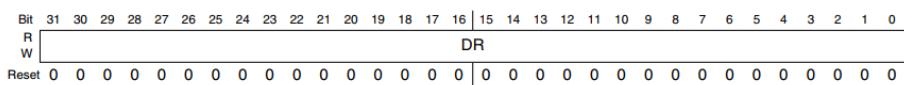


GPIOx_GDIR field descriptions

| Field | Description |
|-------|---|
| GDIR | <p>GPIO direction bits. Bit n of this register defines the direction of the GPIO[n] signal.</p> <p>NOTE: GPIO_GDIR affects only the direction of the I/O signal when the corresponding bit in the I/O MUX is configured for GPIO.</p> <p>0 INPUT — GPIO is configured as input. 1 OUTPUT — GPIO is configured as output.</p> |

设置数据寄存器，设置引脚的输出电平：

Address: Base address + 0h offset



GPIOx_DR field descriptions

| Field | Description |
|-------|--|
| DR | <p>Data bits. This register defines the value of the GPIO output when the signal is configured as an output (GDIR[n]=1). Writes to this register are stored in a register. Reading GPIO_DR returns the value stored in the register if the signal is configured as an output (GDIR[n]=1), or the input signal's value if configured as an input (GDIR[n]=0).</p> <p>NOTE: The I/O multiplexer must be configured to GPIO mode for the GPIO_DR value to connect with the signal. Reading the data register with the input path disabled always returns a zero value.</p> |

7.4.3 写程序

涉及的寄存器挺多，一个一个去执行 ioremap 效率太低。

先定义结构体，然后对结构体指针进行 ioremap。

对于 IOMUXC，可以如下定义：

```
struct iomux {  
    volatile unsigned int unnames[23];  
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO00;  
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO01;  
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO02;  
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03;  
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO04;  
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO05;  
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO06;  
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO07;  
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO08;  
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO09;  
};  
  
struct iomux *iomux = ioremap(0x20e0000, sizeof(struct iomux));
```

对于 GPIO，可以如下定义：

```
struct imx6ull_gpio {  
    volatile unsigned int dr;  
    volatile unsigned int gdir;  
    volatile unsigned int psr;  
    volatile unsigned int icr1;  
    volatile unsigned int icr2;  
    volatile unsigned int imr;  
    volatile unsigned int isr;  
    volatile unsigned int edge_sel;  
};  
  
struct imx6ull_gpio *gpio1 = ioremap(0x209C000, sizeof(struct imx6ull_gpio));  
struct imx6ull_gpio *gpio5 = ioremap(0x20AC000, sizeof(struct imx6ull_gpio));
```

7.4.4 上机实验

先启动 IMX6ULL QEMU 模拟器，挂载 NFS 文件系统。

运行 QEMU 时，
QEMU 内部为主机虚拟出一个网卡，IP 为 10.0.2.2，
IMX6ULL 有一个网卡，IP 为 10.0.2.15，
它连接到主机的虚拟网卡。
这样 IMX6ULL 就可以通过 10.0.2.2 去访问 Ubuntu 了。

然后执行以下命令安装驱动、执行测试程序：

```
insmod 100ask_led.ko
./ledtest /dev/100ask_led0 on
./ledtest /dev/100ask_led0 off
```

7.4.5 课后作业

- 在驱动里有 ioremap，什么时候执行 iounmap？请完善程序
- 驱动程序中有太多的 if 判断，请优化程序减少 if 的使用

8. 驱动设计的思想：面向对象/分层/分离

8.1 面向对象

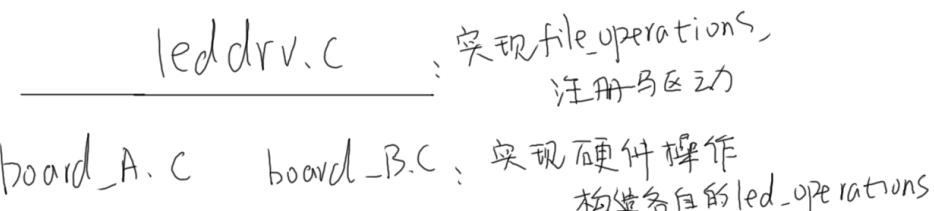
字符设备驱动程序抽象出一个 file_operations 结构体；

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。以假设举例，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。

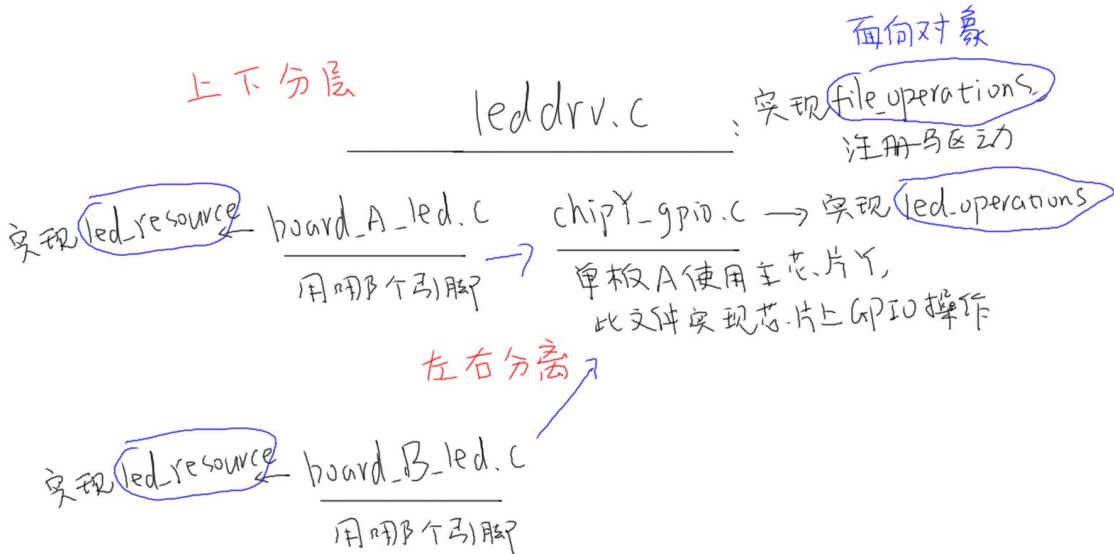
设置输出电平：修改 GPIO5 的 DR 寄存器的 bit4。

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

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。

9. 驱动进化之路：总线设备驱动模型

示例：

```

static struct resource resources[] = {
    {
        .start = WIFI_HOST_WAKE,
        .flags = IORESOURCE_IRQ,
        .name = IRQ_RES_NAME,
    },
};

static struct platform_device ssv_wifi_device = {
    .name = "ssv_wlan",
    .id = 1,
    .num_resources = ARRAY_SIZE(resources),
    .resource = resources,
    .dev = {
        .platform_data = &ssv_wifi_control,
        .release = ssv_wifi_device_release,
    },
};

```

↑ 资源 ↓ 匹配

```

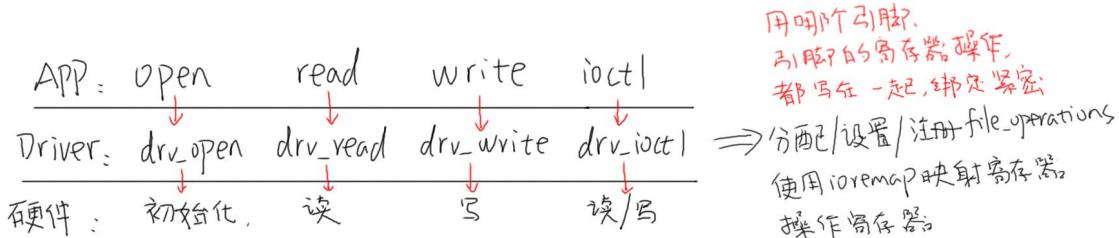
static struct platform_driver wifi_driver = {
    .probe = wifi_probe,
    .remove = wifi_remove,
    .suspend = wifi_suspend,
    .resume = wifi_resume,
    .driver = {
        .name = "ssv_wlan",
    }
};

```

9.1 驱动编写的 3 种方法

以 LED 驱动为例：

① 传统写法

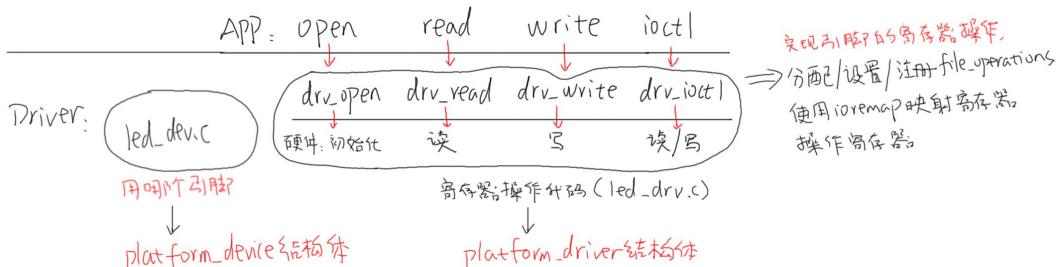


使用哪个引脚，怎么操作引脚，都写死在代码中。

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。

② 总线设备驱动模型



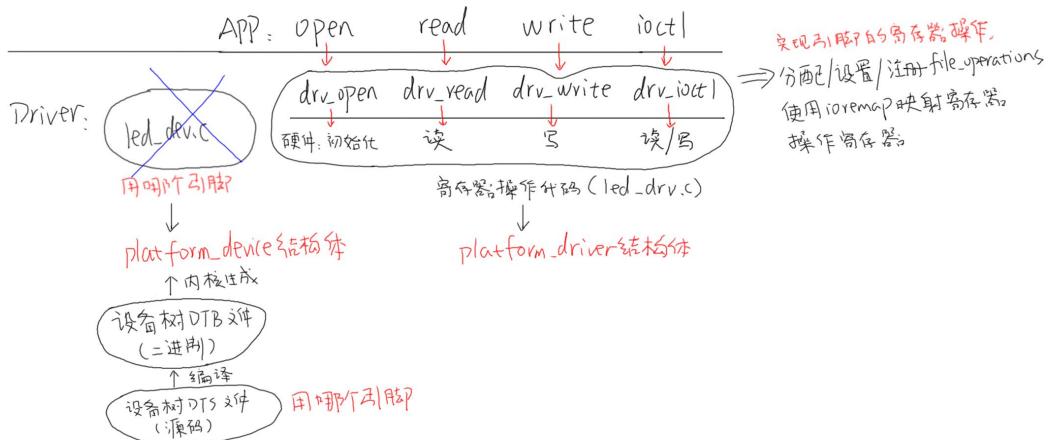
引入 platform_device/platform_driver，将“资源”与“驱动”分离开来。

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

③ 设备树

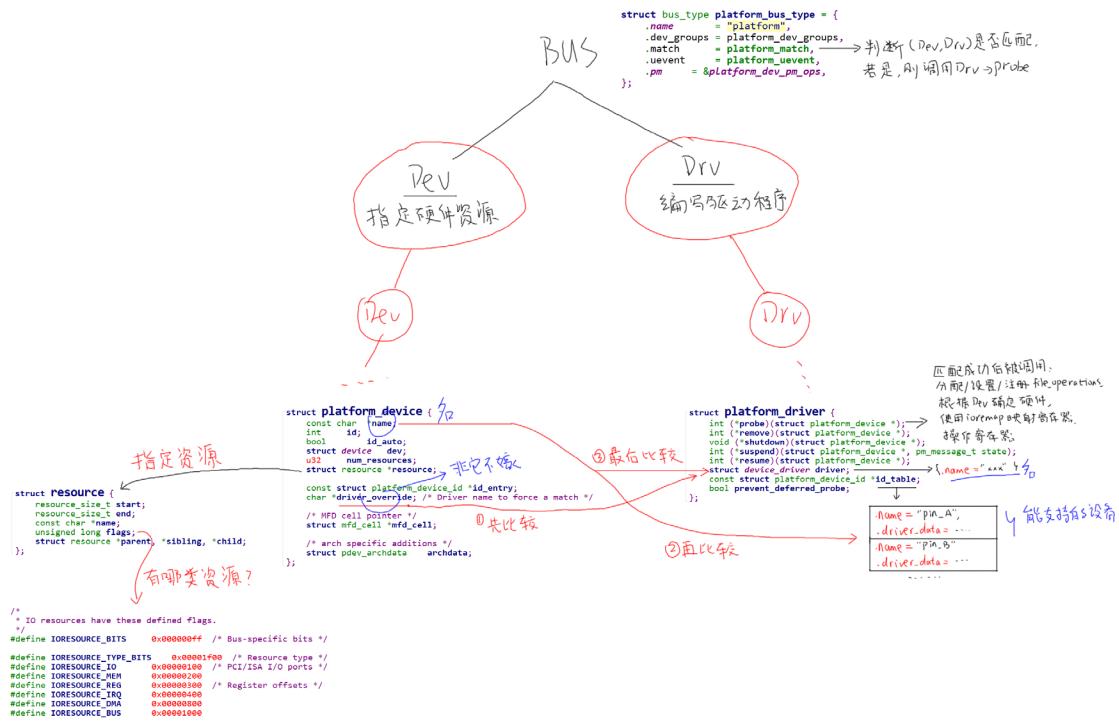


通过配置文件——设备树来定义“资源”。

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

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。

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。

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。

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
```

9.5 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。

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
```

② 获得资源

返回该 dev 中某类型(type)资源中的第几个(num):

```
struct resource *platform_get_resource(struct platform_device *dev,  
                                       unsigned int type, unsigned int num)
```

返回该 dev 所用的第几个(num)中断:

```
int platform_get_irq(struct platform_device *dev, unsigned int num)
```

通过名字(name)返回该 dev 的某类型(type)资源:

```
struct resource *platform_get_resource_byname(struct platform_device *dev,  
                                              unsigned int type,  
                                              const char *name)
```

通过名字(name)返回该 dev 的中断号:

```
int platform_get_irq_byname(struct platform_device *dev, const char *name)
```

9.6 怎么写程序

- 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。
- 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。

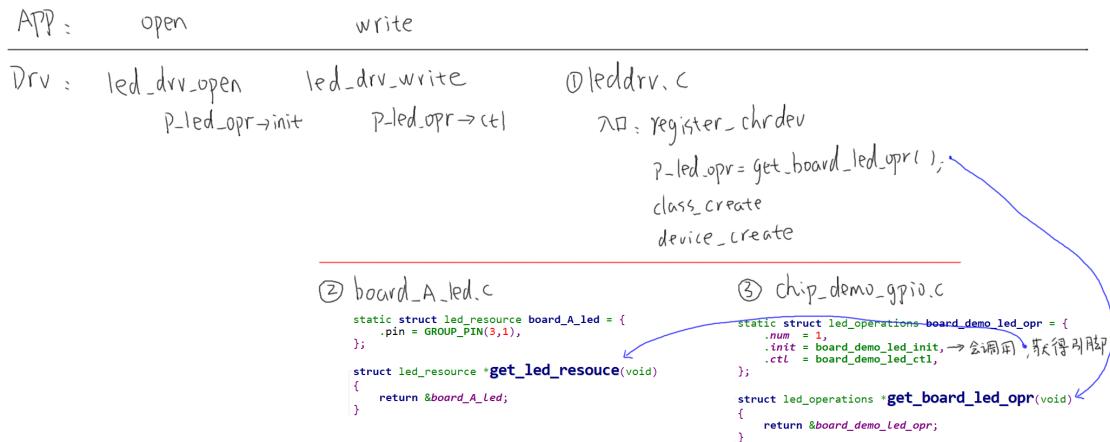
9.7 课后作业

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:

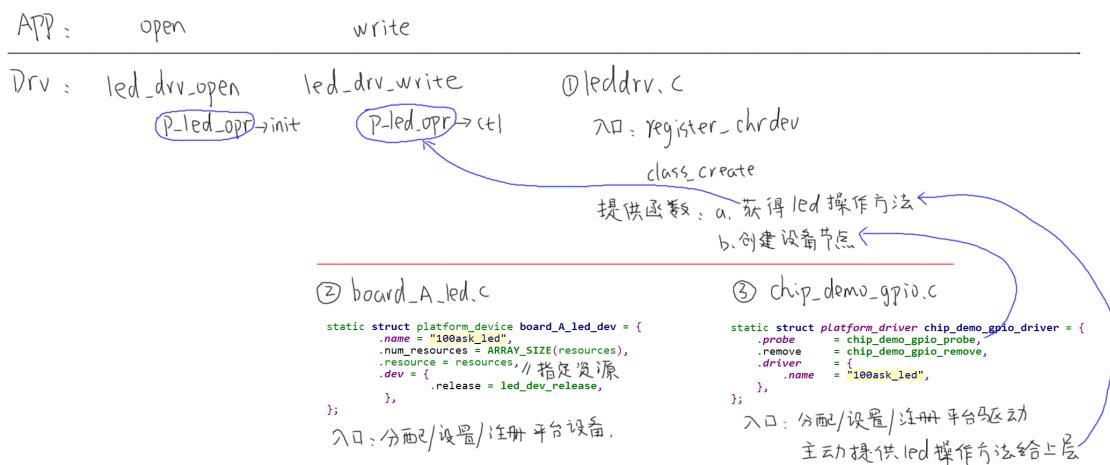
- ① 确定它的名字
- ② 根据它的名字找到对应的 platform_driver
- ③ 进入 platform_device_register/platform_driver_register 内部, 分析 dev 和 drv 的匹配过程

10. 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



11.3 写代码

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：

```
static struct platform_device board_A_led_dev = {
    .name = "100ask_led",
    .num_resources = ARRAY_SIZE(resources),
    .resource = resources,
    .dev = {
        .release = led_dev_release,
    },
};
```

则在调用 platform_device_unregister 时会出现警告，如下图所示：

```
[root@roc-rk3399-pc:/mnt]# insmod board_A_led.ko
[root@roc-rk3399-pc:/mnt]# rmmod board_A_led.ko
[24125.280055] Device '100ask_led.0' does not have a release() function, it is broken and must be fixed.
[24125.295027] -----[ cut here ]-----
[24125.305130] WARNING: at drivers/base/core.c:251
```

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(func_a);
```

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：

```
[root@roc-rk3399-pc:/mnt]# insmod chip_demo_gpio.ko
[24299.917448] chip_demo_gpio: Unknown symbol register_led_operations (err 0)
[24299.935714] chip_demo_gpio: Unknown symbol led_class_destroy_device (err 0)
[24299.950843] chip_demo_gpio: Unknown symbol led_class_create_device (err 0)
[24299.971482] chip_demo_gpio: Unknown symbol register_led_operations (err 0)
[24299.982958] chip_demo_gpio: Unknown symbol led_class_destroy_device (err 0)
[24299.994834] chip_demo_gpio: Unknown symbol led_class_create_device (err 0)
insmod: can't insert 'chip_demo_gpio.ko': unknown symbol in module, or unknown parameter
```

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。

11. 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
```

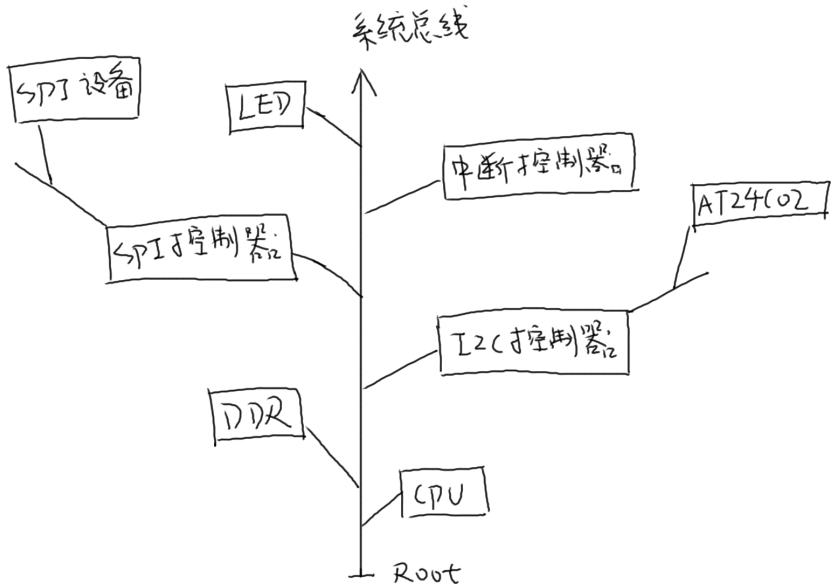
/sys/firmware/devicetree 目录下是以目录结构呈现的 dtb 文件，根节点对应 base 目录，每一个节点对应一个目录，每一个属性对应一个文件。

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”？

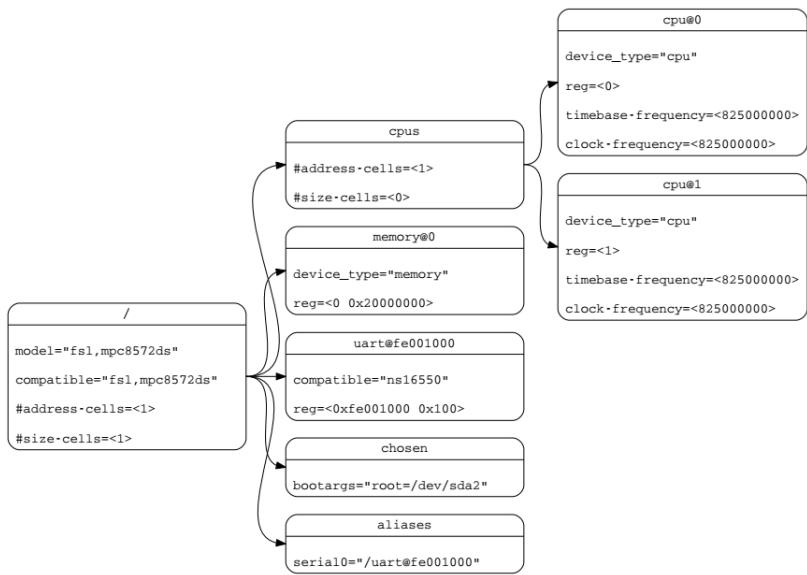


怎么描述这棵树？

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：

```
/dts-v1/;

{
    model="fsl,mpc8572ds"
    compatible="fsl,mpc8572ds"
    #address-cells=<1>
    #size-cells=<1>

    cpus {
        #address-cells=<1>
        #size-cells=<0>
        cpu@0 {
            device_type="cpu"
            reg=<0>
            timebase-frequency=<825000000>
            clock-frequency=<825000000>
        };

        cpu@1 {
            device_type="cpu"
            reg=<1>
            timebase-frequency=<825000000>
            clock-frequency=<825000000>
        };
    };

    memory@0 {
        device_type="memory"
        reg=<0 0x20000000>
    };

    uart@fe001000 {
        compatible="ns16550"
        reg=<0xfe001000 0x100>
    };

    chosen {
        bootargs="root=/dev/sda2";
    };

    aliases {
        serial0="/uart@fe001000"
    };
};
```

11.2.1 Devicetree 格式

① DTS 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]
};
```

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";  
};
```

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", "hs8250";  
example = <0xf00f0000 19>, "a strange property format";
```

11.2.2 dts 文件包含 dtsci 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.dtsci。“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.dtsci 是大部分相同, 小部分不同, 所以需要引脚 xxxx.dtsci 并修改。

dtsci 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 dtsci 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.dtsci"  
  
/{  
....  
};
```

11.2.3 常用的属性

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>;
    };
}
```

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。

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。

③ 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";
};
```

Table 2.4: Values for status property

| Value | Description |
|------------|--|
| "okay" | Indicates the device is operational. 设备正常运行 |
| "disabled" | Indicates that the device is not presently operational, but it might become operational in the future (for example, something is not plugged in, or switched off). 设备不可操作，但是后面可以恢复工作 Refer to the device binding for details on what disabled means for a given device. |
| "fail" | Indicates that the device is not operational. A serious error was detected in the device, and it is unlikely to become operational without repair. 发生了严重错误，需修复 |
| "fail-sss" | Indicates that the device is not operational. A serious error was detected in the device and it is unlikely to become operational without repair. The sss portion of the value is specific to the device and indicates the error condition detected. 发生了严重错误，需修复；sss表示错误信息 |

④ 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>;
    };
}
```

⑤ 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。

⑥ device_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)

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
           // ulimage : 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
```

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 {
        .....
    }
};
```

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>;
```

};

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__DT__ -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 "/include"。

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
```

① 对于 100ask-am335x 单板

设备树文件是：内核源码目录中 arch/arm/boot/dts/100ask-am335x.dtb

要更换板子上的设备树文件，启动板子后，更换这个文件：/boot/mx6ull-14x14-ebf.dtb

② 对于 firefly-rk3288

设备树文件是：内核源码目录中 arch/arm/boot/dts/rk3288-firefly.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/rk3288-firefly.dtb

③ 对于 firefly 的 roc-rk3399-pc

设备树文件是：内核源码目录中 arch/arm64/boot/dts/rk3399-roc-pc.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/rk3399-roc-pc.dtb

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。

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

⑥ 对于正点原子 imx6ull-alpha

设备树文件是：内核源码目录中 arch/arm/boot/dts/imx6ull-14x14-alpha.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/arch/arm/boot/dts/imx6ull-14x14-alpha.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呈现的 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(-l 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 -l 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

```

struct device_node {
    const char *name;
    const char *type;
    phandle phandle;
    const char *full_name;
    struct fwnode_handle fwnode;

    struct property *properties;
    struct property *deadprops; /* removed properties */
    struct device_node *parent;
    struct device_node *child;
    struct device_node *sibling;
    struct kobject kobj;
    unsigned long _flags;
    void *data;
#define defined(CONFIG_SPARC)
    const char *path_component_name;
    unsigned int unique_id;
    struct of_irq_controller *irq_trans;
#endif
} « end device_node » ;

```

```

struct property {
    char *name;
    int length;
    void *value;
    struct property *next;
    unsigned long _flags;
    unsigned int unique_id;
    struct bin_attribute attr;
};

```

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 compatible 属性的子节点

B. 含有特定 compatible 属性的子节点

如果一个节点的 compatible 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 compatible 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 {  
        compatible = "mytest", "simple-bus";  
        mytest@0 {  
            compatible = "mytest_0";  
        };  
    };  
  
    i2c {  
        compatible = "samsung,i2c";  
        at24c02 {  
            compatible = "at24c02";  
        };  
    };  
  
    spi {  
        compatible = "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：

```

static int platform_match(struct device *dev, struct device_driver *drv)
{
    struct platform_device *pdev = to_platform_device(dev);
    struct platform_driver *pdrv = to_platform_driver(drv);

    /* When driver_override is set, only bind to the matching driver */
    ① if (pdev->driver_override)
        return !strcmp(pdev->driver_override, drv->name);

    /* Attempt an OF style match first */
    ② if (of_driver_match_device(dev, drv))
        return 1;

    /* Then try ACPI style match */
    if (acpi_driver_match_device(dev, drv))
        return 1;

    ③ /* Then try to match against the id table */
    if (pdrv->id_table)
        return platform_match_id(pdrv->id_table, pdev) != NULL;

    ④ /* fall-back to driver name match */
    return (strcmp(pdev->name, drv->name) == 0);
} « end platform_match »

```

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。

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：

```

struct device_node {
    const char *name; 来自节点的name属性
    const char *type; 来自节点的device_type属性
    phandle phandle;
    const char *full_name;
    struct fwnode_handle fwnode;

    struct property *properties; 含有compatible属性
}

```

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：

```
struct of_device_id {  
    char    name[32];  
    char    type[32];  
    char    compatible[128];  
    const void *data;  
};
```

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 compatible 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 device_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。

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。

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。

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

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。

设备树的处理过程是： dtb -> device_node -> platform_device

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)
```

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
```

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。

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。

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，它属性 IORESOURCE_MEM 类型的资源；

对于设备树节点中的 interrupts 属性，它属性 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 include/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：

```
struct device_node {
    const char *name;
    const char *type;
    phandle phandle;
    const char *full_name;
    struct fwnode_handle fwnode;

    struct property *properties;
    struct property *deadprops; /* removed properties */
    struct device_node *parent;
    struct device_node *child;
    struct device_node *sibling;
    struct kobject kobj;
    unsigned long _flags;
    void *data;
#ifndef CONFIG_SPARC
    const char *path_component_name;
    unsigned int unique_id;
    struct of_irq_controller *irq_trans;
#endif
} __end_device_node__;
```

```
struct property {
    char *name;
    int length;
    void *value;
    struct property *next;
    unsigned long _flags;
    unsigned int unique_id;
    struct bin_attribute attr;
};
```

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。

② 找到属性

内核源码 include/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。

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?

① 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

② 参考同类型单板的设备树文件

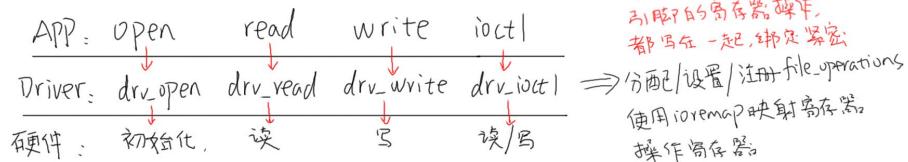
③ 网上搜索

④ 实在没办法时, 只能去研究驱动源码

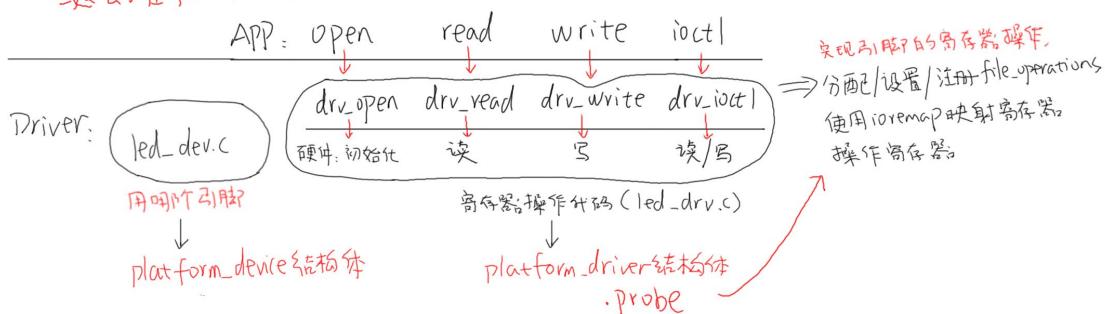
12. LED 模板驱动程序的改造：设备树

12.1 总结 3 种写驱动程序的方法

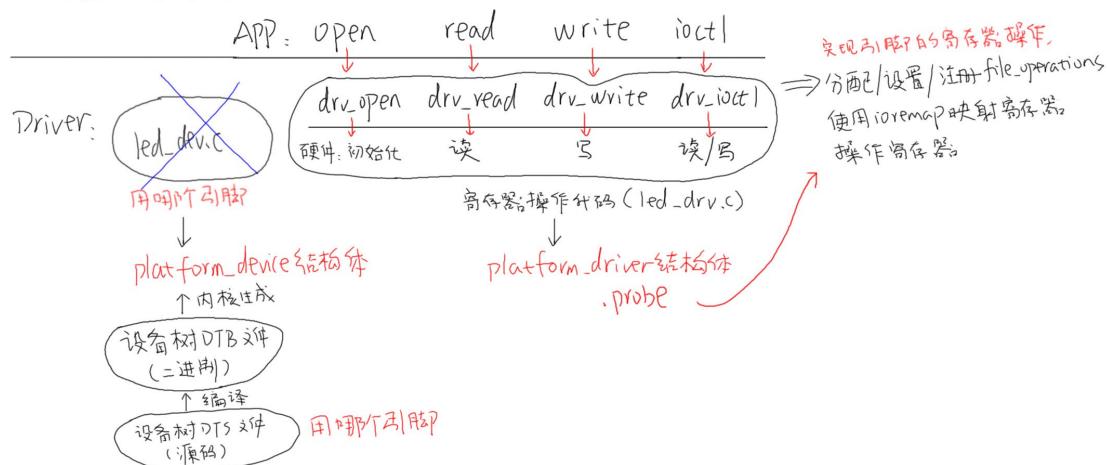
① 资源和驱动在同一个文件里



② 资源用 platform_device 指定
驱动在 platform_driver 实现



③ 资源用设备树指定
驱动在 platform_driver 实现



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：

```

DT$          驱动

xxx {
    compatible = "brcm,bcm2835-aux-uart"; ←

};

static const struct of_device_id bcm2835aux_serial_match[] = {
    { .compatible = "brcm,bcm2835-aux-uart" },
    {}
};

MODULE_DEVICE_TABLE(of, bcm2835aux_serial_match);

static struct platform_driver bcm2835aux_serial_driver = {
    .driver = {
        .name = "bcm2835-aux-uart",
        .of_match_table = bcm2835aux_serial_match,
    },
    .probe  = bcm2835aux_serial_probe,
    .remove = bcm2835aux_serial_remove,
};

```

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)>;
};
```

① 对于 100ask-am335x 单板

设备树文件是：内核源码目录中 arch/arm/boot/dts/100ask-am335x.dts

修改、编译后得到 arch/arm/boot/dts/100ask-am335x.dtb 文件。

要更换板子上的设备树文件，启动板子后，更换这个文件：/boot/mx6ull-14x14-ebf.dtb

② 对于 firefly-rk3288

设备树文件是：内核源码目录中 arch/arm/boot/dts/rk3288-firefly.dts

修改、编译后得到 arch/arm/boot/dts/rk3288-firefly.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/rk3288-firefly.dtb

③ 对于 firefly 的 roc-rk3399-pc

设备树文件是：内核源码目录中 arch/arm64/boot/dts/rk3399-roc-pc.dts
修改、编译后得到 arch/arm64/boot/dts/rk3399-roc-pc.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/rk3399-roc-pc.dtb

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.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。

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.dts
修改、编译后得到 arch/arm/boot/dts/imx6ull-14x14-ebf.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/imx6ull-14x14-ebf.dtb

⑥ 对于正点原子 imx6ull-alpha

设备树文件是：内核源码目录中 arch/arm/boot/dts/imx6ull-14x14-alpha.dts
修改、编译后得到 arch/arm/boot/dts/imx6ull-14x14-alpha.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/arch/arm/boot/dts/imx6ull-14x14-alpha.dtb

12.3.2 修改 platform_driver 的源码

基于：04_led_drv_template_bus_dev_drv,
改好后的源码为：05_led_drv_template_device_tree

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：

- 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。

- 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
```

- 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：

```
[root@roc-rk3399-pc:/sys/bus/platform/drivers/rockchip-i2s]# ls  
bind      → ff890000.i2s  ff8a0000.i2s → uevent      unbind
```

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。

13. APP 怎么读取按键值

APP 读取按键值，需要有按键驱动程序。

为什么要讲按键驱动程序？

APP 去读按键的方法有 4 种：

- ① 查询方式
- ② 休眠-唤醒方式
- ③ poll 方式
- ④ 异步通知方式

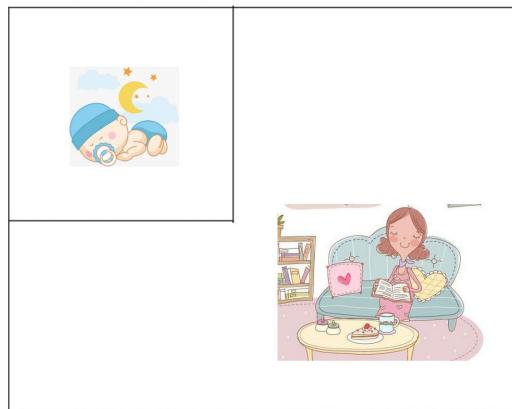
通过这 4 种方式的学习，我们可以掌握如下知识：

- ① 驱动的基本技能：中断、休眠、唤醒、poll 等机制。

这些基本技能是驱动开发的基础，其他大型驱动复杂的地方是它的框架及设计思想，但是基本技术就这些。

- ② APP 开发的基本技能：阻塞、非阻塞、休眠、poll、异步通知。

13.1 妈妈怎么知道孩子醒了



妈妈怎么知道卧室里小孩醒了？

- ① 时时进房间看一下：**查询方式**
简单，但是累
- ② 进去房间陪小孩一起睡觉，小孩醒了会吵醒她：**休眠-唤醒**
不累，但是妈妈干不了活了
- ③ 妈妈要干很多活，但是可以陪小孩睡一会，定个闹钟：**poll 方式**
要浪费点时间，但是可以继续干活。
妈妈要么是被小孩吵醒，要么是被闹钟吵醒。
- ④ 妈妈在客厅干活，小孩醒了他会自己走出房门告诉妈妈：**异步通知**
妈妈、小孩互不耽误。

这 4 种方法没有优劣之分，在不同的场合使用不同的方法。

13.2 APP 读取按键的 4 种方法

跟上述生活场景类似，APP 去读取按键也有 4 种方法：

- ① 查询方式
- ② 休眠-唤醒方式
- ③ poll 方式
- ④ 异步通知方式

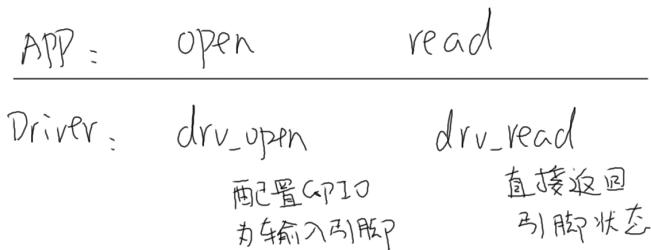
第 2、3、4 种方法，都涉及中断服务程序。中断，就像小孩醒了会哭闹一样，中断不经意间到来，它会做某些事情：唤醒 APP、向 APP 发信号。

所以，在按键驱动程序中，中断是核心。

实际上，中断无论是在单片机还是在 Linux 中都很重要。在 Linux 中，中断的知识还涉及进程、线程等。

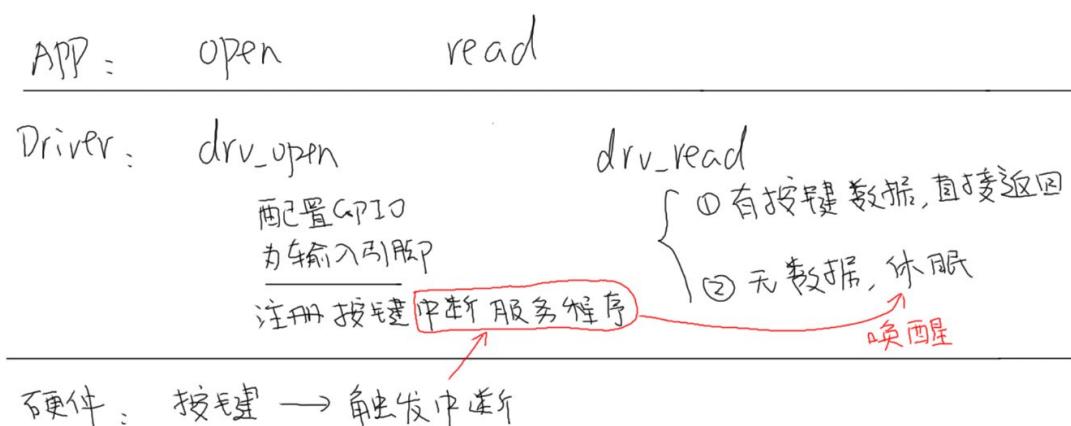
13.2.1 查询方式

这种方法最简单：



驱动程序中构造、注册一个 file_operations 结构体，里面提供有对应的 open,read 函数。
APP 调用 open 时，导致驱动中对应的 open 函数被调用，在里面配置 GPIO 为输入引脚。
APP 调用 read 时，导致驱动中对应的 read 函数被调用，它读取寄存器，把引脚状态直接返回给 APP。

13.2.2 休眠-唤醒方式



驱动程序中构造、注册一个 file_operations 结构体，里面提供有对应的 open,read 函数。

APP 调用 open 时，导致驱动中对应的 open 函数被调用，在里面配置 GPIO 为输入引脚；并且注册 GPIO 的中断处理函数。

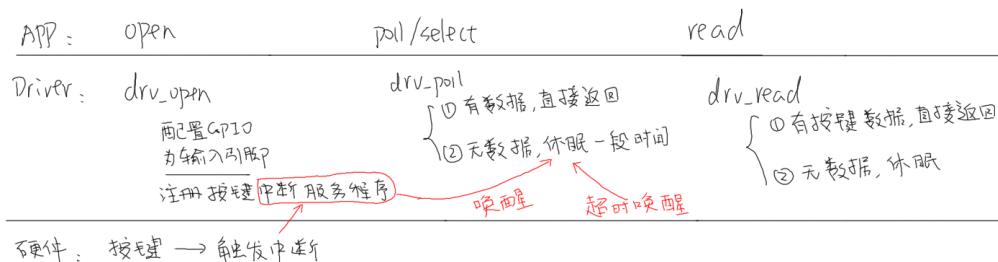
APP 调用 read 时，导致驱动中对应的 read 函数被调用，如果有按键数据则直接返回给 APP；否则 APP 在内核态休眠。

当用户按下按键时，GPIO 中断被触发，导致驱动程序之前注册的中断服务程序被执行。它会记录按键数据，并唤醒休眠中的 APP。

APP 被唤醒后继续在内核态运行，即继续执行驱动代码，把按键数据返回给 APP(的用户空间)。

13.2.3 poll 方式

上面的休眠-唤醒方式有个缺点：如果用户一直没操作按键，那么 APP 就会永远休眠。我们可以给 APP 定个闹钟，这就是 poll 方式。



驱动程序中构造、注册一个 file_operations 结构体，里面提供有对应的 open,read,poll 函数。

APP 调用 open 时，导致驱动中对应的 open 函数被调用，在里面配置 GPIO 为输入引脚；并且注册 GPIO 的中断处理函数。

APP 调用 poll 或 select 函数，意图是“查询”是否有数据，这 2 个函数都可以指定一个超时时间，即在这段时间内没有数据的话就返回错误。这会导致驱动中对应的 poll 函数被调用，如果有按键数据则直接返回给 APP；否则 APP 在内核态休眠一段时间。

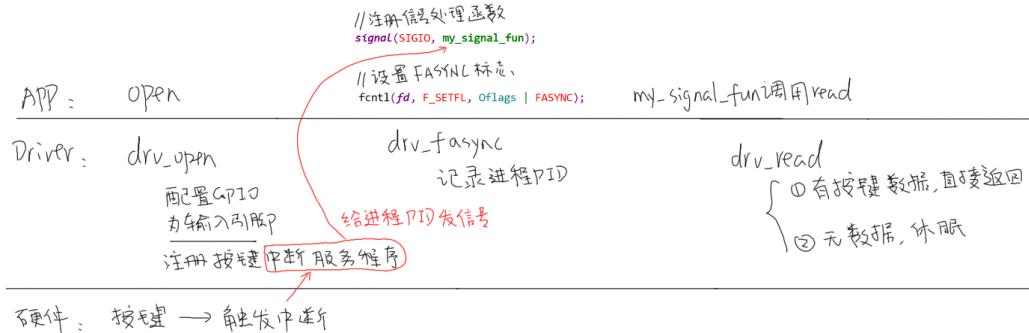
当用户按下按键时，GPIO 中断被触发，导致驱动程序之前注册的中断服务程序被执行。它会记录按键数据，并唤醒休眠中的 APP。

如果用户没按下按键，但是超时时间到了，内核也会唤醒 APP。

所以 APP 被唤醒有 2 种原因：用户操作了按键，超时。被唤醒的 APP 在内核态继续运行，即继续执行驱动代码，把“状态”返回给 APP(的用户空间)。

APP 得到 poll/select 函数的返回结果后，如果确认是有数据的，则再调用 read 函数，这会导致驱动中的 read 函数被调用，这时驱动程序中含有数据，会直接返回数据。

13.2.4 异步通知方式



异步通知的实现原理是：内核给 APP 发信号。信号有很多种，这里发的是 SIGIO。

驱动程序中构造、注册一个 file_operations 结构体，里面提供有对应的 open,read,fasync 函数。

APP 调用 open 时，导致驱动中对应的 open 函数被调用，在里面配置 GPIO 为输入引脚；并且注册 GPIO 的中断处理函数。

APP 给信号 SIGIO 注册自己的处理函数：my_signal_fun。

APP 调用 fcntl 函数，把驱动程序的 flag 改为 FASYNC，这会导致驱动程序的 fasync 函数被调用，它只是简单记录进程 PID。

当用户按下按键时，GPIO 中断被触发，导致驱动程序之前注册的中断服务程序被执行。它会记录按键数据，然后给进程 PID 发送 SIGIO 信号。

APP 收到信号后会被打断，先执行信号处理函数：在信号处理函数中可以去调用 read 函数读取按键值。

信号处理函数返回后，APP 会继续执行下来的代码。

13.2.5 驱动程序提供能力，不提供策略

我们的驱动程序可以实现上述 4 种提供按键的方法，但是驱动程序不应该限制 APP 使用哪种方法。

这就是驱动设计的一个原理：提供能力，不提供策略。就是说，你想用哪种方法都行，驱动程序都可以提供；但是驱动程序不能限制你使用哪种方法。

14. 查询方式的按键驱动程序_编写框架

14.1 LED 驱动回顾

对于 LED, APP 调用 open 函数导致驱动程序的 led_open 函数被调用。在里面, 把 GPIO 配置为输出引脚。安装驱动程序后并不意味着会使用对应的硬件, 而 APP 要使用对应的硬件, 必须先调用 open 函数。所以建议在驱动程序的 open 函数中去设置引脚。

APP 继续调用 write 函数传入数值, 在驱动程序的 led_write 函数根据该数值去设置 GPIO 的数据寄存器, 从而控制 GPIO 的输出电平。

怎么操作寄存器? 从芯片手册得到对应寄存器的物理地址, 在驱动程序中使用 ioremap 函数映射得到虚拟地址。驱动程序中使用虚拟地址去访问寄存器。

用户: ledtest /dev/led0/1/2 on/off

APP: open("/dev/led0/1/2", ...), write(fd, val, 1)

Driver: led_open:

根据子设备号确定哪个 LED
使能 GPIO 模块
配置引脚为 output

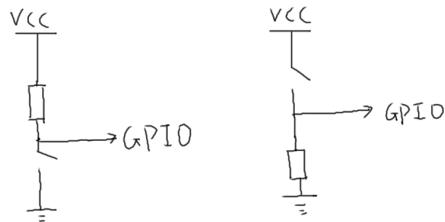
led_write

根据子设备号确定哪个 LED
根据 val 设置引脚的电平

硬件:

14.2 按键驱动编写思路

GPIO 按键的原理图一般有如下 2 种:



按键没被按下时, 上图中左边的 GPIO 电平为高, 右边的 GPIO 电平为低。

按键被按下后, 上图中左边的 GPIO 电平为低, 右边的 GPIO 电平为高。

编写按键驱动程序最简单的方法如下图所示:

用户: button_test /dev/button

APP: open("/dev/button", ...), read

Driver: button_open

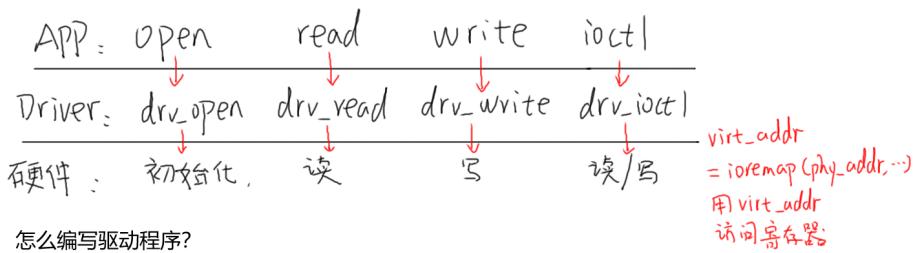
根据子设备号确定是哪个按键
使能引脚
配置引脚为输入功能

button_read

根据子设备号确定是哪个按键
读取寄存器
返回引脚电平

硬件:

回顾一下编写驱动程序的套路：



- ① 确定主设备号，也可以让内核分配
- ② 定义自己的 `file_operations` 结构体 它是核心
- ③ 实现对应的 `drv_open`/`drv_read`/`drv_write` 等函数，填入 `file_operations` 结构体
- ④ 把 `file_operations` 结构体告诉内核：`register_chrdev`
- ⑤ 谁来注册驱动程序啊？得有一个入口函数：安装驱动程序时，就会去调用这个入口函数
- ⑥ 有入口函数就应该有出口函数：卸载驱动程序时，出口函数调用 `unregister_chrdev`
- ⑦ 其他完善：提供设备信息，自动创建设备节点：`class_create`, `device_create`

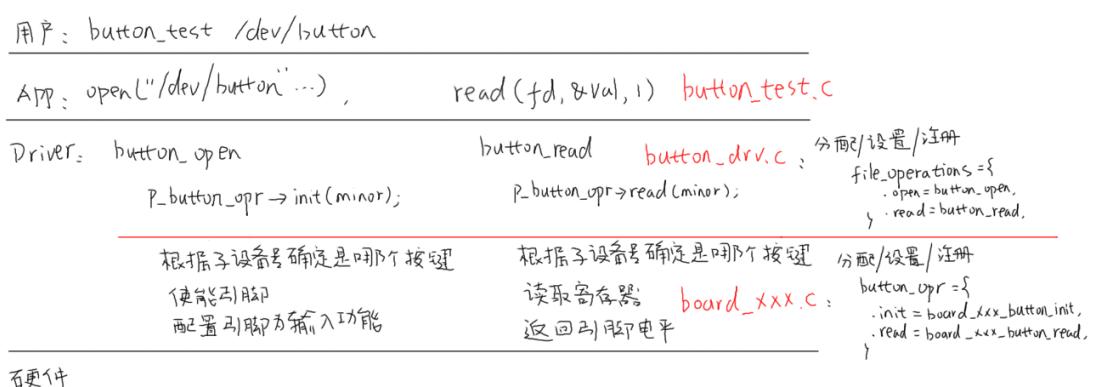
对于使用查询方式的按键驱动程序，我们只需要实现 `button_open`、`button_read`。

14.3 编程：先写框架

我们的目的写出一个容易扩展到各种芯片、各种板子的按键驱动程序，所以驱动程序分为上下两层：

- ① `button_drv.c` 分配/设置/注册 `file_operations` 结构体
 起承上启下的作用，向上提供 `button_open`, `button_read` 供 APP 调用。
 而这 2 个函数又会调用底层硬件提供的 `p_button_opr` 中的 `init`、`read` 函数操作硬件。
- ② `board_xxx.c` 分配/设置/注册 `button_operations` 结构体
 这个结构体是我们自己抽象出来的，里面定义单板 `xxx` 的按键操作函数。

这样的结构易于扩展，对于不同的单板，只需要替换 `board_xxx.c` 提供自己的 `button_operations` 结构体即可。



14.4 测试

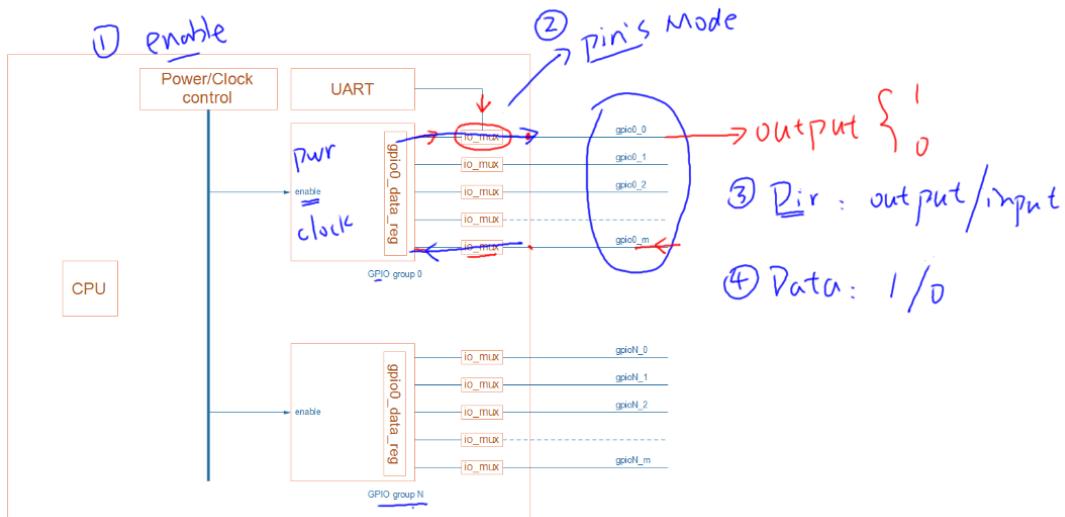
14.5 课后怎业

合并 LED、BUTTON 框架驱动程序：01_led_drv_template、01_button_drv_template，合
并为：gpio_drv_template

15. 具体单板的按键驱动程序(查询方式)

15.0 GPIO 操作回顾

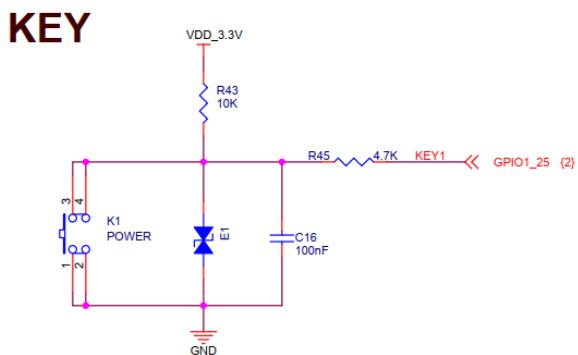
参考第 4 章《普适的 GPIO 引脚操作方法》、第 5 章《具体单板的 GPIO 操作方法》。



15.1 AM335X 的按键驱动程序(查询方式)

15.1.1 先看原理图确定引脚及操作方法

AM335X 是底板+核心板的结构，打开底板原理图 100ask_am335x_v12_原理图.pdf，它有 4 个按键，本视频只操作一个按键，原理图如下：



平时按键电平为高，按下按键后电平为低。

按键引脚为 GPIO1_25。

15.1.2 再看芯片手册确定寄存器及操作方法

| 模块 Module | 模块内部子模块 sub module | 基地址 base addr | 内部寄存器 register | 寄存器偏移地址 offset | 备注 comment |
|--|-----------------------|------------------|--|--------------------------------------|---|
| PRCM | PRM_PER | | | | |
| | CM_PER | 0x44E00000 | CM_PER_GPIO1_CLKCTRL CM_PER_GPIO2_CLKCTRL CM_PER_GPIO3_CLKCTRL | Ach B0h B4h | 使能GPIO1的时钟 使能GPIO2的时钟 使能GPIO3的时钟 |
| Control Module | | 0x44E10000 | conf_gpmc_ad0 conf_gpmc_ad1 | 800h 804h | 设置引脚gpmc_ad0的模式 设置引脚gpmc_ad1的模式 |
| GPIO0 | 0x44E07000 | | | | |
| GPIO1 | 0x4804C000 | | | | |
| GPIO2 | 0x481AC000 | | | | |
| GPIO3 | 0x481AE000 | | | | |
| | | | GPIO_OE GPIO_DATAIN GPIO_DATAOUT GPIO_CLEARDATAOUT GPIO_SETDATAOUT | 134h 138h 13Ch 190h 194h | 设置引脚用于输出 读取引脚电平 设置输出引脚的电平 set-and-clear 设置输出引脚的电平 |
| PRCM: Power, Reset, and Clock Management (电源、复位、时钟管理器) CM: Control Module(控制模块) 或 Clock Module (时钟模块) PRM_PER: Power Reset Module Peripheral Registers (电源/复位模块中关于外设的寄存器) CM_PER: Clock Module Peripheral Registers (时钟模块中关于外设的寄存器) | | | | | |
| AM335X | | | | | |

步骤 1：使能 GPIO1 模块

设置 CM_PER_GPIO1_CLKCTRL 寄存器的 bit[18]为 1, bit[1:0]为 0x2, 该寄存器地址为 0x44E00000+0xAC。

Table 8-59. CM_PER_GPIO1_CLKCTRL Register Field Descriptions

| Bit | Field | Type | Reset | Description |
|-------|--------------------------|------|-------|---|
| 31-19 | Reserved | R | 0h | |
| 18 | OPTFCLKEN_GPIO_1_G_DBCLK | R/W | 0h | Optional functional clock control. 0x0 = FCLK_DIS : Optional functional clock is disabled 0x1 = FCLK_EN : Optional functional clock is enabled |
| 17-16 | IDLEST | R | 3h | Module idle status. 0x0 = Func : Module is fully functional, including OCP 0x1 = Trans : Module is performing transition: wakeup, or sleep, or sleep abortion 0x2 = Idle : Module is in idle mode (only OCP part). It is functional if using separate functional clock 0x3 = Disable : Module is disabled and cannot be accessed |
| 15-2 | Reserved | R | 0h | |
| 1-0 | MODULEMODE | R/W | 0h | Control the way mandatory clocks are managed. 0x0 = DISABLED : Module is disable by SW. Any OCP access to module results in an error, except if resulting from a module wakeup (asynchronous wakeup). 0x1 = RESERVED_1 : Reserved 0x2 = ENABLE : Module is explicitly enabled. Interface clock (if not used for functions) may be gated according to the clock domain state. Functional clocks are guaranteed to stay present. As long as in this configuration, power domain sleep transition cannot happen. 0x3 = RESERVED : Reserved |

步骤 2：把 GPIO1_25 对应的引脚设置为 GPIO 模式

要用哪一个寄存器来把 GPIO1_25 对应的引脚设置为 GPIO 模式？

- ① 在核心板原理图 ET-som335X 原理图.pdf 里搜“GPIO1_25”，可以看到下图，确定 pin number 为 U16：



- ② 在芯片手册 AM335x Sitara™ Processors.pdf 里搜“U16”，可得下图，引脚名为 GPMC_A9，用作 GPIO 时要设置为 mode 7：

| ZCE BALL NUMBER [1] | ZCZ BALL NUMBER [1] | PIN NAME [2] | SIGNAL NAME [3] | MODE [4] |
|---------------------|---------------------|--------------|---|--------------------------------------|
| NA | V16 | GPMC_A8 | gpmc_a8 gmii2_rxrd3 rgmii2_rd3 mmc2_dat6 gpmc_a24 pr1_mii1_rxrd0 mcasp0_aclkx gpio1_24 | 0 1 2 3 4 5 6 7 |
| pin number | | pin name | | |
| NA | U16 | GPMC_A9 (10) | gpmc_a9 gmii2_rxrd2 rgmii2_rd2 mmc2_dat7 / rmi2_crs_dv gpmc_a25 pr1_mii_mr1_clk mcasp0_fsx mode 7 gpio1_25 | 0 1 2 3 4 5 6 7 |

- ③ 在芯片手册 AM335x_datasheet_spruh73p.pdf 中搜 gpmc_a9，

| 864h conf_gpmc_a9 | | | | | | | Section 9.3.1.50 |
|-------------------|------------------------------|------------------------------|-------------------------------|---------------------------|---------------------------|---|------------------|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Reserved | conf_<module>_<pin>-slewctrl | conf_<module>_<pin>-rxactive | conf_<module>_<pin>-putypesel | conf_<module>_<pin>-puden | conf_<module>_<pin>-mmode | | |
| R-0h | R/W-0h | R/W-1h | R/W-0h | R/W-0h | | | R/W-0h |

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

Table 9-60. conf_<module>_<pin> Register Field Descriptions

| Bit | Field | Type | Reset | Description | |
|-------|-------------------------------|------|-------|--|---------------------|
| 31-20 | Reserved | R | 0h | | |
| 19-7 | Reserved | R | 0h | | |
| 6 | conf_<module>_<pin>-slewctrl | R/W | X | Select between faster or slower slew rate 0: Fast 1: Slow Reset value is pad-dependent. | |
| 5 | conf_<module>_<pin>-rxactive | R/W | 1h | Input enable value for the PAD 0: Receiver disabled 1: Receiver enabled | 跟output不一样 这位要设置 |
| 4 | conf_<module>_<pin>-putypesel | R/W | X | Pad pullup/pulldown type selection 0: Pulldown selected 1: Pullup selected Reset value is pad-dependent. | |
| 3 | conf_<module>_<pin>-puden | R/W | X | Pad pullup/pulldown enable 0: Pullup/pulldown enabled 1: Pullup/pulldown disabled Reset value is pad-dependent. | |
| 2-0 | conf_<module>_<pin>-mmode | R/W | X | Pad functional signal mux select. Reset value is pad-dependent. | |

所以，要把 GPIO1_25 对应的引脚设置为 GPIO 模式，要设置 conf_gpmc_a9 寄存器的 bit[5]为 1， bit[2:0]为 7，这个寄存器的地址是 0x44E10000+0x864。

步骤3：设置 GPIO1 内部寄存器，把 GPIO1_25 设置为输入引脚，读数据寄存器

GPIO_OE 寄存器：地址为 0x4804C000+0x134, bit[25]设置为 1。

GPIO_DATAIN 寄存器：地址为 0x4804C000+0x138, 读其 bit[25]。

Table 25-21. GPIO_OE Register Field Descriptions

| Bit | Field | Type | Reset | Description |
|------|-------------|------|-------------|---|
| 31-0 | OUTPUTEN[n] | R/W | FFFFFFFFFFh | Output Data Enable 0h = The corresponding GPIO port is configured as an output. 1h = The corresponding GPIO port is configured as an input. |

Table 25-22. GPIO_DATAIN Register Field Descriptions

| Bit | Field | Type | Reset | Description |
|------|--------|------|-------|--------------------|
| 31-0 | DATAIN | R | 0h | Sampled Input Data |

15.1.3 编程

| | | | | |
|-----------------------------|--|--|--|--|
| 用户： button_test /dev/button | APP： open(" /dev/button" ...), read(fd, &val, 1) button_test.c | Driver： button_open p_button_opr → init(minor); | button_read p_button_opr → read(minor); | button_drv.c : 分配/设置/注册 file_operations = { .open = button_open, .read = button_read, }; |
| 根据子设备号确定是哪个按钮 | 根据子设备号确定是哪个按钮 | 使能IO引脚 | 读取寄存器： board_am335x.c | 返回引脚电平 |
| 配置引脚为输入功能 | | | | 使能GPIO 配置引脚为GPIO 设置引脚为输入 读Data寄存器 |

石更牛

15.1.4 测试

安装驱动程序之后执行测试程序，观察它的返回值(执行测试程序的同时操作按键)：

```
# insmod button_drv.ko
# insmod board_am335x.ko
# ./button_test /dev/100ask_button0
```

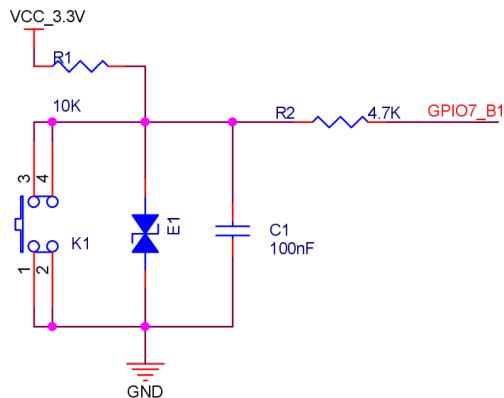
15.1.5 课后作业

- ① 修改 board_am335x.c，增加更多按键
- ② 修改 button_test.c，使用按键来点灯

15.2 RK3288 的按键驱动程序(查询方式)

15.2.1 先看原理图确定引脚及操作方法

Firefly 的 RK3288 开发板上没有按键，我们为它制作的扩展板上有 1 个按键。在扩展板原理图 rk3288_extend_v12_0715.pdf 中可以看到按键，如下：



平时按键电平为高，按下按键后电平为低。

按键引脚为 GPIO7_B1。

15.2.2 再看芯片手册确定寄存器及操作方法

芯片手册为 Rockchip_RK3288_TRM_V1.2_Part1-20170321.pdf, 不过我们总结如下。

| 模块 Module | 基地址 base addr | 内部寄存器 register | 寄存器偏移地址 offset | 备注 comment |
|--------------|------------------|---|--|--|
| CRU | 0xFF760000 | CRU_CLKGATE14_CON CRU_CLKGATE17_CON | 0x198 0xA4 | 使能GPIO1~8的时钟 使能GPIO0的时钟 |
| PMU | 0xFF730000 | 1.使能GPIO7 | | |
| | | PMU_GPIO0_A_IOMUX PMU_GPIO0_B_IOMUX PMU_GPIO0_C_IOMUX | 0x0084 0x0088 0x008C | GPIO0A iomux sel GPIO0B iomux sel GPIO0C iomux sel |
| GRF | 0xFF770000 | GRF_GPIO1D_IOMUX GRF_GPIO2A_IOMUX GRF_GPIO2B_IOMUX GRF_GPIO2C_IOMUX GRF_GPIO3A_IOMUX GRF_GPIO3B_IOMUX GRF_GPIO3C_IOMUX GRF_GPIO3DL_IOMUX GRF_GPIO3DH_IOMUX GRF_GPIO4AL_IOMUX GRF_GPIO4AH_IOMUX GRF_GPIO4BL_IOMUX GRF_GPIO4C_IOMUX GRF_GPIO4D_IOMUX GRF_GPIO5B_IOMUX GRF_GPIO5C_IOMUX GRF_GPIO6A_IOMUX GRF_GPIO6B_IOMUX GRF_GPIO6C_IOMUX GRF_GPIO7A_IOMUX GRF_GPIO7B_IOMUX | 0x000c 0x0010 0x0014 0x0018 0x0020 0x0024 0x0028 0x002c 0x0030 0x0034 0x0038 0x003c 0x0044 0x0048 0x0050 0x0054 0x005c 0x0060 0x0064 0x006c 0x0070 0x0074 0x0078 0x0080 0x0084 | GPIO1D iomux control GPIO2A iomux control GPIO2B iomux control GPIO2C iomux control GPIO3A iomux control GPIO3B iomux control GPIO3C iomux control GPIO3D iomux control GPIO4D iomux control GPIO4A iomux control GPIO4B iomux control GPIO4C iomux control GPIO4D iomux control GPIO5B iomux control GPIO5C iomux control GPIO6A iomux control GPIO6B iomux control GPIO6C iomux control GPIO7A iomux control GPIO7B iomux control GPIO7CL iomux control GPIO7CH iomux control GPIO8A iomux control GPIO8B iomux control |
| GPIO0 | 0xFF750000 | 2.把GPIO7_B1设置为GPIO模式 | | |
| GPIO1 | 0xFF780000 | | | |
| GPIO2 | 0xFF790000 | | | |
| GPIO3 | 0xFF7A0000 | | | |
| GPIO4 | 0xFF7B0000 | | | |
| GPIO5 | 0xFF7C0000 | | | |
| GPIO6 | 0xFF7D0000 | | | |
| GPIO7 | 0xFF7E0000 | 3.设置GPIO7_B1为输入引脚读取引脚电平 | | |
| GPIO8 | 0xFF7F0000 | | | |
| | | GPIO_SWPORTA_DR GPIO_SWPORTA_DDR GPIO_EXT_PORTA | 0x0000 0x0004 0x0050 | Port A data register Port A data direction register Port A clear interrupt register |
| | | RK3288 | | |

步骤 1：使能 GPIO7 模块

设置 CRU_CLKGATE14_CON 寄存器的 bit[7]为 0。

要设置 bit7，必须同时设置 bit23 为 1。

该寄存器地址为 0xFF760000+0x198。

| CRU_CLKGATE14_CON | | | |
|---|------|-------------|--|
| Address: Operational Base + offset (0x0198) | | | |
| Internal clock gating control register14 | | | |
| Bit | Attr | Reset Value | Description |
| 31:16 | WO | 0x0000 | write_mask write mask. bit23也要设为1, 才能设置bit7 When every bit HIGH, enable the writing corresponding bit When every bit LOW, don't care the writing corresponding bit |
| 15:13 | RO | 0x0 | reserved |
| 12 | RW | 0x0 | pclk_alive_niu_gate_en ALIVE_NIU pclk disable When HIGH, disable clock |
| 11 | RW | 0x0 | pclk_grf_gate_en GRF pclk disable When HIGH, disable clock |
| 10:9 | RO | 0x0 | reserved |
| 8 | RW | 0x0 | pclk_gpio8_gate_en GPIO8 pclk disable When HIGH, disable clock |
| 7 | RW | 0x0 | pclk_gpio7_gate_en GPIO7 pclk disable When HIGH, disable clock 设为0 |
| 6 | RW | 0x0 | pclk_gpio6_gate_en GPIO6 pclk disable When HIGH, disable clock |
| 5 | RW | 0x0 | pclk_gpio5_gate_en GPIO5 pclk disable When HIGH, disable clock |
| 4 | RW | 0x0 | pclk_gpio4_gate_en GPIO4 pclk disable When HIGH, disable clock |
| 3 | RW | 0x0 | pclk_gpio3_gate_en GPIO3 pclk disable When HIGH, disable clock |
| 2 | RW | 0x0 | pclk_gpio2_gate_en GPIO2 pclk disable When HIGH, disable clock |

步骤 2：把 GPIO7_B1 对应的引脚设置为 GPIO 模式

设置 GRF_GPIO7B_IOMUX 寄存器的 bit[3:2]为 0b00。

要设置 bit[3:2]，必须同时设置 bit[19:18]为 0b11。

该寄存器地址为 0xFF770000+0x0070。

| GRF_GPIO7B_IOMUX | | | |
|---|------|-------------|--|
| Address: Operational Base + offset (0x0070) | | | |
| GPIO7B iomux control | | | |
| Bit | Attr | Reset Value | Description |
| 31:16 | RW | 0x0000 | write_enable bit~15 write enable When every bit HIGH, enable the writing corresponding bit When every bit LOW, don't care the writing corresponding bit |
| 设置b[19:18]为0b11 才能修改b[3:2] | | | |
| 15:14 | RW | 0x0 | gpio7b7_sel GPIO7B[7] iomux select 2'b00: gpio 2'b01: isp_shuttertrig 2'b10: spi1_txd 2'b11: reserved |
| 13:12 | RW | 0x0 | gpio7b6_sel GPIO7B[6] iomux select 2'b00: gpio 2'b01: isp_prelighttrig 2'b10: spi1_rxn 2'b11: reserved |
| 11:10 | RW | 0x0 | gpio7b5_sel GPIO7B[5] iomux select 2'b00: gpio 2'b01: isp_flashtrigout 2'b10: spi1_csn0 2'b11: reserved |
| 9:8 | RW | 0x0 | gpio7b4_sel GPIO7B[4] iomux select 2'b00: gpio 2'b01: isp_shutteren 2'b10: spi1_clk 2'b11: reserved |
| 7:6 | RW | 0x0 | gpio7b3_sel GPIO7B[3] iomux select 2'b00: gpio 2'b01: usb_drvvbus1 2'b10: edp_hotplug 2'b11: reserved |
| 5:4 | RW | 0x0 | gpio7b2_sel GPIO7B[2] iomux select 2'b00: gpio 2'b01: uart3gps_rtsn 2'b10: usb_drvvbus0 2'b11: reserved |
| 3:2 | RW | 0x0 | gpio7b1_sel GPIO7B[1] iomux select 2'b00: gpio 2'b01: uart3gps_ctsn 2'b10: gps_rfclk 2'b11: gpst1_clk 设为0b00 |

步骤3：设置 GPIO7 内部寄存器，把 GPIO7_B1 设置为输入引脚，读数据寄存器

GPIO_SWPORTA_DDR 方向寄存器：地址为 0xFF7E0000+ 0x0004, bit[9]设置为 0。

GPIO_EXT_PORTA 外部端口寄存器：地址为 0xFF7E0000+ 0x0050, 读其 bit[9]。

注意：

GPIO_A0~A7 对应 bit0~bit7; GPIO_B0~B7 对应 bit8~bit15;

GPIO_C0~C7 对应 bit16~bit23; GPIO_D0~D7 对应 bit24~bit31

GPIO_SWPORTA_DDR

Address: Operational Base + offset (0x0004)

Port A data direction register

| Bit | Attr | Reset Value | Description |
|------|------|-------------|---|
| 31:0 | RW | 0x00000000 | gpio_swporta_ddr Values written to this register independently control the direction of the corresponding data bit in Port A. 1'b0: Input (default) 1'b1: Output |

GPIO_EXT_PORTA

Address: Operational Base + offset (0x0050)

Port A external port register

| Bit | Attr | Reset Value | Description |
|------|------|-------------|--|
| 31:0 | RO | 0x00000000 | gpio_ext_porta When Port A is configured as Input, then reading this location reads the values on the signal. When the data direction of Port A is set as Output, reading this location reads the data register for Port A. |

15.2.3 编程

| | |
|---|---|
| 用户： button_test /dev/button | APP： open("/dev/button") , read(fd, &val, 1) button_test.c |
| Driver： button_open p_button_opr->init(minor); | button_read p_button_opr->read(minor); button_drv.c : 分配/设置/注册 file_operations = { .open = button.open, .read = button.read, }; |
| 根据子设备号确定是哪个按钮 使能引脚 配置引脚为输入功能 | 读取寄存器： board_rk3288.c 返回引脚电平 |
| 硬件 | 使能GPIO7 配置引脚为输入 设置引脚为输出 读写时读寄存器 |

15.2.4 测试

安装驱动程序之后执行测试程序，观察它的返回值(执行测试程序的同时操作按键)：

```
# insmod button_drv.ko
# insmod board_rk3288.ko
# ./button_test /dev/100ask_button0
```

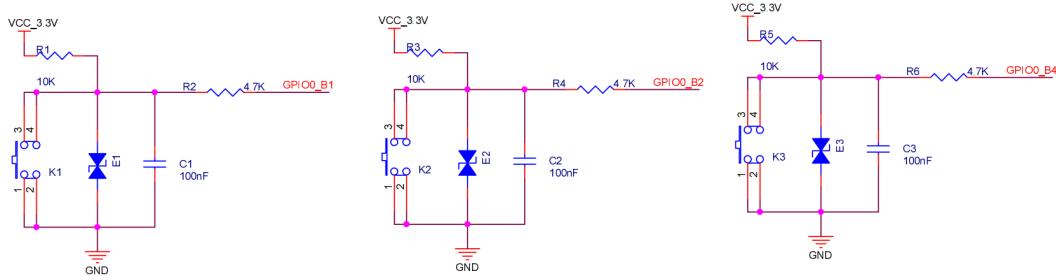
15.2.5 课后作业

- ① 修改 button_test.c, 使用按键来点灯

15.3 RK3399 的按键驱动程序(查询方式)

15.3.1 先看原理图确定引脚及操作方法

Firefly 的 RK3399 开发板上没有按键，我们为它制作的扩展板上有 3 个按键。在扩展板原理图 rk3399_extend_v12_0709final.pdf 中可以看到按键，如下：



平时按键电平为高，按下按键后电平为低。

按键引脚为 GPIO0_B1、GPIO0_B2、GPIO0_B4。

本视频中，只操作一个按键：GPIO0_B1。

15.3.2 再看芯片手册确定寄存器及操作方法

芯片手册为 Rockchip RK3399TRM V1.3 Part1.pdf 和 Rockchip RK3399TRM V1.3 Part2.pdf, 不过我们总结如下。

| 模块 | 地址 | 内部寄存器 | 寄存器偏移地址 | 备注 |
|---------------|------------|--|--|--|
| Module | base addr | register | offset | comment |
| CRU | 0xFF760000 | PMUCRU_CLKGATE_CON1 CRU_CLKGATE_CON31 | 0x0104 0x037c | 使能GPIO0~1的时钟 使能GPIO2~4的时钟 |
| PMU | 0xFF310000 | PMUGRF_GPIO0A_IOMUX PMUGRF_GPIO0B_IOMUX PMUGRF_GPIO1A_IOMUX PMUGRF_GPIO1B_IOMUX PMUGRF_GPIO1C_IOMUX | 0x00000 0x00004 0x00010 0x00014 0x00018 | GPIO0A iomux control GPIO0B iomux control GPIO1A iomux control GPIO1B iomux control GPIO1C iomux control |
| GRF | 0xFF770000 | GRF_GPIO2A_IOMUX GRF_GPIO2B_IOMUX GRF_GPIO2C_IOMUX GRF_GPIO2D_IOMUX GRF_GPIO3A_IOMUX GRF_GPIO3B_IOMUX GRF_GPIO3C_IOMUX GRF_GPIO3D_IOMUX GRF_GPIO4A_IOMUX GRF_GPIO4B_IOMUX GRF_GPIO4C_IOMUX GRF_GPIO4D_IOMUX | 0x0e000 0x0e004 0x0e008 0x0e00c 0x0e010 0x0e014 0x0e018 0x0e01c 0x0e020 0x0e024 0x0e028 0x0e02c | GPIO2A iomux control GPIO2B iomux control GPIO2C iomux control GPIO2D iomux control GPIO3A iomux control GPIO3B iomux control GPIO3C iomux control GPIO3D iomux control GPIO4A iomux control GPIO4B iomux control GPIO4C iomux control GPIO4D iomux control |
| GPIO0 | 0xFF720000 | 3.设置GPIO0_B1为输入引脚 读取其电平 | | |
| GPIO1 | 0xFF730000 | | | |
| GPIO2 | 0xFF780000 | | | |
| GPIO3 | 0xFF788000 | | | |
| GPIO4 | 0xFF790000 | | | |
| | | GPIO_SWPORTA_DR GPIO_SWPORTA_DDR GPIO_EXT_PORTA | 0x0000 0x0004 0x0050 | Port A data register Port A data direction register Port A clear interrupt register |
| RK3399 | | | | |

步骤 1：使能 GPIO0 模块

设置 PMUCRU_CLKGATE_CON1 寄存器的 bit[3]为 0。

要设置 bit3, 必须同时设置 bit19 为 1。

该寄存器地址为 0xFF760000+ 0x0104。

| BIT | Attr | Reset Value | Description |
|-------|------|-------------|--|
| 31:16 | WO | 0x0000 | pmu_rstn_b0 when every bit HIGH, enable the writing corresponding bit when every bit LOW, don't care the writing corresponding bit |
| 15 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit bit[19]设为1 才能修改bit[3] |
| 14 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock |
| 13 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock |
| 12 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock |
| 11 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock |
| 10 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock |
| 9 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock |
| 8 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock |
| 7 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock |
| 6 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock Suggest always on |
| 5 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock Suggest always on |
| 4 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock b[3]设置为0 |
| 3 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock 使能GPIO0 |
| 2 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock |
| 1 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock Suggest always on |
| 0 | RW | 0x0 | pck_wdt_md_pmu_clock_disable_bit when H20, disable clock |

步骤 2：把 GPIO0_B1 对应的引脚设置为 GPIO 模式

设置 PMUGRF_GPIO0B_IOMUX 寄存器的 bit[3:2]为 0b00。

要设置 bit[3:2]，必须同时设置 bit[19:18]为 0b11。

该寄存器地址为 0xFF310000+0x0004。

| PMUGRF_GPIO0B_IOMUX Address: Operational Base + offset (0x00004) GPIO0B Iomux control | | | |
|---|------|-------------|---|
| Bit | Attr | Reset Value | Description |
| 31:16 | RW | 0x0000 | b[19:18] 设为0b11 才能修改b[3:2] |
| 15:12 | RO | 0x0 | reserved |
| 11:10 | RW | 0x0 | gpio0b5_sel GPIO0B[5] iomux select 2'b00: gpio 2'b01: tcpd_vbusfdis 2'b10: tcpdusb2_vbussource3 2'b11: reserved |
| 9:8 | RW | 0x0 | gpio0b4_sel GPIO0B[4] iomux select 2'b00: gpio 2'b01: tcpd_vbusbdis 2'b10: reserved 2'b11: reserved |
| 7:6 | RW | 0x0 | gpio0b3_sel GPIO0B[3] iomux select 2'b00: gpio 2'b01: reserved 2'b10: reserved 2'b11: reserved |
| 5:4 | RW | 0x1 | gpio0b2_sel GPIO0B[2] iomux select 2'b00: gpio 2'b01: reserved 2'b10: reserved 2'b11: reserved |
| 3:2 | RW | 0x1 | gpio0b1_sel GPIO0B[1] iomux select 2'b00: gpio 2'b01: pmu1830_vsel 2'b10: reserved 2'b11: reserved 设为0b00 GPIO模式 |
| 1:0 | RW | 0x0 | gpio0b0_sel GPIO0B[0] iomux select 2'b00: gpio 2'b01: sdmmc_wrprt 2'b10: pmum0_wfi 2'b11: test_clkout2 |

步骤3：设置GPIO0 内部寄存器，把GPIO0_B1 设置为输入引脚，读数据寄存器

这些寄存器的介绍在芯片手册 Rockchip RK3399TRM V1.3 Part2.pdf 中。

GPIO_SWPORTA_DDR 方向寄存器：地址为 0xFF720000+ 0x0004, bit[9]设置为 0。

GPIO_EXT_PORTA 外部端口寄存器：地址为 0xFF720000+ 0x0050, 读其 bit[9]。

注意：

GPIO_A0~A7 对应 bit0~bit7; GPIO_B0~B7 对应 bit8~bit15;

GPIO_C0~C7 对应 bit16~bit23; GPIO_D0~D7 对应 bit24~bit31

GPIO_SWPORTA_DDR

Address: Operational Base + offset (0x0004)

Port A data direction register

| Bit | Attr | Reset Value | Description |
|------|------|-------------|---|
| 31:0 | RW | 0x00000000 | gpio_swporta_ddr Values written to this register independently control the direction of the corresponding data bit in Port A. 0: Input (default) 1: Output |

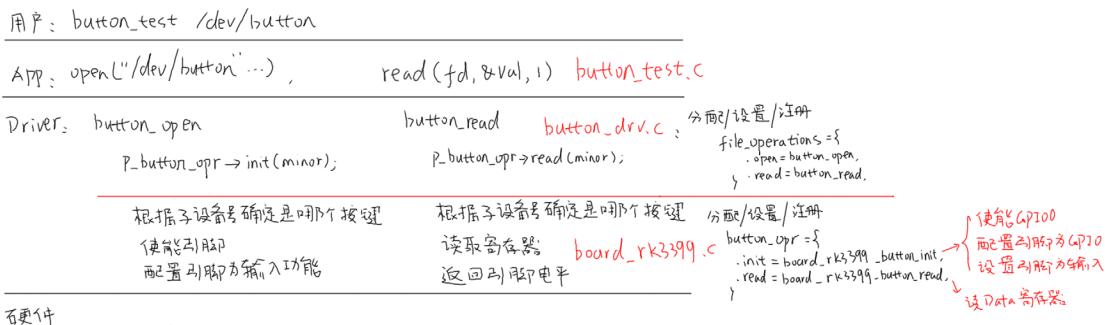
GPIO_EXT_PORTA

Address: Operational Base + offset (0x0050)

Port A external port register

| Bit | Attr | Reset Value | Description |
|------|------|-------------|--|
| 31:0 | RO | 0x00000000 | gpio_ext_porta When Port A is configured as Input, then reading this location reads the values on the signal. When the data direction of Port A is set as Output, reading this location reads the data register for Port A. |

15.3.3 编程



15.3.4 测试

安装驱动程序之后执行测试程序，观察它的返回值(执行测试程序的同时操作按键)：

```
# insmod button_drv.ko
# insmod board_rk3399.ko
# ./button_test /dev/100ask_button0
```

15.3.5 课后作业

① 修改 board_rk3399.c, 增加更多按键

② 修改 button_test.c, 使用按键来点灯

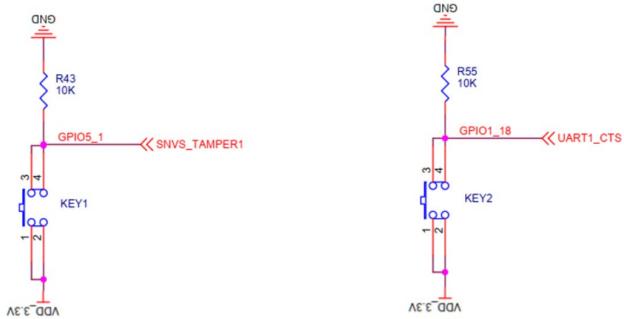
15.4 百问网 IMX6ULL-QEMU 的按键驱动程序(查询方式)

使用 QEMU 模拟的硬件，它的硬件资源可以随意扩展。

在 IMX6ULL QEMU 虚拟开发板上，我们为它设计了 2 个 按键。在 QEMU 的 GUI 上有 4 个按键，右边的 2 个留待以后用于电源管理。

15.4.1 先看原理图确定引脚及操作方法

KEY



平时按键电平为低，按下按键后电平为高。

按键引脚为 GPIO5_IO01、GPIO1_IO18。

15.4.2 再看芯片手册确定寄存器及操作方法

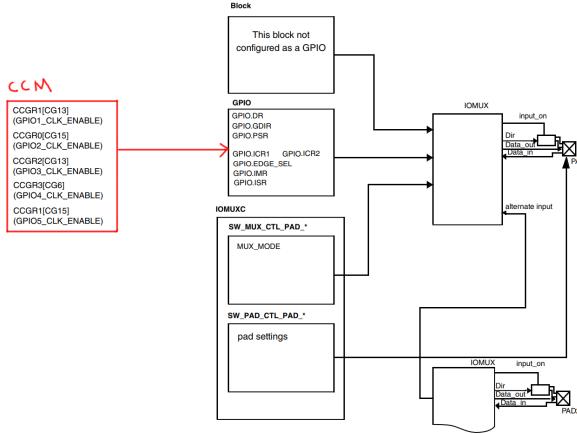


Figure 26-1. Chip IOMUX Scheme

步骤 1：使能 GPIO1、GPIO5

| Address: 20C_4000h base + 6Ch offset = 20C_406Ch | | | | | | | | | | | | | | | | |
|--|------|------|------|------|------|------|-----|-----|----|----|----|----|----|----|----|----|
| Bit | 31 | 30 | 29 | 28 | 27 | 28 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| R | CG15 | CG14 | CG13 | CG12 | CG11 | CG10 | CG9 | CG8 | | | | | | | | |
| W | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Reset | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| R | CG7 | CG6 | CG5 | CG4 | CG3 | CG2 | CG1 | CG0 | | | | | | | | |
| W | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Reset | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

CCM_CCGR1 field descriptions

| Field | Description |
|---------------|--|
| 31–30 CG15 | gpio5 clock (gpio5_clk_enable) |
| 29–28 CG14 | csu clock (csu_clk_enable) |
| 27–26 CG13 | gpio1 clock (gpio1_clk_enable) |
| 25–24 CG12 | uart4 clock (uart4_clk_enable) |
| 23–22 CG11 | gpt serial clock (gpt_serial_clk_enable) |
| 21–20 CG10 | gpt bus clock (gpt_clk_enable) |
| 19–18 CG9 | sim_s clock (sim_s_clk_enable) |
| 17–16 CG8 | adc1 clock (adc1_clk_enable) |
| 15–14 CG7 | epit2 clocks (epit2_clk_enable) |
| 13–12 CG6 | epit1 clocks (epit1_clk_enable) |
| 11–10 CG5 | uart3 clock (uart3_clk_enable) |
| 9–8 CG4 | adc2 clock (adc2_clk_enable) |
| 7–6 CG3 | ecspi4 clocks (ecspi4_clk_enable) |
| 5–4 CG2 | ecspi3 clocks (ecspi3_clk_enable) |
| 3–2 CG1 | ecspi2 clocks (ecspi2_clk_enable) |
| CG0 | ecspi1 clocks (ecspi1_clk_enable) |

设置 b[31:30]、b[27:26]就可以使能 GPIO5、GPIO1，设置为什么值呢？

看下图，设置为 0b11：

| CGR value | Clock Activity Description |
|-----------|---|
| 00 | Clock is off during all modes. Stop enter hardware handshake is disabled. |
| 01 | Clock is on in run mode, but off in WAIT and STOP modes |
| 10 | Not applicable (Reserved). |
| 11 | Clock is on during all modes, except STOP mode. |

- ① 00：该 GPIO 模块全程被关闭
- ② 01：该 GPIO 模块在 CPU run mode 情况下是使能的；在 WAIT 或 STOP 模式下，关闭
- ③ 10：保留
- ④ 11：该 GPIO 模块全程使能

步骤 2：设置 GPIO5_IO01、GPIO1_IO18 为 GPIO 模式

① 对于 GPIO5_IO01，设置如下寄存器：

Address: 229_0000h base + Ch offset = 229_000Ch

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Reserved | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reserved | | | | | | | | | | | | | | | | |
| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| W | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

IOMUXC_SNVS_SW_MUX_CTL_PAD_SNVS_TAMPER1 field descriptions

| Field | Description | | | | | | | | | | | | | | | |
|----------|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 31-5 | This field is reserved. Reserved | | | | | | | | | | | | | | | |
| - | | | | | | | | | | | | | | | | |
| 4 | SION Software Input On Field. Force the selected mux mode Input path no matter of MUX_MODE functionality. | | | | | | | | | | | | | | | |
| SION | 1 ENABLED — Force input path of pad SNVS_TAMPER1 0 DISABLED — Input Path is determined by functionality | | | | | | | | | | | | | | | |
| MUX_MODE | NOTE: ALT5 mode is only valid when TAMPER PIN is used as GPIO. This depends on FUSE setting "TAMPER_PIN_DISABLE[1:0]". Following is the mux information when TAMPER PIN is used as GPIO: SNVS_TAMPER1 ==> GPIO5_01 101 ALT5 — Select mux mode: ALT5 mux port, GPIO5_IO01 of instance - gpio5 Other Reserved | | | | | | | | | | | | | | | |

② 对于 GPIO1_IO18，设置如下寄存器：

Address: 20E_0000h base + 8Ch offset = 20E_008Ch

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Reserved | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reserved | | | | | | | | | | | | | | | | |
| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| W | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

IOMUXC_SW_MUX_CTL_PAD_UART1_CTS_B field descriptions

| Field | Description | | | | | | | | | | | | | | | |
|----------|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 31-5 | This field is reserved. Reserved | | | | | | | | | | | | | | | |
| - | | | | | | | | | | | | | | | | |
| 4 | SION Software Input On Field. Force the selected mux mode Input path no matter of MUX_MODE functionality. | | | | | | | | | | | | | | | |
| SION | 1 ENABLED — Force input path of pad UART1_CTS_B 0 DISABLED — Input Path is determined by functionality | | | | | | | | | | | | | | | |
| MUX_MODE | MUX Mode Select Field. Select 1 of 10 iomux modes to be used for pad: UART1_CTS_B. 0000 ALT0 — Select mux mode: ALT0 mux port: UART1_CTS_B of instance: uart1 0001 ALT1 — Select mux mode: ALT1 mux port: ENET1_RX_CLK of instance: enet1 0010 ALT2 — Select mux mode: ALT2 mux port: USDHC1_WP of instance: usdhc1 0011 ALT3 — Select mux mode: ALT3 mux port: CSI_DATA04 of instance: csi 0100 ALT4 — Select mux mode: ALT4 mux port: ENET2_1588_EVENT1_IN of instance: enet2 0101 ALT5 — Select mux mode: ALT5 mux port: GPIO1_IO18 of instance: gpio1 1000 ALT8 — Select mux mode: ALT8 mux port: USDHC2_WP of instance: usdhc2 1001 ALT9 — Select mux mode: ALT9 mux port: UART5_CTS_B of instance: uart5 | | | | | | | | | | | | | | | |

步骤3：设置 GPIO5_IO01、GPIO1_IO18 为输入引脚，读取引脚电平

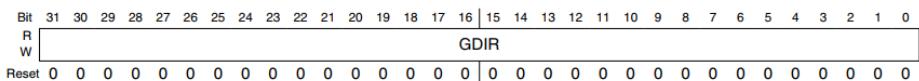
寄存器地址为：

GPIO memory map

| Absolute address (hex) | Register name | Width (in bits) | Access | Reset value | Section/page |
|------------------------|--------------------------------------|-----------------|--------|-------------|--------------|
| 209_C000 | GPIO data register (GPIO1_DR) | 32 | R/W | 0000_0000h | 28.5.1/1358 |
| 209_C004 | GPIO direction register (GPIO1_GDIR) | 32 | R/W | 0000_0000h | 28.5.2/1359 |
| 20A_C000 | GPIO data register (GPIO5_DR) | 32 | R/W | 0000_0000h | 28.5.1/1358 |
| 20A_C004 | GPIO direction register (GPIO5_GDIR) | 32 | R/W | 0000_0000h | 28.5.2/1359 |

设置方向寄存器，把引脚设置为输出引脚：

Address: Base address + 4h offset

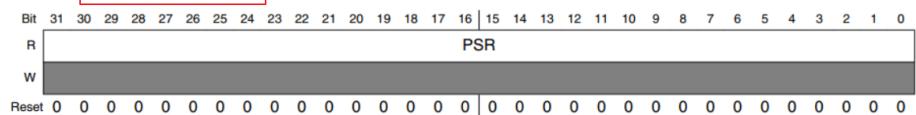


GPIOx_GDIR field descriptions

| Field | Description |
|-------|---|
| GDIR | <p>GPIO direction bits. Bit n of this register defines the direction of the GPIO[n] signal.</p> <p>NOTE: GPIO_GDIR affects only the direction of the I/O signal when the corresponding bit in the I/O MUX is configured for GPIO.</p> <p>0 INPUT — GPIO is configured as input. 1 OUTPUT — GPIO is configured as output.</p> |

读取引脚状态寄存器，得到引脚电平：

Address: Base address + 8h offset



GPIOx_PSR field descriptions

| Field | Description |
|-------|---|
| PSR | <p>GPIO pad status bits (status bits). Reading GPIO_PSR returns the state of the corresponding input signal.</p> <p>Settings:</p> <p>NOTE: The IOMUXC must be configured to GPIO mode for GPIO_PSR to reflect the state of the corresponding signal.</p> |

15.4.3 编程

```

用户: button_test /dev/button
App: open("/dev/button"),      read(fd, &val, 1) button_test.c
Driver: button_open           button_read      button_drv.c 分配/设置/注册
    p_button_opr->int(minor); p_button_opr->read(minor);
根据子设备号确定是哪个按钮
使能引脚
配置引脚为输入功能
根据子设备号确定是哪个按钮
读取寄存器: board_loask_
返回到脚电平 imxfull-qemu.c
    file_operations = {
        .open = button_open,
        .read = button_read,
    }
使能GPIO
而配置引脚为GPIO
设置引脚为输入
读取寄存器
    button_opr = {
        .int = board_loask,
        .read = board_imxfull_button_read,
    }
    button = board_imxfull_button;
}

```

有附件

涉及的寄存器挺多，一个一个去执行 ioremap 效率太低。

先定义结构体，然后对结构体指针进行 ioremap。

对于 IOMUXC，可以如下定义：

```

struct iomux {
    volatile unsigned int unnames[23];
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO00;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO01;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO02;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO04;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO05;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO06;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO07;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO08;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO09;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_UART1_RX_DATA;
    volatile unsigned int IOMUXC_SW_MUX_CTL_PAD_UART1_CTS_B;
};

struct iomux *iomux = ioremap(0x20e0000, sizeof(struct iomux));

```

对于 GPIO，可以如下定义：

```

struct imxfull_gpio {
    volatile unsigned int dr;
    volatile unsigned int gdir;
    volatile unsigned int psr;
    volatile unsigned int icr1;
    volatile unsigned int icr2;
    volatile unsigned int imr;
    volatile unsigned int isr;
    volatile unsigned int edge_sel;
};

struct imxfull_gpio *gpio1 = ioremap(0x209C000, sizeof(struct imxfull_gpio));
struct imxfull_gpio *gpio5 = ioremap(0x20AC000, sizeof(struct imxfull_gpio));

```

15.4.4 测试

先启动 IMX6ULL QEMU 模拟器，挂载 NFS 文件系统。

运行 QEMU 时，
QEMU 内部为主机虚拟出一个网卡，IP 为 10.0.2.2，
IMX6ULL 有一个网卡，IP 为 10.0.2.15，
它连接到主机的虚拟网卡。
这样 IMX6ULL 就可以通过 10.0.2.2 去访问 Ubuntu 了。

安装驱动程序之后执行测试程序，观察它的返回值(执行测试程序的同时操作按键)：

```
# insmod button_drv.ko
# insmod board_drv.ko
# insmod board_100ask_imx6ull-qemu.ko
# ./button_test /dev/100ask_button0
# ./button_test /dev/100ask_button1
```

15.4.5 课后作业

- ① 修改 button_test.c，使用按键来点灯

16. 异常与中断的概念及处理流程

16.1 中断的引入

16.1.1 妈妈怎么知道孩子醒了



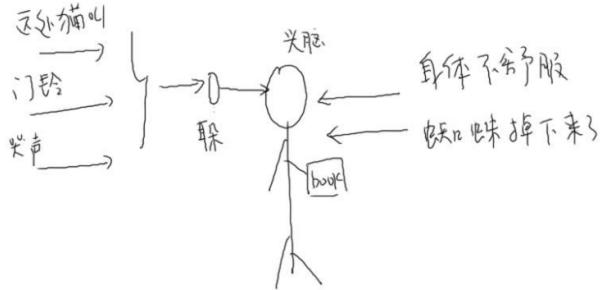
妈妈怎么知道卧室里小孩醒了？

- ① 时时进房间看一下：**查询方式**
简单，但是累
- ② 进去房间陪小孩一起睡觉，小孩醒了会吵醒她：**休眠-唤醒**
不累，但是妈妈干不了活了
- ③ 妈妈要干很多活，但是可以陪小孩睡一会儿，定个闹钟：**poll 方式**
要浪费点时间，但是可以继续干活。
妈妈要么是被小孩吵醒，要么是被闹钟吵醒。
- ④ 妈妈在客厅干活，小孩醒了他会自己走出房门告诉妈妈：**异步通知**
妈妈、小孩互不耽误。

后面的 3 种方式，都需要“小孩来中断妈妈”：中断她的睡眠、中断她的工作。

实际上，能“中断”妈妈的事情可多了：

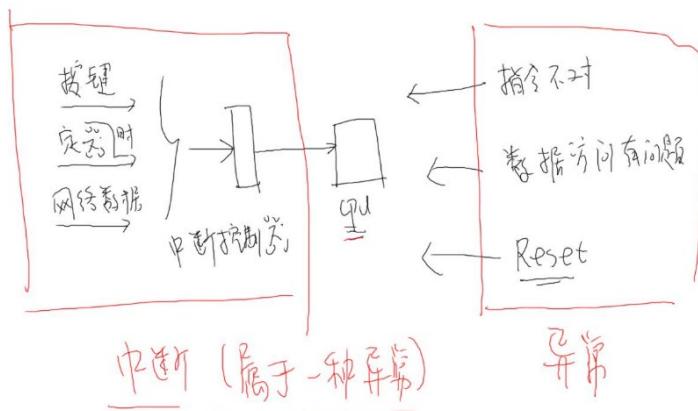
- ① 远处的猫叫：这可以被忽略
- ② 门铃、小孩哭声：妈妈的应对措施不一样
- ③ 身体不舒服：那要赶紧休息
- ④ 有蜘蛛掉下来了：赶紧跑啊，救命



妈妈目前正在看书，被“中断”后她会怎么做？流程如下：

- ① 妈妈正在看书
- ② 发生了各种声音
 - 可忽略的远处猫叫
 - 快递员按门铃
 - 卧室中小孩哭了
- ③ 妈妈怎么办？
 - a. 先在书中放入书签，合上书
 - b. 去处理
 - 对于不同的情况，处理方法不同：
 - 对于门铃：开门取快递
 - 对于哭声：照顾小孩
 - c. 回来继续看书

16.1.2 嵌入系统中也有类似的情况



CPU 在运行的过程中，也会被各种“异常”打断。这些“异常”有：

- ① 指令未定义
- ② 指令、数据访问有问题
- ③ SWI(软中断)
- ④ 快中断
- ⑤ 中断

中断也属于一种“异常”，导致中断发生的情况有很多，比如：

- ① 按键
- ② 定时器
- ③ ADC 转换完成
- ④ UART 发送完数据、收到数据
- ⑤ 等等

这些众多的“中断源”，汇集到“中断控制器”，由“中断控制器”选择优先级最高的中断并通知 CPU。

16.2 中断的处理流程

arm 对异常(中断)处理过程:

① 初始化:

- a. 设置中断源，让它可以产生中断
- b. 设置中断控制器(可以屏蔽某个中断，优先级)
- c. 设置 CPU 总开关(使能中断)

② 执行其他程序：正常程序

③ 产生中断：比如按下按键--->中断控制器--->CPU

④ CPU 每执行完一条指令都会检查有无中断/异常产生

⑤ CPU 发现有中断/异常产生，开始处理。

对于不同的异常，跳去不同的地址执行程序。

这地址上，只是一条跳转指令，跳去执行某个函数(地址)，这个就是异常向量。

③④⑤都是硬件做的。

⑥ 这些函数做什么事情？

软件做的:

- a. 保存现场(各种寄存器)
- b. 处理异常(中断):

分辨中断源，再调用不同的处理函数

c. 恢复现场

16.3 异常向量表

u-boot 或是 Linux 内核，都有类似如下的代码：

```
_start: b reset
    ldr pc, _undefined_instruction
    ldr pc, _software_interrupt
    ldr pc, _prefetch_abort
    ldr pc, _data_abort
    ldr pc, _not_used
    ldr pc, _irq //发生中断时，CPU 跳到这个地址执行该指令 **假设地址为 0x18**
    ldr pc, _fiq
```

这就是异常向量表，每一条指令对应一种异常。

发生复位时，CPU 就去 执行第 1 条指令：b reset。

发生中断时，CPU 就去执行“ldr pc, _irq”这条指令。

这些指令存放的位置是固定的，比如对于 ARM9 芯片中断向量的地址是 0x18。

当发生中断时，CPU 就强制跳去执行 0x18 处的代码。

在向量表里，一般都是放置一条跳转指令，发生该异常时，CPU 就会执行向量表中的跳转指令，去调用更复杂的函数。

当然，向量表的位置并不总是从 0 地址开始，很多芯片可以设置某个 vector base 寄存器，指定向量表在其他位置，比如设置 vector base 为 0x80000000，指定为 DDR 的某个地址。但是表中的各个异常向量的偏移地址，是固定的：复位向量偏移地址是 0，中断是 0x18。

16.4 参考资料

对于 ARM 的中断控制器，述语上称之为 GIC (Generic Interrupt Controller)，到目前已经更新到 v4 版本了。

各个版本的差别可以看这里：

<https://developer.arm.com/ip-products/system-ip/system-controllers/interrupt-controllers>

简单地说，GIC v3/v4 用于 ARMv8 架构，即 64 位 ARM 芯片。

而 GIC v2 用于 ARMv7 和其他更低的架构。

以后在驱动大全里讲解中断时，我们再深入分析，到时会涉及单核、多核等知识。

常见问题

1. 安装驱动时 version magic 不匹配

要想彻底了解内核的 LOCALVERSION 信息，可以看这个帖子：

<https://blog.csdn.net/gatieme/article/details/78510497>

总结一下：

- ① 开发板所用的内核版本：

在开发板上执行“uname -r”命令，可以得到开发板所用内核的版本，比如：

```
[root@firefly-rk3288:~]# uname -r  
4.4.154
```

- ② 在服务器中给开发板编译内核时，这个内核也有一个版本：

进入该内核源码目录，执行“make kernelrelease”命令，可以得到它的版本，比如：

```
book@book-virtual-machine:~/100ask_firefly-rk3288/linux-4.4$ make kernelrelease  
4.4.154+ ←
```

- ③ 编译驱动时，会用到服务器中开发板的内核源码，会带有它的版本信息。

如果①②③的版本信息不匹配，很可能导致驱动程序无法加载，比如：

```
root@firefly-rk3288:/mnt# insmod 100ask_led.ko  
[ 138.352988 100ask_led: version magic '4.4.154' SMP mod_unload ARMv7 p2v8 : should be '4.4.154' SMP mod_unload ARMv7 p2v8 :  
[ 138.3634547] 100ask_led: version magic '4.4.154' SMP mod_unload ARMv7 p2v8 : should be '4.4.154' SMP mod_unload ARMv7 p2v8 :  
insmod: can't insert "100ask_led.ko": invalid module format
```

有 2 个解决方法：

- A. 在 Ubuntu 上重新编译内核，让开发板使用新的内核启动；重新编译驱动，加载新驱动：

这样，①②③三者的内核版本就都一致了。

但是，这种方法有时候不好用，比如开发板上的内核无法更改(出厂固化了)，或者你没有开发板上所用内核的全部源码无法编译出内核，这时就可以使用下面的方法。

- B. 在 Ubuntu 上修改版本号，改为开发板上“uname -r”的结果，然后重新编译内核和驱动：

开发板就可以继续使用原来的内核，并且可以加载编译出来的驱动了。

步骤如下：

- b.1 修改 Ubuntu 上开发板内核源码顶层目录 Makefile，如下图：

```
define filechk_utsrelease.h  
    if [ `echo -n "$(KERNELRELEASE)" | wc -c` -gt $(uts_len) ]; then \  
        echo '"$(KERNELRELEASE)" exceeds $(uts_len) characters' >&2;  
        exit 1;  
    fi;  
    (echo \#define UTS_RELEASE \"$(KERNELRELEASE)\";)  
endef
```

→ 改为开发板上执行“uname -r”的结果

- b.2 重新编译内核，这会生成一些头文件，供驱动使用

- b.3 重新编译驱动