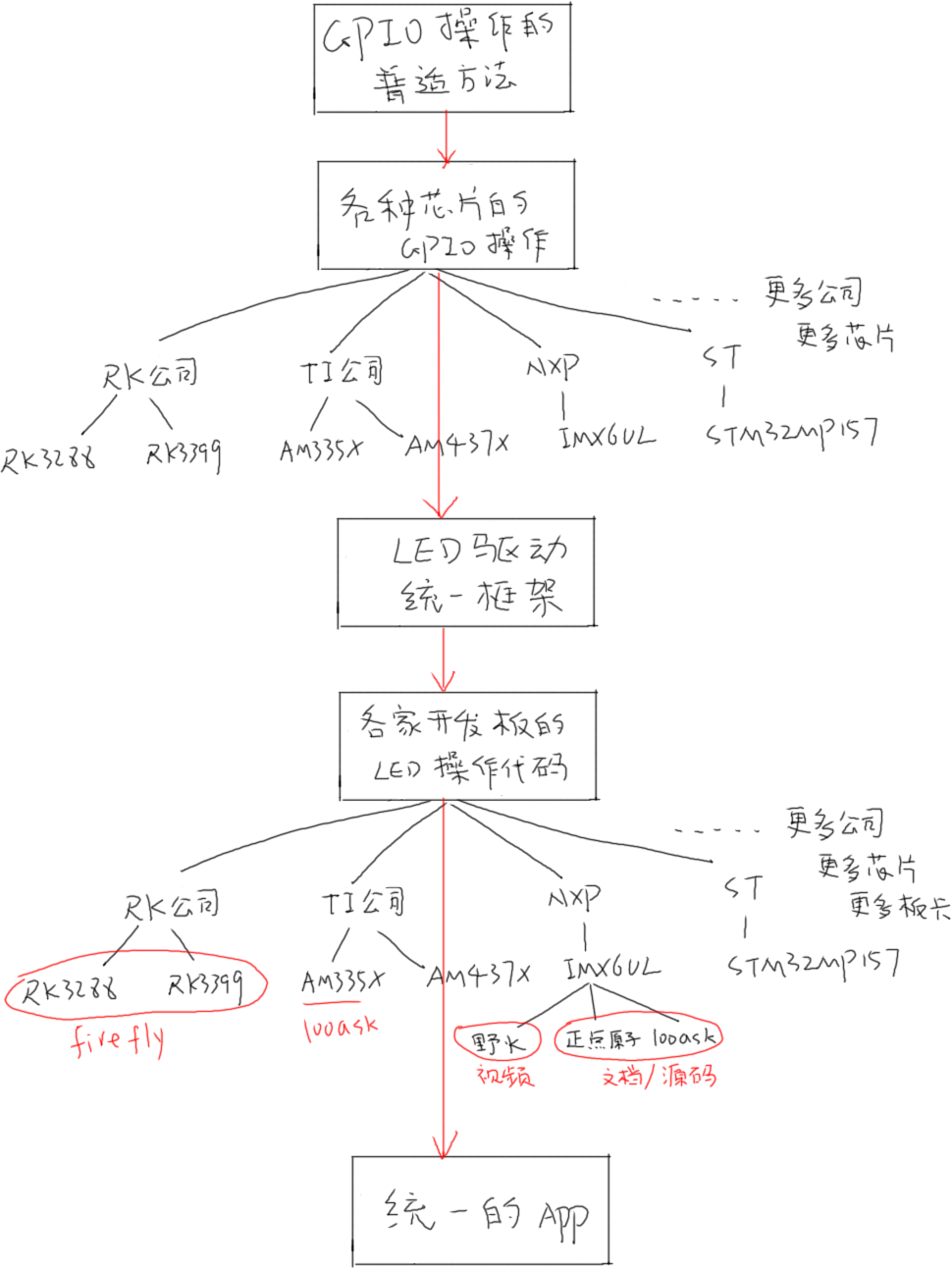
## 1. 嵌入式后Linux驱动开发基础知识的引导与说明

### 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。

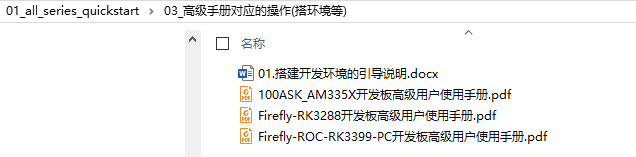
以LED驱动为例，会如下讲解：



### 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)：



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

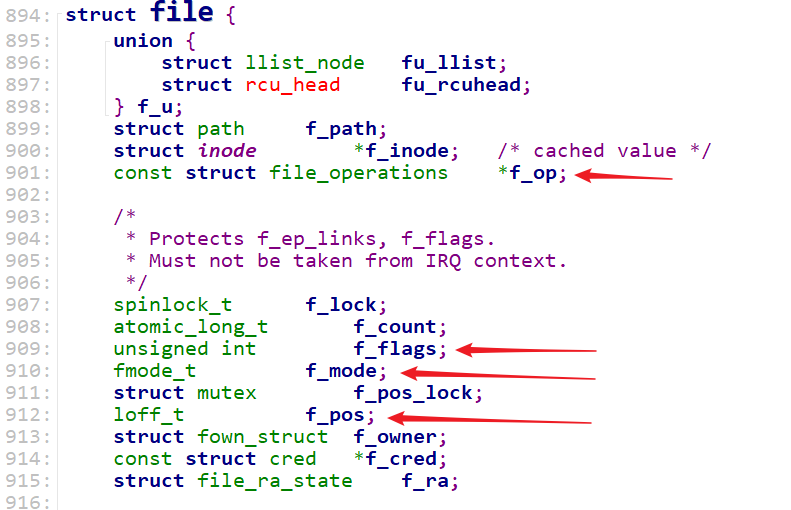
rk3288 linux 4.4.154

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

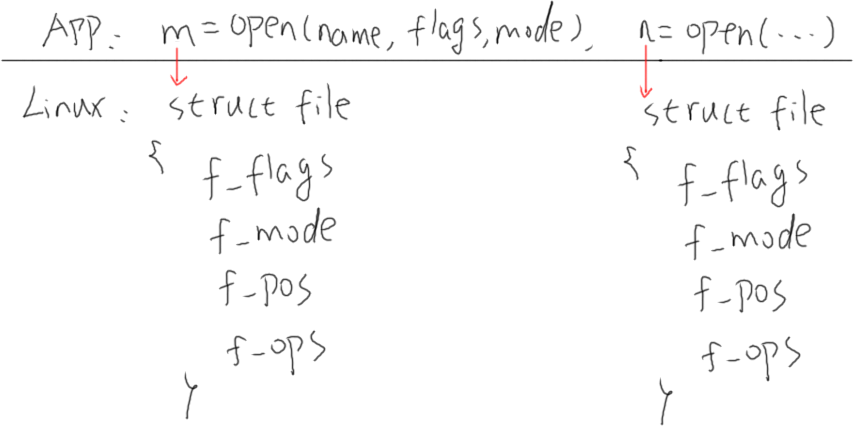
APP打开文件时，可以得到一个整数，这个整数被称为文件句柄。对于APP的每一个文件句柄，在内核里面都有一个“struct file”与之对应。



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)：

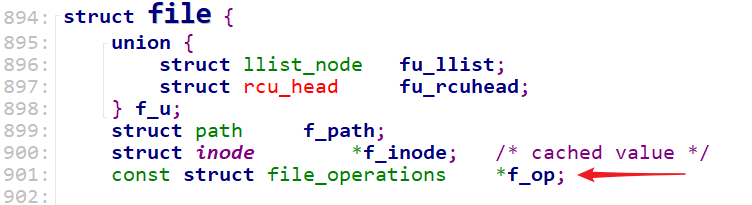
int open(const char \*pathname, int flags, mode\_t mode);

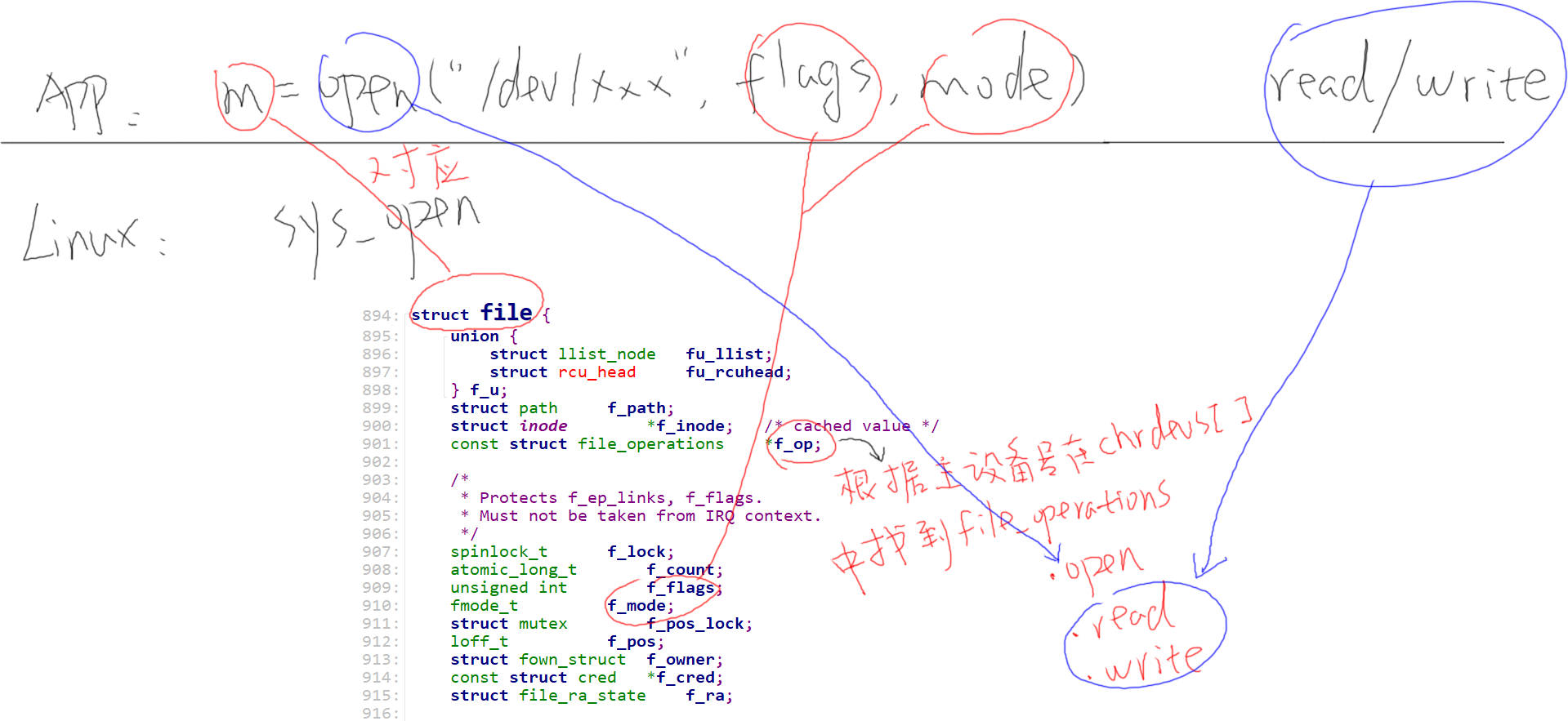
去读写文件时，文件的当前偏移地址也会保存在struct file结构体的f\_pos成员里。



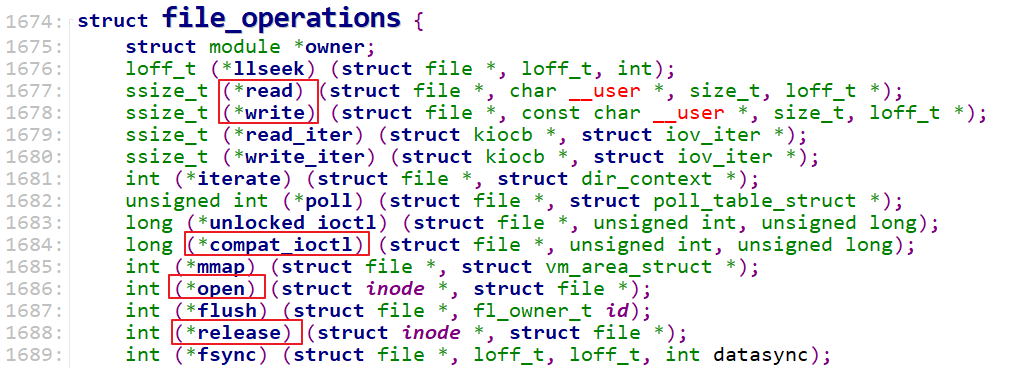
### 2.2 打开字符设备节点时，内核中也有对应的struct file

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。





结构体struct file\_operations的定义如下：



### 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：

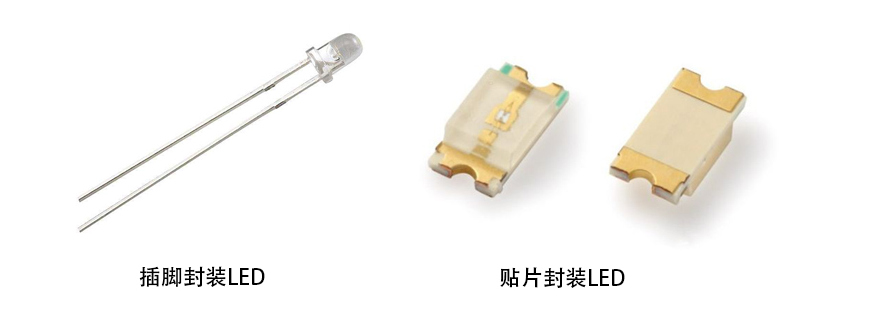
1.看原理图，确定控制LED的引脚;

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

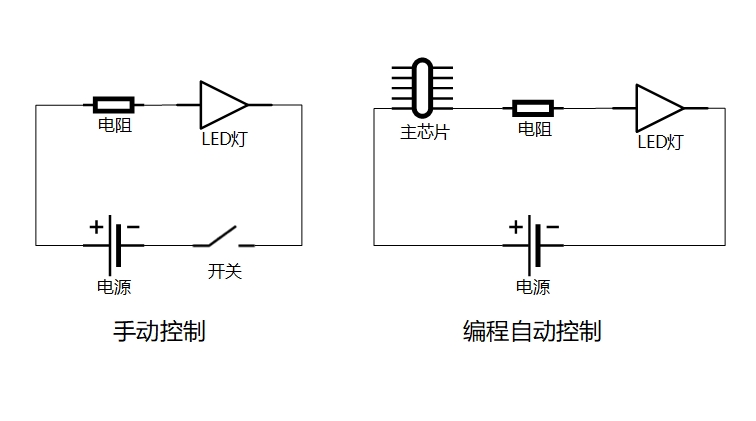
LED样子有很多种，像插脚的，贴片的。



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。



LED的驱动方式，常见的有四种。

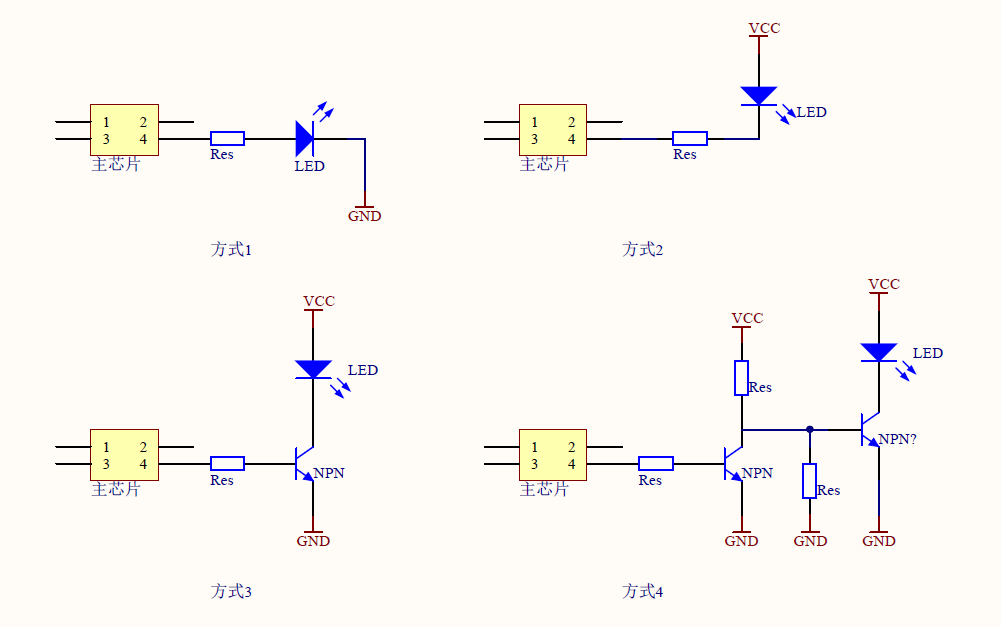
方式1：使用引脚输出3.3V点亮LED，输出0V熄灭LED。

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. 普适的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

1. 有多组GPIO，每组有多个GPIO
2. 使能：电源/时钟
3. 模式(Mode)：引脚可用于GPIO或其他功能
4. 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
5. 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

1. 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

1. 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

1. 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
2. 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

1. 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;

1. 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);

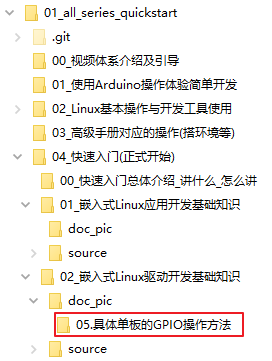
### 4.3 GPIO的其他功能：防抖动、中断、唤醒：

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。

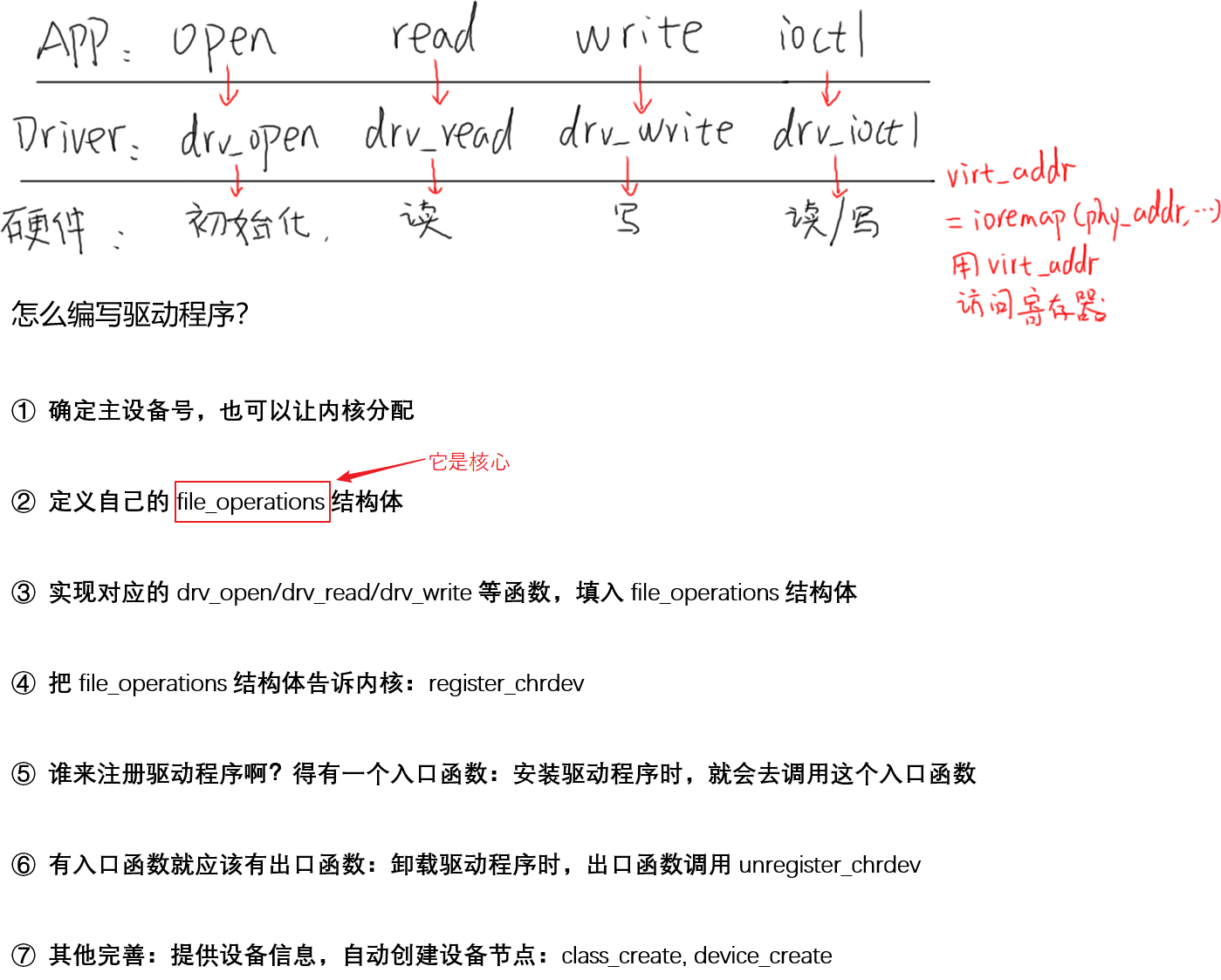
网盘中相同名字的目录下也有对应的视频。



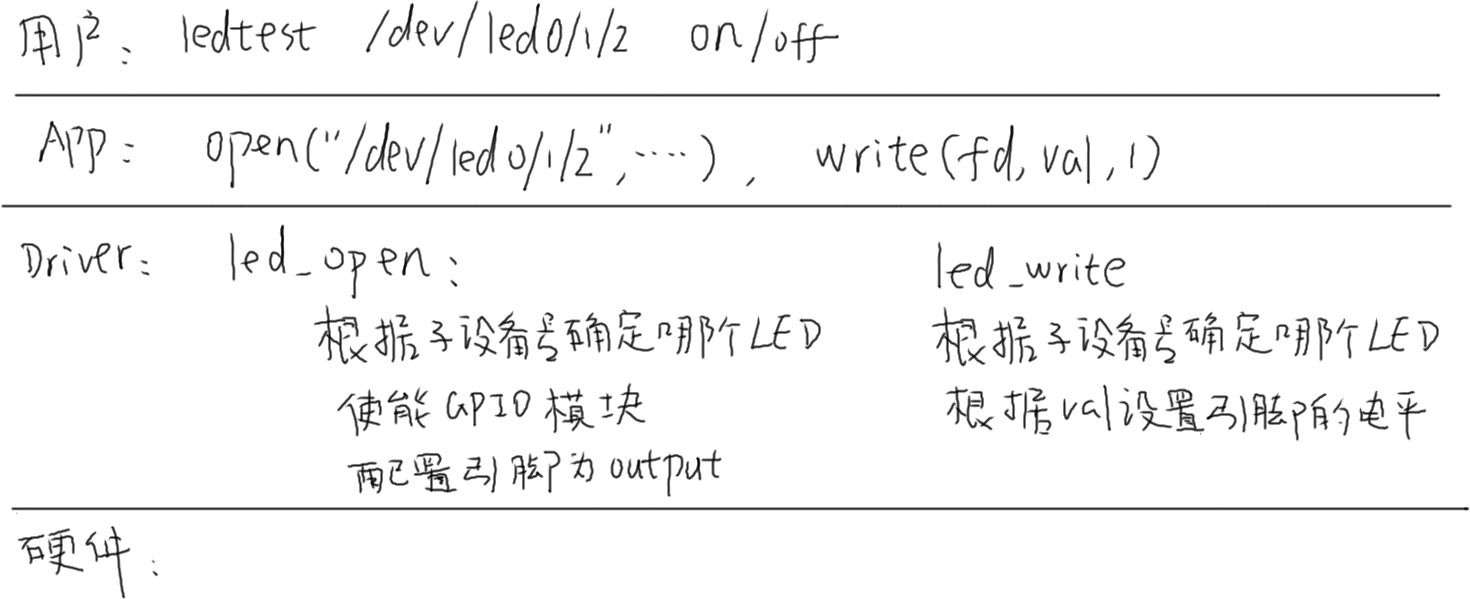
## 6. LED驱动程序框架

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

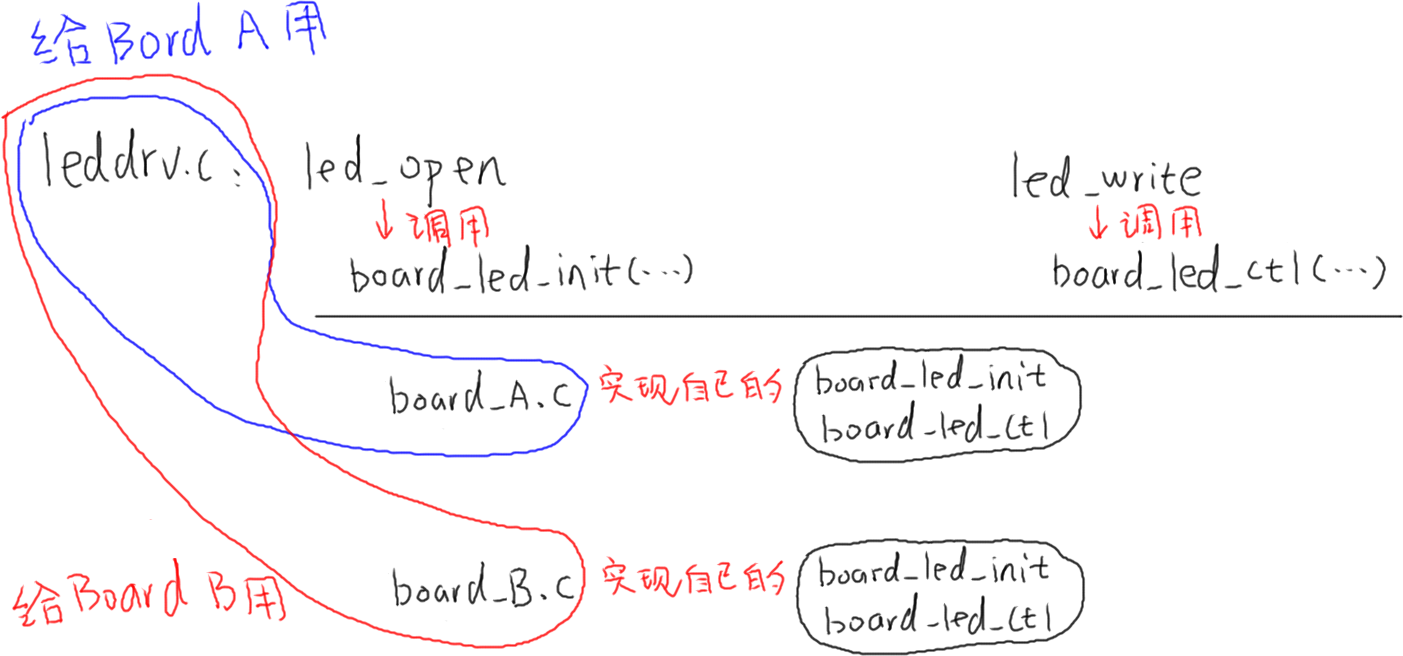


### 6.2 对于LED驱动，我们想要什么样的接口？



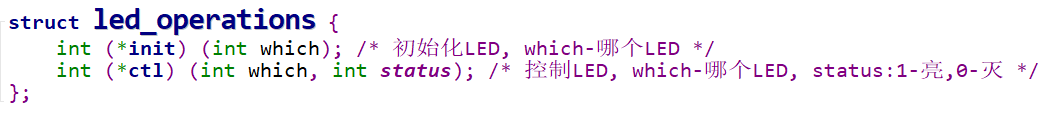
### 6.3 LED驱动要怎么写，才能支持多个板子？分层！

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

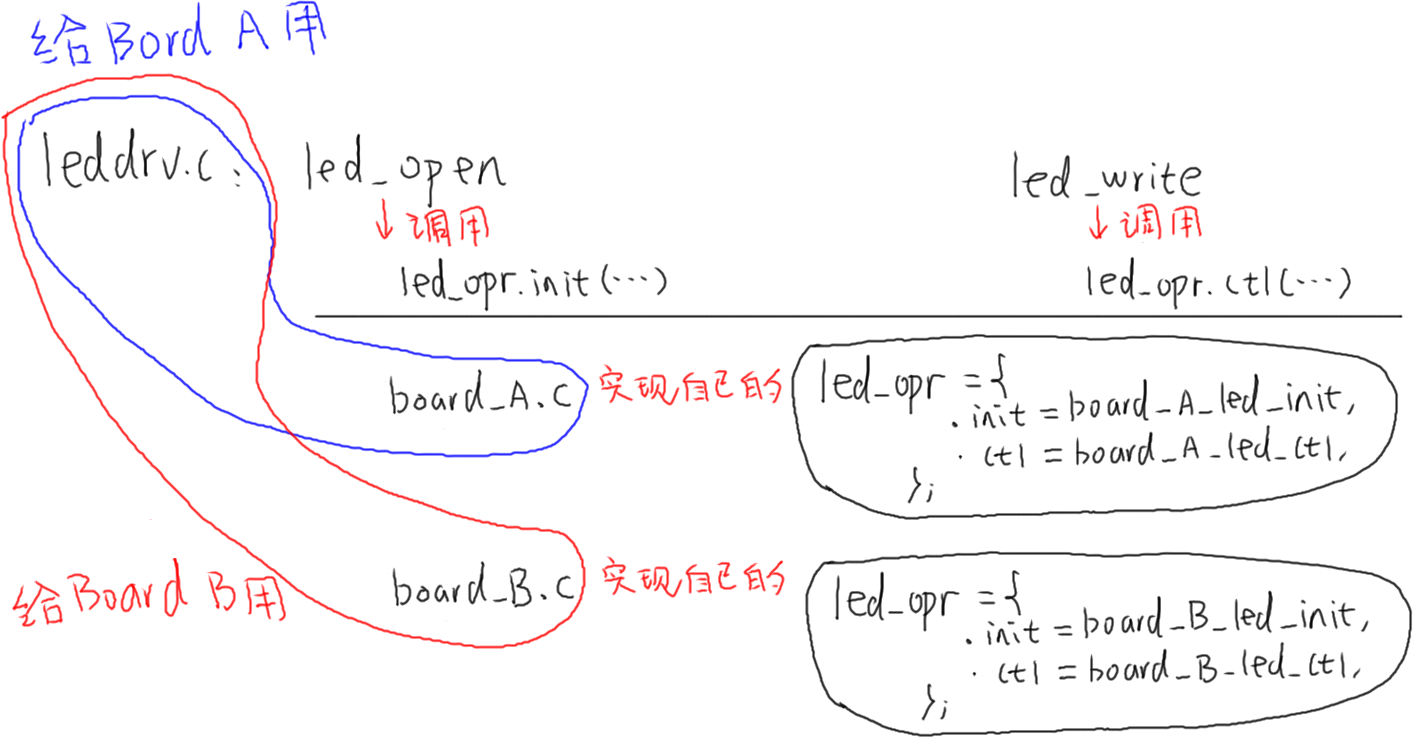


2. 以面向对象的思想，改进代码：

抽象出一个结构体：



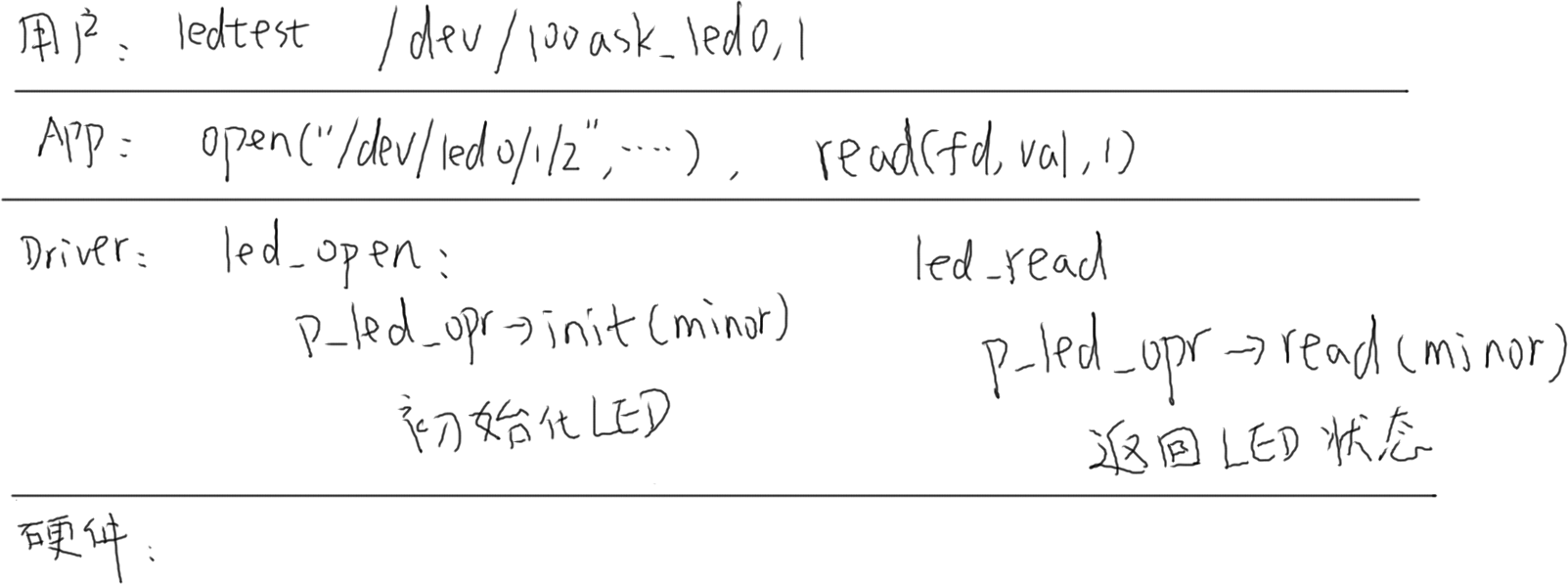
每个单板相关的board\_X.c实现自己的led\_operations结构体，供上层的leddrv.c调用：



### 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：



使用时，要包含头文件：



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(phys\_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)：



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

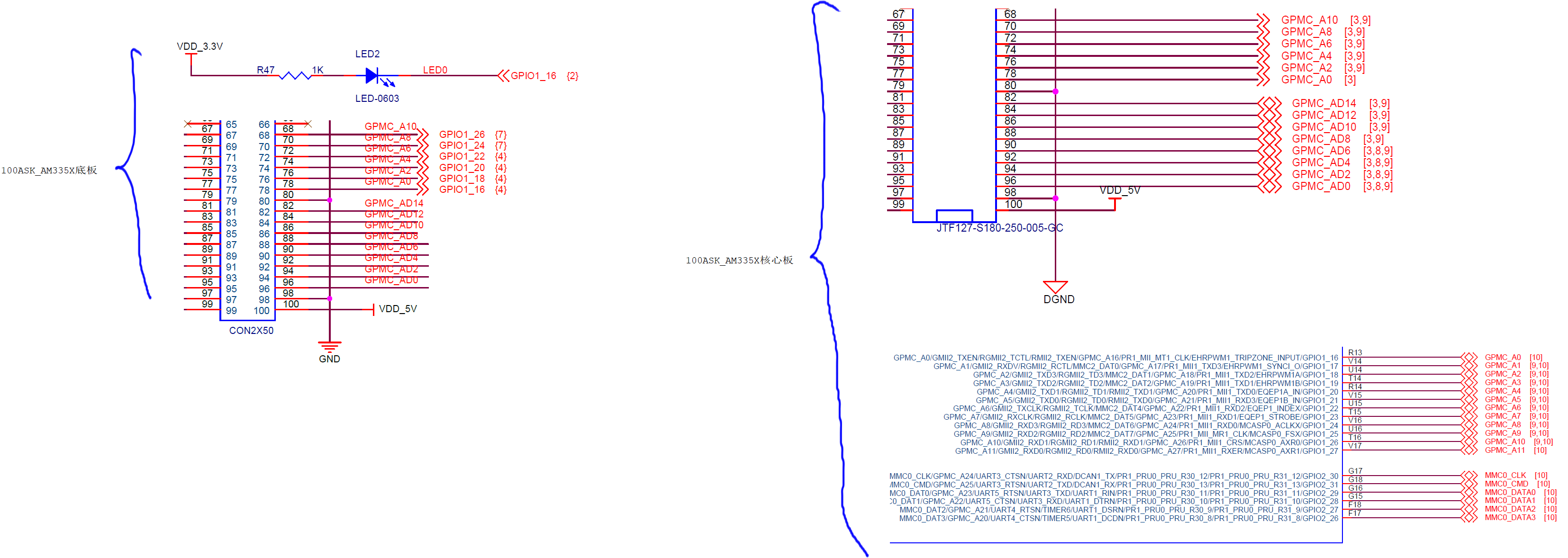
\*p = 0; // 点灯，这句话不会被优化掉

\*p = 1; // 灭灯

### 7.1 AM335X的LED驱动程序

#### 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：



#### ② 所涉及的寄存器操作

a. 使能GPIO1

/\* set PRCM to enalbe 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

\*/

#### ③ 写程序

#### 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？

#### ⑤ 课后作业

a. 在board\_am335x.c里有ioremap，什么时候执行iounmap？请完善程序

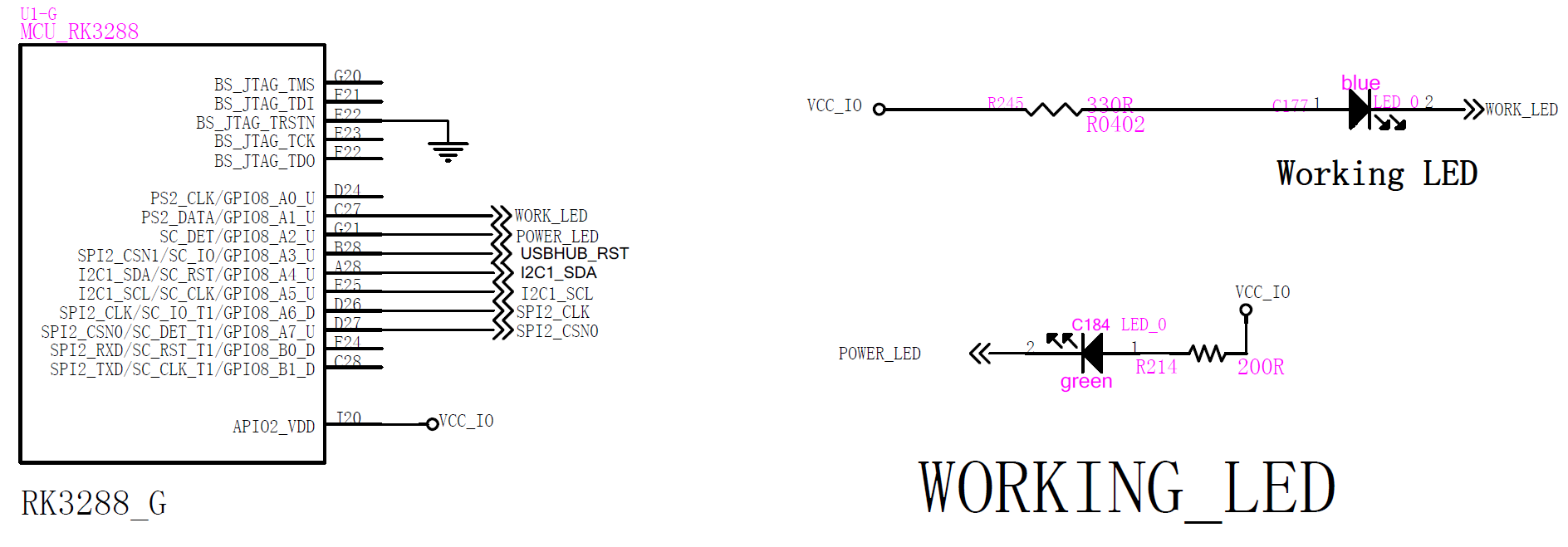
b. 视频里我们只实现了点一个LED，请修改代码实现操作4个LED

### 7.2 RK3288和RK3399的LED驱动程序

#### ① 原理图

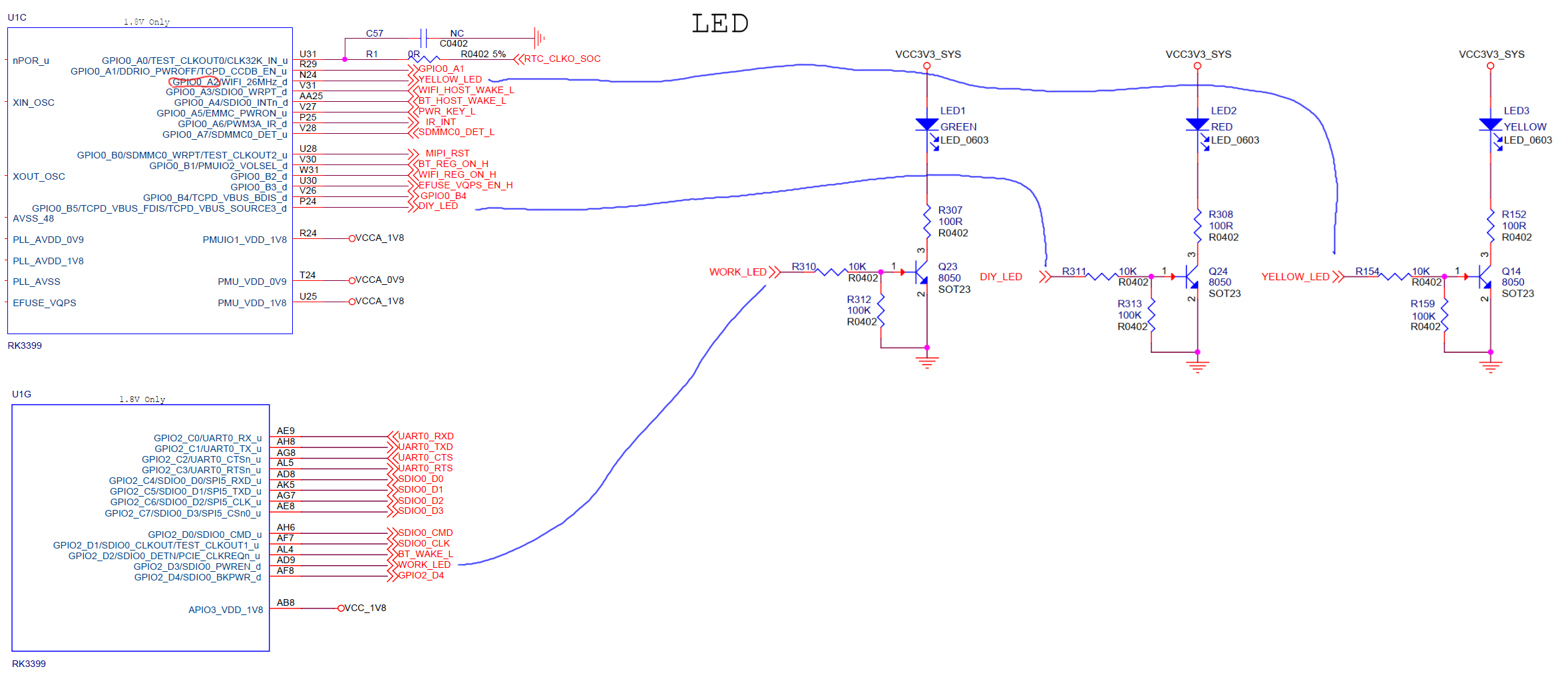
* 1. fireflye RK3288的LED原理图：

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：



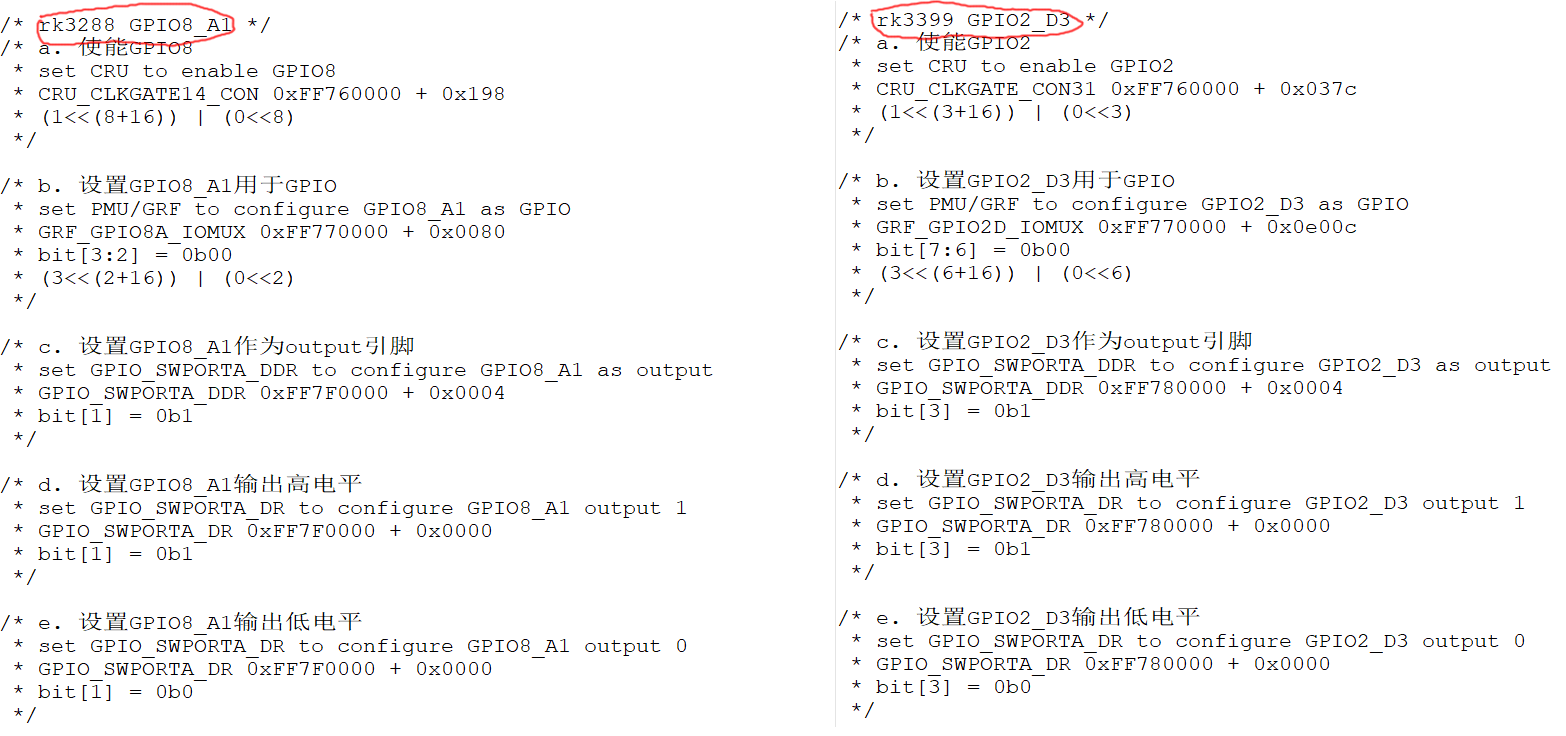
* 1. firefly RK3399的LED原理图：

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：



#### 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：



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)

\*/

/\* b. 设置GPIO8\_A1作为output引脚

\* set GPIO\_SWPORTA\_DDR to configure GPIO8\_A1 as output

\* GPIO\_SWPORTA\_DDR 0xFF7F0000 + 0x0004

\* bit[1] = 0b1

\*/

/\* c. 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

\*/

/\* d. 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

\*/

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[3] = 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[3] = 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[3] = 0b0

\*/

#### ③ 写程序

#### 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“heatbeat”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

#### ④ 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，请修改代码实现操作所有LED

### 7.3 IMX6UL/6ULL的LED驱动程序

#### ① 原理图

a) 野火fire\_imx6ull-pro开发板的LED原理图：

b) 正点原子Atk\_imx6ull-alpha开发板的LED原理图：

## 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，可以看这个贴子：

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

总结一下：

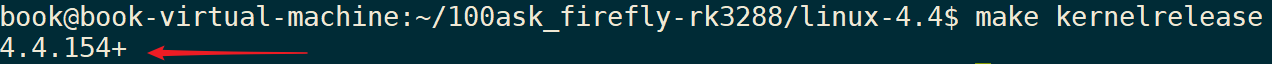
① 开发板所用的内核版本：

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：



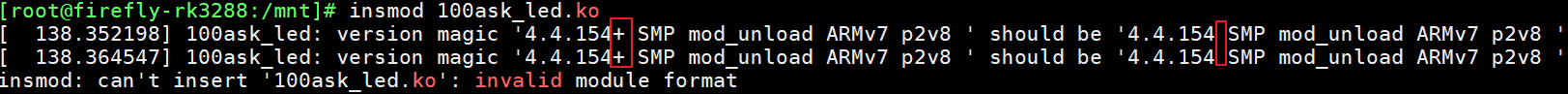
② 在服务器中给开发板编译内核时，这个内核也有一个版本：

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：



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：



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。

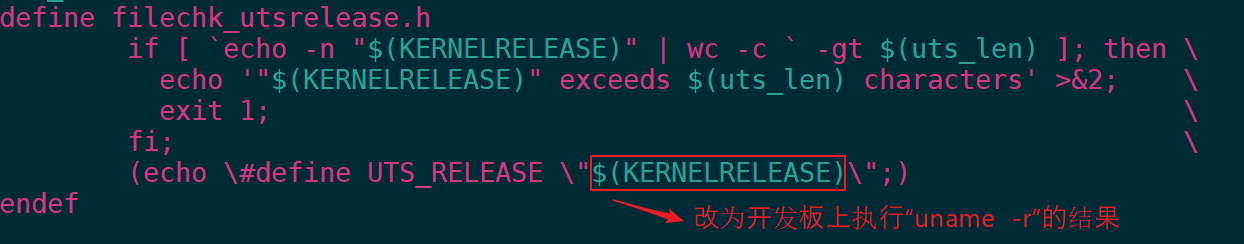
但是，这种方法有时候不好用，比如开发板上的内核无法更改(出厂固化了)，或者你没有开发板上所用内核的全部源码无法编译出内核，这时就可以使用下面的方法。

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：



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