**一、led子系统框架的好处**

内核驱动维护者针对每个种类的驱动设计一套成熟的、标准的、典型的驱动实现，并把不同厂家的同类硬件相同的部分抽出来自己实现好，再把不同部分留出接口给具体的驱动开发工程师来实现，这就叫驱动框架。

编写一个led灯驱动，我们可以使用字符设备驱动，也可以直接利用gpio口。不过linux系统自己也带有led子系统，不同平台都可以用linux的led子系统，应用程序不用做任何的改变，就可以在新的平台上运行，可移植性好。

**二、led子系统源码分析**

内核开发者对驱动框架进行开发和维护、升级，对应led-class.c和led-core.c

SoC厂商的驱动工程师对设备驱动源码进行编写、调试，提供参考版本，对应leds-xxx.c

做产品的厂商的驱动工程师以SoC厂商提供的驱动源码为基础，来做移植和调试

使用led-class.c和led-core.c之前先确认这两个文件是否编译进了内核：

vi drivers/leds/Makefile

obj-$(CONFIG\_NEW\_LEDS) += led-core.o

obj-$(CONFIG\_LEDS\_CLASS) += led-class.o

vi drivers/leds/Kconfig

8 menuconfig NEW\_LEDS

9 bool "LED Support"

19 config LEDS\_CLASS

20 tristate "LED Class Support"

所以要在menuconfig中选中LED Support和LED Class Support两个选项

其中led-core.c都是单独的一些函数源码，函数之间没有必然联系，所以暂时不管。再看led-class.c：

279 leds\_class = class\_create(THIS\_MODULE, "leds"); 在sys/class/目录下创建leds文件夹

282 leds\_class->pm = &leds\_class\_dev\_pm\_ops;

283 leds\_class->dev\_groups = led\_groups;

继续跟进leds\_class\_dev\_pm\_ops：

201 static const struct dev\_pm\_ops leds\_class\_dev\_pm\_ops = {

202 .suspend = led\_suspend,

203 .resume = led\_resume,

204 };

其中led\_suspend和led\_resume都是函数。

跟进led\_suspend：

181 static int led\_suspend(struct device \*dev)

182 {

183 struct led\_classdev \*led\_cdev = dev\_get\_drvdata(dev);获取设备私有数据，如果没有返回NULL

184

185 if (led\_cdev->flags & LED\_CORE\_SUSPENDRESUME)

要使条件成立我们自己的驱动中需要使用LED\_CORE\_SUSPENDRESUME

186 led\_classdev\_suspend(led\_cdev);如果条件成立则执行此函数

187

188 return 0;

189 }

led\_classdev结构体介绍：

struct led\_classdev {

    const char        \*name;         //led名字

    int             brightness;    //当前亮度

    int             max\_brightness;//参考值，最大亮度

    int             flags;  //标志，目前只支持 LED\_SUSPENDED和LED\_CORE\_SUSPENDRESUME

//当设置/sys/class/leds/下的led接口里的brightness(亮度)属性文件时，会回调该函数

    void        (\*brightness\_set)(struct led\_classdev \*led\_cdev,enum led\_brightness brightness);

...................

}

跟进led\_classdev\_suspend：

163 void led\_classdev\_suspend(struct led\_classdev \*led\_cdev)

164 {

165 led\_cdev->flags |= LED\_SUSPENDED;

166 led\_cdev->brightness\_set(led\_cdev, 0);

167 }

上面的brightness\_set函数是我们自己写的驱动封装的函数，第一个参数是结构体led\_classdev，第二个参数是led灯的状态，0代表关闭。也就是说我们自己实现的驱动最终要调用brightness\_set函数，修改它的第二个参数值来控制灯的状态。

那么如何调用到brightness\_set函数接口呢？继续跟进led\_groups：

92 static const struct attribute\_group \*led\_groups[] = {

93 &led\_group,

94 #ifdef CONFIG\_LEDS\_TRIGGERS

95 &led\_trigger\_group,

96 #endif

97 NULL,

98 };

跟进led\_group：

88 static const struct attribute\_group led\_group = {

89 .attrs = led\_class\_attrs,

90 }; ||

\/

82 static struct attribute \*led\_class\_attrs[] = {

83 &dev\_attr\_brightness.attr,

84 &dev\_attr\_max\_brightness.attr,

85 NULL,

86 };

这里要注意dev\_attr\_brightness和dev\_attr\_max\_brightness哪来的呢？

分析下面两个宏函数

60 static DEVICE\_ATTR\_RW(brightness);

==》540 #define DEVICE\_ATTR\_RW(\_name) \

541 struct device\_attribute dev\_attr\_##\_name = \_\_ATTR\_RW(\_name)

其中##\_name是一个变量，内容由\_name参数决定，而我们传递的实际参数是brightness

所以static DEVICE\_ATTR\_RW(brightness)最终结果就是device\_attribute dev\_attr\_brightness

==》89 #define \_\_ATTR\_RW(\_name) \_\_ATTR(\_name, (S\_IWUSR | S\_IRUGO), \

90 \_name##\_show, \_name##\_store)

所以创建的属性文件叫做brightness,并且最终会调用brightness\_show和brightness\_store两个函数，也就是说在终端执行echo 1 > brigntness时调用过程如下：

brightness\_store()

==》\_\_led\_set\_brightness(led\_cdev, state);

==》led\_cdev->brightness\_set(led\_cdev, value);需要我们自己封装。

69 static DEVICE\_ATTR(max\_brightness, 0444, led\_max\_brightness\_show, NULL);

dev\_attr\_max\_brightness和dev\_attr\_brightness的调用方法一致，我们可以通过echo来操作max\_brightness属性文件，只不过这个属性文件存放的值代表的是最大亮度。

**三、通过led子系统实现led灯的亮灭**

5、封装brightness\_set接口

{

配置led灯的数据寄存器使灯亮或灭

}

加载函数

{

1、给struct led\_classdev结构体分配空间

2、初始化led\_classdev的name、brightness、flags、brightness\_set等成员

3、执行led\_classdev\_register函数，在sys/leds/xxx/目录下产生一个设备

4、配置led灯控制寄存器

}

卸载函数

{

kfree();

led\_classdev\_unresigter();

}