Linux下的Backlight子系统（二）

分类： 子系统 linux驱动 Mini2440 2013-01-18 09:01 3580人阅读 评论(0) 收藏 举报

原创作者：南京邮电大学 通信与信息系统专业 研二 魏清

三．Backlight核心驱动

下面我们讲讲backlight子系统。背光子系统目录在/driver/video/backlight下，其中背光子系统核心代码是backlight.c

先查看/driver/video/backlight/Makefile

```
obj-$(CONFIG_BACKLIGHT_CLASS_DEVICE) += backlight.o
```

继续查看/driver/video/backlight/Kconfig

```
config BACKLIGHT_CLASS_DEVICE
        tristate "Lowlevel Backlight controls"
        depends on BACKLIGHT_LCD_SUPPORT
        default m
```

所以配置内核make menuconfig时，需要选中这一项。

下面看backlight背光的核心代码backlight.c

```
static int __init backlight_class_init(void)
{
        backlight_class = class_create(THIS_MODULE, "backlight"); //注册backlight类
        if (IS_ERR(backlight_class)) {
                printk(KERN_WARNING "Unable to create backlight class; errno = %ld\n",
                                PTR_ERR(backlight_class));
                return PTR_ERR(backlight_class);
        }
        backlight_class->dev_attrs = bl_device_attributes;  //添加类属性
        backlight_class->suspend = backlight_suspend;
        backlight_class->resume = backlight_resume;
        return 0;
}
```

我们知道backlight背光子系统的主要就是靠这个类属性，当我们设置背光值就是向类属性中某个成员写背光值，这个类属性就是给用户的一种接口，我们重点看看

```
#define __ATTR(_name,_mode,_show,_store) { \
        .attr = {.name = __stringify(_name), .mode = _mode },    \
        .show     = _show,                      \
        .store    = _store,                     \
}
static struct device_attribute bl_device_attributes[] = {
        __ATTR(bl_power, 0644, backlight_show_power, backlight_store_power),
```

```
    __ATTR(brightness, 0644, backlight_show_brightness,
            backlight_store_brightness),
    __ATTR(actual_brightness, 0444, backlight_show_actual_brightness,
            NULL),
    __ATTR(max_brightness, 0444, backlight_show_max_brightness, NULL),
    __ATTR_NULL,
};
```

很明显，在backlight类中我们创建了bl_power，brightness，actural_brightness，max_brightness四个成员，其中brightness是当前亮度，max_brightness是最大亮度。当用户层通过cat或者echo命令就会触发这些成员。对于这些属性的读写函数，我们先看看读的函数backlight_show_max_brightness吧

```
static ssize_t backlight_show_max_brightness(struct device *dev,
        struct device_attribute *attr, char *buf)
{
    struct backlight_device *bd = to_backlight_device(dev);
    return sprintf(buf, "%d\n", bd->props.max_brightness);  //输出最大亮度
}
```

这个函数很简单，但是重点是引入了几个backlight背光子系统的几个重要的数据结构，我们好好学习下。

首先是backlight背光子系统的设备结构体backlight_device

```
struct backlight_device {
    struct backlight_properties props;  //背光属性
    struct mutex update_lock;
    struct mutex ops_lock;
    struct backlight_ops *ops;   //背光操作函数，类似于file_operations
    struct notifier_block fb_notif;
    struct device dev;  //内嵌设备
};
```

下面先看看背光属性结构体backlight_properties

```
struct backlight_properties {
    int brightness;  //当前背光值
    int max_brightness;  //最大背光值
    int power;
    int fb_blank;
    unsigned int state;
};
```

再看看背光操作函数结构体

```
struct backlight_ops {
    unsigned int options;
#define BL_CORE_SUSPENDRESUME      (1 << 0)
    int (*update_status)(struct backlight_device *);   //改变背光状态
    int (*get_brightness)(struct backlight_device *);  //获取背光值
    int (*check_fb)(struct fb_info *);
};
```

好了，我们继续看backlight类属性中写的函数，例如设置当前背光值函数backlight_store_brightness吧

```
static ssize_t backlight_store_brightness(struct device *dev,
        struct device_attribute *attr, const char *buf, size_t count)
{
    int rc;
    struct backlight_device *bd = to_backlight_device(dev);
```

```
    unsigned long brightness;
    rc = strict_strtoul(buf, 0, &brightness);
    if (rc)
        return rc;
    rc = -ENXIO;
    mutex_lock(&bd->ops_lock);
    if (bd->ops) {
        if (brightness > bd->props.max_brightness)
            rc = -EINVAL;
        else {
            pr_debug("backlight: set brightness to %lu\n",
                 brightness);
            bd->props.brightness =brightness;  //传入背光值
            backlight_update_status(bd);  //调用backlight_update_status设备背光值
            rc = count;
        }
    }
    mutex_unlock(&bd->ops_lock);
    backlight_generate_event(bd, BACKLIGHT_UPDATE_SYSFS);
    return rc;
}
```

跟踪backlight_update_status

```
static inline void backlight_update_status(struct backlight_device *bd)
{
    mutex_lock(&bd->update_lock);
    if (bd->ops && bd->ops->update_status)
        bd->ops->update_status(bd); //调用背光操作函数中改变背光状态函数update_status
    mutex_unlock(&bd->update_lock);
}
```

对于这个backlight背光核心层驱动backlight.c，剩下的就是这个pwm.c给我们提供了哪些接口函数了。

```
struct backlight_device *backlight_device_register(const char *name,
        struct device *parent, void *devdata, struct backlight_ops *ops)
void backlight_device_unregister(struct backlight_device *bd)
EXPORT_SYMBOL(backlight_device_register);  //注册背光设备
EXPORT_SYMBOL(backlight_device_unregister); //注销背光设备
```

这些接口很简单，就不细说了，这样我们的backlight子系统的核心层就介绍完了。

四．基于PWM&Backlight的蜂鸣器驱动

下面我们结合上面的PWM核心层和Backlight背光子系统核心层，根据基于pwm的背光驱动/driver/video/backlight/pwm_bl.c来修改成基于Mini2440的蜂鸣器驱动。

先查看/driver/video/backlight/Makefile

```
obj-$(CONFIG_BACKLIGHT_PWM)   += pwm_bl.o
```

继续查看/driver/video/backlight/Kconfig

```
config BACKLIGHT_PWM
    tristate "Generic PWM based Backlight Driver"
    depends on BACKLIGHT_CLASS_DEVICE && HAVE_PWM
    help
      If you have a LCD backlight adjustable by PWM, say Y to enable
```

this driver.

我们的HAVE_PWM和BACKLIGHT_CLASS_DEVICE分别是在前面讲pwm核心和backlight核心时已经编译了，所以配置内核make menuconfig 时，需要再选中"Generic PWM based Backlight Driver"这项。

好了，我们先把我们的蜂鸣器移植进去吧，首先我们知道蜂鸣器使用的是GPB0端口，该端口如果工作在TOU0模式，就可以通过设备定时器的TCNT和TCMP来控制定时器的波形而来。先打开mini2440的BSP文件mach-mini2440.c，如下添加

```
static struct platform_device s3c_backlight_device = {
    .name           = "pwm-backlight",        //设备名
    .dev        = {
        .parent    = &s3c_device_timer[0].dev,  //该设备基于pwm中的0号定时器
        .platform_data = &s3c_backlight_data,
    },
    .id=0,    //对应的就是pwm0
};
```

添加平台数据

```
static struct platform_pwm_backlight_data s3c_backlight_data = {
    .pwm_id        = 0,  //对应的就是Timer0
    .max_brightness    = 1000,  //最大亮度
    .dft_brightness    = 10 ,    //当前亮度
    .pwm_period_ns    = 800000,  //这就是前面说的T0，即输出时钟周期
    .init        = s3c_bl_init,  //端口初始化
};
```

注意到平台数据中定义了init函数，由于在蜂鸣器的初始化时，需要对GPB0设置为TOUT0模式，所以代码如下编写

```
static int s3c_bl_init(struct device *dev)
{
    s3c2410_gpio_pullup(S3C2410_GPB(0),0);   // GPB0不上拉
    s3c2410_gpio_cfgpin(S3C2410_GPB(0),S3C2410_GPB0_TOUT0); // GPB0设置为TOUT0
    return 0;
}
```

然后把这个s3c_backlight_device加入到mini2440_devices数组

```
static struct platform_device *mini2440_devices[] __initdata = {
    ……
    &s3c_device_timer[0],
    &s3c_backlight_device, //添加
};
```

最后添加头文件

```
#include < linux/pwm_backlight.h>
```

这样配置完后，进行make zImage生成zImage内核镜像。

好了，下面我们分析下基于pwm的背光驱动/driver/video/backlight/pwm_bl.c

```
static struct platform_driver pwm_backlight_driver = {
    .driver         = {
        .name     = "pwm-backlight", //驱动名
        .owner    = THIS_MODULE,
    },
    .probe          = pwm_backlight_probe, //探测函数
    .remove         = pwm_backlight_remove,
```

```
    .suspend  = pwm_backlight_suspend,
    .resume         = pwm_backlight_resume,
};
static int __init pwm_backlight_init(void)
{
    return platform_driver_register(&pwm_backlight_driver);
}
```

注意上面的pwm_backlight_driver中的驱动名"pwm-backlight"和我们刚才移植时添加的设备名"pwm-backlight"是一致的，这样设备和驱动就能匹配成功。下面看探测函数

```
static int pwm_backlight_probe(struct platform_device *pdev)
{
    struct platform_pwm_backlight_data *data = pdev->dev.platform_data;
    struct backlight_device *bl;
    struct pwm_bl_data *pb;  //本驱动的私有结构体
    int ret;
    if (!data) {
        dev_err(&pdev->dev, "failed to find platform data\n");
        return -EINVAL;
    }
    if (data->init) {  //初始化端口，这个端口函数在BSP中定义
        ret = data->init(&pdev->dev);
        if (ret < 0)
            return ret;
    }
    pb = kzalloc(sizeof(*pb), GFP_KERNEL); //分配pwm_bl_data空间
    if (!pb) {
        dev_err(&pdev->dev, "no memory for state\n");
        ret = -ENOMEM;
        goto err_alloc;
    }
    pb->period = data->pwm_period_ns;   //获取周期
    pb->notify = data->notify;
    pb->pwm = pwm_request(data->pwm_id, "backlight"); //注册pwm设备
    if (IS_ERR(pb->pwm)) {
        dev_err(&pdev->dev, "unable to request PWM for backlight\n");
        ret = PTR_ERR(pb->pwm);
        goto err_pwm;
    } else
        dev_dbg(&pdev->dev, "got pwm for backlight\n");
    bl = backlight_device_register(dev_name(&pdev->dev), & pdev->dev,
            pb, &pwm_backlight_ops); //注册backlight设备
    if (IS_ERR(bl)) {
        dev_err(&pdev->dev, "failed to register backlight\n");
        ret = PTR_ERR(bl);
        goto err_bl;
    }
    bl->props.max_brightness = data->max_brightness;
```

```
    bl->props.brightness = data->dft_brightness;
    backlight_update_status(bl);  //先点亮背光
    platform_set_drvdata(pdev, bl); //设置bl为私有数据
    return 0;
err_bl:
    pwm_free(pb->pwm);
err_pwm:
    kfree(pb);
err_alloc:
    if (data->exit)
        data->exit(&pdev->dev);
    return ret;
}
```

对于这个驱动，我们重点关注的是注册backlight设备时传入的参数pwm_backlight_ops，因为我们之前分析backlight背光子系统时说过，背光设备结构体中有个操作背光的函数集合，在我们的pwm_bl.c中，就需要定义这个操作背光的函数集合，也就是pwm_backlight_ops

```
static struct backlight_ops pwm_backlight_ops = {
    .update_status = pwm_backlight_update_status, //更新背光亮度
    .get_brightness      = pwm_backlight_get_brightness, //获取背光亮度
};
```

获取背光亮度函数pwm_backlight_get_brightness很简单，跟踪得到

```
static int pwm_backlight_get_brightness(struct backlight_device *bl)
{
    return bl->props.brightness;
}
```

我们重点看更新背光亮度函数pwm_backlight_update_status

```
static int pwm_backlight_update_status(struct backlight_device *bl)
{
    struct pwm_bl_data *pb = dev_get_drvdata(&bl->dev);
    int brightness = bl->props.brightness;
    int max = bl->props.max_brightness;
    if (bl->props.power != FB_BLANK_UNBLANK)
        brightness = 0;
    if (bl->props.fb_blank != FB_BLANK_UNBLANK)
        brightness = 0;
    if (pb->notify)
        brightness = pb->notify(brightness);
    if (brightness == 0) {  //背光值为0，关闭背光
        pwm_config(pb->pwm, 0, pb->period);
        pwm_disable(pb->pwm);
    } else {   //调用pwm中的API设置背光
        pwm_config(pb->pwm, brightness * pb->period / max, pb->period);
        pwm_enable(pb->pwm);
    }
    return 0;
}
```

好了，这样我们的pwm_bl.c也分析完了。在使用backlight子系统的时候，我们只需要在probe函数中注册pwm和backlight设备，然后定义背光操作函数集合即可。

五．驱动测试

实验环境：内核linux2.6.32.2，arm-linux-gcc交叉编译器，mini2440开发板

下面我们进行对上面的驱动进行测试，按照上面的步骤操作，将上文已经编译好的zImage烧入开发板，通过超级终端控制，能控制蜂鸣器的发出的声音频率。

```
[root@FriendlyARM pwm-backlight.0]# pwd
/sys/class/backlight/pwm-backlight.0
[root@FriendlyARM pwm-backlight.0]# ls
actual_brightness  brightness         max_brightness     uevent
bl_power           device             subsystem
[root@FriendlyARM pwm-backlight.0]# echo 10 >brightness
[root@FriendlyARM pwm-backlight.0]# echo 0 >brightness
[root@FriendlyARM pwm-backlight.0]#
```


个人资料

weiqing1981127
- 访问：110842次
- 积分：1645
- 等级：
- 排名：第13712名
- 原创：51篇
- 转载：14篇
- 译文：0篇
- 评论：33条

- Linux下的Backlight子系统（一）(3028)
- Linux下的USB总线驱动（一）(2969)
- Determining IP information for eth0...failed 错误解决(2954)

评论排行

- Linux下的串口总线驱动（四）(6)
- Linux下的SPI总线驱动（三）(5)
- VFS: Cannot open root device "nfs" or unknown-block(0,255)错误解决(3)
- Linux下的USB总线驱动（二）(2)
- Linux下的I2C总线驱动(2)
- Linux下的SPI总线驱动（二）(2)
- 随笔————写于毕业前夕(2)
- Linux下的PCI总线驱动(2)
- Determining IP information for eth0...failed 错误解决(1)
- Linux内核中的系统调用(1)

推荐文章

  - * 最老程序员创业开发实训4--IOS平台下MVC架构
  - * Android基础入门教程--6.1 数据存储与访问之--文件存储读写
  - * 聊天界面的制作（三）--表情列表发送功能
  - * Linux下编程--文件与IO（三） 文件共享和fcntl函数
  - * Windows 多进程通信API总结
  - Redis学习总结和相关资料

最新评论

- Determining IP information for eth0...failed 错误解决
  weiqubo: 没有用，哎。
- Linux下的网络设备驱动(一)
  skyxiaoyan1: 简洁清晰
- error: variable '__this_module' has initializer but incomplete type错误解决
  hechengwang321: 这种问题，楼主你是怎么做出来的，谢谢楼主
- VFS: Cannot open root device "nfs" or unknown-block(0,255)错误解决
  helloskeety: IP_PNP这个选项在什么位置
- Linux内核中的系统调用
  yzh07137: 请问我尝试用read函数读取idt的内容，但是errno为1提示没权限，问了别人，说要用内核陷入的方...
- Linux下的SPI总线驱动（三）
  bjq1016: @lijing198997:谢谢 经过一段时间的学习，我明白了spi的底层，原来是设备驱动->平台...
- Linux下的SPI总线驱动（三）
  lijing198997: @wangdaobadao:你好，我的设备spidev0.0出来了，但是在执行调试spidev_te...
- Linux下的SPI总线驱动（三）
  lijing198997: @bjq1016:SPI_IOC_MESSAGE 是全双工的传输，你提的问题spi1到spi2,这个...
- Linux下的SPI总线驱动（三）
  wangdaobadao: 大哥啊，转载文章请实验之后再转行不行，真讨厌你们这种不实验直接转载的，还有文章中的大部分内容都没有交...
- Linux下的SPI总线驱动（二）
  bjq1016: 博主以后把代码单独放在代码块中吧。。。谢谢