七.misc类设备与蜂鸣器驱动（下）

原创

MrT_WANG 于 2019-09-29 16:35:31 发布 625 收藏 1
分类专栏： ARM+Linux探索之旅 ARM(linux驱动开发) 文章标签： 蜂鸣器驱动 misc 驱动
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/wangweijundeqq/article/details/101698971
版权

ARM+Linux探索之旅 同时被 2 个专栏收录

67 篇文章 34 订阅
订阅专栏

ARM(linux驱动开发)

10 篇文章 10 订阅
订阅专栏

接上：https://blog.csdn.net/wangweijundeqq/article/details/101698711

目录

三.misc驱动框架源码分析1

3.1、misc源码框架基础

misc设备初始化函数：

注册接口函数：

3.2、misc类设备的注册

四.misc驱动框架源码分析2

4.1、open函数分析

4.2、misc在proc下的展现

4.3、内核互斥锁

五.蜂鸣器驱动源码分析1

5.1、dev_init

5.2、ioctl

六.蜂鸣器驱动源码分析2

三.misc驱动框架源码分析1

1. 3.1、misc源码框架基础

整体框架分析图：

misc设备初始化函数：

<u>**下列代码位于/drivers/char/misc.c**</u>

```
static struct class *misc_class;

static const struct file_operations misc_fops = {
	.owner		= THIS_MODULE,
	.open		= misc_open,
};

static int __init misc_init(void)
{
	int err;

#ifdef CONFIG_PROC_FS
	proc_create("misc", 0, NULL, &misc_proc_fops);
#endif
	misc_class = class_create(THIS_MODULE, "misc");//创建设备类，一定在某个地方有 device_create
	err = PTR_ERR(misc_class);
	if (IS_ERR(misc_class))
		goto fail_remove;

	err = -EIO;
	if (register_chrdev(MISC_MAJOR,"misc",&misc_fops))
			goto fail_printk;

	misc_class->devnode = misc_devnode;
	return 0;

fail_printk:
	printk("unable to get major %d for misc devices\n", MISC_MAJOR);
	class_destroy(misc_class);
fail_remove:
	remove_proc_entry("misc", NULL);
	return err;
}
subsys_initcall(misc_init);
```

//这是字符设备驱动开发基础的提到过的涉及到的旧的注册接口去注册我们设备内部通过__register_chrdev_region 将 misc对应的char_device_struct 结构体类型的变量的地址挂到内核维护的 char_device_struct 结构体类型的指针数组

从这里我们也可以看出 misc 类设备本质是字符设备

注册了一个设备号为 MISC_MAJOR 的设备 misc,为什么这里要注册一个设备？ 分析开放出来的注册接口 misc_register，其中有一句 dev = MKDEV(MISC_MAJOR, misc->minor);使用了主设备号+次设备号生成一个设备号。也就是说在驱动中 device_create 创建设备文件主设备号都为 MISC_MAJOR，次设备号不同。从而实现分类(分出 misc 类的字符设备)!!!
但是这个 misc 设备和后面的注册的杂散设备本质上是一样的，没有说 misc 是老大，也就是说 misc 这个设备的 fops 并不是缺省

https://blog.csdn.net/wangweijundeqq

注册接口函数：

下列代码在/drivers/char/misc.c

```
static LIST_HEAD(misc_list);          //定义和初始化一个维护链表(可以理解为用于管理入口 entry)
static DEFINE_MUTEX(misc_mtx);        //定义和初始化一个互斥锁
```

下列代码在 include/linux/miscdevice.h

```
struct miscdevice  {
    int minor;  /*次设备号*/
    const char *name;
    const struct file_operations *fops;
    struct list_head list;
            /*将内核链表内嵌到结构体*//
    struct device *parent;
    struct device *this_device;
    const char *nodename;
    mode_t mode;
};
```

将来写misc设备类就是定义一个这样的一个结构体变量然后填充，然后调用 misc_register 进行注册

作用就是类似于 led 框架的 led_classdev 结构体，只不过 led_classdev 没有 fops.因为 led 走的是 atrribute 那条路

```
int misc_register(struct miscdevice * misc)
{
    struct miscdevice *c;
    dev_t dev;
    int err = 0;

    INIT_LIST_HEAD(&misc->list);
                    //节点对应链表入口(entry)初始化
    mutex_lock(&misc_mtx);

    list_for_each_entry(c, &misc_list, list) {
        if (c->minor == misc->minor) {
            mutex_unlock(&misc_mtx);
            return -EBUSY;
        }
    }
```

list_for_each_entry:是一个宏定义展开是一个 for 循环，实现的是去内核维护的 misc_list 链表中取出一个一个 miscdevice 的结构体变量的指针，比较 minor 与传进来的 minor，如果相等就说明这个 minor 被使用了；遍历了一遍之后都没有相等则说明......

在本文件中你是找不到 misc_list 这个变量的，怎么回事呢？在开头的 LIST_HEAD(misc_list); 把它展开后

```
static struct list_head misc_list = {
    &(misc_list), &(misc_list)
}
struct list_head {
    struct list_head *next, *prev;
}
```

也就是说 misc_list 是内核维护的一个 misc 设备类的链表，将来添加一个设备就往这个链表添加相应的 miscdevice 的结构体变量的地址(指针)

```
    if (misc->minor == MISC_DYNAMIC_MINOR) {
        int i = find_first_zero_bit(misc_minors, DYNAMIC_MINORS);
        if (i >= DYNAMIC_MINORS) {
            mutex_unlock(&misc_mtx);
            return -EBUSY;
        }
        misc->minor = DYNAMIC_MINORS - i - 1;
        set_bit(i, misc_minors);
    }
```

当传进来的次设备号为 MISC_DYNAMIC_MINOR 宏定义的那个数字，说明是要让内核帮我们分配次设备号，怎么帮？bitmap

```
    dev = MKDEV(MISC_MAJOR, misc->minor);  /* 使用固定的主设备号 + 次设备号构造设备号 */

    misc->this_device = device_create(misc_class, misc->parent, dev,
                          misc, "%s", misc->name);
    if (IS_ERR(misc->this_device)) {
        int i = DYNAMIC_MINORS - misc->minor - 1;
        if (i < DYNAMIC_MINORS && i >= 0)
            clear_bit(i, misc_minors);
        err = PTR_ERR(misc->this_device);
        goto out;
    }

    /*
     * Add it to the front, so that later devices can "override"
     * earlier defaults
     */
    list_add(&misc->list, &misc_list);
 out:
    mutex_unlock(&misc_mtx);
    return err;
}
```

https://blog.csdn.net/wangweijundeqq

(1)misc源码框架本身也是一个模块，内核启动时自动加载

(2)源码框架的主要工作：注册misc类，使用老接口注册字符设备驱动（主设备号10），开放device注册的接口misc_register给驱动工程师

Kernel_x210 Project - Source Insight - [Misc.c (e:\linux\winshare\x210kernel\kernel\drivers\char)]

```
00271: static int __init misc_init(void)
00272: {
00273:         int err;
00274:
00275: #ifdef CONFIG_PROC_FS    //用于浏览misc里面的设备
00276:         proc_create("misc", 0, NULL, &misc_proc_fops);
00277: #endif
00278:         misc_class = class_create(THIS_MODULE, "misc");//创建misc类
00279:         err = PTR_ERR(misc_class);
00280:         if (IS_ERR(misc_class))
00281:                 goto ↓fail_remove;
00282:
00283:         err = -EIO;
00284:         //下面是字符设备驱动的创建
00285:         if (register_chrdev(MISC_MAJOR,"misc",&misc_fops))//很久之前写的代码，用MISC_MAJOR指定了主设备号，没有次设备号
00286:                 goto ↓fail_printk;
00287:         misc_class->devnode = misc_devnode;
00288:         return 0;
00289:
00290: fail_printk:
00291:         printk("unable to get major %d for misc devices\n", MISC_MAJOR);
00292:         class_destroy(misc_class);
00293: fail_remove:
00294:         remove_proc_entry("misc", NULL);
00295:         return err;
00296: } ? end misc_init ?
00297: subsys_initcall(misc_init);//加载misc驱动框架
```

1. **3.2、misc类设备的注册**

(1)驱动工程师需要借助misc来加载自己的驱动时，只需要调用misc_register接口注册自己的设备即可，其余均不用管。驱动工程师将来写程序时， 需要定义出 miscdevice 这个结构体， 然后对其进行填充， 调用 misc_register 对设备进行注册就可以了

Kernel_x210 Project - Source Insight - [Miscdevice.h (e:\linux\winshare\x210kernel\kernel\include\linux)]

```
00044:
00045: struct miscdevice  {
00046:         int minor;
00047:         const char *name;
00048:         const struct file_operations *fops;
00049:         struct list_head list;
00050:         struct device *parent;
00051:         struct device *this_device;
00052:         const char *nodename;
00053:         mode_t mode;
00054: };
00055:
00056: extern int misc_register(struct miscdevice * misc);
00057: extern int misc_deregister(struct miscdevice *misc);
00058:
00059: #define MODULE_ALIAS_MISCDEV(minor)                          \
00060:         MODULE_ALIAS("char-major-" __stringify(MISC_MAJOR)   \
00061:         "-" __stringify(minor))
00062: #endif
```

https://blog.csdn.net/wangweijundeqq

(2)misc_list链表的作用。内核定义了一个misc_list链表用来记录所有内核中注册了的杂散类设备。当我们向内核注册一个misc类设备时，内核就会向misc_list链表中insert一个节点。

```
00182:
00183: int misc_register(struct miscdevice * misc)
00184: {
00185:     struct miscdevice *c;
00186:     dev_t dev;
00187:     int err = 0;
00188:
00189:     INIT_LIST_HEAD(&misc->list);//misc中的链表初始化，让头指针和尾指针都指向自己
00190:
00191:     mutex_lock(&misc_mtx);
00192:     list_for_each_entry(c, &misc_list, list) {  //遍历注册杂散类设的链表，misc_list这个变量是搜索不到的，在当前文件的最开头
00193:         if (c->minor == misc->minor) {       //判断我们指定的minor次设备号是否已经存在，如果已经存在则出错
00194:             mutex_unlock(&misc_mtx);
00195:             return -EBUSY;
00196:         }
00197:     }
```

misc_list 链表在List.h中

```
00022:
00023: #define LIST_HEAD_INIT(name) { &(name), &(name) }
00024:
00025: #define LIST_HEAD(name) \
00026:     struct list_head name = LIST_HEAD_INIT(name)
00027:
```

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }
#define LIST_HEAD(name) \
 struct list_head name = LIST_HEAD_INIT(name)
```

```
原式子：static LIST_HEAD(misc_list);
```

展开后：static struct list_head misc_list = { &(misc_list), &(misc_list) }

就是定义了一个链表指针， 指针初始化指向了它自己。

此链表用于内核对杂散类设备的管理。

总结：这里我们主要分析的两个函数：misc设备初始化函数和misc设备注册函数：

```
\kernel\drivers\char\misc.c

static int __init misc_init(void)
{
 int err;
#ifdef CONFIG_PROC_FS   //用于浏览misc里面的设备
 proc_create("misc", 0, NULL, &misc_proc_fops);
#endif
 misc_class = class_create(THIS_MODULE, "misc");//创建misc类
 err = PTR_ERR(misc_class);
 if (IS_ERR(misc_class))
  goto fail_remove;

 err = -EIO;
 //下面是字符设备驱动的创建
 if (register_chrdev(MISC_MAJOR,"misc",&misc_fops))//很久之前写的代码，用MISC_MAJOR指定了主设备号，没有次设备号
  goto fail_printk;
 misc_class->devnode = misc_devnode;
 return 0;

fail_printk:
 printk("unable to get major %d for misc devices\n", MISC_MAJOR);
 class_destroy(misc_class);
fail_remove:
 remove_proc_entry("misc", NULL);
 return err;
}
```

```
int misc_register(struct miscdevice * misc)
{
 struct miscdevice *c;
 dev_t dev;
 int err = 0;

 INIT_LIST_HEAD(&misc->list);//misc中的链表初始化，让头指针和尾指针都指向自己

 mutex_lock(&misc_mtx);
 list_for_each_entry(c, &misc_list, list) { //遍历注册杂散类设的链表，misc_list这个变量是搜索不到的，在当前文件的最
  if (c->minor == misc->minor) {  //判断我们指定的minor次设备号是否已经存在，如果已经存在则出错
   mutex_unlock(&misc_mtx);
   return -EBUSY;
  }
 }

 if (misc->minor == MISC_DYNAMIC_MINOR) { //把次设备号设置为255，内核会帮我们自动分配次设备号
  int i = find_first_zero_bit(misc_minors, DYNAMIC_MINORS);//使用位图，一个bit表示一个次设备号
  if (i >= DYNAMIC_MINORS) {       //如果找到的可以分配次设备号已经超限
   mutex_unlock(&misc_mtx);
   return -EBUSY;
  }
  misc->minor = DYNAMIC_MINORS - i - 1;     //分配次设备号
  set_bit(i, misc_minors);        //并且将分配的这1位置1，标志这一位已经使用了
 }

 dev = MKDEV(MISC_MAJOR, misc->minor);     //用主次设备号给他合成一个device的设备号

 misc->this_device = device_create(misc_class, misc->parent, dev,
       misc, "%s", misc->name);     //给 misc类中创建设设备
 if (IS_ERR(misc->this_device)) {
  int i = DYNAMIC_MINORS - misc->minor - 1;
  if (i < DYNAMIC_MINORS && i >= 0)
   clear_bit(i, misc_minors);
  err = PTR_ERR(misc->this_device);      //创建的设备添加到misc链表里面去
  goto out;
 }

 /*
  * Add it to the front, so that later devices can "override"
  * earlier defaults
  */
 list_add(&misc->list, &misc_list);//创建的设备添加到misc链表里面来
 out:
 mutex_unlock(&misc_mtx);
 return err;
}
```

1. (3)主设备号和次设备号的作用和区分

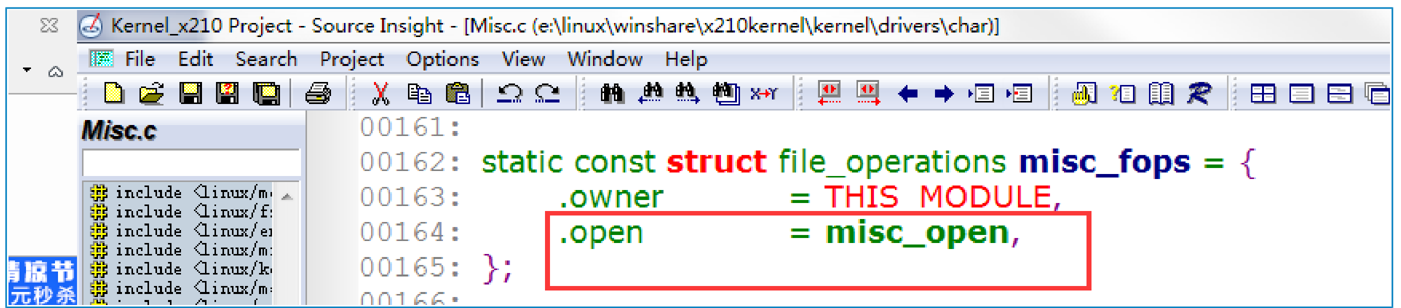
主设备号用来指定一类设备，相当于个类；次设备号用于指定一类设备中的某一个设备

四.misc驱动框架源码分析2

4.1、open函数分析

在 misc_init 初始化函数中调用 register_chrdev 创建 misc 设备驱动的时候， 用到了

misc_fops 结构， 此结构中包含 open 函数：



此结构中只实现了.open 函数， 我们写驱动程序的源代码的时候就需要编写这样的 open 函数等（ 如果原厂没有给我们实现） 。

用户是通过 misc_open 函数， 最后在这里函数里面映射到了驱动中真正提供的 open 函

数， 最后调用了驱动中提供的 open 函数， 大概思路如下：

```
kernel\drivers\char\misc.c
//inode 表示设备节点、 路径； file 表示对应的文件
static int misc_open(struct inode * inode, struct file * file)
{
 ......
 //(1)遍历杂散类的设备链表， 找次设备号
 list_for_each_entry(c, &misc_list, list) {
  if (c->minor == minor) {
   new_fops = fops_get(c->fops); //读出来我们查到的 fops
   break;
  }
 }

 //(2)如果上面没找到次设备号， request_module 后再找一遍
 if (!new_fops) {
  .....
  list_for_each_entry(c, &misc_list, list) {
   ....
  }
  .....

 //(3)如果找到了， 则调用 open 驱动中的函数
 err = 0;
 old_fops = file->f_op;
 file->f_op = new_fops;
 if (file->f_op->open) {
  file->private_data = c;

  //这才是真正的 open,真正写驱动的人实现的。
  //注意： f_op 表示 file_operations -> 中的 open 等函数,
  //它就是写驱动的用户实现的
 err=file->f_op->open(inode,file);
    if (err) {
   fops_put(file->f_op);
   file->f_op = fops_get(old_fops);
  }
 }
 .....
 }
```

在我们蜂鸣器驱动程序：

kernel\drivers\char\buzzer\x210-buzzer.c文件中

```
static struct file_operations dev_fops = {
    .owner   =   THIS_MODULE,
    .open    =   x210_pwm_open,
    .release =   x210_pwm_close,
    .ioctl   =   x210_pwm_ioctl,
};
```

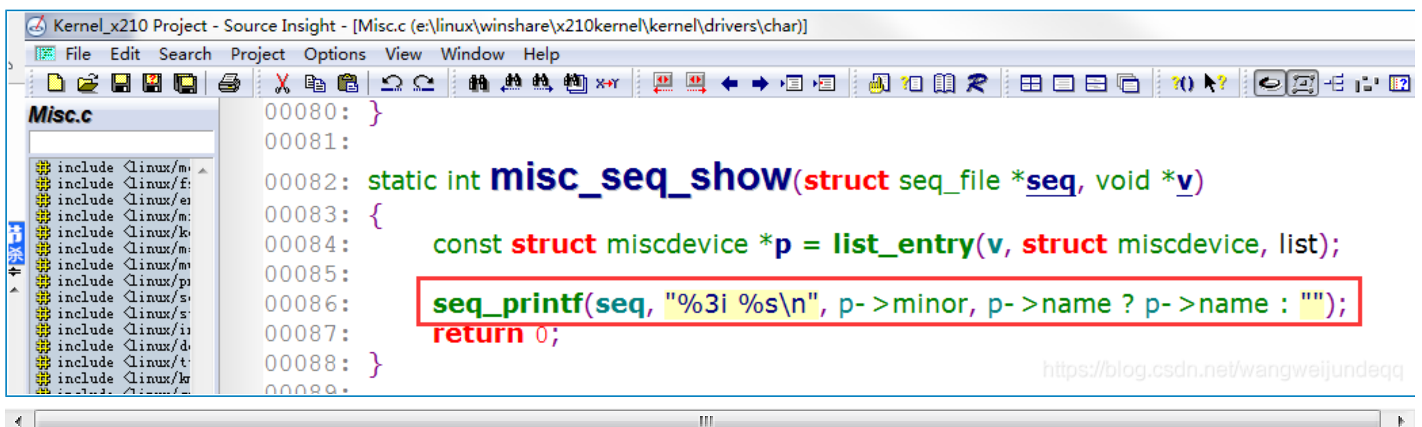
这个结构中的实例是有驱动工程师定义并填充的。

这样就能够对上号： misc 框架中的 misc_open 函数最后调用了驱动中的 open 函数。

4.2、misc在proc下的展现

注意上面的这种对齐方式：

上图这是系统里面已经注册的杂散类设备， 那么这是怎么出现的呢？ 就是通过链表来遍历的。



seq_printf(seq, "%3i %s\n", p->minor, p->name ? p->name : ""); //minor 左对齐， name 右对齐打印

4.3、内核互斥锁

(1)何为互斥锁

(2)定义：DEFINE_MUTEX

(3)上锁mutex_lock和解锁mutex_unlock

(4)内核防止竞争状态的手段：原子访问、自旋锁、互斥锁、信号量

(5)原子访问主要用来做计数、自旋锁后面讲中断会详细讲、互斥锁和信号量很相似（其实就是计数值为1的信号量），互斥锁的出现比信号量晚，实现上比信号量优秀，尽量使用互斥锁。

五.蜂鸣器驱动源码分析1

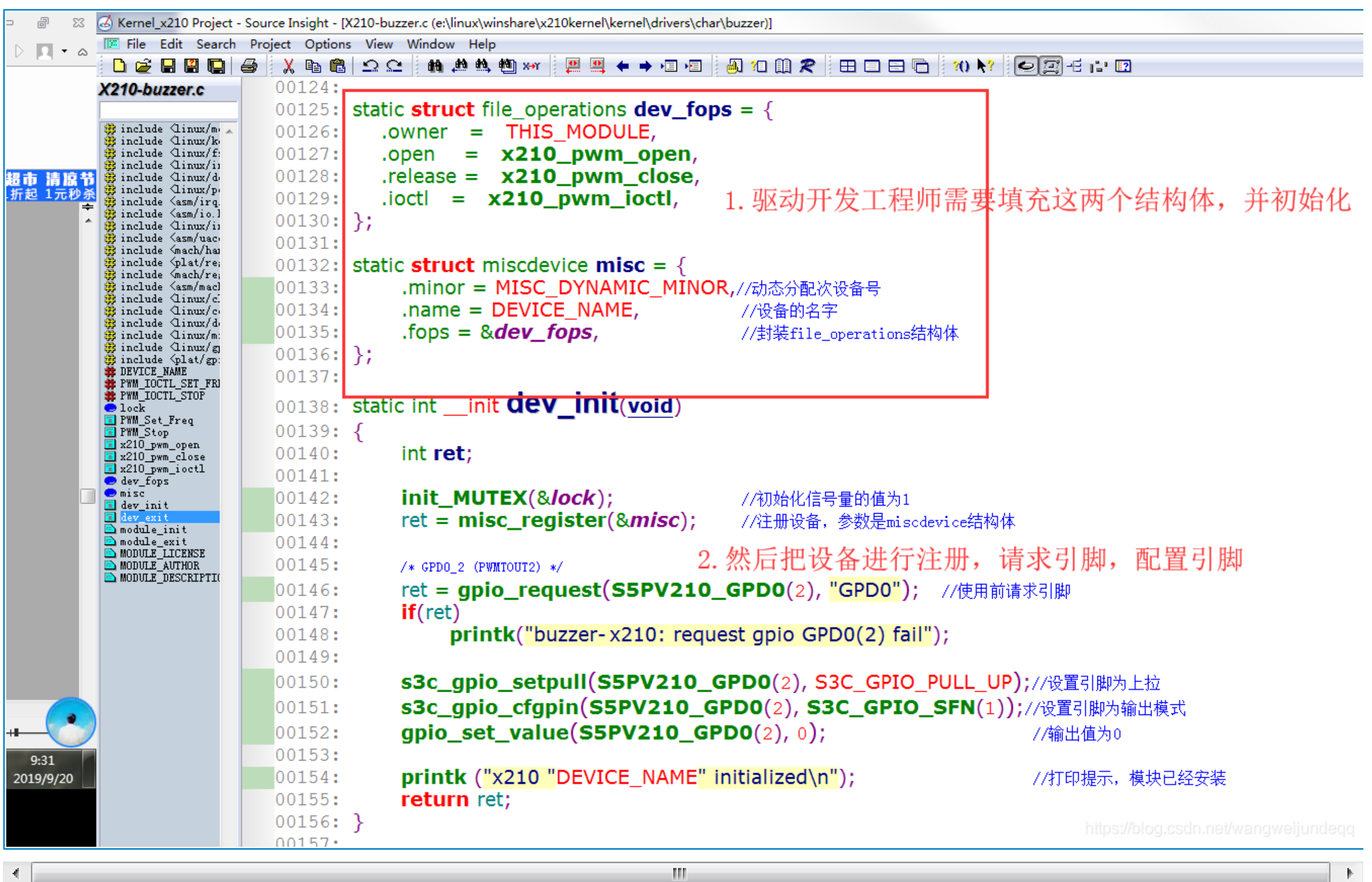
5.1、dev_init

(1)信号量

(2)miscdevice

(3)gpio_request

(4)printk



完成了上面的初始化， 表示引脚可以工作了， 但需要应用程序来具体的控制。

5.2、ioctl

(1)为什么需要ioctl（input output control，输入输出控制）。

Kernel_x210 Project - Source Insight - [X210-buzzer.c (e:\linux\winshare\x210kernel\kernel\drivers\char\buzzer) *]

```
00101:
00102: // PWM:GPF14->PWM0
00103: static int x210_pwm_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
00104: {
00105:     switch (cmd)
00106:     {
00107:         case PWM_IOCTL_SET_FREQ:
00108:             printk("PWM_IOCTL_SET_FREQ:\r\n");
00109:             if (arg == 0)
00110:                 return -EINVAL;
00111:             PWM_Set_Freq(arg);              //设置蜂鸣器的频率
00112:             break;
00113:
00114:         case PWM_IOCTL_STOP:
00115:         default:
00116:             printk("PWM_IOCTL_STOP:\r\n");
00117:             PWM_Stop();                     //关闭蜂鸣器，设置为关闭模式即可
00118:             break;
00119:     }
00120:
00121:     return 0;
00122: } ? end x210_pwm_ioctl ?
```

(2)ioctl怎么用

我们只需要在应用层中使用ioctl函数即可，应用层和驱动层会自动关联，具体操作是怎么实现的，无须关心，只需要知道填充了dev_fops结构体，和驱动层的x210_pwm_ioctl函数关联了即可。

下面需要关心的就是在x210_pwm_ioctl函数里面和硬件操作有关的两个函数了

PWM_Set_Freq函数

PWM_Stop函数

六.蜂鸣器驱动源码分析2

硬件操作有关的代码

Kernel_x210 Project - Source Insight - [X210-buzzer.c (e:\linux\winshare\x210kernel\kernel\drivers\char\buzzer)]

```
00079:
00080: void PWM_Stop( void )
00081: {
00082:     //将GPD0_2设置为input，即关闭
00083:     s3c_gpio_cfgpin(S5PV210_GPD0(2), S3C_GPIO_SFN(0));
00084: }
00085:
```

210-buzzer.c

```
00040: // TCFG0 = tcnt = (pclk/16/16)/freq;
00041: // PWM0输出频率Foutput =Finput/TCFG0= freq
00042: static void PWM_Set_Freq( unsigned long freq )
00043: {
00044:     unsigned long tcon;
00045:     unsigned long tcnt;
00046:     unsigned long tcfg1;
00047:
00048:     struct clk *clk_p;
00049:     unsigned long pclk;
00050:
00051:     //unsigned tmp;
00052:
00053:     //设置GPD0_2为PWM输出
00054:     s3c_gpio_cfgpin(S5PV210_GPD0(2), S3C_GPIO_SFN(2));
00055:
00056:     tcon = __raw_readl(S3C2410_TCON);          //读取寄存器
00057:     tcfg1 = __raw_readl(S3C2410_TCFG1);
00058:
00059:     //mux = 1/16
00060:     tcfg1 &= ~(0xf<<8);                        //这里需要看数据手册
00061:     tcfg1 |= (0x4<<8);
00062:     __raw_writel(tcfg1, S3C2410_TCFG1);        //写进去
00063:
00064:     clk_p = clk_get(NULL, "pclk");             //获取系统时间
00065:     pclk  = clk_get_rate(clk_p);
00066:
00067:     tcnt  = (pclk/16/16)/freq;                 //往定时器里面写的值
00068:
00069:     __raw_writel(tcnt, S3C2410_TCNTB(2));
00070:     __raw_writel(tcnt/2, S3C2410_TCMPB(2));//占空比为50%
00071:
00072:     tcon &= ~(0xf<<12);
00073:     tcon |= (0xb<<12);          //disable deadzone, auto-reload, inv-off, update TCNTB0&TCMPB0, start timer 0
00074:     __raw_writel(tcon, S3C2410_TCON);
```

https://blog.csdn.net/wangweijundeqq