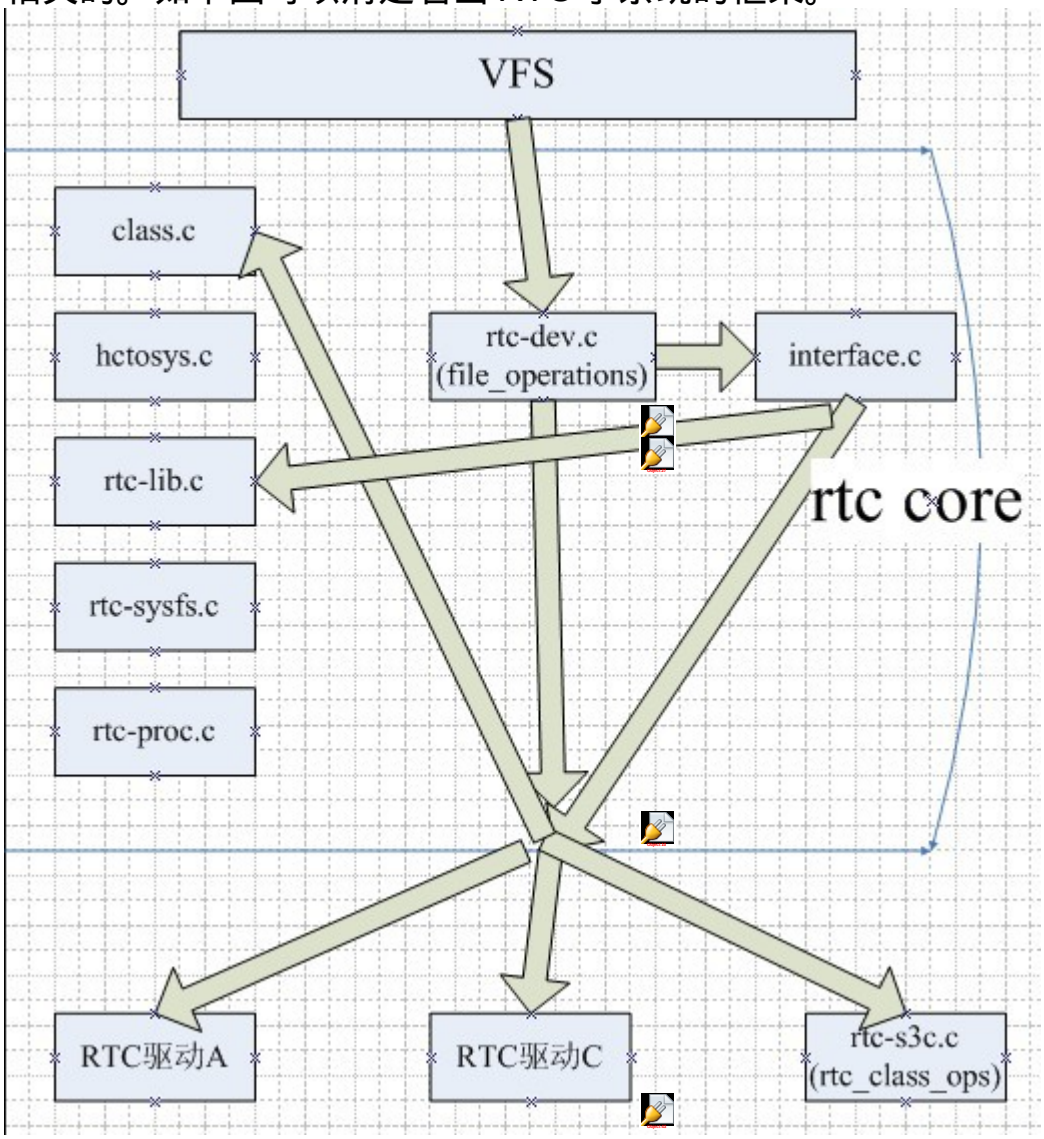
努力成为 linux kernel hacker 的人李万鹏原创作品，为梦而战。转载请标明出处
http://blog.csdn.net/woshixingaaa/archive/2011/05/21/6436215.aspx
RTC(实时时钟)是一种典型的字符设备，作为一种字符设备驱动，RTC 需要有 file_operations 中接口函数的实现，如 open(),release(),read(),poll(),ioctl()等，而典型的 ioctl 包括 RTC_SET_TIME,RTC_ALM_READ,RTC_ALM_SET,RTC_IRQP_SET,RTC_IRQP_READ 等，这些对于所有的 RTC 是通用的，只有底层的具体实现是设备相关的。如下图可以清楚看出 RTC 子系统的框架。



下面介绍几个重要的数据结构：
rtc_device 用来描述 rtc 设备：

```
1. struct rtc_device
2. {
3.     struct device dev;
4.     struct module *owner;
5.     int id;                          //RTC 设备的次设备号
```

```
    char name[RTC_DEVICE_NAME_SIZE];
    const struct rtc_class_ops *ops;
    struct mutex ops_lock;
    struct cdev char_dev;
    unsigned long flags;
    unsigned long irq_data;
    spinlock_t irq_lock;
    wait_queue_head_t irq_queue;
    struct fasync_struct *async_queue;
    struct rtc_task *irq_task;
    spinlock_t irq_task_lock;
    int irq_freq;
    int max_user_freq;
#ifdef CONFIG_RTC_INTF_DEV_UIE_EMUL
    struct work_struct uie_task;
    struct timer_list uie_timer;
    /* Those fields are protected by rtc->irq_lock */
    unsigned int oldsecs;
    unsigned int uie_irq_active:1;
    unsigned int stop_uie_polling:1;
    unsigned int uie_task_active:1;
    unsigned int uie_timer_active:1;
#endif
};
```

rtc_time 用于 get time/set time：

```
struct rtc_time {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

描述报警状态的结构：

```
struct rtc_wkalrm {
    unsigned char enabled;  /* 0 = alarm disabled, 1 = alarm enabled */
    unsigned char pending;  /* 0 = alarm not pending, 1 = alarm pending */
    struct rtc_time time;   /* time the alarm is set to */
};
```

```
struct rtc_class_ops {
    int (*open)(struct device *);     //打开设备时的回调函数，这个函数应该初始化硬件并申请资源
```

```
    void (*release)(struct device *); //这个函数是设备关闭时被调用的，应该注销申请的资源
    int (*ioctl)(struct device *, unsigned int, unsigned long); //ioctl 函数，对想让 RTC 自己实现的命令应返回 ENOIOCTLCMD
    int (*read_time)(struct device *, struct rtc_time *);       //读取时间
    int (*set_time)(struct device *, struct rtc_time *);        //设置时间
    int (*read_alarm)(struct device *, struct rtc_wkalrm *);    //读取下一次定时中断的时间
    int (*set_alarm)(struct device *, struct rtc_wkalrm *);     //设置下一次定时中断的时间
    int (*proc)(struct device *, struct seq_file *);            //procfs 接口
    int (*set_mmss)(struct device *, unsigned long secs);       //将传入的参数 secs 转换为 struct rtc_time 然后调用 set_time 函数。程序员可以不实现这个函数，但
前提是定义好了 read_time/set_time，因为 RTC 框架需要用这两个函数来实现这个功能。
    int (*irq_set_state)(struct device *, int enabled);         //周期采样中断的开关，根据 enabled 的值来设置
    int (*irq_set_freq)(struct device *, int freq);             //设置周期中断的频率
    int (*read_callback)(struct device *, int data);            ///用户空间获得数据后会传入读取的数据，并用这个函数返回的数据更新数据。
    int (*alarm_irq_enable)(struct device *, unsigned int enabled);  //alarm 中断使能开关，根据 enabled 的值来设置
    int (*update_irq_enable)(struct device *, unsigned int enabled); //更新中断使能开关，根据 enabled 的值来设置
};
```

现在来看看 rtc 子系统是怎么注册上的：

```
static int __init rtc_init(void)
{
    rtc_class = class_create(THIS_MODULE, "rtc");
    if (IS_ERR(rtc_class)) {
        printk(KERN_ERR "%s: couldn't create class/n", __FILE__);
        return PTR_ERR(rtc_class);
    }
    rtc_class->suspend = rtc_suspend;
    rtc_class->resume = rtc_resume;
    rtc_dev_init();
    rtc_sysfs_init(rtc_class);
    return 0;
}
void __init rtc_dev_init(void)
{
    int err;
    err = alloc_chrdev_region(&rtc_devt, 0, RTC_DEV_MAX, "rtc");
    if (err < 0)
        printk(KERN_ERR "%s: failed to allocate char dev region/n",
            __FILE__);
}
```

在 class.c 文件函数 rtc_init 中生成 rtc 类，然后调用 rtc-dev.c 文件中的 rtc_dev_init 分配设备号。
在 rtc-dev.c 中声明了 file_operations,因为 rtc 也是一个字符设备:

```
static const struct file_operations rtc_dev_fops = {
    .owner      = THIS_MODULE,
    .llseek     = no_llseek,
    .read       = rtc_dev_read,
    .poll       = rtc_dev_poll,
    .unlocked_ioctl = rtc_dev_ioctl,
    .open       = rtc_dev_open,
    .release    = rtc_dev_release,
    .fasync     = rtc_dev_fasync,
};
```

下面来分析 rtc-s3c.c 源码：
首先看模块的注册和撤销：

```
static int __init s3c_rtc_init(void)
{
    printk(banner);
    return platform_driver_register(&s3c2410_rtc_driver);
}
static void __exit s3c_rtc_exit(void)
{
    platform_driver_unregister(&s3c2410_rtc_driver);
}
```

从上边的代码可以看出 rtc driver 作为 platform_driver 注册进内核，挂在 platform_bus 上。

```
static struct platform_driver s3c2410_rtc_driver = {
    .probe      = s3c_rtc_probe,          //rtc 探测函数
    .remove     = __devexit_p(s3c_rtc_remove),  //rtc 移除函数
    .suspend    = s3c_rtc_suspend,        //rtc 挂起函数
    .resume     = s3c_rtc_resume,         //rtc 恢复函数
    .driver     = {
        .name   = "s3c2410-rtc",          //注意这里的名字一定要和系统中定义平台设备的地方一致，这样才能把平台设备和平台驱动关联起来
        .owner  = THIS_MODULE,
    },
};
```

在 arch/arm/plat-s3c24xx/devs.c 中定义了 rtc 的 platform_device:

```
/* RTC */
static struct resource s3c_rtc_resource[] = {          //定义了 rtc 平台设备会使用的资源
    [0] = {                                 //IO 端口资源范围
        .start = S3C24XX_PA_RTC,
        .end   = S3C24XX_PA_RTC + 0xff,
```

```
        .flags = IORESOURCE_MEM,
    },
    [1] = {                                 //RTC 报警中断资源
        .start = IRQ_RTC,
        .end   = IRQ_RTC,
        .flags = IORESOURCE_IRQ,
    },
    [2] = {                                 //TICK 节拍时间中断资源
        .start = IRQ_TICK,
        .end   = IRQ_TICK,
        .flags = IORESOURCE_IRQ
    }
};
struct platform_device s3c_device_rtc = {               //定义了平台设备
    .name         = "s3c2410-rtc",              //设备名
    .id       = -1,
    .num_resources    = ARRAY_SIZE(s3c_rtc_resource),   //资源数量
    .resource     = s3c_rtc_resource,             //引用上面定义的资源
};
```

平台驱动中定义了 probe 函数，下面来看他的实现：

```
static int __devinit s3c_rtc_probe(struct platform_device *pdev)
{
    struct rtc_device *rtc;
    struct resource *res;
    int ret;
    pr_debug("%s: probe=%p/n", __func__, pdev);
    /* find the IRQs */
    /*获得 IRQ 资源中的第二个，即 TICK 节拍时间中断号*/
    s3c_rtc_tickno = platform_get_irq(pdev, 1);
    if (s3c_rtc_tickno < 0) {
        dev_err(&pdev->dev, "no irq for rtc tick/n");
        return -ENOENT;
    }
    /*获取 IRQ 资源中的第一个，即 RTC 报警中断*/
    s3c_rtc_alarmno = platform_get_irq(pdev, 0);
    if (s3c_rtc_alarmno < 0) {
        dev_err(&pdev->dev, "no irq for alarm/n");
        return -ENOENT;
    }
    pr_debug("s3c2410_rtc: tick irq %d, alarm irq %d/n",
         s3c_rtc_tickno, s3c_rtc_alarmno);
    /* get the memory region */
    /*获取 RTC 平台设备所使用的 IO 端口资源*/
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (res == NULL) {
```

```
        dev_err(&pdev->dev, "failed to get memory region resource/n");
        return -ENOENT;
    }
    /*申请 IO 端口资源所占用的 IO 空间*/
    s3c_rtc_mem = request_mem_region(res->start,
                    res->end-res->start+1,
                    pdev->name);
    if (s3c_rtc_mem == NULL) {
        dev_err(&pdev->dev, "failed to reserve memory region/n");
        ret = -ENOENT;
        goto err_nores;
    }
    /*将 IO 端口占用的 IO 空间映射到虚拟地址，s3c_rtc_base 是这段虚拟地址的起始地址*/
    s3c_rtc_base = ioremap(res->start, res->end - res->start + 1);
    if (s3c_rtc_base == NULL) {
        dev_err(&pdev->dev, "failed ioremap()/n");
        ret = -EINVAL;
        goto err_nomap;
    }
    /* check to see if everything is setup correctly */
    /*对 RTCCON 第 0 位进行操作，使能 RTC*/
    s3c_rtc_enable(pdev, 1);
    pr_debug("s3c2410_rtc: RTCCON=%02x/n",
        readb(s3c_rtc_base + S3C2410_RTCCON));
    /*对 TICNT 第 7 位进行操作，使能节拍时间计数寄存器*/
    s3c_rtc_setfreq(&pdev->dev, 1);
    /*让电源管理支持唤醒功能*/
    device_init_wakeup(&pdev->dev, 1);
    /* register RTC and exit */
    /*注册 rtc 设备，名为"s3c",与 s3c_rtcops 这个 rtc_class_ops 进行关联*/
    rtc = rtc_device_register("s3c", &pdev->dev, &s3c_rtcops,
                THIS_MODULE);
    if (IS_ERR(rtc)) {
        dev_err(&pdev->dev, "cannot attach rtc/n");
        ret = PTR_ERR(rtc);
        goto err_nortc;
    }
    /**/
    rtc->max_user_freq = 128;
    /*将 rtc 这个 rtc_device 存放在&pdev->dev->driver_data*/
    platform_set_drvdata(pdev, rtc);
    return 0;
err_nortc:
    s3c_rtc_enable(pdev, 0);
    iounmap(s3c_rtc_base);
```

```
71. err_nomap:
72.     release_resource(s3c_rtc_mem);
73. err_nores:
74.     return ret;
75. }
```

函数 rtc_device_register 在文件 class.c 中实现:

```
1.  struct rtc_device *rtc_device_register(const char *name, struct device *dev,
2.                  const struct rtc_class_ops *ops,
3.                  struct module *owner)
4.  {
5.      struct rtc_device *rtc;
6.      int id, err;
7.      /*为 idr(rtc_idr)分配内存*/
8.      if (idr_pre_get(&rtc_idr, GFP_KERNEL) == 0) {
9.          err = -ENOMEM;
10.         goto exit;
11.     }
12.     mutex_lock(&idr_lock);
13.     /*分配 ID 号存于 id 中，该 ID 号最终将作为该 RTC 设备的次设备号*/
14.     err = idr_get_new(&rtc_idr, NULL, &id);
15.     mutex_unlock(&idr_lock);
16.     if (err < 0)
17.         goto exit;
18.     id = id & MAX_ID_MASK;
19.     /*为 RTC 结构分配内存*/
20.     rtc = kzalloc(sizeof(struct rtc_device), GFP_KERNEL);
21.     if (rtc == NULL) {
22.         err = -ENOMEM;
23.         goto exit_idr;
24.     }
25.     rtc->id = id;
26.     /*指向原始操作函数集*/
27.     rtc->ops = ops;
28.     rtc->owner = owner;
29.     rtc->max_user_freq = 64;
30.     rtc->dev.parent = dev;
31.     rtc->dev.class = rtc_class;
32.     rtc->dev.release = rtc_device_release;
33.     mutex_init(&rtc->ops_lock);
34.     spin_lock_init(&rtc->irq_lock);
35.     spin_lock_init(&rtc->irq_task_lock);
36.     init_waitqueue_head(&rtc->irq_queue);
37.     strlcpy(rtc->name, name, RTC_DEVICE_NAME_SIZE);
38.     dev_set_name(&rtc->dev, "rtc%d", id);
```

```
    /*rtc->dev.devt = MKDEV(MAJOR(rtc_devt),rtc->id); cdev_init(&rtc->char_dev,&rtc_dev_fops);其中 rtc_devt 是从调用 alloc_chrdev_region 时获得的*/
    rtc_dev_prepare(rtc);
    /*注册该 RTC 设备 rtc->dev*/
    err = device_register(&rtc->dev);
    if (err)
        goto exit_kfree;
    /*cdev_add(&rtc->chr_dev,rtc->dev.devt,1);将 rtc->chrdev 注册到系统中*/
    rtc_dev_add_device(rtc);
    /*在/sys 下添加属性文件*/
    rtc_sysfs_add_device(rtc);
    /*在/proc 中创建入口项"driver/rtc"*/
    rtc_proc_add_device(rtc);
    dev_info(dev, "rtc core: registered %s as %s/n",
            rtc->name, dev_name(&rtc->dev));
    return rtc;
exit_kfree:
    kfree(rtc);
exit_idr:
    mutex_lock(&idr_lock);
    idr_remove(&rtc_idr, id);
    mutex_unlock(&idr_lock);
exit:
    dev_err(dev, "rtc core: unable to register %s, err = %d/n",
            name, err);
    return ERR_PTR(err);
}
```

下边是 s3c_rtc_enable 函数的实现：

```
static void s3c_rtc_enable(struct platform_device *pdev, int en)
{
    void __iomem *base = s3c_rtc_base;
    unsigned int tmp;
    if (s3c_rtc_base == NULL)
        return;
    /*如果禁止，就 disable RTCCON 与 TICNT*/
    if (!en) {
        tmp = readb(base + S3C2410_RTCCON);
        writeb(tmp & ~S3C2410_RTCCON_RTCEN, base + S3C2410_RTCCON);
        tmp = readb(base + S3C2410_TICNT);
        writeb(tmp & ~S3C2410_TICNT_ENABLE, base + S3C2410_TICNT);
    } else {
        /* re-enable the device, and check it is ok */
        /*如果 RTCCON 没有使能，则使能之*/
        if ((readb(base+S3C2410_RTCCON) & S3C2410_RTCCON_RTCEN) == 0){
            dev_info(&pdev->dev, "rtc disabled, re-enabling/n");
```

```
18.
19.         tmp = readb(base + S3C2410_RTCCON);
20.         writeb(tmp|S3C2410_RTCCON_RTCEN, base+S3C2410_RTCCON);
21.     }
22.     /*如果 BCD 的计数选择位为 1，则置位 0，即 Merge BCD counts*/
23.     if ((readb(base + S3C2410_RTCCON) & S3C2410_RTCCON_CNTSEL)){
24.         dev_info(&pdev->dev, "removing RTCCON_CNTSEL/n");
25.         tmp = readb(base + S3C2410_RTCCON);
26.         writeb(tmp& ~S3C2410_RTCCON_CNTSEL, base+S3C2410_RTCCON);
27.     }
28.     /*如果 BCD 的时钟选择为 1，则置位 0，即 XTAL 1/215 divided clock*/
29.     if ((readb(base + S3C2410_RTCCON) & S3C2410_RTCCON_CLKRST)){
30.         dev_info(&pdev->dev, "removing RTCCON_CLKRST/n");
31.         tmp = readb(base + S3C2410_RTCCON);
32.         writeb(tmp & ~S3C2410_RTCCON_CLKRST, base+S3C2410_RTCCON);
33.     }
34.   }
35.}
36.static int __devexit s3c_rtc_remove(struct platform_device *dev)
37.{
38.   /*从系统平台设备中获取 RTC 设备类的数据*/
39.   struct rtc_device *rtc = platform_get_drvdata(dev);
40.    /*清空平台设备中 RTC 驱动数据*/
41.   platform_set_drvdata(dev, NULL);
42.   /*注销 RTC 设备类*/
43.   rtc_device_unregister(rtc);
44.   /*禁止 RTC 节拍时间计数寄存器 TICNT 的使能功能*/
45.   s3c_rtc_setpie(&dev->dev, 0);
46.   /*禁止 RTC 报警控制寄存器 RTCALM 的全局报警使能功能*/
47.   s3c_rtc_setaie(0);
48.   /*释放 RTC 虚拟地址映射空间*/
49.   iounmap(s3c_rtc_base);
50.   /*释放获取的 RTC 平台设备的资源*/
51.   release_resource(s3c_rtc_mem);
52.   /*销毁保存 RTC 平台设备的资源内存空间*/
53.   kfree(s3c_rtc_mem);
54.   return 0;
55.}
```

这里是电源管理部分，在挂起时保存 TICNT 的值，并禁止 RTCCON，TICNT；在休眠的时候开启 RTCCON，并恢复 TICNT 的值。

```
1. #ifdef CONFIG_PM
2. /* RTC Power management control */
3. static int ticnt_save;
4. static int s3c_rtc_suspend(struct platform_device *pdev, pm_message_t state)
5. {
```

```
    /* save TICNT for anyone using periodic interrupts */
    ticnt_save = readb(s3c_rtc_base + S3C2410_TICNT);
    s3c_rtc_enable(pdev, 0);
    return 0;
}
static int s3c_rtc_resume(struct platform_device *pdev)
{
    s3c_rtc_enable(pdev, 1);
    writeb(ticnt_save, s3c_rtc_base + S3C2410_TICNT);
    return 0;
}
#else
#define s3c_rtc_suspend NULL
#define s3c_rtc_resume  NULL
#endif
```

s3c_rtcops 是 RTC 设备在 RTC 核心部分注册的对 RTC 设备进行操作的结构体，类似字符设备在驱动中的 file_operations 对字符设备进行操作的意思。

```
static const struct rtc_class_ops s3c_rtcops = {
    .open       = s3c_rtc_open,
    .release    = s3c_rtc_release,
    .read_time  = s3c_rtc_gettime,
    .set_time   = s3c_rtc_settime,
    .read_alarm = s3c_rtc_getalarm,
    .set_alarm  = s3c_rtc_setalarm,
    .irq_set_freq   = s3c_rtc_setfreq,
    .irq_set_state  = s3c_rtc_setpie,
    .proc           = s3c_rtc_proc,
};
```

这两个是下边会用到的中断处理函数，产生一个时钟中断的时候就更新一下 rtc_irq_data 的值，也就是说只有当产生一个时钟中断(也就是一个滴答 tick)才返回给用户一个时间。

```
static irqreturn_t s3c_rtc_alarmirq(int irq, void *id)
{
    struct rtc_device *rdev = id;
    rtc_update_irq(rdev, 1, RTC_AF | RTC_IRQF);
    return IRQ_HANDLED;
}
static irqreturn_t s3c_rtc_tickirq(int irq, void *id)
{
    struct rtc_device *rdev = id;
    rtc_update_irq(rdev, 1, RTC_PF | RTC_IRQF);
    return IRQ_HANDLED;
}
```

首先来看打开和关闭函数：

```
static int s3c_rtc_open(struct device *dev)
```

```
{
    /*获得平台设备，从平台设备 pdev->dev->driver_data 获取 rtc_device*/
    struct platform_device *pdev = to_platform_device(dev);
    struct rtc_device *rtc_dev = platform_get_drvdata(pdev);
    int ret;
    /*注册 RTC 报警中断的中断处理函数*/
    ret = request_irq(s3c_rtc_alarmno, s3c_rtc_alarmirq,
            IRQF_DISABLED,  "s3c2410-rtc alarm", rtc_dev);
    if (ret) {
        dev_err(dev, "IRQ%d error %d/n", s3c_rtc_alarmno, ret);
        return ret;
    }
    /*注册 TICK 节拍时间中断的中断处理函数*/
    ret = request_irq(s3c_rtc_tickno, s3c_rtc_tickirq,
            IRQF_DISABLED,  "s3c2410-rtc tick", rtc_dev);
    if (ret) {
        dev_err(dev, "IRQ%d error %d/n", s3c_rtc_tickno, ret);
        goto tick_err;
    }
    return ret;
tick_err:
    free_irq(s3c_rtc_alarmno, rtc_dev);
    return ret;
}
```

RTC 设备类关闭接口函数：

```
static void s3c_rtc_release(struct device *dev)
{
    struct platform_device *pdev = to_platform_device(dev);
    struct rtc_device *rtc_dev = platform_get_drvdata(pdev);
    /* do not clear AIE here, it may be needed for wake */
    s3c_rtc_setpie(dev, 0);
    free_irq(s3c_rtc_alarmno, rtc_dev);
    free_irq(s3c_rtc_tickno, rtc_dev);
}
```

更新 RTCALM 寄存器的状态，是否使能：

```
static void s3c_rtc_setaie(int to)
{
    unsigned int tmp;
    pr_debug("%s: aie=%d/n", __func__, to);
    tmp = readb(s3c_rtc_base + S3C2410_RTCALM) & ~S3C2410_RTCALM_ALMEN;
    if (to)
        tmp |= S3C2410_RTCALM_ALMEN;
    writeb(tmp, s3c_rtc_base + S3C2410_RTCALM);
}
```

更新 TICNT 寄存器的状态，是否使能：

```
1. static int s3c_rtc_setpie(struct device *dev, int enabled)
2. {
3.     unsigned int tmp;
4.     pr_debug("%s: pie=%d/n", __func__, enabled);
5.     spin_lock_irq(&s3c_rtc_pie_lock);
6.     tmp = readb(s3c_rtc_base + S3C2410_TICNT) & ~S3C2410_TICNT_ENABLE;
7.     if (enabled)
8.         tmp |= S3C2410_TICNT_ENABLE;
9.     writeb(tmp, s3c_rtc_base + S3C2410_TICNT);
10.    spin_unlock_irq(&s3c_rtc_pie_lock);
11.    return 0;
12.}
```

更新 TICNT 节拍时间计数的值：

```
1. static int s3c_rtc_setfreq(struct device *dev, int freq)
2. {
3.     unsigned int tmp;
4.     if (!is_power_of_2(freq))
5.         return -EINVAL;
6.     spin_lock_irq(&s3c_rtc_pie_lock);
7.     tmp = readb(s3c_rtc_base + S3C2410_TICNT) & S3C2410_TICNT_ENABLE;
8.     tmp |= (128 / freq)-1;
9.     writeb(tmp, s3c_rtc_base + S3C2410_TICNT);
10.    spin_unlock_irq(&s3c_rtc_pie_lock);
11.    return 0;
12.}
13./* Time read/write */
14.static int s3c_rtc_gettime(struct device *dev, struct rtc_time *rtc_tm)
15.{
16.    unsigned int have_retried = 0;
17.    /*获得 rtc IO 端口寄存器的虚拟地址的起始地址*/
18.    void __iomem *base = s3c_rtc_base;
19. retry_get_time:
20.    /*读取 RTC 中 BCD 数中的：分、时、日期、月、年、秒，放到 rtc_time rtc_tm 中*/
21.    rtc_tm->tm_min  = readb(base + S3C2410_RTCMIN);
22.    rtc_tm->tm_hour = readb(base + S3C2410_RTCHOUR);
23.    rtc_tm->tm_mday = readb(base + S3C2410_RTCDATE);
24.    rtc_tm->tm_mon  = readb(base + S3C2410_RTCMON);
25.    rtc_tm->tm_year = readb(base + S3C2410_RTCYEAR);
26.    rtc_tm->tm_sec  = readb(base + S3C2410_RTCSEC);
27.    /* the only way to work out wether the system was mid-update
28.     * when we read it is to check the second counter, and if it
29.     * is zero, then we re-try the entire read
30.     */
31.    /*如果到达 0 秒就检查一下，因为年月日时分可能会有加 1 操作，比如此时是一年的最后天的最后一分一秒，
```

```
    则年月日时分秒都会改变*/
32.  if (rtc_tm->tm_sec == 0 && !have_retried) {
33.      have_retried = 1;
34.      goto retry_get_time;
35.  }
36.  pr_debug("read time %02x.%02x.%02x %02x/%02x/%02x/n",
37.       rtc_tm->tm_year, rtc_tm->tm_mon, rtc_tm->tm_mday,
38.       rtc_tm->tm_hour, rtc_tm->tm_min, rtc_tm->tm_sec);
39.  /*使用 readb 读取寄存器的值得到的是 bcd 格式，必须转换成 bin 格式再保存*/
40.  rtc_tm->tm_sec = bcd2bin(rtc_tm->tm_sec);
41.  rtc_tm->tm_min = bcd2bin(rtc_tm->tm_min);
42.  rtc_tm->tm_hour = bcd2bin(rtc_tm->tm_hour);
43.  rtc_tm->tm_mday = bcd2bin(rtc_tm->tm_mday);
44.  rtc_tm->tm_mon = bcd2bin(rtc_tm->tm_mon);
45.  rtc_tm->tm_year = bcd2bin(rtc_tm->tm_year);
46.  rtc_tm->tm_year += 100;
47.  rtc_tm->tm_mon -= 1;
48.  return 0;
49.}
50.static int s3c_rtc_settime(struct device *dev, struct rtc_time *tm)
51.{
52.  void __iomem *base = s3c_rtc_base;
53.  int year = tm->tm_year - 100;
54.  pr_debug("set time %02d.%02d.%02d %02d/%02d/%02d/n",
55.       tm->tm_year, tm->tm_mon, tm->tm_mday,
56.       tm->tm_hour, tm->tm_min, tm->tm_sec);
57.  /* we get around y2k by simply not supporting it */
58.  /*RTC 时钟的范围是 00~99，由 BCDYEAR 寄存器的 0～7 位存储*/
59.  if (year < 0 || year >= 100) {
60.      dev_err(dev, "rtc only supports 100 years/n");
61.      return -EINVAL;
62.  }
63.  /*将上面保存到 RTC 核心定义的时间结构体中的时间日期值写入对应的寄存器中*/
64.  writeb(bin2bcd(tm->tm_sec),  base + S3C2410_RTCSEC);
65.  writeb(bin2bcd(tm->tm_min),  base + S3C2410_RTCMIN);
66.  writeb(bin2bcd(tm->tm_hour), base + S3C2410_RTCHOUR);
67.  writeb(bin2bcd(tm->tm_mday), base + S3C2410_RTCDATE);
68.  writeb(bin2bcd(tm->tm_mon + 1), base + S3C2410_RTCMON);
69.  writeb(bin2bcd(year), base + S3C2410_RTCYEAR);
70.  return 0;
71.}
```

获取报警时间的值：

```
1.  static int s3c_rtc_getalarm(struct device *dev, struct rtc_wkalrm *alrm)
2.  {
3.      struct rtc_time *alm_tm = &alrm->time;
```

```
4.    void __iomem *base = s3c_rtc_base;
5.    unsigned int alm_en;
6.    /*从 RTC 的报警寄存器中读取*/
7.    alm_tm->tm_sec  = readb(base + S3C2410_ALMSEC);
8.    alm_tm->tm_min  = readb(base + S3C2410_ALMMIN);
9.    alm_tm->tm_hour = readb(base + S3C2410_ALMHOUR);
10.   alm_tm->tm_mon  = readb(base + S3C2410_ALMMON);
11.   alm_tm->tm_mday = readb(base + S3C2410_ALMDATE);
12.   alm_tm->tm_year = readb(base + S3C2410_ALMYEAR);
13.
14.   alm_en = readb(base + S3C2410_RTCALM);
15.   /*根据 RTCALM 寄存器的报警全局使能位来设置报警状态结构 rtc_wkalrm*/
16.   alrm->enabled = (alm_en & S3C2410_RTCALM_ALMEN) ? 1 : 0;
17.   pr_debug("read alarm %02x %02x.%02x.%02x %02x/%02x/%02x/n",
18.       alm_en,
19.       alm_tm->tm_year, alm_tm->tm_mon, alm_tm->tm_mday,
20.       alm_tm->tm_hour, alm_tm->tm_min, alm_tm->tm_sec);
21.
22.   /* decode the alarm enable field */
23.   /*如果 RTCALM 寄存器的秒使能，则将 rtc_wkalrm 中存放的秒数据由 BCD 格式转换为 BIN 格式，否则设置
      为 0xff*/
24.   if (alm_en & S3C2410_RTCALM_SECEN)
25.       alm_tm->tm_sec = bcd2bin(alm_tm->tm_sec);
26.   else
27.       alm_tm->tm_sec = 0xff;
28.   /*如果 RTCALM 寄存器的分钟使能，则将 rtc_wkalrm 中存放的分钟数据由 BCD 格式转换为 BIN 格式，否则
      设置为 0xff*/
29.   if (alm_en & S3C2410_RTCALM_MINEN)
30.       alm_tm->tm_min = bcd2bin(alm_tm->tm_min);
31.   else
32.       alm_tm->tm_min = 0xff;
33.   /*如果 RTCALM 寄存器的小时使能，则将 rtc_wkalrm 中存放的小时数据由 BCD 格式转换为 BIN 格式，否则
      设置为 0xff*/
34.   if (alm_en & S3C2410_RTCALM_HOUREN)
35.       alm_tm->tm_hour = bcd2bin(alm_tm->tm_hour);
36.   else
37.       alm_tm->tm_hour = 0xff;
38.   /*如果 RTCALM 寄存器的日使能，则将 rtc_wkalrm 中存放的日数据由 BCD 格式转换为 BIN 格式，否则设置
      为 0xff*/
39.   if (alm_en & S3C2410_RTCALM_DAYEN)
40.       alm_tm->tm_mday = bcd2bin(alm_tm->tm_mday);
41.   else
42.       alm_tm->tm_mday = 0xff;
43.   /*如果 RTCALM 寄存器的月使能，则将 rtc_wkalrm 中存放的月数据由 BCD 格式转换为 BIN 格式，否则设置
      为 0xff*/
```

```
    if (alm_en & S3C2410_RTCALM_MONEN) {
        alm_tm->tm_mon = bcd2bin(alm_tm->tm_mon);
        alm_tm->tm_mon -= 1;
    } else {
        alm_tm->tm_mon = 0xff;
    }
    /*如果 RTCALM 寄存器的年使能，则将 rtc_wkalrm 中存放的年数据由 BCD 格式转换为 BIN 格式，否则设置为 0xff*/
    if (alm_en & S3C2410_RTCALM_YEAREN)
        alm_tm->tm_year = bcd2bin(alm_tm->tm_year);
    else
        alm_tm->tm_year = 0xffff;
    return 0;
}
```

设置报警时间的值：

```
static int s3c_rtc_setalarm(struct device *dev, struct rtc_wkalrm *alrm)
{
    struct rtc_time *tm = &alrm->time;
    void __iomem *base = s3c_rtc_base;
    unsigned int alrm_en;
    pr_debug("s3c_rtc_setalarm: %d, %02x/%02x/%02x %02x.%02x.%02x/n",
        alrm->enabled,
        tm->tm_mday & 0xff, tm->tm_mon & 0xff, tm->tm_year & 0xff,
        tm->tm_hour & 0xff, tm->tm_min & 0xff, tm->tm_sec);
    /*读取 RTCALM 寄存器的全局使能位，关闭所有报警使能*/
    alrm_en = readb(base + S3C2410_RTCALM) & S3C2410_RTCALM_ALMEN;
    writeb(0x00, base + S3C2410_RTCALM);
    /*如果秒时间在合理范围内，则使能秒报警位，将报警状态寄存器中封装的 time 的秒位由 BIN 格式转换为 BCD，写入秒报警寄存器中*/
    if (tm->tm_sec < 60 && tm->tm_sec >= 0) {
        alrm_en |= S3C2410_RTCALM_SECEN;
        writeb(bin2bcd(tm->tm_sec), base + S3C2410_ALMSEC);
    }
    /*如果分钟时间在合理范围内，则使能分钟报警位，将报警状态寄存器中封装的 time 的分钟位由 BIN 格式转换为 BCD，写入分钟报警寄存器中*/
    if (tm->tm_min < 60 && tm->tm_min >= 0) {
        alrm_en |= S3C2410_RTCALM_MINEN;
        writeb(bin2bcd(tm->tm_min), base + S3C2410_ALMMIN);
    }
    /*如果小时时间在合理范围内，则使能小时报警位，将报警状态寄存器中封装的 time 的小时位由 BIN 格式转换为 BCD，写入小时报警寄存器中*/
    if (tm->tm_hour < 24 && tm->tm_hour >= 0) {
        alrm_en |= S3C2410_RTCALM_HOUREN;
        writeb(bin2bcd(tm->tm_hour), base + S3C2410_ALMHOUR);
    }
```

```
28.    pr_debug("setting S3C2410_RTCALM to %08x/n", alrm_en);
29.    /*使能 RTCALM 寄存器全局报警位*/
30.    writeb(alrm_en, base + S3C2410_RTCALM);
31.    /**/
32.    s3c_rtc_setaie(alrm->enabled);
33.    /*根据全局报警使能的状态来决定是唤醒 RTC 报警中断还是睡眠 RTC 报警中断*/
34.    if (alrm->enabled)
35.        enable_irq_wake(s3c_rtc_alarmno);
36.    else
37.        disable_irq_wake(s3c_rtc_alarmno);
38.    return 0;
39.}
```

下面来分析一下是怎样获取和设置时间的：
通过用户空间的 ioctl，在 rtc-dev.c 中实现了 rtc_dev_ioctl，其中获取和设置时间如下：

```
1.  case RTC_RD_TIME:
2.      mutex_unlock(&rtc->ops_lock);
3.      err = rtc_read_time(rtc, &tm);
4.      if (err < 0)
5.          return err;
6.      if (copy_to_user(uarg, &tm, sizeof(tm)))
7.          err = -EFAULT;
8.      return err;
9.    case RTC_SET_TIME:
10.     mutex_unlock(&rtc->ops_lock);
11.     if (copy_from_user(&tm, uarg, sizeof(tm)))
12.         return -EFAULT;
13.     return rtc_set_time(rtc, &tm);
```

通过 copy_to_user 和 copy_from_user 实现时间在内核空间与用户空间的传递。
这里调用到的 rtc_read_time 和 rtc_set_time 在 interface.c 中实现：

```
1.  int rtc_read_time(struct rtc_device *rtc, struct rtc_time *tm)
2.  {
3.    int err;
4.    err = mutex_lock_interruptible(&rtc->ops_lock);
5.    if (err)
6.        return err;
7.    if (!rtc->ops)
8.        err = -ENODEV;
9.    else if (!rtc->ops->read_time)
10.       err = -EINVAL;
11.   else {
12.       memset(tm, 0, sizeof(struct rtc_time));
```

```
13.         err = rtc->ops->read_time(rtc->dev.parent, tm);
14.     }
15.     mutex_unlock(&rtc->ops_lock);
16.     return err;
17. }
18. int rtc_set_time(struct rtc_device *rtc, struct rtc_time *tm)
19. {
20.     int err;
21.     err = rtc_valid_tm(tm);
22.     if (err != 0)
23.         return err;
24.     err = mutex_lock_interruptible(&rtc->ops_lock);
25.     if (err)
26.         return err;
27.     if (!rtc->ops)
28.         err = -ENODEV;
29.     else if (rtc->ops->set_time)
30.         err = rtc->ops->set_time(rtc->dev.parent, tm);
31.     else if (rtc->ops->set_mmss) {
32.         unsigned long secs;
33.         err = rtc_tm_to_time(tm, &secs);
34.         if (err == 0)
35.             err = rtc->ops->set_mmss(rtc->dev.parent, secs);
36.     } else
37.         err = -EINVAL;
38.     mutex_unlock(&rtc->ops_lock);
39.     return err;
40. }
```

可以看出他们调用了具体 RTC 设备驱动中的 read_time 和 set_time 函数，对应了 s3c2410 中的 s3c_rtc_gettime 和 s3c_rtc_settime,这里使用的 rtc_tm_to_time 函数实现在 rtclib.c 中，/drivers/rtc/interface.c 定义了可供其它模块访问的接口。