网名“鱼树”的学员聂龙浩，

学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细，供大家参考。

也许有错漏，请自行分辨。

目录

驱动框架分析

1. 2.4 内核中的理解：

①，确定主设备号。
②，file_operations 结构
③，register_chrdev(主设备号，名字，file_operations 结构）；
④，入口函数。
⑤，出口函数。
chrdevs 数组，以“主设备号”为下标的“file_operations”结构数组。

2. 2.6 内核这样理解：

①，chrdevs 数组只用 255，则一个内核只能支持 255 个字符设备驱动。以前 2.4 内核确实是有这样的缺点。
但 2.6 内核中很多书上建议不再用“register_chrdev()”了。

②，以前 open 一个字符设备时，虚拟文件系统(VFS)层，有 sys_open,以前是以主设备号为下标，在 chrdevs 数中找到以前注册的 file_oprerations 结构体，现在变化了，是以“主设备号”和“次设备号”两个作为一个整体来找到“file_operations”结构体。

分析“drivers/rtc/rtc-s3c.c”：

```
int register_chrdev(unsigned int major, const char *name,const struct file_operations *fops)
-->cdev = cdev_alloc();
```

搜索 cdev_alloc()，可以见到别人的展开，在“scx200_gpio.c”

```
int __init scx200_gpio_init(void)
-->rc = register_chrdev_region(devid, MAX_PINS, "scx200_gpio");
```

参 1，参 2 的意思：从哪里开始，共有多少个。

```
-->cd = __register_chrdev_region(MAJOR(n), MINOR(n),next - n, name);
-->rc = alloc_chrdev_region(&devid, 0, MAX_PINS, "scx200_gpio");
-->cdev_init(&scx200_gpio_cdev, &scx200_gpio_fileops);
-->cdev_add(&scx200_gpio_cdev, devid, MAX_PINS);
```

③，2.6 内核中对注册字符设备的扩展：对"register_chrdev"的拆分。
❶，若确定了“主设备号”时用“register_chrdev_region()”.没有确定主设备号时用“alloc_chrdev_region()”。区域是指从（某个主设备号、某个次设备号）~（某主设备号，某次设备号+n）都对应于这个 file_operations 结构体。而“register_chrdev()”是从“主设备号 0”到“主设备号 255”都对应“file_operations"结构体。
❷，cdev_init（）；
❸，cdev_add（）；
以上❶~❸步，在"register_chrdev()"中可以见到此过程：

```
int register_chrdev(unsigned int major, const char *name, const struct file_operations
*fops)
-->cd = __register_chrdev_region(major, 0, 256, name);
-->if (major == 0)
-->cdev = cdev_alloc();
-->开始设置
cdev->owner = fops->owner;
cdev->ops = fops;
kobject_set_name(&cdev->kobj, "%s", name);
-->err = cdev_add(cdev, MKDEV(cd->major, 0), 256);
```

```
int register_chrdev_region(dev_t from, unsigned count, const char *name)
```

有主设备号时，有多少个次设备号的范围的字符设备都对些相同“file_operations”结构。
参 1，从哪里开始。
参 2，有多少个。
参 3，名字。

int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,const char *name)
若没有分配主设备号时，先分配一个主设备号放到“&dev”中，
参 1，存放主设备号。
参 2，次设备号的基地址。
参 3，个数。
参 4，名字。

若“devid = MKDEV (major, 0);”时，register_chrdev_region (devid, 2, "hello");是指“(major,0~1)对应'hello_fops'，(major,2~255)不对应 helle_fops”。
“register_chrdev_region()”参 2 是指在有主设备号的情况下，这里有 2 个次设备号（major,0~1）对应于这个“file_operations 结构体 -- hello_fops”。

若“devid = MKDEV (major, 1);”时，register_chrdev_region (devid, 2, "hello");是指“(major,1~2)对应'hello_fops'，(major,3~255)不对应 helle_fops”。

编译测试：

```
/work/drivers_and_test/20th_chrdev_another/hello.c: At top level:
/work/drivers_and_test/20th_chrdev_another/hello.c:32: error: storage size of `hello_cdev' isn't known
```

这是缺少头文件：#include <linux/cdev.h>

```
book@book-desktop:/work/drivers_and_test/20th_chrdev_another/1th$ make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M]  /work/drivers_and_test/20th_chrdev_another/1th/hello.o
/work/drivers_and_test/20th_chrdev_another/1th/hello.c: In function `hello_init':
/work/drivers_and_test/20th_chrdev_another/1th/hello.c:56: warning: implicit declaration of function `class_crea
te'
/work/drivers_and_test/20th_chrdev_another/1th/hello.c:56: warning: assignment makes pointer from integer withou
t a cast
/work/drivers_and_test/20th_chrdev_another/1th/hello.c:57: warning: implicit declaration of function `class_devi
ce_create'
/work/drivers_and_test/20th_chrdev_another/1th/hello.c: In function `hello_exit':
/work/drivers_and_test/20th_chrdev_another/1th/hello.c:67: warning: implicit declaration of function `class_devi
ce_destroy'
/work/drivers_and_test/20th_chrdev_another/1th/hello.c:70: warning: implicit declaration of function `class_dest
roy'
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /work/drivers_and_test/20th_chrdev_another/1th/hello.mod.o
  LD [M]  /work/drivers_and_test/20th_chrdev_another/1th/hello.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
book@book-desktop:/work/drivers_and_test/20th_chrdev_another/1th$
```

```
# insmod hello.ko
# ls /dev/hello* -l
crw-rw----    1 0        0          251,   0 Jan  1 03:40 /dev/hello0
crw-rw----    1 0        0          251,   1 Jan  1 03:40 /dev/hello1
crw-rw----    1 0        0          251,   2 Jan  1 03:40 /dev/hello2
#
```

注册时，就要求了设备号的区域。devid = MKDEV (major, 0);表示了从主设备 major 和次设备号 0 开始。

#define HELLO_CNT　2;　/* 对应次设备号的个数 */　规定了次设备号的个数为 2 个。

```
# ./hello_test /dev/hello0
hello_open
can open /dev/hello0
# ./hello_test /dev/hello1
hello_open
can open /dev/hello1
# ./hello_test /dev/hello2
can't open /dev/hello2
#
```

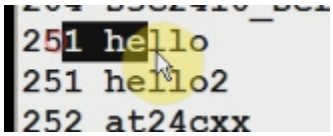
现在是以“主设备号”和“次设备号”一起从内核 chrdevs[]中找到相应的“file_operations”结构体。而 2.4 内核中的方法是以主设备号为下标从 chrdevs[]中找到相应的“file_operations”结构。2.4 中最多有 255 个驱动程序。

```
#define MINORBITS    20
#define MINORMASK    ((1U << MINORBITS) - 1)

#define MAJOR(dev)   ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)   ((unsigned int) ((dev) & MINORMASK))
#define MKDEV(ma,mi)     (((ma) << MINORBITS) | (mi))
```

而且现在的 2.6 内核中，用了 20 位表示次设备号，主设备号是用了 12 位来表示，即有 2^12 * 2^20 个驱动程序。理论上有 4G 个驱动程序。

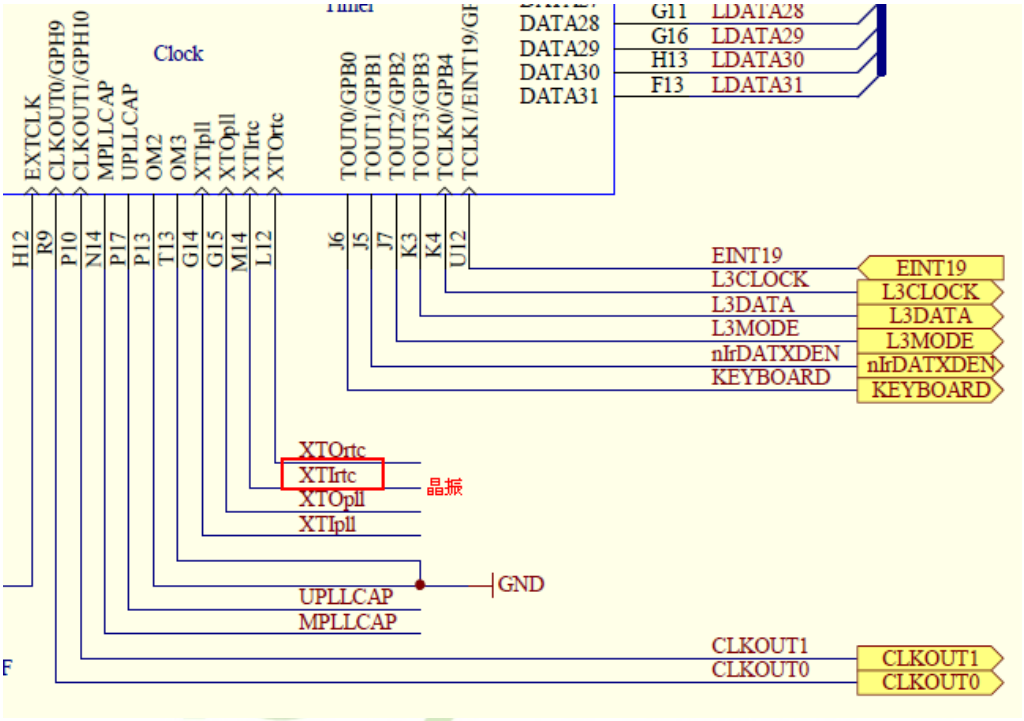
第二个测试程序：
以相同的主设备号，不同的次设备号注册一个新的“file_operations”结构体。

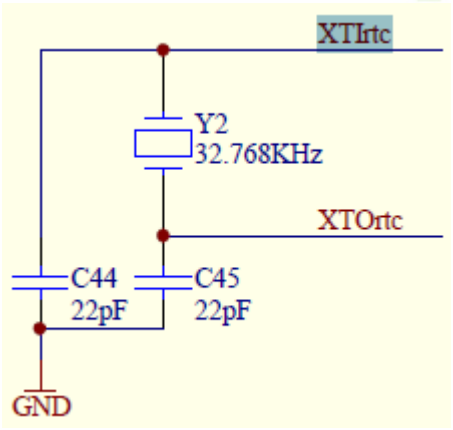



```
# ./hello_test /dev/hello0
hello_open
can open /dev/hello0
# ./hello_test /dev/hello1
hello_open
can open /dev/hello1
# ./hello_test /dev/hello2
hello2_open
can open /dev/hello2
# ls -l /dev/hello*
crw-rw----    1 0        0         251,   0 Jan  1 03:49 /dev/hello0
crw-rw----    1 0        0         251,   1 Jan  1 03:49 /dev/hello1
crw-rw----    1 0        0         251,   2 Jan  1 03:49 /dev/hello2
crw-rw----    1 0        0         251,   3 Jan  1 03:49 /dev/hello3
# ./hello_test /dev/hello3
can't open /dev/hello3
#
```

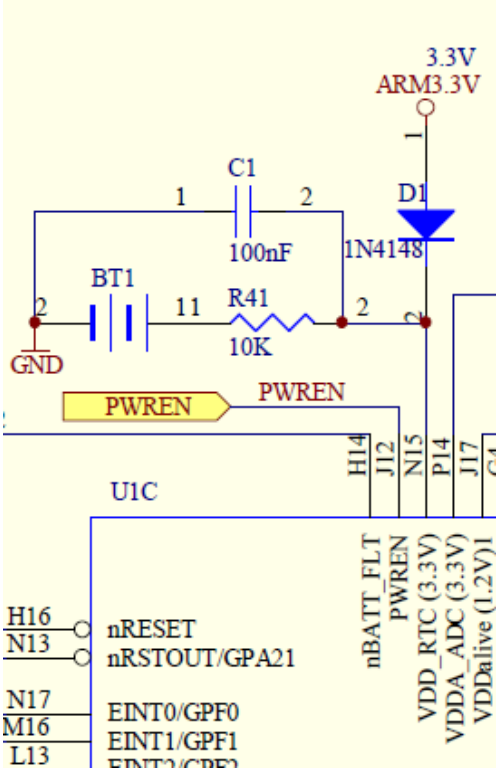
RTC

实时时钟，断电后可以维持。就像手机拔断电池后里面还有个备份电源在维持里面的一个时钟在运行。





给时钟模块使用的晶振。



RTC 时钟的电源。有块电池“BT1”。使得开发板断电后，里面的 RTC 模块还是可以运行的。RTC 模块耗电量很少，一小块电池它就能维持一、二年。

```
drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
```

开发板上电后说无法打开 RTC 设备。

```
# date
Thu Jan  1 00:01:36 UTC 1970
```

开发板上电后，时间总是从 1970 年开始。

内核中有带 RTC 模块的驱动：

drivers/rtc/rtc-s3c.c

```
int __init s3c_rtc_init(void)
-->platform_driver_register(&s3c2410_rtcdrv);注册一个平台驱动
```

```
static struct platform_driver s3c2410_rtcdrv = {
    .probe      = s3c_rtc_probe,
    .remove     = s3c_rtc_remove,
    .suspend    = s3c_rtc_suspend,
    .resume     = s3c_rtc_resume,
    .driver     = {
        .name   = "s3c2410-rtc",
        .owner  = THIS_MODULE,
    },
};
```

内核中有同名“s3c2410-rtc”的设备时，则“.probe”函数才会被调用。

```
int s3c_rtc_probe(struct platform_device *pdev)
-->s3c_rtc_tickno = platform_get_irq(pdev, 1); 从平台设备里获得某些信息。
-->rtc = rtc_device_register("s3c", &pdev->dev, &s3c_rtcops, THIS_MODULE);注册一个RTC设备
```

```
static const struct rtc_class_ops s3c_rtcops = {
    .open       = s3c_rtc_open,
    .release    = s3c_rtc_release,
    .ioctl      = s3c_rtc_ioctl,
    .read_time  = s3c_rtc_gettime,   读时间
    .set_time   = s3c_rtc_settime,   设置时间
    .read_alarm = s3c_rtc_getalarm,  读闹钟
    .set_alarm  = s3c_rtc_setalarm,  设置闹钟
    .proc       = s3c_rtc_proc,
};
```

```
linux-2.6.22.6\drivers\rtc\Class.c 、rtc-dev,c 这是RTC的上一层：
-->rtc_class = class_create(THIS_MODULE, "rtc"); 创建类。
-->rtc_dev_init();在rtc-dev.c中，
   -->err = alloc_chrdev_region(&rtc_devt, 0, RTC_DEV_MAX, "rtc");
```

```
#define RTC_DEV_MAX 16 /* 16 RTCs should be enough for everyone... */
```

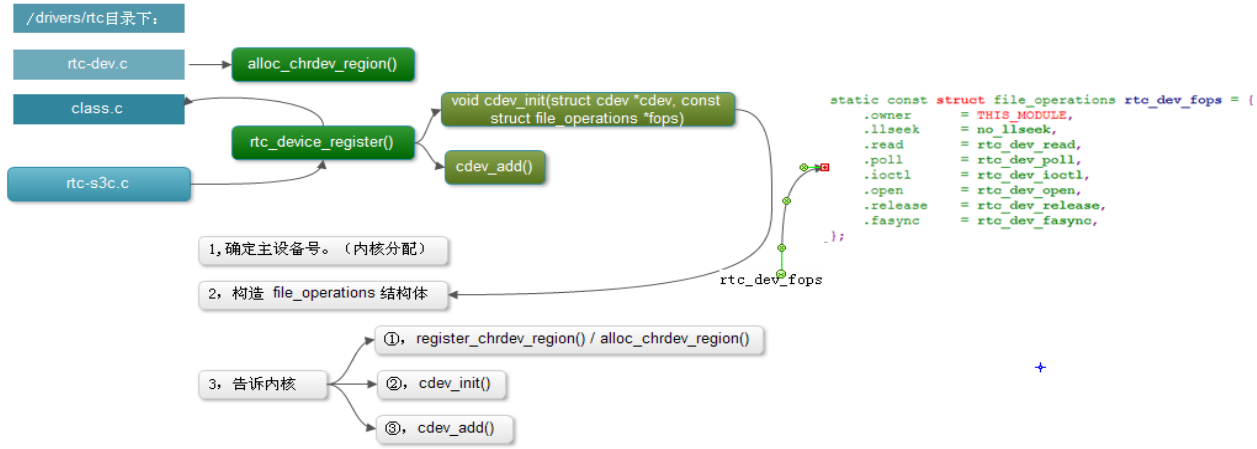
最多有 16 个次设备号。

Class.c 中也实现了“rtc_device_register（）”。

```
rtc_device *rtc_device_register(const char *name, struct device *dev,const struct
rtc_class_ops *ops,
struct module *owner)
-->rtc_dev_prepare(rtc); 准备.
  -->rtc->dev.devt = MKDEV(MAJOR(rtc_devt), rtc->id);
  -->cdev_init(&rtc->char_dev, &rtc_dev_fops);
-->rtc_dev_add_device(rtc);
  -->cdev_add(&rtc->char_dev, rtc->dev.devt, 1);
```

drivers\rtc\rtc-s3c.c

```
s3c_rtc_init
platform_driver_register
s3c_rtc_probe
rtc_device_register("s3c", &pdev->dev, &s3c_rtcops, THIS_MODULE)
rtc_dev_prepare
cdev_init(&rtc->char_dev, &rtc_dev_fops);
                                    rtc_dev_add_device
                                            cdev_add
```



看 RTC 的读写操作：

```
static ssize_t rtc_dev_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
-->struct rtc_device *rtc = to_rtc_device(file->private_data);
用private_data私有数据得到一个rtc_device结构体。
```

app:　　open("/dev/rtc0");

---

kernel: sys_open

　　rtc_dev_fops.open

　　rtc_dev_open

　　// 根据次设备号找到以前用"rtc_device_register"注册的 rtc_device

　　struct rtc_device *rtc = container_of(inode->i_cdev,struct rtc_device, char_dev);

　　const struct rtc_class_ops *ops = rtc->ops;

　　err = ops->open ? ops->open(rtc->dev.parent) : 0;

s3c_rtc_open

关于 rtc_dev_fops 中的读写比较复杂，可以看 busybox 中的 hwclock 的实现：使用 ioctl()

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

app:　　ioctl(fd, RTC_RD_TIME,...)

---

kernel: sys_ioctl

　　rtc_dev_fops.ioctl

rtc_dev_ioctl

struct rtc_device *rtc = file->private_data; 得到 `rtc_device` 结构体

rtc_read_time(rtc, &tm);

err = rtc->ops->read_time(rtc->dev.parent, tm);

s3c_rtc_gettime

“rtc_device”结构中有一个“const struct rtc_class_ops *ops;”。

```
struct rtc_class_ops {
    int (*open)(struct device *);
    void (*release)(struct device *);
    int (*ioctl)(struct device *, unsigned int, unsigned long);
    int (*read_time)(struct device *, struct rtc_time *);
    int (*set_time)(struct device *, struct rtc_time *);
    int (*read_alarm)(struct device *, struct rtc_wkalrm *);
    int (*set_alarm)(struct device *, struct rtc_wkalrm *);
    int (*proc)(struct device *, struct seq_file *);
    int (*set_mmss)(struct device *, unsigned long secs);
    int (*irq_set_state)(struct device *, int enabled);
    int (*irq_set_freq)(struct device *, int freq);
    int (*read_callback)(struct device *, int data);
};
```

```
static const struct rtc_class_ops s3c_rtcops = {
    .open       = s3c_rtc_open,
    .release    = s3c_rtc_release,
    .ioctl      = s3c_rtc_ioctl,
    .read_time  = s3c_rtc_gettime,
    .set_time   = s3c_rtc_settime,
    .read_alarm = s3c_rtc_getalarm,
    .set_alarm  = s3c_rtc_setalarm,
    .proc       = s3c_rtc_proc,
};
```

int s3c_rtc_gettime(struct device *dev, struct rtc_time *rtc_tm)

--> 读相关寄存器

```
retry_get_time:
rtc_tm->tm_min  = readb(base + S3C2410_RTCMIN);  分
rtc_tm->tm_hour = readb(base + S3C2410_RTCHOUR); 时
rtc_tm->tm_mday = readb(base + S3C2410_RTCDATE); 日期
rtc_tm->tm_mon  = readb(base + S3C2410_RTCMON);  月
rtc_tm->tm_year = readb(base + S3C2410_RTCYEAR); 年
rtc_tm->tm_sec  = readb(base + S3C2410_RTCSEC);  秒
```

在开发板启动后，并没有加载 RTC 驱动，但内核中其实已经有了驱动，只是没有加载平台设备。

```
static struct platform_driver s3c2410_rtcdrv = {
    .probe      = s3c_rtc_probe,
    .remove     = s3c_rtc_remove,
    .suspend    = s3c_rtc_suspend,
    .resume     = s3c_rtc_resume,
    .driver     = {
        .name   = "s3c2410-rtc",
        .owner  = THIS_MODULE,
    },
};
```

```
struct platform_device s3c_device_rtc = {
    .name           = "s3c2410-rtc",
    .id         = -1,
    .num_resources      = ARRAY_SIZE(s3c_rtc_resource),
    .resource       = s3c_rtc_resource,
};
```

要注册这个“s3c_device_rtc”平台设备。

```
---- s3c_device_rtc Matches (8 in 7 files) ----
Devs.c (arch\arm\plat-s3c24xx):struct platform_device s3c_device_rtc = {
Devs.c (arch\arm\plat-s3c24xx):EXPORT_SYMBOL(s3c_device_rtc);
Devs.h (include\asm-arm\plat-s3c24xx):extern struct platform_device s3c_device_rtc;
```

可以看到这个“平台设备”并没有使用，上面都是声明和引用。

可以把这个结构体使用起来：common-smdk.c

arch\arm\plat-s3c24xx\Common-smdk.c

```
/* devices we initialise */

static struct platform_device __initdata *smdk_devs[] = {
    &s3c_device_nand,
    &smdk_led4,
    &smdk_led5,
    &smdk_led6,
    &smdk_led7,
    &s3c_device_rtc,
```

加上rtc的平台设备

内核中其他函数会把这个数据“smkd_devs[]”添加进去：

```
platform_add_devices(smdk_devs, ARRAY_SIZE(smdk_devs));
```

修改这个文件后，重新编译内核：make uImage

1. 测试 RTC:

1. 1. 修改 arch\arm\plat-s3c24xx\common-smdk.c

```
static struct platform_device __initdata *smdk_devs[] = {
        &s3c_device_nand,
```

```
        &smdk_led4,
        &smdk_led5,
        &smdk_led6,
        &smdk_led7,
```

改为(在数组 smdk_devs 里加上 s3c_device_rtc):

```
static struct platform_device __initdata *smdk_devs[] = {
        &s3c_device_nand,
        &smdk_led4,
        &smdk_led5,
        &smdk_led6,
        &smdk_led7,
        &s3c_device_rtc,
```

2. 2. make uImage, **使用新内核启动**

3. 3. ls /dev/rtc* -l

```
date /* 显示系统时间 */
date 123015402011.30 /* 设置系统时间 date [MMDDhhmm[[CC]YY][.ss]] */
hwclock -w           /* 把系统时间写入RTC */
```

短电,重启,执行 date