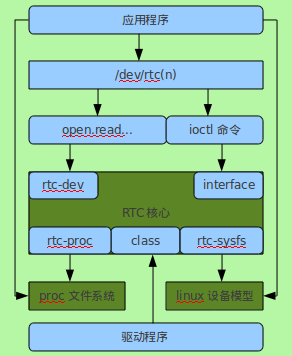
# Linux驱动\_RTC

RTC(real time clock)实时时钟，主要作用是给Linux系统提供时间。RTC因为是电池供电的，所以掉电后时间不丢失。Linux内核把RTC用作“离线”的时间与日期维护器。当Linux内核启动时，它从RTC中读取时间与日期，作为基准值。在运行期间内核完全抛开RTC，以软件的形式维护系统的当前时间与日期，并在需要时将时间回写RTC芯片。

## 1. RTC驱动模型



与RTC核心有关的文件有：

/drivers/rtc/class.c：这个文件向linux设备模型核心注册了一个类RTC，然后向驱动程序提供了注册/注销接口。

/drivers/rtc/rtc-dev.c：这个文件定义了基本的设备文件操作函数，如：open,read。

/drivers/rtc/interface.c：这个文件主要提供了用户程序与RTC驱动的接口函数，用户程序一般通过ioctl与RTC驱动交互，这里定义了每个ioctl命令需要调用的函数。

/drivers/rtc/rtc-sysfs.c：与sysfs有关。

/drivers/rtc/rtc-proc.c：与proc文件系统有关。

/include/linux/rtc.h：定义了与RTC有关的数据结构。

## 2. rtc框架代码分析

### 2.1 rtc platform driver

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**,**

**},**

**};**

static int \_\_init s3c\_rtc\_init**(**void**)**

**{**

printk**(**banner**);**

**return** platform\_driver\_register**(&**s3c2410\_rtcdrv**);**

**}**

熟悉的总线设备模型，这里注册了rtc的platform driver。

### 2.2 rtc platform device

static struct resource s3c\_rtc\_resource**[]** **=** **{**

**[**0**]** **=** **{**

**.**start **=** S3C24XX\_PA\_RTC**,**

**.**end **=** S3C24XX\_PA\_RTC **+** 0xff**,**

**.**flags **=** IORESOURCE\_MEM**,**

**},**

**[**1**]** **=** **{**

**.**start **=** IRQ\_RTC**,**

**.**end **=** IRQ\_RTC**,**

**.**flags **=** IORESOURCE\_IRQ**,**

**},**

**[**2**]** **=** **{**

**.**start **=** IRQ\_TICK**,**

**.**end **=** IRQ\_TICK**,**

**.**flags **=** IORESOURCE\_IRQ

**}**

**};**

struct platform\_device s3c\_device\_rtc **=** **{**

**.**name **=** "s3c2410-rtc"**,**

**.**id **=** **-**1**,**

**.**num\_resources **=** ARRAY\_SIZE**(**s3c\_rtc\_resource**),**

**.**resource **=** s3c\_rtc\_resource**,**

**};**

### 2.3 s3c\_rtc\_probe

当rtc platform driver和device匹配时，probe函数调用。

static int s3c\_rtc\_probe**(**struct platform\_device **\***pdev**)**

**{**

struct rtc\_device **\***rtc**;**

struct resource **\***res**;**

int ret**;**

//获取rtc tick中断号

s3c\_rtc\_tickno **=** platform\_get\_irq**(**pdev**,** 1**);**

//获取rtc alarm中断号

s3c\_rtc\_alarmno **=** platform\_get\_irq**(**pdev**,** 0**);**

//获取rtc操作寄存器地址范围

res **=** platform\_get\_resource**(**pdev**,** IORESOURCE\_MEM**,** 0**);**

//申请内存空间

s3c\_rtc\_mem **=** request\_mem\_region**(**res**->**start**,**

res**->**end**-**res**->**start**+**1**,**

pdev**->**name**);**

//ioremap 得到虚拟地址

s3c\_rtc\_base **=** ioremap**(**res**->**start**,** res**->**end **-** res**->**start **+** 1**);**

//设置rtc寄存器

s3c\_rtc\_enable**(**pdev**,** 1**);**

//set\_rtc\_setfreq()函数用来设置时钟脉冲中断的频率，即多少时间产生一次中断。第一个参数表示RTC的设备结构体，第二个参数表示频率，即多久产生一次中断。如果freq等于1，则表示1秒钟产生一次中断；等于2，表示每秒产生2次中断

s3c\_rtc\_setfreq**(**s3c\_rtc\_freq**);**

//核心函数，注册rtc设备，后面分析

rtc **=** rtc\_device\_register**(**"s3c"**,** **&**pdev**->**dev**,** **&**s3c\_rtcops**,**

THIS\_MODULE**);**

//rtc最大的tick频率为128hz

rtc**->**max\_user\_freq **=** 128**;**

//设置driver\_data，方便的remove的时候做一些反向的操作

platform\_set\_drvdata**(**pdev**,** rtc**);**

**return** 0**;**

**}**

probe中最核心的函数为rtc\_device\_register，向rtc框架注册了该rtc设备。

s3c\_rtc\_enable函数说明如下：

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;**

**if** **(!**en**)** **{**

//en作为参数传递过来如果en==0,关闭电源前的情况

//设置RTCCON寄存器，屏蔽RTC使能

tmp **=** readb**(**base **+** S3C2410\_RTCCON**);**

writeb**(**tmp **&** **~**S3C2410\_RTCCON\_RTCEN**,** base **+** S3C2410\_RTCCON**);**

//设置TICNT寄存器，屏蔽节拍时间中断使能

tmp **=** readb**(**base **+** S3C2410\_TICNT**);**

writeb**(**tmp **&** **~**S3C2410\_TICNT\_ENABLE**,** base **+** S3C2410\_TICNT**);**

**}** **else** **{**

//en!=0的情况，表示系统复位，重新使能RTC驱动

//RTCCON第0位为0，将其设置为1，重新使能

**if** **((**readb**(**base**+**S3C2410\_RTCCON**)** **&** S3C2410\_RTCCON\_RTCEN**)** **==** 0**){**

dev\_info**(&**pdev**->**dev**,** "rtc disabled, re-enabling\n"**);**

tmp **=** readb**(**base **+** S3C2410\_RTCCON**);**

writeb**(**tmp**|**S3C2410\_RTCCON\_RTCEN**,** base**+**S3C2410\_RTCCON**);**

**}**

//设置RTCCON第2位为0，设置BCD计数为混合BCD计数

**if** **((**readb**(**base **+** S3C2410\_RTCCON**)** **&** S3C2410\_RTCCON\_CNTSEL**)){**

dev\_info**(&**pdev**->**dev**,** "removing RTCCON\_CNTSEL\n"**);**

tmp **=** readb**(**base **+** S3C2410\_RTCCON**);**

writeb**(**tmp**&** **~**S3C2410\_RTCCON\_CNTSEL**,** base**+**S3C2410\_RTCCON**);**

**}**

//RTC时钟计数器复位

**if** **((**readb**(**base **+** S3C2410\_RTCCON**)** **&** S3C2410\_RTCCON\_CLKRST**)){**

dev\_info**(&**pdev**->**dev**,** "removing RTCCON\_CLKRST\n"**);**

tmp **=** readb**(**base **+** S3C2410\_RTCCON**);**

writeb**(**tmp **&** **~**S3C2410\_RTCCON\_CLKRST**,** base**+**S3C2410\_RTCCON**);**

**}**

**}**

**}**

### 2.4 rtc\_device\_register

rtc\_device\_register注册了/dev/rtcx字符设备，并创建了对应的sys，proc节点。

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**,**

**};**

struct rtc\_device **\***rtc\_device\_register**(**const char **\***name**,** struct device **\***dev**,**const struct rtc\_class\_ops **\***ops**,**struct module **\***owner**)**

**{**

struct rtc\_device **\***rtc**;**

int id**,** err**;**

//idr机制，这里得到了一个id值，后面介绍

idr\_pre\_get**(&**rtc\_idr**,** GFP\_KERNEL**);**

mutex\_lock**(&**idr\_lock**);**

err **=** idr\_get\_new**(&**rtc\_idr**,** **NULL,** **&**id**);**

mutex\_unlock**(&**idr\_lock**);**

id **=** id **&** MAX\_ID\_MASK**;**

//分配rtc\_device空间

rtc **=** kzalloc**(sizeof(**struct rtc\_device**),** GFP\_KERNEL**);**

//rtc\_device结构体赋值

rtc**->**id **=** id**;**

rtc**->**ops **=** ops**;**

rtc**->**owner **=** owner**;**

rtc**->**max\_user\_freq **=** 64**;**

rtc**->**dev**.**parent **=** dev**;**

rtc**->**dev**.**class **=** rtc\_class**;**

rtc**->**dev**.**release **=** rtc\_device\_release**;**

mutex\_init**(&**rtc**->**ops\_lock**);**

spin\_lock\_init**(&**rtc**->**irq\_lock**);**

spin\_lock\_init**(&**rtc**->**irq\_task\_lock**);**

strlcpy**(**rtc**->**name**,** name**,** RTC\_DEVICE\_NAME\_SIZE**);**

snprintf**(**rtc**->**dev**.**bus\_id**,** BUS\_ID\_SIZE**,** "rtc%d"**,** id**);**

//初始化一些rtc\_device结构体的锁和队列，并准备好字符设备，次设备号为前面idr创建的id

rtc\_dev\_prepare**(**rtc**);**

//sys/class/rtc下面会生成rtcx的目录

err **=** device\_register**(&**rtc**->**dev**);**

//正式添加字符设备

rtc\_dev\_add\_device**(**rtc**);**

//如果rtc都支持wakup和alarm功能，就创建wakealarm属性节点。否则不创建。

//wakeup的能力就是能唤醒suspend-to-RAM/suspend-to-disk设备

//可以通过如下方式判断是否支持wakeup功能

//cat /sys/class/rtc/rtc0/device/power/wakeup

rtc\_sysfs\_add\_device**(**rtc**);**

//创建/proc/device/rtc节点，可以读取rtc系想你

rtc\_proc\_add\_device**(**rtc**);**

**return** rtc**;**

**}**

先来说明下idr机制。

idr在linux内核中指的就是整数ID管理机制，从本质上来说，这就是一种将整数ID号和特定指针关联在一起的机制。

idr机制适用在那些需要把某个整数和特定指针关联在一起的地方。

举个例子，在I2C总线中，每个设备都有自己的地址，要想在总线上找到特定的设备，就必须要先发送该设备的地址。如果我们的PC是一个I2C总线上的主节点，那么要访问总线上的其他设备，首先要知道他们的ID号，同时要在pc的驱动程序中建立一个用于描述该设备的结构体。

此时，问题来了，我们怎么才能将这个设备的ID号和他的设备结构体联系起来呢？最简单的方法当然是通过数组进行索引，但如果ID号的范围很大(比如32位的ID号)，则用数组索引显然不可能；第二种方法是用链表，但如果网络中实际存在的设备较多，则链表的查询效率会很低。遇到这种清况，我们就可以采用idr机制，该机制内部采用radix树实现，可以很方便地将整数和指针关联起来，并且具有很高的搜索效率。

使用方法如下：

**(**1**)**获得idr

要在代码中使用idr，首先要包括**<**linux**/**idr**.**h**>**。接下来，我们要在代码中分配idr结构体，并初始化：

void idr\_init**(**struct idr **\***idp**);**

**(**2**)**为idr分配内存

int idr\_pre\_get**(**struct idr **\***idp**,** unsigned int gfp\_mask**);**

每次通过idr获得ID号之前，需要先分配内存。

返回0表示错误，非零值代表正常

**(**3**)**分配ID号并将ID号和指针关联

int idr\_get\_new**(**struct idr **\***idp**,** void **\***ptr**,** int **\***id**);**

idp**:** 之前通过idr\_init初始化的idr指针

id**:**  由内核自动分配的ID号

ptr**:** 和ID号相关联的指针

**(**4**)**通过ID号搜索对应的指针

void **\***idr\_find**(**struct idr **\***idp**,** int id**);**

返回值是和给定id相关联的指针，如果没有，则返回NULL

**(**5**)**删除ID

要删除一个ID，使用：

void idr\_remove**(**struct idr **\***idp**,** int id**);**

下面我们再看下rtc\_dev\_prepare函数。

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**,**

**};**

void rtc\_dev\_prepare**(**struct rtc\_device **\***rtc**)**

**{**

rtc**->**dev**.**devt **=** MKDEV**(**MAJOR**(**rtc\_devt**),** rtc**->**id**);**

mutex\_init**(&**rtc**->**char\_lock**);**

spin\_lock\_init**(&**rtc**->**irq\_lock**);**

init\_waitqueue\_head**(&**rtc**->**irq\_queue**);**

cdev\_init**(&**rtc**->**char\_dev**,** **&**rtc\_dev\_fops**);**

rtc**->**char\_dev**.**owner **=** rtc**->**owner**;**

**}**

该函数准备好了字符设备，对应的file\_operations为rtc\_dev\_fops，所有的rtc设备操作函数都是一样的。

### 2.5 rtc\_init

在rtc\_device\_register函数中，会碰到rtc\_class变量，这个变量在哪儿创建的呢？

rtc\_init做了很多准备的工作。

static int \_\_init rtc\_init**(**void**)**

**{**

//创建/sys/class/rtc目录

rtc\_class **=** class\_create**(**THIS\_MODULE**,** "rtc"**);**

rtc\_class**->**suspend **=** rtc\_suspend**;**

rtc\_class**->**resume **=** rtc\_resume**;**

//分配rtc设备主设备号

rtc\_dev\_init**();**

// /sys/class/rtc/rtcx下会出现name date time since\_epoch节点。

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\_\_**);**

**}**

static struct device\_attribute rtc\_attrs**[]** **=** **{**

\_\_ATTR**(**name**,** S\_IRUGO**,** rtc\_sysfs\_show\_name**,** **NULL),**

\_\_ATTR**(**date**,** S\_IRUGO**,** rtc\_sysfs\_show\_date**,** **NULL),**

\_\_ATTR**(**time**,** S\_IRUGO**,** rtc\_sysfs\_show\_time**,** **NULL),**

\_\_ATTR**(**since\_epoch**,** S\_IRUGO**,** rtc\_sysfs\_show\_since\_epoch**,** **NULL),**

**{** **},**

**};**

void \_\_init rtc\_sysfs\_init**(**struct class **\***rtc\_class**)**

**{**

rtc\_class**->**dev\_attrs **=** rtc\_attrs**;**

**}**

## 3.RTC字符设备操作分析

最终会创建一个/dev/rtc0这样的字符设备，我们来分析一下该字符设备的操作函数。

rtc字符设备的操作函数集合如下：

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**,**

**};**

### 3.1 rtc\_dev\_open

rtc\_dev\_open会申请alarm和tick中断，中断的用处将在read函数中说明。

static int rtc\_dev\_open**(**struct inode **\***inode**,** struct file **\***file**)**

**{**

int err**;**

//获取该rtc字符设备对应的rtc\_device结构体。

struct rtc\_device **\***rtc **=** container\_of**(**inode**->**i\_cdev**,**

struct rtc\_device**,** char\_dev**);**

//获取实际的rtc操作函数

const struct rtc\_class\_ops **\***ops **=** rtc**->**ops**;**

//每个rtc设备不能同时被打开

**if** **(!(**mutex\_trylock**(&**rtc**->**char\_lock**)))**

**return** **-**EBUSY**;**

//设置private data

file**->**private\_data **=** rtc**;**

//这里会调用s3c\_rtc\_open

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

mutex\_unlock**(&**rtc**->**char\_lock**);**

**return** err**;**

**}**

static int s3c\_rtc\_open**(**struct device **\***dev**)**

**{**

//获取到对应的rtc\_device结构体

struct platform\_device **\***pdev **=** to\_platform\_device**(**dev**);**

struct rtc\_device **\***rtc\_dev **=** platform\_get\_drvdata**(**pdev**);**

int ret**;**

//申请alarm中断

ret **=** request\_irq**(**s3c\_rtc\_alarmno**,** s3c\_rtc\_alarmirq**,**

IRQF\_DISABLED**,** "s3c2410-rtc alarm"**,** rtc\_dev**);**

//申请tick中断

ret **=** request\_irq**(**s3c\_rtc\_tickno**,** s3c\_rtc\_tickirq**,**

IRQF\_DISABLED**,** "s3c2410-rtc tick"**,** rtc\_dev**);**

**return** ret**;**

**}**

### 3.2 rtc\_dev\_read

在RTC设备节点上使用read()函数将会使进程进入睡眠，直到有一个中断发生。中断发生后从RTC设备节点只能读取一个unsigned long类型的变量，无论发生了几次中断，一次只能读取一个unsigned long。读取到的这个unsigned long变量的最低一字节反应已经发生的中断类型，其余的字节反应自上次读取以来发生中断的次数。注意，如果自上次读取以来已发生了多钟类型的中断，并且每种类型的中断都发生了多次，根本没有办法判断每种中断类型各发生了几次。

RTC共支持三种类型的中断：更新中断、周期中断和alarm/wakeupalarm中断。更新中断是在RTC的时间变化的时候发生的中断，由于RTC的最小单位是秒，所以更新中断的频率是1HZ；周期中断是频率高于1HZ的中断（必须高于1HZ）?，中断频率必须是2^N（N>=1，整数）HZ，并且只有Root用户能使用64HZ以上的频率；alarm/wakeupalarm中断是在未来某个时刻发生的中断，它只能发生一次，发生过后需要重新设置才能再次发生, 其中alarm中断只能将中断发生时刻设定在24小时之内，wakeupalarm中断可以设定为未来的任意时刻。

#define RTC\_IRQF 0x80 /\* 任意中断发生它都会置位\*/

#define RTC\_PF 0x40 /\* 周期中断发生后该位置位\*/

#define RTC\_AF 0x20 /\* alarm/wakeup alarm中断 \*/

#define RTC\_UF 0x10 /\* 更新中断\*/

当有多种类型的中断发生后而用户空间有没来得及读取的时候，会有多个位被置1。高字节中的中断次数是所有中断次数之和。

所以这里的read不是应用程序用来获取时间的，而是有其他的作用，他帮助应用程序周期性的完成一些工作，例如借助tick中断，一直read rtc设备，可以完成一些定时的任务。

中断处理函数如下：

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**,** tick\_count**++,** RTC\_PF **|** RTC\_IRQF**);**

**return** IRQ\_HANDLED**;**

**}**

void rtc\_update\_irq**(**struct rtc\_device **\***rtc**,**unsigned long num**,** unsigned long events**)**

**{**

spin\_lock**(&**rtc**->**irq\_lock**);**

//设置中断数据，用于read读出 低8位用于存储中断的类型

rtc**->**irq\_data **=** **(**rtc**->**irq\_data **+** **(**num **<<** 8**))** **|** events**;**

spin\_unlock**(&**rtc**->**irq\_lock**);**

spin\_lock**(&**rtc**->**irq\_task\_lock**);**

//如果有irq\_task，则调用，这里没有

**if** **(**rtc**->**irq\_task**)**

rtc**->**irq\_task**->**func**(**rtc**->**irq\_task**->**private\_data**);**

spin\_unlock**(&**rtc**->**irq\_task\_lock**);**

//唤醒poll队列，read读

wake\_up\_interruptible**(&**rtc**->**irq\_queue**);**

//唤醒sigio,read读

kill\_fasync**(&**rtc**->**async\_queue**,** SIGIO**,** POLL\_IN**);**

**}**

read操作源码如下：

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**);**

//声明一个等待队列入口

DECLARE\_WAITQUEUE**(**wait**,** current**);**

unsigned long data**;**

ssize\_t ret**;**

//将这个入口加入到RTC的irq等待队列里

add\_wait\_queue**(&**rtc**->**irq\_queue**,** **&**wait**);**

//进入循环

**do** **{**

//把进程的状态改成TASK\_INTERRUPTIBLE

\_\_set\_current\_state**(**TASK\_INTERRUPTIBLE**);**

spin\_lock\_irq**(&**rtc**->**irq\_lock**);**

//获取中断数据

data **=** rtc**->**irq\_data**;**

rtc**->**irq\_data **=** 0**;**

spin\_unlock\_irq**(&**rtc**->**irq\_lock**);**

//如果取到了数据，直接返回

**if** **(**data **!=** 0**)** **{**

ret **=** 0**;**

**break;**

**}**

//如果是非阻塞read，直接返回

**if** **(**file**->**f\_flags **&** O\_NONBLOCK**)** **{**

ret **=** **-**EAGAIN**;**

**break;**

**}**

**if** **(**signal\_pending**(**current**))** **{**

ret **=** **-**ERESTARTSYS**;**

**break;**

**}**

//进程调度，睡眠

schedule**();**

**}** **while** **(**1**);**

set\_current\_state**(**TASK\_RUNNING**);**

remove\_wait\_queue**(&**rtc**->**irq\_queue**,** **&**wait**);**

**if** **(**ret **==** 0**)** **{**

/\* Check for any data updates \*/

**if** **(**rtc**->**ops**->**read\_callback**)**

data **=** rtc**->**ops**->**read\_callback**(**rtc**->**dev**.**parent**,**

data**);**

//将数据返回到用户空间

**if** **(sizeof(**int**)** **!=** **sizeof(**long**)** **&&** count **==** **sizeof(**unsigned int**))**

ret **=** put\_user**(**data**,** **(**unsigned int \_\_user **\*)**buf**)** **?:**

**sizeof(**unsigned int**);**

**else**

ret **=** put\_user**(**data**,** **(**unsigned long \_\_user **\*)**buf**)** **?:**

**sizeof(**unsigned long**);**

**}**

**return** ret**;**

**}**

### 3.3 rtc\_dev\_ioctl

//alarm操作

**(**1**)**RTC\_ALM\_SET

设置alarm中断的触发时刻，使用的方式如下：

ioctl**(**fd**,**RTC\_ALM\_SET**,** **&**rtc\_tm**);**

其中第三个参数是一个struct rtc\_time结构体**.**

注意，alarm中断的触发时间只能是24小时内的一个时刻，所以只有时、分、秒的部分是有效的，struct rtc\_time的年、月、日部分会被忽略。为了保证正确性，struct rtc\_time的tm\_isdst成员应该设为**-**1。

如当前时间是13**:**00**:**00**,**而传入的时刻是14**:**00**:**00**,**则意味着alarm中断将在一小时后发生。而如果传入的是12**:**00**:**00**,**则意味着明天中午发生中断。总之，alarm中断不能超过24小时。

**(**2**)**RTC\_ALM\_READ

读取已经设置的alarm中断时刻，使用的方式如下：

ioctl**(**fd**,**RTC\_ALM\_READ**,** **&**rtc\_tm**);**

**(**3**)**RTC\_WKALM\_SET

设置wakeupalarm中断的触发时刻，它和alarm中断的唯一不同之处就是wakeupalarm中断的触发时刻可以在未来的任意时刻。它的使用方式如下：

ioctl**(**fd**,**RTC\_WKALM\_SET**,** **&**alarm**);**

第三个参数是一个struct rtc\_wkalrm结构体**,**struct rtc\_wkalrm的第一个成员enabled为1的话可以使ioctl**()**返回的时候中断功能就已经开启了。如果这里里没有开启中断功能，可以使用RTC\_AIE\_ON命令来开启，这个命令的用法在下面。

由于alarm和wakeupalarm在内核中表现为一种中断，因此同一时刻只有一个是有效的。也就是说不可以使用alarm中断的同时使用wakeupalarm中断。只有最后一次设置的alarm或wakeupalarm中断触发时刻是有效的。

**(**4**)**RTC\_WKALM\_RD

读取wakeupalarm中断的触发时刻。它的使用方式如下：

ioctl**(**fd**,**RTC\_WKALM\_RD**,** **&**alarm**);**

//更新中断操作

**(**5**)**RTC\_IRQP\_SET

设置周期中断的频率。使用的方法如下：

ioctl**(**fd**,**RTC\_IRQP\_SET**,** tmp**);**

tmp是一个unsigned long类型的变量，它的值必须是2的幂，也就是2**^**N（N**>=**1**)**。非Root用户无法使用64HZ以上的周期中断。

**(**6**)**RTC\_IRQP\_READ

读取周期中断的频率。使用方式如下：

ioctl**(**fd**,**RTC\_IRQP\_READ**,** **&**tmp**);**

tmp是一个unsigned long类型的变量，它保存返回的周期中断频率。

//RTC时间操作

**(**7**)**RTC\_SET\_TIME

设置RTC的时间。这个命令和中断无关，用于更新RTC芯片的当前时间。使用的方式如下：

ioctl**(**fd**,**RTC\_SET\_TIME**,** **&**rtc\_tm**)**

第三个参数以一个struct rtc\_time变量**.**

**(**8**)**RTC\_RD\_TIME

读取RTC硬件中的当前时刻。使用的方式如下：

ioctl**(**fd**,**RTC\_RD\_TIME**,** **&**rtc\_tm**);**

## 4. 测试