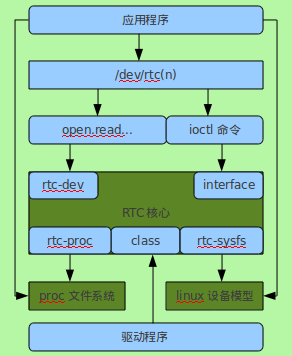
**linux RTC 驱动模型分析**

        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。另外如果RTC提供了IRQ中断并且可以定时，那么RTC还可以作为内核睡眠时唤醒内核的闹钟。应用程序可以用RTC提供的周期中断做一些周期的任务。 linux有两种rtc驱动的接口，一个是老的接口，专门用在PC机上的。另外一钟新接口是基于linux设备驱动程序的。这个新的接口创建了一个RTC驱动模型，实现了RTC的大部分基本功能。而底层驱动无须考虑一些功能的实现，只需将自己注册的RTC核心中，其他工作由RTC核心来完成。下面分析RTC新接口的驱动模型。  
一. 驱动模型结构  
        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等  
        /drivers/rtc/interface.c     顾名思义，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

        RTC驱动模型结构如下图：



二. 基本数据结构  
  1. struct rtc\_device 结构

1. **struct** rtc\_device
2. {
3. **struct** device dev;
4. **struct** module \*owner;
6. **int** id;
7. **char** name[RTC\_DEVICE\_NAME\_SIZE];
9. **const** **struct** rtc\_class\_ops \*ops;
10. **struct** mutex ops\_lock;
12. **struct** cdev char\_dev;
13. unsigned **long** flags;
15. unsigned **long** irq\_data;
16. spinlock\_t irq\_lock;
17. wait\_queue\_head\_t irq\_queue;
18. **struct** fasync\_struct \*async\_queue;
20. **struct** rtc\_task \*irq\_task;
21. spinlock\_t irq\_task\_lock;
22. **int** irq\_freq;
23. **int** max\_user\_freq;
24. #ifdef CONFIG\_RTC\_INTF\_DEV\_UIE\_EMUL
25. **struct** work\_struct uie\_task;
26. **struct** timer\_list uie\_timer;
27. /\* Those fields are protected by rtc->irq\_lock \*/
28. unsigned **int** oldsecs;
29. unsigned **int** uie\_irq\_active:1;
30. unsigned **int** stop\_uie\_polling:1;
31. unsigned **int** uie\_task\_active:1;
32. unsigned **int** uie\_timer\_active:1;
33. #endif
34. };

        这个结构是RTC驱动程序的基本数据结构，但是他不像其他核心的基本结构一样，驱动程序以他为参数调用注册函数注册到核心。这个结构是由注册函数返回给驱动程序的。  
  2. struct rtc\_class\_ops 结构

1. **struct** rtc\_class\_ops {
2. **int** (\*open)(**struct** device \*);
3. **void** (\*release)(**struct** device \*);
4. **int** (\*ioctl)(**struct** device \*, unsigned **int**, unsigned **long**);
5. **int** (\*read\_time)(**struct** device \*, **struct** rtc\_time \*);
6. **int** (\*set\_time)(**struct** device \*, **struct** rtc\_time \*);
7. **int** (\*read\_alarm)(**struct** device \*, **struct** rtc\_wkalrm \*);
8. **int** (\*set\_alarm)(**struct** device \*, **struct** rtc\_wkalrm \*);
9. **int** (\*proc)(**struct** device \*, **struct** seq\_file \*);
10. **int** (\*set\_mmss)(**struct** device \*, unsigned **long** secs);
11. **int** (\*irq\_set\_state)(**struct** device \*, **int** enabled);
12. **int** (\*irq\_set\_freq)(**struct** device \*, **int** freq);
13. **int** (\*read\_callback)(**struct** device \*, **int** data);
14. **int** (\*alarm\_irq\_enable)(**struct** device \*, unsigned **int** enabled);
15. **int** (\*update\_irq\_enable)(**struct** device \*, unsigned **int** enabled);
16. };

        这个结构是RTC驱动程序要实现的基本操作函数，注意这里的操作不是文件操作。驱动程序通过初始化这样一个结构，将自己实现的函数与RTC核心联系起来。这里面的大部分函数都要驱动程序来实现。而且这些函数都是操作底层硬件的，属于最底层的函数。  
  3. struct rtc\_time 结构

1. **struct** rtc\_time {
2. **int** tm\_sec;
3. **int** tm\_min;
4. **int** tm\_hour;
5. **int** tm\_mday;
6. **int** tm\_mon;
7. **int** tm\_year;
8. **int** tm\_wday;
9. **int** tm\_yday;
10. **int** tm\_isdst;
11. };

        代表了时间与日期，从RTC设备读回的时间和日期就保存在这个结构体中  
三. class.c   
  1. 模块初始化函数：rtc\_init

1. **static** **int** \_\_init rtc\_init(**void**)
2. {
3. rtc\_class = class\_create(THIS\_MODULE, "rtc");
4. **if** (IS\_ERR(rtc\_class)) {
5. printk(KERN\_ERR "%s: couldn't create class\n", \_\_FILE\_\_);
6. **return** PTR\_ERR(rtc\_class);
7. }
8. rtc\_class->suspend = rtc\_suspend;
9. rtc\_class->resume = rtc\_resume;
10. rtc\_dev\_init();
11. rtc\_sysfs\_init(rtc\_class);
12. **return** 0;
13. }

        rtc\_init首先调用class\_create创建了一个类--rtc。我们知道类是一个设备的高层视图，他抽象出了底层的实现细节。类的作用就是向用户空间提供设备的信息，驱动程序不需要直接处理类。然后初始化类结构的相应成员，rtc\_suspend，rtc\_resume这两个函数也是在class.c中实现的。接下来调用rtc\_dev\_init()，这个函数为RTC设备动态分配设备号，保存在rtc\_devt中。最后调用rtc\_sysfs\_init，初始化rtc\_class的属性。  
  2. 为底层驱动提供接口：rtc\_device\_register，rtc\_device\_unregister

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;
8. **if** (idr\_pre\_get(&rtc\_idr, GFP\_KERNEL) == 0) {
9. err = -ENOMEM;
10. **goto** exit;
11. }
13. mutex\_lock(&idr\_lock);
14. err = idr\_get\_new(&rtc\_idr, NULL, &id);
15. mutex\_unlock(&idr\_lock);
16. /\*--------------------(1)---------------------\*/
17. **if** (err < 0)
18. **goto** exit;
19. id = id & MAX\_ID\_MASK;
20. rtc = kzalloc(**sizeof**(**struct** rtc\_device), GFP\_KERNEL);
21. **if** (rtc == NULL) {
22. err = -ENOMEM;
23. **goto** exit\_idr;
24. }
26. rtc->id = id;
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;
34. mutex\_init(&rtc->ops\_lock);
35. spin\_lock\_init(&rtc->irq\_lock);
36. spin\_lock\_init(&rtc->irq\_task\_lock);
37. init\_waitqueue\_head(&rtc->irq\_queue);
39. strlcpy(rtc->name, name, RTC\_DEVICE\_NAME\_SIZE);
40. dev\_set\_name(&rtc->dev, "rtc%d", id);
41. /\*-------------------(2)--------------------\*/
42. rtc\_dev\_prepare(rtc);

45. err = device\_register(&rtc->dev);
46. **if** (err)
47. **goto** exit\_kfree;
48. /\*-------------------(3)--------------------\*/
49. rtc\_dev\_add\_device(rtc);
50. rtc\_sysfs\_add\_device(rtc);
51. rtc\_proc\_add\_device(rtc);
53. dev\_info(dev, "rtc core: registered %s as %s\n",
54. rtc->name, dev\_name(&rtc->dev));
55. /\*-------------------(4)--------------------\*/
56. **return** rtc;
58. exit\_kfree:
59. kfree(rtc);
60. exit\_idr:
61. mutex\_lock(&idr\_lock);
62. idr\_remove(&rtc\_idr, id);
63. mutex\_unlock(&idr\_lock);
64. exit:
65. dev\_err(dev, "rtc core: unable to register %s, err = %d\n",
66. name, err);
67. **return** ERR\_PTR(err);
68. }

    （1）：处理一个idr的结构，idr在linux内核中指的就是整数ID管理机制，从本质上来说，idr是一种将整数ID号和特定指针关联在一起的机制。这个机制最早是在2003年2月加入内核的，当时是作为POSIX定时器的一个补丁。现在，在内核的很多地方都可以找到idr的身影。详细实现请参照相关内核代码。这里从内核中获取一个idr结构，并与id相关联。  
    （2）：分配了一个rtc\_device的结构--rtc，并且初始化了相关的成员：id, rtc\_class\_ops等等。  
    （3）：首先调用rtc\_dev\_prepare（在rtc-dev.c中定义）。因为RTC设备本质来讲还是字符设备，所以这里初始化了字符设备相关的结构：设备号以及文件操作。然后调用device\_register将设备注册到linux设备模型核心。这样在模块加载的时候，udev daemon就会自动为我们创建设备文件rtc(n)。  
    （4）：先后调用rtc\_dev\_add\_device，rtc\_sysfs\_add\_device，rtc\_proc\_add\_device三个函数。rtc\_dev\_add\_device注册字符设备，rtc\_sysfs\_add\_device只是为设备添加了一个闹钟属性，rtc\_proc\_add\_device 创建proc文件系统接口。  
四. rtc-dev.c   
        rtc-dev.c 初始化了一个file\_operations结构--rtc\_dev\_fops，并定义了这些操作函数。  
  1. rtc\_dev\_fops rtc基本的文件操作

**[cpp]** [view plain](http://blog.csdn.net/yaozhenguo2006/article/details/6824970) [copy](http://blog.csdn.net/yaozhenguo2006/article/details/6824970)

1. **static** **const** **struct** file\_operations rtc\_dev\_fops = {
2. .owner      = THIS\_MODULE,
3. .llseek     = no\_llseek,
4. .read       = rtc\_dev\_read,
5. .poll       = rtc\_dev\_poll,
6. .unlocked\_ioctl = rtc\_dev\_ioctl,
7. .open       = rtc\_dev\_open,
8. .release    = rtc\_dev\_release,
9. .fasync     = rtc\_dev\_fasync,
10. ;

  2. 函数的实现（以rtc\_dev\_read为例）   

1. rtc\_dev\_read(**struct** file \*file, **char** \_\_user \*buf, **size\_t** count, loff\_t \*ppos)
2. {
3. **struct** rtc\_device \*rtc = file->private\_data;
5. DECLARE\_WAITQUEUE(wait, current);
6. unsigned **long** data;
7. ssize\_t ret;
8. **if** (count != **sizeof**(unsigned **int**) && count < **sizeof**(unsigned **long**))
9. **return** -EINVAL;
10. add\_wait\_queue(&rtc->irq\_queue, &wait);
11. **do** {
12. \_\_set\_current\_state(TASK\_INTERRUPTIBLE);
14. spin\_lock\_irq(&rtc->irq\_lock);
15. data = rtc->irq\_data;
16. rtc->irq\_data = 0;
17. spin\_unlock\_irq(&rtc->irq\_lock);
19. **if** (data != 0) {
20. ret = 0;
21. **break**;
22. }
23. **if** (file->f\_flags & O\_NONBLOCK) {
24. ret = -EAGAIN;
25. **break**;
26. }
27. **if** (signal\_pending(current)) {
28. ret = -ERESTARTSYS;
29. **break**;
30. }
31. schedule();
32. } **while** (1);
33. set\_current\_state(TASK\_RUNNING);
34. remove\_wait\_queue(&rtc->irq\_queue, &wait);
36. **if** (ret == 0) {
37. /\* Check for any data updates \*/
38. **if** (rtc->ops->read\_callback)
39. data = rtc->ops->read\_callback(rtc->dev.parent,
40. data);
42. **if** (**sizeof**(**int**) != **sizeof**(**long**) &&
43. count == **sizeof**(unsigned **int**))
44. ret = put\_user(data, (unsigned **int** \_\_user \*)buf) ?:
45. **sizeof**(unsigned **int**);
46. **else**
47. ret = put\_user(data, (unsigned **long** \_\_user \*)buf) ?:
48. **sizeof**(unsigned **long**);
49. }
50. **return** ret;
51. }

        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。如果要使用这个功能，应用程序首先保证RTC驱动程序提供这样的功能。这个功能是这样实现的：进程读取/dev/rtc(n)，进程睡眠直到RTC中断将他唤醒。我们可以发现，这里的睡眠是ldd3中提到的手工睡眠。这个函数的手工休眠过程如下：首先调用DECLARE\_WAITQUEUE(wait, current)，声明一个等待队列入口，然后调用add\_wait\_queue将这个入口加入到RTC的irq等待队列里，然后进入循环。在循环里首先把进程的状态改成TASK\_INTERRUPTIBLE，这样进程就不能再被调度运行。但是现在进程还在运行，没有进入睡眠状态。程序然后读取RTC里面的irq\_data，如果不是零，那么程序跳出这个循环，进程不会睡眠。因为这个irq\_data在rtc的中断处理程序会被赋值，而读过之后就会清零，所以如果数据不是零的话说明发生过一次中断。如果是零那么没有发生中断，调用schedule，进程会被调度出可运行队列，从而让出处理器，真正进入睡眠。跳出循环代表被唤醒，然后将进程状态改变为可运行，移除等待队列入口。最后将读回的数据传给用户空间。  
五. interface.c   
        interface.c里的所有函数的实现都对应于rtc-dev.c 中ioctl相应的命令。对应关系如下：  
RTC\_ALM\_READ                     rtc\_read\_alarm           读取闹钟时间  
RTC\_ALM\_SET                      rtc\_set\_alarm            设置闹钟时间  
RTC\_RD\_TIME                      rtc\_read\_time            读取时间与日期  
RTC\_SET\_TIME                     rtc\_set\_time             设置时间与日期  
RTC\_PIE\_ON RTC\_PIE\_OFF           rtc\_irq\_set\_state              开关RTC全局中断的函数  
RTC\_AIE\_ON RTC\_AIE\_OFF           rtc\_alarm\_irq\_enable     使能禁止RTC闹钟中断  
RTC\_UIE\_OFF RTC\_UIE\_ON           rtc\_update\_irq\_enable    使能禁止RTC更新中断  
RTC\_IRQP\_SET                     rtc\_irq\_set\_freq         设置中断的频率  
       以上就是所有ioctl的命令与实现的对应关系。其中如果不涉及中断的话，有两个命令需要我们特别关心一下，就是RTC\_RD\_TIME与RTC\_SET\_TIME。因为RTC最基本的功能就是提供时间与日期。这两个命令恰恰是获取时间和设置时间。下面分析一下这两个命令的实现，也就是rtc\_set\_alarm与rtc\_read\_time函数的实现：  
  1. rtc\_read\_time 函数

1. **int** rtc\_read\_time(**struct** rtc\_device \*rtc, **struct** rtc\_time \***tm**)
2. {
3. **int** err;
5. err = mutex\_lock\_interruptible(&rtc->ops\_lock);
6. **if** (err)
7. **return** err;
9. **if** (!rtc->ops)
10. err = -ENODEV;
11. **else** **if** (!rtc->ops->read\_time)
12. err = -EINVAL;
13. **else** {
14. memset(**tm**, 0, **sizeof**(**struct** rtc\_time));
15. err = rtc->ops->read\_time(rtc->dev.parent, **tm**);
16. }
18. mutex\_unlock(&rtc->ops\_lock);
19. **return** err;
20. }

        这个函数用了一个信号来保证在同一时刻只有一个进程可以获取时间。锁定了这个信号量后，调用rtc->ops里面read函数，这个函数是由具体的驱动程序实现的，操作底层硬件。读回的时间存放在rtc\_time结构里面的。  
  2. rtc\_set\_time 函数

1. **int** rtc\_set\_time(**struct** rtc\_device \*rtc, **struct** rtc\_time \***tm**)
2. {
3. **int** err;
5. err = rtc\_valid\_tm(**tm**);
6. **if** (err != 0)
7. **return** err;
9. err = mutex\_lock\_interruptible(&rtc->ops\_lock);
10. **if** (err)
11. **return** err;
13. **if** (!rtc->ops)
14. err = -ENODEV;
15. **else** **if** (rtc->ops->set\_time)
16. err = rtc->ops->set\_time(rtc->dev.parent, **tm**);
17. **else** **if** (rtc->ops->set\_mmss) {
18. unsigned **long** secs;
19. err = rtc\_tm\_to\_time(**tm**, &secs);
20. **if** (err == 0)
21. err = rtc->ops->set\_mmss(rtc->dev.parent, secs);
22. } **else**
23. err = -EINVAL;
25. mutex\_unlock(&rtc->ops\_lock);
26. **return** err;
27. }

        这个函数其实和rtc\_read\_time函数差不多，同样是锁定信号量，同样是调用底层驱动函数。但是这里的设置时间提供了两个调用：一个是set\_time，一个是set\_mmss。因为有的RTC硬件只计算秒数，不关心墙钟时间，所以如果是这样的RTC，必须实现set\_mmss来设置时间。  
六. rtc-sysfs.c 部分  
        这个部分主要是有关sysfs的操作。rtc-sysfs.c中定义了这样一个设备属性组，如下：

1. **static** **struct** device\_attribute rtc\_attrs[] = {
2. \_\_ATTR(name, S\_IRUGO, rtc\_sysfs\_show\_name, NULL),
3. \_\_ATTR(date, S\_IRUGO, rtc\_sysfs\_show\_date, NULL),
4. \_\_ATTR(time, S\_IRUGO, rtc\_sysfs\_show\_time, NULL),
5. \_\_ATTR(since\_epoch, S\_IRUGO, rtc\_sysfs\_show\_since\_epoch, NULL),
6. \_\_ATTR(max\_user\_freq, S\_IRUGO | S\_IWUSR, rtc\_sysfs\_show\_max\_user\_freq,
7. rtc\_sysfs\_set\_max\_user\_freq),
8. \_\_ATTR(hctosys, S\_IRUGO, rtc\_sysfs\_show\_hctosys, NULL),
9. { },
10. };

       这个属性组是在class.c的模块初始化函数中，由rtc\_sysfs\_init函数赋值给rtc\_class->dev\_attrs的，以后属于这个类的设备都会有这些属性。但是我们知道要想一个设备结构拥有一种属性，必须调用device\_create\_file，这样才会使这个属性出现在sysfs相关设备目录里。但是在这里的代码中只是给这个类的dev\_attrs域赋值了这个属性组指针，而没有调用device\_create\_file。我原来以为是在rtc\_device\_resgister函数中，由rtc\_sysfs\_add\_device完成这个工作，但是这个函数只是给设备添加了闹钟属性，并没有处理这个属性组。最后发现这个工作是由device\_register来完成的。这里的调用关系有点复杂：

device\_register调用device\_add

device\_add调用 device\_add\_attrs

 device\_add\_attrs调用device\_add\_attributes

device\_add\_attributes调用device\_create\_file来完成设备的属性设置的。

        设置完属性后，在/sys/class/rtc/rtc(n)的目录下就会出现name，date，time等文件，用户读这些文件的时候就会调用相应的函数。如读取name文件，就会调用rtc\_sysfs\_show\_name函数，这个函数也是在rtc-sysfs.c中实现的，作用是读取并显示时间。

七. rtc-proc.c   
       这个文件提供RTC的proc文件系统接口。proc文件系统是软件创建的文件系统，内核通过他向外界导出信息，下面的每一个文件都绑定一个函数，当用户读取这个文件的时候，这个函数会向文件写入信息。rtc-proc.c中初始化了一个文件操作：

1. **static** **const** **struct** file\_operations rtc\_proc\_fops = {
2. .open       = rtc\_proc\_open,
3. .read       = seq\_read,
4. .llseek     = seq\_lseek,
5. .release    = rtc\_proc\_release,
6. };

        RTC驱动在向RTC核心注册自己的时候，由注册函数rtc\_device\_resgister调用rtc\_proc\_add\_device来实现proc接口的初始化，这个函数如下定义：

1. **void** rtc\_proc\_add\_device(**struct** rtc\_device \*rtc)
2. {
3. **if** (rtc->id == 0)
4. proc\_create\_data("driver/rtc", 0, NULL, &rtc\_proc\_fops, rtc);
5. }

        他主要调用了proc\_create\_data。proc\_create\_data完成创建文件节点的作用，并将文件的操作函数与节点联系起来。调用这个函数后，在/proc/driver目录下就会有一个文件rtc，应用程序打开这个文件就会调用rtc\_proc\_open函数，这个函数如下定义：

1. **static** **int** rtc\_proc\_open(**struct** inode \*inode, **struct** file \*file)
2. {
3. **struct** rtc\_device \*rtc = PDE(inode)->data;
4. **if** (!try\_module\_get(THIS\_MODULE))
5. **return** -ENODEV;
6. **return** single\_open(file, rtc\_proc\_show, rtc);
7. }

        我们知道一个proc的文件必须与一个操作函数组成一个proc入口项，这个文件才能正常工作。这个函数最主要作用就是调用single\_open，创建一个proc文件入口项，使其操作函数是rtc\_proc\_show，并初始化seq\_file接口。rtc\_proc\_show函数如下定义：

1. **static** **int** rtc\_proc\_show(**struct** seq\_file \*seq, **void** \*offset)
2. {
3. **int** err;
4. **struct** rtc\_device \*rtc = seq->**private**;
5. **const** **struct** rtc\_class\_ops \*ops = rtc->ops;
6. **struct** rtc\_wkalrm alrm;
7. **struct** rtc\_time **tm**;
9. err = rtc\_read\_time(rtc, &**tm**);
10. **if** (err == 0) {
11. seq\_printf(seq,
12. "rtc\_time\t: %02d:%02d:%02d\n"
13. "rtc\_date\t: %04d-%02d-%02d\n",
14. **tm**.tm\_hour, **tm**.tm\_min, **tm**.tm\_sec,
15. **tm**.tm\_year + 1900, **tm**.tm\_mon + 1, **tm**.tm\_mday);
16. }
18. err = rtc\_read\_alarm(rtc, &alrm);
19. **if** (err == 0) {
20. seq\_printf(seq, "alrm\_time\t: ");
21. **if** ((unsigned **int**)alrm.time.tm\_hour <= 24)
22. seq\_printf(seq, "%02d:", alrm.time.tm\_hour);
23. **else**
24. seq\_printf(seq, "\*\*:");
25. **if** ((unsigned **int**)alrm.time.tm\_min <= 59)
26. seq\_printf(seq, "%02d:", alrm.time.tm\_min);
27. **else**
28. seq\_printf(seq, "\*\*:");
29. **if** ((unsigned **int**)alrm.time.tm\_sec <= 59)
30. seq\_printf(seq, "%02d\n", alrm.time.tm\_sec);
31. **else**
32. seq\_printf(seq, "\*\*\n");
34. seq\_printf(seq, "alrm\_date\t: ");
35. **if** ((unsigned **int**)alrm.time.tm\_year <= 200)
36. seq\_printf(seq, "%04d-", alrm.time.tm\_year + 1900);
37. **else**
38. seq\_printf(seq, "\*\*\*\*-");
39. **if** ((unsigned **int**)alrm.time.tm\_mon <= 11)
40. seq\_printf(seq, "%02d-", alrm.time.tm\_mon + 1);
41. **else**
42. seq\_printf(seq, "\*\*-");
43. **if** (alrm.time.tm\_mday && (unsigned **int**)alrm.time.tm\_mday <= 31)
44. seq\_printf(seq, "%02d\n", alrm.time.tm\_mday);
45. **else**
46. seq\_printf(seq, "\*\*\n");
47. seq\_printf(seq, "alarm\_IRQ\t: %s\n",
48. alrm.enabled ? "yes" : "no");
49. seq\_printf(seq, "alrm\_pending\t: %s\n",
50. alrm.pending ? "yes" : "no");
51. }
53. seq\_printf(seq, "24hr\t\t: yes\n");
55. **if** (ops->proc)
56. ops->proc(rtc->dev.parent, seq);
58. **return** 0;
59. }

        这个函数就是最后给用户显示信息的函数了，可以看出他通过调用rtc\_deivce中的操作函数，读取时间，日期和一些其他的信息显示给用户。   
六. 总结

        RTC核心使底层硬件对用户来说是透明的，并且减少了编写驱动程序的工作量。RTC新的驱动接口提供了更多的功能，使系统可以同时存在多个RTC。/dev，sysfs，proc这三种机制的实现使得应用程序能灵活的使用RTC，RTC核心虽然表面上看上去很简单，但是还是涉及到很多知识，有些东西书上讲的还是不够详细，还需要通过分析代码加深理解。 另外RTC核心代码的组织方式也值得学习，不同功能的代码放在不同的文件中，简单明了。