android alarm driver &framework 关键流程

时间  2016-07-01

标签 [android](http://www.voidcn.com/tag/android) [kernel](http://www.voidcn.com/tag/kernel) [alarm](http://www.voidcn.com/tag/alarm) [framework](http://www.voidcn.com/tag/framework) [driver](http://www.voidcn.com/tag/driver) 栏目 [Android](http://www.voidcn.com/column/android)

原文   [http://blog.csdn.net/songyinzhong/article/details/51799507](javascript:void())

android alarm driver如下：   
alarm-dev.c@\kernel\drivers\staging\android

static int \_\_init alarm\_dev\_init(void)

{

int err*;*

int i*;*

err = misc\_register(&alarm\_device)*;*

if (err)

return err*;*

alarm\_init(&alarms[ANDROID\_ALARM\_RTC\_WAKEUP]**.u.alrm**,

ALARM\_REALTIME, devalarm\_alarmhandler)*;*

hrtimer\_init(&alarms[ANDROID\_ALARM\_RTC]**.u.hrt**,

CLOCK\_REALTIME, HRTIMER\_MODE\_ABS)*;*

alarm\_init(&alarms[ANDROID\_ALARM\_ELAPSED\_REALTIME\_WAKEUP]**.u.alrm**,

ALARM\_BOOTTIME, devalarm\_alarmhandler)*;*

hrtimer\_init(&alarms[ANDROID\_ALARM\_ELAPSED\_REALTIME]**.u.hrt**,

CLOCK\_BOOTTIME, HRTIMER\_MODE\_ABS)*;*

hrtimer\_init(&alarms[ANDROID\_ALARM\_SYSTEMTIME]**.u.hrt**,

CLOCK\_MONOTONIC, HRTIMER\_MODE\_ABS)*;*

for (i = 0*; i < ANDROID\_ALARM\_TYPE\_COUNT; i++) {*

alarms[i]**.type** = i*;*

if (!is\_wakeup(i))

alarms[i]**.u.hrt.function** = devalarm\_hrthandler*;*

}

wakeup\_source\_init(&alarm\_wake\_lock, "alarm")*;*

return 0*;*

}

driver init主要做了三件事，   
1. 注册android alarm 设备：alarm\_device。   
2. 向linux alarm子系统注册alarm处理函数devalarm\_alarmhandler，如果是允许唤醒系统的alarm，还会注册devalarm\_hrthandler。当系统linux alarm到期时， 会回调这两个handler处理。   
3. 初始化alarm 的wakeup source。

再细看handler里的处理：

static void devalarm\_triggered(struct devalarm \*alarm)

{

unsigned long flags;

uint32\_t alarm\_type\_mask = 1U << alarm->**type;**

alarm\_dbg(INT, "%s: type %d\n", \_\_func\_\_, alarm->**type);**

spin\_lock\_irqsave(&alarm\_slock, flags);

if (alarm\_enabled & alarm\_type\_mask) {

\_\_pm\_wakeup\_event(&alarm\_wake\_lock, 5000); */\* 5secs \*/*

alarm\_enabled &= ~alarm\_type\_mask;

alarm\_pending |= alarm\_type\_mask;

wake\_up(&alarm\_wait\_queue);

}

spin\_unlock\_irqrestore(&alarm\_slock, flags);

}

主要做了一个动作是唤醒alarm的等待队列alarm\_wait\_queue， 后面分析再看如何与这里对接 。

先看上层AlarmManagerService初始化的函数，   
主要有两步， 一是call native init， 二是启动thread一直等alarm消息：

public AlarmManagerService(Context context) {

mContext = context;

mDescriptor = init();

....

if (mDescriptor != -1) {

mWaitThread.start();

} else {

Slog.w(TAG, "Failed to open alarm driver. Falling back to a handler.");

}

}

其中AlarmManagerService是由System\_server init时new出来：   
initAndLoop（）@SystemServer.java

Slog**.i**(TAG, "Alarm Manager")*;*

alarm = new AlarmManagerService(context)*;*

ServiceManager**.addService**(Context**.ALARM**\_SERVICE, alarm)*;*

init对应初始化，完成alarm设备打开：   
com\_android\_server\_AlarmManagerService.cpp@ \frameworks\base\services\jni

static jint android\_server\_AlarmManagerService\_init(JNIEnv\* env, jobject obj)

{

return open("/dev/alarm", O\_RDWR);

}

mwaitThread start后，一直循环block等待alarm driver消息， :

public void run()

{

ArrayList<Alarm> triggerList = new ArrayList<Alarm>();

while (true)

{

int result = waitForAlarm(mDescriptor);

native实现 ：

static jint android\_server\_AlarmManagerService\_waitForAlarm(JNIEnv\* env, jobject obj, jint fd)

{

int result = 0;

do

{

result = ioctl(fd, ANDROID\_ALARM\_WAIT);

} while (result < 0 && errno == EINTR);

if (result < 0)

{

ALOGE("Unable to wait on alarm: %s\n", strerror(errno));

return 0;

}

return result;

}

ioctl对应driver逻辑：

static long alarm\_do\_ioctl(struct file \*file, unsigned int cmd,

struct timespec \*ts)

{

...

case ANDROID\_ALARM\_WAIT:

rv = alarm\_wait();

break;

alarm\_wait实现 如下， 此处会一直wait等待wake up alarm\_wait\_queue状态 ， 就接上了最开始提到的向linux alarm子系统注册的alarm handler来wakeup的地方. 从而完成了具体alarm到期事件 向framework的通知 。

static int alarm\_wait(void)

{

unsigned long flags;

int rv = 0;

spin\_lock\_irqsave(&alarm\_slock, flags);

alarm\_dbg(IO, "alarm wait\n");

if (!alarm\_pending && wait\_pending) {

\_\_pm\_relax(&alarm\_wake\_lock);

wait\_pending = 0;

}

spin\_unlock\_irqrestore(&alarm\_slock, flags);

rv = wait\_event\_interruptible(alarm\_wait\_queue, alarm\_pending);

if (rv)

return rv;

spin\_lock\_irqsave(&alarm\_slock, flags);

rv = alarm\_pending;

wait\_pending = 1;

alarm\_pending = 0;

spin\_unlock\_irqrestore(&alarm\_slock, flags);

return rv;

}

**相关文章**

* 1. [Android学习之Alarm driver](http://www.voidcn.com/article/p-qzdcmbxg-qy.html)
* 2. [Android Framework层Power键关机流程](http://www.voidcn.com/article/p-xitfewuw-mk.html)
* 3. [Android Framework层Power键关机流程（二，关机流程）](http://www.voidcn.com/article/p-nrbplyqz-ex.html)
* 4. [Alarm设置流程](http://www.voidcn.com/article/p-cptbwzvj-beh.html)
* 5. [framework按键处理流程](http://www.voidcn.com/article/p-mxtdcdbx-bbd.html)
* 6. [Android framework层 按键的处理流程](http://www.voidcn.com/article/p-kgfhvhvb-bht.html)
* 7. [USB gadget driver framework](http://www.voidcn.com/article/p-zjuaaoqs-zd.html)
* 8. [Android 关机流程分析-----（1）Framework层](http://www.voidcn.com/article/p-utdaswft-beq.html)
* 9. [android关机流程-framework部分](http://www.voidcn.com/article/p-uurohsoy-dx.html)
* 10. [Android 关机流程 从kernel到framework](http://www.voidcn.com/article/p-awrbztla-vt.html)
* [更多相关文章...](http://www.voidcn.com/relative/p-vmkhfavr-da.html)

# Android学习之Alarm driver

置顶 2011年11月21日 23:17:38 [lail3344](https://me.csdn.net/Melody_lu123) 阅读数：8520 标签： [android](http://so.csdn.net/so/search/s.do?q=android&t=blog)[interface](http://so.csdn.net/so/search/s.do?q=interface&t=blog)[class](http://so.csdn.net/so/search/s.do?q=class&t=blog)[makefile](http://so.csdn.net/so/search/s.do?q=makefile&t=blog)[timer](http://so.csdn.net/so/search/s.do?q=timer&t=blog)更多

个人分类： [driver](https://blog.csdn.net/Melody_lu123/article/category/926968)[android](https://blog.csdn.net/Melody_lu123/article/category/877908)[kernel](https://blog.csdn.net/Melody_lu123/article/category/877909)[linux](https://blog.csdn.net/Melody_lu123/article/category/910458)

Updated: 2013/03/18， 刚有时间浏览了最新的Android builder submit 2013的相关slides，其中有关于最新的android 对kernel的upstream的状态的总结，大家可以看看  
<http://events.linuxfoundation.org/images/stories/slides/abs2013_stultz.pdf>

Alarm正在staging中，不久应该也会进入main stream

Alarm是Android提供的一个硬件时钟,基于内核的rtc机制完成.

相关知识点:

内核中的各种时间函数

ktime\_get\_ts 参见POSIX.1b clock types, 可以从<The Linux Programming Interface>的第23.5.1找到相应说明  
    当应用程序希望不会受到手动的改变系统时钟的影响时最有用.在Linux中,这个时间反应了从系统启动开始流逝的时间.  
  
getnstimeofday 类似与user space的gettimeofday, 反映的是从1970年开始的毫秒

还有一些各种时间格式之间的转换函数在[timer.c](http://lxr.linux.no/linux+v2.6.35.14/kernel/time.c#L416)

一些有用的资源,在需要用到时钟那进行控制的事情,一定要记着选择正确的精度和函数,并多参考现有的其它driver.

Documentation/timers/timers-howto.txt

<http://www.ibm.com/developerworks/linux/library/l-timers-list/index.html?ca=drs->

<http://stackoverflow.com/questions/4655711/measuring-execution-time-of-a-function-inside-linux-kernel><http://stackoverflow.com/questions/6360210/androidlinux-uptime-using-clock-monotonic>其中提到了最新引入内核的CLOCK\_BOOTTIME.<https://lwn.net/Articles/429925/>全面介绍了最新的内核中时钟的概念, 也提到了Android的alarm timer的实现

关于driver初始化的优先级

分为两个层次:initcall, Makefile.同样的initcal中以Makefile中的顺序排序,如果不在同一个initcall的层次,则以initcall的顺序调用. initcal的顺序参见[init.h](http://lxr.linux.no/linux+v2.6.35.14/include/linux/init.h#L131)

...

#define [late\_initcall](http://lxr.linux.no/linux+v2.6.35.14/+code=late_initcall)([fn](http://lxr.linux.no/linux+v2.6.35.14/+code=fn)) [\_\_define\_initcall](http://lxr.linux.no/linux+v2.6.35.14/+code=__define_initcall)(*"7"*,[fn](http://lxr.linux.no/linux+v2.6.35.14/+code=fn),7)

..

 关于alarm的一些总体认识和定义,参照include/linux/android\_alarm.h中的描述:

alarm提供的接口与hrtimer的接口类似,增加了从suspend状态wakeup的功能.并且提供了一个给周期性timer使用的递减的实时时钟,确保它在系统suspend的时候或者当wall time被改变的时候依然能够继续工作.

作为platform驱动被platform\_driver\_register所注册.  
使用了WAKE\_LOCK\_SUSPEND的wake lock来保证在系统不会进入full的suspend状态,从而能够提供比较好的低延迟.  
  
Android的alarm提供了两个设备

一个alarm的platform driver(alarm.c)

关于platform driver的相关理解参见<Device Driver学习之platform driver是神马>

其中值得注意的是, 它通过rtc\_alarm\_interface这个一个class interface,使得alarm使用了rtc提供的一些接口. 这里使用msm\_rtc这个rtc设备作为alarm设备的时钟源.**这里就体现了一个重要的知识点,如何使用既有的设备构建新的设备(即,使用class\_interface提供的机制,使得platform alarm设备在msm\_rtc设备被添加到system的时候,通过platform的core通知到platform alarm设备class\_interface的add\_dev,然后把platform alarm设备注册入添加到系统中)!!!**  
[Class Device http://www.linuxjournal.com/node/6872/print](http://www.linuxjournal.com/node/6872/print), 这篇文章虽然有点老,但是对于基本概念还是阐述的挺好的.

# dmesg | grep "using rtc device"

<6>[    7.471466] using rtc device, msm\_rtc, for alarms

<6>[    7.471618] rs30000048:00010000 rs30000048:00010000: rtc core: registered msm\_rtc as rtc0

一个暴露给用户使用的接口misc的alarm接口(alarm-dev.c)

其中定义了一些列的ioctl,来操纵platform alarm driver提供的功能.

哪些地方使用到了alarm提供的接口:

通过grep android的代码,如下地方使用了alarm的user接口:

development/simulator/wrapsim/FakeDev.c: \*  /dev/alarm  
frameworks/base/cmds/runtime/main\_runtime.cpp:    fd = open("/dev/alarm", O\_RDWR);  
frameworks/base/services/jni/com\_android\_server\_AlarmManagerService.cpp:    return open("/dev/alarm", O\_RDWR);  
frameworks/base/libs/utils/SystemClock.cpp:    fd = open("/dev/alarm", O\_RDWR);  
frameworks/base/libs/utils/SystemClock.cpp:        int fd = open("/dev/alarm", O\_RDONLY);

system/core/toolbox/date.c:    fd = open("/dev/alarm", O\_RDWR);  
system/core/toolbox/date.c:        fd = open("/dev/alarm", O\_RDWR);  
system/core/toolbox/uptime.c:    fd = open("/dev/alarm", O\_RDONLY);  
system/core/toolbox/alarm.c:    afd = open("/dev/alarm", O\_RDWR);  
system/core/rootdir/ueventd.rc:/dev/alarm                0664   system     radio

用户层可以使用的由alarm驱动提供的四种硬件时钟:

RTC\_WAKEUP(ANDROID\_ALARM\_RTC\_WAKEUP):在指定的时刻（设置Alarm的时候），唤醒设备来触发Intent。

RTC(ANDROID\_ALARM\_RTC):在一个显式的时间触发Intent，但不唤醒设备。

ELAPSED\_REALTIME(ANDROID\_ALARM\_ELAPSED\_REALTIME):从设备启动后，如果流逝的时间达到总时间，那么触发Intent，但不唤醒设备。流逝的时间包括设备睡眠的任何时间。注意一点的是，时间流逝的计算点是自从它最后一次启动算起。

ELAPSED\_REALTIME\_WAKEUP(ANDROID\_ALARM\_ELAPSED\_REALTIME\_WAKEUP):从设备启动后，达到流逝的总时间后，如果需要将唤醒设备并触发Intent。

以上4个时钟都是通过alarm service提供给用户使用的,它们通过对应的JNI侧()来获得alarm驱动提供的硬件时钟.  
  
还有一个在alarm驱动中定义的硬件时钟是ANDROID\_ALARM\_SYSTEMTIME, 它没有暴露给java层的用户来使用. 它是返回上文所提到的ktime\_get\_ts函数返回的值.目前还没有用户侧的程序需要使用它.

另外,值得注意的是Android提供了对alarm设备的测试程序(mydroid/system/core/toolbox/alarm.c), 但是,没有放在编译的Android.mk中,所以需要修改toolbox的Android.mk才能把它编译出来.

# https://www.linuxidc.com/Linux/2013-08/88965.htmAndroid RTC 自下而上分析

|  |  |  |
| --- | --- | --- |
| [日期：2013-08-17] | 来源：Linux社区  作者：邹南 | [字体：[大](javascript:ContentSize(16)) [中](javascript:ContentSize(0)) [小](javascript:ContentSize(12))] |

在[Android](http://www.linuxidc.com/topicnews.aspx?tid=11)中，定时alarm功能是很常用的，现在来分析下怎么实现的，这里将采用自下而上的方式讲解。

**相关阅读：**

时钟RTC驱动分析 [http://www.linuxidc.com/Linux/2013-06/86043.htm](https://www.linuxidc.com/Linux/2013-06/86043.htm)

S3C2440上RTC时钟驱动开发实例分析 [http://www.linuxidc.com/Linux/2013-06/86042.htm](https://www.linuxidc.com/Linux/2013-06/86042.htm)

实时时钟RTC之hwclock简介 [http://www.linuxidc.com/Linux/2012-09/69915.htm](https://www.linuxidc.com/Linux/2012-09/69915.htm)

S3C6410实时时钟RTC 秒字符设备 [http://www.linuxidc.com/Linux/2012-09/69906.htm](https://www.linuxidc.com/Linux/2012-09/69906.htm)

Linux驱动分析之RTC详解 [http://www.linuxidc.com/Linux/2012-08/67808.htm](https://www.linuxidc.com/Linux/2012-08/67808.htm)

arm： cortex-a8  
        Board: FS\_S5PC100  
        Linux: 2.6.29  
        Android: 2.1

1、 RTC控制器

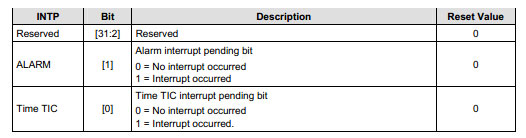
1.1 feature

实时时钟（RTC）单元可以通过备用电池供电，因此，即使系统电源关闭，它也可以继续工作。RTC 可以通过STRB/LDRB 指令将8 位BCD 码数据送至CPU。这些BCD 数据包括秒，分，时，日期，星期，月和年。RTC 单元通过一个外部的32.768KHz晶振提供时钟。RTC具有定时报警的功能。RTC 控制器功能说明：

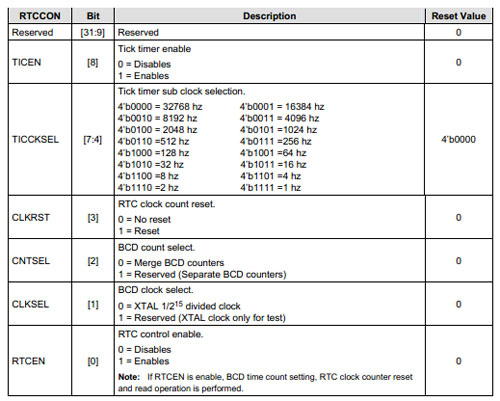
时钟数据采用BCD 编码  
        能够对闰年的年月日进行自动处理  
        具有告警功能，当系统处于关机状态时，能产生告警中断;  
        具有独立的电源输入  
        提供毫秒级时钟中断，该中断可用于作为嵌入式操作系统的内核时钟

1.2 register description

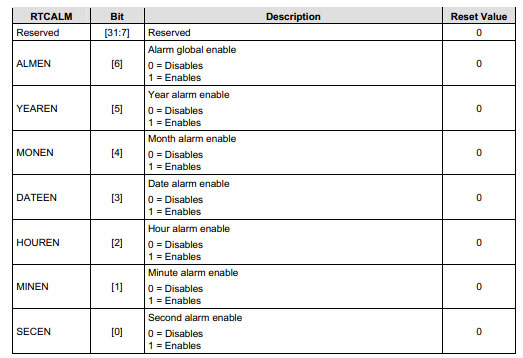
(1) RTC 中断挂起寄存器



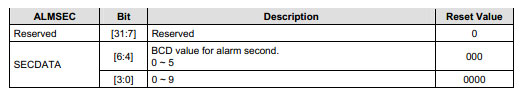
（2） RTC控制器寄存器



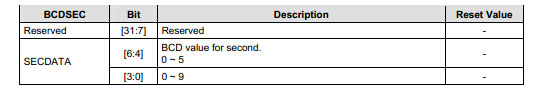
（3）RTC报警使能寄存器



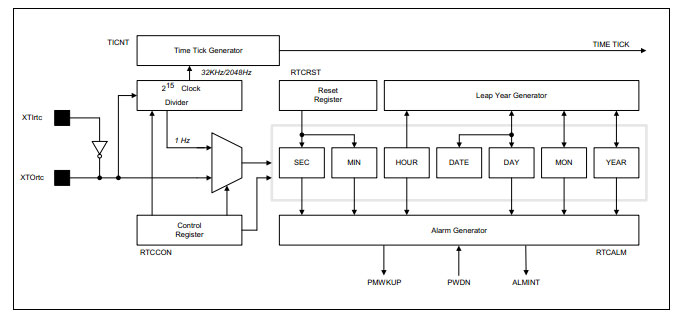
（4） alarm 值寄存器 （SEC/MIN/HOUR/ DATE/DAY/MON/YEAR）



（5）BCD值寄存器（SEC/MIN/HOUR/ DATE/DAY/MON/YEAR）



1.3 如何program?



从上图可以看到RTC有三个功能块，一个是计时器，一个是定时器，一个是“嘀嗒”产生器，后两个都可以产生中断，前者只要配置后，在power-off的情况下也会计时，当然要有后备电池的供电才可能。

1.计时器

计时器只要你配置好BCD寄存器即可，也就是填入当前的时间，年月日，时分秒。星期这些数据即可。当然在填写之前应该先将控制器寄存器使能。

2.定时器

定时器和计时器不同的地方在于，定时器的值寄存器填写的是触发中断的时间，当BCD计时器的值增加到定时器的值时，就会产生一个中断。该中断可以再掉电模式下或者是普通模式。

3.“嘀嗒”产生器

该部分功能是为OS所提供的功能，初始化后就可以，该值可提供给RTOS。

2、 linux driver

上面只是把RTC硬件给过了一遍，现在往上升一级，开始看下linux下是如何driver一个RTC的。假设你的linux kernel source code的主目录名字为LINUX。

先关注一个目录：LINUX/driver/rtc/

在该目录下，有一堆以rtc-为前缀的文件，这些文件都是各种板子上用的rtc底层驱动代码，我们要看的只有3个，rtc-s3c.c ,alarm.c, alarm-dev.c 。

看下第一个，rtc-s3c.c 是三星产的arm芯片所专用的一个rtc驱动，要说专用也不为过，看看编写者就知道了，看看怎么实现：

它用的是平台设备驱动，

static struct platform\_driver s3c2410\_rtc\_driver = {  
                .probe = s3c\_rtc\_probe,  
                .remove = \_\_devexit\_p(s3c\_rtc\_remove),  
                .suspend = s3c\_rtc\_suspend,  
                .resume = s3c\_rtc\_resume,  
                .driver = {  
                        .name = "s3c2410-rtc",  
                        .owner = THIS\_MODULE,  
                },  
        };

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,  
                .irq\_set\_freq = s3c\_rtc\_setfreq,  
                .irq\_set\_state = s3c\_rtc\_setpie,  
                .proc = s3c\_rtc\_proc,  
        };

看这两个结构体，我认为就已经达到目的，第一个结构体是平台设备中的driver部分，也就是s3c\_rtc\_probe,是个很重要的函数，在这里面，第二个结构体被顺利注册进rtc子系统。Rtc的所用到的结构体被定义在，LINUX/include/linux/rtc.h里面。

struct rtc\_device  
        {  
                struct device dev;  
                struct module \*owner;

        int id;  
                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  
        };

这个结构体是核心部分，内核中就是靠它传递信息，不管在哪使用，都要靠它间接的调用底层信息。比如在alarm.c 中。

alarm\_ioctl这个函数中，多次使用了rtc\_set\_time/rtc\_get\_time，这些函数虽然是定义在rtc目录下的interface.c 中，但实质还是rtc-s3c.c中结构体 rtc\_class\_ops所指过去的函数。

那么我可以告诉你了，为什么多了一个alarm.c ，因为在android中它为了使得平台无关性提高，因此大量的增加过渡代码层，HAL就是这种性质的存在。  
alarm.c在用户空间中会多一个/dev/alarm 节点，而rtc-s3c.c 会产生/dev/rtc这样的节点。

Android在HAL层中，是对/dev/alarm这个结点进行操作。

# Android RTC 自下而上分析

|  |  |  |
| --- | --- | --- |
| [日期：2013-08-17] | 来源：Linux社区  作者：邹南 | [字体：[大](javascript:ContentSize(16)) [中](javascript:ContentSize(0)) [小](javascript:ContentSize(12))] |

3、JNI 的实现

直接看代码

static jint [Android](http://www.linuxidc.com/topicnews.aspx?tid=11)\_server\_AlarmManagerService\_init(JNIEnv\* env, jobject obj)  
        {  
                return open("/dev/alarm", O\_RDWR);  
        }

static void android\_server\_AlarmManagerService\_set(JNIEnv\* env, jobject obj, jint fd, jint type, jlong nanoseconds)  
        {  
                struct timespec ts;  
                ts.tv\_sec = NANOSECONDS\_TO\_SECONDS(nanoseconds);  
                ts.tv\_nsec = nanoseconds - SECONDS\_TO\_NANOSECONDS(ts.tv\_sec);

                int result = ioctl(fd, ANDROID\_ALARM\_SET(type), &ts);

        }

static JNINativeMethod sMethods[] = {  
                /\* name, signature, funcPtr \*/  
                {"init", "()I", (void\*)android\_server\_AlarmManagerService\_init},  
                {"close", "(I)V", (void\*)android\_server\_AlarmManagerService\_close},  
                {"set", "(IIJ)V", (void\*)android\_server\_AlarmManagerService\_set}，  
        };

颜色部分说明了一个问题，那就是android也就那么回事，JNI就是按照特定写法的JAVA版的linux c应用程序。完事。如果你觉得不是这么回事，那么你就有事了，去复习复习linux文件IO，看看如何用一个文件描述符打开设备，操作设备等等。

4、 framework层

frameworks/base/services/java/com/android/server/AlarmManagerService.java   
        frameworks/base/core/java/android/app/AlarmManager.java

下面的是直接提供给app层的API接口，它是AlarmManagerService.java的一个封装。

这里只是简单的解释下service到底在此做什么了。

其实也没做什么，仅仅是把上面分析的JNI拿来在此调用一下而已。然后包装一下，将功能实现得更完美些。

下面是 AlarmManagerService这个类中摘出来的小段：

private native int init();  
        private native void close(int fd);  
        private native void set(int fd, int type, long nanoseconds);  
        private native int waitForAlarm(int fd);  
        private native int setKernelTimezone(int fd, int minuteswest);

这些就是JNI实现过来的接口。呵呵。

5、APP层

packages/apps/AlarmClock/src/com/android/alarmclock/ 这个目录下，就是系统自带定时器的源代码，比如Alarms.java 中：第一个导入的包就是  
import android.app.AlarmManager; 怎样，到现在是否感受到了android从下至上分析的快感？

# Android RTC 自下而上分析

|  |  |  |
| --- | --- | --- |
| [日期：2013-08-17] | 来源：Linux社区  作者：邹南 | [字体：[大](javascript:ContentSize(16)) [中](javascript:ContentSize(0)) [小](javascript:ContentSize(12))] |

很久之前写的一个流程文档，从上层界面一直调用到内核的过程，最近同事跟我要，我看了下又在整理了下，纯属个人分析（不过都运行验证过），不对的请大牛指出

Alarm 调用流程，alarm的流程实现了从上层应用一直到下面driver的调用流程，下面简单阐述：

涉及代码;  
./packages/apps/DeskClock/src/com/[Android](http://www.linuxidc.com/topicnews.aspx?tid=11)/deskclock/Alarms.java  
./frameworks/base/core/java/android/app/AlarmManager.java  
./frameworks/base/services/java/com/android/server/AlarmManagerService.java  
./frameworks/base/services/jni/com\_android\_server\_AlarmManagerService.cpp  
./kernel/kernel/drivers/rtc/alarm-dev.c  
./kernel/kernel/include/linux/android\_alarm.h  
./kernel/kernel/drivers/rtc/alarm.c  
./kernel/kernel/drivers/rtc/interface.c  
./kernel/kernel/drivers/rtc/rtc-pcf8563.c

./packages/apps/DeskClock/src/com/android/deskclock/AlarmReceiver.java

./kernel/arch/arm/configs/mmp2\_android\_defconfig  
./kernel/kernel/kernel/.config

点击Clock 应用程序，然后设置新闹钟，会调到  Alarms.java  里面的  
public static long setAlarm(Context context, Alarm alarm) {  
....  
setNextAlert(context);  
....  
}  
然后这里面也会调用到    
public static void setNextAlert(final Context context) {  
        if (!enableSnoozeAlert(context)) {  
            Alarm alarm = calculateNextAlert(context);  //new 一个新的alarm  
            if (alarm != null) {  
                enableAlert(context, alarm, alarm.time);  
            } else {  
                disableAlert(context);  
            }  
        }  
    }  
然后继续调用到   
private static void enableAlert(Context context, final Alarm alarm, final long atTimeInMillis) {  
.......  
am.set(AlarmManager.RTC\_WAKEUP, atTimeInMillis, sender);  //这里是RTC\_WAKEUP, 这就保证了即使系统睡眠了，都能唤醒，闹钟工作（android平台关机闹钟好像不行）  
.....  
}

然后就调用到了AlarmManager.java 里面方法  
    public void set(int type, long triggerAtTime, PendingIntent operation) {  
        try {  
            mService.set(type, triggerAtTime, operation);  
        } catch (RemoteException ex) {  
        }  
    }

然后就调用到了AlarmManagerService.java  里面方法  
public void set(int type, long triggerAtTime, PendingIntent operation) {  
        setRepeating(type, triggerAtTime, 0, operation);  
    }

然后继续调用  
public void setRepeating(int type, long triggerAtTime, long interval,   
            PendingIntent operation) {  
.....  
synchronized (mLock) {  
            Alarm alarm = new Alarm();  
            alarm.type = type;  
            alarm.when = triggerAtTime;  
            alarm.repeatInterval = interval;  
            alarm.operation = operation;

            // Remove this alarm if already scheduled.  
            removeLocked(operation);

            if (localLOGV) Slog.v(TAG, "set: " + alarm);

            int index = addAlarmLocked(alarm);  
            if (index == 0) {  
                setLocked(alarm);  
            }  
        }  
    }

然后就调用到  
private void setLocked(Alarm alarm)  
    {  
    ......  
    set(mDescriptor, alarm.type, alarmSeconds, alarmNanoseconds);  //mDescriptor  这里的文件是 /dev/alarm  
    .....  
}

这里就调用到jni了  
private native void set(int fd, int type, long seconds, long nanoseconds);

这就调用到了com\_android\_server\_AlarmManagerService.cpp 里面  
static JNINativeMethod sMethods[] = {  
    /\* name, signature, funcPtr \*/  
    {"init", "()I", (void\*)android\_server\_AlarmManagerService\_init},  
    {"close", "(I)V", (void\*)android\_server\_AlarmManagerService\_close},  
    {"set", "(IIJJ)V", (void\*)android\_server\_AlarmManagerService\_set},  
    {"waitForAlarm", "(I)I", (void\*)android\_server\_AlarmManagerService\_waitForAlarm},  
    {"setKernelTimezone", "(II)I", (void\*)android\_server\_AlarmManagerService\_setKernelTimezone},  
};

set 对应的是android\_server\_AlarmManagerService\_set， 具体是  
static void android\_server\_AlarmManagerService\_set(JNIEnv\* env, jobject obj, jint fd, jint type, jlong seconds, jlong nanoseconds)  
{  
#if HAVE\_ANDROID\_OS  
    struct timespec ts;  
    ts.tv\_sec = seconds;  
    ts.tv\_nsec = nanoseconds;  
      
    int result = ioctl(fd, ANDROID\_ALARM\_SET(type), &ts);  
    if (result < 0)  
    {  
        LOGE("Unable to set alarm to %lld.%09lld: %s\n", seconds, nanoseconds, strerror(errno));  
    }  
#endif  
}

然后ioctl 就调用到了alarm-dev.c  
static long alarm\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg)  
{  
....  
    case ANDROID\_ALARM\_SET(0):  
        if (copy\_from\_user(&new\_alarm\_time, (void \_\_user \*)arg,  
            sizeof(new\_alarm\_time))) {  
            rv = -EFAULT;  
            goto err1;  
        }  
from\_old\_alarm\_set:  
        spin\_lock\_irqsave(&alarm\_slock, flags);  
        pr\_alarm(IO, "alarm %d set %ld.%09ld\n", alarm\_type,  
            new\_alarm\_time.tv\_sec, new\_alarm\_time.tv\_nsec);  
        alarm\_enabled |= alarm\_type\_mask;  
        alarm\_start\_range(&alarms[alarm\_type],  
            timespec\_to\_ktime(new\_alarm\_time),  
            timespec\_to\_ktime(new\_alarm\_time));  
        spin\_unlock\_irqrestore(&alarm\_slock, flags);  
        if (ANDROID\_ALARM\_BASE\_CMD(cmd) != ANDROID\_ALARM\_SET\_AND\_WAIT(0)  
            && cmd != ANDROID\_ALARM\_SET\_AND\_WAIT\_OLD)  
            break;  
        /\* fall though \*/  
....

case ANDROID\_ALARM\_SET\_RTC:  
        if (copy\_from\_user(&new\_rtc\_time, (void \_\_user \*)arg,  
            sizeof(new\_rtc\_time))) {  
            rv = -EFAULT;  
            goto err1;  
        }  
        rv = alarm\_set\_rtc(new\_rtc\_time);  
        spin\_lock\_irqsave(&alarm\_slock, flags);  
        alarm\_pending |= ANDROID\_ALARM\_TIME\_CHANGE\_MASK;  
        wake\_up(&alarm\_wait\_queue);  
        spin\_unlock\_irqrestore(&alarm\_slock, flags);  
        if (rv < 0)  
            goto err1;  
        break;  
....  
}

然后这边就调用到了alarm\_start\_range  设置闹钟，  alarm\_set\_rtc  设置RTC  
这两个函数在 android\_alarm.h 声明，在 alarm.c 里实现  
这是android\_alarm.h 里面的声明  
void alarm\_start\_range(struct alarm \*alarm, ktime\_t start, ktime\_t end);  
int alarm\_try\_to\_cancel(struct alarm \*alarm);  
int alarm\_cancel(struct alarm \*alarm);  
ktime\_t alarm\_get\_elapsed\_realtime(void);

/\* set rtc while preserving elapsed realtime \*/  
int alarm\_set\_rtc(const struct timespec ts);

下面看alarm.c 里面实现：  
int alarm\_set\_rtc(struct timespec new\_time)  
{  
....  
ret = rtc\_set\_time(alarm\_rtc\_dev, &rtc\_new\_rtc\_time);  
....  
}

alarm.c  里面实现了 alarm\_suspend  alarm\_resume 函数  
就是如果系统没有suspend的时候，设置闹钟并不会往rtc 芯片的寄存器上写数据，因为不需要唤醒系统，所以闹钟数据时间什么的就通过上层写到设备文件/dev/alarm  
里面就可以了，AlarmThread 会不停的去轮寻下一个时间有没有闹钟，直接从设备文件 /dev/alarm 里面读取  
第二种，系统要是进入susupend的话，alarm 的alarm\_suspend  就会写到下层的rtc芯片的寄存器上去， 然后即使系统suspend之后，闹钟通过rtc 也能唤醒系统

这里就调用到了interface.c 里面  //这里面 int rtc\_set\_alarm(struct rtc\_device \*rtc, struct rtc\_wkalrm \*alarm) 差不多 也是跟下面一样  
int rtc\_set\_time(struct rtc\_device \*rtc, struct rtc\_time \*tm)  
{  
....  
    err = rtc->ops->set\_time(rtc->dev.parent, tm);  
....  
}

然后set\_time 就看到具体的是那个RTC芯片，这边我们是rtc-pcf8563.c  
static const struct rtc\_class\_ops pcf8563\_rtc\_ops = {  
    .read\_time    = pcf8563\_rtc\_read\_time,  
    .set\_time    = pcf8563\_rtc\_set\_time,  
    .read\_alarm    = pcf8563\_rtc\_read\_alarm,  
    .set\_alarm    = pcf8563\_rtc\_set\_alarm,  
};  
然后就到了  
static int pcf8563\_rtc\_set\_time(struct device \*dev, struct rtc\_time \*tm)  
{  
    unsigned char buf[TIME\_NUM];  
    int ret;

    ret = data\_calc(buf, tm, TIME\_NUM);  
    if (ret < 0)  
        goto out;  
    ret = i2c\_smbus\_write\_i2c\_block\_data(pcf8563\_info->client, PCF8563\_RTC\_SEC, TIME\_NUM, buf);  //这边就调用i2c统一接口，往pcf8563rtc芯片寄存器里面写出数据  
out:  
    return ret;  
}

到此，闹钟时间就已经写到rtc 芯片的寄存器里面，第二个参数就是寄存器的名字，后面的buf就是要写入的时间，rtc芯片是额外供电的，所以系统suspend之后，系统kernel都关了，但是rtc里面还有电，寄存器里面数据还是有的（掉电就会丢失数据），所以闹钟到了，通过硬��中断机制就可以唤醒系统。

上面那个rtc下面有几十个rtc芯片驱动代码，没有结构基本一样，都有基本操作函数，注册函数，都是对各自芯片上特有的寄存器操作，为什么调用的是pcf8563rtc呢？这个要看你系统用的是那个芯片，这个可以通过./kernel/kernel/kernel/.config  查看，这边的pcf8563rtc  是当前系统正在使用的芯片型号    
# CONFIG\_RTC\_DRV\_ISL1208 is not set  
# CONFIG\_RTC\_DRV\_X1205 is not set  
CONFIG\_RTC\_DRV\_PCF8563=y  
# CONFIG\_RTC\_DRV\_PCF8583 is not set  
# CONFIG\_RTC\_DRV\_M41T80 is not set

下面是系统唤醒之后，闹钟怎么工作的流程，简单阐述  
系统没有suspend的话直接走下面流程，如果suspend的话会被RTC唤醒，然后还是走下面的流程

 private class AlarmThread extends Thread  
    {  
        public AlarmThread()  
        {  
            super("AlarmManager");  
        }  
          
        public void run()  
        {   
        while (true)  
            {  
        int result = waitForAlarm(mDescriptor); //这里调用jni调用static jint android\_server\_AlarmManagerService\_waitForAlarm，主要还是对 /dev/alarm  操作  
        ....  
        Alarm alarm = it.next();  
                        try {  
                            if (localLOGV) Slog.v(TAG, "sending alarm " + alarm);  
                            alarm.operation.send(mContext, 0,  
                                    mBackgroundIntent.putExtra(  
                                            Intent.EXTRA\_ALARM\_COUNT, alarm.count),  
                                    mResultReceiver, mHandler);  
        ....  
        }

    }  
      }

static jint android\_server\_AlarmManagerService\_waitForAlarm(JNIEnv\* env, jobject obj, jint fd)  
{  
#if HAVE\_ANDROID\_OS  
    int result = 0;  
      
    do  
    {  
        result = ioctl(fd, ANDROID\_ALARM\_WAIT);  
    } while (result < 0 && errno == EINTR);  
      
    if (result < 0)  
    {  
        LOGE("Unable to wait on alarm: %s\n", strerror(errno));  
        return 0;  
    }  
      
    return result;  
#endif  
}

AlarmManagerService  里面有个AlarmThread  会一直轮询 /dev/alarm文件，如果打开失败就直接返回，成功就会做一些动作，比如查找时间最近的  
alarm，比如睡眠被闹钟唤醒的时候，这边就发一个intent出去，然后在AlarmReceiver.java里面弹出里面会收到就会调用下面的  
        context.startActivity(alarmAlert);

然后弹出alarm  这个界面  
        Class c = AlarmAlert.class;  
其中public class AlarmAlert extends AlarmAlertFullScreen  所以系统睡眠之后被alarm唤醒弹出的alarm就是这边start的  
public class AlarmReceiver extends BroadcastReceiver {

    /\*\* If the alarm is older than STALE\_WINDOW, ignore.  It  
        is probably the result of a time or timezone change \*/  
    private final static int STALE\_WINDOW = 30 \* 60 \* 1000;

    @Override  
    public void onReceive(Context context, Intent intent) {  
    .........  
        Intent alarmAlert = new Intent(context, c);  
        alarmAlert.putExtra(Alarms.ALARM\_INTENT\_EXTRA, alarm);  
        alarmAlert.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK  
                | Intent.FLAG\_ACTIVITY\_NO\_USER\_ACTION);  
        context.startActivity(alarmAlert);  
    ........  
}

到这里alarm 就显示出来了

验证了流程，suspend和不suspend的时候alarm的区别跟上面说的一样

更多Android相关信息见[Android](https://www.linuxidc.com/topicnews.aspx?tid=11) 专题页面 [http://www.linuxidc.com/topicnews.aspx?tid=11](https://www.linuxidc.com/topicnews.aspx?tid=11)

[linux](http://www.linuxidc.com/)

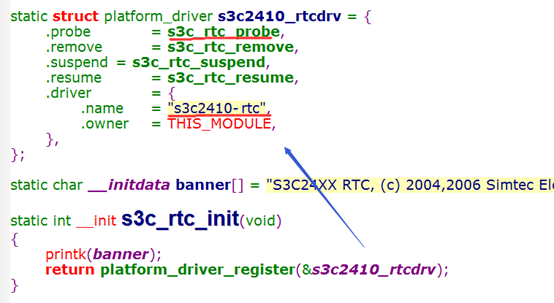
|  |
| --- |
| * [上一页](https://www.linuxidc.com/Linux/2013-08/88965p2.htm) * [1](https://www.linuxidc.com/Linux/2013-08/88965.htm) * [2](https://www.linuxidc.com/Linux/2013-08/88965p2.htm) * 3 |

# [30.Linux-RTC驱动分析及使用](https://www.cnblogs.com/lifexy/p/7839625.html)

 linux中的rtc驱动位于drivers/rtc下,里面包含了许多开发平台的RTC驱动，我们这里是以S3C24xx为主,所以它的RTC驱动为rtc-s3c.c

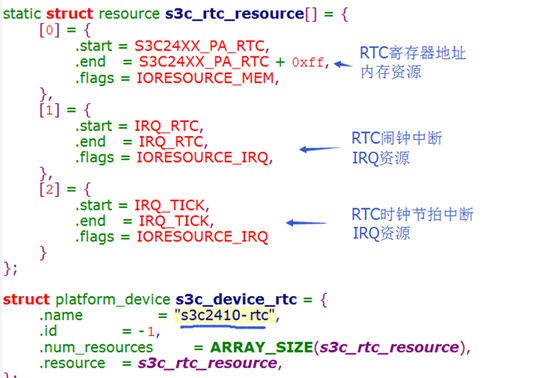
**1.进入./drivers/rtc/rtc-s3c.c**

还是首先进入入口函数,如下图所示:



这里注册了一个“s3c2410-rtc”名称的平台设备驱动

而“s3c2410-rtc”的平台设备,在./arch/arm/plat-s3c24xx/dev.c里定义了,只不过这里没有注册,如下图所示:



当内核匹配到有与它名称同名的平台设备,就会调用.probe函数,接下来我们便进入s3c2410\_rtcdrv->probe函数中看看,做了什么:

[复制代码](javascript:void(0);)

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

{

struct rtc\_device \*rtc; //rtc设备结构体

struct resource \*res;

int ret;

s3c\_rtc\_tickno = platform\_get\_irq(pdev, 1); 　　　　　//获取IRQ\_TICK节拍中断资源

s3c\_rtc\_alarmno = platform\_get\_irq(pdev, 0); 　　　　　　//获取IRQ\_RTC闹钟中断资源

res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0); //获取内存资源

s3c\_rtc\_mem = request\_mem\_region(res->start,res->end-res->start+1,pdev->name);//申请内存资源

s3c\_rtc\_base = ioremap(res->start, res->end - res->start + 1); //对内存进行重映射

s3c\_rtc\_enable(pdev, 1); 　　　　 //设置硬件相关设置,使能RTC寄存器

s3c\_rtc\_setfreq(s3c\_rtc\_freq); //设置TICONT寄存器，使能节拍中断,设置节拍计数值

/\*1.注册RTC设备\*/

rtc = rtc\_device\_register("s3c", &pdev->dev, &s3c\_rtcops,THIS\_MODULE);

rtc->max\_user\_freq = 128;

platform\_set\_drvdata(pdev, rtc);

return 0;

}

[复制代码](javascript:void(0);)

显然最终会调用rtc\_device\_register()函数来向内核注册rtc\_device设备,注册成功会返回一个已注册好的rtc\_device,

而s3c\_rtcops是一个rtc\_class\_ops结构体,里面就是保存如何操作这个rtc设备的函数,比如读写RTC时间,读写闹钟时间等,注册后,会保存在rtc\_device->ops里

该函数在drivers/rtc/Class.c文件内被定义。Class.c文件主要定义了RTC子系统,

而内核初始化,便会进入Class.c,进入rtc\_init()->rtc\_dev\_init(),来注册字符设备:

err = alloc\_chrdev\_region(&rtc\_devt, 0, RTC\_DEV\_MAX, "rtc");   
　　　　　　　　// RTC\_DEV\_MAX=16,表示只注册0~15个次设备号,设备编号保存在rtc\_devt中

**2.它与rtc\_device\_register()函数注册RTC设备,会有什么关系？**

接下来便来看rtc\_device\_register(),代码如下:

[复制代码](javascript:void(0);)

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; //定义一个rtc\_device结构体

... ...

rtc = kzalloc(sizeof(struct rtc\_device), GFP\_KERNEL); //分配rtc\_device结构体为全局变量

/\*设置rtc\_device\*/

　　　　rtc->id = id;

rtc->ops = ops; //将s3c\_rtcops保存在rtc\_device->ops里

rtc->owner = owner;

rtc->max\_user\_freq = 64;

rtc->dev.parent = dev;

rtc->dev.class = rtc\_class;

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

... ...

rtc\_dev\_prepare(rtc); //1.做提前准备,初始化cdev结构体

... ...

rtc\_dev\_add\_device(rtc); //2.在/dev下创建rtc相关文件,将cdev添加到系统中

rtc\_sysfs\_add\_device(rtc); //在/sysfs下创建rtc相关文件

rtc\_proc\_add\_device(rtc); //在/proc下创建rtc相关文件

... ...

　　　　return rtc;

}

[复制代码](javascript:void(0);)

上面的rtc\_dev\_prepare(rtc)和rtc\_dev\_add\_device(rtc)主要做了以下两个(位于./drivers/rtc/rtc-dev.c):

cdev\_init(&rtc->char\_dev, &rtc\_dev\_fops); //绑定file\_operations

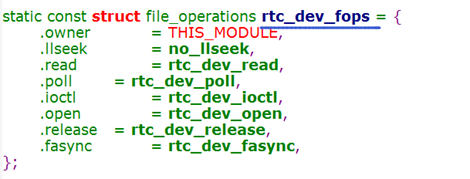
cdev\_add(&rtc->char\_dev, rtc->dev.devt, 1);　　　　//注册rtc->char\_dev字符设备,添加一个从设备到系统中

显然这里的注册字符设备,和我们上节讲的**<http://www.cnblogs.com/lifexy/p/7827559.html>**一摸一样的流程

**所以“s3c2410-rtc”平台设备驱动的.probe主要做了以下几件事:**

* 1.设置RTC相关寄存器
* 2.分配rtc\_device结构体
* 3.设置rtc\_device结构体
* -> 3.1 将struct  rtc\_class\_ops  s3c\_rtcops放入rtc\_device->ops,实现对RTC读写时间等操作
* 4. 注册rtc->char\_dev字符设备,且该字符设备的操作结构体为: struct file\_operations rtc\_dev\_fops

**3.上面的file\_operations操作结构体rtc\_dev\_fops 的成员,如下图所示:**



**3.1当我们应用层open(”/dev/rtcXX”)时,就会调用rtc\_dev\_fops-> rtc\_dev\_open(),我们来看看如何open的:**

[复制代码](javascript:void(0);)

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

{

struct rtc\_device \*rtc = container\_of(inode->i\_cdev,struct rtc\_device, char\_dev);//获取对应的rtc\_device

const struct rtc\_class\_ops \*ops = rtc->ops; 　　　　　　　　　　　　　　 //最终等于s3c\_rtcops

file->private\_data = rtc; //设置file结构体的私有成员等于rtc\_device,再次执行ioctl等函数时,直接就可以提取file->private\_data即可

err = ops->open ? ops->open(rtc->dev.parent) : 0; //调用s3c\_rtcops->open

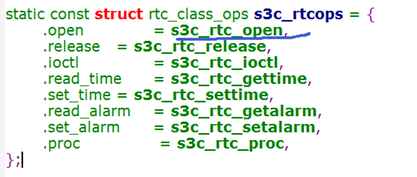
mutex\_unlock(&rtc->char\_lock);

return err;

}

[复制代码](javascript:void(0);)

**显然最终还是调用rtc\_device下的s3c\_rtcops->open:**



而s3c\_rtc\_open()函数里主要是申请了两个中断，一个闹钟中断，一个计时中断:

[复制代码](javascript:void(0);)

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

{

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

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

int ret;

ret = request\_irq(s3c\_rtc\_alarmno, s3c\_rtc\_alarmirq,IRQF\_DISABLED, "s3c2410-rtc alarm", rtc\_dev); //申请闹钟中断

if (ret) {

dev\_err(dev, "IRQ%d error %d\n", s3c\_rtc\_alarmno, ret);

return ret;

}

ret = request\_irq(s3c\_rtc\_tickno, s3c\_rtc\_tickirq,IRQF\_DISABLED, "s3c2410-rtc tick", rtc\_dev);//申请计时中断

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;

}

[复制代码](javascript:void(0);)

**3.2 当我们应用层open后,使用 ioctl(int fd, unsigned long cmd, ...)时,就会调用rtc\_dev\_fops-> rtc\_dev\_ioctl ():**

[复制代码](javascript:void(0);)

static int rtc\_dev\_ioctl(struct inode \*inode, struct file \*file,unsigned int cmd, unsigned long arg)

{

struct rtc\_device \*rtc = file->private\_data; //提取rtc\_device

void \_\_user \*uarg = (void \_\_user \*) arg;

... ...

switch (cmd) {

case RTC\_EPOCH\_SET:

case RTC\_SET\_TIME: //设置时间

if (!capable(CAP\_SYS\_TIME))

return -EACCES;

break;

case RTC\_IRQP\_SET: //改变中断触发速度

... ...}

... ...

switch (cmd) {

case RTC\_ALM\_READ: //读闹钟时间

err = rtc\_read\_alarm(rtc, &alarm); //调用s3c\_rtcops-> read\_alarm

if (err < 0)

return err;

if (copy\_to\_user(uarg, &alarm.time, sizeof(tm))) //长传时间数据

return -EFAULT;

break;

case RTC\_ALM\_SET: //设置闹钟时间 , 调用s3c\_rtcops-> set\_alarm

... ...

case RTC\_RD\_TIME: //读RTC时间, 调用s3c\_rtcops-> read\_alarm

... ...

case RTC\_SET\_TIME: //写RTC时间,调用s3c\_rtcops-> set\_time

... ...

case RTC\_IRQP\_SET: //改变中断触发频率,调用s3c\_rtcops-> irq\_set\_freq

... ...

}

[复制代码](javascript:void(0);)

**最终还是调用s3c\_rtcops下的成员函数**,我们以s3c\_rtcops-> read\_alarm()函数为例,看看如何读出时间的:

[复制代码](javascript:void(0);)

static int s3c\_rtc\_gettime(struct device \*dev, struct rtc\_time \*rtc\_tm)

{

unsigned int have\_retried = 0;

void \_\_iomem \*base = s3c\_rtc\_base; //获取RTC相关寄存器基地址

retry\_get\_time:

/\*获取年,月,日,时,分,秒寄存器\*/

rtc\_tm->tm\_min = readb(base + S3C2410\_RTCMIN);

rtc\_tm->tm\_hour = readb(base + S3C2410\_RTCHOUR);

rtc\_tm->tm\_mday = readb(base + S3C2410\_RTCDATE);

rtc\_tm->tm\_mon = readb(base + S3C2410\_RTCMON);

rtc\_tm->tm\_year = readb(base + S3C2410\_RTCYEAR);

rtc\_tm->tm\_sec = readb(base + S3C2410\_RTCSEC);

/\* 判断秒寄存器中是0，则表示过去了一分钟，那么小时，天，月，等寄存器中的值都可能已经变化，需要重新读取这些寄存器的值\*/

if (rtc\_tm->tm\_sec == 0 && !have\_retried) {

have\_retried = 1;

goto retry\_get\_time;

}

/\*将获取的寄存器值,转换为真正的时间数据\*/

BCD\_TO\_BIN(rtc\_tm->tm\_sec);

BCD\_TO\_BIN(rtc\_tm->tm\_min);

BCD\_TO\_BIN(rtc\_tm->tm\_hour);

BCD\_TO\_BIN(rtc\_tm->tm\_mday);

BCD\_TO\_BIN(rtc\_tm->tm\_mon);

BCD\_TO\_BIN(rtc\_tm->tm\_year);

rtc\_tm->tm\_year += 100; //存储器中存放的是从1900年开始的时间，所以加上100

rtc\_tm->tm\_mon -= 1;

return 0;

}

[复制代码](javascript:void(0);)

同样, 在s3c\_rtcops-> set\_time()函数里,也是向相关寄存器写入RTC时间

**所以,总结如下所示:**

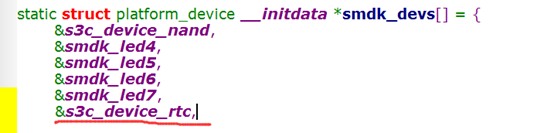
* rtc\_device->char\_dev:　  字符设备,与应用层、以及更底层的函数打交道
* rtc\_device->ops:　　　　更底层的操作函数,直接操作硬件相关的寄存器,被rtc\_device->char\_dev调用

**4.修改内核**

我们单板上使用ls /dev/rtc\*,找不到该字符设备, 因为内核里只定义了s3c\_device\_rtc这个RTC平台设备,没有注册,所以平台驱动没有被匹配上,接下来我们来修改内核里的注册数组

**4.1进入arch/arm/plat-s3c24xx/Common-smdk.c**

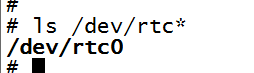
如下图所示,在smdk\_devs[]里,添加RTC的平台设备即可,当内核启动时,就会调用该数组,将里面的platform\_device统统注册一遍



然后将Common-smdk.c代替虚拟机的内核目录下的Common-smdk.c,重新make uImage编译内核即可

**5.测试运行**

启动后,如下图所示, 使用ls /dev/rtc\*,就找到了rtc0这个字符设备



**5.1接下来,便开始设置RTC时间**

在linux里有两个时钟:

硬件时钟(2440里寄存器的时钟)、系统时钟(内核中的时钟)

所以有两个不同的命令: date命令、hwclock命令

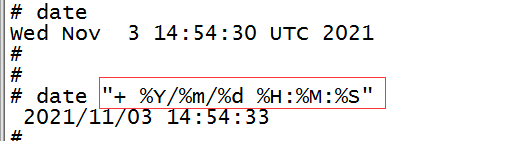
**5.2 date命令使用:**

输入date查看系统时钟:

https://images2017.cnblogs.com/blog/1182576/201711/1182576-20171115155011093-236574226.png

如果觉得不方便也可以指定格式显示日期，需要在字符串前面加”+”

如下图所示,输入了  date  "+ %Y/%m/%d %H:%M:%S"

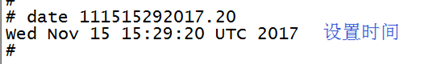


* %M:表示秒
* %m:表示月
* %Y:表示年,当只需要最后两位数字,输入%y即可

**date命令设置时间格式如下:**

date  月日时分年.秒

如下图所示,输入date 111515292017.20，即可设置好系统时钟

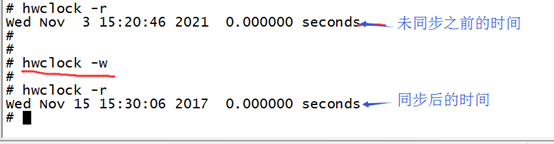


**5.3 hwclock命令使用:**

常用参数如下所示

  -r, --show          读取并打印硬件时钟（read hardware clock and print result ）  
  -s, --hctosys      将硬件时钟同步到系统时钟（set the system time from the hardware clock ）  
  -w, --systohc     将系统时钟同步到硬件时钟（set the hardware clock to the current system time ）

如下图所示,使用hwclock -w，即可同步硬件时钟



**嵌入式Linux——RTC驱动（1）：RTC框架分析**

2018年04月15日 17:33:43 [moxue10](https://me.csdn.net/W1107101310) 阅读数：215

 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/W1107101310/article/details/79947761

**简介：**

        本文通过分层的方式介绍RTC驱动，通过分析RTC在不同层次中所做的不同工作，以及各个层次之间的关系来了解RTC驱动框架。本文分为两部分，第一部分总的介绍RTC框架，而第二部分将结合详细的代码来分析各个层次的关系以及在本层中他们所做的事情。

[**Linux内核**](https://www.baidu.com/s?wd=Linux%E5%86%85%E6%A0%B8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)**：linux-2.6.22.6**

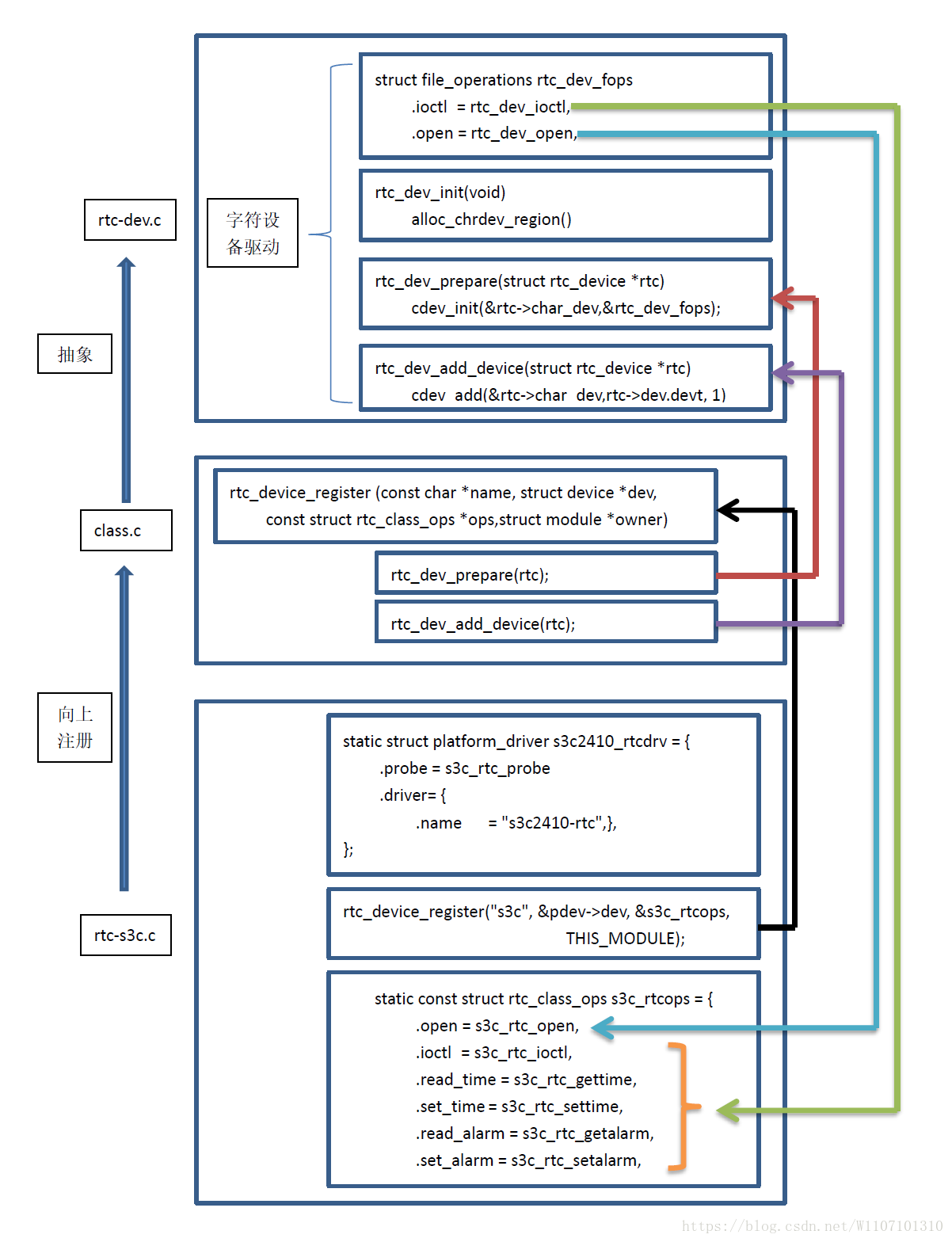
**所用开发板：JZ2440 V3（S3C2440A）**

**声明 ：**

        本文是结合韦东山老师的开发板**JZ2440 V3（S3C2440A）**所写。所以希望大家在看本文之前对开发板中的RTC有所了解。因为我们在后面分析代码时也会介绍对寄存器的操作。而我在上一篇文章：[S3C2440A 第十七章——实时时钟](https://blog.csdn.net/w1107101310/article/details/79937831) 中翻译了芯片手册这部分的内容，如果你对那部分知识不是很了解，可以去看一下。

**第一部分：介绍RTC框架**

        我这两天看了一些关于介绍RTC框架的文章，但是发现大家很少使用框架图来说明这个 框架（或许是有我没有看到），而有也是将文件系统或是其他的接口相关的文件都加到了这个框架图上而他们内部的关系却不是很明确，然而这会使得RTC的框架看起来有点复杂，而对初学者来说有些困难，所以我自己画了一个框架图，同时在框架图中标出了他们的调用关系。希望对大家分析RTC框架有用。



        从上面的图中我们可以看出RTC框架大致可以分为三部分：**rtc-s3c.c** ，**class.c** 以及**rtc-dev.c**。他们依次向上抽象注册。同时又对下面底层的函数进行回调，从而构成了整个完整的RTC框架。我们现在先分开介绍。

**rtc-s3c.c 层：**

**该层是RTC框架的底层，与硬件打交道**。用时又由于这是RTC框架的底层，所以各个芯片对应的这一层可能会不一样，所以**该层是与具体芯片相关的**。同时又由于内核面向对象的思想，所以在该层一定要有一个结构体表示RTC设备——rtc\_device。

**class.c层：**

**该层作为rtc-s3c.c 和 rtc-dev.c 的中间层，对这两层有一个连接作用**。同时由于RTC设备最终要注册进内核，所以该层也**有对RTC实行注册中转的作用**。总的来说该层的作用就是连接中转。同时由于该层已经与具体硬件没有关系了，所以该层的代码不用修改。

**rtc-dev.c 层：**

**该层作为RTC对上层的抽象层。在该层中将RTC设备抽象为字符设备，并用字符设备的框架对其进行注册**。同时由于该层是纯软件的概念了，所以不能对相关的硬件进行操作。而要操作相关硬件时要通过回调函数找到对相关硬件的操作函数，并执行该函数。这用我们就又实现了从抽象到具体的过程。

    通过上面的分析，你也许就明白了，其实上面这三层是相互关联，相互调用的关系。从rtc-s3c.c ，class.c 到 rtc-dev.c经过层层抽象，将RTC设备抽象到字符设备实现对RTC设备向内核的注册，而又通过rtc-dev.c对rtc-s3c.c 的回调实现从抽象到具体的操作。上面就是这三层的关系。而我们现在分析各个层中的函数来详细了解他们内部的关系。

**第二部分：结合代码分析RTC框架**

        我们先分析rtc-s3c.c 进而分析class.c 最后rtc-dev.c，从中我们了解各个层中函数调用的过程（在代码分析这部分我会删除部分不重要的判断或是其他的语句，来使代码看起来更加的清晰明了）。

**rtc-s3c.c ：**

        我们从**入口函数**开始分析：

1. static int \_\_init s3c\_rtc\_init(void)
2. {
3. printk(banner);
4. return platform\_driver\_register(&s3c2410\_rtcdrv);
5. }

        从入口函数中我们可以看出他最主要的是注册了一个平台驱动结构体：**s3c2410\_rtcdrv**。那我们就要看看这个平台驱动结构体做了什么：

1. static struct platform\_driver s3c2410\_rtcdrv = {
2. .probe = s3c\_rtc\_probe,
3. .remove = s3c\_rtc\_remove,
4. .suspend = s3c\_rtc\_suspend,
5. .resume = s3c\_rtc\_resume,
6. .driver = {
7. .name = "s3c2410-rtc",
8. .owner = THIS\_MODULE,
9. },
10. };

        从上面我们可以看出他是一个名为s3c2410-rtc的平台驱动，而在平台——设备——驱动模型中我们知道，既然有这个驱动那就一定有一个与其同名的平台设备，通过搜索我们在**arch\arm\plat-s3c24xx\devs.c**文件中找到了与其同名的设备：

1. static struct resource s3c\_rtc\_resource[] = {
2. [0] = {
3. .start = S3C24XX\_PA\_RTC,
4. .end = S3C24XX\_PA\_RTC + 0xff,
5. .flags = IORESOURCE\_MEM,
6. },
7. [1] = {
8. .start = IRQ\_RTC,
9. .end = IRQ\_RTC,
10. .flags = IORESOURCE\_IRQ,
11. },
12. [2] = {
13. .start = IRQ\_TICK,
14. .end = IRQ\_TICK,
15. .flags = IORESOURCE\_IRQ
16. }
17. };
19. struct platform\_device s3c\_device\_rtc = {
20. .name = "s3c2410-rtc",
21. .id = -1,
22. .num\_resources = ARRAY\_SIZE(s3c\_rtc\_resource),
23. .resource = s3c\_rtc\_resource,
24. };

        而通过上面关于资源的介绍我们知道了**RTC设备的寄存器地址，以及相关的中断号**。而这些将在我们后面对RTC做硬件初始化和设置的时候用到。

        下面我们回到正文，继续分析代码，此时我们认为平台设备与平台驱动匹配，所以我们进入**probe函数**。

1. static int s3c\_rtc\_probe(struct platform\_device \*pdev)
2. {
3. struct rtc\_device \*rtc; */\* 分配一个rtc\_device设备，该结构体对应RTC设备 \*/*
4. struct resource \*res;
5. int ret;
7. */\* 找到中断 \*/*
8. s3c\_rtc\_tickno = platform\_get\_irq(pdev, 1); */\* 获得资源中的IRQ\_TICK中断 \*/*
9. s3c\_rtc\_alarmno = platform\_get\_irq(pdev, 0); */\* 获得资源中的IRQ\_RTC中断 \*/*
11. */\* 获得寄存器资源 \*/*
12. res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0); */\* 获得资源地址 \*/*
13. s3c\_rtc\_mem = request\_mem\_region(res->start, */\* 为获得的寄存器申请空间 \*/*
14. res->end-res->start+1,
15. pdev->name);
16. s3c\_rtc\_base = ioremap(res->start, res->end - res->start + 1); */\* 对资源接口进行重映射 \*/*

19. */\* 检测是否设置正确 \*/*
20. s3c\_rtc\_enable(pdev, 1); */\* rtc使能 \*/*
21. s3c\_rtc\_setfreq(s3c\_rtc\_freq); */\* 设置RTC的滴答时钟频率 \*/*
23. */\* 注册RTC \*/*
24. rtc = rtc\_device\_register("s3c", &pdev->dev, &s3c\_rtcops, */\* 将rtc\_class\_ops结构体向上注册 \*/*
25. THIS\_MODULE);
27. rtc->max\_user\_freq = 128;
29. platform\_set\_drvdata(pdev, rtc); */\* 将设置到的RTC设备放入平台设备中，方便后面的调用 \*/*
30. return 0;
31. }

        从上面的程序中我们知道在probe中主要做了以下五件事情：

**1. 找到中断  
2. 获得寄存器资源  
3. rtc使能  
4. 设置RTC的滴答时钟频率  
5. 注册RTC**

        由于第一和第二在上面已经讲得很清楚了，所以我们直接讲第三个：**rtc使能**

1. static void s3c\_rtc\_enable(struct platform\_device \*pdev, int en)
2. {
3. void \_\_iomem \*base = s3c\_rtc\_base; */\* 获得寄存器基地址 \*/*
4. unsigned int tmp;
6. */\* re-enable the device, and check it is ok \*/*
8. if ((readb(base+S3C2410\_RTCCON) & S3C2410\_RTCCON\_RTCEN) == 0){
9. dev\_info(&pdev->dev, "rtc disabled, re-enabling\n");
11. tmp = readb(base + S3C2410\_RTCCON);
12. writeb(tmp|S3C2410\_RTCCON\_RTCEN, base+S3C2410\_RTCCON); */\* 使能RTC控制 \*/*
13. }
15. if ((readb(base + S3C2410\_RTCCON) & S3C2410\_RTCCON\_CNTSEL)){
16. dev\_info(&pdev->dev, "removing RTCCON\_CNTSEL\n");
18. tmp = readb(base + S3C2410\_RTCCON);
19. writeb(tmp& ~S3C2410\_RTCCON\_CNTSEL, base+S3C2410\_RTCCON); */\* 使能合并BCD计数器 \*/*
20. }
22. if ((readb(base + S3C2410\_RTCCON) & S3C2410\_RTCCON\_CLKRST)){
23. dev\_info(&pdev->dev, "removing RTCCON\_CLKRST\n");
25. tmp = readb(base + S3C2410\_RTCCON);
26. writeb(tmp & ~S3C2410\_RTCCON\_CLKRST, base+S3C2410\_RTCCON); */\* RTC不重启 \*/*
27. }
29. }

        而从上面看，他主要是通过操作硬件相关寄存器，进而控制RTC，实现：

**1. 使能RTC控制  
2. 使能合并BCD计数器  
3. RTC不重启**

        那么我们接着分析第四件事：**设置RTC的滴答时钟频率**

1. static void s3c\_rtc\_setfreq(int freq)
2. {
3. unsigned int tmp;
5. spin\_lock\_irq(&s3c\_rtc\_pie\_lock);
6. tmp = readb(s3c\_rtc\_base + S3C2410\_TICNT) & S3C2410\_TICNT\_ENABLE;
8. s3c\_rtc\_freq = freq;
10. tmp |= (128 / freq)-1;
12. writeb(tmp, s3c\_rtc\_base + S3C2410\_TICNT);
13. spin\_unlock\_irq(&s3c\_rtc\_pie\_lock);
14. }

        设置滴答时钟的中断周期，通过芯片手册：

**— 周期= ( n+1 ) / 128 秒  
— n: 滴答时钟计数值(1~127)**

        而通过上面的设置就是设置的中断周期。

        而下面我们就要讲这一层中最重要的函数，注册RTC函数： rtc\_device\_register。而该函数的详细定义在class.c层中。我们一会儿分析这个函数，而我们观察这个函数：

1. rtc = rtc\_device\_register("s3c", &pdev->dev, &s3c\_rtcops,
2. THIS\_MODULE);

        他定义了一个与RTC硬件操作相关的结构体：**s3c\_rtcops**。

1. static const struct rtc\_class\_ops s3c\_rtcops = {
2. .open = s3c\_rtc\_open,
3. .release = s3c\_rtc\_release,
4. .ioctl = s3c\_rtc\_ioctl,
5. .read\_time = s3c\_rtc\_gettime,
6. .set\_time = s3c\_rtc\_settime,
7. .read\_alarm = s3c\_rtc\_getalarm,
8. .set\_alarm = s3c\_rtc\_setalarm,
9. .proc = s3c\_rtc\_proc,
10. };

        可以说这个结构体是这层的重点，因为**他从硬件上说明了如何操作RTC设备**。而对RTC的基本操作也可以从上面找到。我们会在后面讲rtc-dev.c层时再调回这里来讲解这些函数。而在这里我们就不讲了。

**class.c：**

        下面我们就进入class.c层，来看看这个中间层中做了什么。我们从上面讲解的**rtc\_device\_register函数**入手，看RTC设备在这层向上注册的时候做了什么。

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. }

14. mutex\_lock(&idr\_lock);
15. err = idr\_get\_new(&rtc\_idr, NULL, &id);
16. mutex\_unlock(&idr\_lock);
18. */\* 设置填充rtc\_device结构体 \*/*
19. id = id & MAX\_ID\_MASK;
20. rtc = kzalloc(sizeof(struct rtc\_device), GFP\_KERNEL);
21. rtc->id = id; */\* rtc的ID \*/*
22. rtc->ops = ops; */\* rtc的操作函数，即s3c\_rtcops \*/*
23. rtc->owner = owner;
24. rtc->max\_user\_freq = 64;
25. rtc->dev.parent = dev;
26. rtc->dev.class = rtc\_class;
27. rtc->dev.release = rtc\_device\_release;
28. mutex\_init(&rtc->ops\_lock);
29. spin\_lock\_init(&rtc->irq\_lock);
30. spin\_lock\_init(&rtc->irq\_task\_lock);
31. strlcpy(rtc->name, name, RTC\_DEVICE\_NAME\_SIZE);
32. snprintf(rtc->dev.bus\_id, BUS\_ID\_SIZE, "rtc%d", id);
33. */\* 设备准备，为注册字符设备做准备 \*/*
34. rtc\_dev\_prepare(rtc);
35. */\* 将RTC设备放入设备层 \*/*
36. err = device\_register(&rtc->dev);
38. */\* 添加字符设备到系统 \*/*
39. rtc\_dev\_add\_device(rtc);
40. rtc\_sysfs\_add\_device(rtc);
41. rtc\_proc\_add\_device(rtc);
43. return rtc;
44. }

        从上面看，他主要做了：

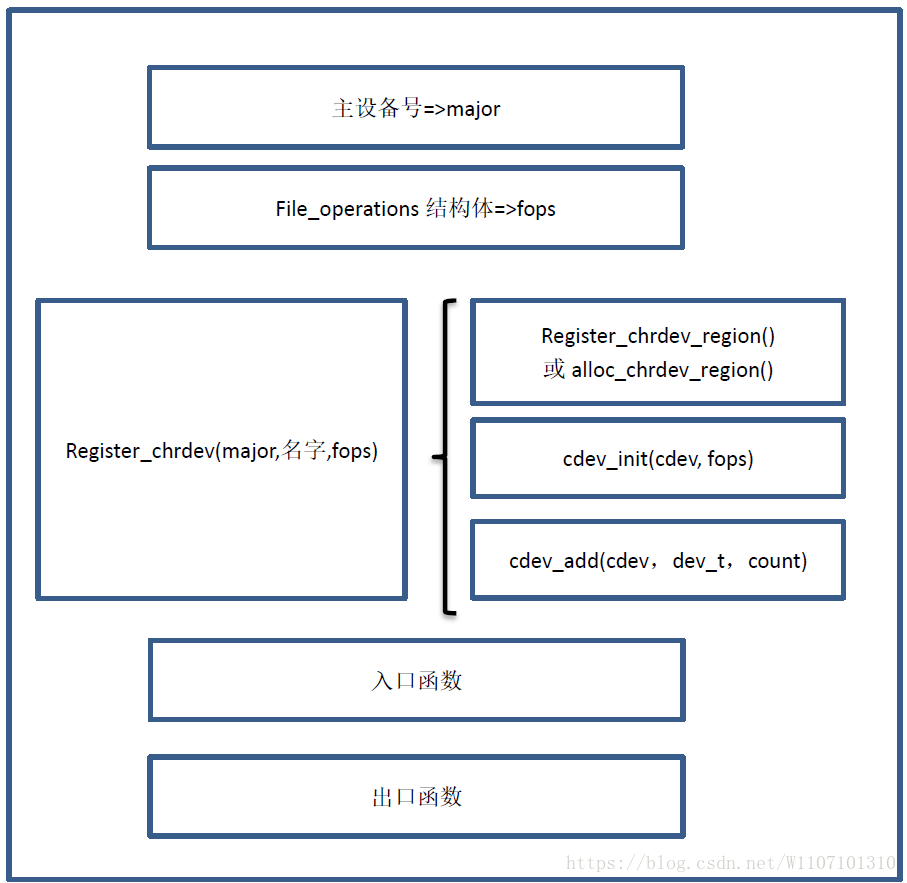
**1. 设置填充rtc\_device结构体  
2. 设备准备，为注册字符设备做准备  
3. 将RTC设备放入设备层  
4. 添加字符设备**

        我们会在下面详细分析2和4，同时又由于2和4的具体实现是在rtc-dev.c中，所以我们会在下层中分析。而1在上面的程序中已经讲得很清楚了，而唯一要我们注意的就是rtc\_device这个结构体，我在下面说明，而3中是将RTC设备放入设备层。

1. struct rtc\_device
2. {
3. struct device dev;
4. struct module \*owner;
6. int id; *//代表是那个rtc设备*
7. char name[RTC\_DEVICE\_NAME\_SIZE]; *//代表rtc设备的名称*
9. const struct rtc\_class\_ops \*ops; *//rtc操作函数集，需要驱动实现*
10. struct mutex ops\_lock; *//操作函数集的互斥锁*
12. struct cdev char\_dev; *//代表rtc字符设备，因为rtc就是个字符设备*
13. unsigned long flags; *//rtc的状态标志，例如RTC\_DEV\_BUSY*
15. unsigned long irq\_data; *//rtc中断数据*
16. spinlock\_t irq\_lock; *//访问数据是要互斥，需要spin\_lock*
17. wait\_queue\_head\_t irq\_queue; *//数据查询中用到rtc队列*
18. struct fasync\_struct \*async\_queue; *//异步队列*
20. struct rtc\_task \*irq\_task; *//在中断中使用task传输数据*
21. spinlock\_t irq\_task\_lock; *//task传输互斥*
22. int irq\_freq; *//rtc的中断频率*
23. int max\_user\_freq; *//rtc的最大中断频率*
25. struct timerqueue\_head timerqueue; *//定时器队列*
26. struct rtc\_timer aie\_timer; *//aie(alaram interrupt enable)定时器*
27. struct rtc\_timer uie\_rtctimer; *//uie(update interrupt enable)定时器*
28. struct hrtimer pie\_timer; */\* sub second exp, so needs hrtimer \*/* *//pie(periodic interrupt enable)定时器*
29. int pie\_enabled; *//pie使能标志*
30. struct work\_struct irqwork;
31. */\* Some hardware can't support UIE mode \*/*
32. int uie\_unsupported; *//uie使能标志*
34. #ifdef CONFIG\_RTC\_INTF\_DEV\_UIE\_EMUL *//RTC UIE emulation on dev interface配置项，目前没有开启*
35. struct work\_struct uie\_task;
36. struct timer\_list uie\_timer;
37. */\* Those fields are protected by rtc->irq\_lock \*/*
38. unsigned int oldsecs;
39. unsigned int uie\_irq\_active:1;
40. unsigned int stop\_uie\_polling:1;
41. unsigned int uie\_task\_active:1;
42. unsigned int uie\_timer\_active:1;
43. #endif
44. };

**rtc-dev.c：**

        通过上面的不断分析我们终于分析到了这一层，而这一层要做的就是将我们的RTC设备抽象为一个字符设备，其实这也体现了内核的编程思想，即**找到共性的，通用的地方将他们抽象出来作为一个抽象的与具体设备无关的层**这就是我们的rtc-dev.c层，**而将不能移植的部分局域化，集中在某几个特征文件中**。从而在软件上实现层次化和模块化。那么我们就在这个层中找一下他抽象为什么样的字符设备，同时了解一下他的回调函数。在分析代码前我们先回忆一下我们的**字符设备是怎么注册**的：



        下面我们开始分析代码从中看他是**如何实现字符驱动的注册的**。

        我们先从他的**入口函数**开始分析：

1. void \_\_init rtc\_dev\_init(void)
2. {
3. int err;
4. err = alloc\_chrdev\_region(&rtc\_devt, 0, RTC\_DEV\_MAX, "rtc");
6. }

        我们可以看出他一开始就为字符设备注册了一份区域，同时我们也知道这个**字符设备的次设备号的基地址为0，最多可以注册16个这样的设备**。而上面的RTC\_DEV\_MAX的宏定义为：

#define RTC\_DEV\_MAX 16 */\* 16 RTCs should be enough for everyone... \*/*

        而分析完入口函数，我们接着上面class.c中要分析的两个函数：rtc\_dev\_prepare(rtc)和rtc\_dev\_add\_device(rtc)继续分析。

我们先分析**rtc\_dev\_prepare(rtc)**：

1. void rtc\_dev\_prepare(struct rtc\_device \*rtc)
2. {
3. rtc->dev.devt = MKDEV(MAJOR(rtc\_devt), rtc->id); */\* 获得设备号 \*/*
5. mutex\_init(&rtc->char\_lock);
6. spin\_lock\_init(&rtc->irq\_lock);
7. init\_waitqueue\_head(&rtc->irq\_queue);
9. cdev\_init(&rtc->char\_dev, &rtc\_dev\_fops); */\* 初始化cdev结构体 \*/*
10. rtc->char\_dev.owner = rtc->owner;
11. }

        从上面看其最主要的就是**初始化cdev结构体，通过初始化cdev机构体，并记住其中的file\_operations结构体，为后面的注册cdev做准备**。而在上面我们看到了file\_operations结构体：**rtc\_dev\_fops**。我们看看他里面都有哪些操作函数：

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. .ioctl = rtc\_dev\_ioctl,
7. .open = rtc\_dev\_open,
8. .release = rtc\_dev\_release,
9. .fasync = rtc\_dev\_fasync,
10. };

        从中我们可以看出他主要是**对字符设备的操作**，我们一会儿会分析他们如何回调RTC设备中的操作函数的，而现在我们发现，在注册字符驱动中我们就剩将cdev结构体注册进内核了。而我们在class.c中也就剩**rtc\_dev\_add\_device(rtc)**函数了。那我们看他里面会不会有注册函数那。

1. void rtc\_dev\_add\_device(struct rtc\_device \*rtc)
2. {
3. if (cdev\_add(&rtc->char\_dev, rtc->dev.devt, 1))
4. printk(KERN\_WARNING "%s: failed to add char device %d:%d\n",
5. rtc->name, MAJOR(rtc\_devt), rtc->id);
6. else
7. pr\_debug("%s: dev (%d:%d)\n", rtc->name,
8. MAJOR(rtc\_devt), rtc->id);
9. }

        我们发现里面确实有注册函数：**cdev\_add(&rtc->char\_dev, rtc->dev.devt, 1)**。而通过这个函数实现将字符设备向内核的注册。

        我们分析到现在就分析完了三个层次，也了解了他们层层抽象，层层调用的关系。而我们现在要想的是，**如果我们想对这个RTC设备操作该怎么操作？我们的用户可以操作到字符设备层，而字符设备又是怎么操作到RTC设备**，这就需要我们分析了。

        我们还是顺着程序往下看，看看他是怎么操作的。而要看字符设备的操作就是要看他的file\_operations结构体。我们现在先从**open函数**开始看：

1. static int rtc\_dev\_open(struct inode \*inode, struct file \*file)
2. {
3. int err;
4. struct rtc\_device \*rtc = container\_of(inode->i\_cdev,
5. struct rtc\_device, char\_dev); */\* 获得RTC设备结构体 \*/*
6. const struct rtc\_class\_ops \*ops = rtc->ops; */\* 获得RTC设备的操作函数 \*/*
8. file->private\_data = rtc;
10. err = ops->open ? ops->open(rtc->dev.parent) : 0; */\* 如果RTC设备存在open函数，调用他的open函数 \*/*
12. return err;
13. }

        从上面我简化的函数可以看出**字符设备的open函数其实就是回调RTC设备的open函数**，现在就实现了用抽象层到物理层的回调。我们看看**RTC设备的open函数**做了什么：

1. static int s3c\_rtc\_open(struct device \*dev)
2. {
3. struct platform\_device \*pdev = to\_platform\_device(dev);
4. struct rtc\_device \*rtc\_dev = platform\_get\_drvdata(pdev);
5. int ret;
7. ret = request\_irq(s3c\_rtc\_alarmno, s3c\_rtc\_alarmirq,
8. IRQF\_DISABLED, "s3c2410-rtc alarm", rtc\_dev);
10. ret = request\_irq(s3c\_rtc\_tickno, s3c\_rtc\_tickirq,
11. IRQF\_DISABLED, "s3c2410-rtc tick", rtc\_dev);

14. return ret;
15. }

        从上面看很简单就是为RTC设备申请了两个中断。

        下面我们在分析**字符驱动中的ioctl函数**：

1. static int rtc\_dev\_ioctl(struct inode \*inode, struct file \*file,
2. unsigned int cmd, unsigned long arg)
3. {
4. int err = 0;
5. struct rtc\_device \*rtc = file->private\_data;
6. const struct rtc\_class\_ops \*ops = rtc->ops;
7. struct rtc\_time tm;
8. struct rtc\_wkalrm alarm;
9. void \_\_user \*uarg = (void \_\_user \*) arg;
11. */\* check that the calling task has appropriate permissions*
12. *\* for certain ioctls. doing this check here is useful*
13. *\* to avoid duplicate code in each driver.*
14. *\*/*
15. switch (cmd) {
16. case RTC\_EPOCH\_SET:
17. case RTC\_SET\_TIME:
18. ·············
19. case RTC\_IRQP\_SET:
20. ·············
21. case RTC\_PIE\_ON:
22. ·············
23. }
25. */\* avoid conflicting IRQ users \*/*
26. if (cmd == RTC\_PIE\_ON || cmd == RTC\_PIE\_OFF || cmd == RTC\_IRQP\_SET) {
27. spin\_lock\_irq(&rtc->irq\_task\_lock);
28. if (rtc->irq\_task)
29. err = -EBUSY;
30. spin\_unlock\_irq(&rtc->irq\_task\_lock);
32. if (err < 0)
33. return err;
34. }
36. */\* try the driver's ioctl interface \*/*
37. if (ops->ioctl) {
38. err = ops->ioctl(rtc->dev.parent, cmd, arg);
39. if (err != -ENOIOCTLCMD)
40. return err;
41. }
43. */\* if the driver does not provide the ioctl interface*
44. *\* or if that particular ioctl was not implemented*
45. *\* (-ENOIOCTLCMD), we will try to emulate here.*
46. *\*/*
48. switch (cmd) {
49. case RTC\_ALM\_READ:
50. err = rtc\_read\_alarm(rtc, &alarm);
51. ········
52. case RTC\_ALM\_SET:
53. if (copy\_from\_user(&alarm.time, uarg, sizeof(tm)))
54. return -EFAULT;
56. alarm.enabled = 0;
57. alarm.pending = 0;
58. alarm.time.tm\_wday = -1;
59. alarm.time.tm\_yday = -1;
60. alarm.time.tm\_isdst = -1;
62. */\* RTC\_ALM\_SET alarms may be up to 24 hours in the future.*
63. *\* Rather than expecting every RTC to implement "don't care"*
64. *\* for day/month/year fields, just force the alarm to have*
65. *\* the right values for those fields.*
66. *\**
67. *\* RTC\_WKALM\_SET should be used instead. Not only does it*
68. *\* eliminate the need for a separate RTC\_AIE\_ON call, it*
69. *\* doesn't have the "alarm 23:59:59 in the future" race.*
70. *\**
71. *\* NOTE: some legacy code may have used invalid fields as*
72. *\* wildcards, exposing hardware "periodic alarm" capabilities.*
73. *\* Not supported here.*
74. *\*/*
75. {
76. unsigned long now, then;
78. err = rtc\_read\_time(rtc, &tm);
79. if (err < 0)
80. return err;
81. rtc\_tm\_to\_time(&tm, &now);
83. alarm.time.tm\_mday = tm.tm\_mday;
84. alarm.time.tm\_mon = tm.tm\_mon;
85. alarm.time.tm\_year = tm.tm\_year;
86. err = rtc\_valid\_tm(&alarm.time);
87. if (err < 0)
88. return err;
89. rtc\_tm\_to\_time(&alarm.time, &then);
91. */\* alarm may need to wrap into tomorrow \*/*
92. if (then < now) {
93. rtc\_time\_to\_tm(now + 24 \* 60 \* 60, &tm);
94. alarm.time.tm\_mday = tm.tm\_mday;
95. alarm.time.tm\_mon = tm.tm\_mon;
96. alarm.time.tm\_year = tm.tm\_year;
97. }
98. }
100. err = rtc\_set\_alarm(rtc, &alarm);
101. break;
103. case RTC\_RD\_TIME: */\* 读RTC的时间 \*/*
104. err = rtc\_read\_time(rtc, &tm);
105. if (err < 0)
106. return err;
108. if (copy\_to\_user(uarg, &tm, sizeof(tm)))
109. return -EFAULT;
110. break;
112. case RTC\_SET\_TIME: */\* 设置RTC的时间 \*/*
113. if (copy\_from\_user(&tm, uarg, sizeof(tm)))
114. return -EFAULT;
116. err = rtc\_set\_time(rtc, &tm);
117. break;
119. case RTC\_IRQP\_READ:
120. if (ops->irq\_set\_freq)
121. err = put\_user(rtc->irq\_freq, (unsigned long \_\_user \*)uarg);
122. break;
124. case RTC\_IRQP\_SET:
125. if (ops->irq\_set\_freq)
126. err = rtc\_irq\_set\_freq(rtc, rtc->irq\_task, arg);
127. break;

130. case RTC\_WKALM\_SET:
131. if (copy\_from\_user(&alarm, uarg, sizeof(alarm)))
132. return -EFAULT;
134. err = rtc\_set\_alarm(rtc, &alarm);
135. break;
137. case RTC\_WKALM\_RD:
138. err = rtc\_read\_alarm(rtc, &alarm);
139. if (err < 0)
140. return err;
142. if (copy\_to\_user(uarg, &alarm, sizeof(alarm)))
143. return -EFAULT;
144. break;

147. default:
148. err = -ENOTTY;
149. break;
150. }
152. return err;
153. }

        从上面看这个函数中调用了很多RTC设备的操作函数，也是主要的对RTC设备的控制和操作。

        而分析到这里我对RTC驱动框架就分析完了，还是那句老话，其实这个框架就是一个从RTC设备到字符设备的抽象，同时又通过字符设备回调RTC设备的操作函数，实现对RTC设备的控制。希望我的分析对你有用。

**参考文章：**

[ARM-Linux驱动--RTC(实时时钟)驱动分析](https://blog.csdn.net/geekcome/article/details/6584527)  
[Linux RTC驱动模型分析](https://blog.csdn.net/longwang155069/article/details/52302383)  
[RTC驱动模型分析](http://www.it165.net/os/html/201504/12612.html)  
[linux RTC 驱动模型分析](https://blog.csdn.net/yaozhenguo2006/article/details/6824970)  
[30.Linux-RTC驱动分析及使用](https://www.cnblogs.com/lifexy/p/7839625.html)