# 驱动架构

sysfs文件系统

rtc-dev

rtc-sysfs

rtc-proc

rtc-interface

rtc设备驱动

rtc设备

用户空间

硬件

字符设备接口

proc文件系统

内

核

空

间

核心

驱动

rtc class

rtc核心相关文件：

class.c：负责注册rtc类以及提供rtc设备注册/注销方法。

rtc-dev.c：定义设备文件的操作函数用于与用户空间交互。

interface.c：提供了与rtc设备的接口函数。

rtc-sysfs.c：定义了sysfs相关操作接口

rtc-proc.c：proc接口

# rtc设备注册

内核使用rtc\_device结构来描述一个rtc设备：

struct rtc\_device {

struct device dev;//设备结构

struct module \*owner;

int id;//rtc设备编号，会创建/dev/rtc<id>设备

char name[RTC\_DEVICE\_NAME\_SIZE];//设备名称

const struct rtc\_class\_ops \*ops;//rtc操作函数集

struct mutex ops\_lock;//操作函数集的互斥锁

struct cdev char\_dev;//rtc为一个字符设备

unsigned long flags;//状态标志

unsigned long irq\_data;//中断数据

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;//中断频率

int max\_user\_freq;//最大中断频率

struct timerqueue\_head timerqueue;//定时器队列

struct rtc\_timer aie\_timer;//报警中断定时器

struct rtc\_timer uie\_rtctimer;//更新中断定时器

struct hrtimer pie\_timer; //周期中断高精度定时器

int pie\_enabled;//周期中断使能标志

struct work\_struct irqwork;

int uie\_unsupported;

#ifdef CONFIG\_RTC\_INTF\_DEV\_UIE\_EMUL

struct work\_struct uie\_task;

struct timer\_list uie\_timer;

/\* Those fields are protected by rtc->irq\_lock \*/

unsigned int oldsecs;

unsigned int uie\_irq\_active:1;

unsigned int stop\_uie\_polling:1;

unsigned int uie\_task\_active:1;

unsigned int uie\_timer\_active:1;

#endif

};

rtc的操作函数集struct rtc\_class\_ops描述了与rtc设备的通信方法:

struct rtc\_class\_ops {

int (\*open)(struct device \*);

void (\*release)(struct device \*);

int (\*ioctl)(struct device \*, unsigned int, unsigned long);//命令控制

int (\*read\_time)(struct device \*, struct rtc\_time \*);//读取时间

int (\*set\_time)(struct device \*, struct rtc\_time \*);//设置事件

int (\*read\_alarm)(struct device \*, struct rtc\_wkalrm \*);//读取闹钟

int (\*set\_alarm)(struct device \*, struct rtc\_wkalrm \*);//设置闹钟

int (\*proc)(struct device \*, struct seq\_file \*);//proc接口

int (\*set\_mmss64)(struct device \*, time64\_t secs);

int (\*set\_mmss)(struct device \*, unsigned long secs);

int (\*read\_callback)(struct device \*, int data);//读回调函数

int (\*alarm\_irq\_enable)(struct device \*, unsigned int enabled);//闹钟中断使能

int (\*read\_offset)(struct device \*, long \*offset);

int (\*set\_offset)(struct device \*, long offset);

int (\*read\_scratch)(struct device \*, unsigned int, u32\*);

int (\*write\_scratch)(struct device \*, unsigned int, u32);

void (\*power\_off\_program)(struct device \*);

unsigned int scratch\_size;

};

初始化rtc操作函数集后使用如下函数注册rtc设备，内核会自己构建rtc\_device然后返回给驱动程序：

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

.name：RTC设备名称，一般设置为芯片名称。

.dev：父设备，根据rtc芯片为spi接口还是i2c接口一般为spi\_device.dev或 i2c\_client.dev

.ops：rtc操作函数集，这是编写设备驱动时的实现重点。

.owner：rtc设备所属模块，一般为THIS\_MODULE。

如果使用rtc\_device\_register()注册，则还需要注销操作：

void rtc\_device\_unregister(struct rtc\_device \*rtc);

# rtc时间操作

读取/设置rtc时间时使用如下结构传递时间，该结构与C标准库头文件time.h中定义的struct tm结构是类似的(注意与内核的time.h中的struct tm并不一样)，所以如果去看hwclock的源码，其实在设置或读取rtc时间所传递的参数其实是struct tm。

struct rtc\_time {

int tm\_sec;//秒-[0,59]

int tm\_min;//分-[0,59]

int tm\_hour;//时-[0,23]

int tm\_mday;//月份中的天-[1,31]

int tm\_mon;//月-[0,11]

int tm\_year;//自1900年以来的年份

int tm\_wday;//自周日起的天数-[0,6]

int tm\_yday;//自1月1日起的天数-[0,365]

int tm\_isdst;//夏令时标志

};

注意，rtc和posix纪元是不同的，rtc是从1900-01-01 00:00:00开始的，而posix是从1970-01-01 00:00:00开始的。

rtc以二进制编码的十进制(BCD)格式存储时间，每个4位表示0~9的数字(0001 0010 --> 12)，内核提供如下函数完成BCD格式与十进制之前的转换，数据从rtc中读取后需要使用bcd2bin()转换才能赋值给struct rtc\_time成员，同样向rtc设备设置时间前也需要使用bin2bcd()将struct rtc\_time成员转换为BCD格式：

bcd2bin();//BCD转十进制

bin2bcd();//十进制转BCD

读取rtc时间：

int rtc\_read\_time(struct rtc\_device \*rtc, struct rtc\_time \*tm);

设置rtc时间：

int rtc\_set\_time(struct rtc\_device \*rtc, struct rtc\_time \*tm);

设置/获取时间使用的struct rtc\_time的成员都是有边界要求的，使用如下函数可以验证，时间有效则返回0，但是看rtc\_read\_time()/rtc\_set\_time()的实现其实在读取rtc设备时间后以及在设置rtc设备前，内核已经做了边界检查：

int rtc\_valid\_tm(struct rtc\_time \*tm);

# rtc闹钟操作

rtc闹钟使用如下结构表示:

struct rtc\_wkalrm {

unsigned char enabled; //闹钟状态 0-已禁用 1-已启用

unsigned char pending; //闹钟状态 0-未挂起 1-已挂起

struct rtc\_time time; //闹钟时间

};

读取闹钟发生时间:

int rtc\_read\_alarm(struct rtc\_device \*rtc, struct rtc\_wkalrm \*alarm);

设置闹钟发生时间，在实现rtc设备驱动的read\_alarm()/set\_alarm()时也会用到上面介绍的bcd2bin()/bin2bcd()：

int rtc\_set\_alarm(struct rtc\_device \*rtc, struct rtc\_wkalrm \*alarm);

使能闹钟中断：

int rtc\_alarm\_irq\_enable(struct rtc\_device \*rtc, unsigned int enabled);

rtc芯片的闹钟中断引脚一般与CPU的gpio连接，中断发生后在中断回调函数中需要调用如下函数向内核通知rtc irq事件：

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

.rtc：引发irq的rtc设备。

.num：报告中断的个数，一般为1.

.events：中断标志，为RTC\_IRQF或上以下一个或多个标志。

#define RTC\_IRQF 0x80 //表示以下三项中的一项或多项是活动的

#define RTC\_PF 0x40 //周期性中断

#define RTC\_AF 0x20 //报警中断\*

#define RTC\_UF 0x10 //更新1Hz的RTC 的中断

带有闹钟功能的rtc设备可以用于唤醒源，可以使用如下函数将设备声明为唤醒源：

int device\_init\_wakeup(struct device \*dev, bool enable);

# 内核配置

系统启动时都需要将系统时间与rtc时间同步，系统一般会启动一个服务在开机时完成时间同步，当然也可以让内核来完成时间同步，内核的时间同步在hctosys.c中完成，需要配置如下选项:

CONFIG\_RTC\_HCTOSYS=y

通过如下如下配置指定完成时间同步的rtc设备:

CONFIG\_RTC\_HCTOSYS\_DEVICE="rtc0"

# 用户空间操作

## sysfs接口

rtc设备注册完毕后，内核会自动创建/sys/class/rtc/rtcx目录，目录中包含很多只读属性：

data：当前日期。

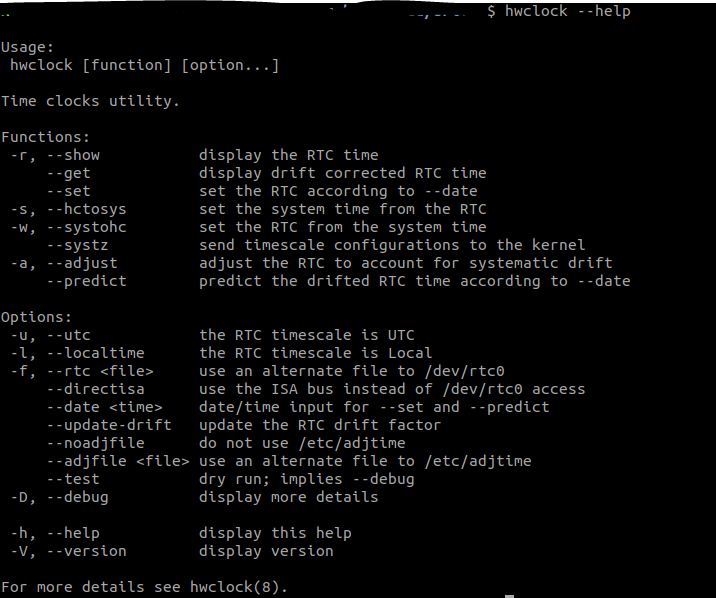
time：当前时间。

hctosys：表示该rtc设备是否是CONFIG\_RTC\_HCTOSYS\_DEVICE指定的设备。

since\_epoch：表示从UNIX纪元(1970-01-01 00:00:00)以来的秒数。

## hwclock工具

hwclock是操作rtc设备最常用的工具，可以用来显示rtc时间、将系统时间同步到rtc时间、将rtc时间同步到系统时间等。命令默认使用/dev/rtc0，如果想使用其他的rtc设备，可通过-f参数执行，其他参数含义如下，更详细的介绍可通过man hwclock查看。



如果想分析hwclock的源码，可下载util-linux软件包源码，其sys-utils/hwclock.c & sys-utils/hwclock-rtc.c就是hwclock的源码，其实它就是调用RTC\_RD\_TIME/RTC\_SET\_TIME命令获取或设置rtc时间，当前将系统时间同步到rtc时间或将rtc时间同步到系统时间需要很多的时间转换函数。

# 示例说明

示例完成了ds1340 rtc驱动，其实内核已经包含了此芯片的驱动ds1307.c，示例只是为了演示rtc实现方法。

设备树应该包含如下信息:

ds1340: rtc@68 {

compatible = "dallas,ds1340";

reg = <0x68>;

};

执行结果如下：

