第四十二章 Linux RTC 驱动实验

RTC 也就是实时时钟，用于记录当前系统时间，对于 Linux 系统而言时间是非常重要的，就和我们使用 Windows 电脑或手机查看时间一样，我们在使用 Linux 设备的时候也需要查看时间。本章我们就来学习一下如何编写 Linux 下的 RTC 驱动程序。

42.1 Linux 内核 RTC 驱动简介

RTC 设备驱动是一个标准的字符设备驱动，应用程序通过 open、release、read、write 和 ioctl 等函数完成对 RTC 设备的操作，本章我们主要学习如何使用 STM32MP1 内部自带的 RTC 外设。

Linux 内核将 RTC 设备抽象为 rtc_device 结构体，因此 RTC 设备驱动就是申请并初始化 rtc_device，最后将 rtc_device 注册到 Linux 内核里面，这样 Linux 内核就有一个 RTC 设备的。至于 RTC 设备的操作肯定是用一个操作集合(结构体)来表示的，我们先来看一下 rtc_device 构体，此结构体定义在 include/linux/rtc.h 文件中，结构体内容如下(删除条件编译)：

示例代码 42.1.1 rtc_device 结构体

```
100 struct rtc_device {
101     struct device dev;                  /*  设备            */
102     struct module *owner;
103
104     int id;                             /* ID               */
105
106     const struct rtc_class_ops *ops;    /* RTC 设备底层操作函数 */
107     struct mutex ops_lock;
108
109     struct cdev char_dev;               /* 字符设备         */
110     unsigned long flags;
111
112     unsigned long irq_data;
113     spinlock_t irq_lock;
114     wait_queue_head_t irq_queue;
115     struct fasync_struct *async_queue;
116
117     int irq_freq;
118     int max_user_freq;
119
120     struct timerqueue_head timerqueue;
121     struct rtc_timer aie_timer;
122     struct rtc_timer uie_rtctimer;
123     struct hrtimer pie_timer; /* sub second exp, so needs hrtimer */
124     int pie_enabled;
125     struct work_struct irqwork;
126     /* Some hardware can't support UIE mode */
127     int uie_unsupported;
......
159 };
```

我们需要重点关注的是 ops 成员变量，这是一个 rtc_class_ops 类型的指针变量，rtc_class_ops 为 RTC 设备的最底层操作函数集合，包括从 RTC 设备中读取时间、向 RTC 设备写入新的时间值等。因此，rtc_class_ops 是需要用户根据所使用的 RTC 设备编写的，此结

构体定义在 include/linux/rtc.h 文件中，内容如下：

示例代码 42.1.2 rtc_class_ops 结构体

```
75 struct rtc_class_ops {
76     int (*ioctl)(struct device *, unsigned int, unsigned long);
77     int (*read_time)(struct device *, struct rtc_time *);
78     int (*set_time)(struct device *, struct rtc_time *);
79     int (*read_alarm)(struct device *, struct rtc_wkalrm *);
80     int (*set_alarm)(struct device *, struct rtc_wkalrm *);
81     int (*proc)(struct device *, struct seq_file *);
82     int (*alarm_irq_enable)(struct device *, unsigned int enabled);
83     int (*read_offset)(struct device *, long *offset);
84     int (*set_offset)(struct device *, long offset);
85 };
```

看名字就知道 rtc_class_ops 操作集合中的这些函数是做什么的了，但是我们要注意，rtc_class_ops 中的这些函数只是最底层的 RTC 设备操作函数，并不是提供给应用层的 file_operations 函数操作集。RTC 是个字符设备，那么肯定有字符设备的 file_operations 函数操作集，Linux 内核提供了一个 RTC 通用字符设备驱动文件，文件名为 drivers/rtc/dev.c，dev.c 文件提供了所有 RTC 设备共用的 file_operations 函数操作集，如下所示：

示例代码 42.1.3 RTC 通用 file_operations 操作集

```
431 static const struct file_operations rtc_dev_fops = {
432     .owner      = THIS_MODULE,
433     .llseek     = no_llseek,
434     .read       = rtc_dev_read,
435     .poll       = rtc_dev_poll,
436     .unlocked_ioctl = rtc_dev_ioctl,
437     .open       = rtc_dev_open,
438     .release    = rtc_dev_release,
439     .fasync     = rtc_dev_fasync,
440 };
```

看到示例代码 42.1.3 是不是很熟悉了，标准的字符设备操作集。应用程序可以通过 ioctl 函数来设置/读取时间、设置/读取闹钟的操作，对应的 rtc_dev_ioctl 函数就会执行。rtc_dev_ioctl 最终会通过操作 rtc_class_ops 中的 read_time、set_time 等函数来对具体 RTC 设备的读写操作。我们简单来看一下 rtc_dev_ioctl 函数，函数内容如下(有省略)：

示例代码 42.1.4 rtc_dev_ioctl 函数代码段

```
202 static long rtc_dev_ioctl(struct file *file,
203             unsigned int cmd, unsigned long arg)
204 {
205     int err = 0;
206     struct rtc_device *rtc = file->private_data;
207     const struct rtc_class_ops *ops = rtc->ops;
208     struct rtc_time tm;
209     struct rtc_wkalrm alarm;
210     void __user *uarg = (void __user *)arg;
211
```

```
212     err = mutex_lock_interruptible(&rtc->ops_lock);
213     if (err)
214         return err;
......
253     switch (cmd) {
......
317     case RTC_RD_TIME:       /* 读取时间 */
318         mutex_unlock(&rtc->ops_lock);
319
320         err = rtc_read_time(rtc, &tm);
321         if (err < 0)
322             return err;
323
324         if (copy_to_user(uarg, &tm, sizeof(tm)))
325             err = -EFAULT;
326         return err;
327
328     case RTC_SET_TIME:      /* 设置时间 */
329         mutex_unlock(&rtc->ops_lock);
330
331         if (copy_from_user(&tm, uarg, sizeof(tm)))
332             return -EFAULT;
333
334         return rtc_set_time(rtc, &tm);
.....
385     default:
386         /* Finally try the driver's ioctl interface */
387         if (ops->ioctl) {
388             err = ops->ioctl(rtc->dev.parent, cmd, arg);
389             if (err == -ENOIOCTLCMD)
390                 err = -ENOTTY;
391         } else {
392             err = -ENOTTY;
393         }
394         break;
395     }
396
397 done:
398     mutex_unlock(&rtc->ops_lock);
399     return err;
400 }
```

第 317 行，RTC_RD_TIME 为时间读取命令。

第 320 行，如果是读取时间命令的话就调用 rtc_read_time 函数获取当前 RTC 时钟，rtc_read_time 会调用__rtc_read_time 函数，__rtc_read_time 函数内容如下：

示例代码 42.1.5 __rtc_read_time 函数代码段

```
84 static int __rtc_read_time(struct rtc_device *rtc, struct rtc_time
*tm)
85 {
86     int err;
87
88     if (!rtc->ops) {
89         err = -ENODEV;
90     } else if (!rtc->ops->read_time) {
91         err = -EINVAL;
92     } else {
93         memset(tm, 0, sizeof(struct rtc_time));
94         err = rtc->ops->read_time(rtc->dev.parent, tm);
95         if (err < 0) {
96             dev_dbg(&rtc->dev, "read_time: fail to read: %d\n",
97                 err);
98             return err;
99         }
100
101         rtc_add_offset(rtc, tm);
102
103         err = rtc_valid_tm(tm);
104         if (err < 0)
105             dev_dbg(&rtc->dev, "read_time: rtc_time isn't valid\n");
106     }
107     return err;
108 }
```

从第 94 行可以看出，__rtc_read_time 函数会通过调用 rtc_class_ops 中的 read_time 成员变量来从 RTC 设备中获取当前时间。rtc_dev_ioctl 函数对其他的命令处理都是类似的，比如 RTC_ALM_READ 命令会通过 rtc_read_alarm 函数获取到闹钟值，而 rtc_read_alarm 函数经过层层调用，最终会调用 rtc_class_ops 中的 read_alarm 函数来获取闹钟值。

至此，Linux 内核中 RTC 驱动调用流程就很清晰了，如图 42.1.1 所示：

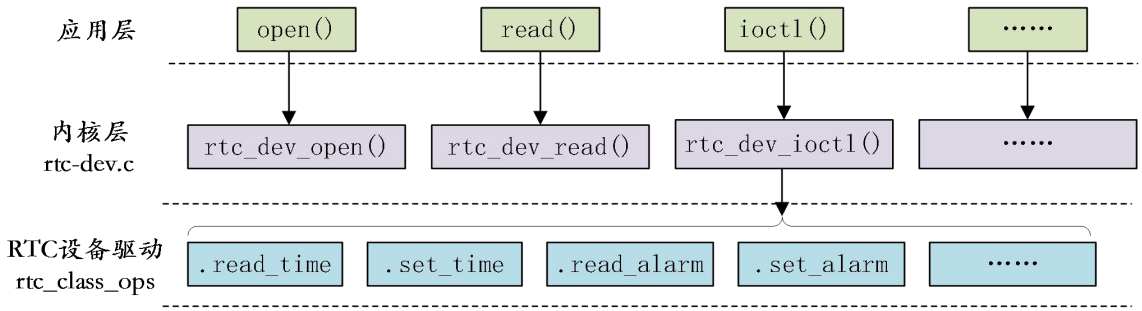

图 42.1.1 Linux RTC 驱动调用流程

当 rtc_class_ops 准备好以后需要将其注册到 Linux 内核中，这里我们可以使用 rtc_device_register 函数完成注册工作。此函数会申请一个 rtc_device 并且初始化这个 rtc_device，最后向调用者返回这个 rtc_device，此函数原型如下：

```
struct rtc_device *rtc_device_register(const char                *name,
                                       struct device             *dev,
                                       const struct rtc_class_ops *ops,
                                       struct module             *owner)
```

函数参数和返回值含义如下：

name：设备名字。

dev：设备。

ops：RTC 底层驱动函数集。

owner：驱动模块拥有者。

返回值：注册成功的话就返回 rtc_device，错误的话会返回一个负值。

当卸载 RTC 驱动的时候需要调用 rtc_device_unregister 函数来注销注册的 rtc_device，函数原型如下：

```
void rtc_device_unregister(struct rtc_device    *rtc)
```

函数参数和返回值含义如下：

rtc：要删除的 rtc_device。

返回值：无。

还有另外一对 rtc_device 注册函数 devm_rtc_device_register 和 devm_rtc_device_unregister，分别为注册和注销 rtc_device。

42.2 STM32MP1 内部 RTC 驱动分析

先直接告诉大家，STM32MP1 的 RTC 驱动我们不用自己编写，因为 ST 已经写好了。其实对于大多数的 SOC 来讲，内部 RTC 驱动都不需要我们去编写，半导体厂商会编写好。但是这不代表我们就偷懒了，虽然不用编写 RTC 驱动，但是我们得看一下这些原厂是怎么编写 RTC 驱动的。

分析驱动，先从设备树入手，打开 stm32mp151.dtsi，在里面找到如下 rtc 设备节点，节点内容如下所示：

示例代码 42.2.1 stm32mp151.dtsi 文件 rtc 节点

```
1746    rtc: rtc@5c004000 {
1747        compatible = "st,stm32mp1-rtc";
1748        reg = <0x5c004000 0x400>;
1749        clocks = <&scmi0_clk CK_SCMI0_RTCAPB>,
1750                 <&scmi0_clk CK_SCMI0_RTC>;
1751        clock-names = "pclk", "rtc_ck";
1752        interrupts-extended = <&exti 19 IRQ_TYPE_LEVEL_HIGH>;
1753        status = "disabled";
1754    };
```

第 1747 行设置兼容属性 compatible 的值为“st,stm32mp1-rtc”，因此在 Linux 内核源码中搜索此字符串即可找到对应的驱动文件，此文件为 drivers/rtc/rtc-stm32.c，在 rtc-stm32.c 文件中找到如下所示内容：

示例代码 42.2.2 设备 platform 驱动框架

```
719     static const struct of_device_id stm32_rtc_of_match[] = {
720         { .compatible = "st,stm32-rtc", .data = &stm32_rtc_data },
721         { .compatible = "st,stm32h7-rtc", .data = &stm32h7_rtc_data },
722         { .compatible = "st,stm32mp1-rtc", .data = &stm32mp1_data },
```

```
723         {}
724     };
725     MODULE_DEVICE_TABLE(of, stm32_rtc_of_match);
726 
......
1020    static struct platform_driver stm32_rtc_driver = {
1021        .probe      = stm32_rtc_probe,
1022        .remove     = stm32_rtc_remove,
1023        .driver     = {
1024            .name   = DRIVER_NAME,
1025            .pm = &stm32_rtc_pm_ops,
1026            .of_match_table = stm32_rtc_of_match,
1027        },
1028    };
1029 
1030    module_platform_driver(stm32_rtc_driver);
```

第 719~723 行，设备树 ID 表。第 722 行，刚好有一个 compatible 属性和设备树的 rtc 的 compatible 属性值一样，所以 rtc 设备节点会和此驱动匹配。

第 1020~1028 行，标准的 platform 驱动框架，当设备和驱动匹配成功以后 stm32_rtc_probe 函数就会执行，我们来看一下 stm32_rtc_probe 函数，函数内容如下(有省略)：

示例代码 42.2.3 stm32_rtc_probe 函数代码段

```
789 static int stm32_rtc_probe(struct platform_device *pdev)
790 {
791     struct stm32_rtc *rtc;
792     const struct stm32_rtc_registers *regs;
793     struct resource *res;
794     int ret;
795 
796     rtc = devm_kzalloc(&pdev->dev, sizeof(*rtc), GFP_KERNEL);
797     if (!rtc)
798         return -ENOMEM;
799 
800     res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
801     rtc->base = devm_ioremap_resource(&pdev->dev, res);
......
856     ret = clk_prepare_enable(rtc->rtc_ck);
857     if (ret)
858         goto err;
......
872     ret = stm32_rtc_init(pdev, rtc);
873     if (ret)
874         goto err;
875 
876     rtc->irq_alarm = platform_get_irq(pdev, 0);
```

```
877     if (rtc->irq_alarm <= 0) {
878         ret = rtc->irq_alarm;
879         goto err;
880     }
......
892     rtc->rtc_dev = devm_rtc_device_register(&pdev->dev, pdev->name,
893                         &stm32_rtc_ops, THIS_MODULE);
894     if (IS_ERR(rtc->rtc_dev)) {
895         ret = PTR_ERR(rtc->rtc_dev);
896             dev_err(&pdev->dev, "rtc device registration failed,
err=%d\n",
897             ret);
898         goto err;
899     }
900
901     /* Handle RTC alarm interrupts */
902       ret = devm_request_threaded_irq(&pdev->dev, rtc->irq_alarm,
NULL,
903                     stm32_rtc_alarm_irq, IRQF_ONESHOT,
904                     pdev->name, rtc);
905     if (ret) {
906             dev_err(&pdev->dev, "IRQ%d (alarm interrupt) already
claimed\n",
907             rtc->irq_alarm);
908         goto err;
909     }
......
940
941     return 0;
......
954 }
```

第 796 行，调用 devm_kzalloc 申请 rtc 大小的空间，返回申请空间的首地址。

第 800 行，调用 platform_get_resource 函数从设备树中获取到 RTC 外设寄存器基地址。

第 801 行，调用函数 devm_ioremap_resource 完成内存映射，得到 RTC 外设寄存器物理基地址对应的虚拟地址。

第 856 行，调用 clk_prepare_enable 函数使能时钟。

第 872 行，初始化 STM32MP1 rtc 的寄存器。

第 876 行，获取设备树的中断号。

第 892 行，调用 devm_rtc_device_register 函数向系统注册 rtc_devcie，RTC 底层驱动集为 stm32_rtc_ops。stm32_rtc_ops 操作集包含了读取/设置 RTC 时间，读取/设置闹钟等函数。

第 902 行，调用 devm_request_threaded_irq 函数请求 RTC 中断，中断服务函数为 stm32_rtc_alarm_irq，用于 RTC 闹钟中断。

stm32_rtc_ops 内容如下所示：

示例代码 42.2.4 rtc_class_ops 操作集

```
623 static const struct rtc_class_ops stm32_rtc_ops = {
624     .read_time     = stm32_rtc_read_time,
625     .set_time      = stm32_rtc_set_time,
626     .read_alarm    = stm32_rtc_read_alarm,
627     .set_alarm     = stm32_rtc_set_alarm,
628     .alarm_irq_enable = stm32_rtc_alarm_irq_enable,
629 };
```

我们就以第 624 行的 stm32_rtc_read_time 函数为例讲解一下 rtc_class_ops 的各个 RTC 底层操作函数该如何去编写。stm32_rtc_read_time 函数用于读取 RTC 时间值，此函数内容如下所示：

示例代码 42.2.5 stm32_rtc_read_time 代码段

```
364 static int stm32_rtc_read_time(struct device *dev,
                                    struct rtc_time *tm)
365 {
366     struct stm32_rtc *rtc = dev_get_drvdata(dev);
367     const struct stm32_rtc_registers *regs = &rtc->data->regs;
368     unsigned int tr, dr;
369
370     /* Time and Date in BCD format */
371     tr = readl_relaxed(rtc->base + regs->tr);
372     dr = readl_relaxed(rtc->base + regs->dr);
373
374     tm->tm_sec = (tr & STM32_RTC_TR_SEC) >> STM32_RTC_TR_SEC_SHIFT;
375     tm->tm_min = (tr & STM32_RTC_TR_MIN) >> STM32_RTC_TR_MIN_SHIFT;
376                 tm->tm_hour    =    (tr    &    STM32_RTC_TR_HOUR)    >>
STM32_RTC_TR_HOUR_SHIFT;
377
378                 tm->tm_mday    =    (dr    &    STM32_RTC_DR_DATE)    >>
STM32_RTC_DR_DATE_SHIFT;
379                 tm->tm_mon    =    (dr    &    STM32_RTC_DR_MONTH)    >>
STM32_RTC_DR_MONTH_SHIFT;
380                 tm->tm_year    =    (dr    &    STM32_RTC_DR_YEAR)    >>
STM32_RTC_DR_YEAR_SHIFT;
381                 tm->tm_wday    =    (dr    &    STM32_RTC_DR_WDAY)    >>
STM32_RTC_DR_WDAY_SHIFT;
382
383     /* We don't report tm_yday and tm_isdst */
384
385     bcd2tm(tm);
386
387     return 0;
388 }
```

第 371~372 行，调用 readl_relaxed 读取 STM32MP1 的 RTC_TR 和 RTC_DR 这两个寄存器的值，其中 TR 寄存器为 RTC 时间寄存器，保存着时、分、秒信息；DR 为 RTC 的日期寄存器，保存着年、月、日信息。通过这两个寄存器我们就可以得到 RTC 时间

第 374~381 行，前两行获取到了 TR 和 DR 这两个寄存器的值，这里需要从这两个寄存器值中提取出具体的年、月、日和时、分、秒信息。

第 385 行，上面得到的时间信息为 BCD 格式的，这里通过 bcd2tm 函数将 BCD 格式转换为 rtc_time 格式， rtc_time 结构体定义如下：

示例代码 42.2.6 rtc_time 结构体类型

```
20 struct rtc_time {
21     int tm_sec;
22     int tm_min;
23     int tm_hour;
24     int tm_mday;
25     int tm_mon;
26     int tm_year;
27     int tm_wday;
28     int tm_yday;
29     int tm_isdst;
30 };
```

42.3 RTC 时间查看与设置

42.3.1 使能内部 RTC

在 Linux 内核移植的时候，设备树是经过精简的，就没有启动 RTC 功能。打开 stm32mp157d-atk.dts 文件，添加如下代码所示：

示例代码 42.3.1.1 rtc 节点信息

```
1 &rtc {
2     status = "okay";
3 };
```

追加的 RTC 节点内容很简单，就是把 status 属性改为“okay”。接着我们重新编译设备树，然后使用新编译的 stm32mp157d-atk.dtb 文件启动开发板。

42.3.2 查看时间

RTC 是用来记时的，因此最基本的就是查看时间，Linux 内核启动的时候可以看到系统时钟设置信息，如图 42.3.2.1 所示：

```
stm32_rtc 5c004000.rtc: registered as rtc0
stm32_rtc 5c004000.rtc: Date/Time must be initialized
stm32_rtc 5c004000.rtc: registered rev:1.2
```

图 42.3.2.1 Linux 启动 log 信息

从图 42.3.2.1 中可以看出，Linux 内核在启动的时候将 rtc 设置为 rtc0，大家的启动信息可能会和图 42.3.2.1 中不同，但是基本上都是一样的。

如果要查看时间的话输入“date”命令即可，结果如图 42.3.2.2 所示：

```
[root@ATK-stm32mp1]:~$ date
Sat Jan  1 03:30:29 UTC 2000
[root@ATK-stm32mp1]:~$
```

图 42.3.2.2 当前时间值

从图 42.3.2.2 可以看出，当前时间为 2000 年 1 月 1 日 03:30:29，很明显时间不对，我们需要重新设置 RTC 时间。

RTC 时间设置也是使用的 date 命令，输入“date --help”命令即可查看 date 命令如何设置系统时间，结果如图 42.3.2.3 所示：

```
[root@ATK-stm32mp1]:~$ date --help
BusyBox v1.31.1 (2020-12-22 16:23:11 CST) multi-call binary.

Usage: date [OPTIONS] [+FMT] [TIME]

Display time (using +FMT), or set time

        [-s,--set] TIME Set time to TIME
        -u,--utc        Work in UTC (don't convert to local time)
        -R,--rfc-2822   Output RFC-2822 compliant date string
        -I[SPEC]        Output ISO-8601 compliant date string
                        SPEC='date' (default) for date only,
                        'hours', 'minutes', or 'seconds' for date and
                        time to the indicated precision
        -r,--reference FILE     Display last modification time of FILE
        -d,--date TIME  Display TIME, not 'now'
        -D FMT          Use FMT (strptime format) for -d TIME conversion

Recognized TIME formats:
        hh:mm[:ss]
        [YYYY.]MM.DD-hh:mm[:ss]
        YYYY-MM-DD hh:mm[:ss]
        [[[[[YY]YY]MM]DD]hh]mm[.ss]
        'date TIME' form accepts MMDDhhmm[[YY]YY][.ss] instead
[root@ATK-stm32mp1]:~$
```

图 42.3.2.3 date 命令帮助信息

比如现在设置当前时间为 2021 年 5 月 2 日 18:53:00，因此输入如下命令：

```
date -s "2021-05-02 18:53:00"
```

设置完成以后再次使用 date 命令查看一下当前时间就会发现时间改过来了，如图 42.3.2.4 所示：

```
[root@ATK-stm32mp1]:~$ date
Sun May  2 18:53:23 UTC 2021
[root@ATK-stm32mp1]:~$
```

图 42.3.2.4 当前时间

大家注意我们使用“date -s”命令仅仅是修改了当前时间，此时间还没有写入到 STM32MP1 内部 RTC 里面或其他的 RTC 芯片里面，因此系统重启以后时间又会丢失。我们需要将当前的时间写入到 RTC 里面，这里要用到 hwclock 命令，输入如下命令将系统时间写入到 RTC 里面：

```
hwclock -w          //将当前系统时间写入到 RTC 里面
```

时间写入到 RTC 里面以后就不怕系统重启以后时间丢失了，如果 STM32MP1 开发板底板接了纽扣电池，那么开发板即使断电了时间也不会丢失。大家可以尝试一下不断电重启和断电重启这两种情况下开发板时间会不会丢失。