AN0111
入门指南

AT32 RTC入门指南

前言

AT32 的 RTC 接口用于实现日历、时钟功能，本文主要就 RTC 的基本功能进行讲解和案列解析。

注：本应用笔记对应的代码是基于雅特力提供的V2.x.x 板级支持包（BSP）而开发，对于其他版本BSP，需要注意使用上的区别。

支持型号列表：

	具备 RTC 的型号

目录

表目录

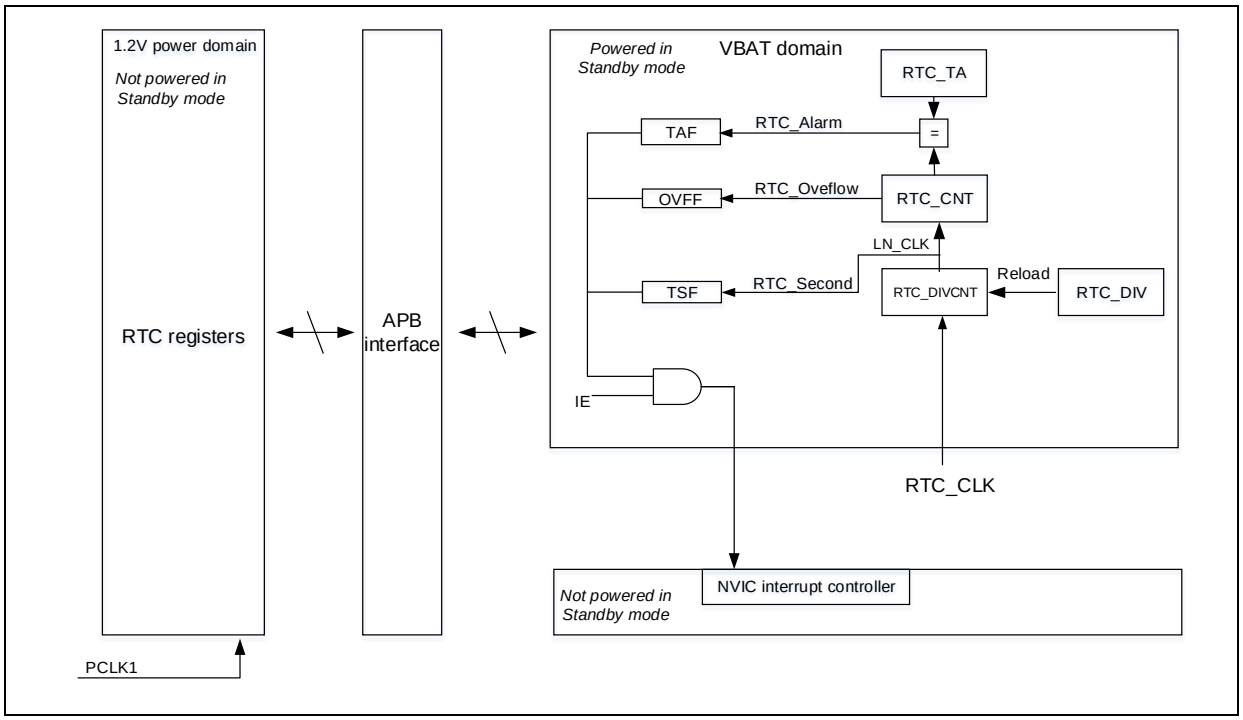
图目录

1 RTC 接口简介

RTC 计数逻辑位于电池供电域，内部为一个 32 位递增计数器，只要电池供电域有电，RTC 便会一直运行，不受系统复位以及 VDD 掉电影响，RTC 主要具有以下功能：

- 日历功能：32 位计数器，通过转换得到年、月、日、时、分、秒
- 闹钟功能
- 入侵检测功能
- 校准功能

图 1. RTC 框图



2 RTC 功能

2.1 寄存器访问

寄存器写保护

上电复位后 RTC 寄存器处于写保护状态，需要先解除写保护，才能写配置 RTC 寄存器。

解锁步骤：

1) 使能 PWC 接口时钟

```
crm_periph_clock_enable(CRM_PWC_PERIPH_CLOCK, TRUE);
```

2) 使能 BPR 接口时钟

```
crm_periph_clock_enable(CRM_BPR_PERIPH_CLOCK, TRUE);
```

3) 解锁电池供电域写保护

```
pwc_battery_powered_domain_access(TRUE);
```

RTC 寄存器同步

由于 RTC 由电池供电域的计数逻辑和 APB1 接口的寄存器组成，寄存器的读写存在同步逻辑。

- 寄存器写：需要等待上一次的 RTC 寄存器配置完成后（CFGF = 1），才能进行新的写操作。
- 寄存器读：当寄存器值从电池供电域更新到 APB1 接口时 UPDF 标志置 1。当在系统复位、电源复位、从待机、深度睡眠模式唤醒后，有可能寄存器还未完全同步，所以需要先软件将 UPDF 标志清除，然后等待 UPDF 标志置 1，以读取正确的值。

RTC 同步相关函数

等待上一次 RTC 寄存器配置完成（写寄存器之前使用）

```
void rtc_wait_config_finish(void);
```

等待 RTC 寄存器更新完成（读取寄存器之前使用）

```
void rtc_wait_update_finish(void);
```

RTC 寄存器写

写 RTC_DIV、RTC_TA、RTC_CNT 寄存器需要先进入配置模式（CFGEN = 1），然后才能对寄存器进行写操作，当退出配置模式（CFGEN = 0）时，就会将寄存器值实际写到电池供电域，这个过程至少需要 3 个 RTCCLK 周期。

下表列举了 RTC 寄存器受写保护状态，以及写入的条件：

表 1. RTC 寄存器

寄存器简称	是否受写保护	是否需要进入配置模式
RTC_CTRLH、RTC_CTRLL	是	否
RTC_DIVH、RTC_DIVL	是	是
RTC_DIVCNTH、RTC_DIVCNTL	-	-
RTC_CNTH、RTC_CNTL	是	是
RTC_TAH、RTC_TAL	是	是

寄存器复位

RTC 寄存器处于电池供电域，可以 CRM_BPDC 的 BPDRST 进行电池供电域复位，也可以由提供的库函数对每个寄存器写默认值进行复位。

RTC 复位相关函数

电池供电域复位

```
void crm_battery_powered_domain_reset(confirm_state new_state);
```

或者

```
void bpr_reset(void);
```

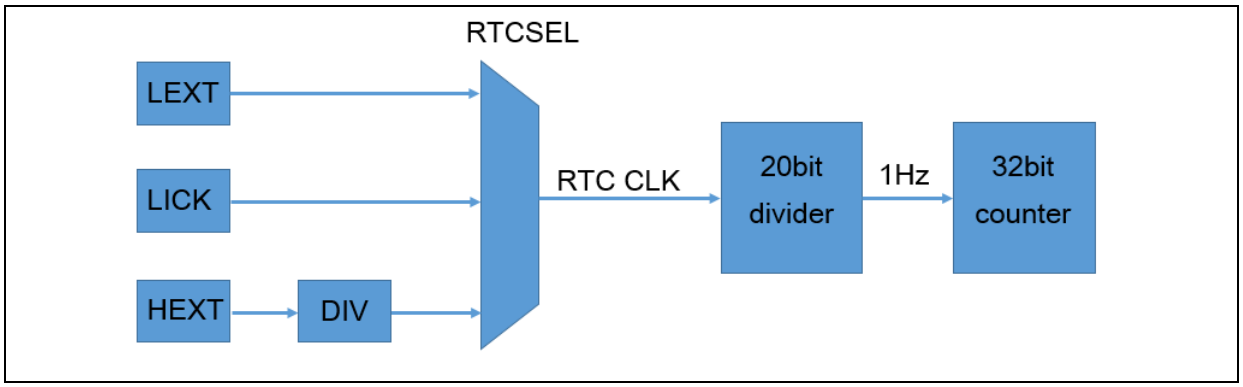
两个函数功能一样，只是 bpr_reset()封装了前一个函数。

2.2 时钟设置

时钟源选择

RTC 时钟源经过选择后输入到分频器，最终得到 1Hz 的时钟用来更新日历。

图 2. RTC 时钟结构



RTC 的时钟源共有 3 种可以选择：

- LEXT：外部低速晶振，通常为 32.768kHz
- LICK：内部低速晶振，通常典型值为 40kHz 范围（30~60kHz），详情请见各型号的 datasheet
- HEXT_DIV：外部高速晶振分频后得到的时钟，不同型号分频值请见下表

表 2. 各型号 HEXT 的预分频值

型号	预分频值
AT32F403Axx	固定 128
AT32F407xx	固定 128
AT32F413xx	固定 128

表 3. 各时钟源优缺点对比

时钟源	频率	优点	缺点
LEXT	32.768kHz	精度最高，能在电池供电下以及低功耗模式下工作	需要一颗晶振，增加元件成本，增大 PCB 布线面积
LICK	典型值 40kHz 范围(30kHz~60kHz)	能在电池供电下以及低功耗模式下工作，节省一颗晶振，降低了 PCB 布线面积	时间精度低，时间不准

时钟源	频率	优点	缺点
HEXT	主晶振频率	精度也比较高（取决于主晶振精度），节省一颗晶振，降低了 PCB 布线面积	不能在电池供电和低功耗模式下工作

RTC 时钟源设置相关函数

选择对应时钟使能

```
void crm_clock_source_enable(crm_clock_source_type source, confirm_state new_state)
```

选择 RTC 时钟

```
void crm_rtc_clock_select(crm_rtc_clock_type value)
```

使能 RTC 时钟

```
void crm_rtc_clock_enable(confirm_state new_state)
```

预分频器设置

RTC_CLK 通过 20 位预分频器后获得 1Hz 时钟，计算公式如下：

$$f_{1Hz} = \frac{f_{RTC\_CLK}}{DIV + 1}$$

表 4. 分频设置举例

时钟源	频率	DIV	日历时钟
LEXT	32.768kHz	32767	1Hz
LICK	典型值 40kHz	39999	1Hz
HEXT	8MHz，128 分频	62499	1Hz

RTC 分频设置相关函数

设置 RTC 预分频器

```
void rtc_divider_set(uint32_t div_value);
```

获取 RTC 预分频器值

```
uint32_t rtc_divider_get(void);
```

RTC 时钟初始化举例：

```
/* 使能 LEXT 时钟 */
crm_clock_source_enable(CRM_CLOCK_SOURCE_LEXT, TRUE);

/* 等待 LEXT 时钟稳定 */
while(crm_flag_get(CRM_LEXT_STABLE_FLAG) == RESET)
{
}
```

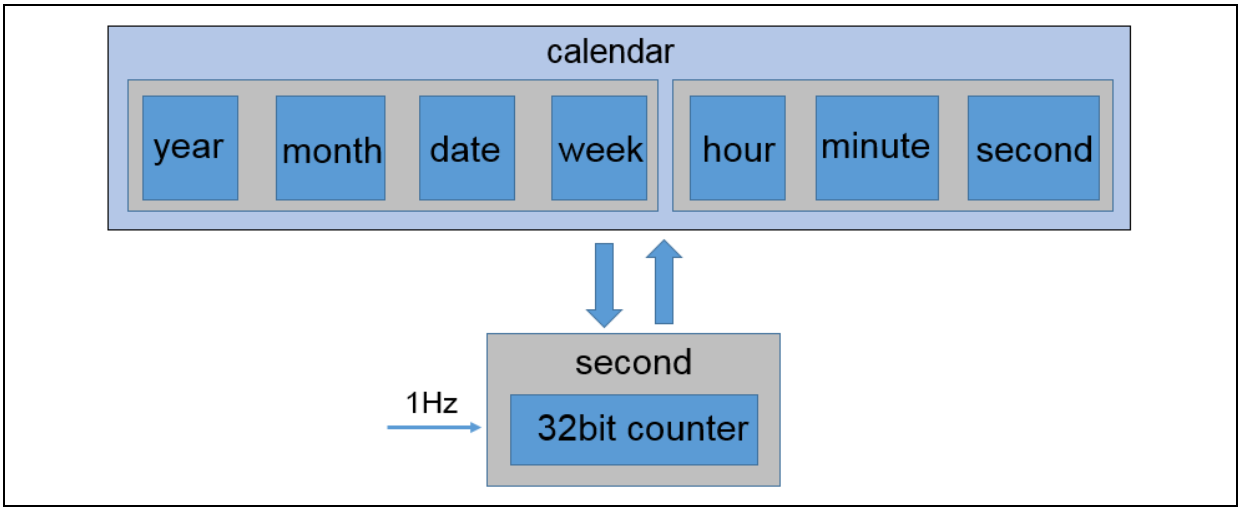
```
/* 选择 LEXT 时钟作为 RTC 时钟 */
crm_rtc_clock_select(CRM_RTC_CLOCK_LEXT);

/* 使能 RTC 时钟 */
crm_rtc_clock_enable(TRUE);

/* 配置 RTC 分频器 DIV=32767 */
rtc_divider_set(32767);
```

2.3 日历

RTC 内部是一个 32 位的计数器，通常使用中该计数器 1 秒增加 1，也就是该计数器相当于秒钟，然后根据当前的秒钟值，通过转换得到年、月、日、星期、时、分、秒，实现日历的功能，修改计数器的值便可修改时间和日期。

根据使用需要还可以产生秒中断：若秒中断使能（TSIEN=1），每隔一秒产生一个秒中断。

图 3. 日历转换



计数相关函数

设置 RTC 计数值

```
void rtc_counter_set(uint32_t counter_value);
```

获取 RTC 计数值

```
uint32_t rtc_counter_get(void);
```

秒钟转换成日历

先规定一个起始时间，例如 1970-1-1 00:00:00 对应计数器为 0，现在比如计数值为 200000，那么换算成时间为：

- 天数：200000 / 86400 = 2
- 小时：(200000 % 86400) / 3600= 7
- 分钟：(200000 % 3600) / 60= 33
- 秒钟：200000 % 60 = 20

所以现在的时间对应为 1970-1-3 07:33:20，对应日历转换成秒钟也是相同的思路。

在 BSP 的例程 project\at_start_f403a\examples\rtc\calendar 中，我们提供了秒钟与日历的相互转换函数。

设置日历值（日历转换成秒钟）

```
uint8_t rtc_time_set(calendar_type *calendar);
```

结构体 calendar_type 里面参数含义如下：

- year：年
- month：月
- day：日
- hour：时
- min：分
- sec：秒
- week：星期几

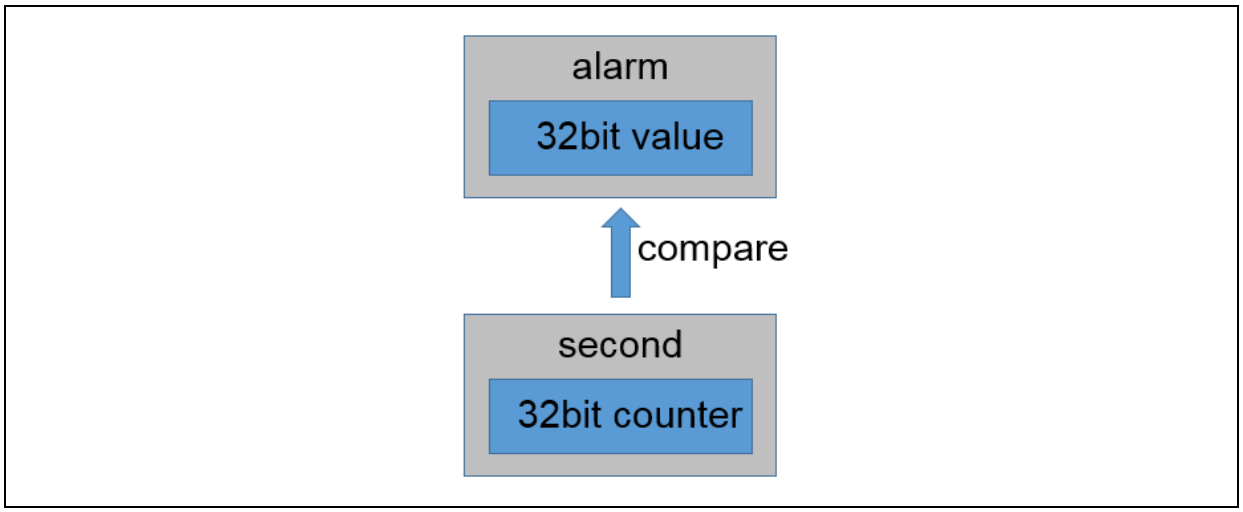
读取日历值（秒钟转换成日历）

```
void rtc_time_get(void);
```

2.4 闹钟

RTC 闹钟是一个 32 位的值，当闹钟值和计数值相等时产生闹钟事件（TAF 置 1），当中断使能时，会产生中断。

图 4. 闹钟匹配



闹钟相关函数

闹钟值设置函数

```
void rtc_alarm_set(uint32_t alarm_value);
```

中断使能函数

```
void rtc_interrupt_enable(uint16_t source, confirm_state new_state);
```

标志获取函数

```
flag_status rtc_flag_get(uint16_t flag);
```

标志清除函数

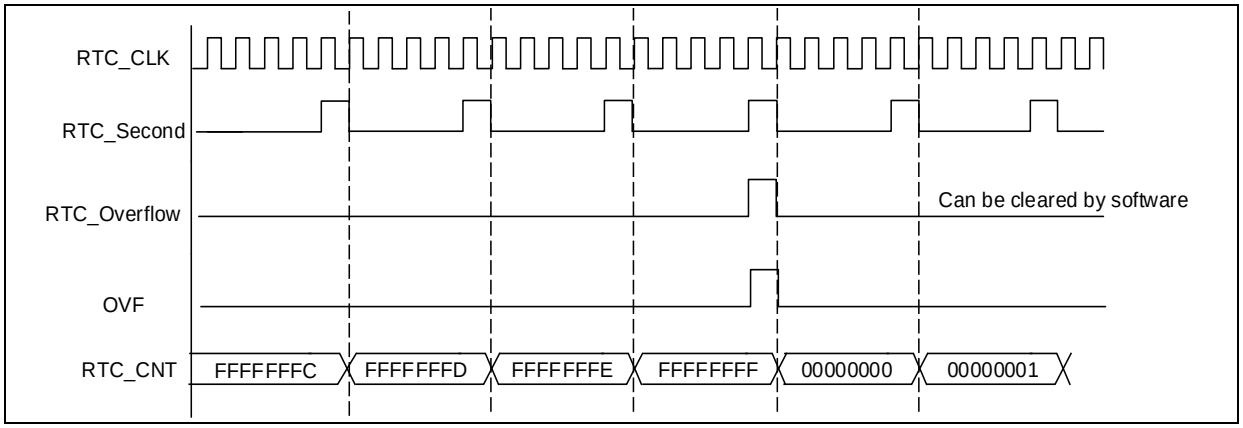
```
void rtc_flag_clear(uint16_t flag);
```

2.5 计数值溢出

由于计数值为 32 位，所以存在溢出问题，当计数值为 0xFFFFFFFF 溢出到 0x00000000 时，产生溢出事件，OVFF 标志置 1 当闹钟使能后，由于溢出后，秒与日历的相转换关系便不正确，所以用户需妥善处理溢出事件。

0xFFFFFFFF 所能代表的最大时间为 136 年，例程起始时间为 1975，所以能够到 2106 年不溢出。

图 5. 计数值溢出示例（分频值为 4）



2.6 中断

当发生闹钟、秒、溢出事件时，RTC 可产生中断。闹钟中断有两种配置模式：

- 不配置 EXINT 线使用 RTC_IRQn 中断向量，此种方式不能唤醒 DEEPSLEEP 和 STANDBY 模式；
- 配置 EXINT 线使用 RTCAlarm_IRQn 中断向量，此种方式可以唤醒 DEEPSLEEP 和 STANDBY 模式。

要使能 RTC 闹钟（不需要唤醒低功耗模式）、秒、溢出中断可按以下操作配置：

- 使能 RTC 中断对应的 NVIC 通道。
- 使能对应的 RTC 中断控制位。

要使能 RTC 闹钟（需要唤醒低功耗模式）中断可按以下操作配置：

- EXINT 线 17 配置为中断模式并使能，有效沿选择上升沿。
- 使能 RTC 中断对应的 NVIC 通道。
- 使能对应的 RTC 中断控制位。

下表说明了 RTC 时钟源、事件以及中断对唤醒低功耗模式的影响：

表 5. RTC 唤醒低功耗模式

时钟源	事件	唤醒 SLEEP	唤醒 DEEPSLEEP	唤醒 STANDBY
HEXT	闹钟（RTCAlarm_IRQn）	√	×	×
	闹钟（RTC_IRQn）	√	×	×
	溢出	√	×	×
	秒	√	×	×
LICK	闹钟（RTCAlarm_IRQn）	√	√	√
	闹钟（RTC_IRQn）	√	×	×
	溢出	√	×	×
	秒	√	×	×
LEXT	闹钟（RTCAlarm_IRQn）	√	√	√
	闹钟（RTC_IRQn）	√	×	×
	溢出	√	×	×
	秒	√	×	×

表 6. 中断控制

中断事件	事件标志	中断使能位
闹钟	TAF	TAIEN
秒	TSF	TSIEN
溢出	OVFF	OVFIEN

表 7. 事件对应中断向量

事件	中断向量	中断函数
闹钟（RTCAlarm_IRQn）	RTCAlarm_IRQn	RTCAlarm_IRQHandler
闹钟（RTC_IRQn）	RTC_IRQn	RTC_IRQHandler
溢出	RTC_IRQn	RTC_IRQHandler
秒	RTC_IRQn	RTC_IRQHandler

中断、事件相关函数

中断使能函数

```
void rtc_interrupt_enable(uint16_t source, confirm_state new_state);
```

标志获取函数

```
flag_status rtc_flag_get(uint16_t flag);
```

标志清除函数

```
void rtc_flag_clear(uint16_t flag);
```

中断配置示例 1：以 AT32F403A 的闹钟为例，使用 RTCAlarm_IRQn 中断向量

```
/* 配置 EXINT 线 */
exint_default_para_init(&exint_init_struct);
exint_init_struct.line_enable = TRUE; //使能
```

```
  exint_init_struct.line_mode = EXINT_LINE_INTERRUPUT; //中断模式
  exint_init_struct.line_select = EXINT_LINE_17;//EXINT 线 17
  exint_init_struct.line_polarity = EXINT_TRIGGER_RISING_EDGE;//上升沿触发
  exint_init(&exint_init_struct);

  /* 使能闹钟中断 NVIC 向量：RTCAlarm_IRQn */
  nvic_irq_enable(RTCAlarm_IRQn, 0, 1);

  /* 设置闹钟值 */
  rtc_alarm_set(seccount);

  /* 使能闹钟中断 */
  rtc_interrupt_enable(RTC_TA_INT, TRUE);
```

中断处理函数

```
void RTCAlarm_IRQHandler(void)
{
  if(rtc_flag_get(RTC_TA_FLAG) != RESET)
  {
    /* 清除闹钟标志 */
    rtc_flag_clear(RTC_TA_FLAG);

    /* 清除 EXINT 线 17 标志 */
    exint_flag_clear(EXINT_LINE_17);
  }
}
```

中断配置示例 2：以 AT32F403A 的闹钟为例，使用 RTC_IRQn 中断向量

```
  /* 使能闹钟中断 NVIC 向量：RTC_IRQn */
  nvic_irq_enable(RTC_IRQn, 0, 1);

  /* 设置闹钟值 */
  rtc_alarm_set(seccount);

  /* 使能闹钟中断 */
  rtc_interrupt_enable(RTC_TA_INT, TRUE);
```

中断处理函数

```
void RTC_IRQHandler(void)
{
  if(rtc_flag_get(RTC_TA_FLAG) != RESET)
  {
    /* 清除闹钟标志 */
    rtc_flag_clear(RTC_TA_FLAG);
  }
}
```

3 电池供电域功能

3.1 电池供电数据寄存器

电池供电域一共提供了 42 个 16 位电池供电数据寄存器，可以在只由电池供电下保存数据，不会被系统复位所复位，只能通过电池供电域复位或入侵事件进行复位。在写电池供电数据寄存器时，需要先解除读保护，解锁方式同 2.1 章节相同。

电池供电域数据操作相关函数

写电池供电数据寄存器

```
void bpr_data_write(bpr_data_type bpr_data, uint16_t data_value);
```

读电池供电数据寄存器

```
uint16_t bpr_data_read(bpr_data_type bpr_data);
```

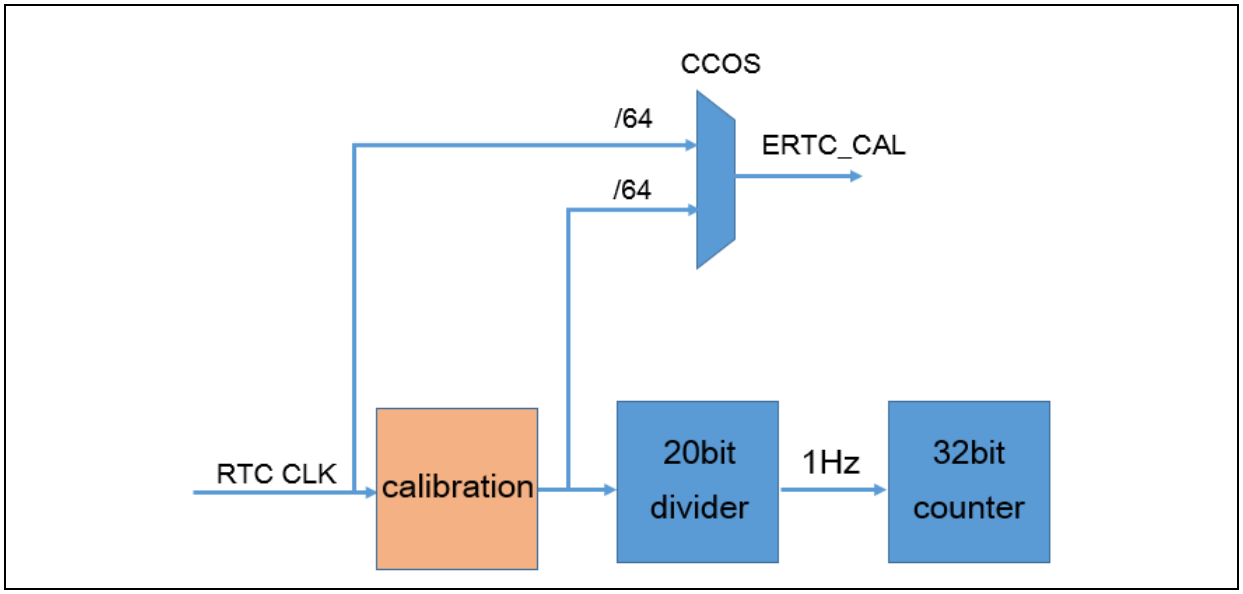
电池供电域复位

```
void bpr_reset(void);
```

3.2 RTC 校准

电池供电域还提供了 RTC 校准功能，通过 RTC_CALVAL 寄存器进行配置。

图 6. RTC 校准



当 RTC_CLK 为 32.768kHz 时，校准周期为 2^{20} 个 RTC_CLK 约 32 秒。CALVAL[7：0]值指定了 2^{20} 个 RTC_CLK 中忽略的脉冲数，最多可忽略 127 个脉冲，这可以将时钟调慢，调慢范围为 0~121ppm。

可以选择将校准前或校准后的 RTC 时钟 64 分频后输出到 PC13 脚。

校准设置相关函数

校准值设置函数

```
void bpr_rtc_clock_calibration_value_set(uint8_t calibration_value);
```

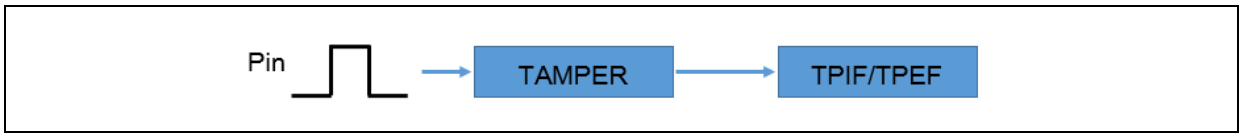
校准时钟输出设置函数

```
void bpr_rtc_output_select(bpr_rtc_output_type output_source);
```

3.3 入侵检测

电池供电域提供了 1 组入侵检测 TAMPER，当在发生入侵事件时，TPEF 标志位置 1，同时将自动清除电池供电数据寄存器(RTC_BPRx)的值；若已使能入侵中断，将产生入侵中断，同时 TPIF 标志位置 1。入侵检测引脚固定为 PC13。

图 7. 入侵检测



入侵检测模式分为高电平检测和低电平检测。

入侵检测相关函数

入侵检测有效电平设置

```
void bpr_tamper_pin_active_level_set(bpr_tamper_pin_active_level_type active_level);
```

入侵检测使能

```
void bpr_tamper_pin_enable(confirm_state new_state);
```

入侵检测标志获取

```
flag_status bpr_flag_get(uint32_t flag);
```

入侵检测标准清除

```
void bpr_flag_clear(uint32_t flag);
```

入侵检测中断使能

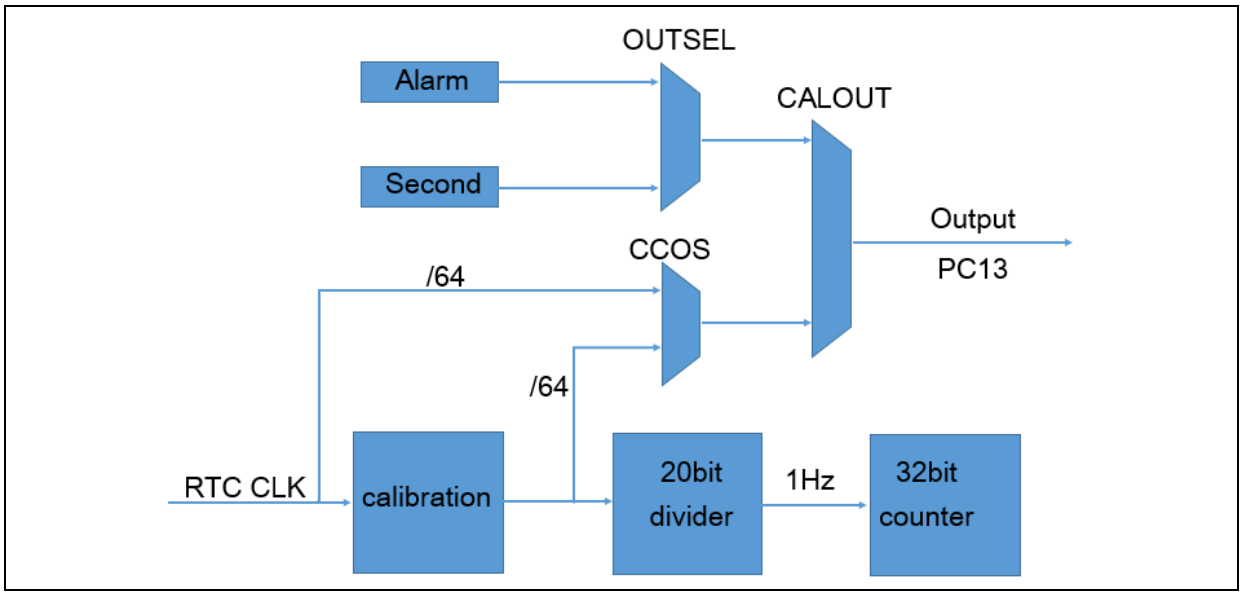
```
void bpr_interrupt_enable(confirm_state new_state);
```

3.4 事件输出功能

电池供电域提供了一组复用功能输出，在 PC13 脚可以输出以下事件：

- 校准输出：校准前 64 分频输出、校准后 64 分频输出。
- 事件输出：闹钟事件、秒事件

图 8. 事件输出



当输出模式为事件输出时（闹钟事件、秒事件），可以通过 OUTM 选择输出类型为脉冲输出（输出脉冲的宽度为一个 RTC 时钟的周期）或者是翻转输出（每来一次闹钟或秒输出事件，相对应管脚翻转一次）。

事件输出相关函数

事件输出设置并使能

```
void bpr_rtc_output_select(bpr_rtc_output_type output_source);
```

4 案例 使用日历以及闹钟功能

4.1 功能简介

演示日历功能、闹钟功能的使用。

4.2 资源准备

1) 硬件环境：

对应产品型号的 AT-START BOARD

2) 软件环境：

project\at_start_f4xx\examples\rtc\calendar

注：所有project都是基于keil 5而建立，若用户需要在其他编译环境上使用，请参考AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates中各种编译环境（例如IAR6/7,keil 4/5）进行简单修改即可。

4.3 软件设计

1) 配置流程

- 开启 PWC、BPR 时钟
- 解锁电池供电域写保护
- 检查日历是否已经初始化，如果正确就跳过初始化，如果不正确就初始化日历以及闹钟
- 主函数里每秒打印一次日历信息
- 在 21-05-01 12:00:05 时刻发生闹钟。

2) 代码介绍

- main 函数代码描述

```
int main(void)
{
  calendar_type time_struct;

  /* 配置 NVIC 优先级组 */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

  /* 初始化系统时钟 */
  system_clock_config();

  /* 初始化 ATSTART 板子资源 */
  at32_board_init();

  /* 初始化串口 */
  uart_print_init(115200);

  /* RTC 初始化 */
  time_struct.year  = 2021;
  time_struct.month = 5;
```

```
  time_struct.date   = 1;
  time_struct.hour   = 12;
  time_struct.min    = 0;
  time_struct.sec    = 0;
  rtc_init(&time_struct);

  /* 闹钟初始化 */
  alarm_init();

  printf("initial ok\r\n");

  while(1)
  {
    /* 秒更新了 */
    if(rtc_flag_get(RTC_TS_FLAG) != RESET)
    {
      at32_led_toggle(LED3);

      /* 获取当前日历 */
      rtc_time_get();

      /* 打印日期：年-月-日 */
      printf("%d/%d/%d ", calendar.year, calendar.month, calendar.date);

      /* 打印时间：时：分：秒 */
      printf("%02d:%02d:%02d %s\r\n", calendar.hour, calendar.min, calendar.sec,
weekday_table[calendar.week]);

      /* 等待上一次寄存器配置同步完成 */
      rtc_wait_config_finish();

      /* 清除秒钟标志 */
      rtc_flag_clear(RTC_TS_FLAG);

      /* 等待寄存器配置同步完成 */
      rtc_wait_config_finish();
    }
  }
}
```

■ RTC 初始化 rtc_init 函数代码描述

```
uint8_t rtc_init(calendar_type *calendar)
{
  /* 开启 PWC BPR 时钟 */
  crm_periph_clock_enable(CRM_PWC_PERIPH_CLOCK, TRUE);
```

```
crm_periph_clock_enable(CRM_BPR_PERIPH_CLOCK, TRUE);

/*解锁电池供电域写保护 */
pwc_battery_powered_domain_access(TRUE);

/* 检测 RTC 是否初始化 */
if(bpr_data_read(BPR_DATA1) != 0x1234)
{
  /* 复位电池供电域寄存器 */
  bpr_reset();

  /* 使能 LEXT 时钟 */
  crm_clock_source_enable(CRM_CLOCK_SOURCE_LEXT, TRUE);
  /* 等待 LEXT 时钟稳定 */
  while(crm_flag_get(CRM_LEXT_STABLE_FLAG) == RESET);
  /* 选择 RTC 时钟源 */
  crm_rtc_clock_select(CRM_RTC_CLOCK_LEXT);

  /* 使能 RTC 时钟 */
  crm_rtc_clock_enable(TRUE);

  /* 等待 RTC 寄存器同步 */
  rtc_wait_update_finish();

  /* 等待上一次寄存器配置同步完成 */
  rtc_wait_config_finish();

  /* 配置 RTC 分频器 */
  rtc_divider_set(32767);

   /* 等待上一次寄存器配置同步完成 */
  rtc_wait_config_finish();

  /* 设置时间 */
  rtc_time_set(calendar);

  /* 写入标志到电池供电域寄存器 */
  bpr_data_write(BPR_DATA1, 0x1234);

  return 1;
}
else
{
  /* 等待 RTC 寄存器同步 */
  rtc_wait_update_finish();
```

```
    /* 等待上一次寄存器配置同步完成 */
    rtc_wait_config_finish();

    return 0;
  }
}
```

- 闹钟中断函数代码描述

```
void RTC_IRQHandler(void)
{
  /*闹钟中断标志置起 */
  if(rtc_flag_get(RTC_TA_FLAG) != RESET)
  {
    at32_led_toggle(LED4);

    /* 清除闹钟中断标志 */
    rtc_flag_clear(RTC_TA_FLAG);
  }
}
```

4.4 实验效果

- 信息通过串口打印出来，在电脑上通过串口助手观看打印信息。
- 主函数里每秒打印一次日历信息。
- 在 21-05-01 12:00:05 时刻发生闹钟，发生闹钟时 LED4 点亮。

5 案例 使用 LICK 时钟并校准

5.1 功能简介

使用 LICK 时钟作为 RTC 时钟，并通过定时器测量出 LICK 时钟频率，通过得到的频率值，调整 RTC 分频，达到在一定范围内校准时间的效果。

5.2 资源准备

1) 硬件环境：

对应产品型号的 AT-START BOARD

2) 软件环境：

project\at_start_f4xx\examples\rtc\lick_calibration

注：所有project都是基于keil 5而建立，若用户需要在其他编译环境上使用，请参考AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates中各种编译环境（例如IAR6/7,keil 4/5）进行简单修改即可。

5.3 软件设计

1) 配置流程

- RTC 初始化
- 配置测量 LICK 频率的定时器
- 根据测量到的频率重新配置 RTC 分频

2) 代码介绍

- main 函数代码描述

```
int main(void)
{
  tmr_input_config_type tmr_ic_init_structure;

  /* 初始化系统时钟 */
  system_clock_config();

  /* 初始化 ATSTART 板子资源 */
  at32_board_init();

  /* 初始化串口 */
  uart_print_init(115200);

  /* RTC 初始化 */
  rtc_configuration();

  printf("\r\n\nstart calib\r\n\r\n");

  /* 获取系统时钟频率 */
  crm_clocks_freq_get(&crm_clocks);
```

```
/* 使能定时器 */
crm_periph_clock_enable(CRM_TMR5_PERIPH_CLOCK, TRUE);
crm_periph_clock_enable(CRM_IOMUX_PERIPH_CLOCK, TRUE);

/* 将 LICK 连接到定时器 */
gpio_pin_remap_config(TMR5CH4_MUX, TRUE);

/* 定时器初始化 */
tmr_base_init(TMR5, 0xFFFF, 0);
tmr_cnt_dir_set(TMR5, TMR_COUNT_UP);
tmr_clock_source_div_set(TMR5, TMR_CLOCK_DIV1);

/* 定时器初始化 */
tmr_input_default_para_init(&tmr_ic_init_structure);
tmr_ic_init_structure.input_channel_select = TMR_SELECT_CHANNEL_4;
tmr_ic_init_structure.input_polarity_select = TMR_INPUT_RISING_EDGE;
tmr_ic_init_structure.input_mapped_select = TMR_CC_CHANNEL_MAPPED_DIRECT;
tmr_ic_init_structure.input_filter_value = 0;
tmr_input_channel_init(TMR5, &tmr_ic_init_structure, TMR_CHANNEL_INPUT_DIV_1);

/* 初始化变量 */
operationcomplete = 0;

/* 使能定时器输入捕获 */
tmr_counter_enable(TMR5, TRUE);

/* 清除定时器标志 */
tmr_flag_get(TMR5, TMR_C4_FLAG);

/* 使能定时器 */
tmr_interrupt_enable(TMR5, TMR_C4_INT, TRUE);

/* 初始化中断 */
nvic_configuration();

/* 等待测量完成 */
while(operationcomplete != 2);

/* 计算 LICK 频率 */
if(periodvalue != 0)
{
  lickfreq = (uint32_t)((uint32_t)(crm_clocks.apb1_freq * 2) / (uint32_t)periodvalue);
}

printf("apb1_freq      = %d\r\n", crm_clocks.apb1_freq);
printf("period_value = %d\r\n", periodvalue);
```

```
  printf("lick_freq       = %d\r\n", lickfreq);

  /* 调整 RTC 分频 */
  rtc_divider_set((lickfreq - 1));

  /* 等待寄存器写完成 */
  rtc_wait_config_finish();

  /* turn on led2 */
  at32_led_on(LED2);

  while(1)
  {
  }
}
```

5.4 实验效果

- 信息通过串口打印出来，在电脑上通过串口助手观看打印信息。
- 通串口打印出当前测量出的 LICK 的频率以及分频值。
- 每秒钟打印一次日历。

6 案例 读写电池供电数据寄存器

6.1 功能简介

对电池供电数据寄存器(RTC_BPRx)进行读写访问。

6.2 资源准备

1) 硬件环境：

对应产品型号的 AT-START BOARD

2) 软件环境：

project\at_start_f4xx\examples\bpr\bpr_data

注：所有project都是基于keil 5而建立，若用户需要在其他编译环境上使用，请参考AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates中各种编译环境（例如IAR6/7,keil 4/5）进行简单修改即可。

6.3 软件设计

1) 配置流程

- 开启 PWC、BPR 时钟
- 解锁电池供电域写保护
- 检查电池供电域数据是否正确
- 关闭 PWC、BPR 时钟降低功耗

2) 代码介绍

- main 函数代码描述

```
int main(void)
{
  /* 初始化系统时钟 */
  system_clock_config();

  /* 初始化串口 */
  uart_print_init(115200);

  /* 开启 PWC BPR 时钟 */
  crm_periph_clock_enable(CRM_PWC_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_BPR_PERIPH_CLOCK, TRUE);

  /*解锁电池供电域写保护 */
  pwc_battery_powered_domain_access(TRUE);

  /* 清除入侵检测标志 */
  bpr_flag_clear(BPR_TAMPER_EVENT_FLAG);

  /* 检查电池供电域数据 */
  if(bpr_reg_check() == TRUE)
  {
```

```
    printf("bpr reg => none reset\r\n");
  }
  else
  {
    printf("bpr reg => reset\r\n");
  }

  /* 复位电池供电域寄存器 */
  bpr_reset();

  /*写电池供电域寄存器 */
  bpr_reg_write();

  /* 检查电池供电域数据 */
  if(bpr_reg_check() == TRUE)
  {
    printf("write bpr reg ok\r\n");
  }
  else
  {
    printf("write bpr reg fail\r\n");
  }

  /* 关闭 PWC BPR 时钟降低功耗 */
  crm_periph_clock_enable(CRM_PWC_PERIPH_CLOCK, FALSE);
  crm_periph_clock_enable(CRM_BPR_PERIPH_CLOCK, FALSE);

  while(1)
  {
  }
}
```

6.4 实验效果

- 信息通过串口打印出来，在电脑上通过串口助手观看打印信息。
- 如果寄存器里数据正确打印 bpr reg => none reset。
- 如果寄存器里数据正确打印 bpr reg => reset。
- 主函数里每秒打印一次日历信息。

7 案例 入侵检测

7.1 功能简介

演示入侵检测功能使用，PC13 脚当检测到一个上升沿后将触发入侵检测，当入侵事件发生时，电池供电数据寄存器将会被清除。

7.2 资源准备

1) 硬件环境：

对应产品型号的 AT-START BOARD

2) 软件环境：

project\at_start_f4xx\examples\bpr\tamper

注：所有project都是基于keil 5而建立，若用户需要在其他编译环境上使用，请参考AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates中各种编译环境（例如IAR6/7,keil 4/5）进行简单修改即可。

7.3 软件设计

1) 配置流程

- RTC 初始化
- 初始化入侵检测
- 初始化电池供电寄存器

2) 代码介绍

- main 函数代码描述

```
int main(void)
{
  /* 配置 NVIC 优先级组 */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

  /* 初始化系统时钟 */
  system_clock_config();

  /* 初始化 ATSTART 板子资源 */
  at32_board_init();

  /* 入侵检测中断配置 */
  nvic_irq_enable(TAMPER_IRQn, 0, 0);

  /* 开启 PWC BPR 时钟 */
  crm_periph_clock_enable(CRM_PWC_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_BPR_PERIPH_CLOCK, TRUE);

  /* 解锁电池供电域写保护 */
  pwc_battery_powered_domain_access(TRUE);
```

```
  /* 禁止入侵检测功能 */
  bpr_tamper_pin_enable(FALSE);

  /* 禁止入侵检测中断 */
  bpr_interrupt_enable(FALSE);

  /* 配置入侵检测为低电平检测 */
  bpr_tamper_pin_active_level_set(BPR_TAMPER_PIN_ACTIVE_LOW);

  /* 清除入侵检测标志 */
  bpr_flag_clear(BPR_TAMPER_EVENT_FLAG);

  /* 写数据到电池供电域寄存器 */
  bpr_reg_write();

  /* 使能入侵检测中断 */
  bpr_interrupt_enable(TRUE);

  /* 使能入侵检测 */
  bpr_tamper_pin_enable(TRUE);

  /* 检查电池供电域寄存器值 */
  if(bpr_reg_check() == TRUE)
  {
    at32_led_on(LED2);
  }
  else
  {
    at32_led_off(LED2);
  }

  while(1)
  {
  }
}
```

■ 入侵检测中断处理函数代码描述

```
void TAMPER_IRQHandler(void)
{
  if(bpr_flag_get(BPR_TAMPER_INTERRUPT_FLAG) != RESET)
  {
    /* 检查电池供电寄存器是否被清除掉 */
    if(bpr_reg_judge() == 0)
    {
      /* ok, bpr registers are reset as expected */
```

```
        at32_led_on(LED3);
      }
      else
      {
        /* bpr registers are not reset */
        at32_led_off(LED3);
      }

      /* 清除入侵检测中断标志 */
      bpr_flag_clear(BPR_TAMPER_INTERRUPT_FLAG);

      /* 清除入侵检测事件标志 */
      bpr_flag_clear(BPR_TAMPER_EVENT_FLAG);

      /* 禁止入侵检测 */
      bpr_tamper_pin_enable(FALSE);

      /* 使能入侵检测 */
      bpr_tamper_pin_enable(TRUE);
    }
}
```

7.4 实验效果

- 信息通过串口打印出来，在电脑上通过串口助手观看打印信息。
- 当发生入侵事件时（PC13 出现低电平），LED3 亮表示电池供电寄存器被清除。

8 文档版本历史

表 8. 文档版本历史

日期	版本	变更
2021.12.22	2.0.0	最初版本
2022.02.11	2.0.1	2.6章节增加闹钟中断处理代码

重要通知 - 请仔细阅读

买方自行负责对本文所述雅特力产品和服务的选择和使用，雅特力概不承担与选择或使用本文所述雅特力产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为雅特力授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在雅特力的销售条款中另有说明，否则，雅特力对雅特力产品的使用和/或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途（及其依据任何司法管辖区的法律的对应情况），或侵犯任何专利、版权或其他知识产权的默示保证。

雅特力产品并非设计或专门用于下列用途的产品：（A）对安全性有特别要求的应用，例如：生命支持、主动植入设备或对产品功能安全有要求的系统；（B）航空应用；（C）航天应用或航天环境；（D）武器，且/或（E）其他可能导致人身伤害、死亡及财产损害的应用。如果采购商擅自将其用于前述应用，即使采购商向雅特力发出了书面通知，风险及法律责任仍将由采购商单独承担，且采购商应独力负责在前述应用中满足所有法律和法规要求。

经销的雅特力产品如有不同于本文档中提出的声明和/或技术特点的规定，将立即导致雅特力针对本文所述雅特力产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大雅特力的任何责任。