第十四章 RTC 实时计数器

实时计数器的缩写 RTC 是 Real-time counter，在低频时钟源 LFCLK 上提供一个通用的低功耗定时器。

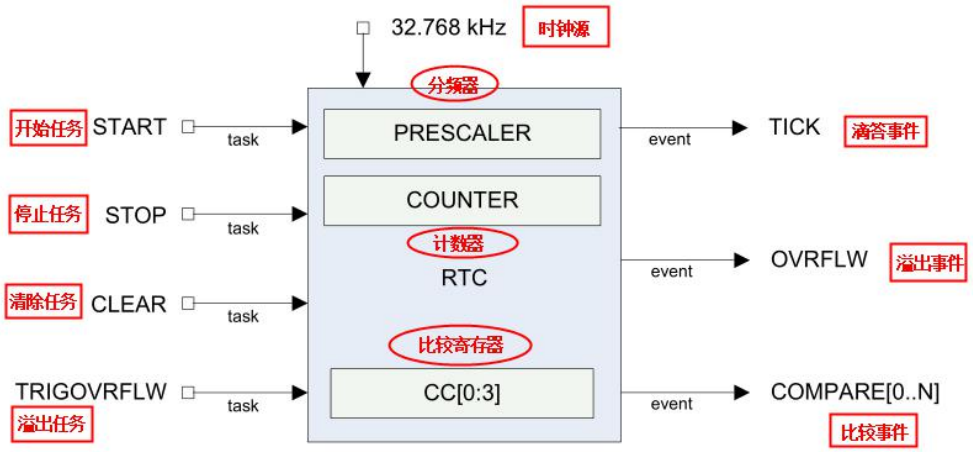
实时时钟的缩写是 RTC(Real-time Clock)，它为人们提供精确的实时时间,或者为电子系统提供精确的时间基准,目前实时时钟芯片大多采用精度较高的晶体振荡器作为时钟源。钟芯片为了在主电源掉电时，还可以工作，需要外加电池供电。

因此一定要注意区别 Real-time counter 和 Real-time Clock，nrf52832 芯片只提供 Real-time counter 功能。

14.1 原理分析

14.1.1 RTC 的内部结构

RTC 模块具有一个 24 位计数器、12 位预分频器、捕获/比较寄存器和一个滴答事件生成器用于低功耗、无滴答 RTOS 的实现。RTC 模块需要低频时钟源提供运行时钟，因此在使用 RTC 之前，软件必须明确启动 LFCLK 时钟。RTC 模块的内部结构如下图所示：



实时计数器 RTC 是一个 24 位的低频时钟，带分频、TICK、比较和溢出事件。

●RTC 时钟源

实时计数器 RTC 运行于 LFCLK 下。COUNTER 的分辨率为 30.517us。当 HFCLK 关闭和 PCLKK16M 也不可以用时，RTC 也可运行。

●分频器溢出时间和分辨率

RTC 的计数器 COUNTER 增量的计数频率按照下面公式计算：

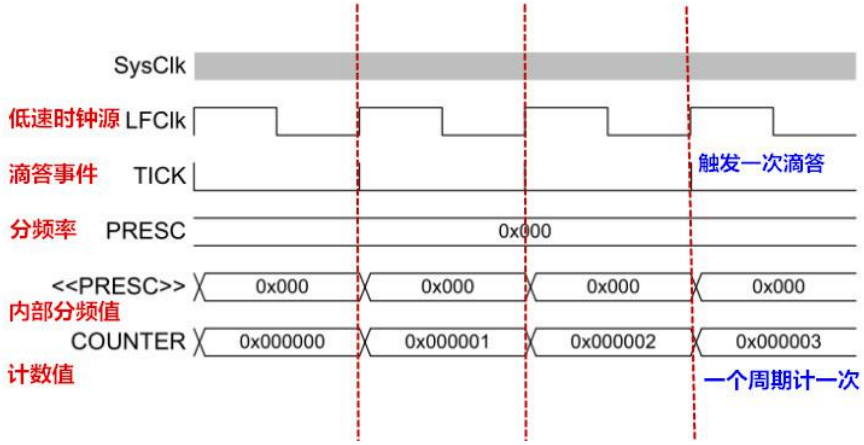
$$f_{RTC} = \frac{32,768kHz}{PRESCALER + 1}$$

其中 PRESCALER 为分频寄存器，该寄存器在 RTC 停止时可读可写。在 RTC 开启时，PRESCALER 寄存器只能读溢出，写无效。PRESCALER 在 START、CLEAR 和 TRIGOVRFLW 事件时都会重新启动，分频值被锁存在这些任务的内部寄存器(<<PRESC>>)。

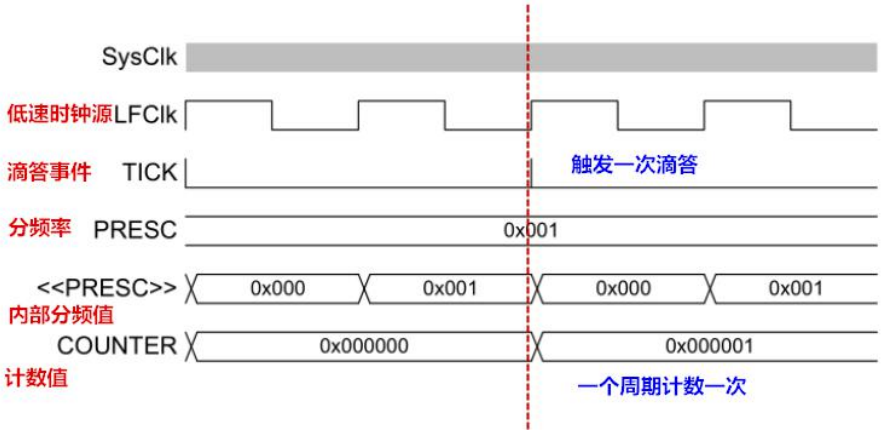
●COUNTER 计数器寄存器

当内部 PRESCALER 寄存器（<< PRESC >>）为 0x00 时，COUNTER 在 LFCLK 上递增。<< PRESC >> 从 PRESCALER 寄存器重新加载。 如果启用，TICK 事件将在 COUNTER 的每个增量处发生。默认情况下禁用 TICK 事件。

例如：PRESCALER 寄存器为分频为 0，则计数时钟频率为 LFClk 的频率。一个周期内部 PRESCALER 寄存器（<< PRESC >>）保存为 0x00，COUNTER 递增计数一次。同时出发一次滴答事件。



PRESCALER 寄存器为分频为 1，则计数时钟频率为 LFClk/2 的频率，那么两个时钟周期为一个计数周期。在一个计数周期中，内部 PRESCALER 寄存器（<< PRESC >>）为 0x00 时，COUNTER 递增计数一次。同时出发一次滴答事件。



14.1.2 RTC 的事件

RTC 输出会产生三种事件类型，如下描述：

●溢出事件

RTC 计数器提供一个溢出事件。在模块输入端通过 TRIGOVRFLW 溢出任务将 COUNTER 计数器的值设置为 0xFFFFF0。计数器计数 16 次， COUNTER 计数到 0xFFFFFF，从 0xFFFFFF 溢出到 0 时就发生

OVRFLW 溢出事件。

重要说明：默认情况下禁用 OVRFLW 事件。

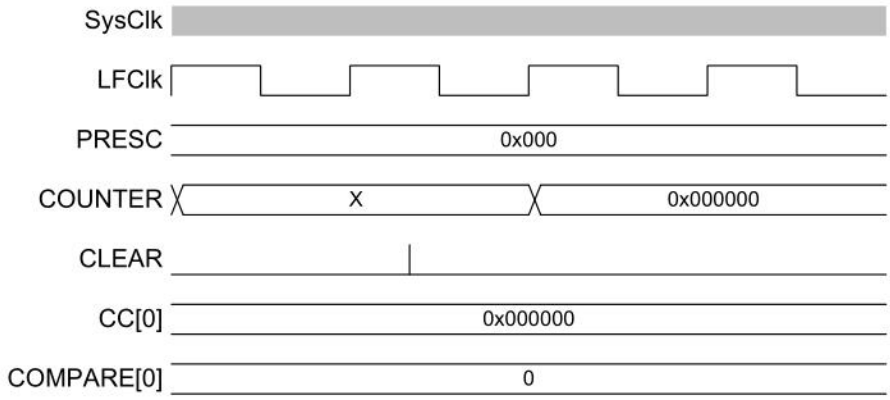
●TICK 事件

TICK 事件可实现低功耗“无滴答”RTOS 实现，因为它可选择为 RTOS 提供常规中断源，而无需使用 ARM ® SysTick 功能。使用 RTC TICK 事件而不是 SysTick 可以在关闭 CPU 的同时保持 RTOS 调度处于活动状态。

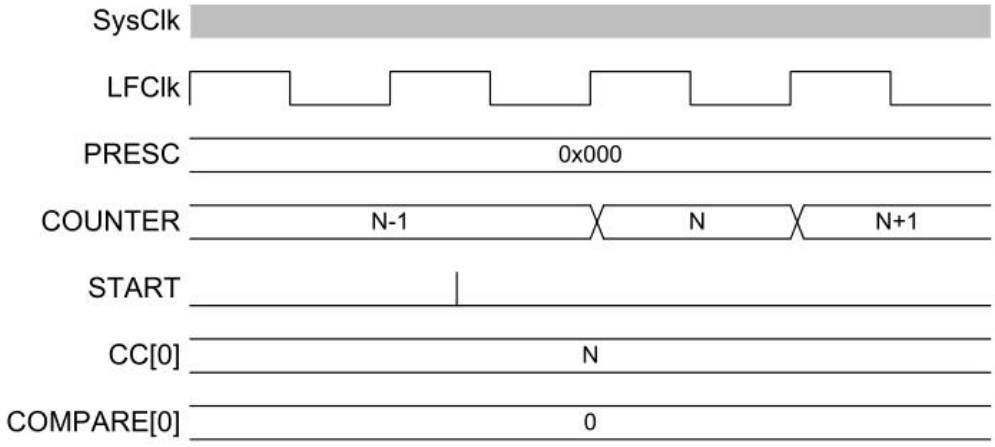
●比较事件

RTC 模块提供 COMPARE 比较事件的输出，当 COUNTER 计数器里的值和 CC[n]比较寄存器内存的值相等的时候，就会触发 COMPARE 比较事件。但是设置 CC[n]比较寄存器时，应注意以下 RTC 比较事件的行为：

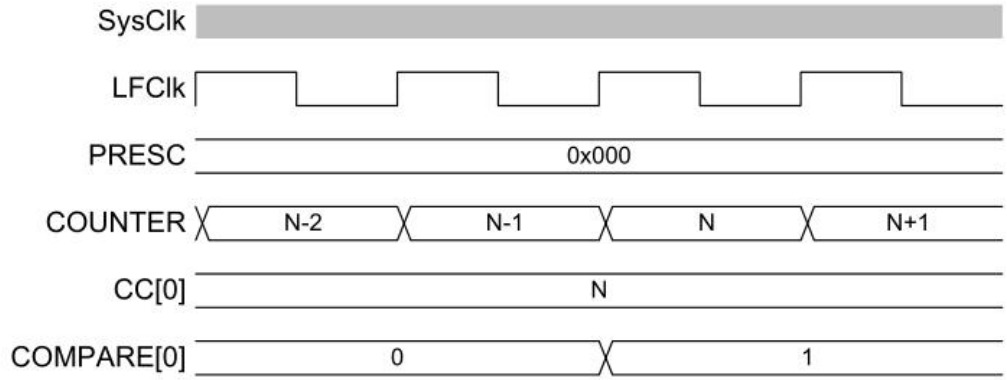
• 如果在设置 CLEAR 任务时 CC[n]寄存器值为 0，则不会触发 COMPARE 事件。



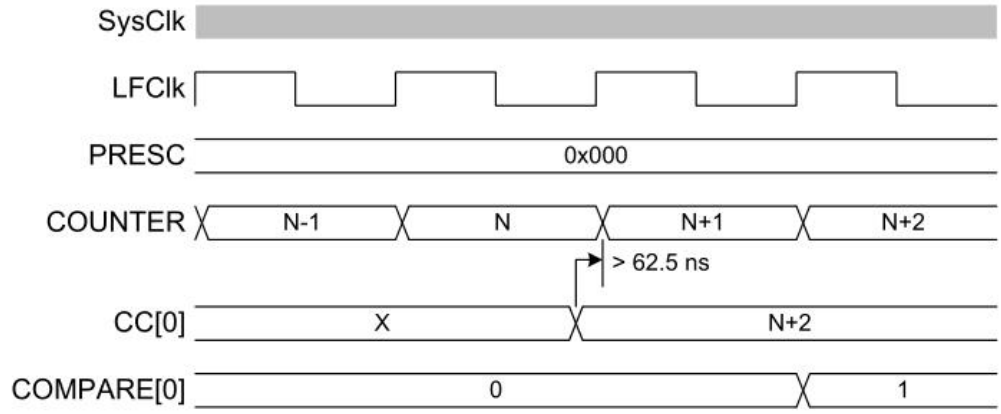
• 如果 CC[n]寄存器为 N 且设置 START 任务时 COUNTER 值为 N，则不会触发 COMPARE 事件。



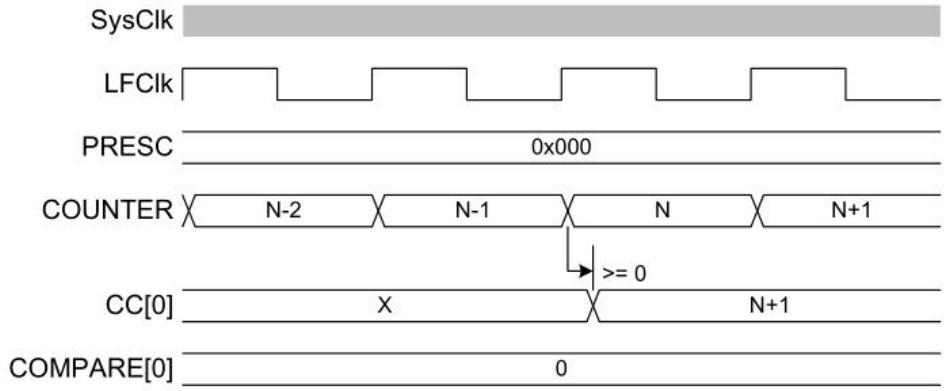
• 当 CC[n]寄存器为 N 且 COUNTER 值从 N-1 转换为 N 时，发生 COMPARE。



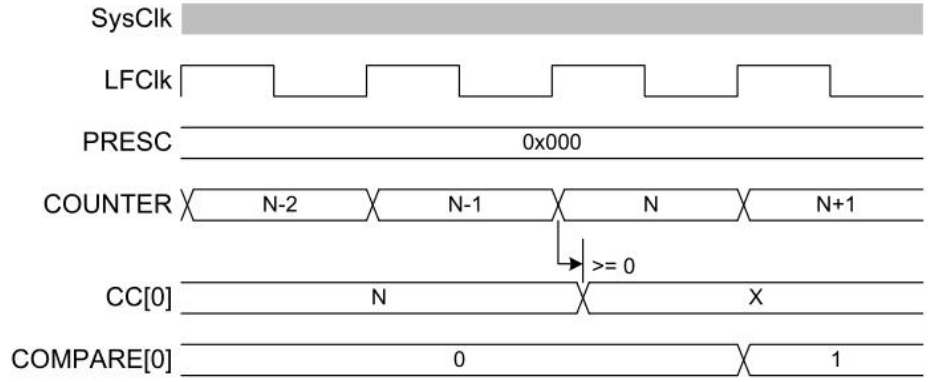
• 如果 COUNTER 为 N，则将 N + 2 写入 CC[n]寄存器可保证在 N + 2 处触发 COMPARE 事件。



• 如果 COUNTER 为 N，则将 N 或 N + 1 写入 CC 寄存器可能不会触发 COMPARE 事件。



• 如果 COUNTER 为 N 且当写入新 CC 值时当前 CC[n]寄存器值为 N + 1 或 N + 2，则在新值生效之前，匹配可以触发先前的 CC[n]值。 如果在写入新值时当前 CC[n]值大于 N + 2，则由于旧值将不会发生事件。



●事件控制

上面总结了 RTC 的三个事件类型。为了优化 RTC 功耗，RTC 不需要使用的事件，可以单独禁用该事件，以防止在触发这些事件时请求 PCLK16M 和 HFCLK 时钟，消耗电流增加功耗。这种功能称为事件控制，RTC 模块提供 EVTEN 寄存器来实现管理。

例如，如果应用程序不需要 TICK 事件，则应该禁用该事件。因为它经常发生，如果 HFCLK 可以长时间关闭电源，则可能会增加功耗。

EVTEN 寄存器：地址偏移量 0x340，启用或禁用事件：

位数	Field	Value ID	Value	描述
第 1 位	TICK			启用或禁用滴答事件
		Disabled	0	禁用。
		Enabled	1	启用。
第 2 位	OVRFLW			启用或禁用溢出事件
		Disabled	0	禁用。
		Enabled	1	启用。
第 16 位	COMPARE0			启用或禁用比较事件 0
		Disabled	0	禁用。
		Enabled	1	启用。
第 17 位	COMPARE1			启用或禁用比较事件 0
		Disabled	0	禁用。
		Enabled	1	启用。
第 18 位	COMPARE2			启用或禁用比较事件 1
		Disabled	0	禁用。
		Enabled	1	启用。
第 19 位	COMPARE3			启用或禁用比较事件 2
		Disabled	0	禁用。
		Enabled	1	启用。

●任务和事件抖动/延迟

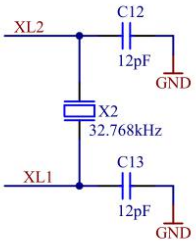
RTC 中的抖动或延迟是由于外设时钟是低频时钟（LFCLK）它与更快的 PCLK16M 不同步。外设接口（PCLK16M 域的一部分）中的寄存器在 LFCLK 域中具有一组镜像寄存器。例如，可从 CPU 访问的 COUNTER 值位于 PCLK16M 域中，并在从 LFCLK 域中名为 COUNTER 的内部寄存器读取时锁存。COUNTER 是每次 RTC 滴答时实际修改的寄存器,这些寄存器必须在时钟域（PCLK16M 和

LFCLK）之间同步，因此会出现任务和事件抖动/延迟，具体可以参考数据手册。

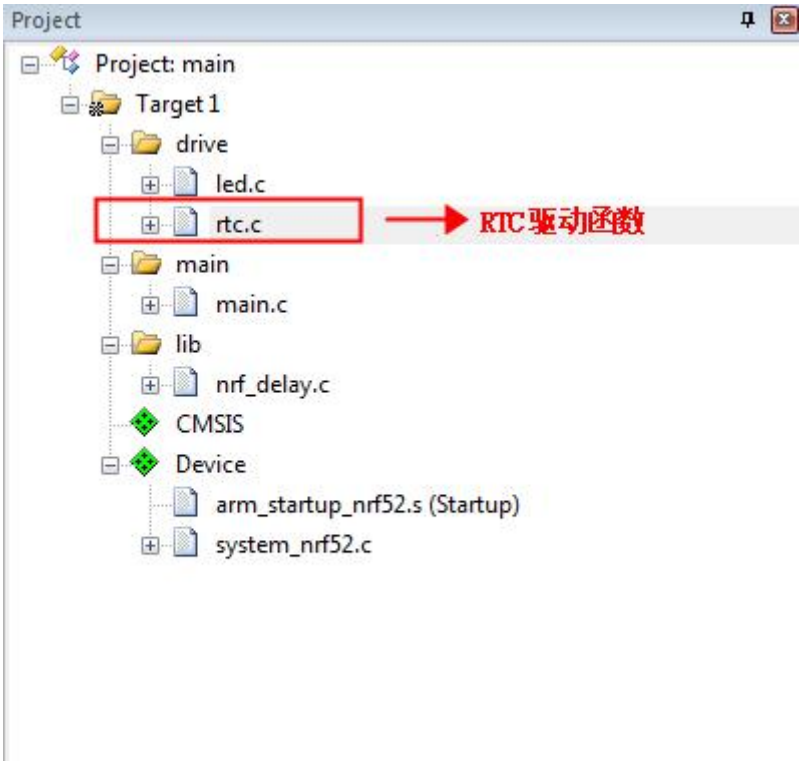
14.2 应用实例编写

14.2.1 TICK 滴答与比较事件应用

硬件方面，为了降低功耗，采用外设低速晶振。如下图所示： 青云 QY-nRF52832 开发板上，通过管脚 P0.27 和管脚 P0.26 连接低速外部晶振，晶振大小为 32.768KHZ。



在代码文件中，实验五建立了一个演示历程，我们打开看看需要那些库文件。打开 user 文件夹中的 RTC 工程：



如上图所示：只需要自己编写红色框框里的两个文件就 OK 了,因为采用子函数的方式其中 led.c 在控制 LED 灯的时候已经写好，现在我们就来讨论下如何编写 rtc.c 这个驱动子文件。

rtc.c 文件主要是要起到两个作用：第一：初始化使能 RTC 相关参数。第二：设置 RTC 匹配和比较中断。完成这两个功能够就可以在 main.c 文件中直接调用本驱动了。

下面我们就结合寄存器来详细分析下 RTC 的设置：nrf528322 的 RTC 的基本 RTC 时钟结构，

RTC 在广义的 MCU 方面的定义就是一个独立的定时器，因此其寄存器的设置应该和 time 定时器的设置相似，这个从 nrf528322 的手册上寄存器定义可以很明显的看出来。回忆一下之前 time 定时器的内容，time 是高速时钟 HFCLK 提供时钟，而 RTC 则是由 LFCLK 提供时钟。下面我们来配置时钟源，设置代码如下

```
01.    /** 启动 LFCLK 晶振功能*/
02.   void lfclk_config(void)
03.  {
04.       NRF_CLOCK->LFCLKSRC = (CLOCK_LFCLKSRC_SRC_Xtal <<
05.                                          CLOCK_LFCLKSRC_SRC_Pos);//设置 32K 时钟为时钟源
06.       NRF_CLOCK->EVENTS_LFCLKSTARTED   = 0;//关 32K 震荡 event 事件
07.       NRF_CLOCK->TASKS_LFCLKSTART       = 1;//开 32K 震荡 task 任务
08.       while (NRF_CLOCK->EVENTS_LFCLKSTARTED == 0)//等待 32K 震荡 event 事件发生
09.       {
10.         //等待循环
11.       }
12.       NRF_CLOCK->EVENTS_LFCLKSTARTED = 0;
13.  }
```

Register	Offset	Description
TASKS_HFCLKSTART	0x000	Start HFCLK crystal oscillator
TASKS_HFCLKSTOP	0x004	Stop HFCLK crystal oscillator
TASKS_LFCLKSTART	0x008	Start LFCLK source
TASKS_LFCLKSTOP	0x00C	Stop LFCLK source
TASKS_CAL	0x010	Start calibration of LFRC or LFULP oscillator
TASKS_CTSTART	0x014	Start calibration timer
TASKS_CTSTOP	0x018	Stop calibration timer
EVENTS_HFCLKSTARTED	0x100	HFCLK oscillator started
EVENTS_LFCLKSTARTED	0x104	LFCLK started
EVENTS_DONE	0x10C	Calibration of LFCLK RC oscillator complete event
EVENTS_CTTO	0x110	Calibration timer timeout
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
HFCLKRUN	0x408	Status indicating that HFCLKSTART task has been triggered
HFCLKSTAT	0x40C	HFCLK status
LFCLKRUN	0x414	Status indicating that LFCLKSTART task has been triggered
LFCLKSTAT	0x418	LFCLK status
LFCLKSRCCOPY	0x41C	Copy of LFCLKSRC register, set when LFCLKSTART task was triggered
LFCLKSRC	0x518	Clock source for the LFCLK
CTIV	0x538	Calibration timer interval (retained register, same reset behaviour as RESETREAS)
TRACECONFIG	0x55C	Clocking options for the Trace Port debug interface

上面红色框框的几个寄存器就是代码里定义的寄存器。

1：时钟源的选择与配置：

首先配置时钟源的选择寄存器 NRF_CLOCK->LFCLKSRC 为低速时钟，低速时钟有三钟选择：

```
01. #define CLOCK_LFCLKSRC_SRC_RC (0UL) /* 内部 32KHz RC 时钟. */
02. #define CLOCK_LFCLKSRC_SRC_Xtal (1UL) /* 外部 32KHz 晶振振荡*/
03. #define CLOCK_LFCLKSRC_SRC_Synth (2UL) /*内部 32KHz 从 HFCLK 系统时钟产生. */
```

分别为：内部 32KHz RC 时钟模块：内部包含的 RC 模块，不需要外接任何晶振。

外部 32KHz 晶振振荡：外部外接的 32.768kHz 的低速晶振产生的振荡时钟。

内部 32KHz 从 HFCLK 系统时钟产生：通过高速时钟分频参数的 32KHz 时钟，依赖外接高

速晶振。

然后配置寄存器开启低速时钟。首先用寄存器器 TASKS_LFCLKSTART 开低速时钟振荡任务，打开后就会触发低速时钟开始事件，一旦低速时钟开始事件 EVENTS_LFCLKSTARTED 被置位，就开启了低速时钟，这时候就可以跳出等待循环。

我们选择外部低速时钟振荡作为时钟源。这里就把低速时钟源设置完成了。

2：RTC 模块的配置：

本实验的要求来实现 8Hz 的计数频率，实现计数时间 125ms 一次滴答，滴答后产生一次中断控制 led1 灯翻转一次。当翻转 24 次接近 3000ms，也就是 3s 的时候进行模拟比较报警，报警控制另一个 led2 灯点亮报警。来应用滴答事件和比较事件。

关于寄存器解释很清楚，关键是知道如何使用，我们直接对着代码段分析：

```
01.    void rtc_config(void)
02. {
03.       NVIC_EnableIRQ(RTC0_IRQn);                  // 使能 RTC 中断.
04.       NRF_RTC0->PRESCALER = COUNTER_PRESCALER;
05.      //12 位预分频器的计数器频率（ 32768 /（预分频器+1 ））
06.      //当 RTC 停止时候才能设置
07.      // 设置预分频值 prescaler
08.       NRF_RTC0->CC[0]   = COMPARE_COUNTERTIME * RTC_FREQUENCY;
09.    // 设置比较寄存器值
10.
11.       // 启用 Tick 事件和节拍中断
12.       NRF_RTC0->EVTENSET          = RTC_EVTENSET_TICK_Msk;
13.       NRF_RTC0->INTENSET          = RTC_INTENSET_TICK_Msk;
14.
15.       // 启用比较匹配事件和比较匹配中断
16.       NRF_RTC0->EVTENSET          = RTC_EVTENSET_COMPARE0_Msk;
17.       NRF_RTC0->INTENSET          = RTC_INTENSET_COMPARE0_Msk;
18. }
19.
```

第 3 行：上面一段代码的编写严格按照了寄存器要求进行，首先是使能 RTC 嵌套中断，这个 NVIC 是 cortex m 系列处理器通用的中断方式。

第 4 行~第 8 行：我们设计 RTC 的计数频率，根据手册给出的公式：

$$f_{RTC} = \frac{32,768kHz}{PRESCALER + 1}$$

根据这个公式，下面举两个例子：

（1）. 设置计数频率为 100 Hz (10 ms 一个计数周期)

代入公式 PRESCALER = round(32.768 kHz / 100 Hz) - 1 = 327

RTC 的时钟频率 fRTC = 99.9 Hz

10009.576 μs 一个计数周期

（2）. 设置计数频率为 8 Hz (125ms 一个计数周期)

代入公式 PRESCALER = round (32.768 kHz / 8 Hz) － 1 = 4095

RTC 的时钟频率 fRTC = 8 Hz

125 ms 一个计数周期

Prescaler	Counter resolution	Overflow
0	30.517 µs	512 seconds
2^8-1	7812.5 µs	131072 seconds
2^{12}-1	125 ms	582.542 hours

PRESCALER 寄存器设置 RTC 预分频计数器，这个寄存器是我们需要设置的，在代码中，根据要求的参数值进行设置：

```
20. #define LFCLK_FREQUENCY    (32768UL) //
21. #define RTC_FREQUENCY      (8UL)
22. #define   COUNTER_PRESCALER     ((LFCLK_FREQUENCY/RTC_FREQUENCY) - 1)
23. NRF_RTC0->PRESCALER = COUNTER_PRESCALER;
```

这个参数是更加上面举的例子 2 里。代入公式 PRESCALER = round (32.768 kHz / 8 Hz) － 1 = 4095 来设置这个 RTC 预分频计数器。也就是 125 ms 一个计数周期产生一次 TICK 滴答事件。

然后设置比较器值，根据分析需要翻转转 24 次才能接近 3000ms，因此比较次数也是 24 次，设置 CC[0] 寄存器的值为 24。

18.2.6 CC[N] (n=0..3)

Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RW	-	-	-	-	-	-	-	-	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

ID	Field	Value	Description
A			Compare value

设置完成后，使能计数器中断和比较中断：

EVTENSET 寄存器：使能事件通道

位数	Field	Value ID	Value	描述
第 1 位	TICK			写 1 去使能滴答事件通道
		Set	1	使能
		Disabled	0	读：禁用。
		Enabled	1	写：启用。
第 2 位	OVRFLW			写 1 去使能溢出事件通道
		Set	1	使能
		Disabled	0	读：禁用。
		Enabled	1	写：启用。
第 16 位	COMPARE0			写 1 去使能比较 0 事件通道
		Set	1	使能
		Disabled	0	读：禁用。
		Enabled	1	写：启用。
第 17 位	COMPARE1			写 1 去使能比较 1 事件通道
		Set	1	使能
		Disabled	0	读：禁用。
		Enabled	1	写：启用。
第 18 位	COMPARE2			写 1 去使能比较 2 事件通道

		Set	1	使能
		Disabled	0	读：禁用。
		Enabled	1	写：启用。
第 19 位	COMPARE3			写 1 去使能比较 3 事件通道
		Set	1	使能
		Disabled	0	读：禁用。
		Enabled	1	写：启用。

INTENSET 寄存器：使能中断寄存器

位数	Field	Value ID	Value	描述
第 1 位	TICK			写 1 去使能滴答事件中断
		Set	1	使能
		Disabled	0	读：禁用。
		Enabled	1	写：启用。
第 2 位	OVRFLW			写 1 去使能溢出事件中断
		Set	1	使能
		Disabled	0	读：禁用。
		Enabled	1	写：启用。
第 16 位	COMPARE0			写 1 去使能比较 0 事件中断
		Set	1	使能
		Disabled	0	读：禁用。
		Enabled	1	写：启用。
第 17 位	COMPARE1			写 1 去使能比较 1 事件中断
		Set	1	使能
		Disabled	0	读：禁用。
		Enabled	1	写：启用。
第 18 位	COMPARE2			写 1 去使能比较 2 事件中断
		Set	1	使能
		Disabled	0	读：禁用。
		Enabled	1	写：启用。
第 19 位	COMPARE3			写 1 去使能比较 3 事件中断
		Set	1	使能
		Disabled	0	读：禁用。
		Enabled	1	写：启用。

配置如下所示：

```
01.     //启用 Tick 事件和节拍中断
02.     NRF_RTC0->EVTENSET      = RTC_EVTENSET_TICK_Msk;
03.     NRF_RTC0->INTENSET      = RTC_INTENSET_TICK_Msk;
04.
05.     //启用比较匹配事件和比较匹配中断
06.     NRF_RTC0->EVTENSET      = RTC_EVTENSET_COMPARE0_Msk;
07.     NRF_RTC0->INTENSET      = RTC_INTENSET_COMPARE0_Msk;
```

设置中断功能后，中断做的内容就比较简单了，判断中断发生后翻转 LED 灯，两种中断事件，

一个是 RTC 滴答事件，一个是 RTC 比较事件。

```
01.    void RTC0_IRQHandler()
02. {    //判断滴答事件
03.      if ((NRF_RTC0->EVENTS_TICK != 0) &&
04.          ((NRF_RTC0->INTENSET & RTC_INTENSET_TICK_Msk) != 0))
05.      {
06.          NRF_RTC0->EVENTS_TICK = 0;//清除滴答事件
07.          LED1_Toggle();//翻转 LED1 灯
08.      }
09.      //判断比较事件
10.      if ((NRF_RTC0->EVENTS_COMPARE[0] != 0) &&
11.          ((NRF_RTC0->INTENSET & RTC_INTENSET_COMPARE0_Msk) != 0))
12.      {
13.          NRF_RTC0->EVENTS_COMPARE[0] = 0;//清除比较事件
14.          LED2_Toggle();//翻转 LED2 灯
15.      }
16. }
```

那么主函数就是十分的简单了，直接调用我们写好的驱动函数，LED 灯指示定时器相应的变化。函数如下所示：

```
01. /******************** (C) COPYRIGHT 2019 青风电子 ********************
02.  * 文件名   ：main
03.  * 描述     ：
04.  * 实验平台：青风 nrf52832 开发板
05.  * 描述     ：RTC 滴答中断和比较中断
06.  * 作者     ：青风
07.  * 店铺     ：qfv5.taobao.com
08. *********************************************************************/
09. #include "nrf52.h"
10. #include   "led.h"
11. #include   "rtc.h"
12.
13. int main(void)
14. {
15.     LED_Init();
16.    LED1_Close();
17.    LED2_Close();
18.     lfclk_config();//启动内部 LFCLK 晶振功能
19.     rtc_config();//配置 RTC
20.
21.     NRF_RTC0->TASKS_START = 1;//开启 rtc
22.     while (1)
23.      {
24.          // Do nothing.
```

```
25.     }
26. }
```

实验下载到青云 nRF52832 开发板后的实验现象如下，led1 灯 125ms 翻转，led2 灯 3s 后的报警点亮。

14.2.2 RTC 组件库的使用

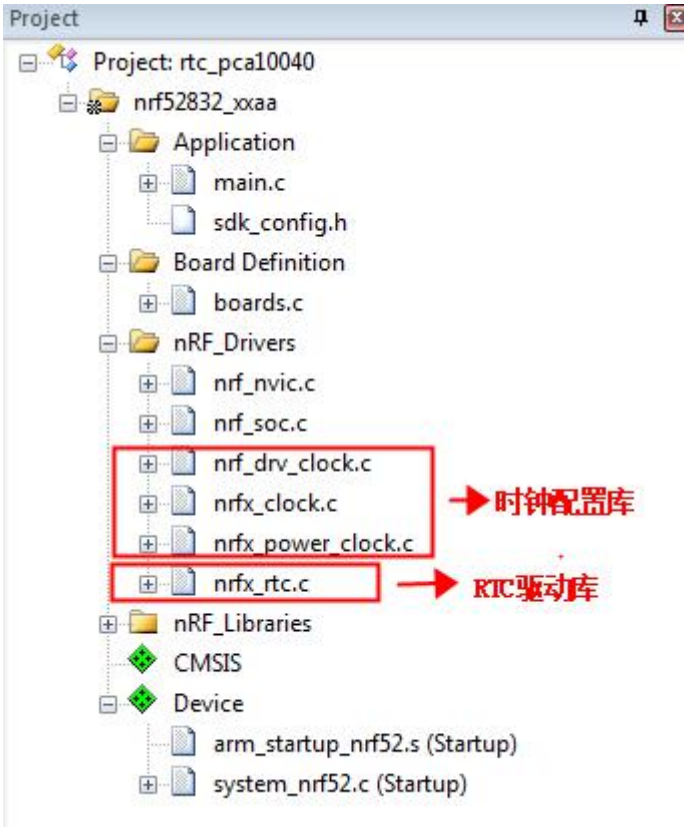
14.2.2.1 库函数工程的搭建

官方 SDK 中，提供了关于 RTC 的驱动组件库文件，下面我们来演示如果采用 RTC 组件库进行工程的编写。首先建立工程，在工程中加入两类关键的库文件：

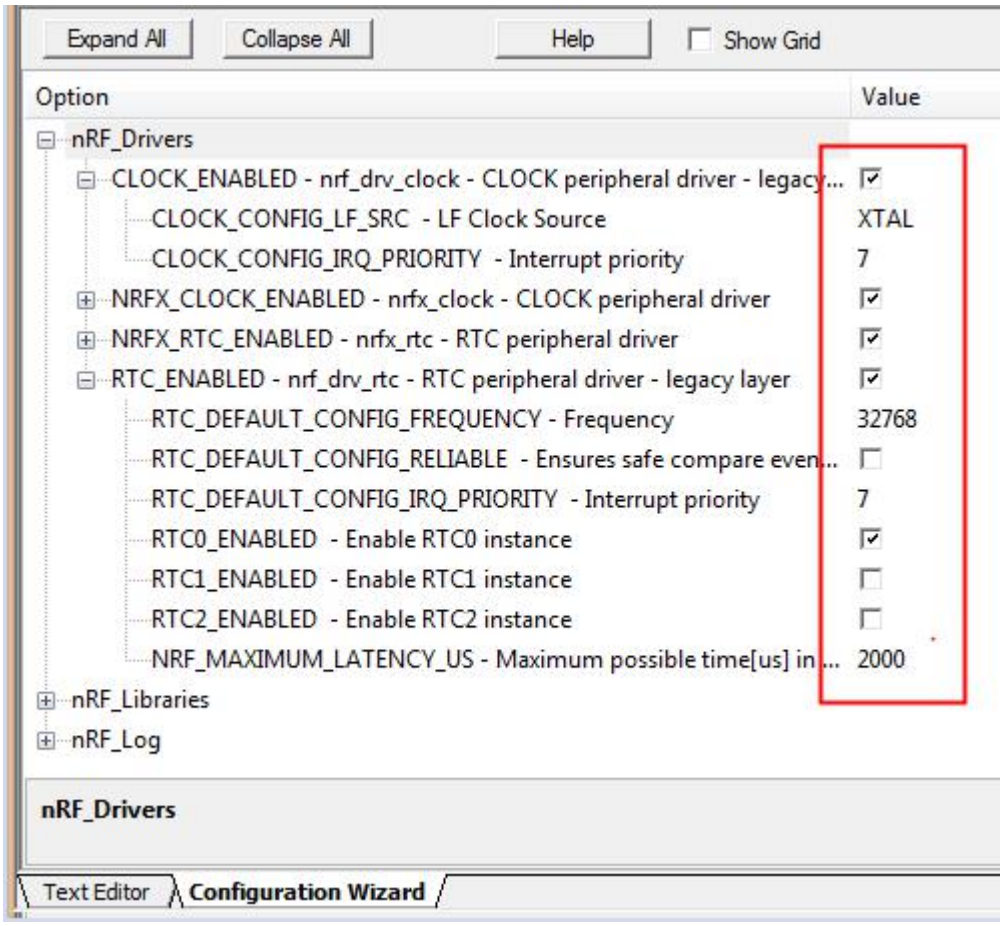
一类为：nrf_drv_clock.c 、nrfx_clock.c 和 nrfx_power_clock.c 这三个驱动文件，为时钟的驱动库。nrf 开头的为旧版本时钟驱动，nrfx 开头的为新版本时钟驱动库，官方为了兼容代码，SDK15 之后的库函数编程都必须进行调用。

另一类为：nrf_drv_rtc.c，就是 RTC 实时计数器的的驱动库。

我们只需要在 main.c 主函数中编写自己的应用，工程目录树如下图所示：



同时需要在 sdk_config.h 文件中配置时钟和 RTC 的使能，可以把配置关于时钟和 RTC 的配置代码直接复制到 sdk_config.h 的 text editor 文本中。如果复制成功，打开 Configuration wizard 配置导航卡，会观察到如下图所示的选项被勾选：



14.2.2.2 库函数 API 介绍

在编写应用前，先来介绍几个比较重要的 rtc 驱动组件库函数，弄清楚函数的意义和用法：

●#define nrf_drv_rtc_init nrfx_rtc_init：RTC 初始化函数

函数：nrfx_err_t nrfx_rtc_init(nrfx_rtc_t const * const p_instance, nrfx_rtc_config_t const * p_config, nrfx_rtc_handler_t handler);
*功能：初始化 RTC 驱动。初始化后，实例处于电源关闭状态
* 参数： p_instance 指向本实例的指针 * 参数： p_config 指向初始化配置的指针 * 参数： handler 用户的事件处理程序
* 返回值 NRF_SUCCESS 如果初始化成功 * 返回值 NRF_ERROR_INVALID_STATE 如果实例已经初始化了.

这个函数中配置 RTC 的参数 p_config 采用结构体形式，结构体中定义了 RTC 的几个关键参数：分频值、中断优先级、滴答下中断处理程序的最大时间长度、标记，配置这个是初始化 RTC 模块的关键：

```
/*RTC 驱动实例配置结构体*/
typedef struct
{
```

```
    uint16_t    prescaler;              /*分频值*/
    uint8_t     interrupt_priority;     /*中断优先级*/
    uint8_t     tick_latency;           /*滴答下中断处理程序的最大长度 (最大 7.7 ms). */
    bool        reliable;               /* 标记. */
} nrfx_rtc_config_t;
```

●#define nrf_drv_rtc_tick_enable nrfx_rtc_tick_enable：RTC 使能滴答函数：

函数：void nrfx_rtc_tick_enable(nrfx_rtc_t const * const p_instance, bool enable_irq);
*功能：使能 RTC 的 tick 功能.
* 参数： p_instance 指向本实例的指针 * 参数： enable_irq 真（True）的就使能中断，假（False）的就关闭中断
* 返回值 无

●#define nrf_drv_rtc_enable nrfx_rtc_enable：RTC 使能驱动函数：

函数：void nrfx_rtc_enable(nrfx_rtc_t const * const p_instance);
*功能：使能 RTC 驱动实例
*参数：p_instance 指向实例的指针.
*返回值 无

●#define nrf_drv_rtc_cc_set nrfx_rtc_cc_set：RTC 比较事件设置

函数：nrfx_err_t nrfx_rtc_cc_set(nrfx_rtc_t const * const p_instance, uint32_t channel, uint32_t val, bool enable_irq);
*功能：选择 RTC 比较频道
*参数： p_instance 指向实例的指针. *参数： channel 实例使用的频道 *参数： val 比较寄存器里设置的值 *参数： enable_irq 真（True）的就使能中断，假（False）的就关闭中断
*返回值 NRF_SUCCESS 如果设置成功 *返回值 NRF_ERROR_TIMEOUT 如果超时错误

RTC 比较事件函数中如果 RTC 模块未初始化或通道参数错误，该函数将发回错误超时。

如果 RTC 模块处于关机状态，则该函数将实现 RTC 模块的启动。在配置 RTC 时，驱动程序没有进入临界区，这意味着在一定时间内它可以被抢占。当驱动程序被抢占，同时设置的值对应的时间很短，这时会出现这样的风险：驱动程序设置的比较值要小于当前的计数值。为了避免这种风险的发生，提供一种可靠模式。通过在给定 RTC 模块的宏定义中设置：rtcn_config_reliability 为 1，来实现可靠模式。可靠模式就是通过设置的比较值是否要小于当前的计数值，如果要小于则产生 NRF_ERROR_TIMEOUT 错误超时，可靠模式要做出以下要求：

-以节拍(8 位值)为单位的最大抢占时间要小于 7.7 ms (对于 precaler = 0, RTC 频率为 32 kHz)。

-请求的绝对比较值不大于(0x00FFFFFF) 。确保这一点是用户的责任。

14.2.2.3 程序代码的编写

介绍完基本函数，下面开始编写程序。

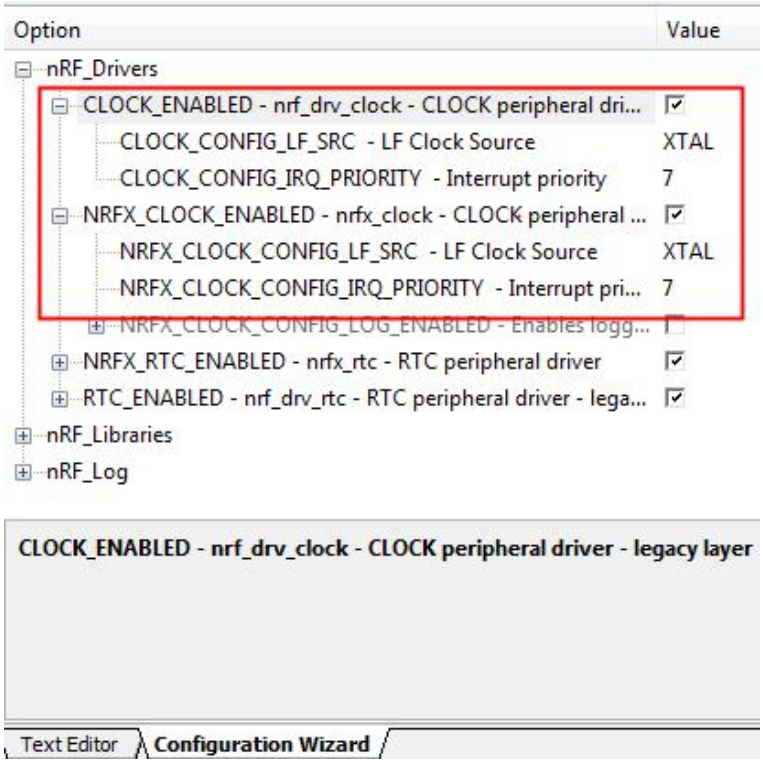
1：首先要求配置时钟，RTC 选取的时钟为低速时钟，调用函数 nrf_drv_clock_init 进行时钟配，这个函数内部调用了 nrfx_clock_enable()时钟使能函数。

nrfx_clock_enable()时钟使能函数初始化话了两个参数：一个时钟选选择，一个中断优先级设置；

在 sdk_config.h 的 text editor 文本中，有如下的代码设置：

```
// <e> NRFX_CLOCK_ENABLED - nrfx_clock - CLOCK peripheral driver
//==========================================================
#ifndef NRFX_CLOCK_ENABLED
#define NRFX_CLOCK_ENABLED 1
#endif
// <o> NRFX_CLOCK_CONFIG_LF_SRC  -设置低速时钟源来源

// <0=> RC
// <1=> XTAL
// <2=> Synth

#ifndef NRFX_CLOCK_CONFIG_LF_SRC
#define NRFX_CLOCK_CONFIG_LF_SRC 1
#endif

// <o> NRFX_CLOCK_CONFIG_IRQ_PRIORITY  - 设置中断优先级：
// <0=> 0 (highest)
// <1=> 1
// <2=> 2
// <3=> 3
// <4=> 4
// <5=> 5
// <6=> 6
// <7=> 7

#ifndef NRFX_CLOCK_CONFIG_IRQ_PRIORITY
#define NRFX_CLOCK_CONFIG_IRQ_PRIORITY 7
#endif
```

打开 Configuration wizard 配置导航卡，会观察到如下图所示的选项被勾选：



低速时钟配置完成后，发出低速时钟请求，具体代码如下：

```
01. static void lfclk_config(void)//低速时钟配置
02. {
03.     ret_code_t err_code = nrf_drv_clock_init();//时钟驱动初始化
04.     APP_ERROR_CHECK(err_code);
05.     nrf_drv_clock_lfclk_request(NULL);//请求低速时钟
06. }
```

2：时钟配置好后，再来设置 RTC，这个函数的主要工作就是初始化 RTC，同时设置 RTC 中断，最后对 RTC 进行使能。本例主要要实例两个中断，一个是 RTC 常用的滴答 TICK 中断，一个就是 RTC 比较中断。

```
01. static void rtc_config(void)
02. {
03.     uint32_t err_code;
04.
05.     //初始化 RTC 实例
06.     nrf_drv_rtc_config_t config = NRF_DRV_RTC_DEFAULT_CONFIG;
07.     config.prescaler = 4095;
08.     err_code = nrf_drv_rtc_init(&rtc, &config, rtc_handler);
09.     APP_ERROR_CHECK(err_code);
10.
11.     //启用滴答事件和中断
12.     nrf_drv_rtc_tick_enable(&rtc,true);
13.
14.     //将比较通道设置为在 COMPARE_COUNTERTIME 秒后触发中断
```

```
15.     err_code = nrf_drv_rtc_cc_set(&rtc,0,COMPARE_COUNTERTIME * 8,true);
16.     APP_ERROR_CHECK(err_code);
17.
18.     //启动 RTC 实例
19.     nrf_drv_rtc_enable(&rtc);
20. }
```

第 6 行，对 RTC 初始化，在程序开头先通过一个申明使用的&rtc：const nrf_drv_rtc_t rtc = NRF_DRV_RTC_INSTANCE，那么初始化 nrf_drv_rtc_init 就是配置这个 RTC，使用默认配置。RTC 中断回调操作为 rtc_handler。那么库组件中给出的默认配置如下：

```
#define NRFX_RTC_DEFAULT_CONFIG
{     //计数频率
    .prescaler= RTC_FREQ_TO_PRESCALER(NRFX_RTC_DEFAULT_CONFIG_FREQUENCY), \
     //中断优先级
    .interrupt_priority = NRFX_RTC_DEFAULT_CONFIG_IRQ_PRIORITY,     \
    //标记
    .reliable= NRFX_RTC_DEFAULT_CONFIG_RELIABLE,                \
    //滴答下中断处理程序的最大长度
    .tick_latency= NRFX_RTC_US_TO_TICKS(NRFX_RTC_MAXIMUM_LATENCY_US,
                                        NRFX_RTC_DEFAULT_CONFIG_FREQUENCY),
}
```

其中计数频率是根据 RTC 的计数频率的公式：$PRESCALER = (32.768kHZ / f_{RTC}) - 1$，在 nrf_rtc.h 文件中设置宏定义：

```
#define   RTC_FREQ_TO_PRESCALER(FREQ)   (uint16_t)(((RTC_INPUT_FREQ)/(FREQ))-1)
```

RTC_INPUT_FREQ 为 32768，通过设置配置文件 sdk_config.h 中的 FREQ 的大小来计算分配值，例如默认设置为 32768，则计数频率值为 1Hz。重新配置为 4095。则计数频率值为 8Hz。

interrupt_priority 为设置的 RTC 的 IRQ 中断的优先级。

Reliable 参数为设置的是否使能比较事件的可靠模式。

.tick_latency 参数为中断处理程序的最大时间长度(以滴答为单位)。

第 8 行：调用 nrfx_rtc_init 函数初始化 RTC 计数器。第二个形参&config 指向前面定义的 RTC 配置参数，第三个形参定义一个中断回调处理事件 rtc_handler。

而第一个形参&rtc，指向我们自定义的 RTC 模块，因为 RTC 有 3 个模块：RTC0、RTC1 和 RTC2。比如例子中定义：

```
21. const nrf_drv_rtc_t rtc = NRF_DRV_RTC_INSTANCE(0);
22.
23. #define NRFX_RTC_INSTANCE(id)                                   \
24. {                                                               \
25.     .p_reg              = NRFX_CONCAT_2(NRF_RTC, id),           \
26.     .irq                = NRFX_CONCAT_3(RTC, id, _IRQn),        \
27.     .instance_id        = NRFX_CONCAT_3(NRFX_RTC, id, _INST_IDX), \
28.     .cc_channel_count = NRF_RTC_CC_CHANNEL_COUNT(id),           \
29. }
```

NRFX_CONCAT_n 函数的功能就是把其包含的 n 个参数连接在一起定义，所有上面的定义可以看成如下：

```
30. #define NRFX_RTC_INSTANCE(0)          \
31. {
32.     .p_reg    = NRF_RTC0,                   \\使用的 RTC0 模块，指向其基础地址
33.     .irq      = RTC0 _IRQn,                  \\使用的 RTC0 模块的中断优先级
34.     .instance_id= NRFX_RTC0_INST_IDX \\使用的 RTC0 模块索引标志
35.     .cc_channel_count = NRF_RTC_CC_CHANNEL_COUNT(0),   \\使用的 RTC0 模块的 CC 通道
36.                                                                  \\数
37. }
```

第 12 行：启动 RTC 滴答 tick，并且使能滴答 tick 中断，那么在 RTC 的计数频率下，计数一次就发生一次中断，因此 125ms 触发一次中断。

第 15 行：设置 RTC 比较中断，当 RTC 的计数值达到了比较寄存器内的值的时候触发 RTC 比较中断，其中比较值设置为：COMPARE_COUNTERTIME*8=3*8，相当于计数 24 次触发一次比较中断，时间为 125ms*24=3s。

第 19 行：使能 RTC，开 RTC 电源，RTC 开始运行。

3. 配置好 RTC 时钟后，需要编写 rtc_handler 中断回调操作函数，功能是执行 RTC0 中断。中断中触发 TICK 滴答中断和 COMPARE0 匹配中断，代码如下：

```
01. static void rtc_handler(nrf_drv_rtc_int_type_t int_type)//rtc 中断配置
02. {
03.     if (int_type == NRF_DRV_RTC_INT_COMPARE0)//如果中断类型为比较中断
04.     {
05.         nrf_gpio_pin_toggle(COMPARE_EVENT_OUTPUT);//翻转比较事件的管脚
06.     }
07.     else if (int_type == NRF_DRV_RTC_INT_TICK)//如果中断类型为滴答中断
08.     {
09.         nrf_gpio_pin_toggle(TICK_EVENT_OUTPUT);//翻转滴答事件的管脚
10.     }
11. }
```

中断处理函数第 3 行：判断触发中断类型是不是比较类型，如果是比较类型，则翻转 LED2。第 7 行：判断触发中断类型是不是滴答中断，如果是滴答类型，则翻转 LED1。最后主函数就十分简单了，配置好 LED 灯，系统时钟，RTC 后，循环等待，等待中断触发执行：

```
01. int main(void)
02. {
03.     leds_config();//led 配置
04.     lfclk_config();//时钟配置
05.     rtc_config();//RTC 配置
06.
07.     while (true)
08.     {
09.
10.     }
11. }
```

实验下载到青云 nRF52832 开发板后的实验现象如下，led1 灯 125ms 翻转，led2 灯 3s 后的报警点亮。

14.2.3 RTC 溢出事件

RTC 计数器还提供一个溢出事件。在模块输入端通过 TRIGOVRFLW 溢出任务将 COUNTER 计数器的值设置为 0xFFFFF0。因此计数器计数 16 次， COUNTER 计数到 0xFFFFFF，从 0xFFFFFF 溢出到 0 时就发生 OVRFLW 溢出中断事件。默认情况下禁用 OVRFLW 事件，因此需要使用溢出中断的时候需要使能 OVRFLW 事件。下面我们以寄存器和组件库两种方式对比进行学习。

14.2.3.1 溢出事件寄存器应用

溢出事件需要使用的寄存器列表：

寄存器名称	地址偏移	功能描述
TASKS_START	0x000	启动 RTC COUNTER
TASKS_TRIGOVRFLW	0x00C	将 COUNTER 设置为 0xFFFFF0
INTENSET	0x304	启用中断
EVTENSET	0x344	使能事件通道
PRESCALER 预分频器	0x508	COUNTER 频率的 12 位预分频器（32768 /（PRESCALER + 1））。

RTC 溢出代码的基本思路就是，首先需要配置 TRIGOVRFLW 溢出任务，每触发一次溢出事件会开始一次 0xFFFFF0 到 0xFFFFFF 的计数。工程目录以 RTC 滴答和比较事件的寄存器工程为模板。下面来分析编写代码：

```
/**
    功能，配置 RTC 溢出事件
 */
 void rtc_config(void)
{
    NRF_RTC0->PRESCALER= COUNTER_PRESCALER;        // 设置预分频值
    NRF_RTC0->TASKS_TRIGOVRFLW=1;                  //触发溢出任务

    // 使能溢出通道和溢出中断
    NRF_RTC0->EVTENSET= RTC_EVTENSET_OVRFLW_Msk;
    NRF_RTC0->INTENSET= RTC_INTENSET_OVRFLW_Msk;
    NVIC_EnableIRQ(RTC0_IRQn);                     // 使能 RTC 中断.
}
```

上面代码中编写 RTC 溢出事件初始化的配置函数 rtc_config()，首先设置 RTC 的计数频率，通过设置分频器来进行确认，计数频率和分频值的公式前面一节已经讲述，这里不再累述。然后设置 TASKS_TRIGOVRFLW 寄存器为 1，触发溢出计数一次。最后，使能溢出中断和溢出事件通道，同时使能 RTC 的 IRQ 中断嵌套。

```
/******************** (C) COPYRIGHT 2019 青风电子 ********************
 * 文件名  ：main
 * 描述    ：
```

```
17.  * 实验平台：青风 nrf52XX 开发板
18.  * 描述      ：RTC TICK 溢出事件中断
19.  * 作者      ：青风
20.  * 店铺      ：qfv5.taobao.com
21. ********************************************************************/
22. #include "nrf52.h"
23. #include   "led.h"
24. #include   "rtc.h"
25.
26. int main(void)
27. {
28.       LED_Init();
29.       lfclk_config();
30.       rtc_config();
31.       NRF_RTC0->TASKS_START = 1;
32.
33.       while (1)
34.       {
35.            // Do nothing.
36.       }
37. }
```

主函数中，调用 RTC 溢出事件的初始化函数 rtc_config()，同时通过置位寄存器 TASKS_START 来开启 RTC 时钟计数器。最后通过一个 while 循环函数进行等待，等待计数器溢出中断的发生。当 EVENTS_OVRFLW 溢出事件发生后，进入 RTC 中断函数，在函数中，首先清除溢出标志，同时翻转 LED1 灯。如果再次溢出，则需要在中断中再开启触发一次 TRIGOVRFLW 溢出。具体代码如下所示：

```
38. void RTC0_IRQHandler()
39. {
40.       if ((NRF_RTC0->EVENTS_OVRFLW!= 0) &&
41.            ((NRF_RTC0->INTENSET & RTC_INTENSET_OVRFLW_Msk) != 0))
42.       {
43.            NRF_RTC0->EVENTS_OVRFLW = 0;//清除事件
44.             NRF_RTC0->TASKS_TRIGOVRFLW=1;//重新触发溢出
45.            LED1_Toggle();//led 灯翻转
46.       }
47. }
```

因为本来定义的 RTC 计数器的频率在 8Hz，因此计数 16 次大概需要 2s 的时间，因此实验下载到青云 nRF52832 开发板后的实验现象如下，led1 灯 2s 翻转一次。

14.2.3.2 组件库下溢出事件的应用

RTC 溢出计数的组件库的工程目录和前面滴答和比较库函数的工程树一致，可以直接使用前面搭建的工程。库函数下编程需要使用库函数取代寄存器来实现 RTC 的溢出事件和溢出中断的使能，并且触发溢出计数。因此需要实现下面两个函数 api：

●#define nrf_drv_rtc_overflow_enable nrfx_rtc_overflow_enable 函数：使能溢出事件和溢出中断

函数：void nrfx_rtc_overflow_enable(nrfx_rtc_t const * const p_instance, bool enable_irq);
*功能：函数启用溢出。此函数启用溢出事件和中断(可选)。如果模块没有通电，该函数将无效。
* 参数： p_instance 指向本实例的指针 * 参数： enable_irq true 启用中断。False 禁用中断
* 返回值 无

●__STATIC_INLINE void nrf_rtc_task_trigger 函数：使能溢出事件和溢出中断

函数：__STATIC_INLINE void nrf_rtc_task_trigger(NRF_RTC_Type * p_rtc, nrf_rtc_task_t task);
*功能：开始溢出滴答计数
* 参数：p_rtc 指向外围寄存器结构的指针 * 参数：task 要求的任务
* 返回值 无

首先编写 RTC 溢出配置函数。初始化配置 RTC 计数频率为 8Hz，因此计数 16 次需要 2s 时间。然后调用函数 nrf_drv_rtc_overflow_enable 来使能溢出事件和溢出中断，把函数的 enable_irq 设置为 true。再使能 RTC 指定的模块 RTC0。使能完成后，触发溢出计数。具体代码如下：

```
01. static void rtc_config(void)
02. {
03.     uint32_t err_code;
04.
05.     //初始化 RTC 模块
06.     nrf_drv_rtc_config_t config = NRF_DRV_RTC_DEFAULT_CONFIG;
07.     config.prescaler = 4095;
08.     err_code = nrf_drv_rtc_init(&rtc, &config, rtc_handler);
09.     APP_ERROR_CHECK(err_code);
10.
11.     //使能溢出事件和溢出中断
12.     nrf_drv_rtc_overflow_enable(&rtc,true);
13.     APP_ERROR_CHECK(err_code);
14.
15.     //使能 RTC
16.    nrf_drv_rtc_enable(&rtc);
17.    //触发溢出计数
18.   nrf_rtc_task_trigger(rtc.p_reg,NRF_RTC_TASK_TRIGGER_OVERFLOW);
19. }
```

主函数中通过一个 while 循环函数进行等待，等待计数器溢出中断的发生。当 EVENTS_OVRFLW 溢出事件发生后，进入 RTC 中断函数。触发 RTC 事件回调 rtc_handler。在回调函数中，翻转 LED1 灯。如果再次溢出，则需要在回调中再次开启触发 TRIGOVRFLW 溢出。具体代码如下所示：

```
20. static void rtc_handler(nrf_drv_rtc_int_type_t int_type)
21. {
22.     if (int_type == NRF_DRV_RTC_INT_OVERFLOW)
```

```
23.         {
24.             nrf_gpio_pin_toggle(BSP_LED_0);
25.               nrf_rtc_task_trigger(rtc.p_reg,NRF_RTC_TASK_TRIGGER_OVERFLOW);
26.         }
27.
28. }
```

编译工程后，实验下载到青云 nRF52832 开发板后的实验现象如下，led1 灯 2s 翻转一次。