# 第十一章 定时器 TIME

## 11.1 原理分析

和其他 MCU 处理器一样，在 nRF52832 中定时器的功能是十分强大的。其内部包含了 5 个定时器 TIMER 模块:TIMER0、TIMER1、TIMER2、TIMER3、TIMER4，如下表所示。

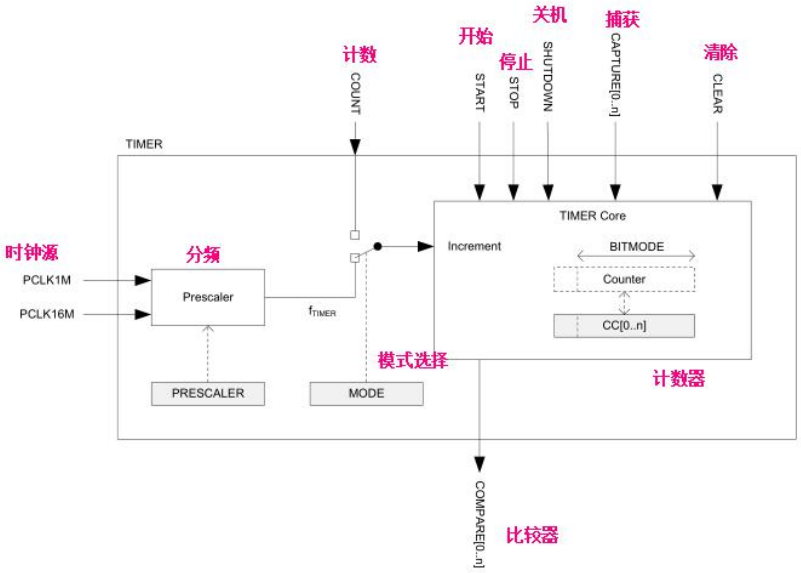
| 地址偏移 | 寄存器名称 | 功能描述 |
|---|---|---|
| 0x40008000 | TIMER0 定时器 0 | 该定时器实例有 4 个 CC 寄存器（CC[0..3]） |
| 0x40009000 | TIMER1 定时器 1 | 该定时器实例有 4 个 CC 寄存器（CC [0..3]） |
| 0x4000A000 | TIMER2 定时器 2 | 该定时器实例有 4 个 CC 寄存器（CC[0..3]） |
| 0x4001A000 | TIMER3 定时器 3 | 该定时器实例有 6 个 CC 寄存器（CC [0..5]） |
| 0x4001B000 | TIMER4 定时器 4 | 该定时器实例有 6 个 CC 寄存器（CC[0..5]） |

定时器有着不同的位宽选择，位宽的大小直接决定了计数器的最大溢出时间。处理器可以通过 BITMODE 选择不同的位宽，该寄存器位于计数器内部。如下表所示可选位宽为 8、16、24 和 32 位：

●BITMODE 寄存器：配置计时器使用的位宽数:

| 位数 | Field | Value ID | Value | 描述 |
|---|---|---|---|---|
| 第 0~1 位 | BITMODE | 16bit | 0 | 16bit 宽度定时器 |
| | | 8bit | 1 | 8bit 宽度定时器 |
| | | 24bit | 2 | 24bit 宽度定时器 |
| | | 32bit | 3 | 32bit 宽度定时器 |

下图为定时器内部结构图，下面就来详细分析下其基本工作原理以及相关概念：




1. 时钟源

首先 TIMER 工作在高频时钟源（HFLCK）下，同时包含了一个 4bit（1 / 2X）的分频

（PRESCALER)，可以对高频时钟源（HFLCK）进行分频。框图入口处给了两个时钟源表示两种时钟输入模式：1MHz 模式（PCLK1M)和 16MHz 模式（PCLK16M)。时钟源通过分频器分频后输出一个频率 $f_{TIMER}$，系统将会通过这个参数来自动选择时钟源，而不需要工程师设置寄存器。

当 $f_{TIMER} > 1MHZ$ 时，系统自动选择 PCLK16M 作为时钟源。

当 $f_{TIMER} <= 1MHZ$ 时，系统会自动用 PCLK1M 替代 PCLK16M 作为时钟源以减少功耗。

2.分频器

分频器对输入的时钟源进行分频。输出的频率计算公式如下：

$$f_{TIMER} = \frac{HFCLK}{2^{PRESCALER}}$$

公式中的 HFLCK 不管是使用哪种时钟源输入，计算分频值的时候都使用 16MHz。PRESCALER 为一个 4bit 的分频器，分频值为 0~15。当 PRESCALER 的值大约 9 后，其计算值仍然为 9，即 $f_{TIMER}$ 的最小值为 $16/2^9$。通过设置寄存器 PRESCALER 可以控制定时器的频率：

●PRESCALER 寄存器：预分频寄存器

| 位数 | Field | Value ID | Value | 描述 |
|---|---|---|---|---|
| 第 0~3 位 | PRESCALER | | [0..9] | 预分频的值 |

3. 工作模式

定时器 TIMER 可以工作在两种模式下：定时器模式（timer）和计数器模式（counter）。工作模式通过寄存器 MODE 进行选择。当 MODE 设置为 0 的时候为定时器模式，设置为 1 的时候为计数器模式，设置为 2 时，选择低功耗的计算器模式。

●MODE 寄存器：定时器模式设置寄存器

| 位数 | Field | Value ID | Value | 描述 |
|---|---|---|---|---|
| 第 0~1 位 | MODE | Timer | 0 | 选择定时器模式 |
| | | Counter | 1 | 选择计数器模式 |
| | | LowPowerCounter | 2 | 选择低功耗计数器模式 |

定时器模式下为递增计数（incrememt)，每一个时钟频率下，计数器自动加一。

计数模式下，每触发一次寄存器 COUNT event，定时器内部计数器寄存器就会加一。

4.比较/捕获功能

定时模式下设定比较（Compare）/捕获（Capture）寄存器 CC[n]的值，可以设置定时的时间（Timer value）。

当定时时间的值跟 CC[n]寄存器的值相等时，将触发一个 COMPARE [n] event。COMPARE [n] event 可以触发中断，如果是周期性的触发，则需要在触发后清除计数值，否则会一直计数，直到溢出。

计数模式下，每次触发 COUNT 任务时，TIMER 的内部计数器寄存器都会递增 1，计数器模式下不使用定时器的频率和预分频器。同样 COUNT 任务在定时器模式下无效。通过设定一个 CAPTURE Task，捕获的计数器的值存储到 CC[n]寄存器内，然后对 CC[n]寄存器进行读取计数的值。

5.任务延迟和优先级

任务延迟：TIMER 启动后，CLEAR 任务，COUNT 任务和 STOP 任务将保证在 PCLK16M 的一个时钟周期内生效。

任务优先级：如果同时触发 START 任务和 STOP 任务，即在 PCLK16M 的同一时段内，则优先执行 STOP 任务。

## 11.2 定时器定时功能

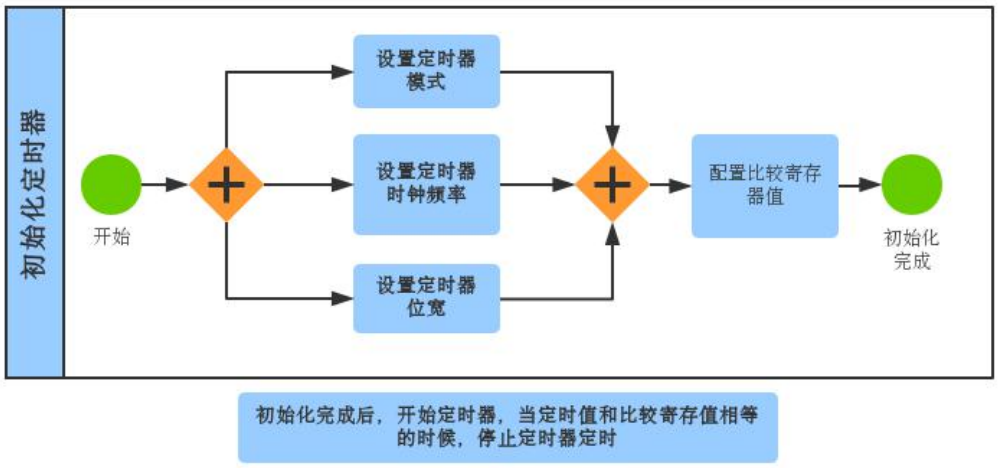
### 11.2.1 寄存器编写

在代码文件中，实建立了一个演示历程，我们打开看看需要编写那些文件。打开 pca10040 文件夹中的 time 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 在上一节控制 LED 灯的时候已经写好，现在我们就来讨论下如何编写 time.c 这个驱动子文件和主函数 main.c 文件。

time.c 文件主要是要起到两个作用：第一：初始化定时器参数。第二：设置定时时间函数。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定时器的设置步骤，可以如下图所示：



定时器首先需要设置的三个参数，分别为：定时器的模式、定时器的位宽、定时器的时钟频率。本例中需要进行定时操作，因此还需要设置比较寄存器里的值。如果初始化完成后，定时器就开始定时，当定时时间的值跟 CC[n]寄存器的值相等时，将触发一个 COMPARE [n] event。这时候我们关掉定时器定时。这样根据 CC[n]寄存器的值就是实现了一个指定的时间长度的。下表是定时器的寄存器列表，详细说明如下表示所示：

| 寄存器名称 | 地址偏移 | 功能描述 |
|---|---|---|
| TASKS_START | 0x000 | 启动定时/计数器 |
| TASKS_STOP | 0x004 | 停止定时/计数器 |
| TASKS_COUNT | 0x008 | 计数器递增（仅限计数器模式） |
| TASKS_CLEAR | 0x00C | 清除计数器 |
| TASKS_SHUTDOWN | 0x010 | 关闭定时/计数器 |
| TASKS_CAPTURE[0] | 0x040 | 将定时器值捕获到 CC [0]寄存器 |
| TASKS_CAPTURE[1] | 0x044 | 将定时器值捕获到 CC [1]寄存器 |
| TASKS_CAPTURE[2] | 0x048 | 将定时器值捕获到 CC [2]寄存器 |
| TASKS_CAPTURE[3] | 0x04C | 将定时器值捕获到 CC [3]寄存器 |
| TASKS_CAPTURE[4] | 0x050 | 将定时器值捕获到 CC [4]寄存器 |
| TASKS_CAPTURE[5] | 0x054 | 将定时器值捕获到 CC [5]寄存器 |
| EVENTS_COMPARE[0] | 0x140 | 比较 CC [0]匹配的事件 |
| EVENTS_COMPARE[1] | 0x144 | 比较 CC [1]匹配事件 |
| EVENTS_COMPARE[2] | 0x148 | 比较 CC [2]匹配的事件 |
| EVENTS_COMPARE[3] | 0x14C | 比较 CC [3]匹配的事件 |
| EVENTS_COMPARE[4] | 0x150 | 比较 CC [4]匹配的事件 |
| EVENTS_COMPARE[5] | 0x154 | 比较 CC [5]匹配的事件 |
| SHORTS | 0x200 | 快捷方式注册 |
| INTENSET | 0x304 | 启用中断 |
| INTENCLR | 0x308 | 禁用中断 |
| MODE | 0x504 | 定时器模式选择 |
| BITMODE | 0x508 | 配置 TIMER 使用的位数 |
| PRESCALER 预分频器 | 0x510 | 定时器预分频器寄存器 |
| CC[n] n=0~5 | 0x540~0x554 | 比较寄存器[n],n=0~5 |

根据前面的分析过程，这时我们配置一个定时器代码设置如下所示：

```
01.     p_timer->MODE = TIMER_MODE_MODE_Timer;          // 设置为定时器模式
02.     p_timer->PRESCALER = 9;                         //9 分频
03.     p_timer->BITMODE = TIMER_BITMODE_BITMODE_16Bit;  // 16 bit 模式.
04.     p_timer->TASKS_CLEAR     = 1;                   // 清定时器.
05.     // 分频后的时钟*31.25 后为 1ms
06.     p_timer->CC[0] = number_of_ms * 31;
07.     p_timer->CC[0] += number_of_ms / 4;             //设置比较寄存器的值
08.     p_timer->TASKS_START     = 1;                   // 开始定时器
09.       while (p_timer->EVENTS_COMPARE[0] == 0)       //触发后会把比较事件置为 1
```

```
10.         {
11.         }
12.         p_timer->EVENTS_COMPARE[0]    = 0;        //清 0 比较事件寄存器
13.         p_timer->TASKS_STOP                = 1;                    // 停止定时
```

上面一段代码的编写严格按照了寄存器要求进行：

第 01 行：是 MODE 设置，也就是模式设置，我们设置定时器模式。

第 02 行：设置预分频值，PRESCALER 寄存器设置预分频计数器，前面原理里已经讲过，要产生 Ftimer 时钟，必须把外部提供的高速时钟 HFCLK 首先进行分频。

$$f_{TIMER} = \frac{HFCLK}{2^{PRESCALER}}$$

代码里，我们设置为 9 分频，按照分频计算公式，Ftimer 定时器频率时钟为 31250 Hz。

第 03 行：是 BITMODE 寄存器，BITMODE 寄存器就是定时器的位宽。就比如一个水桶，这个水桶的深度是多少，定时器的位宽就是定时器计满需要的次数，也就是最大的计算次数。

例如：我们设置为 16bit，计数的次数最大应该是 2 的 16 次方次。假设定时器频率时钟为 1kHz，定时器 1ms 计数一次。

最大定时时间=最大的计数次数*计数一次的时间=65535*1ms=65535ms

因此位宽与定时器频率共同决定了定时器可以定时的最大时间。

第 04 行：表示定时器开始计数之前需要清空，从 0 开始计算。

第 06~07 行：设定定时比较的值，也就是比较/捕获功能，当定时时间的值跟 CC[n]寄存器的值相等时，将触发一个 COMPARE [n] event。由于分频后的 Ftimer 时钟下定时器 1/31250 s 计数一次，1ms 需要计 31.25 次，那么 CC[n]存放定时时间（单位 ms）*31.25。

第 08 行：设置好了就开始启动定时器。

第 09 行：判断比较事件寄存器是否被置为 1，如果置为 1，表示定时器的值跟 CC[n]寄存器的值相等了，则跳出循环。

第 12 行：把置为 1 的比较事件寄存器清 0。

第 13 行：定时时间到了停止定时器。

把这段代码进行封装，声明为一个 ms 定时的函数：

void nrf_timer_delay_ms(timer_t timer,  uint_fast16_t volatile number_of_ms)，后面主函数中就可以直接调用该函数进行定时了。

定时器初始化时，定时器的时钟由外部的高速时钟提供，我们必须首先初始化进行 HFCLK 时钟开启：

```
01.  // 开始 16 MHz 晶振.
02.      NRF_CLOCK->EVENTS_HFCLKSTARTED    = 0;
03.      NRF_CLOCK->TASKS_HFCLKSTART        = 1;
04.      // 等待外部振荡器启动
05.      while (NRF_CLOCK->EVENTS_HFCLKSTARTED == 0)
06.      {
07.          // Do nothing.
08.      }
```

同时加入定时器选择的功能。

```
01.   switch (timer)
02.      {
```

```
        case TIMER0:
            p_timer = NRF_TIMER0;
            break;

        case TIMER1:
            p_timer = NRF_TIMER1;
            break;

        case TIMER2:
            p_timer = NRF_TIMER2;
            break;

        default:
            p_timer = 0;
            break;
    }
```

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 灯指示定时器定时的时间进行相应的变化。函数如下所示：

```
//******************** (C) COPYRIGHT 2019 青风电子 ********************
* 文件名  ：main
* 描述    ：
* 实验平台：青云 nRF52832 开发板
* 描述    ：定时器定时
* 作者    ：青风
* 店铺    ：qfv5.taobao.com
**********************************************************************/
#include "nrf52.h"
#include  "led.h"
#include  "time.h"

#define TIMER_DELAY_MS          (1000UL)  //定义延迟 1000ms

int main(void)
{
    //
     LED_Init();
    while (1)
    {
        LED1_Toggle();
        //使用定时器 0 产生 1s 定时
        nrf_timer_delay_ms(TIMER0, TIMER_DELAY_MS);

        LED1_Toggle();
        // 使用定时器 1 产生 1s 定时
```

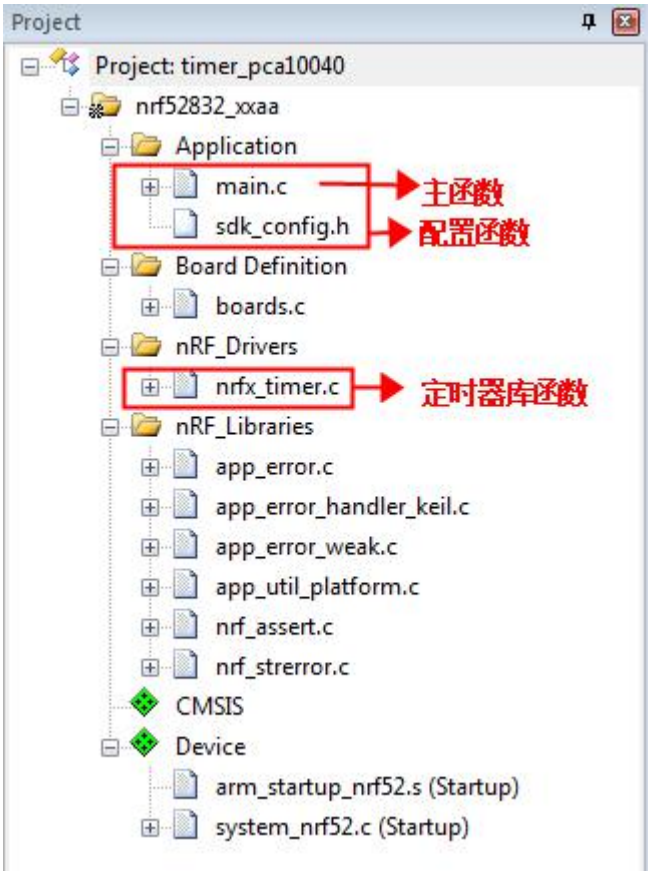
```
27.             nrf_timer_delay_ms(TIMER1, TIMER_DELAY_MS);
28.
29.             LED1_Toggle();
30.             // 使用定时器 2 产生 1s 定时
31.             nrf_timer_delay_ms(TIMER2, TIMER_DELAY_MS);
32.         }
33. }
```

编译代码后，用 keil 把工程下载到青风 QY-nRF52832 蓝牙开发板上。LED 灯会已 1s 的定时时间进行闪烁。

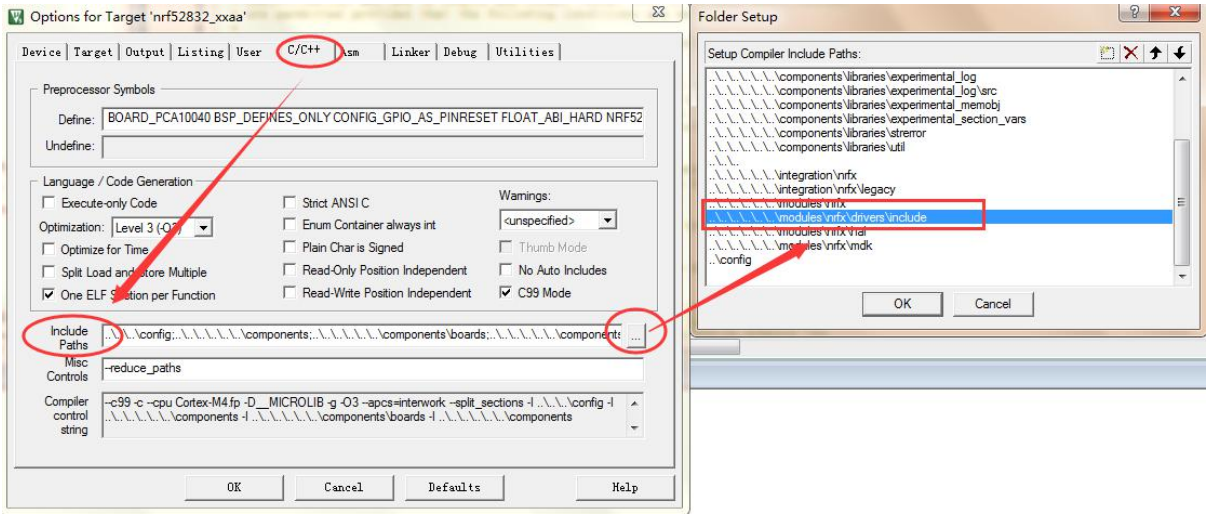
### 11.2.2 定时器组件的应用

为了在以后方便结合协议栈一起编程，学习使用官方定时器库函数的组件进行编程也是需要读者进一步做的工作。寄存器编程相比于组件库函数编程，优点便于理解整个定时器的工作原理，直观的设置寄存器。但是缺点是程序编写功能没有库函数完整。组件库编程的核心就是理解组件库函数的 API，在调用组件库函数时首先需要弄清其函数定义。

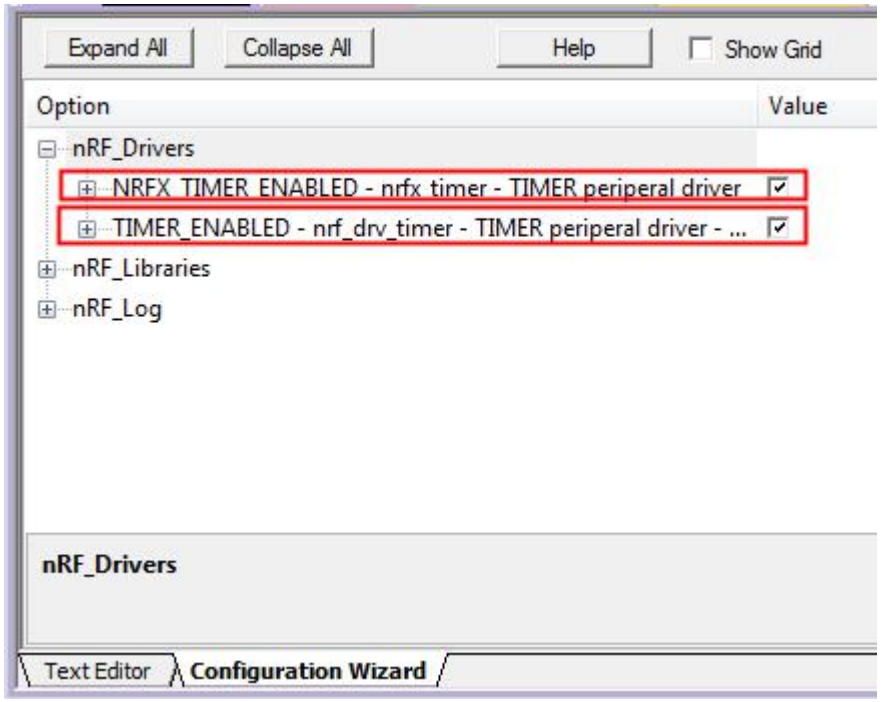
工程的建立可以以前面 GPIOE 的组件库函数工程为模板，这里面我们只需要改动的如下框中的几个地方：




主函数 main.c 文件，sdk_config.h 配置文件这两个文件需要我们编写和修改的。而 nrfx_timer.c 文件则需要我们添加的库文件。nrfx_timer.c 文件的路径在 SDK 的//modules/nrfx/drivers/include 文件夹里，添加库文件完成后，注意在 Options for Target 选项卡的 C/C++中，点击 include paths 路径选项中添加硬件驱动库的文件路径，如下图所示：




工程搭建完毕后，首先我们需要来修改 sdk_config.h 配置文件，库函数的使用是需要对库功能进行使能的，因此需要在 sdk_config.h 配置文件中，设置对应模块的使能选项。关于定时器的配代码选项较多，我们不就一一展开，大家可以直接把对应的配置代码复制到自己建立的工程中的 sdk_config.h 文件里。如果复制代码后，在 sdk_config.h 配置文件的 configuration wizard 配置导航卡中看见如下两个参数选项被勾选，表明配置修改成功：




定时器使用组件库编程的基本原理和寄存器的编程基本一致。下面我们来介绍下需要调用的几个定时器相关设置函数：

1：首先我们需要再主函数前定义使用哪个定时器，使用 NRF_DRV_TIMER_INSTANCE (id)中 ID 来定义，按照前面的原理介绍，这个 ID 值可以为 0,1,2 三个值，如果设置定时器 0 则如下代码：

```
const nrf_drv_timer_t TIMER_LED = NRF_DRV_TIMER_INSTANCE(0);//设置使用的定时器
```

然后配置定时器 0，对定时器初始化，使用库中的 nrf_drv_timer_init 函数。该函数用于定义定时器的相关配置，以及中断回调的函数：

| 函数：ret_code_t nrfx_timer_init(nrf_drv_timer_t const * const p_instance,<br>nrf_drv_timer_config_t const * p_config,<br>nrf_timer_event_handler_t timer_event_handler); |
|---|
| *功能：函数用于初始化计时器。 |
| * 参数 p_instance 前面选择的定时器 0 或者 1、2<br>* 参数 p_config 初始化配置。如果没有配置，默认下则设置为 NULL.<br>* 参数 timer_event_handler 定时器回调中断声明。 |
| * 返回值：NRFX_SUCCESS 驱动程序初始化成功。<br>* 返回值：NRFX_ERROR_INVALID_STATE 驱动程序已经初始化。 |

重点说明下参数 p_config ，这个参数是 nrfx_timer_config_t 结构体类型，这个结构体内给了定时器的相关配置函数：

```
01. typedef struct
02. {
03.     nrf_timer_frequency_t   frequency;          /**<频率. */
04.     nrf_timer_mode_t        mode;               /**< 模式选择 */
05.     nrf_timer_bit_width_t   bit_width;          /**< 位宽 */
06.     uint8_t                 interrupt_priority; /**< 定时器中断优先级*/
07.     void*                   p_context;          /**< 上下文传递参数*/
08. } nrfx_timer_config_t;
```

如果结构体设置为 NULL。此时默认为初始设置。在 nrfx_time.h 文件中定义默认配置的结构体，具有代码如下所示：

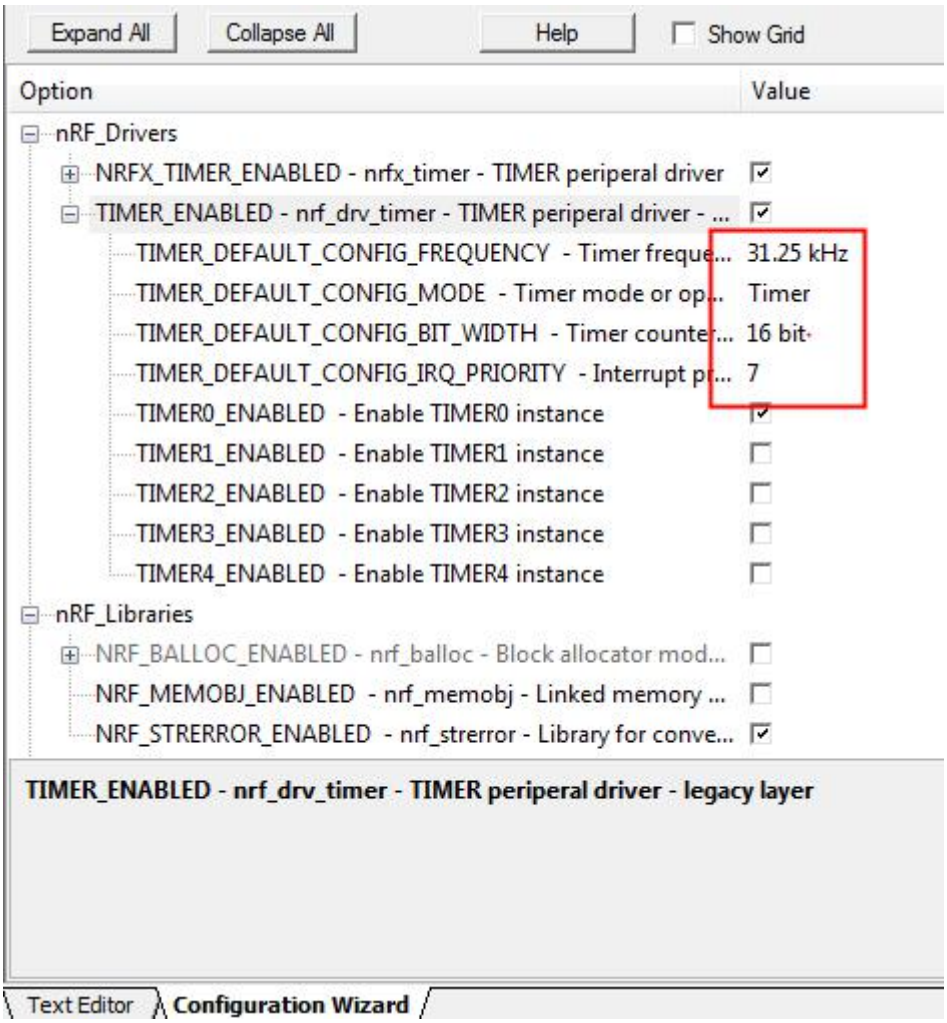
```
09. #define NRFX_TIMER_DEFAULT_CONFIG
10. {
11.     .frequency    = (nrf_timer_frequency_t)NRFX_TIMER_DEFAULT_CONFIG_FREQUENCY,
12.     .mode         = (nrf_timer_mode_t)NRFX_TIMER_DEFAULT_CONFIG_MODE,
13.     .bit_width = (nrf_timer_bit_width_t)NRFX_TIMER_DEFAULT_CONFIG_BIT_WIDTH,
14.     .interrupt_priority = NRFX_TIMER_DEFAULT_CONFIG_IRQ_PRIORITY,
15.     .p_context = NULL
16. }
```

这几个定义参数的值在 sdk_config.h 配置文件中进行了赋值，这个默认配置参数值是可以自由修改，具体代码如下所示：

```
17. // <o> NRFX_TIMER_DEFAULT_CONFIG_FREQUENCY  - Timer frequency if in Timer mode
18.   //定时器的主频
19. // <0=> 16 MHz
20. // <1=> 8 MHz
21. // <2=> 4 MHz
22. // <3=> 2 MHz
23. // <4=> 1 MHz
24. // <5=> 500 kHz
25. // <6=> 250 kHz
26. // <7=> 125 kHz
27. // <8=> 62.5 kHz
28. // <9=> 31.25 kHz
29.
```

```
#ifndef NRFX_TIMER_DEFAULT_CONFIG_FREQUENCY
#define NRFX_TIMER_DEFAULT_CONFIG_FREQUENCY 9
#endif
  //定时器的模式
// <o> NRFX_TIMER_DEFAULT_CONFIG_MODE  - Timer mode or operation

// <0=> Timer
// <1=> Counter

#ifndef NRFX_TIMER_DEFAULT_CONFIG_MODE
#define NRFX_TIMER_DEFAULT_CONFIG_MODE 0
#endif
//定时器的位宽
// <o> NRFX_TIMER_DEFAULT_CONFIG_BIT_WIDTH  - Timer counter bit width

// <0=> 16 bit
// <1=> 8 bit
// <2=> 24 bit
// <3=> 32 bit

#ifndef NRFX_TIMER_DEFAULT_CONFIG_BIT_WIDTH
#define NRFX_TIMER_DEFAULT_CONFIG_BIT_WIDTH 0
#endif
//定时器中断的优先级
// <o> NRFX_TIMER_DEFAULT_CONFIG_IRQ_PRIORITY  - Interrupt priority

// <0=> 0 (highest)
// <1=> 1
// <2=> 2
// <3=> 3
// <4=> 4
// <5=> 5
// <6=> 6
// <7=> 7

#ifndef NRFX_TIMER_DEFAULT_CONFIG_IRQ_PRIORITY
#define NRFX_TIMER_DEFAULT_CONFIG_IRQ_PRIORITY 7
#endif
```

读者可以直接在文本 Text Editor 中直接修改，或者在 configuration wizard 配置导航卡中点击修改，导航卡中修改如下图所示：




默认设置 31.25KHz 时钟，定时器模式，16bit 位宽，中断优先级为低。这个参数可以根据自己的要求进行修改。

2：采用库函数实现定时器定时，基本原理和寄存器方式相同，就是通过定时器的值和 cc[n]寄存器的值进行比较，如果相等，就触发比较事件，则时候就停止定时器或者触发比较事件中断。本例我们采用中断方式来处理定时，那么首先介绍下面两个组件库函数的 API:

函数 nrfx_timer_ms_to_ticks 用于计算指定定时时间下 cc[n]寄存器的值。用于比较事件的触发，该函数可以方便的算出 cc[n]寄存器的值，方便编程。

| 函数： uint32_t nrfx_timer_ms_to_ticks(nrfx_timer_t const * const p_instance,<br>uint32_t time_ms); |
|---|
| *功能：函数用于将时间(以毫秒为单位)转换为计时器滴答。 |
| *参数 p_instance 前面选择的定时器 0 或者 1、2<br>*参数 timer_ms 定时多少 ms |
| * 返回值：定时器滴答的次数 |

函数 nrfx_timer_extended_compare 用于使能定时器比较通道，使能比较中断，设置触发比较寄存器 cc[n]等参数：

| 函数： void nrfx_timer_extended_compare(nrfx_timer_t const * const p_instance,<br>nrf_timer_cc_channel_t cc_channel, |
|---|

| | | |
|---|---|---|
| | uint32_t<br>nrf_timer_short_mask_t<br>bool | cc_value,<br>timer_short_mask,<br>enable_int); |

*功能：函数用于在扩展比较模式下设置计时器通道。

| | |
|---|---|
| * 参数 p_instance | 前面选择的定时器 0 或者 1、2 |
| * 参数 cc_channel | 捕获/比较寄存器通道 |
| * 参数 cc_value | 比较的值 |
| * 参数 timer_short_mask | 停止或者清除比较事件和定时器任务 |
| * 参数 enable | 使能或者关掉比较器中断 |
| * 返回值：无 | |

该函数中的参数都比较好理解，其中 timer_short_mask 停止或者清除比较事件和定时器任务的快捷方式，就是在寄存器 **SHORTS** 中设置的内容，可以对照 SHORTS 寄存器列表进行比较：

```
01.       typedef enum
02. {
03.       NRF_TIMER_SHORT_COMPARE0_STOP_MASK =
04.    TIMER_SHORTS_COMPARE0_STOP_Msk,     //基于比较 0 事件来停止定时器的快捷方式。
05.       ...................
06.       ...................
07.       NRF_TIMER_SHORT_COMPARE3_STOP_MASK =
08. TIMER_SHORTS_COMPARE3_STOP_Msk,     ///基于比较 3 事件来停止定时器的快捷方式。
09.
10. #if defined(TIMER_INTENSET_COMPARE4_Msk) || defined(__NRFX_DOXYGEN__)
11.       //仅适用于定时器 4 和 5
12.       NRF_TIMER_SHORT_COMPARE4_STOP_MASK =
13. TIMER_SHORTS_COMPARE4_STOP_Msk,     ///基于比较 4 事件来停止定时器的快捷方式。
14. #endif
15. #if defined(TIMER_INTENSET_COMPARE5_Msk) || defined(__NRFX_DOXYGEN__)
16.
17.       NRF_TIMER_SHORT_COMPARE5_STOP_MASK =
18. TIMER_SHORTS_COMPARE5_STOP_Msk,     //基于比较 5 事件来停止定时器的快捷方式。
    #endif
19.
20.       NRF_TIMER_SHORT_COMPARE0_CLEAR_MASK =
    TIMER_SHORTS_COMPARE0_CLEAR_Msk, //基于比较 0 事件来清除定时器计数的快捷方
21. 式。
22. ...................
23. ...................
24.       NRF_TIMER_SHORT_COMPARE3_CLEAR_MASK =
25. TIMER_SHORTS_COMPARE3_CLEAR_Msk, ////基于比较 3 事件来清除定时器计数的快捷方
26. 式。
27. #if defined(TIMER_INTENSET_COMPARE4_Msk) || defined(__NRFX_DOXYGEN__)
28.     //仅适用于定时器 4 和 5
29.       NRF_TIMER_SHORT_COMPARE4_CLEAR_MASK =
30. TIMER_SHORTS_COMPARE4_CLEAR_Msk, ////基于比较 4 事件来清除定时器计数的快捷方
```

```
式。
#endif
#if defined(TIMER_INTENSET_COMPARE5_Msk) || defined(__NRFX_DOXYGEN__)
    NRF_TIMER_SHORT_COMPARE5_CLEAR_MASK =
TIMER_SHORTS_COMPARE5_CLEAR_Msk, ////基于比较 5 事件来清除定时器计数的快捷方
式。
#endif
} nrf_timer_short_mask_t;
```

也就是说存在两种快捷方式：

（1）：第一种：停止定时器。当定时器计数的值和比较寄存器 CC[n]的值相等的时候，触发了 COMPARE [n] event 比较事件，可以用这个事件可以快捷的停止定时器。也就是说一旦发生 COMPARE [n] event 比较事件定时器就会被停止运行。

（2）：第二种：清零定时器的计数值。当定时器计数的值和比较寄存器 CC[n]的值相等的时候，触发了 COMPARE [n] event 比较事件，可以用这个事件可以快捷的清零定时器。这样的后果就是导致了定时器从 0 重新开始计数。我们可以用这种方式实现定时器的快捷定时。

实例代码：那么定时器的配置的代码程序如下所示，设置定时器捕获/比较触发设备、通道、滴答时间等参数:

```
uint32_t time_ms = 1000; //定时器比较事件的时间为 1s
    uint32_t time_ticks;
     //初始化定时器为默认配置
     nrf_drv_timer_config_t timer_cfg = NRF_DRV_TIMER_DEFAULT_CONFIG;
     //配置定时器，同时注册定时器的回调函数
    err_code = nrf_drv_timer_init(&TIMER_LED, NULL, timer_led_event_handler);
    APP_ERROR_CHECK(err_code);
    //设置定时器的滴答时间。算出 CC[n]比较寄存器的值
    time_ticks = nrf_drv_timer_ms_to_ticks(&TIMER_LED, time_ms);
    //设置定时器捕获/比较 触发设备、通道、滴答时间
    nrf_drv_timer_extended_compare(
                                    &TIMER_LED, NRF_TIMER_CC_CHANNEL0, time_ticks,
                                    NRF_TIMER_SHORT_COMPARE0_CLEAR_MASK, true);
```

注意：nrf_drv_timer_extended_compare 函数中的 cc_value 的值也可以直接用寄存器例子中计算公式出来的 CC[n]寄存器的值，1s 定时下 CC[n]寄存器的值为 31250，如下写法：

```
nrf_drv_timer_extended_compare( &TIMER_LED, NRF_TIMER_CC_CHANNEL0, 31250,
                                    NRF_TIMER_SHORT_COMPARE0_CLEAR_MASK, true);
```

3：最后来使能定时器 0，打开定时器开始计时。组件库中提供两个库函数 API 来实现打开和关闭定时器。

| 函数： void nrfx_timer_enable(nrfx_timer_t const * const p_instance); |
|---|
| *功能：函数功能用于使能定时器。 |
| *参数 p_instance 前面选择指向的定时器 0 或者 1、2 的指针 |
| * 返回值：无 |

| 函数： void nrfx_timer_disable(nrfx_timer_t const * const p_instance); |
|---|
| *功能：函数功能用于关闭定时器。注意，只有在调用这个函数之后，计时器将在 SYSTEM_ON |

| |
|---|
| 状态有尽可能低的功耗。 |
| *参数 p_instance 前面选择指向的定时器 0 或者 1、2 的指针 |
| * 返回值：无 |

特别注意：nrfx_timer_disable 函数内部是调用触发处理寄存器 **TASKS_SHUTDOWN**，来实现关闭计时器的任务。因为一旦定时器被开启后，就会有消耗电流，这种情况下如果采用 TASKS_SHUTDOWN 的方式关闭定时器，能够最大限度的减速电流功耗。而 TASKS_STOP 停止定时定时器的方式则不会。

本例中，我们只需要使能定时器，在主函数最后循环等待定时器中断的发生，代码如下所示

```
01.     //使能定时器
02.     nrf_drv_timer_enable(&TIMER_LED);
03.
04.    while (1)  //等待中断的发生
05.     {
06.     }
```

当定时器定时时间到了，定时时间的值就跟 CC[n]寄存器的值相等时，将触发一个 COMPARE [n] event，产生一个 NRF_TIMER_EVENT_COMPARE0 回调中断处理，来执行 LED 灯的翻转。

```
01. /*  定时器中断回调操作*/
02. void timer_led_event_handler(nrf_timer_event_t event_type, void* p_context)
03. {
04.     static uint32_t i;
05.     uint32_t led_to_invert = (1 << leds_list[(i++) % LEDS_NUMBER]);
06.
07.     switch(event_type)
08.     {
09.         case NRF_TIMER_EVENT_COMPARE0:
10.             LEDS_INVERT(led_to_invert);
11.             break;
12.
13.         default:
14.             break;
15.     }
16. }
```

这个 NRF_TIMER_EVENT_COMPARE0 回调处理事件是在定时器中断中进行判断的，在回调中进行处理。库函数 nrf_time.h 文件中，提供了 nrf_timer_event_t 结构体对应定义了 6 个中断事件，读者可以根据自己的设置进行选择：

```
01. typedef enum
02. {
03.     //来自比较通道 0 的事件
04.     NRF_TIMER_EVENT_COMPARE0 = offsetof(NRF_TIMER_Type, EVENTS_COMPARE[0]),
05.    //来自比较通道 1 的事件
06.     NRF_TIMER_EVENT_COMPARE1 = offsetof(NRF_TIMER_Type, EVENTS_COMPARE[1]),
07.    //来自比较通道 2 的事件
08.     NRF_TIMER_EVENT_COMPARE2 = offsetof(NRF_TIMER_Type, EVENTS_COMPARE[2]),
09.    //来自比较通道 3 的事件
```

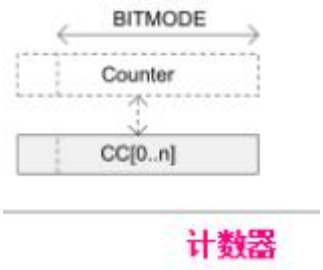
```
    NRF_TIMER_EVENT_COMPARE3 = offsetof(NRF_TIMER_Type, EVENTS_COMPARE[3]),
    //下面两种情况仅限于定时器 4 和定时器 5
#if defined(TIMER_INTENSET_COMPARE4_Msk) || defined(__NRFX_DOXYGEN__)
    //来自比较通道 4 的事件
    NRF_TIMER_EVENT_COMPARE4 = offsetof(NRF_TIMER_Type, EVENTS_COMPARE[4]),
#endif
#if defined(TIMER_INTENSET_COMPARE5_Msk) || defined(__NRFX_DOXYGEN__)
    //来自比较通道 5 的事件
    NRF_TIMER_EVENT_COMPARE5 = offsetof(NRF_TIMER_Type, EVENTS_COMPARE[5]),
#endif
    } nrf_timer_event_t;
```

因此，定时器库函数进行定时，实验整体思路和寄存器配置方式相同。编写后把实验程序下载到青云 nRF52832 开发板后的实验现象如下，LED1~4 灯以 1s 的定时轮流闪烁。

## 11.3 定时器计数功能

### 11.3.1 寄存器编写

计数模式下，每次触发 COUNT 任务时，TIMER 的内部计数器 counter 都会递增 1。但是计数器 counter 计数器内的值是无法读取的，这时需要通过设定一个 CAPTURE Task，捕获的计数器 counter 的值存储到 CC[n]寄存器内，然后再对 CC[n]寄存器进行读取，读取的值就是计数器计数的值。




下面就开始搭建工程，本例中需要使用串口打印计数值，因此工程可以直接采用串口 uart 的例程框架。同时寄存器方式下不需要添加库文件，因此工程目录相比串口例程没有任何变化。

在 main.c 主函数中，我们首先编写定时器初始化函数。由于计数器模式下不使用定时器的频率和预分频器，同样 COUNT 任务在定时器模式下无效。因此初始化的时候，只需要初始化两项：设置计数器模式和设置定时器的位宽。

```
void timer0_init(void)
{
  NRF_TIMER0->MODE = TIMER_MODE_MODE_Counter;     // 设置定时器为计数器模式
  NRF_TIMER0->BITMODE = TIMER_BITMODE_BITMODE_24Bit;   // 设为 24-bit 位宽
}
```

然后通过置位 TASKS_START 寄存器启动定时器。设置寄存器 TASKS_COUNT 为 1，则触发 COUNT 任务，这时才能够开始计数。

计数器的触发有两种方式：

第一种：通过 TASKS_COUNT 计数任务寄存器手动赋值，你赋值为 1，计数器就计数一次。赋值 N 次，就计数 N 次，相当于计数你手动触发的次数。这样实际上是毫无意义的，本例仅仅演示这种方式。

第二钟：通过 PPI 一端的 event 事件，去触发 PPI 另外一端 TASKS_COUNT 任务，那么事件触发的次数就是计数器计数的次数。这种方式就可以用来对外部信号进行捕获计数，常用于实现脉宽测量，下一章会对这种用法进行举例。

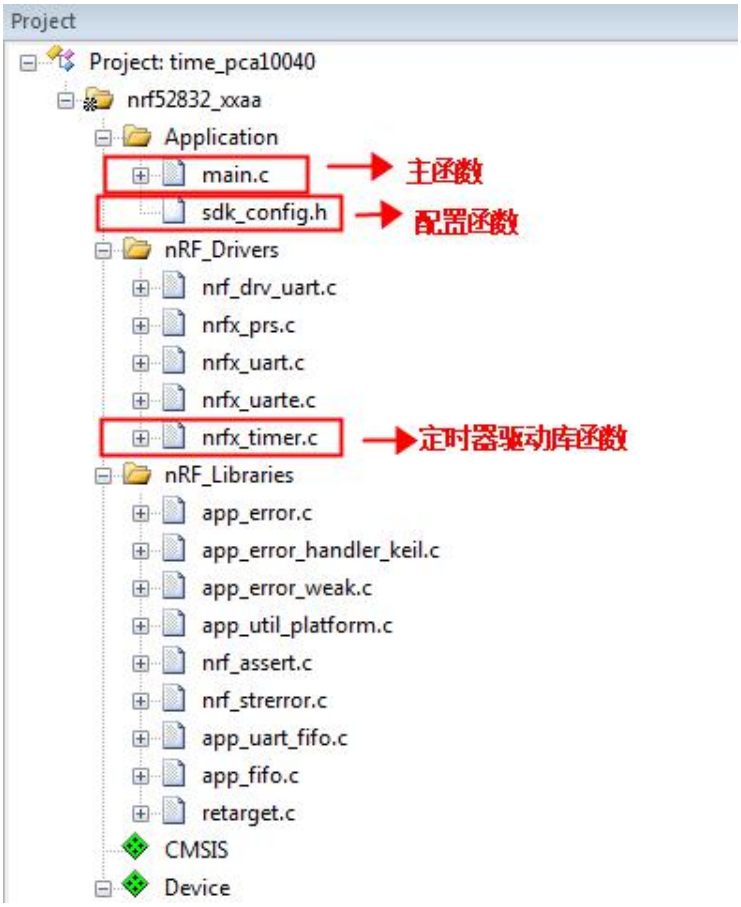
当把 TASKS_CAPTURE[0]寄存器值 1，启动捕获功能。这时就可以捕获计数器 counter 的值，并且存储到 CC[n]寄存器内，然后再对 CC[n]寄存器进行读取了。具体代码如下所示

```
06.       //启动定时器
07.       NRF_TIMER0->TASKS_START=1;
08.
09.       while (1)
10.       {
11.         printf(" 2019.5.1  青风!\r\n");
12.         NRF_TIMER0->TASKS_COUNT = 1;     /*  计数器加 1 */
13.        /*  启动捕获任务.计数的值捕获放入到 CC[n]寄存器  */
14.        NRF_TIMER0->TASKS_CAPTURE[0] = 1;
15.         //获取计数值
16.          timVal = NRF_TIMER0->CC[0];
17.        //串口打印计数值
18.          printf("conut value:    %d\r\n", timVal);
19.          nrf_delay_ms(1000);//每隔 1s 计数一次
20.     }
```

编译程序后下载到青风 nrf52832 开发板内。打开串口调试助手，选择开发板串口端号，设置波特率为 115200，数据位为 8，停止位为 1。


## 11.3.2 计数器组件库编程

工程的建立可以以前面 UARTE 的组件库函数工程为模板，这里面我们只需要改动的如下框中的几个地方:




其中 nrfx_timer.c 文件则需要我们添加的库文件。nrfx_timer.c 文件的路径在 SDK 的//modules/nrfx/drivers/include 文件夹里，添加库文件完成后，然后点击 include paths 路径选项中添加硬件驱动库的文件路径。同时需要来修改 sdk_config.h 配置文件，库函数的使用是需要对库功能进行使能的，因此需要在 sdk_config.h 配置文件中，设置对应模块的使能选项。这两个过程和上一节的定时器组件例程的工程搭建方法相同，这里不再累述。

其中主函数 main.c 文件需要我们编写，下面就具体讨论如何编写该函数文件，其编写过程和寄存器方式相同，首先对定时器进行初始化，上一节定时器已经谈过，初始化调用的函数为 nrfx_timer_init，本节采用计算器方式，只需把默认配置中的模式改为计数器模式，位宽采用默认位宽就可以：

```
01.  void timer0_init(void)
02. {
03.   uint32_t err_code = NRF_SUCCESS;
04.
05.   //定义定时器配置结构体，并使用默认配置参数初始化结构体
06.     nrfx_timer_config_t timer_cfg = NRFX_TIMER_DEFAULT_CONFIG;
07.   //Timer0 配置为计数模式
08.     timer_cfg.mode = NRF_TIMER_MODE_COUNTER;
09.
10.   //初始化定时器，定时器工作于计数模式时，没有事件，所以无需回调函数
11.     err_code = nrfx_timer_init(&TIMER_COUNTER, &timer_cfg, my_timer_event_handler);
```

```
12.     APP_ERROR_CHECK(err_code);
13. }
```

下面一段程序为主函代码段，在主函数中，首先调用计数器初始化函数 timer0_init()，然后采用 nrfx_timer_enable 函数打开计数器。这两个函数在上一节已经介绍过。

和寄存器方式同样的，本例采用 1s 触发一次计数，然后启动捕获，捕获 CC[n]寄存器里的值并且读出来，用串口助手进行打印。这个过程需要调用下面两个函数：

nrfx_timer_increment 函数，用于触发一次计数：

| 函数：void nrfx_timer_increment(nrfx_timer_t const * const p_instance); |
|---|
| *功能：函数用于递增计时器。 |
| * 参数 p_instance 指向驱动程序实例结构的指针。 |
| * 返回值：无 |

nrfx_timer_capture 函数，用于启动捕获，把捕获的值放入到 CC[n]寄存器里的值并且读出来：

| 函数：uint32_t nrfx_timer_capture(nrfx_timer_t const * const p_instance,<br>nrf_timer_cc_channel_t cc_channel); |
|---|
| *功能：函数用于捕获计时器值。 |
| * 参数 p_instance 指向驱动程序实例结构的指针。<br>* 参数 cc_channel 捕获的通道数。 |
| * 返回值：捕获的值。 |

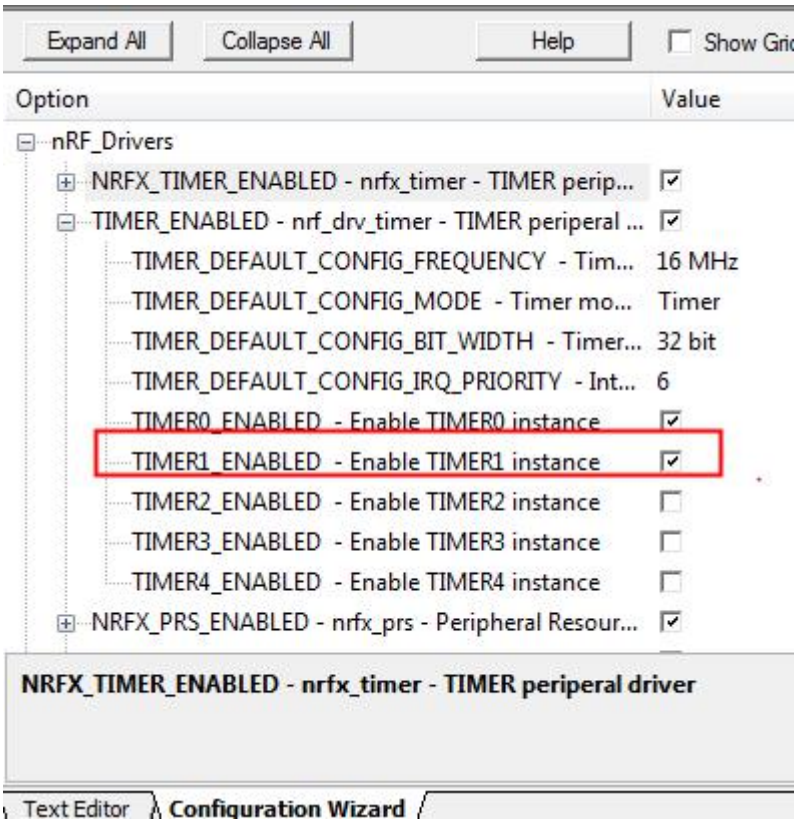
首先需要定义一个定时器的实例结构体，声明为 TIMER_COUNTER，如下所示：

```
const nrfx_timer_t TIMER_COUNTER = NRFX_TIMER_INSTANCE(0);
```

这个声明需要使能在配置文件 sdk_config.h 里使能对应的定时器模块。

例：

如果我们声明为 NRFX_TIMER_INSTANCE(1)，则需要勾选如下图的定时器模块 1 使能：



最后主函数的代码总结编写如下所示：

```
01.   timer0_init();//计数器初始化
02.    nrfx_timer_enable(&TIMER_COUNTER);//使能计数器
03.
04.                  while (1)
05.       {
06.         printf(" 2019.5.1  青风!\r\n");
07.                //定时器计数值加 1
08.         nrfx_timer_increment(&TIMER_COUNTER);
09.         //获取计数值
10.         timVal = nrfx_timer_capture(&TIMER_COUNTER,NRF_TIMER_CC_CHANNEL0);
11.         //串口打印计数值
12.           printf("conut value:    %d\r\n", timVal);
13.             nrf_delay_ms(1000);
14. }
```

编译程序后下载到青风 nrf52832 开发板内。打开串口调试助手，选择开发板串口端号，设置波特率为 115200，数据位为 8，停止位为 1。