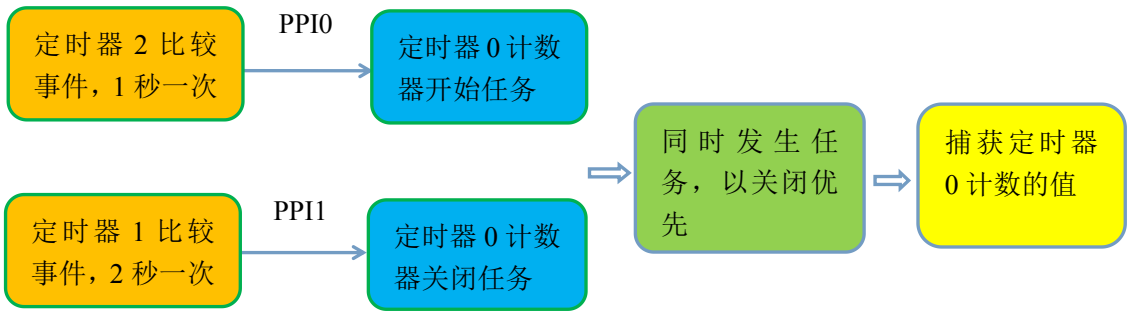
第十三章 定时器和 PPI 的联合应用

13.1 PPI 之定时器计数

13.1.1 PPI 定时器计数寄存器编程

PPI 不仅仅适用于 GPIOTE 的事件触发，还可以用于其他事件来触发任务。其中定时器的应用较为广泛。本章就同时演示了几种 PPI 与定时器的综合应用。本节首先来讨论如何采用 PPI 来启动和关闭定时器的应用。定时器 2 和定时 1 通过 PPI 控制定时器 0 的开启和关闭，那么定时器 0 的打开和关闭可以精确进行控制，具体程序流程如下图所示：



具体代码分析如下所示。

首先对 PPI 通道进行配置：

第 4~5 行：配置 PPI 通道 0，一端 EEP 终点接定时器 1 的比较事件，出发另外一端 TEP 终点的定时器 0 停止任务。当定时器 1 发送比较事件的时候，就会触发定时器 0 关闭。

第 8~9 行：配置 PPI 通道 1，一端 EEP 终点接定时器 2 的比较事件，出发另外一端 TEP 终点的定时器 0 启动任务。当定时器 1 发送比较事件的时候，就会触发定时器 0 启动。

第 12 行：最后使能 PPI 通道 0 和 PPI 通道 1。

```
01. static void ppi_init(void)
02. {
03.    // 配置 PPI 通道 0，一端接定时器 1 的比较事件，出发另外一端的定时器 0 停止任务
04.     NRF_PPI->CH[1].EEP = (uint32_t)(&NRF_TIMER1->EVENTS_COMPARE[0]);
05.     NRF_PPI->CH[0].TEP = (uint32_t)(&NRF_TIMER0->TASKS_STOP);
06.
07.       // 配置 PPI 通道 1，一端接定时器 2 的比较事件，出发另外一端的定时器 0 开始任务
08.      NRF_PPI->CH[1].EEP = (uint32_t)(&NRF_TIMER2->EVENTS_COMPARE[0]);
09.      NRF_PPI->CH[1].TEP = (uint32_t)(&NRF_TIMER0->TASKS_START);
10.
11.      // 使能 PPI 通道 0 和通道 1
12.      NRF_PPI->CHEN=(PPI_CHEN_CH0_Enabled<<PPI_CHEN_CH0_Pos)|
```

```
13.                             (PPI_CHEN_CH1_Enabled << PPI_CHEN_CH1_Pos);
14. }
```

配置定时器 0：

定时器 0 作为被触发开启和关闭的任务，把定时器 0 设置为计数模式。定时器分频值为 9 分频。定时器位宽为 16bit。

```
01.  void timer0_init(void)
02. {   // 设置定时器 0 为计数器模式
03.    NRF_TIMER0->MODE       = TIMER_MODE_MODE_Counter;
04.    NRF_TIMER0->PRESCALER     = 9;//设置定时器 0 的分频
05.    NRF_TIMER0->BITMODE = TIMER_BITMODE_BITMODE_16Bit;   // 设置定时器 0 的位宽
06. }
```

配置定时器 1 和定时器 2：

第 9~17 行：配置定时器 1 为定时器模式，位宽 BITMODE = 16 bit，分频值 PRESCALER = 9，设置比较寄存器 cc 的值为 0xFFFF，那么触发定时器比较事件的时间为：

触发时间=0xFFFF/(SysClk/2^PRESCALER)= 65535/31250 = 2.097 sec

第 28~35 行：配置定时器 2 为定时器模式，位宽 BITMODE = 16 bit，分频值 PRESCALER = 9，设置比较寄存器 cc 的值为 0x7FFF，那么触发定时器比较事件的时间为：

触发时间=0x7FFF/(SysClk/2^PRESCALER)= 32767/31250 = 1.048 sec

配置代码具体如下所示：

```
01. /**
02. 初始化定时器 1
03. */
04. static void timer1_init(void)
05. {
06.       // 配置定时器 1 每 2 秒触发一次
07.       // BITMODE = 16 bit
08.       // PRESCALER = 9
09.       // 触发时间=0xFFFF/(SysClk/2^PRESCALER)= 65535/31250 = 2.097 sec
10.       NRF_TIMER1->BITMODE = (TIMER_BITMODE_BITMODE_16Bit <<
11.                                          TIMER_BITMODE_BITMODE_Pos);
12.       NRF_TIMER1->PRESCALER = 9;
13.       NRF_TIMER1->SHORTS= (TIMER_SHORTS_COMPARE0_CLEAR_Enabled <<
14.                                          TIMER_SHORTS_COMPARE0_CLEAR_Pos);
15.       // 触发比较中断事件
16.       NRF_TIMER1->MODE= TIMER_MODE_MODE_Timer;
17.       NRF_TIMER1->CC[0] = 0xFFFFUL;   // 设置比较寄存器
18. }
19.
20. /** 初始化定时器 2
21. */
22. static void timer2_init(void)
23. {
24.       // 设置定时器 2 每一秒触发一次.
25.       // BITMODE = 16 bit
```

```
        // PRESCALER = 9
        //  触发时间=0x7FFF/(SysClk/2^PRESCALER)= 32767/31250 = 1.048 sec */
        NRF_TIMER2->BITMODE = (TIMER_BITMODE_BITMODE_16Bit <<
                                          TIMER_BITMODE_BITMODE_Pos);
        NRF_TIMER2->PRESCALER = 9;
        NRF_TIMER2->SHORTS = (TIMER_SHORTS_COMPARE0_CLEAR_Enabled <<
                                        TIMER_SHORTS_COMPARE0_CLEAR_Pos);
        //  触发比较中断事件
        NRF_TIMER2->MODE            = TIMER_MODE_MODE_Timer;
        NRF_TIMER2->CC[0]           = 0x7FFFUL;     //  设置比较寄存器
}
```

主函数的功能就是启动定时器 1 和定时器 2 开始运行。定时器 1 每 2 秒会触发一次比较事件，触发关闭定时器 0 任务，定时器 2 每 1 秒会触发一次比较事件，触发启动定时器 0 任务。当同时出现启动和关闭定时器操作的时候，以关闭定时器为准。因此定时器 0，会出现 1,3,5...秒启动计数，2,4,6..秒关闭计数的状态。那么定时器 0 计数的时候会出现每隔 1s 停止计数 1 次的现象，计数值会保持 1s 不变化。具体代码如下所示：

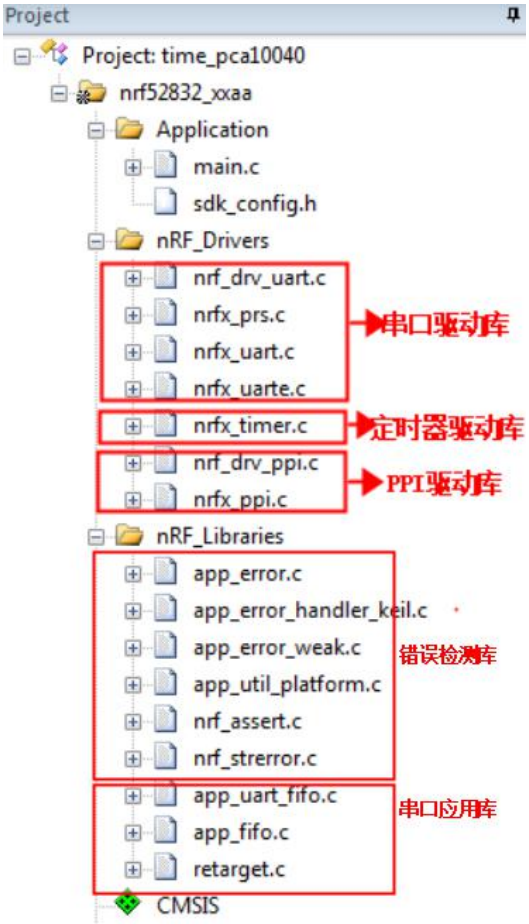
```
/*主函数，定时器 0 设置为计数器模式，通过捕获定时器 0 计数的值在串口输出显示*/
int main(void)
{
    uint32_t err_code;
   uint32_t timVal = 0;

    timer0_init();   //定时器 0 的初始化
    timer1_init();   //定时器 1 初始化
    timer2_init();   //定时器 2 初始化
    ppi_init();      //PPI 的配置
    const app_uart_comm_params_t comm_params =
      {
          RX_PIN_NUMBER,
          TX_PIN_NUMBER,
          RTS_PIN_NUMBER,
          CTS_PIN_NUMBER,
          UART_HWFC,
          false,
#if defined (UART_PRESENT)
          NRF_UART_BAUDRATE_115200
#else
          NRF_UARTE_BAUDRATE_115200
#endif
      };

    APP_UART_FIFO_INIT(&comm_params,
                              UART_RX_BUF_SIZE,
                              UART_TX_BUF_SIZE,
```

```
                                    uart_error_handle,
                                    APP_IRQ_PRIORITY_LOWEST,
                                    err_code);
    APP_ERROR_CHECK(err_code);

    //启动定时器
    NRF_TIMER1->TASKS_START = 1;
    NRF_TIMER2->TASKS_START = 1;
    while (1)
    {
      printf(" 2019.5.1  青风!\r\n");
          /*  计数器加 1 */
      NRF_TIMER0->TASKS_COUNT = 1;
      /*  捕获定时器 0 计数的值  */
      NRF_TIMER0->TASKS_CAPTURE[0]= 1;
      //获取计数值
      timVal = NRF_TIMER0->CC[0];
      //串口打印计数值
      printf("conut value:    %d\r\n", timVal);
      nrf_delay_ms(1048);
    }
}
```

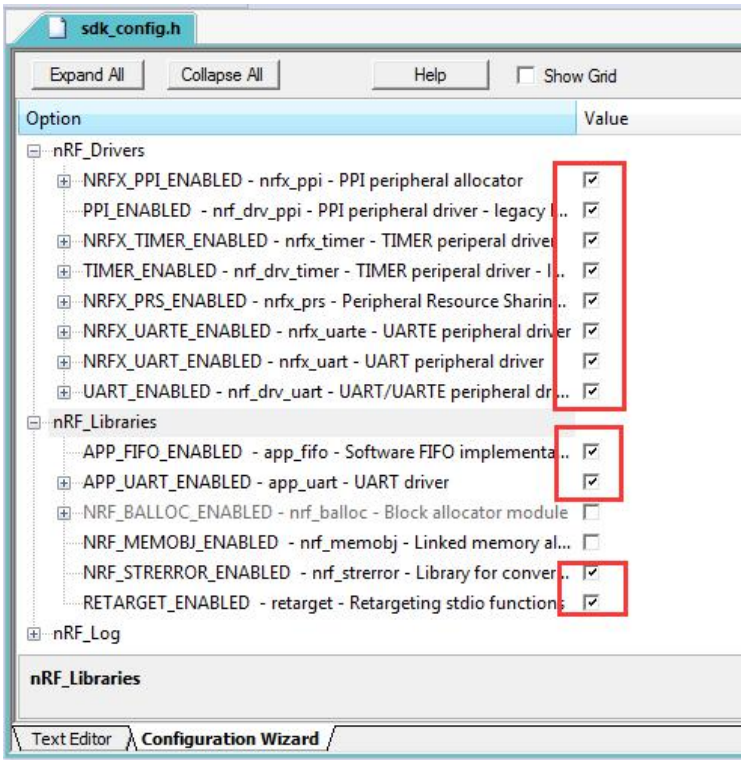
把该例子程序编译后下载到青风 nrf52832 开发板内。连接开发板串口，打开串口调试助手，设置波特率为 115200，数据位为 8，停止位为 1，如下图所示。这时候串口调试助手输出的计数器的计数值会保持 1s 不变：

13.2.1 PPI 定时器计数库函数编程

本节讲解如何采用组件库的方式来实现 PPI 启动定时器的功能。组件库的应用涉及到了定时器、PPI 和 UARTE 的库函数 API 调用。库函数调用方式和配置，和前面定时器、PPI 和 UARTE 的章节相同。具体的搭建工程如下图所示：



然后添加工程路径，这里就不再讲述，可以参考前面几章的内容。同时需要在 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：



代码中，对 PPI 通道进行配置：

第 10~14 行：调用函数 nrf_drv_ppi_channel_alloc 注册 PPI 通道 my_ppi_channel1。调用 nrf_drv_ppi_channel_assign 函数在一端 EEP 终点分配定时器 2 的比较事件，另外一端 TEP 终点分配定时器 0 开始任务。当定时器 1 发送比较事件的时候，就会触发定时器 0 使能。

第 17 行：使能注册的 PPI 通道 my_ppi_channel1。

第 21~25 行：调用函数 nrf_drv_ppi_channel_alloc 注册 PPI 通道 my_ppi_channel2。调用 nrf_drv_ppi_channel_assign 函数在一端 EEP 终点分配定时器 1 的比较事件，另外一端 TEP 终点分配定时器 0 停止任务。当定时器 1 发送比较事件的时候，就会触发定时器 0 关闭。

第 28 行：使能注册的 PPI 通道 my_ppi_channel2。

```
01. /** @brief 初始化 PPI 外设.*/
02. static void ppi_init(void)
03. {
04.     uint32_t err_code = NRF_SUCCESS;
05.
06.     err_code = nrf_drv_ppi_init();
07.     APP_ERROR_CHECK(err_code);
08.
09.     // 配置 PPIT 通道 0，一端接定时器 1 的比较事件，出发另外一端的定时器 0 停止任务
10.     err_code = nrf_drv_ppi_channel_alloc(&my_ppi_channel1);
11.     APP_ERROR_CHECK(err_code);
12.     err_code = nrf_drv_ppi_channel_assign(my_ppi_channel1,
13.             nrf_drv_timer_event_address_get(&timer2, NRF_TIMER_EVENT_COMPARE0),
14.                 nrf_drv_timer_task_address_get(&timer0, NRF_TIMER_TASK_START));
15.     APP_ERROR_CHECK(err_code);
16.     //使能 PPI 通道
```

```
17.       err_code = nrf_drv_ppi_channel_enable(my_ppi_channel1);
18.       APP_ERROR_CHECK(err_code);
19.
20.     // 配置 PPIT 通道 1，一端接定时器 2 的比较事件，出发另外一端的定时器 0 开始任务
21.     err_code = nrf_drv_ppi_channel_alloc(&my_ppi_channel2);
22.       APP_ERROR_CHECK(err_code);
23.       err_code = nrf_drv_ppi_channel_assign(my_ppi_channel2,
24.                         nrf_drv_timer_event_address_get(&timer1, NRF_TIMER_EVENT_COMPARE0),
25.                                   nrf_drv_timer_task_address_get(&timer0, NRF_TIMER_TASK_STOP));
26.       APP_ERROR_CHECK(err_code);
27.       //使能 PPI 通道
28.       err_code = nrf_drv_ppi_channel_enable(my_ppi_channel2);
29.       APP_ERROR_CHECK(err_code);
30. }
```

配置定时器 0：

第 5 行：采用默认定时器的配置，在 sdk_config.h 文件中，把定时器分频值为 9 分频、定时器位宽为 16bit。

第 7 行：单独对定时器 0 模式进行配置，设置为 NRF_TIMER_MODE_COUNTER 计数模式。

第 10 行：调用函数 nrfx_timer_init,设置定时器 0。

```
01.  void timer0_init(void)
02. {
03.     uint32_t err_code = NRF_SUCCESS;
04.     //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
05.       nrfx_timer_config_t timer_cfg = NRFX_TIMER_DEFAULT_CONFIG;
06.     //Timer0 配置为计数模式
07.       timer_cfg.mode = NRF_TIMER_MODE_COUNTER;
08.
09.     //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
10.       err_code = nrfx_timer_init(&timer0, &timer_cfg, my_timer_event_handler);
11.       APP_ERROR_CHECK(err_code);
12. }
```

配置定时器 1 和定时器 2：

第 7~8 行：调用函数 nrf_drv_timer_init()采用配置 NRF_DRV_TIMER_DEFAULT_CONFIG 设置定时器 1 为定时器模式、位宽 BITMODE = 16 bit、分频值 PRESCALER = 9。

第 10 行： 调用函数 nrf_drv_timer_extended_compare()设置比较寄存器 cc 的值为 0xFFFF，那么触发定时器 1 比较事件的时间为：

触发时间=0xFFFF/(SysClk/2^PRESCALER)= 65535/31250 = 2.097 sec

第 13 行：调用函数 nrf_drv_timer_enable()使能定时器 1。

第 21~22 行：调用函数 nrf_drv_timer_init()采用配置 NRF_DRV_TIMER_DEFAULT_CONFIG 设置定时器 2 为定时器模式、位宽 BITMODE = 16 bit、分频值 PRESCALER = 9。

第 24 行： 调用函数 nrf_drv_timer_extended_compare()设置比较寄存器 cc 的值为 0x7FFF，那么触发定时器 2 比较事件的时间为：

触发时间=0x7FFF/(SysClk/2^PRESCALER)= 32767/31250 = 1.048 sec

第 27 行：调用函数 nrf_drv_timer_enable()使能定时器 2。

配置代码具体如下所示：

```
01. static void timer1_init(void)
02. {
03.       uint32_t time_ticks1;
04.       ret_code_t err_code;
05.
06.       //配置定时器比较事件
07.       nrf_drv_timer_config_t timer_cfg = NRF_DRV_TIMER_DEFAULT_CONFIG;
08.       err_code = nrf_drv_timer_init(&timer1, &timer_cfg,   NULL);
09.       APP_ERROR_CHECK(err_code);
10.     nrf_drv_timer_extended_compare(&timer1, NRF_TIMER_CC_CHANNEL0, 0xFFFFUL,
11.                                    NRF_TIMER_SHORT_COMPARE0_CLEAR_MASK, false);
12.       //使能定时器 1
13.       nrf_drv_timer_enable(&timer1);
14. }
15.
16. static void timer2_init(void)
17. {
```

```
    uint32_t time_ticks;
     ret_code_t err_code;
     //配置定时器比较事件
      nrf_drv_timer_config_t timer_cfg = NRF_DRV_TIMER_DEFAULT_CONFIG;
     err_code = nrf_drv_timer_init(&timer2, &timer_cfg,  NULL);
     APP_ERROR_CHECK(err_code);
     nrf_drv_timer_extended_compare(&timer2, NRF_TIMER_CC_CHANNEL0, 0x7FFFUL,
                                    NRF_TIMER_SHORT_COMPARE0_CLEAR_MASK, false);
     //使能定时器 2
     nrf_drv_timer_enable(&timer2);
}
```

主函数的功能就是调用前面编写的定时器 0、定时器 1 和定时器 2 初始化函数。由于定时器 1 每 2 秒会触发一次比较事件，触发关闭定时器 0 任务；定时器 2 每 1 秒会触发一次比较事件，触发启动定时器 0 任务。

当同时出现启动和关闭定时器操作的时候，以关闭定时器为准。因此定时器 0 会出现 1,3,5...秒启动计数，2,4,6..秒关闭计数的状态。那么定时器 0 计数的时候会出现每隔 1s 停止计数 1 次的现象，计数值会保持 1s 不变化。具体代码如下所示：

```
/**主函数*/
int main(void)
{
  uint32_t err_code;
   uint32_t timVal = 0;
    timer0_init();//定时器 0 初始化
   timer1_init();//定时器 1 初始化
   timer2_init();//定时器 2 初始化
   ppi_init();//初始化 PPI
  const app_uart_comm_params_t comm_params =
      {
          RX_PIN_NUMBER,
          TX_PIN_NUMBER,
          RTS_PIN_NUMBER,
          CTS_PIN_NUMBER,
          UART_HWFC,
          false,
#if defined (UART_PRESENT)
          NRF_UART_BAUDRATE_115200
#else
          NRF_UARTE_BAUDRATE_115200
#endif
      };

     APP_UART_FIFO_INIT(&comm_params,
                        UART_RX_BUF_SIZE,
                        UART_TX_BUF_SIZE,
```

```
28.                                 uart_error_handle,
29.                                 APP_IRQ_PRIORITY_LOWEST,
30.                                 err_code);//初始化串口
31.       APP_ERROR_CHECK(err_code);
32.        nrf_drv_timer_enable(&timer0);
33.        while (1)
34.        {
35.          printf(" 2019.5.1  青风!\r\n");
36.          nrfx_timer_increment(&timer0);
37.          //获取计数值
38.          timVal = nrfx_timer_capture(&timer0,NRF_TIMER_CC_CHANNEL0);
39.          //串口打印计数值
40.          printf("conut value:    %d\r\n", timVal);
41.          nrf_delay_ms(1048);
42.      }
43. }
```

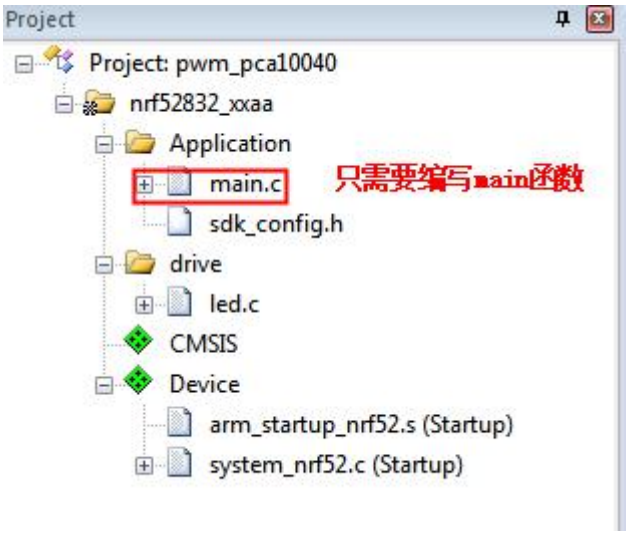
把该例子程序编译后下载到青风 nrf52832 开发板内。连接开发板串口，打开串口调试助手，设置波特率为 115200，数据位为 8，停止位为 1，如下图所示。这时候串口调试助手输出的计数器的计数值会保持 1s 不变：

13.2 定时器与 PPI 之软件 PWM

13.2.1 PPI 应用之 PWM

在代码文件中，实验建立了一个演示历程。打开 user 文件夹中的工程如图所示：本节将使用

寄存器直接编程，则只需要在主函数 main.c 的文件编写 ppi 的驱动内容。



根据前面讲解的原理，PPI 的结构其实非常简单的。PPI 实际上提供了一种直连的机制，这种机制可以把一个外设发生的事件（event）来触发另一个外设的任务（task），整个过程不需要 CPU 进行参与。

因此一个任务（task）通过 PPI 通道和事件（event）进行互连。PPI 通道由两个终点寄存器组成，分别为：事件终点寄存器 (EEP) 和任务终点寄存器 (TEP)。可以把外设任务（task）通过任务寄存器的地址与任务终点寄存器(TEP)进行赋值。同理，也可以把外设事件通过事件（event）寄存器的地址与事件终点寄存器(EEP) 进行赋值 。

PPI 输出软件 PWM 实际上是由定时器来触发的 GPIOTE 的输出电平变化实现的。那么 PPI 的作用就是连接定时器作为任务（task），触发的 GPIOTE 的输出作为事件（event）。例子中，我们编写两路 PWM 输出，所以需要使用 4 路 PPI 通道。每两路 PPI 通道产生一个 PWM 输出，其中一路作为占空比的输出控制，一路作为 PWM 周期的控制。整个过程 CPU 不参与其中。

那么按照上面的分析，我们首先来配置 PPI 通道，设置代码如下：

```
01.    void ppi_set(void)
02. {
03.      // PPI 通道 0 的 EFP 和 TEP 设置  PWM1 占空比的时间
04.     //把定时器 0 的比较事件作为事件，GPIOTE0 的输出作为任务
05.     //通过定时器 0 比较事件来触发 GPIOTE0 的输出
06.      NRF_PPI->CH[0].EEP = (uint32_t)(&NRF_TIMER0->EVENTS_COMPARE[0]);
07.      NRF_PPI->CH[0].TEP = (uint32_t)(&NRF_GPIOTE->TASKS_OUT[0]);
08.
09.      // PPI 通道 1 的 EFP 和 TEP 设置  PWM2 周期的时间
10.      //把定时器 0 的比较事件作为事件，GPIOTE0 的输出作为任务
11.     //通过定时器 0 比较事件来触发 GPIOTE0 的输出
12.      NRF_PPI->CH[1].EEP = (uint32_t)(&NRF_TIMER0->EVENTS_COMPARE[1]);
13.      NRF_PPI->CH[1].TEP = (uint32_t)(&NRF_GPIOTE->TASKS_OUT[0]);
14.
15.      NRF_PPI->CH[2].EEP = (uint32_t)(&NRF_TIMER0->EVENTS_COMPARE[2]);
16.      NRF_PPI->CH[2].TEP = (uint32_t)(&NRF_GPIOTE->TASKS_OUT[1]);
17.
18.      NRF_PPI->CH[3].EEP = (uint32_t)(&NRF_TIMER0->EVENTS_COMPARE[3]);
```

```
19.       NRF_PPI->CH[3].TEP = (uint32_t)(&NRF_GPIOTE->TASKS_OUT[1]);
20.
21.       // 使能 PPI 通道 1 和通道 0，2，3
22.         NRF_PPI->CHENSET = 0x0f;
23. }
```

这个代码实际上完成了 2 个事情：

第 03 行~19 行：配置 PPI 通道实际上就是设置 CH[n].EEP 和 CH[n].TEP 的地址。一旦连接好了任务和事件，整个过程不需要 CPU 参与，和 DMA 有点相似。把定时器 0 的比较事件作为事件，GPIOTE0 的输出作为任务。当比较寄存器 EVENTS_COMPARE[n]被置为 1 的时候，则会触发 TASKS_OUT[n]事件。

第 22 行：最后是使能通道，实际上开通道和关通道有两种方式：

•方法 1：通过独立设置 CHEN, CHENSET, and CHENCLR 寄存器。

•方法 2 ：通过 PPI 频道组的使能和关断任务。

我们选择通过 CHENSET 设置来使能通道。

然后我们再来配置定时器，在 PPI 中把定时器的寄存器 COMPARE[n]作为事件 EEP，那么定时器计数器计数到预设值 CC[n]寄存器的值时，启动比较事件将把 COMPARE [n]置为 1，所以在定时器的配置中，通过配置 CC[n]寄存器的值来触发 COMPARE [n]。代码如下所示：

```
01. void timer0_init(void)
02.  {
03.       NRF_TIMER0->PRESCALER    = 4;        //2^4    16 分频成 1M 时钟源
04.       NRF_TIMER0->MODE =TIMER_MODE_MODE_Timer;//设置为定时模式
05.       NRF_TIMER0->BITMODE = 3;       //32bit
06.       NRF_TIMER0->CC[1] = 1000;       //cc[1]的值等于是 1s，这里相当于方波的周期为 1s
07.       NRF_TIMER0->CC[0] =10;          //调节占空比
08.
09.      NRF_TIMER0->CC[2] = 1000;       //cc[2]的值等于是 1s，这里相当于方波的周期为 1s
10.       NRF_TIMER0->CC[3] =990;          //调节占空比
11.
12.       NRF_TIMER0->SHORTS = 1<<1; //设置到计数到 cc1 中的值时  自动清 0 重新开始计数
13.       NRF_TIMER0->SHORTS = 1<<2; //设置到计数到 cc2 中的值时  自动清 0 重新开始计数
14.
15.       NRF_TIMER0->TASKS_START = 1;       //启动 timer
16. }
```

第 3 行，设置定时器的预分频寄存器的值，根据定时器一节里的计算公式，预分频后定时器的频率为 1MHz。

第 4 行，设置定时器的模式为定时模式。

第 5 行，设置定时器的位宽为 32bit 位宽。

第 6 行~第 7 行，分别设置定时器的两个预设寄存器的值，那么一个做为最后输出 PWM 波的周期，一个作为占空比的时间。

第 9 行~第 10 行，设置第二个 PWM 波的对应周期和占空比的值。

第 12 行~第 13 行，由于前面 cc1 和 cc2 作为周期值，那么需要再计数到对应设置值的时候，清 0 定时器重新计数。

第 15 行，启动定时器 0。

上面就把定时器一端设置完了，ppi 另一端 GPIOTE 输出也需要设置。设置两个输出管脚连接

到 LED 灯，代码如下：

```
#define PWM_OUT1                LED_0
#define PWM_OUT2                LED_1

void gpiote_init(void){
    NRF_GPIOTE->CONFIG[0] = ( 3 << 0 )              //作为 task 模式
                          | ( PWM_OUT1 << 8) //设置 PWM 输出引脚
                          | ( 3 << 16 )       //设置 task 为翻转 PWM 引脚的电平
                          | ( 1 << 20);       //初始输出电平为高
  NRF_GPIOTE->CONFIG[1] = ( 3 << 0 )              //作为 task 模式
                          | ( PWM_OUT2 << 8) //设置 PWM 输出引脚
                          | ( 3 << 16 )       //设置 task 为翻转 PWM 引脚的电平
                          | ( 1 << 20);       //初始输出电平为高
}
```

GPIOTE 输出主要是配置模式、输出管脚、设置 task 为翻转 PWM 引脚的电平、初始化输出的电平。

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。初始化 PPI、定时器、GPIOTE 后，循环等待。PWM 波就会通过 IO 管脚输出到 LED 灯上，通过两类 PWM 波输出对比，不同占空比的 LED 灯亮度不同：

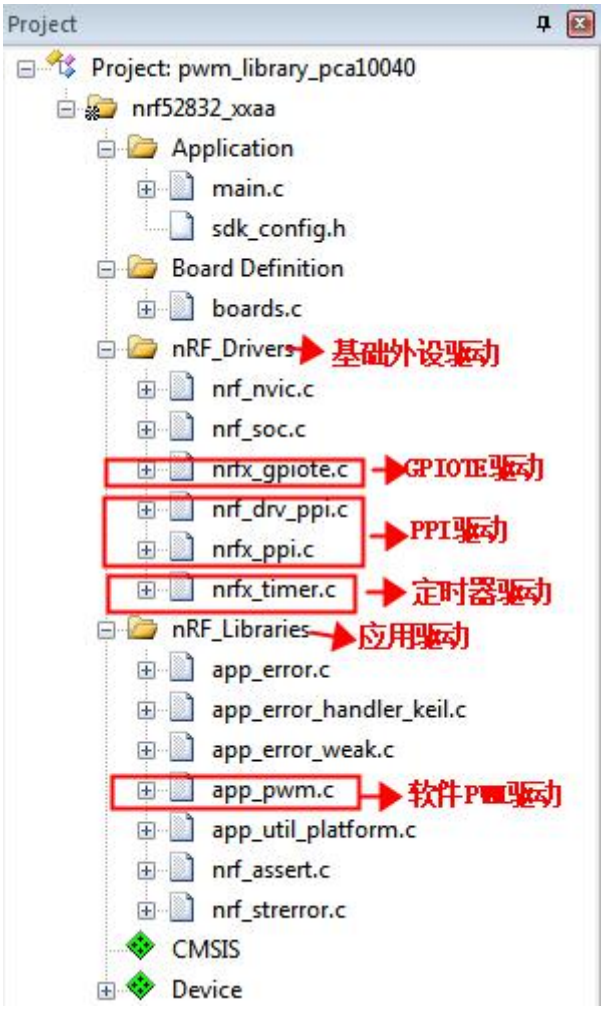
```
int main(void){

    gpiote_init();
    ppi_set();
    timer0_init();
    while(1);
}
```

把该例子程序编译后下载到青风 nrf52832 开发板内。PWM 波就会通过 IO 管脚输出到 LED1 灯和 LED2 灯上，通过两个 PWM 波输出对比，不同占空比的 LED1 和 LED2 灯亮度不同。

用示波器测试 PWM 的波形如下图所示：

13.2.2 软件 PWM 组件库编程

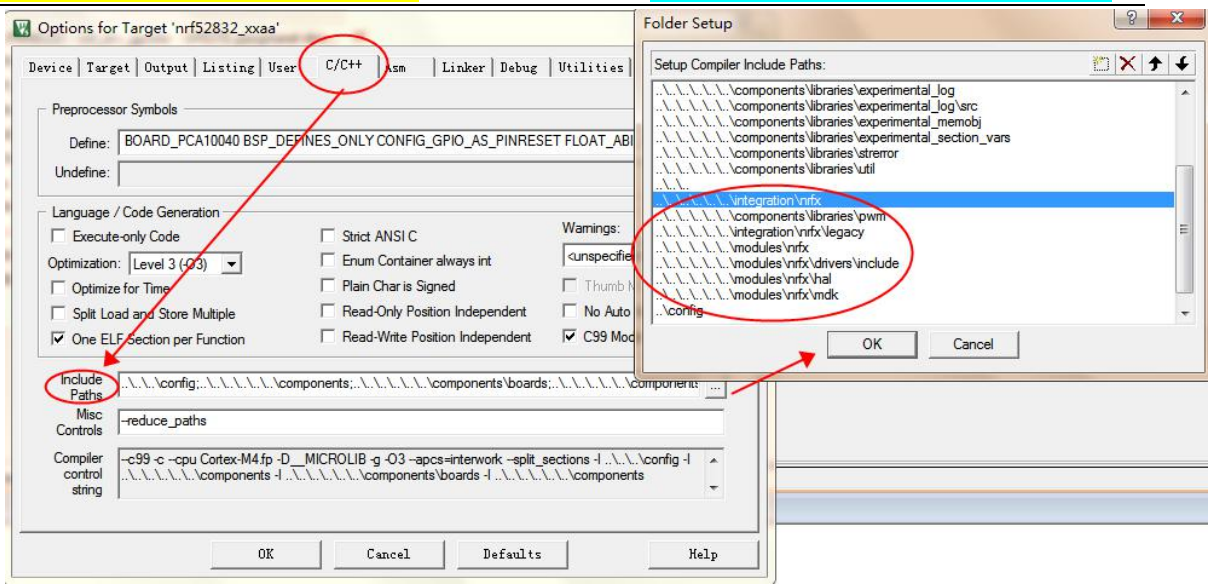
通过上面的寄存器编程，读者已经深入了解定时器 PWM 输出的编写方法与过程，那么在这个基础之上，再来理解组件库编程就变的十分容易了。组件库来实现输出 PWM 的过程实际上与寄存器一样，只是官方提供了很多基础的库函数，库函数为了照顾所有的功能，内容比较繁杂，理解时需要深入到基础代码中学习。组件库的函数工程代码如下，需要添加如下图框图中的驱动库：



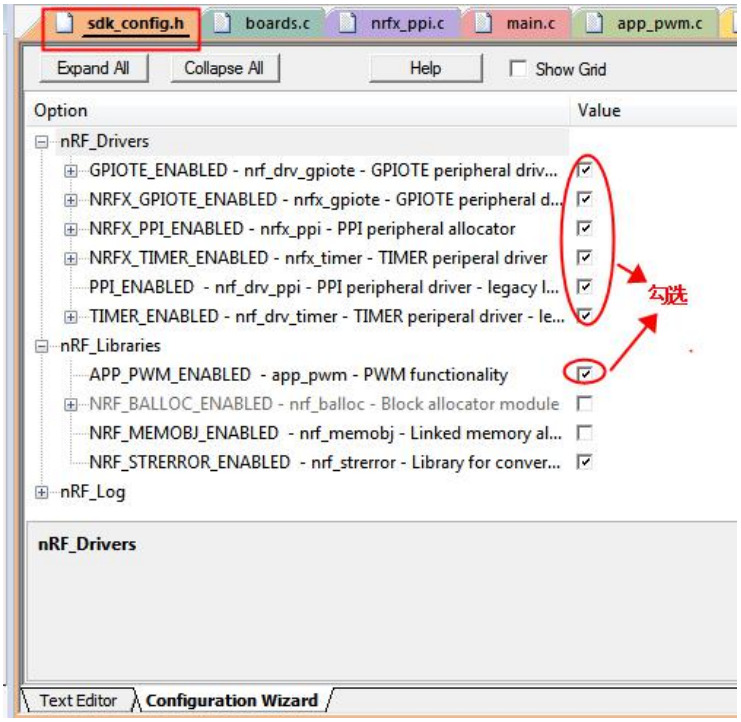
需要添加的函数：

新增文件名称	功能描述
nrfx_gpiote.c	新版本 gpiote 串口兼容库
nrfx_drv_ppi.c	新版本 ppi 兼容库
nrf_ppi.c	老版本 ppi 驱动库
nrfx_timer.c	新版本 time 定时器兼容库
app_pwm.c	应用软件 PWM 的驱动库

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：



组件库在实现输出 PWM 的功能时，提供了两个关键函数，这两个函数介绍如下：

1：函数 app_pwm_init 软件 PWM 的初始化，并且声明回调函数：

函数：ret_code_t app_pwm_init(app_pwm_t const * const p_instance, app_pwm_config_t const * const p_config, app_pwm_callback_t p_ready_callback);
*功能：初始化 PWM 输出.

* 参数：p_instance 创建的 PWM 使用实例 * 参数：p_config PWM 参数的初始化. * 参数：p_ready_callback 执行回调函数.
* 返回值 NRF_SUCCESS 如果初始化成功. * 返回值 NRF_ERROR_NO_MEM 如果没有足够的资源. * 返回值 NRF_ERROR_INVALID_PARAM 如果给的一个无效的结构体配置. * 返回值 NRF_ERROR_INVALID_STATE 如果 timer/PWM 正在使用或者初始化失败

2：函数 app_pwm_channel_duty_set 用于设置软件 PWM 的占空比的值：

函数：ret_code_t app_pwm_channel_duty_set(app_pwm_t const * const p_instance, uint8_t channel, app_pwm_duty_t duty);
*功能：设置 PWM 的占空比的值. 说明：占空比的改变需要一个完整的 PWM 时钟周期来完成。如果在当前更改完成之前尝试对给定实例的任何通道进行另一个更改，则新尝试将导致错误 NRF_ERROR_BUSY。
* 参数：p_instance 创建的 PWM 使用实例 * 参数：channel 控制占空比的 PPI 频道 * 参数： duty 占空比（0-100）
* 返回值 NRF_SUCCESS 如果操作成功 * 返回值 NRF_ERROR_BUSY 如果 PWM 还没准备好. * 返回值 NRF_ERROR_INVALID_STATE 如果给定的 PWM 实例没有初始化.

这两个函数是实现 PWM 输出的关键了，首先来看下如何配置 app_pwm_init 函数，该函数主要需要配置第二个参数：app_pwm_config_t const * const p_config，这个参数是一个结构体形式，官方给出了单 PWM 输出和双 PWM 输出的定义：

APP_PWM_DEFAULT_CONFIG_1CH 和 APP_PWM_DEFAULT_CONFIG_2CH,比如双通道定义:

```
:#define APP_PWM_DEFAULT_CONFIG_2CH(period_in_us, pin0, pin1)
    {
    .pins   = {pin0, pin1},
    .pin_polarity={APP_PWM_POLARITY_ACTIVE_LOW,APP_PWM_POLARITY_ACTIVE_LO
                   W},
    .num_of_channels = 2,
    .period_us= period_in_us
    }
```

结构体主要包括：

.pins 为管脚;

.pin_polarity 为管脚极性;

.num_of_channels 为通道数量;

.period_us 为 PWM 的周期。

配置好后就了就可以直接使用 app_pwm_init 函数了。这个函数相当于寄存器配置中配置 PWM 周期的定时器比较事件配置。深入到 app_pwm_ini 函数内部，大家会找到这段配置定时器和定时器初始化配置参数：

```
    // 初始化定时器，估算定时器频率
    nrf_timer_frequency_t   timer_freq = pwm_calculate_timer_frequency(p_config->period_us);
    nrf_drv_timer_config_t timer_cfg   = {
        .frequency          = timer_freq,
        .mode               = NRF_TIMER_MODE_TIMER,//定时器模式
        .bit_width          = NRF_TIMER_BIT_WIDTH_16,//16 位宽
        .interrupt_priority = APP_IRQ_PRIORITY_LOW,//低优先级
        .p_context          = (void *) (uint32_t) p_instance->p_timer->instance_id//例子 ID
    };
    err_code = nrf_drv_timer_init(p_instance->p_timer, &timer_cfg,
                                  pwm_ready_tick);//带入注册参数
```

同时函数内部采用了 nrf_drv_ppi_channel_assign 分配 PPI 通道，设置了两个 PPI 通道，一个作为主通道，一个第二从通道使用。

```
#define PWM_MAIN_CC_CHANNEL                 2
#define PWM_SECONDARY_CC_CHANNEL            3
```

在看 app_pwm_channel_duty_set 函数的设置，这个函数设置占空比，同时需要给出占空比触发时使用的 PPI 通道，我们在例子里分频 PPI 通道 0 和通道 1 作为占空比触发的 PPI 通道，代码可以设置如下：

```
    app_pwm_channel_duty_set(&PWM1, 0, value)
```

主函数就可以直接调用库函数，首先设置创建一个 PWM 使用实例，然后配置 PWM 初始化，再使能 PWM。通过一个 for 循环不停改变 PWM 占空比，我们可以通过 LED 灯亮度的变化来观察占空比的变化，具体代码如下所示：

```
APP_PWM_INSTANCE(PWM1,1);                   // 创建一个使用定时器 1 产生 PWM 波的实例

static volatile bool ready_flag;            // 使用一个标志位表示 PWM 状态

void pwm_ready_callback(uint32_t pwm_id)    // PWM 回调功能
{
    ready_flag = true;
}

int main(void)
{
    ret_code_t err_code;

    /* 2-个频道的 PWM 参数定义 200Hz（5000us=5ms), 通过开发板 LED 管脚输出. */
    app_pwm_config_t   pwm1_cfg = APP_PWM_DEFAULT_CONFIG_2CH(5000L, BSP_LED_0, BSP_LED_1);

    /* 切换两个频道的极性 */
    pwm1_cfg.pin_polarity[1] = APP_PWM_POLARITY_ACTIVE_HIGH;
```

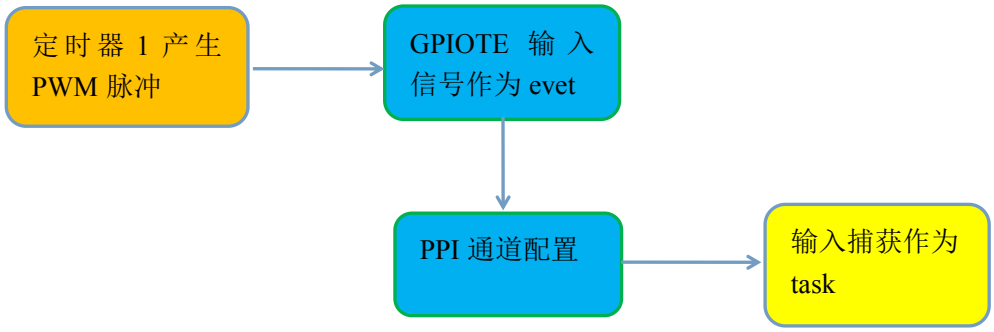
```
19.
20.     /* 初始化和使能 PWM. */
21.     err_code = app_pwm_init(&PWM1,&pwm1_cfg,pwm_ready_callback);
22.     APP_ERROR_CHECK(err_code);
23.     app_pwm_enable(&PWM1);//使能 PWM
24.
25.     uint32_t value;
26.     while(true)
27.     {
28.         for (uint8_t i = 0; i < 40; ++i)
29.         {
30.             value = (i < 20) ? (i * 5) : (100 - (i - 20) * 5);
31.             ready_flag = false;
32.             /* 设置占空比 ：不停设置直到 PWM 准备好. */
33.             while (app_pwm_channel_duty_set(&PWM1, 0, value) == NRF_ERROR_BUSY);
34.             /* 等待回调 */
35.             while(!ready_flag);
36.             APP_ERROR_CHECK(app_pwm_channel_duty_set(&PWM1, 1, value));
37.             nrf_delay_ms(25);
38.         }
39.     }
40.
41. }
```

由于在代码中设置的管脚极性相反，输出 PWM 占空比相反，所以观察两个 LED 灯的亮度变化正好相反。一个不停的变亮，到达最亮后开始反方向变暗，一个不停变暗，到达熄灭后开始反方向不停变量。

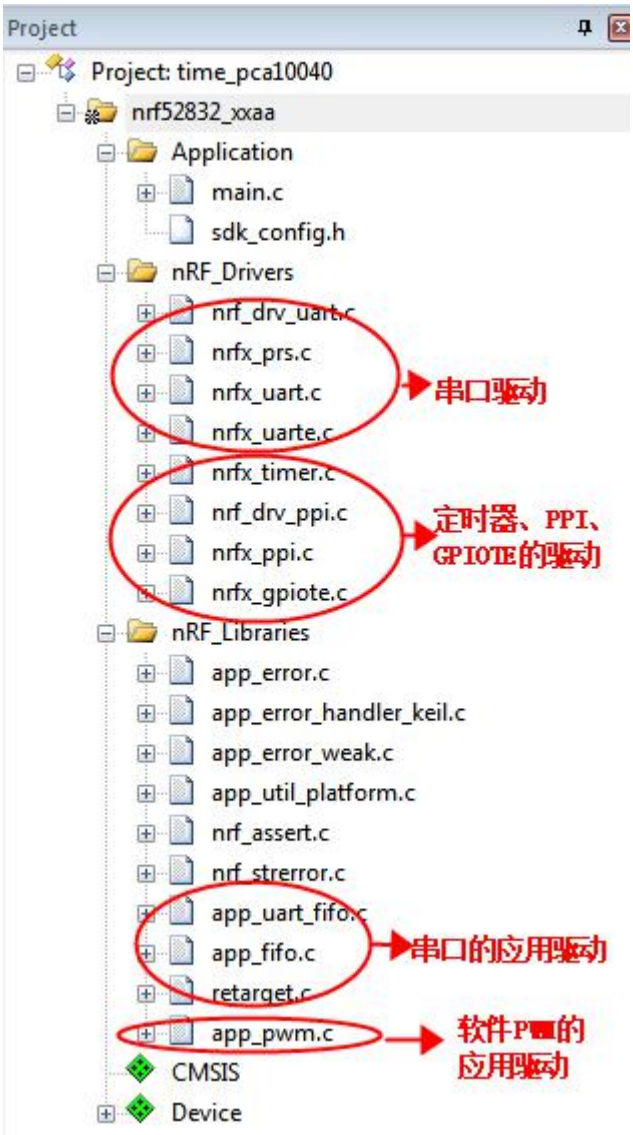
13.3 PPI 之输入捕获

13.3.1 原理分析

本实验通过 PPI 来实现定时器的输入捕获功能。本例的 PPI 的功能就是通过一端的 GPIOTE 的输入脉冲信号作为事件 evet，触发 PPI 另一端的定时器捕获功能作为任务 task，并且把定时器捕获到的脉冲计数值通过串口输出。而输入脉冲信号则通过上一节的软件 PWM 来提供。因此综合了 GPIOTE、定时器、软件 PWM 的多项功能的综合性应用。



那么工程中需要调用几个关键的组件库：PPI 的组件库、定时器的组件库、GPIOTE 的组件库、串口的组件库。然后更加这几个组件库来编写主函数 main。那么工程目录树添加入下图所示：



关于工程路径的添加，sdk_cofig.h 配置文件的配置，参考前面章节的讲述。具体可以参考配套

的工程代码。

13.2.2 应用实例编写

程序设计就按照下面几步来实现：首先需要设置一个脉冲输入，使用 PWM 方式输出信号，为了方便串口输出观察，我们把 PWM 的频率定为 5HZ。PWM 波的设置在上一节已经讲过，这里不再累述，具体代码如下，我们这里只需要一路 PWM 输出：

```
01. void PWM_OUT(uint32_t value)
02. {
03.
04.    ret_code_t err_code;
05.
06.       /* 2-个频道的 PWM, 0.5Hz（2000000us=2000ms), 通过 开发板 LED 管脚输出. */
07.       app_pwm_config_t pwm1_cfg = APP_PWM_DEFAULT_CONFIG_2CH(2000000L, OUTPUT,
    BSP_LED_1);
08.       /* 切换两个频道的极性 */
09.       pwm1_cfg.pin_polarity[1] = APP_PWM_POLARITY_ACTIVE_HIGH;
10.       /* 初始化和使能 PWM. */
11.       err_code = app_pwm_init(&PWM1,&pwm1_cfg,pwm_ready_callback);
12.       APP_ERROR_CHECK(err_code);
13.       app_pwm_enable(&PWM1);//使能 PWM
14.        /* 只采用其中一路 PWM 输出. */
15.       while (app_pwm_channel_duty_set(&PWM1, 0, value) == NRF_ERROR_BUSY);
16.
17.
18. }
```

设置好的 PWM 脉冲信号需要被采样，采样管脚可以使用 gpiote 输入作为事件来触发 PPI，因此这一步来设置 gpiote 输入，具体代码如下：

```
01. static void gpio_init(void)
02. {
03.       ret_code_t err_code;
04.       //GPIOE 驱动初始化
05.       err_code = nrf_drv_gpiote_init();
06.       APP_ERROR_CHECK(err_code);
07.       //配置设置 GPIOTE 输入参数
08.       nrf_drv_gpiote_in_config_t in_config = GPIOTE_CONFIG_IN_SENSE_HITOLO(1);
09.       in_config.pull = NRF_GPIO_PIN_PULLUP;
10.        //GPIOTE 输入初始化，设置触发输入中断
11.       err_code = nrf_drv_gpiote_in_init(INPUT, &in_config, NULL);
12.       APP_ERROR_CHECK(err_code);
13.       //设置 GPIOE 输入事件使能
14.       //nrf_drv_gpiote_in_event_enable(INPUT, 1);
15. }
```

上面代码第 3 行：对 GPIOTE 这个模块进行初始化。

都 7 行到第 11 行：配置 GPIOTE 输入事件。参数设置输入管脚为：INPUT，可以在程序前对 INPUT 进行定义为任何端口。那么实验的时候就需要把 PWM 输出管脚和 INPUT 管脚通过杜邦线进行短接。

管脚配置采用：

nrf_drv_gpiote_in_config_t in_config = GPIOTE_CONFIG_IN_SENSE_HITOLO(1)进行定义。nrf_drv_gpiote_in_config_t 作为一个结构体定义了作为输入管脚以下几个参数：

```
typedef struct
{
    nrf_gpiote_polarity_t  sense;          /*中断触发的方式 */
    nrf_gpio_pin_pull_t    pull;           /*下拉上拉模式 */
    bool       is_watcher;                 /* 设为 1 时，输入管脚被输出管脚跟踪.*/
    bool       hi_accuracy;                /*设置 1 时，输入事件设置为高电平感应*/
    bool        skip_gpio_setup : 1;       /*GPIO 配置不改变 */
} nrf_drv_gpiote_in_config_t;
```

那么本例中配置为输入管脚类型为：NRF_GPIO_PIN_PULLUP，高电平感应，触发中断类型为高电平到低电平。

然后初始化 PPI 外设。设置 PPI 的输入捕获，以 GPIOTE 输入作为 event 事件，以定时器计数器计数作为 task 任务。捕获到的脉冲个数通过计数器计数，具体代码如下：

```
01. static void ppi_init(void)
02. {
03.     uint32_t err_code = NRF_SUCCESS;
04.     err_code = nrf_drv_ppi_init();//PPI 驱动初始化
05.     APP_ERROR_CHECK(err_code);
06.
07.     // 配置 PPI 通道 2 触发事件和发起的任务
08.     err_code = nrf_drv_ppi_channel_alloc(&ppi_channel2);//分频通道
09.     APP_ERROR_CHECK(err_code);
10.     err_code = nrf_drv_ppi_channel_assign(ppi_channel2,
11.                                 nrf_drv_gpiote_in_event_addr_get(INPUT),
12.
    nrf_drv_timer_task_address_get(&timer0,NRF_TIMER_TASK_COUNT));
13.     APP_ERROR_CHECK(err_code);
14.
15.     err_code = nrf_drv_ppi_channel_enable(ppi_channel2);//PPI 通道使能
16.     APP_ERROR_CHECK(err_code);
17. }
```

上面代码第 4 行：对 PPI 驱动进行初始化，这个函数比较简单，实际只是设置了 PPI 的状态标志位，表面当前 PPI 的工作状态。工作状态分配了一个结构体：

```
typedef enum
{
    NRF_DRV_STATE_UNINITIALIZED, /*没有初始化 */
    NRF_DRV_STATE_INITIALIZED, /* 初始化了但是电源未打开 */
    NRF_DRV_STATE_POWERED_ON/* 初始化了，同时电源打开 */
```

} nrf_drv_state_t;

第 8 行：分配 PPI 通道，设置 PPI 使用第几个通道。

第 10 行：分配前面 PPI 通道两端的地址，这也是设置 PPI 的关键了。对应各个外设，SDK 的库函数都分配了地址的获取函数，比如本例中使用的：

GPIOTE 的事件 event 地址获取函数：nrf_drv_gpiote_in_event_addr_get

TIMER 的任务 task 地址获取函数： nrf_drv_timer_task_address_get

第 15 行：使能前面分配的 PPI 通道。

PPI 设置好后，做为 TASK 的定时器也需要进行配置，对使用的定时器 0 进行初始化，关于定时器的初始化前面章节有具体介绍，这里就不累述，具体代码如下：

```
01. static void timer0_init(void)
02. {
03.     nrf_drv_timer_config_t TIME_config= NRF_DRV_TIMER_DEFAULT_CONFIG;
04.     TIME_config.mode=NRF_TIMER_MODE_COUNTER;
05.       ret_code_t err_code = nrf_drv_timer_init(&timer0, &TIME_config, timer_event_handler);
06.       APP_ERROR_CHECK(err_code);
07. }
```

主函数调用之前初始化函数，同时需要使用串口输出计数值，因此还需要对串口进行配置，然后调用定时器比较器捕获功能，使用函数：

nrf_drv_timer_capture(&timer0,NRF_TIMER_CC_CHANNEL2) 这个函数触发一个捕获 capture 任务，把计数器 COUNT 内的值，通过 CHANNEL 频道拷贝到 CC[N]寄存器中，函数返回该寄存器的值作为捕获的值，然后我们才有串口进行输出。主函数的具体代码如下所示：

```
01. int main(void)
02. {
03.       timer0_init(); //初始化定时器 0
04.        gpio_init();
05.       ppi_init();       //PPI 的初始化
06.       PWM_OUT(10);
07.       uint32_t err_code;
08.       const app_uart_comm_params_t comm_params =
09.         {
10.             RX_PIN_NUMBER,
11.             TX_PIN_NUMBER,
12.             RTS_PIN_NUMBER,
13.             CTS_PIN_NUMBER,
14.             APP_UART_FLOW_CONTROL_ENABLED,
15.             false,
16.             UART_BAUDRATE_BAUDRATE_Baud38400
17.         };
18.       APP_UART_FIFO_INIT(&comm_params,
19.                          UART_RX_BUF_SIZE,
20.                          UART_TX_BUF_SIZE,
21.                          uart_error_handle,
22.                          APP_IRQ_PRIORITY_LOW,
23.                          err_code);
```

```
24.         APP_ERROR_CHECK(err_code);
25.         // 开功耗
26.         NRF_POWER->TASKS_CONSTLAT = 1;
27.         // 开始定时器
28.         nrf_drv_timer_enable(&timer0);
29.         //定时器计数器捕获的值输出
30.         while (true)
31.         {
32.             printf("Current cout: %d\n\r",
    (int)nrf_drv_timer_capture(&timer0,NRF_TIMER_CC_CHANNEL2));
33.             nrf_delay_ms(1000);//延迟 1s
34.         }
35. }
```

编译程序后下载到青风 nrf52832 开发板内。同时把 IO 口 P0.02 和 P0.03 用杜邦线短接。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。由于 while 循环中设置了 1s 的延迟，而脉冲信号为 2s 一个周期，所以捕获值要用串口输出两次才发生一个变化。如下图所示：