第十二章 PPI 模块的使用

nRF52832 是 cortex m4 内核，内部设置了 PPI 方式，PPI 和 DMA 功能有些类似，也是用于不同外设之间进行互连，而不需要 CPU 进行参与。PPI 主要的连接对象是任务和事件。下面将详细进行讨论：

12.1 原理分析

12.1 PPI 的结构

PPI 称为可编程外设接口，PPI 可以使不同外设自动通过任务和事件来连接，不需要 CPU 参与，可以有效的降低功耗，提高处理器处理效率。

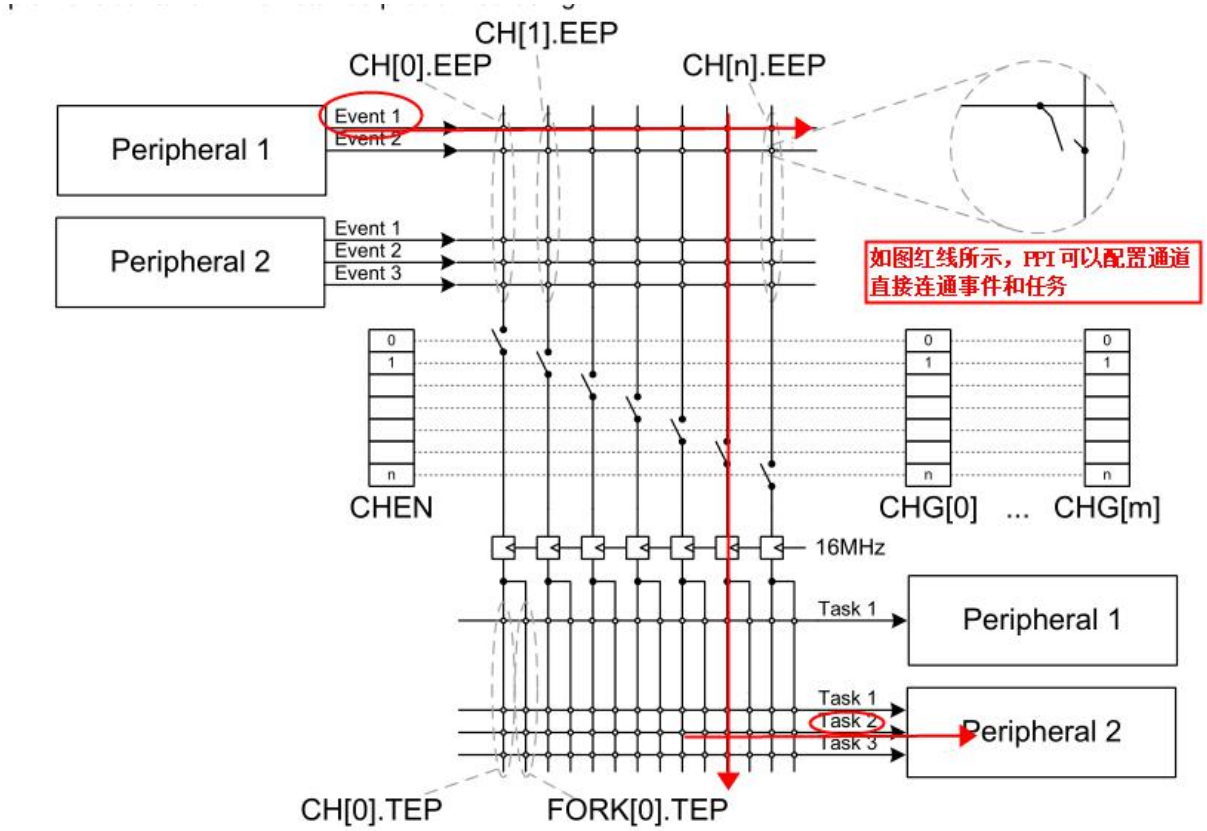
PPI 的两端一端连接的是事件端点（EEP），一端连接的是任务端点（TEP)。因此 PPI 可以通过一个外设上发生的事件自动的触发另一个外设上的任务。首先外设事件需要通过与事件相关的寄存器地址连接到一个事件端点（EEP），另一端外设任务事件也需要通过此任务相关的任务寄存器地址连接到一个任务端点（TEP），当两端连接好后就可以通过 PPI 自动触发了。

连通事件端点（EEP)和任务端点（TEP)的称为 PPI 通道。一般有两种方法打开和关闭 PPI 通道：

1：通过 Groups 的 ENABLE 和 DISABLE 任务来使能或者关闭 PPI 通道组中的 PPI 通道。在触发这些任务前，必须配置哪些 PPI 通道属于哪个 PPI 组。

2：使用 CHEN,CHENSET 和 CHENSET 和 CHENCLR 通道单独打开或关掉 PPI 通道；

PPI 任务（比如 CHG0EN）可以像其他任务一样被 PPI 触发，也就是说 PPI 任务可以作为 PPI 通道上的一个 TEP。一个事件可使用多个通道来触发多个任务。同样的，一个任务可以被多个事件触发。



除了完全可编程的外围互连之外，PPI 系统还具有一组通道，其中事件终点（EEP）和任务终点（TEP）在硬件中固定。 这些固定信道可以与普通 PPI 信道相同的方式单独启用，禁用或添加到 PPI 信道组。

Instance	Channel	Number of channels	Number of groups
PPI	0-19	20	6
PPI (fixed)	20-31	12	

下表中为预编程通道，一些 PPI 通道是预编程的。这些通道不能通过 CPU 进行配置，但是可以添加到组（Groups），可以像其他通用的 PPI 通道一样使能打开或者关闭。

Channel	EEP	TEP
20	TIMER0->EVENTS_COMPARE[0]	RADIO->TASKS_TXEN
21	TIMER0->EVENTS_COMPARE[0]	RADIO->TASKS_RXEN
22	TIMER0->EVENTS_COMPARE[1]	RADIO->TASKS_DISABLE
23	RADIO->EVENTS_BCMATCH	AAR->TASKS_START
24	RADIO->EVENTS_READY	CCM->TASKS_KSGEN
25	RADIO->EVENTS_ADDRESS	CCM->TASKS_CRYPT
26	RADIO->EVENTS_ADDRESS	TIMER0->TASKS_CAPTURE[1]
27	RADIO->EVENTS_END	TIMER0->TASKS_CAPTURE[2]
28	RTC0->EVENTS_COMPARE[0]	RADIO->TASKS_TXEN
29	RTC0->EVENTS_COMPARE[0]	RADIO->TASKS_RXEN
30	RTC0->EVENTS_COMPARE[0]	TIMER0->TASKS_CLEAR
31	RTC0->EVENTS_COMPARE[0]	TIMER0->TASKS_START

在每个 PPI 通道上，信号与 16 MHz 时钟同步，以避免任何内部违反设置和保持时序。 因此，与 16 MHz 时钟同步的事件将延迟一个时钟周期，而其他异步事件将延迟最多一个 16 MHz 时钟周期。请注意，快捷方式（在每个外设中的 SHORTS 寄存器中定义）不受此 16 MHz 同步的影响，因此不会延迟。

12.2 fork 从任务机制：

fork 机制也称为从任务机制。每个 TEP 都实现了一个 fork 机制，可以在触发 TEP 中指定的任务的同时触发第二个任务。第二个任务配置在 FORK 寄存器组的任务端点寄存器中，例如 FORK.TEP [0]与 PPI 通道 CH [0]相关联。

12.3 Group 分组机制：

PPI 通道可以进行分组，多个 PPI 通道可以分为一组，那么该组内的 PPI 通道就可以统一进行管理，同时打开或者关闭 group 内所有的 PPI 通道。

当一个通道属于两个组 m 和 n，并且 CHG [m] .EN 和 CHG [n] .DIS 同时发生时（m 和 n 可以相等或不同），该通道上的 EN 具有优先权。PPI 任务（例如，CHG [0] .EN）可以像任何其他任务一样通过 PPI 触发，这意味着它们可以作为 TEP 连接到 PPI 通道。一个事件可以通过使用多个通道触发多个任务，并且一个任务可以以相同的方式由多个事件触发。

关于 PPI 的基本原理就讲到这个位置，为了深入理解 PPI 的应用，下面会通过一个实例进行分析。

12.2 PPI 之 GPIOTE 应用

12.2.1 寄存器编写

PPI 作为触发通道，两端分别连接任务和事件，通过任务来触发事件的发生，可以不通过 CPU 进行处理，大大的节省了系统资源。本例我们来演示 GPIOTE 的 PPI 应用。通过 GPIOTE 事件来触发 GPIOTE 任务。

首先来配置 GPIOTE 的任务和事件。GPIOTE 的任务和事件在前面的 GPIOTE 的章节中详细讲述过。本实验需要把按键 1 绑定到 GPIOTE 通道 0 上作为事件，把 LED1 灯绑定到 GPIOTE 通道 1 上作任务。具体代码如下所示：

```
01. /**
02.  * 初始 GPIO 端口，设置 PIN_IN 为输入管教, PIN_OUT 为输出管脚,
03.  */
04. static void gpiote_init(void)
05. {
06.   nrf_gpio_cfg_input(BSP_BUTTON_0,NRF_GPIO_PIN_PULLUP);//设置管脚位上拉输入
07.
08.  NRF_GPIOTE->CONFIG[0] =
09.  //绑定 GPIOTE 通道 0，极性为高电平到低电平
10.  (GPIOTE_CONFIG_POLARITY_HiToLo << GPIOTE_CONFIG_POLARITY_Pos)
11. | (BSP_BUTTON_0<< GPIOTE_CONFIG_PSEL_Pos)  // 配置任务输入状态
12. | (GPIOTE_CONFIG_MODE_Event << GPIOTE_CONFIG_MODE_Pos);//事件模式
13.
14.       NRF_GPIOTE->CONFIG[1] =
```

```
15. //绑定 GPIOTE 通道 1，极性为翻转
16.  (GPIOTE_CONFIG_POLARITY_Toggle << GPIOTE_CONFIG_POLARITY_Pos)
17.  | (BSP_LED_0 << GPIOTE_CONFIG_PSEL_Pos) //  配置任务输出状态
18.  | (GPIOTE_CONFIG_MODE_Task << GPIOTE_CONFIG_MODE_Pos);//任务模式
19.  }
```

配置完了 GPIOTE 任务和事件，再来设置 PPI 的端点。PPI 的 CH[n].EEP 寄存器作为通道 n 的事件终点，接到 GPIOTE 的输入事件上。CH[n].TEP 寄存器作为通道 n 的任务终点，接到 GPIOTE 的输出任务上。然后使能 PPI 的通道，通过配置 CHEN 寄存器实现。这时候，当发送了 GPIOTE 的输入事件，也就是按键按下后，会触发另一端的输出任务，输出的任务为翻转电平，会实现 LED 灯的亮灭控制。具体代码如下所示：

```
01. nrf_ppi_channel_t my_ppi_channel;
02. void ppi_init(void)
03. {
04.       // 配置 PPI 的端口，通道 0 一端接按键任务，另外一端接输出事件
05.       NRF_PPI->CH[0].EEP = (uint32_t)(&NRF_GPIOTE->EVENTS_IN[0]);
06.       NRF_PPI->CH[0].TEP = (uint32_t)(&NRF_GPIOTE->TASKS_OUT[1]);
07.
08.       // 使能 PPI 的通道 0
09.       NRF_PPI->CHEN = (PPI_CHEN_CH0_Enabled << PPI_CHEN_CH0_Pos);
10. }
```

最后，在主函数中，调用 PPI 配置函数和 GPIOTE 配置函数后，CPU 就也什么都不做，一直等待循环。所有的操作都交给了 PPI 通道执行。

```
01. /**
02.  * 主函数，配置 PPI 的通道
03.  */
04. int main(void)
05. {
06.      gpiote_init();
07.       ppi_init();
08.       while (true)
09.       {
10.            // 循环等待
11.       }
12. }
```

把该例子程序编译后下载到青风 nrf52832 开发板内。按下按键 1，可以使得 LED1 灯进行翻转。

12.2.2 PPI 组件库的应用

SDK 的库函数内，提供了 PPI 的编程组件库，本例将通过 PPI 的库函数 API 来实现一个 GPIOTE 的应用。

12.2.2.1：库函数介绍：

GPIOTE 的应用中，PPI 的编程组件库函数主要是使用如下几个函数，这些函数可以方便的配置 PPI 的应用。

1：函数 nrf_drv_ppi_init，该函数主要是用于初始化 PPI 模块，判断 PPI 当前的状态。

函数 ret_code_t nrf_drv_ppi_init(void);
*功能：函数功能是用于初始化 PPI 模块
*返回值： NRF_SUCCESS 如果模块被成功初始化 *返回值： NRF_ERROR_MODULE_ALREADY_INITIALIZED 如果模块已经被初始化

2.函数 nrfx_ppi_channel_alloc，该函数主要用于分配未使用的 PPI 通道。

函数 nrfx_err_t nrfx_ppi_channel_alloc(nrf_ppi_channel_t * p_channel);
*功能：函数功能是用于分配第一个未使用的 PPI 通道。
*参数[输出]： p_channel 指向已分配的 PPI 通道的指针。
返回值：NRFX_SUCCESS 如果成功分配了通道。 返回值：NRFX_ERROR_NO_MEM 如果没有可用的通道可用。

3：函数 nrfx_ppi_channel_enable，该函数用于使能 PPI 通道，开启 PPI。

函数 nrfx_err_t nrfx_ppi_channel_enable(nrf_ppi_channel_t channel);
*功能：函数功能是启用 PPI 通道的功能
*参数[输入]： channel 启用的 PPI 通道
返回值：NRFX_SUCCESS 如果通道已成功启用。 返回值：NRFX_ERROR_INVALID_STATE 如果没有分配用户可配置的通道。 返回值：NRFX_ERROR_INVALID_PARAM 如果用户无法启用通道。

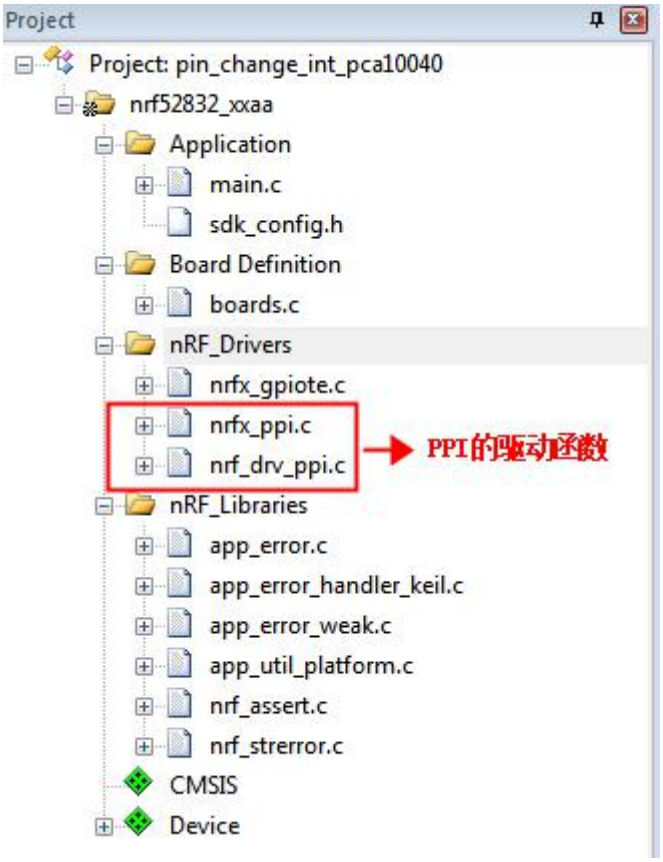
4：函数 nrfx_ppi_channel_assign，该函数主要用于分配 EEP 事件终点和 TEP 任务终点：

函数 nrfx_err_t nrfx_ppi_channel_assign(nrf_ppi_channel_t channel, uint32_t eep, uint32_t tep);
*功能：函数功能是启用 PPI 通道的功能
*参数[输入]： channel 要分配端点的 PPI 通道 *参数[输入]： eep 事件的端点地址。 *参数[输入]： tep 任务的端点地址。
返回值：NRFX_SUCCESS 如果成功分配了信道。 返回值：NRFX_ERROR_INVALID_STATE 如果没有为用户分配通道。 返回值：NRFX_ERROR_INVALID_PARAM 如果用户无法配置通道。

下面的工程实例中将采用以上的 PPI 函数进行配置编程。

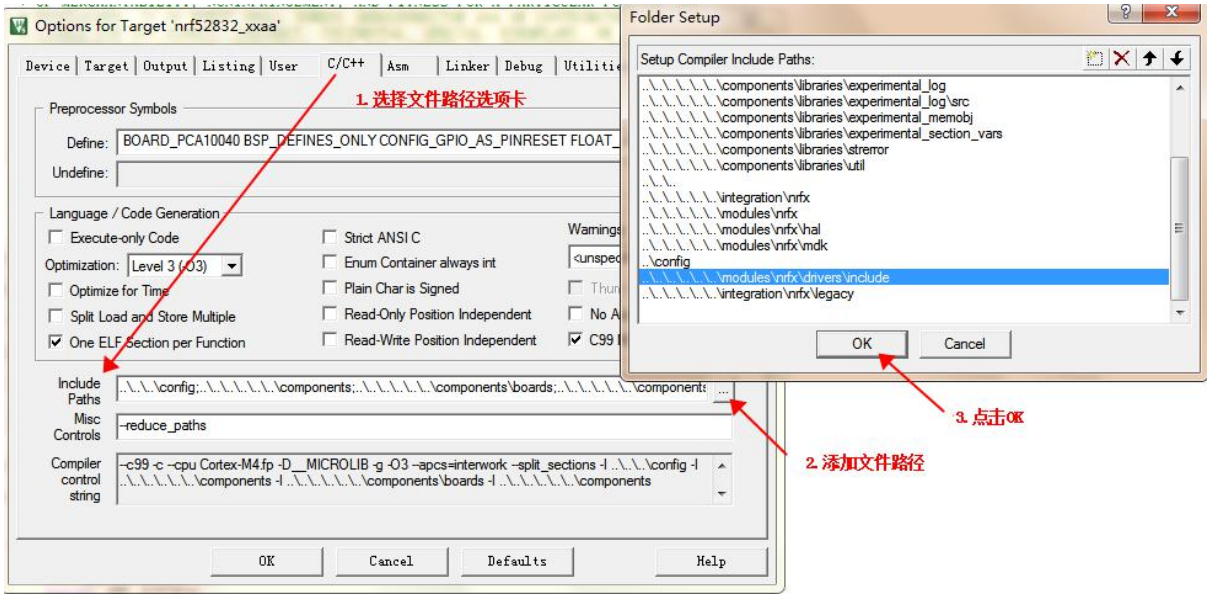
12.2.2.2：工程代码的编写：

PPI 的 GPIOTE 应用工程的可以使用前面 GPIOTE 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:

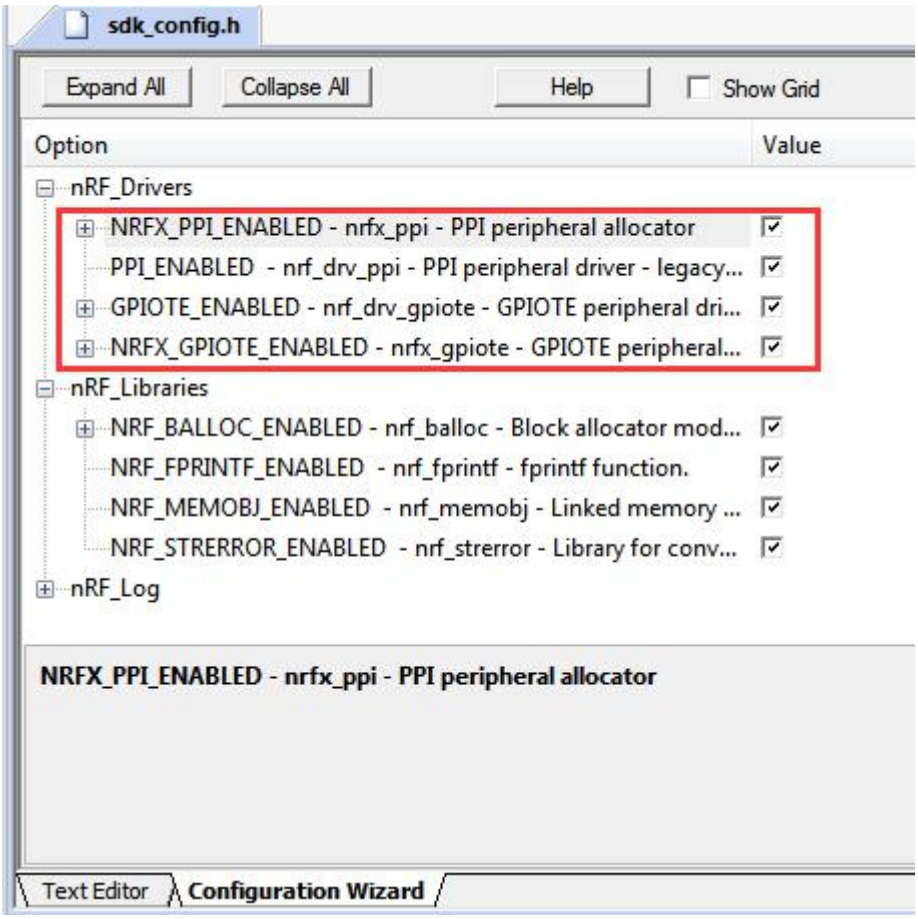


主函数 main.c 文件，sdk_config.h 配置文件这两个文件需要我们编写和修改的。而 nrfx_ppi.c 文件和 nrf_drv_ppi 则是需要我们添加的库文件。nrfx_ppi.c 文件的路径在 SDK 的 //modules/nrfx/drivers/include 文件夹里，nrf_drv_ppi 文件的路径在 SDK 的\\integration\nrfx\legacy 文件夹内。

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：



同时在主函数 main.c 中，需要调用头文件#include "nrf_drv_gpiote.h"和#include "nrf_drv_ppi.h"。

工程搭建完毕后，首先来编写 GPIOTE 的初始化函数，初始化 GPIOTE 端口设置 PIN_IN 为输入任务管脚, PIN_OUT 为输出任务管。这个方面的相关配置在 GPIOTE 的章节教学中详细讲解过，这里不再累述，具体代码如下所示：

```
/** 初始 GPIOTE 端口，设置 PIN_IN 为输入管教, PIN_OUT 为输出管脚,*/
static void gpiote_init(void)
{
    ret_code_t err_code;
    //初始化 GPIOTE
    err_code = nrf_drv_gpiote_init();
    APP_ERROR_CHECK(err_code);
    //配置输出为翻转电平
    nrf_drv_gpiote_out_config_t out_config =  GPIOTE_CONFIG_OUT_TASK_TOGGLE(true);
    //绑定输出端口
    err_code = nrf_drv_gpiote_out_init(PIN_OUT, &out_config);
    APP_ERROR_CHECK(err_code);
   //配置为输出任务模式使能
   nrf_drv_gpiote_out_task_enable(PIN_OUT);

    //配置输入为高电平到低电平
    nrf_drv_gpiote_in_config_t in_config = GPIOTE_CONFIG_IN_SENSE_HITOLO (true);
    in_config.pull = NRF_GPIO_PIN_PULLUP;
     //绑定输入端口
    err_code = nrf_drv_gpiote_in_init(PIN_IN, &in_config, NULL);
    APP_ERROR_CHECK(err_code);
     //配置输入事件使能
    nrf_drv_gpiote_in_event_enable(PIN_IN, true);
}
```

下面我们具体来讨论下如何使用库函数来配置 PPI 的功能，其代码具体如下所示：

```
void ppi_init(void)
{
  ret_code_t err_code;
   //初始化 PPI 的模块
  err_code = nrf_drv_ppi_init();
  APP_ERROR_CHECK(err_code);

  //配置 PPI 的频道
  err_code = nrfx_ppi_channel_alloc(&my_ppi_channel);
  APP_ERROR_CHECK(err_code);

  //设置 PPI 通道 my_ppi_channel 的 EEP 和 TEP 两端对应的硬件
  err_code = nrfx_ppi_channel_assign(my_ppi_channel,
                                     nrfx_gpiote_in_event_addr_get(PIN_IN),
                                     nrfx_gpiote_out_task_addr_get(PIN_OUT));
  APP_ERROR_CHECK(err_code);
```

```
17.    //使能 PPI 通道
18.    err_code = nrfx_ppi_channel_enable(my_ppi_channel);
19.    APP_ERROR_CHECK(err_code);
```

第 5 行：初始化 PPI 模块，调用 nrf_drv_ppi_init 函数。

第 9 行：配置 PPI 的频道，调用 nrfx_ppi_channel_alloc 函数，设置一个&my_ppi_channel 的为 nrf_ppi_channel_t 类型的宏，指向没有被使用的 PPI 通道。

第 13 行： 设置 PPI 通道 my_ppi_channel 的 EEP 事件终点和 TEP 任务终点两端对应的硬件地址。

对应事件地址：调用 API 函数 nrfx_gpiote_in_event_addr_get(PIN_IN)来实现获取，并绑定到 PPI 的 EEP 事件终点上。对应任务地址：调用 API 函数 nrfx_gpiote_out_task_addr_get(PIN_OUT)来实现获取，并绑定到 PPI 的 TEP 任务终点上。

第 18 行：使能 PPI 通道，调用函数 nrfx_ppi_channel_enable 对之前申请的 PPI 通道 my_ppi_channel 进行使能。

最后，在主函数中，调用 PPI 配置函数和 GPIOTE 配置函数后，CPU 就也什么都不做，一直等待循环。所有的操作都交给了 PPI 通道执行。

```
01. }
02. /**
03.  * 主函数，配置 PPI 的通道
04.  */
05. int main(void)
06. {
07.     gpiote_init();
08.     ppi_init();
09.     while (true)
10.     {
11.         // Do nothing.
12.     }
13. }
```

把该例子程序编译后下载到青风 nrf52832 开发板内。按下按键 1，可以使得 LED1 灯进行翻转。

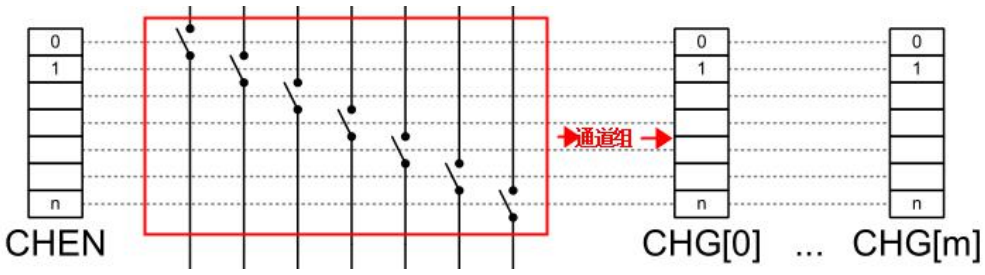
12.3 PPI 之 gruop 分组和 fork 从任务应用

12.3.1 gruop 分组应用

12.3.1.1 PPI group 分组原理及应用

1：基本原理

PPI 通道可以进行分组，多个 PPI 通道可以分为一组。PPI 具有 6 个 group 组，每个组都可以包含多个 PPI 通道。如下图所示：



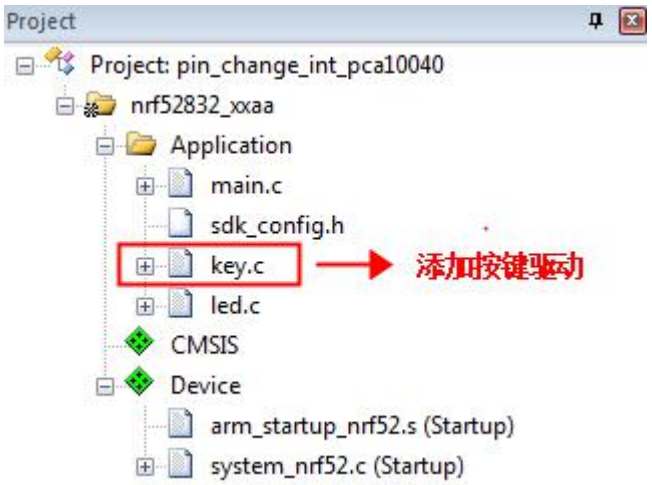
图中，m=5。一共 6 个组，每个组都可以把 0~n 的 PPI 通道包含到其中。那么包含到一个 group 组内的 PPI 通道就可以进行统一的管理与操作。比如打开或者关掉 PPI 通道。通过寄存器 CHG [n] .EN 和 CHG [n] .DIS 来打开或者关掉所有包含在组里的 PPI 通道。

PPI 组事件 CHG [0] .EN 也可以像任何其他任务一样通过 PPI 触发，这意味着它们可以作为 TEP 连接到 PPI 通道。可以通过 PPI 事件来管理一个 group 组。

2：按键扫描方式管理 group 组

本节通过两种方式对探讨对 group 组进行管理。第一种方式验证按键扫描方式来使能 group 组和禁止 group 组的方式。第二种方式采用 PPI 通道触发组事件 CHG [0] .EN 来使能 group 组和禁止 group 组。

代码工程搭建如下图所示，由于采用寄存器方式编写工程比较简单，只需要加入前面章节编写的按键的驱动函数就可以：



主函数配置代码如下所示，首先配置 GPIOTE 事件和任务，由于要使用到 PPI 组，这里我们配置两个 GPIOTE 事件和任务。按键 1 和按键 2 分别配置为 GPIOTE 输入事件，LED1 和 LED2 则配置为 GPIOTE 输出任务。具体配置如下所示：

```
/**
 * 初始 GPIO 端口，设置 PIN_IN 为输入管教, PIN_OUT 为输出管脚,
 */
static void gpiote_init(void)
{
  nrf_gpio_cfg_input(KEY_1,NRF_GPIO_PIN_PULLUP);//设置管脚位上拉输入
  nrf_gpio_cfg_input(KEY_2,NRF_GPIO_PIN_PULLUP);//设置管脚位上拉输入
  //////////////////////////////////////////////
      NRF_GPIOTE->CONFIG[0] =
```

```
10.  //绑定 GPIOTE 通道 0，输入电平高到低
11.  (GPIOTE_CONFIG_POLARITY_HiToLo << GPIOTE_CONFIG_POLARITY_Pos)
12.  | (KEY_1 << GPIOTE_CONFIG_PSEL_Pos)  // 配置任务输入状态
13.  | (GPIOTE_CONFIG_MODE_Event << GPIOTE_CONFIG_MODE_Pos);//事件模式
14.
15.  NRF_GPIOTE->CONFIG[1] =
16.  //绑定 GPIOTE 通道 1，输出电平翻转
17.  (GPIOTE_CONFIG_POLARITY_Toggle << GPIOTE_CONFIG_POLARITY_Pos)
18.  | (LED_0 << GPIOTE_CONFIG_PSEL_Pos) // 配置任务输出状态
19.  | (GPIOTE_CONFIG_MODE_Task << GPIOTE_CONFIG_MODE_Pos);//任务模式
20.
21.     ////////////////////////////////////////////
22.       NRF_GPIOTE->CONFIG[2] =
23.    //绑定 GPIOTE 通道 2，输入电平高到低
24.  (GPIOTE_CONFIG_POLARITY_HiToLo << GPIOTE_CONFIG_POLARITY_Pos)
     | (KEY_2   << GPIOTE_CONFIG_PSEL_Pos)    // 配置任务输入状态
25.  | (GPIOTE_CONFIG_MODE_Event << GPIOTE_CONFIG_MODE_Pos);//事件模式
26.
27.  NRF_GPIOTE->CONFIG[3] =
28.  //绑定 GPIOTE 通道 3，输出电平翻转
29.  (GPIOTE_CONFIG_POLARITY_Toggle << GPIOTE_CONFIG_POLARITY_Pos)
30.  | (LED_1<< GPIOTE_CONFIG_PSEL_Pos) // 配置任务输出状态
31.  | (GPIOTE_CONFIG_MODE_Task << GPIOTE_CONFIG_MODE_Pos);//任务模式
32. }
```

再来分配 PPI 的组，本例使用两个 PPI 通道，分别为 PPI 通道 0 和 PPI 通道 1。PPI 通道 0 一端 EEP 事件终点接 GPIOTE 事件 0，另一端 TEP 任务终点接 GPIOTE 任务 1。PPI 通道 1 一端 EEP 事件终点接 GPIOTE 事件 2，另一端 TEP 任务终点接 GPIOTE 任务 3。最后把 PPI 通道 0 和 PPI 通道 1 同时配置到 group0 上去，通过 CHG[0]赋值 0x03，绑定 PPI 通道 0 和 PPI 通道 1 到 group0 组。具体代码如下所示：

```
01. void ppi_init(void)
02. {
03.      // 配置 PPI 通道 0，一端接 GPIOTE 事件 0，一端接 GPIOTE 任务 1
04.      NRF_PPI->CH[0].EEP = (uint32_t)(&NRF_GPIOTE->EVENTS_IN[0]);
05.      NRF_PPI->CH[0].TEP = (uint32_t)(&NRF_GPIOTE->TASKS_OUT[1]);
06.    // 配置 PPI 通道 1，一端接 GPIOTE 事件 2，一端接 GPIOTE 任务 3
07.    NRF_PPI->CH[1].EEP = (uint32_t)(&NRF_GPIOTE->EVENTS_IN[2]);
08.      NRF_PPI->CH[1].TEP = (uint32_t)(&NRF_GPIOTE->TASKS_OUT[3]);
09.      //把通道 0 和通道 1  绑定到 PPI group0 之上
10.    NRF_PPI->CHG[0]=0x03;
11. }
```

如果采用按键扫描的方式对 PPI 组进行管理，则需要在主函数中，通过判断按键是否按下来使能组或者禁止组。因此参考按键扫描的方式，在按键 3 按下后，使能 PPI 组；在按键 4 按下后关闭 PPI 组，具体代码如需所示：

```
01. /**主函数，配置 PPI 的组使能或者关闭*/
```

```
02. int main(void)
03. {
04.     gpiote_init();//初始化 GPIOTE
05.     ppi_init();//初始化 PPI
06.     KEY_Init();//按键初始化
07.     LED_Init();//led 灯初始化
08.     LED3_Close();
09.     LED4_Close();
10.       while (true)
11.       {
12.             if( KEY3_Down()== 0)//判定按键是否按下
13.  {
14.        LED4_Close();
15.          NRF_PPI->TASKS_CHG[0].EN = 1;//使能 PPI group0
16.        LED3_Toggle();
17.  }
18.              if( KEY4_Down()== 0)//判定按键是否按下
19.  {      LED3_Close();
20.        NRF_PPI->TASKS_CHG[0].EN = 0;//关闭 PPI group0
21.        LED4_Toggle();
22.       }
23. }
24.       }
```

把该例子程序编译后下载到青风 nrf52832 开发板内。默认 PPI group 组是关闭的，此时按下按键 1 或者按键 2，LED1 灯或者 LED2 灯不会发生变化。如果按下按键 3，使能了 PPI group 组，此时再按下按键 1，可以使得 LED1 灯进行翻转；按下按键 2，可以使得 LED2 灯进行翻转。

我们如果想关闭 PPI group 组，按下按键 4 则可以实现。

3：事件 CHG [0] .EN 或者 CHG [0] .DIS 管理 group 组

事件 CHG [0] .EN 或者 CHG [0] .DIS 都可以作为 PPI 的 TEP 任务终点，因此可以采用任何事件作为 PPI 的 EEP 事件终点来触发 group 组的管理。下面我们采用一个简单的 GPIOTE 按键输入事件来触发 group 组的使能。具体代码如下所示，在 PPI 初始化中，增加一个 PPI 通道 2。该通道的 EEP 端接按键 0 的 GPIOTE 输入事件，另外一个 TEP 端接 CHG [0] .EN 通道使能任务。最后需要单独使能 PPI 通道 2。

```
01. void ppi_init(void)
02. {    //配置按键 1 作为触发事件，连接 PPI 通道 2 的一端。
03.     NRF_PPI->CH[2].EEP = (uint32_t)(&NRF_GPIOTE->EVENTS_IN[0]);
04.     //触发 group 组的使能
05.     NRF_PPI->CH[2].TEP = (uint32_t)(&NRF_PPI->TASKS_CHG[0].EN);
06.   // 使能 PPI 的通道 2
07.     NRF_PPI->CHEN = (PPI_CHEN_CH2_Enabled << PPI_CHEN_CH2_Pos);
08.
09.   // 配置 PPI 通道 0，一端接 GPIOTE 事件 0，一端接 GPIOTE 任务 1
10.       NRF_PPI->CH[0].EEP = (uint32_t)(&NRF_GPIOTE->EVENTS_IN[0]);
11.       NRF_PPI->CH[0].TEP = (uint32_t)(&NRF_GPIOTE->TASKS_OUT[1]);
```

```
12.     // 配置 PPI 通道 1，一端接 GPIOTE 事件 2，一端接 GPIOTE 任务 3
13.     NRF_PPI->CH[1].EEP = (uint32_t)(&NRF_GPIOTE->EVENTS_IN[2]);
14.       NRF_PPI->CH[1].TEP = (uint32_t)(&NRF_GPIOTE->TASKS_OUT[1]);
15.       //把通道 0 和通道 1  绑定到 PPI group0 之上
16.     NRF_PPI->CHG[0]=0x03;
17. }
```

这种状态下，如果第一次按键 1 没有按下，按键 1 和按键 2 都不能触发 LED1 灯和 LED2 灯的翻转。只有当按键 1 按下后，使能了 group 组 0，按键 1 和按键 2 才能通过 PPI 触发分别 LED1 灯和 LED2 灯的翻转。那么主函数里就可以什么都不做，把相关的操作交给 PPI 来实现。主函数代码如下所示：

```
18. /**
19.   * 主函数，配置 PPI 的通道
20.   */
21. int main(void)
22. {
23.     gpiote_init();
24.       ppi_init();
25.     KEY_Init();
26.       while (true)
27.       {
28.       }
29. }
```

把该例子程序编译后下载到青风 nrf52832 开发板内。默认 PPI group 组是关闭的，此时按下按按键 2，LED2 灯不会发生变化。如果按下按键 1，使能了 PPI group 组，此时再按下按键 1，可以使得 LED1 灯进行翻转；按下按键 2，可以使得 LED2 灯进行翻转。

12.3.1.2 PPI 组件库实现 PPI group 分组

1：函数 nrfx_ppi_channel_include_in_group：这个函数把指定的 PPI 通道包含到通道组中。

函数：__STATIC_INLINE nrfx_err_t nrfx_ppi_channel_include_in_group(nrf_ppi_channel_t channel, nrf_ppi_channel_group_t group)
{
return nrfx_ppi_channels_include_in_group(nrfx_ppi_channel_to_mask(channel), group);
}

*功能：在通道组中包含 PPI 通道的函数
*参数[输入]： channel PPI 要添加的通道。 *参数[输入]： group 包含该通道的通道组。
返回值：NRFX_SUCCESS 如果成功包含通道 返回值：NRFX_ERROR_INVALID_STATE 如果组不是已分配的组。 返回值：NRFX_ERROR_INVALID_PARAM 如果组不是应用程序组或通道不是应用程序通道。

2:函数 nrfx_ppi_group_alloc 用于分配一个未被使用的 PPI 组，并且分配对应的 PPI 通道组指针。

函数：nrfx_err_t nrfx_ppi_group_alloc(nrf_ppi_channel_group_t * p_group);
*功能：用于分配 PPI 通道组的函数。该函数分配第一个未使用的 PPI 组。
*参数[输入]：p_group 指向已分配的 PPI 通道组的指针。
返回值：NRFX_SUCCESS 如果成功分配了通道组。 返回值：NRFX_ERROR_NO_MEM 如果没有可用的通道组要使用。

3：函数 nrfx_ppi_group_enable 用于使能 PPI 通道组，可以统一的打开包含的 PPI 通道。

函数：nrfx_err_t nrfx_ppi_group_enable(nrf_ppi_channel_group_t group);
*功能：启用 PPI 通道组的功能。
*参数[输入]： group 要启用的通道组。
返回值：NRFX_SUCCESS 如果成功启用了组。 返回值：NRFX_ERROR_INVALID_STATE 如果组不是已分配的组。 返回值：NRFX_ERROR_INVALID_PARAM 如果组不是应用程序组。

4：函数 nrfx_ppi_group_disable 用于关闭 PPI 通道组，可以统一的禁止包含的 PPI 通道。

函数：nrfx_err_t nrfx_ppi_group_disable(nrf_ppi_channel_group_t group);
*功能：用于禁用 PPI 通道组的函数。
*参数[输入]： group 要禁止的通道组。
返回值：NRFX_SUCCESS 如果成功禁止了组。 返回值：NRFX_ERROR_INVALID_STATE 如果组不是已分配的组。 返回值：NRFX_ERROR_INVALID_PARAM 如果组不是应用程序组。

库函数的工程采用上一节的 PPI 的 GPIOTE 应用的库函数工程，sdk_config.h 的配置相同。下面来分析下代码，具体代码如下所示：

首先配置 2 个 GPIOTE 输入任务和 2 个 GPIOTE 输出事件。两个输入任务分别接按键 1 和按键 2，两个输出事件分别接 LED1 和 LED2 灯，关于 GPIOTE 的配置这里就不再累述了。

```
01. nrf_ppi_channel_t my_ppi_channel1;
02. nrf_ppi_channel_t my_ppi_channel2;
03. nrf_ppi_channel_group_t qf_ppi_group;
04. /**
05.  * 初始 GPIO 端口，设置 PIN_IN 为输入管教, PIN_OUT 为输出管脚,
06.  */
07. static void gpiote_init(void)
08. {
09.     ret_code_t err_code;
```

```
10.
11.         err_code = nrf_drv_gpiote_init();
12.         APP_ERROR_CHECK(err_code);
13.             ////////////////////////////
14.        //配置输出任务 1
15.         nrf_drv_gpiote_out_config_t out_config1 =    GPIOTE_CONFIG_OUT_TASK_TOGGLE(true);
16.         err_code = nrf_drv_gpiote_out_init(LED_1, &out_config1);
17.         APP_ERROR_CHECK(err_code);
18.          nrf_drv_gpiote_out_task_enable(LED_1);
19.
20.        //配置输出任务 2
21.         nrf_drv_gpiote_out_config_t out_config2 =    GPIOTE_CONFIG_OUT_TASK_TOGGLE(true);
22.         err_code = nrf_drv_gpiote_out_init(LED_2, &out_config2);
23.         APP_ERROR_CHECK(err_code);
24.        nrf_drv_gpiote_out_task_enable(LED_2);
25.
26.        //配置输入事件 3
27.         nrf_drv_gpiote_in_config_t in_config1 = GPIOTE_CONFIG_IN_SENSE_HITOLO (true);
28.         in_config1.pull = NRF_GPIO_PIN_PULLUP;
29.         err_code = nrf_drv_gpiote_in_init(BSP_BUTTON_0, &in_config1, NULL);
30.         APP_ERROR_CHECK(err_code);
31.         nrf_drv_gpiote_in_event_enable(BSP_BUTTON_0, true);
32.
33.        //配置输入事件 4
34.        nrf_drv_gpiote_in_config_t in_config2 = GPIOTE_CONFIG_IN_SENSE_HITOLO (true);
35.         in_config2.pull = NRF_GPIO_PIN_PULLUP;
36.         err_code = nrf_drv_gpiote_in_init(BSP_BUTTON_1, &in_config2, NULL);
37.         APP_ERROR_CHECK(err_code);
38.         nrf_drv_gpiote_in_event_enable(BSP_BUTTON_1, true);
39. }
```

这里主要探讨下使用库函数如何把 PPI 通道包含到通道 group 组中，在 PPI 初始化函数配置 PPI 通道 group 组，具体代码如下所示：

```
01. void ppi_init(void)
02. {
03.   ret_code_t err_code;
04.
05.     //初始化 PPI 的模块
06.    err_code = nrf_drv_ppi_init();
07.    APP_ERROR_CHECK(err_code);
08.
09.    //配置 PPI 的频道
10.    err_code = nrfx_ppi_channel_alloc(&my_ppi_channel1);
11.    APP_ERROR_CHECK(err_code);
12.    //设置 PPI 通道 my_ppi_channe11 的 EEP 和 TEP  两端对应  输出任务 1 和输入事件 3
```

```
13.     err_code = nrfx_ppi_channel_assign(my_ppi_channel1,
14.                                        nrfx_gpiote_in_event_addr_get(BSP_BUTTON_0),
15.                                        nrfx_gpiote_out_task_addr_get(LED_1));
16.     APP_ERROR_CHECK(err_code);
17.    //配置 PPI 的频道
18.     err_code = nrfx_ppi_channel_alloc(&my_ppi_channel2);
19.     APP_ERROR_CHECK(err_code);
20.     //设置 PPI 通道 my_ppi_channel2 的 EEP 和 TEP 两端对应 输出任务 2 和输入事件 4
21.     err_code = nrfx_ppi_channel_assign(my_ppi_channel2,
22.                                        nrfx_gpiote_in_event_addr_get(BSP_BUTTON_1),
23.                                        nrfx_gpiote_out_task_addr_get(LED_2));
24.     APP_ERROR_CHECK(err_code);
25.
26.     //申请 PPI 组，分配的组号保存到 qf_ppi_group
27.     err_code = nrfx_ppi_group_alloc(&qf_ppi_group);
28.     APP_ERROR_CHECK(err_code);
29.
30.     //PPI 通道 my_ppi_channel1 加入到 PPI 组 my_ppi_group
31.     err_code = nrfx_ppi_channel_include_in_group(my_ppi_channel1,qf_ppi_group);
32.     APP_ERROR_CHECK(err_code);
33.     //PPI 通道 my_ppi_channel2 加入到 PPI 组 qf_ppi_group
34.     err_code = nrfx_ppi_channel_include_in_group(my_ppi_channel2,qf_ppi_group);
35.     APP_ERROR_CHECK(err_code);
36. }
```

第 6 行：调用函数 nrf_drv_ppi_init 对 PPI 进行初始化。

第 10~13 行：首先调用函数 nrfx_ppi_channel_alloc 对 PPI 通道 my_ppi_channe11 进行声明，然后调用函数 nrfx_ppi_channel_assign 置 PPI 通道 my_ppi_channe11 的 EEP 和 TEP 两端对应输出任务 1 和输入事件 3。

第 16~21 行：先调用函数 nrfx_ppi_channel_alloc 对 PPI 通道 my_ppi_channe12 进行声明，然后调用函数 nrfx_ppi_channel_assign 置 PPI 通道 my_ppi_channe12 的 EEP 和 TEP 两端对应输出任务 2 和输入事件 4。

第 27 行：调用函数 nrfx_ppi_group_alloc 申请 PPI 组，分配的组号保存并命名为 qf_ppi_group。

第 31~35 行：调用函数 nrfx_ppi_channel_include_in_group 分别把 PPI 通道 my_ppi_channe11 和 PPI 通道 my_ppi_channe12 加入到 PPI 组 qf_ppi_group 中去。

主函数采用按键扫描的方式来管理 PPI 组。当按键 3 按下后，调用函数 nrfx_ppi_group_enable 对 PPI 组进行使能；当按键 4 按下后，调用函数 nrfx_ppi_group_disable 对 PPI 组进行禁止。具体代码如下所示：

```
01. /**
02.  * 主函数，配置 PPI 的通道
03.
04.  */
05. int main(void)
06. {    ret_code_t err_code;
07.     gpiote_init();
```

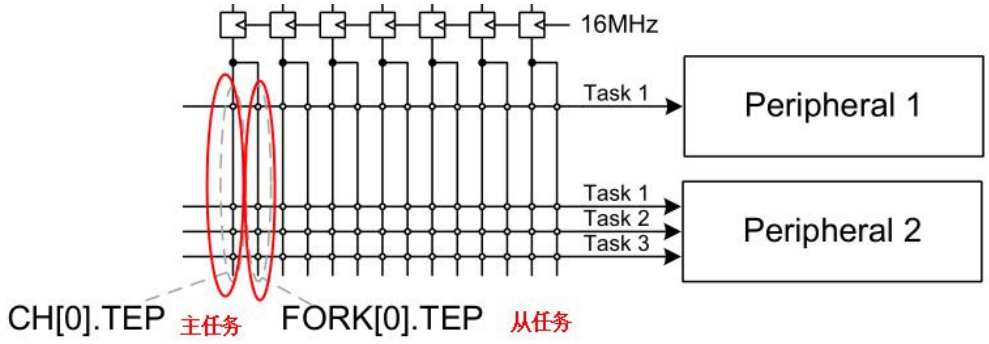
```
    ppi_init();
    qf_led_key_init();
    while (true)
    {
        //检测按键 S3 是否按下
     if(nrf_gpio_pin_read(BUTTON_3) == 0)
     {
        //D3 点亮，D4 熄灭，指示：PPI 组使能
        nrf_gpio_pin_clear(LED_3);
        nrf_gpio_pin_set(LED_4);
        while(nrf_gpio_pin_read(BUTTON_3) == 0){}//等待按键释放
        //使能 PPI 组 my_ppi_group
        err_code = nrfx_ppi_group_enable(qf_ppi_group);
        APP_ERROR_CHECK(err_code);
     }
     //检测按键 S4 是否按下
     if(nrf_gpio_pin_read(BUTTON_4) == 0)
     {
        //D4 点亮，D3 熄灭，指示：PPI 组禁止
        nrf_gpio_pin_clear(LED_4);
        nrf_gpio_pin_set(LED_3);
        while(nrf_gpio_pin_read(BUTTON_4) == 0){}//等待按键释放
        //禁止 PPI 组 my_ppi_group
        err_code = nrfx_ppi_group_disable(qf_ppi_group);
        APP_ERROR_CHECK(err_code);
     }
    }
}
```

把该例子程序编译后下载到青风 nrf52832 开发板内。默认 PPI group 组是关闭的，此时按下按键 1 或者按键 2，LED1 灯或者 LED2 灯不会发生变化。如果按下按键 3，使能了 PPI group 组，此时再按下按键 1，可以使得 LED1 灯进行翻转；按下按键 2，可以使得 LED2 灯进行翻转。如果按下按键 4，则禁止 PPI group 组，此时按下按键 1 或者按键 2，LED1 灯或者 LED2 灯不会发生变化。

12.3.2 fork 从任务应用

12.3.2.1 PPI fork 从任务寄存器应用

fork 机制也称为从任务机制。每个 TEP 都实现了一个 fork 机制，可以在触发 TEP 中指定的任务的同时触发第二个任务。第二个任务配置在 FORK 寄存器组的任务端点寄存器中。如下图所示，因此，我们可以采用一个事件触发两个任务，一个主任务，一个从任务。



如上图所示，当某个事件触发任务 CH[n].TEP 任务的时候，如果配置 FORK[n].TEP 终点连接一个从任务，那么也可以同时别触发。所有编写程序步骤如下所示

1.首先需要配置一个 EEP 事件终点、一个 CH[n].TEP 任务终点和 FORK[n].TEP 从任务终点。本例采用 GPIOTE 按键输入作为事件，两个 GPIOTE 输出作为任务，因此配置代码如下所示：

```
/**
 * 初始 GPIO 端口，设置 PIN_IN 为输入管教, PIN_OUT 为输出管脚,
 */
static void gpiote_init(void)
{
 nrf_gpio_cfg_input(BUTTON_1,NRF_GPIO_PIN_PULLUP);//设置管脚位上拉输入
    //配置一个 GPIOTE 输入任务
  NRF_GPIOTE->CONFIG[0] =
   //绑定通道 0
   (GPIOTE_CONFIG_POLARITY_HiToLo << GPIOTE_CONFIG_POLARITY_Pos)
  | (BUTTON_1<< GPIOTE_CONFIG_PSEL_Pos)  // 配置事件输入
  | (GPIOTE_CONFIG_MODE_Event << GPIOTE_CONFIG_MODE_Pos);//设置实际模式
     //配置一个 GPIOTE 输出
 NRF_GPIOTE->CONFIG[1] =
  //绑定通道 1
 (GPIOTE_CONFIG_POLARITY_Toggle << GPIOTE_CONFIG_POLARITY_Pos)
 | (LED_1 << GPIOTE_CONFIG_PSEL_Pos) // 配置任务输出状态
 | (GPIOTE_CONFIG_MODE_Task << GPIOTE_CONFIG_MODE_Pos);//任务模式
  //配置一个 GPIOTE 输出作为分支端
    NRF_GPIOTE->CONFIG[2] =
  //绑定通道 2
 (GPIOTE_CONFIG_POLARITY_Toggle << GPIOTE_CONFIG_POLARITY_Pos)
 | (LED_2 << GPIOTE_CONFIG_PSEL_Pos) // 配置任务输出状态
 | (GPIOTE_CONFIG_MODE_Task << GPIOTE_CONFIG_MODE_Pos);//任务模式
}
```

PPI 中，除了赋值 CH[n].EEP 终点和 CH[n].TEP 中断外，还需要再赋值一个 FORK[n].TEP 从任务终点地址。赋值完成后，通过寄存器 CHEN 对 PPI 通道进行使能。

```
void ppi_init(void)
{
    // 配置 PPI 一端接输入事件 0，一端接输出任务 1
    NRF_PPI->CH[0].EEP = (uint32_t)(&NRF_GPIOTE->EVENTS_IN[0]);
```

```
05.       NRF_PPI->CH[0].TEP = (uint32_t)(&NRF_GPIOTE->TASKS_OUT[1]);
06.     //输出端接通道 0 的 fork 分支端
07.       NRF_PPI->FORK[0].TEP= (uint32_t)(&NRF_GPIOTE->TASKS_OUT[2]);
08.       // 使能通道 0
09.       NRF_PPI->CHEN = (PPI_CHEN_CH0_Enabled << PPI_CHEN_CH0_Pos);
10. }
```

那么主函数里就可以什么都不做，把相关的操作交给 PPI 来实现。只需要调用 GPIOTE 初始化函数和 PPI 初始化函数，主函数代码如下所示：

```
01. /** 主函数，配置 PPI 的通道*/
02. int main(void)
03. {
04.     gpiote_init();
05.       ppi_init();
06.       while (true)
07.       {
08.             // Do nothing.
09.       }
10. }
```

把该例子程序编译后下载到青风 nrf52832 开发板内。按下按键 1，可以使得 LED1 灯进行翻转，同时 LED2 灯也会进行翻转。

12.3.2.2 PPI 组件库实现 PPI fork 从任务

库函数下 FORK 从任务的编程，需要使用到库函数 API，如下所示：

1：函数 nrfx_ppi_channel_fork_assign：分配从任务 Fork 端点到 PPI 通道之上

函数：nrfx_err_t nrfx_ppi_channel_fork_assign(nrf_ppi_channel_t channel, uint32_t fork_tep);
*功能：用于为 PPI 通道分配或清除 fork 端点。
*参数[输入]： channel 要分配端点的 PPI 通道。 *参数[输入]： fork_tep Fork 任务端点地址或清除为 0
返回值：NRFX_SUCCESS 如果成功分配了通道。 返回值：NRFX_ERROR_INVALID_STATE 如果没有为用户分配通道。 返回值：NRFX_ERROR_INVALID_PARAM 如果通道用户不可配置。 返回值：NRFX_ERROR_NOT_SUPPORTED 如果不支持该功能。

首先需要通过库函数来配置对应的 GPIOTE 任务和事件，这个配置在 GPIOTE 章节已经详细讲述过，这里不再累述，具体代码如下所示：

```
01. /**
02.  * 初始 GPIO 端口，设置 PIN_IN 为输入管教, PIN_OUT 为输出管脚,
03.  */
04. static void gpiote_init(void)
05. {
```

```
    ret_code_t err_code;

    err_code = nrf_drv_gpiote_init();
    APP_ERROR_CHECK(err_code);
    //配置一个 GPIOTE 输出
    nrf_drv_gpiote_out_config_t out_config =  GPIOTE_CONFIG_OUT_TASK_TOGGLE(true);
    err_code = nrf_drv_gpiote_out_init(BSP_LED_0, &out_config);
    APP_ERROR_CHECK(err_code);
    nrf_drv_gpiote_out_task_enable(BSP_LED_0);

  //配置一个 GPIOTE 输出作为分支端
    nrf_drv_gpiote_out_config_t out_config2 =  GPIOTE_CONFIG_OUT_TASK_TOGGLE(true);
    err_code = nrf_drv_gpiote_out_init(BSP_LED_1, &out_config2);
    APP_ERROR_CHECK(err_code);
    nrf_drv_gpiote_out_task_enable(BSP_LED_1);

  //配置一个 GPIOTE 输入任务
    nrf_drv_gpiote_in_config_t in_config = GPIOTE_CONFIG_IN_SENSE_HITOLO (true);
    in_config.pull = NRF_GPIO_PIN_PULLUP;
    err_code = nrf_drv_gpiote_in_init(BSP_BUTTON_0, &in_config, NULL);
    APP_ERROR_CHECK(err_code);
    nrf_drv_gpiote_in_event_enable(BSP_BUTTON_0, true);
}
```

PPI 初始化中，首先需要分配 PPI 通道的的 EEP 终点和 TEP 终点两端对应事件和任务。同时还需要调用函数 nrfx_ppi_channel_fork_assign 来分配从任务的 Fork 端点到 PPI 通道之上。具体代码如下所示：

```
void ppi_init(void)
{
  ret_code_t err_code;
   //初始化 PPI 的模块
  err_code = nrf_drv_ppi_init();
  APP_ERROR_CHECK(err_code);

  //配置 PPI 的频道
  err_code = nrfx_ppi_channel_alloc(&my_ppi_channel);
  APP_ERROR_CHECK(err_code);

  //设置 PPI 通道 my_ppi_channel 的 EEP 和 TEP 两端对应的硬件
  err_code = nrfx_ppi_channel_assign(my_ppi_channel,
                                     nrfx_gpiote_in_event_addr_get(BSP_BUTTON_0),
                                     nrfx_gpiote_out_task_addr_get(BSP_LED_0));
  APP_ERROR_CHECK(err_code);

//配置 PPI 通道 0 的分支任务端点
```

```
    err_code = nrfx_ppi_channel_fork_assign(my_ppi_channel,
                                            nrf_drv_gpiote_out_task_addr_get(BSP_LED_1));
    //使能 PPI 通道
    err_code = nrfx_ppi_channel_enable(my_ppi_channel);
    APP_ERROR_CHECK(err_code);
}
```

那么主函数里只需要调用 GPIOTE 初始化函数和 PPI 初始化函数，触发操作交给 PPI 执行。因此主函数代码如下所示：

```
/*主函数，配置 PPI 的通道*/
int main(void)
{
    gpiote_init();
    ppi_init();
    while (true)
    {
        // Do nothing.
    }
}
```

把该例子程序编译后下载到青风 nrf52832 开发板内。按下按键 1，可以使得 LED1 灯进行翻转，同时 LED2 灯也会进行翻转。