第七章 GPIOTE 与外部中断

7.1 GPIOTE 原理分析

GPIO 任务和事件(GPIOTE)模块提供了使用任务和事件访问 GPIO 引脚的功能。每个 GPIOTE 通道可以被分配到一个引脚。GPIOTE 其实就是对 GPIO 口进行操作，同时引入了外部中断的概念。比如按键控制分为两种情况，第一种是按键扫描，这种情况下，CPU 需要不停的工作，来判断 GPIO 口是否被拉低或者置高，效率是比较低的。另一种方式为外部中断控制，中断控制的效率很高，一旦系统 IO 口出现上升沿或者下降沿电平就会触发执行中断内的程序。在 nRF52832 内普通 IO 管脚设置成为 GPIO，中断和任务管脚设置称为 GPIOTE。

nRF5x 系列处理器将 GPIO 的中断的快速触发做成了一个单独的模块 GPIOTE，这个模块不仅提供了 GPIO 的中断功能，同时提供了通过 task 和 event 的方式来访问 GPIO 的功能。GPIOTE 的后缀 T 即为 task，后缀 E 即为 event。

Event 称为事件，来源与 GPIO 的输入、定时器的匹配中断等可以产生中断的外设来触发。Task 称为任务，就是执行某一个特定功能，比如翻转 IO 端口等。那么事件 event 触发应用的任务 task。task 和 event 的主要是为了和 52832 中的 PPI（可编程外围设备互联系统）模块的配合使用，PPI 模块可以将 event 和 task 分别绑定在它的两端，当 event 发生时，taks 就会自动触发。这种机制不需要 CPU 参与，极大的减小了内核负荷，降低了功率，特别适合与 BLE 定功耗蓝牙里进行应用。

GPIOTE 实际上就两种模式，一个任务模式，一个事件模式。其中任务模式作为输出使用，而事件模式就作为中断触发使用。

任务模式（task）：每个 GPIOTE 通道最多可以使用三个任务来执行对引脚的写操作。两个任务是固定的输出高电平（SET）和输出低电平（CLR），一个输出任务（OUT）可配置为执行以下操作：

- 置位（Set）
- 清零（Clear）
- 切换（Toggle）

事件模式（event）：可以从以下输入条件之一在每个 GPIOTE 通道中生成事件：

- 上升的边缘
- 下降的边缘
- 任何改变

任务模式有三种状态：置位，清零，翻转。事件模式三种触发状态：上升沿触发，下降沿触发，任意变化触发。TASK 任务通过通道 OUT[0]~OUT[7]设置输出三种触发状态，Event 则可以通过检测信号产生 PORT event 事件，也可以产生 IN[n] event 事件。

整个 GPIOTE 寄存器的个数也是非常少的，如下表所示：

寄存器名称	地址偏移	功能描述
TASKS_OUT[n]n=0~7	0x000~0x01c	写入 CONFIG [0] .PSEL 中指定的引脚的任务。 引脚上的操作配置为 CONFIG [0] .POLARITY。
TASKS_SET[n]n=0~7	0x030~0x04c	写入 CONFIG [0] .PSEL 中指定的引脚的任务。 对引脚的操作是将其设置为高。
TASKS_CLR[n]n=0~7	0x060~0x07c	写入 CONFIG [0] .PSEL 中指定的引脚的任务。 对引脚的操作是将其设置为低电平。
EVENTS_IN[n]n=0~7	0x100~0x11c	从 CONFIG [0] .PSEL 中指定的引脚生成的事件
EVENTS_PORT	0x17c	从启用了 SENSE 机制的多个输入 GPIO 引脚生成的事件
INTENSET	0x304	启用中断
INTENCLR	0x308	禁止中断
CONFIG[n]n=0~7	0x510~0x52c	OUT [n]，SET [n]和 CLR [n]任务以及 IN [n]事件的配置

GPIOTE 模块提供的了 8 个通道，这 8 个通道都是通过 CONFIG[0]~CONFIG[7]寄存器来配置。这八个通道可以通过单独设置，分别和普通的 GPIO 绑定。当需要使用 GPIOTE 的中断功能时可以设置相关寄存器的相关位让某个通道作为 event 事件模式，同时配置触发 event 的动作。比如绑定的引脚有上升沿跳变或者下降沿跳变触发 event，然后配置中断使能寄存器，配置让其 event 产生时是触发输入中断。这样就实现了 GPIO 的中断方式。

1：GPIO 绑定 GPIOTE 通道

那么如何实现和普通 GPIO 端口的绑定了？关键就是设置 GPIOTE 的 CONFIG[n]n=0~7 寄存器，该寄存器如下表所示：

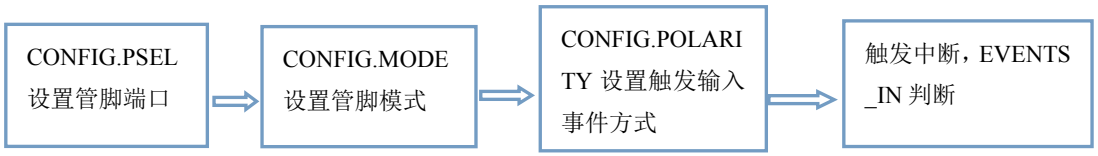
位数	Field	Value ID	Value	描述
第 1~2 位	MODE	Disabled Event Task	0 1 3	禁用。 PSEL 指定的引脚的不会绑定 GPIOTE 模式。 事件模式：把 PSEL 绑定的对应管脚设置为输入和 IN[n]事件模式。由 POLARITY 域决定操作。 任务模式：把 PSEL 绑定的对应管脚设置为输出和 SET [N]，CLR [n]和 OUT [n]任务模式。当作为一个任务启用时，GPIOTE 模块将获取该引脚，并且该引脚不能再作为常规输出引脚从 GPIO 模块写入
第 8~12 位	PSEL		[0..31]	设置与 SET [N]，CLR [n]和 OUT [n]的任务和 IN [n]的事件相绑定的 GPIO 管脚号
第 16~17 位	POLARITY	None LoToHi HiToLo Toggle	0 1 2 3	无任何影响 任务模式下：OUT[n]任务输出为高电平；事件模式下：管脚上升沿到时候产生 IN[n]输入事件。 任务模式下：OUT[n]任务输出为低电平；事件模式下：管脚下降沿到时候产生 IN[n]输入事件。 任务模式下：OUT[n]任务输出为翻转电平；事件模式下：任意管脚变化都能产生 IN[n]输入事件。
第 20 位	OUTINIT	 Low High	 0 1	当 GPIOTE 处于任务模式时：输出的初始值。 任务触发前 pin 的初始值为低电平 任务触发前 pin 的初始值为高电平

如上表所描述，每个 GPIOTE 通道通过 CONFIG.PSEL 字段与一个物理 GPIO 引脚相关联绑定。在 CONFIG.MODE 中选择事件模式时：CONFIG.PSEL 指定的引脚将被配置为输入，从而覆盖 GPIO 中的 DIR 设置。同样，当在 CONFIG.MODE 中选择任务模式时：CONFIG.PSEL 指定的引脚将被配置为覆盖 GPIO 中 DIR 设置和 OUT 值的输出。

当在 CONFIG.MODE 中选择 Disabled 时，CONFIG.PSEL 指定的引脚将使用普通 GPIO 中 PIN [n] .CNF 寄存器的配置，也就是不绑定。因此只能将一个 GPIOTE 通道分配给一个物理引脚。

2：当设置为事件模式：

设置事件模式，事件模式就是输入，通过输入信号触发事件中断。那么首先在寄存器 CONFIG.PSEL 域设置管脚，当设置了一个 GPIO 管脚绑定了 GPIOTE 通道后，再 CONFIG.MODE 域设置为事件模式；之后在 CONFIG.POLARITY 域中设置触发事件模式的输入电平，当对应电平输入 GPIOTE 通道后就产生中断，EVENTS_IN 判断对应端口中断事件是否发生。



3：当设置为任务模式：

因为任务模式为输出模式。首先设置 CONFIG.PSEL 域设置绑定 IO 管脚，再设置 CONFIG.MODE 域设置 GPIOTE 为任务模式 ；再来设置 CONFIG.POLARITY 域中设置 OUT[n]任务输出:

- 置位（Set）
- 清零（Clear）
- 切换（Toggle）

设置完 CONFIG 配置寄存器后，再来触发任务：

TASKS_OUT[n] 触发 CONFIG.POLARITY 域中设置 OUT[n]值

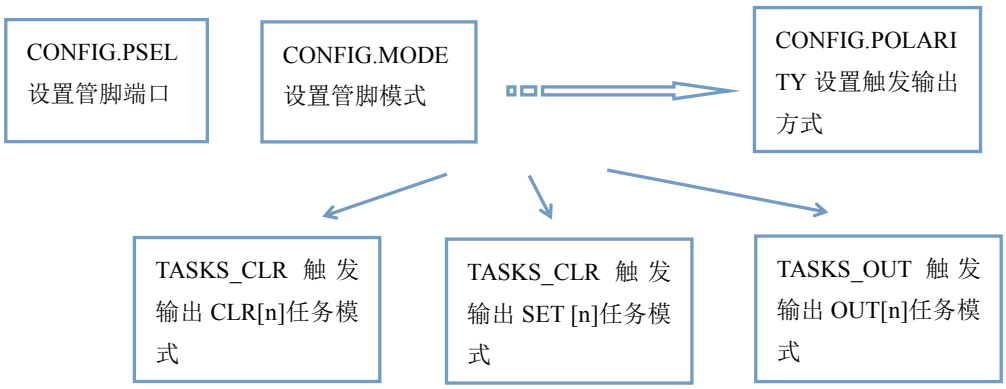
TASKS_SET[n] 触发输出高电平（SET）

TASKS_CLR[n] 触发输出低电平（CLR）

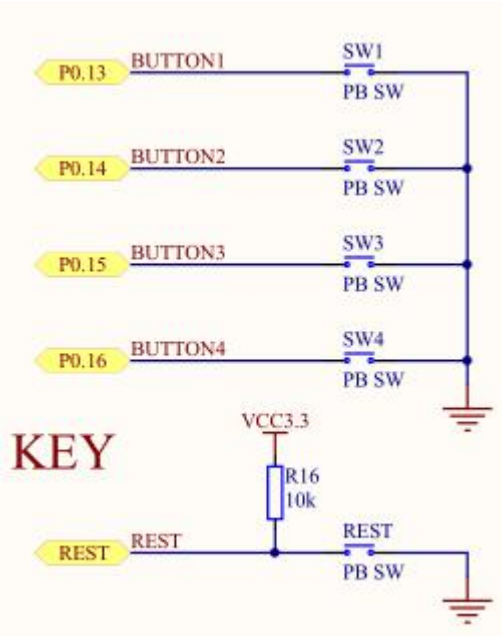
当三个状态触发同时申请，则根据下表的优先级决定先执行那钟设置：

任务状态	优先级
TASKS_OUT	1
TASKS_CLR	2
TASKS_SET	3

任务模式配置的具体流程如下图所示：



硬件方面：如图所示，在青云 nRF52832 豪华开发板上连接了四个按键（SW1、SW2、SW3、SW4），分别连接 P0.13、P0.14、P0.15、P0.16。可以通过按键配置 GPIOTE 输入来控制 led 的亮灭。



7.2 GPIOTE 事件应用

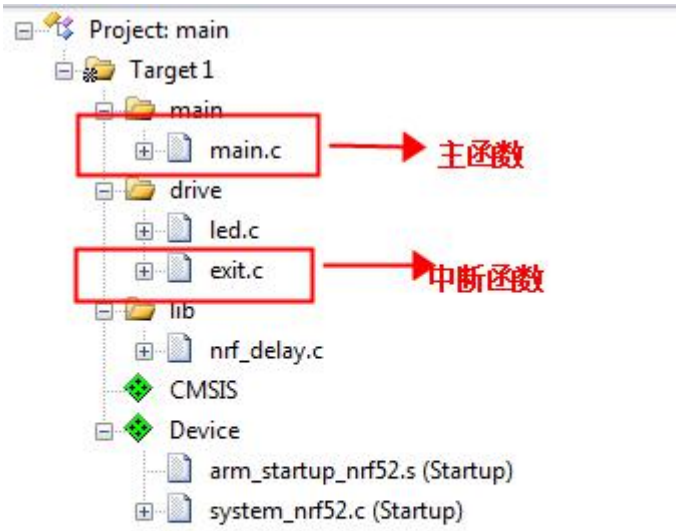
7.2.1 按键中断

下面我就来介绍下 nRF52832 的按键中断控制方式。中断控制的效率很高，一旦系统 IO 口出现上升沿或者下降沿电平就会触发执行中断内的程序，这样可以大大节省了 cpu 的占有率。中断是指由于接收到来自外围硬件（相对于中央处理器和内存）的异步信号或来自软件的同步信号，而进行相应的硬件 / 软件处理。发出这样的信号称为进行中断请求（interrupt request，IRQ）。硬件中断导致处理器通过一个上下文切换（context switch）来保存执行状态（以程序计数器和程序状态字等寄存器信息为主）；软件中断则通常作为 CPU 指令集中的一个指令，以可编程的方式直接指示这种

上下文切换，并将处理导向一段中断处理代码。中断在计算机多任务处理，尤其是实时系统中尤为有用，这样的系统，包括运行于其上的操作系统，也被称为“中断驱动的”。简单的来说就比如某个人正在做某事，突然来了个电话，他就要停下手中的事情去接电话，中断相当于这个电话。触发中断后跳出原来运行的程序去执行中断处理。

在使用 nRF52832 完成中断时，需要设置如下几个地方:第一：中断嵌套的设置。第二：外部 GPIOTE 中断函数的设置。当 IO 管脚为低的时候可以判断管脚已经按下。通过 key 的中断来控制 led 的亮灭。硬件上设计是比较简单的，这个普通的 MCU 的中断用法一致。

在代码文件中，实验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 key 工程：



如上图所示：读者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 在前面章节控制 LED 灯的时候已经写好，现在我们就来讨论下如何编写 exit.c 这个驱动子文件。

exit.c 文件主要是要起到两个作用：第一：初始化开发板上的按键中断。第二：编写中断执行代码。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。

我们使用到了按键中断，实际上使用到了事件模式，下面将主要讨论这个模式。在 CONFIG 这个寄存器里详细的进行了事件模式的配置，根据上面总结的事件模式配置方式流程，代码如下所示：

```
NRF_GPIOTE->CONFIG[0] =
(GPIOTE_CONFIG_POLARITY_HiToLo << GPIOTE_CONFIG_POLARITY_Pos)
                                    | (13<< GPIOTE_CONFIG_PSEL_Pos)
                                | (GPIOTE_CONFIG_MODE_Event << GPIOTE_CONFIG_MODE_Pos);
//中断配置
```

上面一段代码的编写严格按照了寄存器要求进行，首先是 MODE，也就是模式设置，该位用来配置本 GPIOTE 通道是作为 event 还是 task 的，这里我们设置成事件模式。PSEL 设置对应的绑定的 IO 管脚，我们选择了 SW1 管脚 P0.13 作为触发管脚，POLARIY 极性设置为下降沿触发。

设置好了工作方式后，我们就需要进行中断的使能了：

```
NVIC_EnableIRQ(GPIOTE_IRQn);//中断嵌套设置
```

```
02.  NRF_GPIOTE->INTENSET  = GPIOTE_INTENSET_IN0_Set <<
     GPIOTE_INTENSET_IN0_Pos;// 使能中断类型:
```

上面的任务基本上就可以把 IO 管脚中断配置好了，如果你搞清楚寄存器，那么这个配置也是十分简单的。

中断函数的设计，主要任务就是要求判断中断发生后，要对 LED 灯进行翻转，当然你可以加入其它更多的任务。

```
01.  void GPIOTE_IRQHandler(void)
02. {
03.      if ((NRF_GPIOTE->EVENTS_IN[0] == 1) &&
04.          (NRF_GPIOTE->INTENSET & GPIOTE_INTENSET_IN0_Msk))
05.      {   Delay(10000); //延迟消抖
06.          NRF_GPIOTE->EVENTS_IN[0] = 0; //中断事件清零.
07.      }
08.     LED_Toggle();//led 灯翻转
09. }
```

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，判断按键按下后就可以翻转 IO 口，LED 灯指示相应的变化。函数如下所示：

```
01.  /********************* (C) COPYRIGHT 2019 青风电子 *********************
02.  * 文件名  ：main
03.  * 实验平台：青云 nRF528xx 蓝牙开发板
04.  * 描述    ：按键中断
05.  * 作者    ：青风
06.  * 社区    ：www.qfv8.com
07. ********************************************************************************/
08. #include "nrf52.h"
09. #include "nrf_gpio.h"
10. #include "exit.h"
11. #include "led.h"
12.
13. int main(void)
14. {
15.  LED_Init();
16.   LED_Open();
17.  /*config key*/
18.  EXIT_KEY_Init();
19.  while(1)
20.  {
21.      }
22. }
```

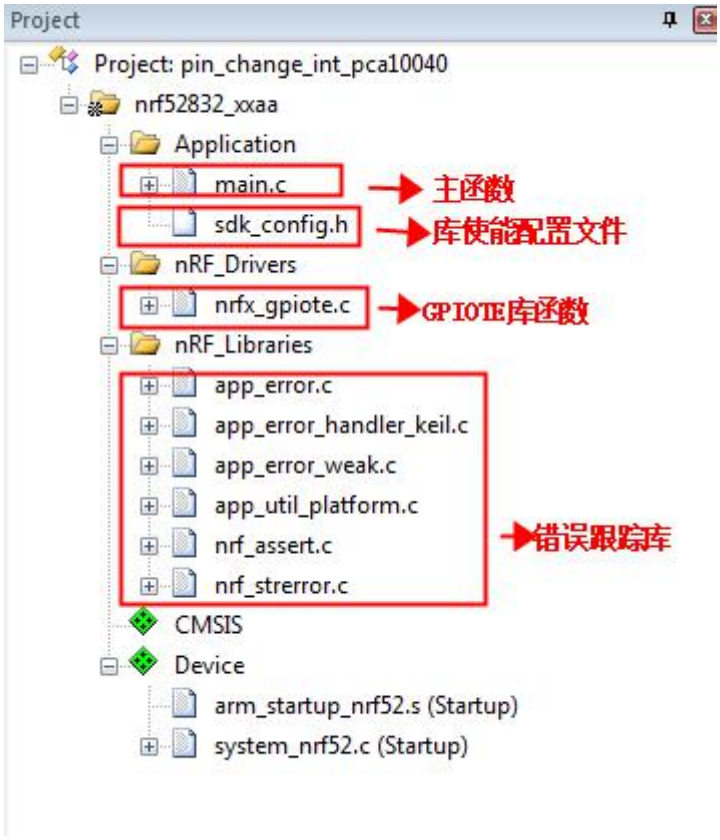
实验下载到青云 nRF52832 开发板后的实验现象如下：按下复位键后，按下按键 1 后，LED1 灯会对应翻转。

7.2.2 GPIOTE 事件组件的应用

在官方 SDK 各个版本中，对各个外设提供了直接的驱动组件。这些组件用现在流行的说法就是驱动库。特别是针对后面 BLE 蓝牙应用代码，组件库的是运用更加频繁。外设组件库的加入，对应完善外设驱动功能、减小工程师工作量，都有则极大的效果。然而驱动组件库的使用，比上一节直接操作寄存器更加复杂，需要读者深入理解库内函数的定义以及参数的设定。这小节将带领大家进入组件库的领域，看如何采用驱动组件库来设置一个 GPIOTE 的应用。同时以上一节寄存器直接操作的过程，也就相对容易理解复杂组件库的编写 GPIOTE 的步骤。

首先来看看工程树如下图所示，SDK 中提供了 nrf_drv_gpiote.c 这个文件做为 GPIOTE 驱动组件库，内包含了很多 GPIOTE 的操作的 API 函数，我们不全部展开，只结合应用，详细说明部分函数 API。而 nrf_drv_common.c 文件为中断嵌套的配置文件，使用中断的时候需要添加进去。组件库的函数都使用了 APP_ERROR_CHECK(err_code)函数进行错误定位，因此 app_error.c 和 nrf_assert.c 做为错误定位组件也需要加入工程中。

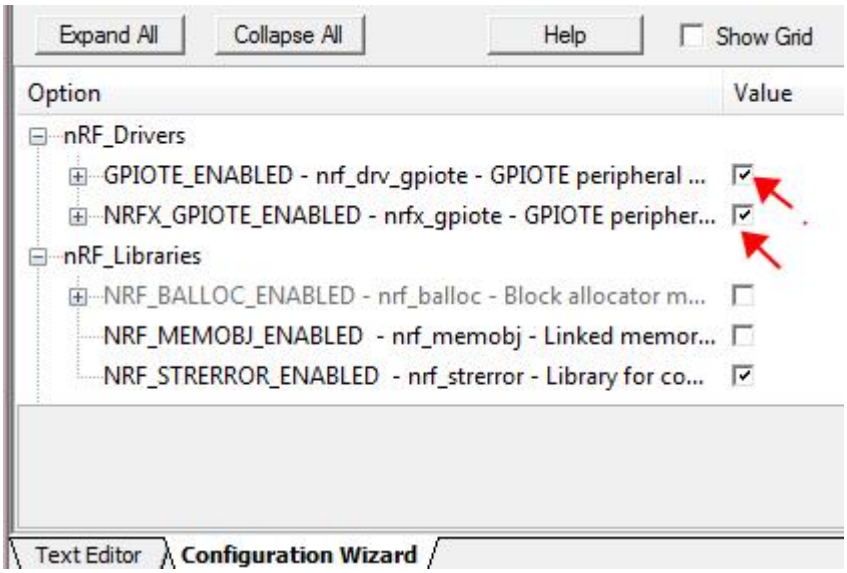
添加好必须的库函数后，再添加对应的文件路径，过程参考第 3 章内容，这里不在累述。打开工程文件夹目录为 pac10040，如下图所示：



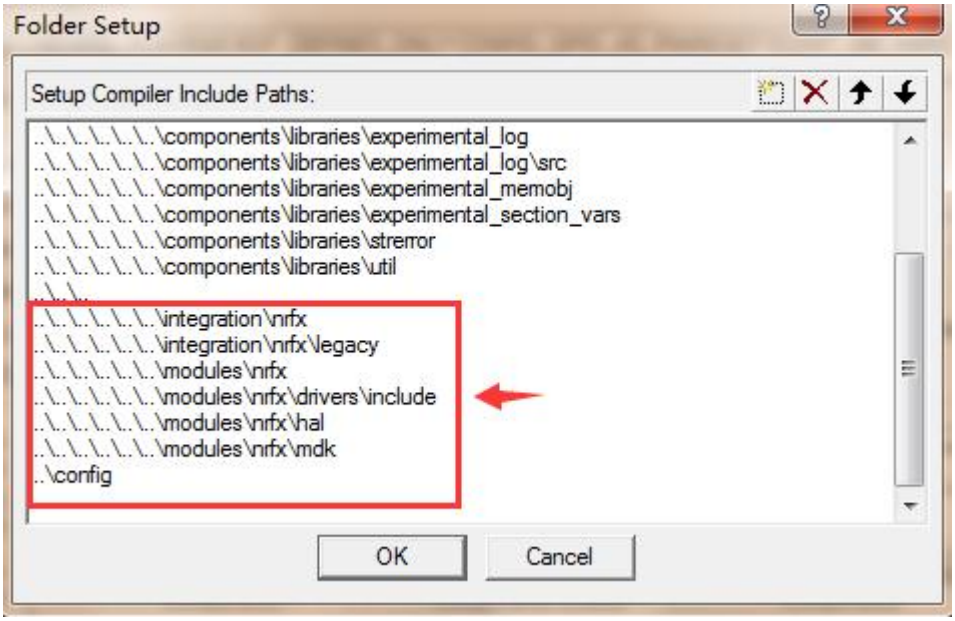
官方提供了一个驱动 nrfx_gpiote 的 GPIOTE 驱动库，但这个驱动库有带错误跟踪函数，所以工程中还必须添加如上图所示的错误跟踪库。同时区别与寄存器编程，组件库需要配置 sdk_config.h 配置文件。

打开 sdk_config.h 配置文件，打开配置向导 Configuration Wizard，勾选如下两个使能项目：

GPIOTE_ENABLE 使能 GPIOTE 驱动库；NRFX_GPIOTE__ENABLE 使能 GPIOTE 兼容库；



同时需要在 C/C++中添加硬件 GPIOTE 的工程文件和头文件路径，如下所示：



由于驱动组件库是可以直接调用的，那么编程者的任务就只有编写主函数 main。其基本架构和寄存器直接操作相似，下面来对照代码一一分析与解剖：

```
01. //GPIOE 驱动初始化
02.         err_code = nrf_drv_gpiote_init();
03.         APP_ERROR_CHECK(err_code);
```

调用 API 函数 nrf_drv_gpiote_init，对 GPIOE 进行初始化，进入 nrf_drv_gpiote_init 函数内部，发现其函数主要功能是设置中断类型为 PORT 中断，也就是端口事件，中断优先级为低，如下面所示：

```
139 
140 ret_code_t nrf_drv_gpiote_init(void)
141 {
142     if (m_cb.state != NRF_DRV_STATE_UNINITIALIZED)
143     {
144         return NRF_ERROR_INVALID_STATE;
145     }
146 
147     uint8_t i;
148     for (i = 0; i < NUMBER_OF_PINS; i++)
149     {
150         pin_in_use_clear(i);
151     }
152     for (i = 0; i < (NUMBER_OF_GPIO_TE+GPIOTE_CONFIG_NUM_OF_LOW_POWER_EVENTS); i++)
153     {
154         channel_free(i);
155     }
156 
157     nrf_drv_common_irq_enable(GPIOTE_IRQn, GPIOTE_CONFIG_IRQ_PRIORITY);
158     nrf_gpiote_int_enable(GPIOTE_INTENSET_PORT_Msk);
159     m_cb.state = NRF_DRV_STATE_INITIALIZED;
160 
161     return NRF_SUCCESS;
162 }
163 
```

设置中断优先级

使能中断类型为PORT中断

然后设置控制 LED 灯的输入管教，管脚配置为低电平向高电平 GPIOTE_CONFIG _OUTINIT_Low：

```
04.  //配置设置 GPIOE 输出参数  从低电平到高电平
05.      nrf_drv_gpiote_out_config_t out_config = GPIOTE_CONFIG_OUT_SIMPLE(0);
06.      //GPIOE 输出初始化
07.      err_code = nrf_drv_gpiote_out_init(LED_1, &out_config);
08.      APP_ERROR_CHECK(err_code);
```

再配置按键输入为 GPIOTE 输入，触发 POLARIY 极性方式设置为翻转，管脚为上拉输入，按键 BUTTON_1 在库中宏定义为管脚 P0.17。代码如下：

```
09.      //配置设置 GPIOTE 输入参数
10.      nrf_drv_gpiote_in_config_t in_config = GPIOTE_CONFIG_IN_SENSE_TOGGLE(1);
11.      in_config.pull = NRF_GPIO_PIN_PULLUP;
12.       //GPIOTE 输入初始化，设置触发输入中断
13.      err_code = nrf_drv_gpiote_in_init(BUTTON_1, &in_config, in_pin_handler);
14.      APP_ERROR_CHECK(err_code);
```

最后的配置就是关键，设置工作模式，进入 nrf_drv_gpiote_in_event_enable 函数内部，函数中调用了 nrf_gpiote_event_enable(channel)函数，也就是使能了通道 0 为事件模式：

```
15.   //设置 GPIOE 输入事件使能
16.      nrf_drv_gpiote_in_event_enable(BUTTON_1, true);
```

中断函数的设计，主要任务就是要求判断中断发生后，要对 LED 灯进行翻转，当然你可以加入按键防抖判断使得效果更加好。

```
17.    void in_pin_handler(nrf_drv_gpiote_pin_t pin, nrf_gpiote_polarity_t action)//中断回调函数
18. {
19.       nrf_drv_gpiote_out_toggle(PIN_OUT);
20. }
```

那么主函数就是十分的简单了，配置好 GPIOTE 后，循环等待中断的发生,当按键按下后，会触发中断，led 灯发生翻转：

```
21. 
22. /**
23.  *主函数，初始化后循序等待
```

```
24.   */
25.  int main(void)
26.  {
27.      gpio_init();
28.
29.      while (true)
30.      {
31.
32.      }
33.  }
```

总结：通过对比发现，驱动组件库编写 GPIOTE 中断的流程思路一致，但是如何想熟练使用驱动组件库编程就需要对驱动库内的函数有相当熟悉的了解。这就需要读者深入到库函数代码定义中进行阅读了，本节作为一个抛砖引玉，带领大家逐步的深入到组件库的开发中。

7.3 GPIOTE PORT 应用

在上节中，把普通的 GPIO 端口配置为 GPIOTE 中断输入事件，能够绑定的只有 8 个通道，如果我们中断的数据量超过了 8 个，多的中断无法处理，如何出现这种情况，怎么处理？显然芯片设计厂家为了针对这种情况，特别在 GPIOTE 模块中提出了 GPIOTE PORT 功能。

GPIOTE PORT 是从使用 GPIO DETECT 信号的多个 IO 输入引脚来生成的事件。该事件将在 DETECT 信号的上升沿而产生。也就是说这个功能可以通过普通的 32 个 IO 端口产生，相当与一个总通道，32 个 IO 端口共用这个通道来申请中断。

同时 GPIO DETECT 信号就是通过 GPIO 的 SENSE 寄存器打开，此功能始终处于启用状态。就是外围设备本身是休眠状态时，也不需要请求时钟或其他功率密集型基础架构来启用此功能。因此此功能可用于在系统启动时从 WFI 或 WFE 类型的睡眠时，来唤醒 CPU、所有外设和 CPU 空闲。达到唤醒系统启动模式下的最低功耗模式。

此时为了在配置源时防止来自 PORT 事件的虚假中断，用户应首先禁用 PORT 事件中的中断（通过 INTENCLR.PORT），然后配置源（PIN_CNF [n] .SENSE），清除配置期间可能发生的任何潜在事件（向 EVENTS_PORT 写入'1'），最后启用中断（通过 INTENSET.PORT）。寄存器的配置描述如上所述相当简单，下面主要探讨下组件库下如何实现。我们采用上一节 GPIOTE 事件输入组件的例子工程，直接在上面进行修改 main.c 文件，工程结构树不变动。

采用组件库编写 GPIOTE 输入事件与 GPIOTE PORT 事件的主要区别有两个地方：

第一：配置事件的时候选择 IN 事件还是 PORT 事件，这个通过配置函数实现：

GPIOTE_CONFIG_IN_SENSE_HITOLO(false);

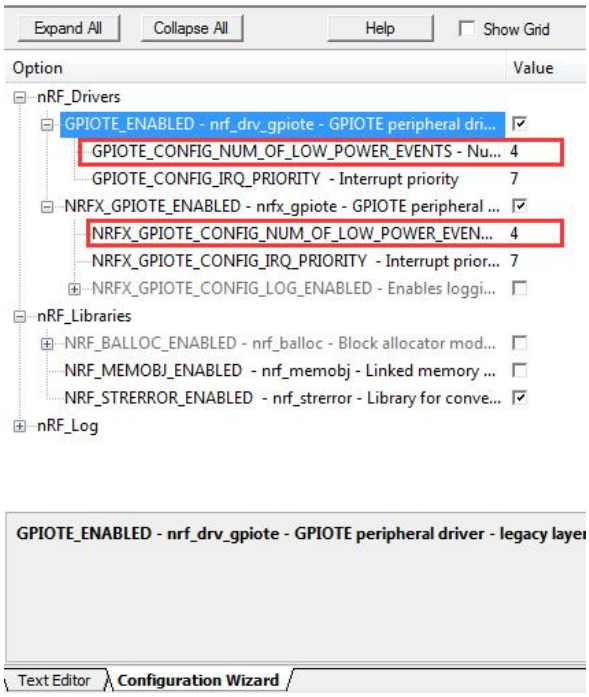
当函数参数是 false 的时候，选择 PORT 事件；当函数参数是 ture 的时候，选择 IN 事件；

第二：所有 32 个 IO 端口触发的中断都是 INTENSET.PORT，因此配置都指向一个中断配置就可以了，演示代码里我们把四个轻触按键全部绑定到 PORT 事件上去。具体代码如下所示：

```
01.      /**
02.  配置 GPIOTE 初始化
03.   */
04.  static void gpio_init(void)
05.  {    //配置 LED 灯输出
```

```
06.         nrf_gpio_cfg_output(LED_1);
07.       nrf_gpio_cfg_output(LED_2);
08.       nrf_gpio_cfg_output(LED_3);
09.       nrf_gpio_cfg_output(LED_4);
10.       ret_code_t err_code;
11.          //初始化 GPIOTE
12.         err_code = nrf_drv_gpiote_init();
13.         APP_ERROR_CHECK(err_code);
14.
15.         //配置 SENSE 模式，选择 fales 为 PORT 事件
16.         nrf_drv_gpiote_in_config_t in_config = GPIOTE_CONFIG_IN_SENSE_HITOLO(false);
17.         in_config.pull = NRF_GPIO_PIN_PULLUP;
18.
19.       //配置按键 0 绑定 POTR
20.         err_code = nrf_drv_gpiote_in_init(BSP_BUTTON_0, &in_config, in_pin_handler);
21.         APP_ERROR_CHECK(err_code);
22. //使能中断事件
23.         nrf_drv_gpiote_in_event_enable(BSP_BUTTON_0, true);
24.
25.          //配置按键 1 绑定 POTR
26.       err_code = nrf_drv_gpiote_in_init(BSP_BUTTON_1, &in_config, in_pin_handler);
27.         APP_ERROR_CHECK(err_code);
28. //使能中断事件
29.         nrf_drv_gpiote_in_event_enable(BSP_BUTTON_1, true);
30.   //配置按键 2 绑定 POTR
31.         err_code = nrf_drv_gpiote_in_init(BSP_BUTTON_2, &in_config, in_pin_handler);
32.         APP_ERROR_CHECK(err_code);
33. //使能中断事件
34.         nrf_drv_gpiote_in_event_enable(BSP_BUTTON_2, true);
35.   //配置按键 3 绑定 POTR
36.       err_code = nrf_drv_gpiote_in_init(BSP_BUTTON_3, &in_config, in_pin_handler);
37.         APP_ERROR_CHECK(err_code);
38. //使能中断事件
39.       nrf_drv_gpiote_in_event_enable(BSP_BUTTON_3, true);
40. }
```

如果绑定到一个 PORT 事件上有四个中断，那么还需要在配置文件 sdk_config.h 中，对中断配置的事件数目需要修改为 4，如下图所示：

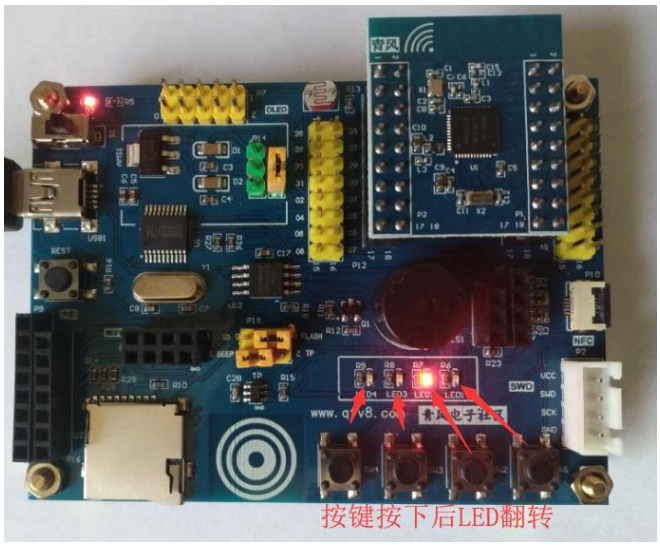


in_pin_handler 作为四个 IO 端口同时申请的 PORT 事件中断的回调处理函数，其内部必须识别是哪个 IO 口号发生的事件。由于没有独立的中断标志，因此需要用判断语句来判断是哪个 PIN 发生的回调事件，具体代码如下所示：

```
/**
   GPIOTE 中断处理
 */
void in_pin_handler(nrf_drv_gpiote_pin_t pin, nrf_gpiote_polarity_t action)
{
        //事件由按键 S1 产生，即按键 S1 按下
    if(pin == BUTTON_1)
      {
        //翻转指示灯 D1 的状态
       nrf_gpio_pin_toggle(LED_1);
      }
      //事件由按键 S2 产生，即按键 S2 按下
       else if(pin == BUTTON_2)
      {
        //翻转指示灯 D2 的状态
       nrf_gpio_pin_toggle(LED_2);
      }
      //事件由按键 S3 产生，即按键 S3 按下
       else if(pin == BUTTON_3)
      {
        //翻转指示灯 D3 的状态
       nrf_gpio_pin_toggle(LED_3);
```

```
23.         }
24.         //事件由按键 S4 产生，即按键 S4 按下
25.         else if(pin == BUTTON_4)
26.         {
27.           //翻转指示灯 D4 的状态
28.         nrf_gpio_pin_toggle(LED_4);
29.         }
30.
31. }
```

本实验编译后，使用 KEIL 下载到青云 nRF52832EK 开发板后的实验现象如下：下载后按下按键 1，LED1 灯翻转,下载后按下按键 2，LED2 灯翻转,下载后按下按键 3，LED3 灯翻转,下载后按下按键 4，LED4 灯翻转。

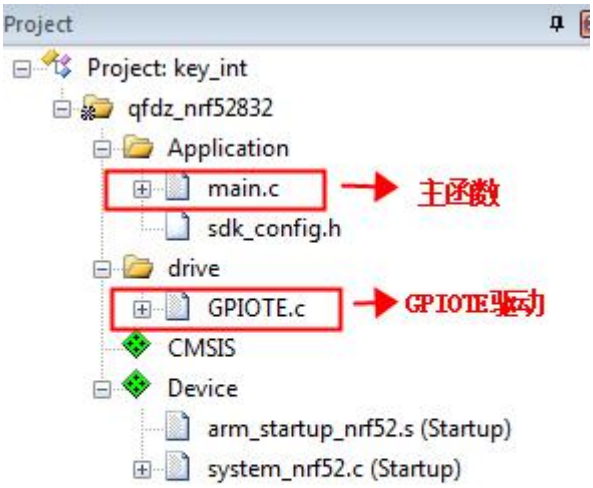


7.4 GPIOTE 任务应用

7.4.1 GPIOTE 任务触发 LED

GPIOTE 具有任务模式，任务模式就是输出模式。如果把 GPIO 管脚绑定了 GPIOTE 通道后，把它配置为任务模式，则可以实现输出功能。任务模式的使用不是孤立的，一般都是由事件来触发任务，如果在事件和任务中间架设一个通道，也就是后面会将的 PPI，那么整个过程不需要 CPU 参与了，大大节省 mcu 的资源。本例首先简单的演示下任务是如何输出的，我们用输出端口来控制一个 LED 灯，完成我们输出的功能。

在 7.1 节中，对应 GPIOTE 的任务输出的步骤有一个归纳，我们首先通过寄存器方式探讨如何搭建任务输出的功能。寄存器的工程目录树比较简单，不需要加入错误检测库和配置 sdk_config.h 文件，只需要我们自己通过寄存器编写一个 GPIOTE 的驱动，然后主函数调用这个驱动就可以，工程目录树如下所示：



首先是 GPIOTE 任务初始化，我们初始化两个通道。初始化首先首先设置通道 CONFIG[0] .PSEL 域设置绑定 GPIOTE0 管脚，CONFIG[1] .PSEL 域设置绑定 GPIOTE1 管脚；再设置两个通道的 CONFIG.MODE 域设置 GPIOTE 为 Task 任务模式 ；再来设置 CONFIG.POLARITY 域中设置 OUT[0]任务输出为翻转电平，OUT[1]任务输出为低电平。具体代码如下所示：

```
#define GPIOTE0            19
#define GPIOTE1            20

void GPIOTE_TASK_Init(void)
{

    NVIC_EnableIRQ(GPIOTE_IRQn);//中断嵌套设置

    NRF_GPIOTE->CONFIG[0] =
(GPIOTE_CONFIG_POLARITY_Toggle << GPIOTE_CONFIG_POLARITY_Pos)//绑定通道 0
            | (GPIOTE0 << GPIOTE_CONFIG_PSEL_Pos) // 配置任务输出状态
            | (GPIOTE_CONFIG_MODE_Task << GPIOTE_CONFIG_MODE_Pos);//任务模式

  NRF_GPIOTE->CONFIG[1] =
(GPIOTE_CONFIG_POLARITY_HiToLo << GPIOTE_CONFIG_POLARITY_Pos)//绑定通道 1
            | (GPIOTE1<< GPIOTE_CONFIG_PSEL_Pos)  // 配置任务输出状态
            | (GPIOTE_CONFIG_MODE_Task << GPIOTE_CONFIG_MODE_Pos);//任务模式
}
```

初始化代码里把 GPIOTE0 绑定到了 19 管脚，GPIOTE1 绑定到了 20 管脚。这两个管脚分布接到了 LED3 和 LED4 管脚，当 OUT[0]任务输出为 GPIOTE_CONFIG_POLARITY_Toggle 翻转电平可以使 LED3 翻转，OUT[1]任务输出为 GPIOTE_CONFIG_POLARITY_HiToLo 低电平可以点亮 LED4。通过这种方式验证下任务输出模式。

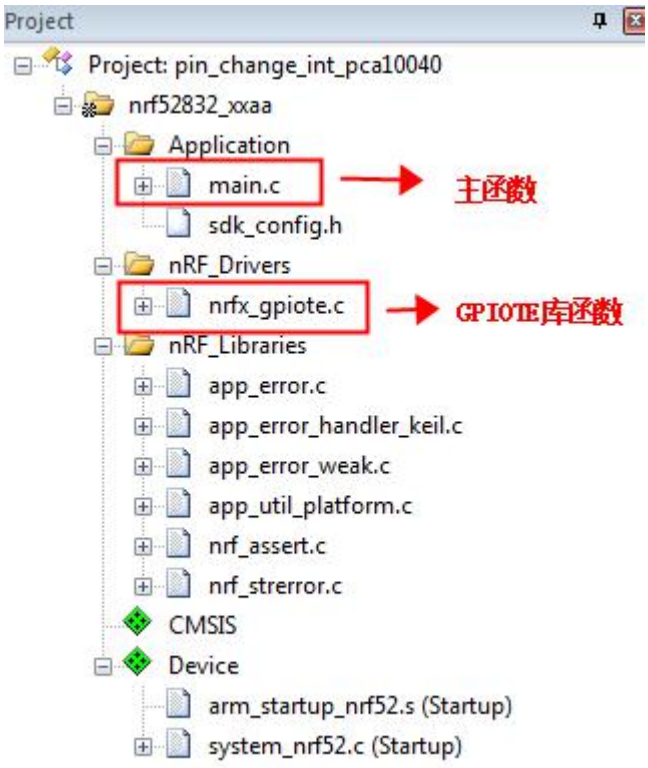
主函数中，需要配置 TASKS_OUT[n]寄存器，使能 CONFIG.POLARITY 域的设置。具体代码如下所示：

```
/******************** (C) COPYRIGHT 2019 青风电子 ********************
 * 文件名  ：main
 * 出品论坛 ：www.qfv8.com
 * 实验平台：青云 nRF52xx 蓝牙开发板
 * 描述    ：寄存器方式 GPIOTE 输出模式
 * 作者    ：青风
 **********************************************************************/

#include "nrf52.h"
#include "nrf_gpio.h"
#include "GPIOTE.h"
#include "led.h"
#include "nrf_delay.h"

int main(void)
{

  /*配置按键中断*/
    GPIOTE_TASK_Init();
  while(1)
  {
        //触发输出任务模式
        NRF_GPIOTE->TASKS_OUT[0]=1;
        NRF_GPIOTE->TASKS_OUT[1]=1;
        nrf_delay_ms(500);
        }
}
```

实验下载到青云 nRF52832 开发板后的实验现象如下：LED3 灯会对应 500ms 时间翻转闪烁，LED4 会保持常亮。

7.4.2 组件方式的任务配置

下面再来探讨下驱动库如何实现任务的配置，驱动库的实现步骤应该和寄存器方式对应，关键点就是如何调用驱动库的函数。工程目录和 7.2.2 节的工程目录相同，配置 sdk_config.h 中使能选项相同。如下图所示：



本例里同样的对比寄存器方式编写两路的 GPOITE 任务输出，一路配置为输出翻转，一路设置为输出低电平。和 GPIOTE 事件相反，初始化任务应该是输出，同时需要使能任务和触发任务的驱动库函数。下面介绍下如下三个函数：

1：函数 nrf_drv_gpiote_out_init 等同于函数 nrfx_gpiote_out_init，该函数介绍如下所示：

函数：nrfx_err_t nrfx_gpiote_out_init(nrfx_gpiote_pin_t pin, nrfx_gpiote_out_config_t const * p_config);
*功能：函数功能是用于初始化 GPIOTE 输出引脚。输出引脚可以由 CPU 或 PPI 控制。最初的配置指定使用哪种模式。如果使用 PPI 模式，驱动程序尝试分配一个可用的 GPIOTE 通道。如果没有信道可用时，返回一个错误。
* 参数： pin 对应的管脚 p_config 初始化配置
*返回值： NRFX_SUCCESS 如果初始化成功 * 返回值： NRFX_ERROR_INVALID_STATE 如果没有初始化驱动或者管脚已经被使用 * 返回值： NRFX_ERROR_NO_MEM 如果没有 GPIOTE 频道是可用的

2 函数 nrf_drv_gpiote_out_task_enable 等同于函数 nrfx_gpiote_out_task_enable，该函数介绍如下所示：

函数：void nrfx_gpiote_out_task_enable(nrfx_gpiote_pin_t pin);

*功能：函数功能是用于启用 GPIOTE 输出 pin 任务。
*参数： pin 对应的管脚

那么任务初始化函数遵循 7.1 节讲解的配置任务的步骤，同时需要调用以上两个配置函数：

首先设置绑定 IO 管脚，同时设置 OUT[n]任务输出模式，再启动 GPIOTE 为任务模式，最后主函数里触发输出，具体代码如下所示:

```
/******************** (C) COPYRIGHT 2019 青风电子 ********************
 * 文件名  ：main
 * 出品论坛 ：www.qfv8.com
 * 实验平台：青云 nRF528xx 蓝牙开发板
 * 描述    ：GPIOTE 任务组件库编程
 * 作者    ：青风
 **********************************************************************/

#include <stdbool.h>
#include "nrf.h"
#include "nrf_drv_gpiote.h"
#include "app_error.h"
#include "nrf_delay.h"

#define GPIOTE0            19
#define GPIOTE1            20


 void GPIOTE_TASK_Init(void)
 {

  ret_code_t err_code;
 //初始化 GPIOTE 程序模块
 err_code = nrf_drv_gpiote_init();
   APP_ERROR_CHECK(err_code);

 //定义 GPIOTE 输出初始化结构体，主要是配置为翻转模式
 nrf_drv_gpiote_out_config_t config1 = GPIOTE_CONFIG_OUT_TASK_TOGGLE(true);
 //绑定 GPIOTE 输出引脚
   err_code = nrf_drv_gpiote_out_init(GPIOTE0, &config1);
   APP_ERROR_CHECK(err_code);
 //配置为引脚 LED_3 所在 GPIOTE 通道的任务模式
 nrf_drv_gpiote_out_task_enable(GPIOTE0);


      //定义 GPIOTE 输出初始化结构体，主要是配置为低电平输出
 nrf_drv_gpiote_out_config_t config2 = GPIOTE_CONFIG_OUT_TASK_LOW;
 //绑定 GPIOTE 输出引脚
```

```
    err_code = nrf_drv_gpiote_out_init(GPIOTE1, &config2);
    APP_ERROR_CHECK(err_code);
  //配置为引脚 LED_4 所在 GPIOTE 通道的任务模式
  nrf_drv_gpiote_out_task_enable(GPIOTE1);

  }

```

主函数中，只需要每隔 500ms 的使能一次任务输出，具体代码如下所示：

```
int main(void)
{

    GPIOTE_TASK_Init();
    while(true)
        { //触发输出，即指示灯 D3，D4 翻转状态
      nrf_drv_gpiote_out_task_trigger(GPIOTE0);
       nrf_drv_gpiote_out_task_trigger(GPIOTE1);
       nrf_delay_ms(500);
  }
}
```

实验下载到青云 nRF52832 开发板后的实验现象如下：LED3 灯会对应 500ms 时间翻转闪烁，LED4 会保持常亮。