青风带你玩蓝牙 nRF52832 系列教程

----------作者：青风

出品论坛：　www.qfv8.com　青风电子社区

作者： 青风
出品论坛: www.qfv8.com
淘宝店：http://qfv5.taobao.com
QQ 技术群：346518370
硬件平台：青云 QY-nRF52832 开发板

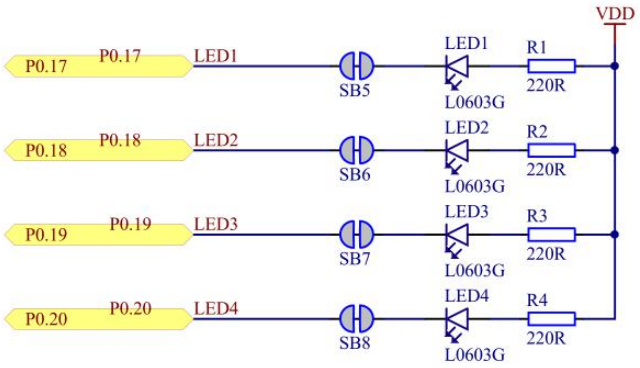
6.1 GPIO 的应用

在讲第一个外设实例之前，我要先对许多初学硬件芯片的朋友说明几个关键的学习问题：首先是学习资料的准备，在新的处理器出来后，我们要如何入门，如何进行开发，这时相关的技术手册就是必须的了，以后我们的讲解与分享中都会用到芯片的技术手册，来分析下如何采用手册查找相关说明，实际上这也是工程师的必经之路。

6.1.1 原理分析：

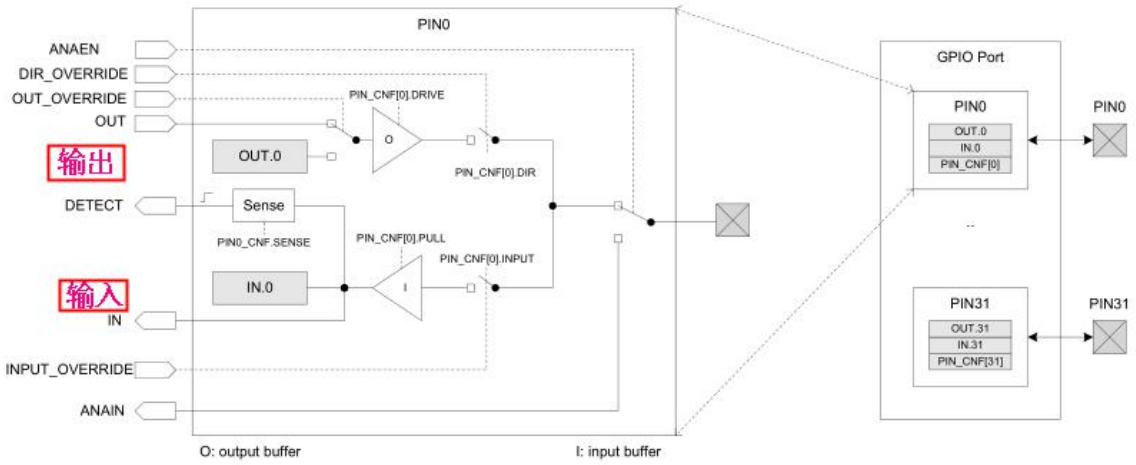
对于一个处理器来说，最简单的控制莫过于通过 I/O 端口输出的电平控制设备，本节就讲述一个经典的 LED 灯控制来开启大小 nRF52832 系列处理器的开发之旅。

硬件方面青云 QY-nRF52832 开发板上，通过管脚 P0.17 到管脚 P0.20 连接 4 个 LED 灯，我们下面的任务首先来点亮它。IO 管脚接分别接一个发光二级管，因此当把 IO 管脚定义为输出低电平的时候，在二级管两端产生电势差，就可以点亮发光二极管了。

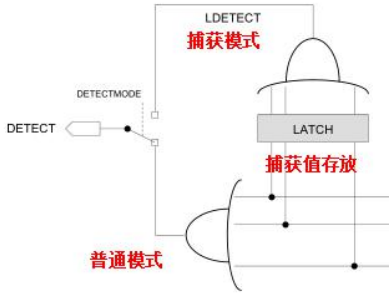


ARM CORTEX M4 的 IO 口配置有多种状态需要设置，那么下面我们一一介绍：
首先看看 IO 口的模式，查看 nRF52832 参考手册，端口可以配置为 4 种模式：输入模式，输出模式，复用模式，模拟通道模式。由于 nRF52832 的 IO 管脚复用了其它的外设功能，比如 I2C,SPI,UART

等。而通用 IO 口具有输入和输出模式：



中间的 Sense 寄存器可以捕捉 GPIO 端口状态，如果选择 LDETECT 模式，则可以把相关状态存储在 LATCH 寄存器内，结构如下图所示：



实际 nRF52832 上所包含的寄存器是非常的简单的，下面把数据手册的说明进行了标注：

Register	Offset	Description
REGISTERS		
OUT	0x504	Write GPIO port
OUTSET	0x508	Set individual bits in GPIO port
OUTCLR	0x50C	Clear individual bits in GPIO port
IN	0x510	Read GPIO port
DIR	0x514	Direction of GPIO pins
DIRSET	0x518	Setting DIR register
DIRCLR	0x51C	Clearing DIR register
PIN_CNF[0]	0x700	Configuration of pin 0
..	..	..
PIN_CNF[31]	0x77C	Configuration of pin 31

端口输出（写）

写1或者写0

端口输入（读）

IO口方向设置

方向置位

方向清零

对应0到31端口位设置

LATCH	0x5	状态存入的寄存器	Latch indicating which GPIO pins have met the criteria set in PIN_CNF[n].SENSE register
DETECTMODE	0x524	传感模式选择	Select between default DETECT signal behaviour and LDETECT mode

如果大家使用 nRF52832 官方提供的库函数编程，可以在"nrf_gpio.h"库文件中找到设置 IO 模式的结构体 nrf_gpio_port_dir_t，这里完全是对照参考手册进行编写的，实际上 nRF52832 的库是比较简单的，大家完全可以直接用寄存器操作 ：

```
typedef enum
{
    NRF_GPIO_PIN_DIR_INPUT    = GPIO_PIN_CNF_DIR_Input,
///< Input
    NRF_GPIO_PIN_DIR_OUTPUT = GPIO_PIN_CNF_DIR_Output
    ///< Output
} nrf_gpio_port_dir_t;
```

首先我们来介绍下输入和输出模式也就是 NRF_GPIO_PORT_DIR_INPUT 和 NRF_GPIO_PORT_DIR_OUTPUT。其中输出模式寄存器为推挽输出。输入的模式可以分为上拉和下拉模式，这就比较简单了，同样大家使用库函数编程的时候，可以在"nrf_gpio.h"文件中找到设置输入模式的结构体 nrf_gpio_pin_pull_t ，如数据手册上描述：

C RW PULL

	DISABLED	0	No pull
ID RW Field	Value ID	Value	Description
	PULLDOWN	1	Pull down on pin
	PULLUP	2	Pull up on pin

"nrf_gpio.h"库文件也给出了定义：

```
typedef enum
{
 NRF_GPIO_PIN_NOPULL=GPIO_PIN_CNF_PULL_Disabled,      //无上拉下拉
 NRF_GPIO_PIN_PULLDOWN=GPIO_PIN_CNF_PULL_Pulldown,    // 下拉
 NRF_GPIO_PIN_PULLUP= GPIO_PIN_CNF_PULL_Pullup,              //上拉
```

并且对于输入管脚可以设置不同的 sense 级别，如手册上所描述：

E	RW	SENSE			Pin sensing mechanism
		DISABLED	0		Disabled
		HIGH	1		Sense for high level
		LOW	2		Sense for low level

"nrf_gpio.h"库文件也给出了定义：

```
typedef enum
{
    NRF_GPIO_PIN_NOSENSE= GPIO_PIN_CNF_SENSE_Disabled, ///<    Pin sense level disabled.
    NRF_GPIO_PIN_SENSE_LOW= GPIO_PIN_CNF_SENSE_Low, ///<    Pin sense low level.
    NRF_GPIO_PIN_SENSE_HIGH = GPIO_PIN_CNF_SENSE_High,///<    Pin sense high level.
} nrf_gpio_pin_sense_t;
```

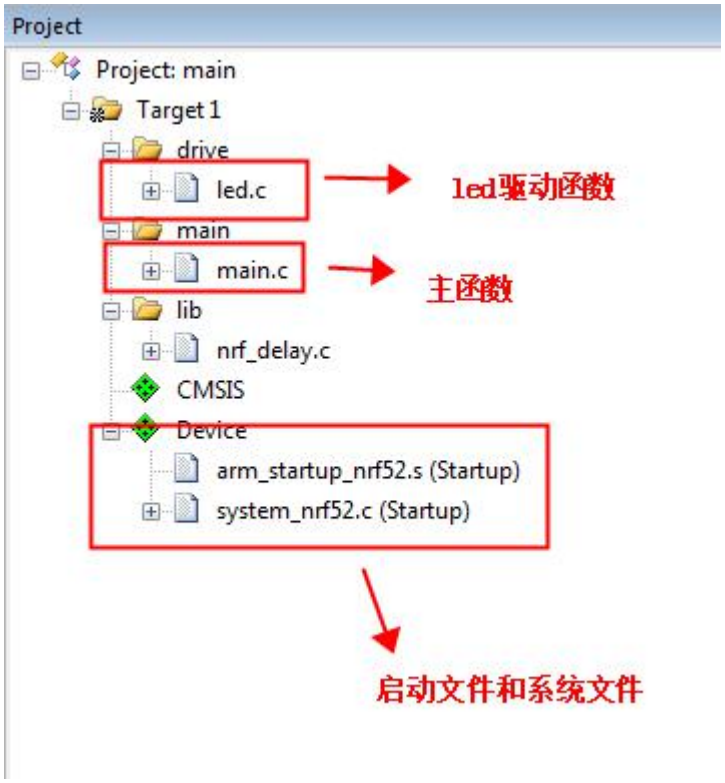
对应选择了 NRF_GPIO_PORT_DIR_OUTPUT 输出，也可以选择管脚的输出驱动模式，结构体如下所示：

```
typedef enum
{
    NRF_GPIO_PIN_S0S1 = GPIO_PIN_CNF_DRIVE_S0S1, ///< !< Standard '0', standard '1'
    NRF_GPIO_PIN_H0S1 = GPIO_PIN_CNF_DRIVE_H0S1, ///< !< High drive '0', standard '1'
    NRF_GPIO_PIN_S0H1 = GPIO_PIN_CNF_DRIVE_S0H1, ///< !< Standard '0', high drive '1'
    NRF_GPIO_PIN_H0H1 = GPIO_PIN_CNF_DRIVE_H0H1, ///< !< High drive '0', high 'drive '1"
    NRF_GPIO_PIN_D0S1 = GPIO_PIN_CNF_DRIVE_D0S1, ///< !< Disconnect '0' standard '1'
    NRF_GPIO_PIN_D0H1 = GPIO_PIN_CNF_DRIVE_D0H1, ///< !< Disconnect '0', high drive '1'
    NRF_GPIO_PIN_S0D1 = GPIO_PIN_CNF_DRIVE_S0D1, ///< !< Standard '0'. disconnect '1'
    NRF_GPIO_PIN_H0D1 = GPIO_PIN_CNF_DRIVE_H0D1, ///< !< High drive '0', disconnect '1'
} nrf_gpio_pin_drive_t;
```

6.1.2 软件编写：

6.1.2.1 点亮第一个 LED 灯

按照前第 3 章里的介绍，首先建立一个工程项目，采用库函数来在驱动 IO 口首先要添加几个驱动库，如下图所示：



上图 lib 组下红色框框中的 led.c 文件都是我们需要编写驱动，为了方便在后面的工程中移植使用，单独编写一个驱动文件。后面的工程中，只需要编写 main.c 主函数调用就 OK，整个工程项目大家如果加入分层的思想那么就对之后的移植非常有利。打个比方:底层和应用程隔离。底层驱动和应用层无关，main.c 使用的函数在 led.c 驱动中已经写好，这些才和硬件有关，这是需要移植到不同硬件时，main 主函数是可以不做任何修改的，只需要修改和底层相关的 led.c 驱动。下面分析下 led .c 的驱动编写，先需要对 I/O 端口进行配置，配置成输入还是输出：

```
void LED_Init(void)
{
  nrf_gpio_cfg_output(LED_0);
  nrf_gpio_cfg_output(LED_1);
  nrf_gpio_cfg_output(LED_2);
  nrf_gpio_cfg_output(LED_3);
```

然后编写开灯和关灯的程序，比如 led1 灯，I/O 口输出低电平，则是开灯。输出高电平，则是关灯。电平变化，则是 LED 灯翻转。直接调用 nrf_gpio.h 中的库函数，代码如下：

```
void LED1_Open(void)//LED1 灯开灯
{
  nrf_gpio_pin_clear(LED_0);
}

void LED1_Close(void)//LED1 灯关灯
{
 nrf_gpio_pin_set(LED_0);
}
void LED1_Toggle(void)//LED1 灯翻转
{
```

那么主函数的编写就比较简单了，我们需要调用下面 2 个头文件，一个库函数 nrf_gpio.h 头，一个编写的驱动函数 led.h 头，才能够直接使用我们定义的子函数。如下使用 LED_Open()函数就能够点亮一个 LED 灯了，是不是很简单：

```
01. #include "nrf_gpio.h"
02. #include "led.h"
03.
04. int main(void)
05. {
06.     //初始化 led 灯
07.     LED_Init();
08.     while(true)
09.     {
10.         LED1_Open();
11.         LED2_Close();
12.     }
13. }
```

那么加入一个小的延迟 delay 函数和打开与关闭 LED 子函数相结合，就可以实现 LED 闪烁的功能了，可以直接调用官方库驱动 nrf_delay.h 中的延迟函数，函数如下所示：

```
01. #include "nrf_delay.h"
02. #include "nrf_gpio.h"
03. #include "led.h"
04.
05. int main(void)
06. {
07.     //初始化 led 灯
08.     LED_Init();
09.     while(true)
10.     {
11.         LED1_Open();
12.         LED2_Close();
13.         nrf_delay_ms(500); //延迟 500ms
14.         LED2_Open();
15.         LED1_Close();
16.         nrf_delay_ms(500);
17.     }
18. }
```

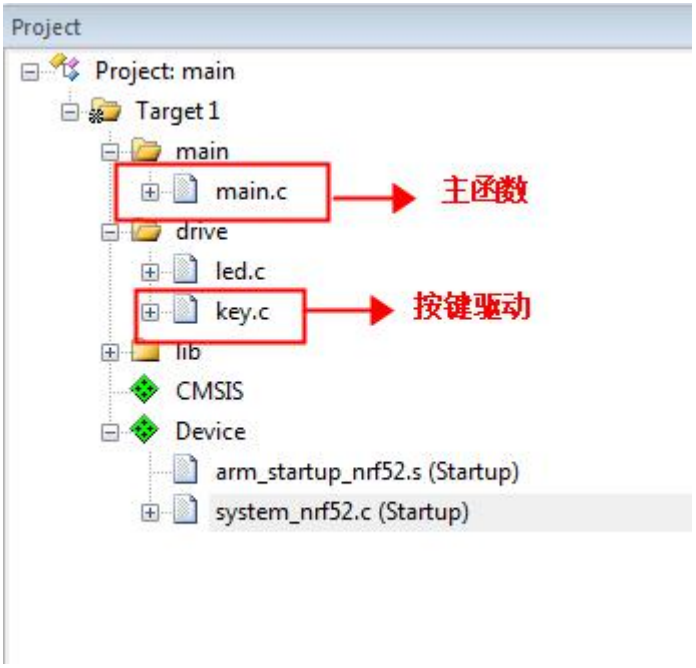
下载到青云 QY-nRF52832 蓝牙开发板上运行后的效果如下图所示，LED 开始闪烁：

6.2.1.2 按键输入扫描

下面我就来首先介绍下 nRF52832 的按键扫描控制方式。当 IO 管脚为低的时候可以判断管脚

已经按下。通过 key 的按下来控制 led 的亮灭。硬件上设计是比较简单的，这个普通的 MCU 的用法一致。

在代码文件中，实验二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：开发者只需要自己编写红色框框里的三个文件就 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 key.c 这个驱动子文件。

Key.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。第二：扫描判断按键是否有按下，按键扫描是通过 MCU 不停的判断端口的状态来实现的。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。下面看看代码：

```
01.  #include "key.h"
02. void KEY_Init(void)
03. {
04.    nrf_gpio_cfg_input(16,NRF_GPIO_PIN_PULLUP);//设置管脚位上拉输入
05.    nrf_gpio_cfg_input(17,NRF_GPIO_PIN_PULLUP);//设置管脚位上拉输入
06.  }
07.
08.  void Delay(uint32_t temp)
09. {
10.    for(; temp!= 0; temp--);
11. }
12.
13. uint8_t   KEY1_Down(void)
14. {
15.          /*检测是否有按键按下 */
16.       if( nrf_gpio_pin_read(KEY_1)== 0 )
17.    {
18.          /*延时消抖*/
```

```
19.          Delay(10000);
20.        if(nrf_gpio_pin_read(KEY_1)== 0 )
21.             {
22.                          /*等待按键释放 */
23.                          while(nrf_gpio_pin_read(KEY_1)== 0 );
24.                          return     0     ;
25.                      }
26.             else
27.                          return 1;
28.        }
29.  else
30.        return 1;
31. }
```

上面代码中 KEY_Init 函数首先进行 IO 管脚初始化，相关寄存器在上一节已经谈过，这里我们设置时要注意，开发板没有接上拉电阻提高管脚的驱动能力，因此设置的时候最好把管脚设置为带上拉的输入类型，nrf_gpio_cfg_input 函数在官方给出的库函数里给了定义。

按键扫描时通过函数 nrf_gpio_pin_read 读取管脚状态，判断 IO 是否被拉低，如果被拉低就可以说明按键被按下了。同时为了防止按键抖动，加入一个软件延迟函数去抖。

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控制翻转 IO 口，LED 灯指示相应的变化。函数如下所示：

```
01. #include "nrf52.h"
02. #include "nrf_gpio.h"
03. #include "led.h"
04. #include "key.h"
05.
06. int main(void)
07. {
08.    LED_Init();//led 初始化
09.    KEY_Init();//按键初始化
10.
11.    while(1)
12.    {
13.        if( KEY1_Down()== 0)//判定按键是否按下
14.  {
15.         LED_Toggle();
16.  }
17.    }
18. }
```

实验编译后下载到青云 nRF52832 开发板后的实验现象如下：下载后按下按键 1，LED 灯翻转。

6.2 GPIOTE 与外部中断

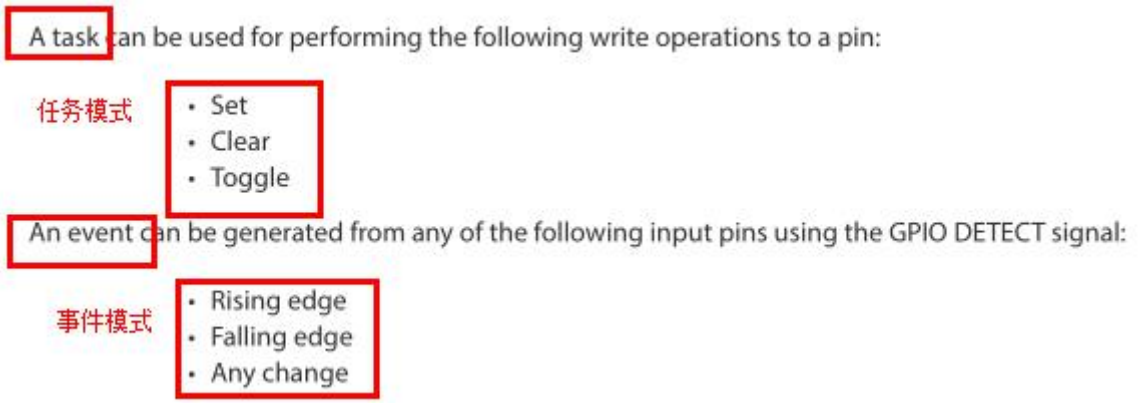
6.2.1 原理分析

按键的输入其实就是对 GPIO 口进行操作，同时引入了外部中断的概念。实际上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。

GPIOET 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。整个 GPIOTE 寄存器的个数也是非常少的，如下图所示：

Registers	Offset	Description
TASKS		
OUT[0]	0x000	Task for writing to pin specified by PSEL in CONFIG[0].
OUT[1]	0x004	Task for writing to pin specified by PSEL in CONFIG[1].
OUT[2]	0x008	Task for writing to pin specified by PSEL in CONFIG[2].
OUT[3]	0x00C	Task for writing to pin specified by PSEL in CONFIG[3].
EVENTS		
IN[0]	0x100	Event generated from pin specified by PSEL in CONFIG[0].
IN[1]	0x104	Event generated from pin specified by PSEL in CONFIG[1].
IN[2]	0x108	Event generated from pin specified by PSEL in CONFIG[2].
IN[3]	0x10C	Event generated from pin specified by PSEL in CONFIG[3].
PORT	0x17C	Event generate from multiple input pins.
REGISTERS		
INTENSET	0x304	Interrupt enable set register.
INTENCLR	0x308	Interrupt enable clear register.
CONFIG[0]	0x510	Configuration for OUT[0] task and IN[0] event.
CONFIG[1]	0x514	Configuration for OUT[1] task and IN[1] event.
CONFIG[2]	0x518	Configuration for OUT[2] task and IN[2] event.
CONFIG[3]	0x51C	Configuration for OUT[3] task and IN[3] event.

任务模式配置

事件模式配置通道

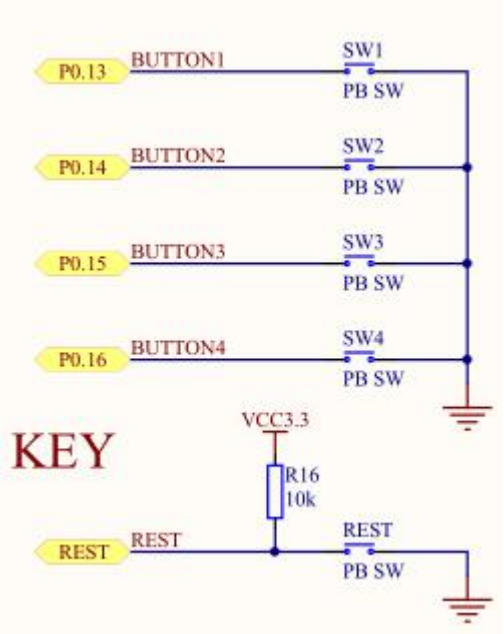
使能中断寄存器

模式详细配置寄存器

TASKS 任务通过 OUT[0]~OUT[3]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。

GPIOTE 模块主要提供的了 4 个通道，四个通道都是通过 CONFIG[0]~CONFIG[3]来配置，这四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，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 产生时是触发中断。这样就实现了 GPIO 的中断方式。

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），可以通过按键输入来控制 led 的亮灭。



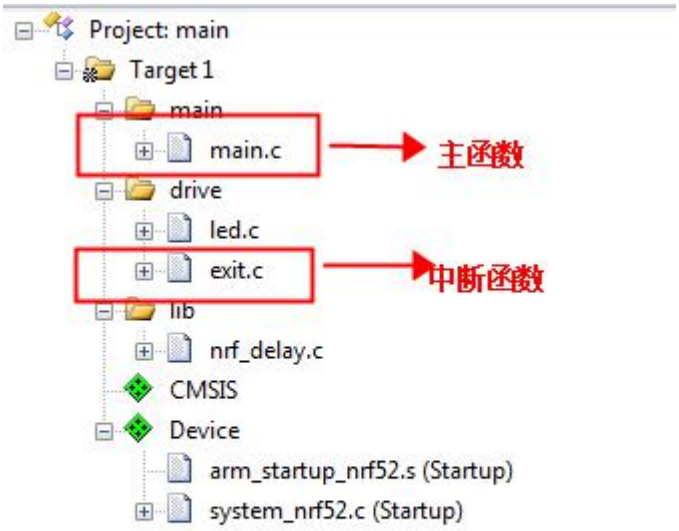
6.2.2 应用实例编写

6.2.2.1 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 GPIOET 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，任务模式后面有专门的历程进行讨论。在 CONFIG 这个寄存器里详细的进行了事件模式的配置，如下图所示，三个红色框框里的寄存器位我们需要进行配置：

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

ID	RW	Field	Value ID	Value	Description
A	RW	MODE			Mode
			DISABLED	0	Disabled. Pin specified by PSEL will not be acquired by the GPIOTE module.
			EVENT	1	Event mode. The pin specified by PSEL will be configured as an input and the IN[n] event will be generated if operation specified in POLARITY occurs on the pin.
			TASK	3	Task mode. The pin specified by PSEL will be configured as an output and triggering the OUT[n] task will perform the operation specified by POLARITY on the pin. When enabled as a task the GPIOTE module will acquire the pin and the pin can no longer be written as a regular output pin from the GPIO module.
B	RW	PSEL		[0..31]	Pin number associated with OUT[n] task and IN[n] event.
C	RW	POLARITY			When in task mode: Operation to be performed on output when OUT[n] task is triggered. When In event mode: Operation on input that shall trigger IN[n] event.
			LOTOHI	1	Task mode: Set pin from OUT[n] task. Event mode: Generate IN[n] event when rising edge on pin.
			HITOLO	2	Task mode: Clear pin from OUT[n] task. Event mode: Generate IN[n] event when falling edge on pin.
			TOGGLE	3	Task mode: Toggle pin from OUT[n]. Event mode: Generate IN[n] when any change on pin.
D	RW	OUTINIT			When in task mode: Initial value of the output when the GPIOTE channel is configured.
					When in event mode: No effect.
			LOW		Task mode: Initial value of pin before task triggering is low.
			HIGH		Task mode: Initial value of pin before task triggering is high.

下面看看代码：

```
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
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。

```
void GPIOTE_IRQHandler(void)
{

    if ((NRF_GPIOTE->EVENTS_IN[0] == 1) &&
```

```
05.             (NRF_GPIOTE->INTENSET & GPIOTE_INTENSET_IN0_Msk))
06.       {
07.             NRF_GPIOTE->EVENTS_IN[0] = 0; //中断事件清零.
08.       }
09.     LED_Toggle();//led 灯翻转
10. }
```

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 2014 青风电子 ********************
02.  * 文件名  ：main
03.  * 描述    ：
04.  * 实验平台：青云 nRF52832 蓝牙开发板
05.  * 描述    ：按键中断
06.  * 作者    ：青风
07.  * 店铺    ：qfv5.taobao.com
08. *********************************************************************************/
09. #include "nrf51.h"
10. #include "nrf_gpio.h"
11. #include "exit.h"
12. #include "led.h"
13.
14. int main(void)
15. {
16.  LED_Init();
17.   LED_Open();
18.  /*config key*/
19.  EXIT_KEY_Init();
20.  while(1)
21.  {
22.       }
23. }
```

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，按下按键后 led 灯不停翻转。

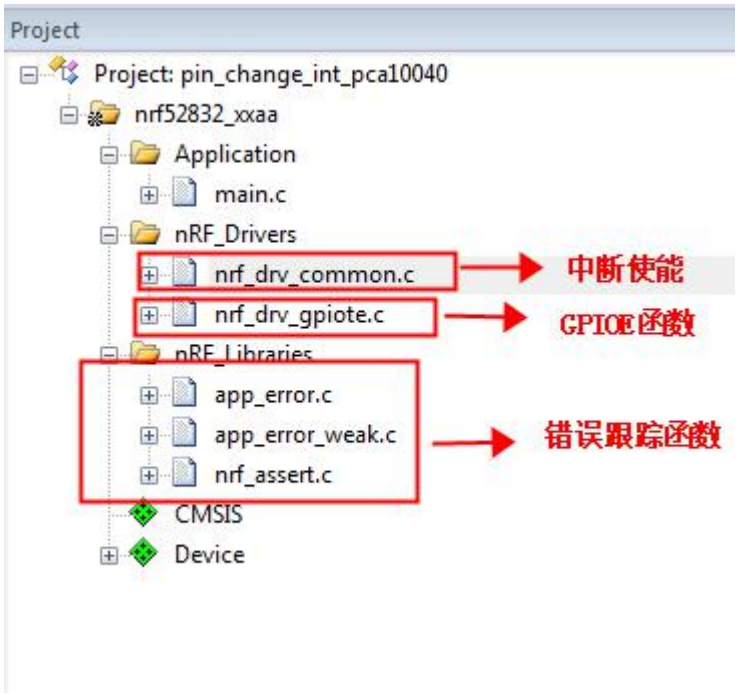
6.2.2.2 GPIOTE 组件的应用

在官方 SDK9 以及之后的版本中，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 pac10400，如下图所示：



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，总结其设计步骤：

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：

```
ret_code_t nrf_drv_gpiote_init(void)
{
    if (m_cb.state != NRF_DRV_STATE_UNINITIALIZED)
    {
        return NRF_ERROR_INVALID_STATE;
    }

    uint8_t i;
    for (i = 0; i < NUMBER_OF_PINS; i++)
    {
        pin_in_use_clear(i);
    }
    for (i = 0; i < (NUMBER_OF_GPIO_TE+GPIOTE_CONFIG_NUM_OF_LOW_POWER_EVENTS); i++)
    {
        channel_free(i);
    }
                                                    设置中断优先级
    nrf_drv_common_irq_enable(GPIOTE_IRQn, GPIOTE_CONFIG_IRQ_PRIORITY);
    nrf_gpiote_int_enable(GPIOTE_INTENSET_PORT_Msk);   使能中断类型为PORT中断
    m_cb.state = NRF_DRV_STATE_INITIALIZED;

    return NRF_SUCCESS;
}
```

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：

```
//配置设置 GPIOE 输出参数 从低电平到高电平
    nrf_drv_gpiote_out_config_t out_config = GPIOTE_CONFIG_OUT_SIMPLE(0);
    //GPIOE 输出初始化
    err_code = nrf_drv_gpiote_out_init(LED_1, &out_config);
    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：

```
//配置设置 GPIOE 输入参数
    nrf_drv_gpiote_in_config_t in_config = GPIOTE_CONFIG_IN_SENSE_TOGGLE(1);
    in_config.pull = NRF_GPIO_PIN_PULLUP;
     //GPIOE 输入初始化，设置触发输入中断
    err_code = nrf_drv_gpiote_in_init(BUTTON_1, &in_config, in_pin_handler);
    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：

```
//设置 GPIOE 输入事件使能
    nrf_drv_gpiote_in_event_enable(BUTTON_1, 0);
```

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。

```
 void in_pin_handler(nrf_drv_gpiote_pin_t pin, nrf_gpiote_polarity_t action)//中断回调函数
{
    if(nrf_gpio_pin_read(PIN_IN)== 0)//按键防抖
  {
 nrf_drv_gpiote_out_toggle(PIN_OUT);
  }
}
```

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：

```
24.
25. /**
26.  *主函数，初始化后循序等待
27.  */
28. int main(void)
29. {
30.     gpio_init();
31.
32.     while (true)
33.     {
34.
35.     }
36. }
```

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。

6.3 定时器 TIME

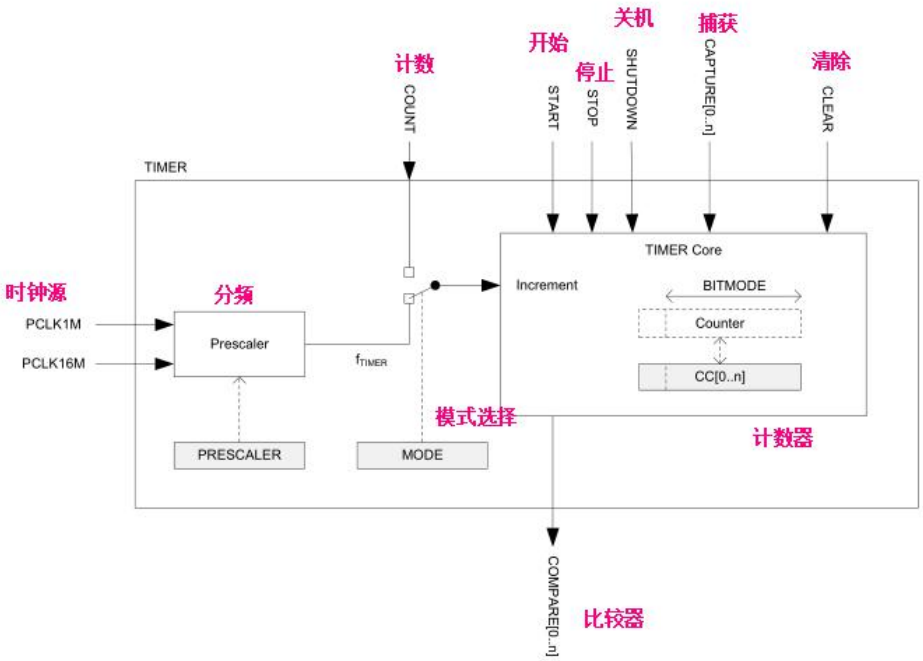
6.3.1 原理分析

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:TIMER0、TIMER1、TIMER2、TIMER3、TIMER4，如下表所示。

Base address	Peripheral	Instance	Description	Configuration
0x40008000	TIMER	TIMER0	Timer 0	This timer instance has 4 CC registers (CC[0..3])
0x40009000	TIMER	TIMER1	Timer 1	This timer instance has 4 CC registers (CC[0..3])
0x4000A000	TIMER	TIMER2	Timer 2	This timer instance has 4 CC registers (CC[0..3])
0x4001A000	TIMER	TIMER3	Timer 3	This timer instance has 6 CC registers (CC[0..5])
0x4001B000	TIMER	TIMER4	Timer 4	This timer instance has 6 CC registers (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，该寄存器位于下图计数器内部。如上表所示：可选位宽为 8、16、24 和 32 位：

Bit number				31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Id				A A
Reset 0x00000000				0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Id	**RW**	**Field**	**Value Id**	**Value**	**Description**
A	RW	BITMODE			Timer bit width
			16Bit	0	16 bit timer bit width
			08Bit	1	8 bit timer bit width
			24Bit	2	24 bit timer bit width
			32Bit	3	32 bit timer bit width



上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 为的分频器（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。

Bit number				31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Id																																A	A	A	A
Reset				0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
Id	RW	Field	Value Id	Value								Description																							
A	RW	PRESCALER		[0..9]								Prescaler value																							

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。

Bit number				31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Id																																			A
Reset				0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Id	RW	Field	Value Id	Value								Description																							
A	RW	MODE																																	
			Timer	0								Select Timer mode																							
			Counter	1								Select Counter mode																							

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。

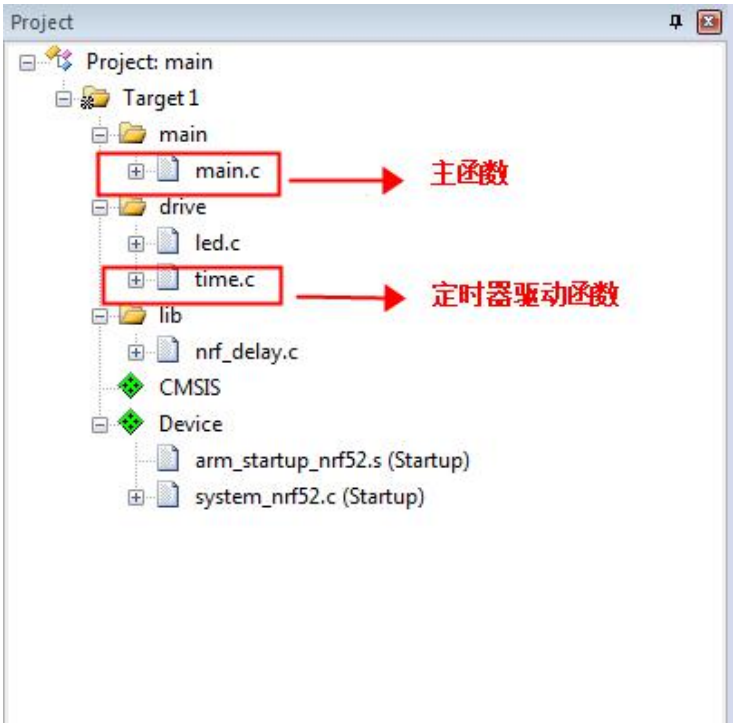
计数模式下，通过设定一个 CAPTURE Task，当计数器与比较/捕获寄存器设定的值（Timer value）相等的时候，将产生捕获 CAPTURE[n] event。捕获的值会被存储到 CC[n]寄存器内读取。

COMPARE [n] event 和 CAPTURE[n] event 都可以触发中断，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。

6.3.2 应用实例编写

6.3.2.1 定时器定时

在代码文件中，实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 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。

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：

Register	Offset	Description
TASKS		
START	0x000	Start Timer
STOP	0x004	Stop Timer
COUNT	0x008	Increment Timer (Counter mode only)
CLEAR	0x00C	Clear timer
CAPTURE[0]	0x040	Capture Timer value to CC0 register
CAPTURE[1]	0x044	Capture Timer value to CC1 register
CAPTURE[2]	0x048	Capture Timer value to CC2 register
CAPTURE[3]	0x04C	Capture Timer value to CC3 register
EVENTS		
COMPARE[0]	0x140	Compare event on CC[0] match
COMPARE[1]	0x144	Compare event on CC[1] match
COMPARE[2]	0x148	Compare event on CC[2] match
COMPARE[3]	0x14C	Compare event on CC[3] match
REGISTERS		
SHORTS	0x200	Shortcuts
INTENSET	0x304	Write-only - configures which events generate a Timer interrupt
INTENCLR	0x308	Write-only - configures which events do not generate a Timer interrupt
MODE	0x504	Timer mode selection
BITMODE	0x508	Configure the number of bits used by the TIMER
PRESCALER	0x510	Timer prescaler register
CC[0]	0x540	Capture/Compare register 0
CC[1]	0x544	Capture/Compare register 1
CC[2]	0x548	Capture/Compare register 2
CC[3]	0x54C	Capture/Compare register 3

上面红色框框的几个寄存器大家要注意了，这个是下面设置要用的几个寄存器，英文解释很清楚，关键是搞清楚怎么用，我们之间对着代码段分析：

```
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。我们设置为 16bit。

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 行，第 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*31.25 后为 1ms，那么 CC[n]存放定时时间*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。

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。

```
    switch (timer)
    {
        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：

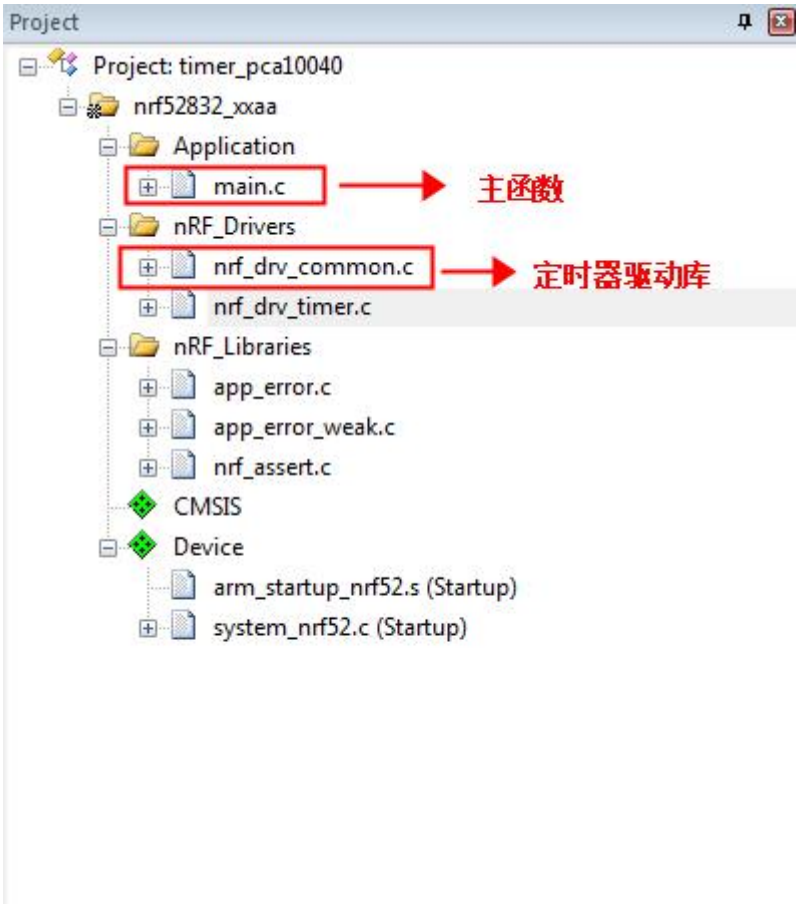
```
//******************** (C) COPYRIGHT 2017 青风电子 ********************
* 文件名  ：main
* 描述    ：
* 实验平台：青云 nRF52832 开发板
* 描述    ：定时器定时
* 作者    ：青风
* 店铺    ：qfv5.taobao.com
**********************************************************************/
#include "nrf52.h"
#include  "led.h"
#include  "time.h"

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();
```

```
24.             // 使用定时器 1 产生 1s 定时
25.             nrf_timer_delay_ms(TIMER1, TIMER_DELAY_MS);
26.
27.             LED1_Toggle();
28.             // 使用定时器 2 产生 1s 定时
29.             nrf_timer_delay_ms(TIMER2, TIMER_DELAY_MS);
30.         }
31. }
```

实验下载到青云 nRF52832 开发板后的实验现象如下，参数 1s 的定时闪烁。

6.3.2.2 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，但是缺点是程序比较长，功能没有库函数完整。组件库编程的核心就是理解主机库函数，在调用组件库函数时首先需要弄清其函数定义。工程建立如下：



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：

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 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：

● ret_code_t nrf_drv_timer_init(nrf_drv_timer_t const * const p_instance,
nrf_drv_timer_config_t const * p_config,
nrf_timer_event_handler_t timer_event_handler);

* 参数 p_instance 前面选择的定时器 0 或者 1、2
* 参数 p_config 初始化配置.如果没有配置，默认下则设置为 NULL.
* 参数 timer_event_handler 定时器回调中断声明.

重点说明下参数 p_config ，这个参数是 nrf_drv_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：

```
typedef struct
{
    nrf_timer_frequency_t   frequency;          /**<频率. */
    nrf_timer_mode_t        mode;               /**< 模式选择 */
    nrf_timer_bit_width_t   bit_width;          /**< 位宽 */
    uint8_t                 interrupt_priority; /**< 定时器中断优先级*/
    void*                   p_context;          /**< 上下文传递参数*/
} nrf_drv_timer_config_t;
```

如果结构体不设置，就默认为初始设置。如何需要设置，这对结构体赋值，比如广泛给的默认配置：

```
#define TIMER0_CONFIG_FREQUENCY     NRF_TIMER_FREQ_16MHz
#define TIMER0_CONFIG_MODE          TIMER_MODE_MODE_Timer
#define TIMER0_CONFIG_BIT_WIDTH     TIMER_BITMODE_BITMODE_32Bit
#define TIMER0_CONFIG_IRQ_PRIORITY  APP_IRQ_PRIORITY_LOW
```

设置 16M 高速时钟，定时器模式，32bit 位宽，中断优先级为低。

设置定时器的滴答时间，这个函数定义了定时器定时 ms 的参数，和前面寄存器设置配置类似：

● uint32_t nrf_drv_timer_ms_to_ticks(nrf_drv_timer_t const * const p_instance,
uint32_t timer_ms);

*参数 p_instance 前面选择的定时器 0 或者 1、2
*参数 timer_ms 定时多少 ms

设置定时器捕获/比较触发设备、通道、滴答时间等参数：

- void nrf_drv_timer_extended_compare(nrf_drv_timer_t const * const p_instance,
  nrf_timer_cc_channel_t cc_channel,
  uint32_t cc_value,
  nrf_timer_short_mask_t timer_short_mask,
  bool enable);

  * 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

最后来使能定时器 0，打开定时器开始计时。

```
nrf_drv_timer_enable(&TIMER_LED);
```

具体的初始化步骤如下，调用上面介绍的组件库函数，作为主函数：

```
01.     //配置定时器，同时注册定时器的回调函数
02.     err_code = nrf_drv_timer_init(&TIMER_LED, NULL, timer_led_event_handler);
03.     APP_ERROR_CHECK(err_code);
04.     //设置定时器的滴答时间
05.     time_ticks = nrf_drv_timer_ms_to_ticks(&TIMER_LED, time_ms);
06.     //设置定时器捕获/比较 触发设备、通道、滴答时间
07.     nrf_drv_timer_extended_compare(
08.             &TIMER_LED, NRF_TIMER_CC_CHANNEL0, time_ticks,
    NRF_TIMER_SHORT_COMPARE0_CLEAR_MASK, true);
09.     //使能定时器
10.     nrf_drv_timer_enable(&TIMER_LED);
```

在主函数最后循环等待定时器中断返回，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，产生一个回调中断处理，来执行 LED 灯的翻转。

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

```
17.     }
18. }
```

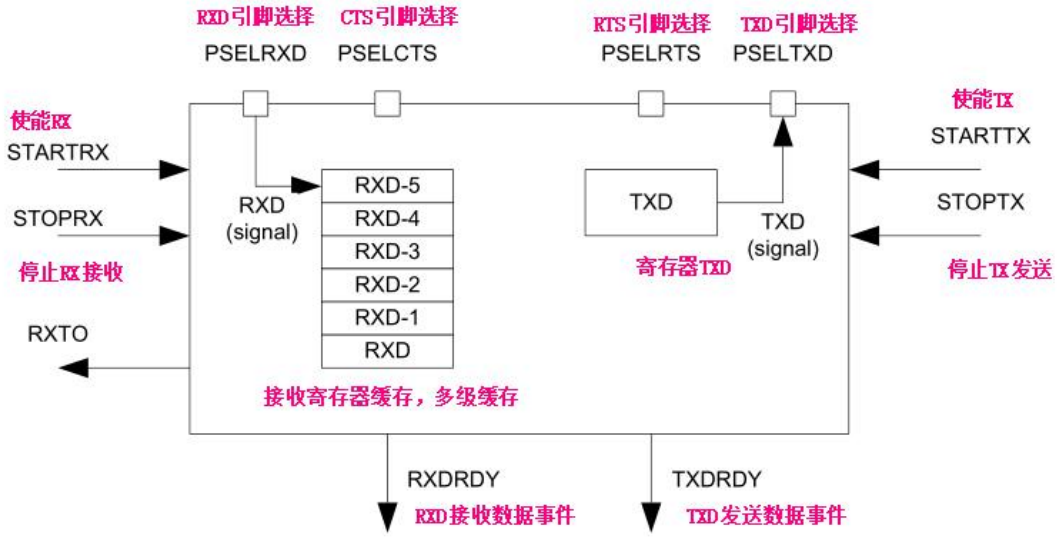
实验整体思路和寄存器配置方式相同，编写后把实验程序下载到青云 nRF52832 开发板后的实验现象如下，参数 1s 的定时闪烁。对于更多组件函数的定义请大家详细阅读组件源代码已进一步了解

6.4 串口外设应用

6.4.1 原理分析

串口 UART 也称为通用异步收发器。是各种处理器中常用了通信接口，在 nRF52 芯片中，UART 具有以下特点：

- 全双工操作
- 自动流控
- 奇偶校验产生第 9 位数据



◆硬件配置：

根据 PSELRXD、PSELCTS、PSELRTS 和 PSELTXD 寄存器的配置可以相应的将 RXD、CTS(发送清除、低有效)、TXD、RTS（发送请求、低有效）映射到物理的引脚上。如果这些寄存器的任意一个设为 0xffffffff，相关的 UART 信号就不会连接到任务物理引脚上。这四个寄存器及其配置只能在 UART 使能时可用，可以在芯片为系统 ON 模式时保持。系统 OFF 模式下，为了保证 UART 上的引脚信号电平正确，UART 的引脚必须在 GPIO 外设中按照下表进行配置：

◆UART 挂起：

UART 串口可以通过触发 SUPSPEND 寄存器任务来挂起。SUPSPEND 将会影响 UART 发送器和 UART 接收器，设置后使发生器停止发送，接收器停止接收。在 UART 挂起后，通过相应的触发 STARTTX 和 STARTRX 就可以重新开启发送和接收。

当触发 SUPSPEND 任务时，UART 接收器和触发 STOPRX 任务一样的工作。

在触发 SUPSPEND 任务后，正在进行的 TXD 字节传输将在 UART 挂起前完成。

◆错误条件：

在一个错误帧出现的情况下，如果再此帧中没有检测到有效的停止位会产生一个 ERROR 事件。另外，在中断时，如果 RXD 保持低电平超过一个数据帧长度时，也会产生一个 ERROR 事件。

◆流量控制：

nRF52 芯片的 UART 可以分为带流量控制和不带流量控制两种方式。不带流量控制时，不需要连接 CTS 和 RTS 两个管脚，可以视为两个管脚一值有效。

带流量控制时，RTS 引脚作为输出，由 UART 硬件模块自动控制。与接收寄存器的多级硬件缓冲 Buff 协调工作。比如在硬件缓冲已经接收满了 6 个字节的时，RTS 引脚就输出高电平的终止信号，当缓冲中的数据都被读出后回复有效信号（低电平）。

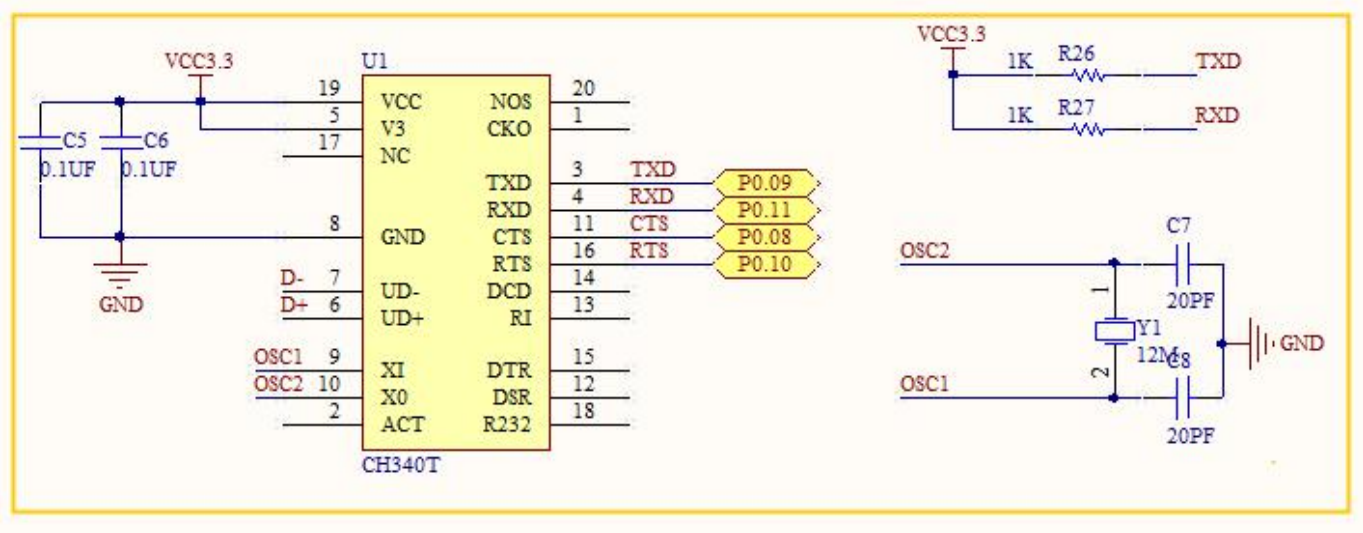
CTS 作为输入由外部输入。当 CTS 有效时（低电平）模块可以发送，当为无效时，模块自动暂停发送，并在 CTS 恢复有效时继续发送。

那么当 uart 模块的 rts 与 cts 交叉相接。 如果发送方发送太快，当接收方的接收硬件 buff 已经存满了字节后，接收方自动无效 RTS 信号，表示不能接收了。因为接收方 RTS 与发送方 CTS 相接，使得发送方的 CTS 也编程无效信号，于是发送方自动停止发送。这样就保证了接收方不会接收溢出。流量控制也就是体现在这里。

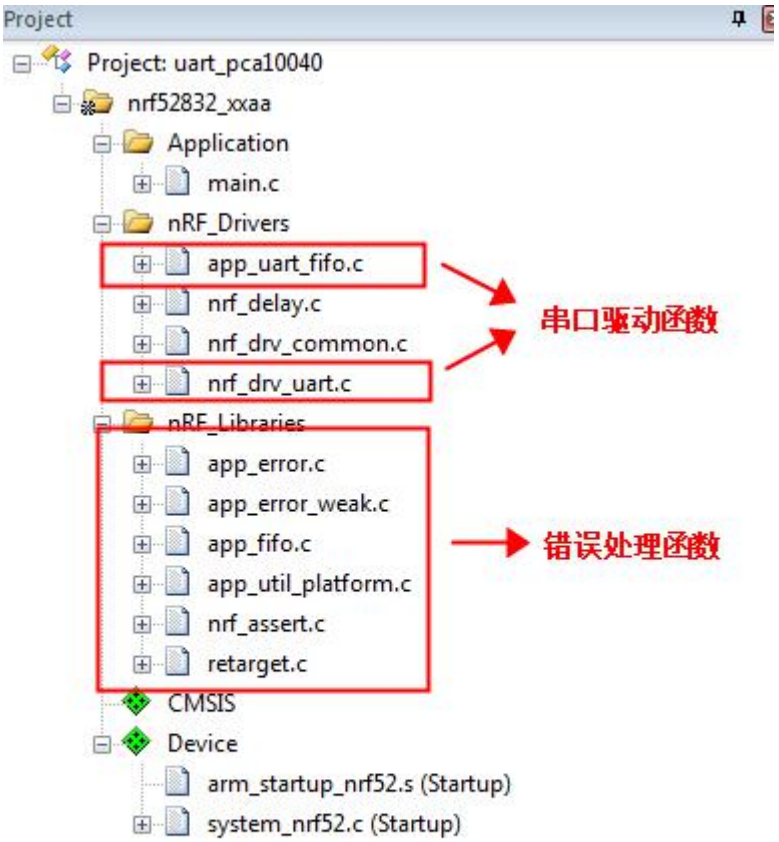
6.4.2 应用实例编写

6.4.2.1 串口 printf 输出

硬件连接方面，通过高质量芯片 CH340T 把串口信号转换成 usb 输出， TXD--接 P0.09 端 RXD--接 P0.11 端 CTS--接 P0.08 端 RTS--接 P0.10 端其中数据口 TXD 和 RXD 分别接 1K 电阻上拉，提高驱动能力，如下图所示。



在代码文件中，实验建立了一个演示历程，我们还是采用分层思想，直接通过官方提供的组件库进行编程。打开文件夹中的 uart 工程，添加 UART 的组件库如下图所示：



如上图所示：nrf_drv_uart.c 文件和 app_uart_fifi.c 文件是官方编写好的驱动库文件。这两个文件在后面编写带协议栈的 BLE 应用时，是可以直接用于其中配置串口功能的。因此理解这两个文件提供的 API 函数是利用组件编写串口的必由之路。

在 app_uart.h 文件中提供了两个关键的 uart 的初始化函数 APP_UART_FIFO_INIT 和 APP_UART_INIT，一个是带 FIFO 缓冲的初始化串口函数，一个是不带 FIFO 缓冲的初始化函数，一般情况下使用带软件缓冲的 FIFI 的函数，减小数据溢出错误的发生几率。配置代码具体如下：

```
01.    APP_UART_FIFO_INIT(&comm_params,        //串口参数
02.                       UART_RX_BUF_SIZE,//RX 缓冲大小
03.                       UART_TX_BUF_SIZE,//TX 缓冲大小
04.                       uart_error_handle,//错误处理
05.                       APP_IRQ_PRIORITY_LOW,//中断优先级
06.                       err_code);//配置串口
```

这个函数配置参数较多，首先是&comm_params 参数，这个参数配置了串口管脚，流量控制，波特率等关键参数。这些参数通过一个结构体进行定义，参数作为结构体的内容，代码如下所示：

```
07.        const app_uart_comm_params_t comm_params =
08.          {
09.              RX_PIN_NUMBER,
10.              TX_PIN_NUMBER,
11.              RTS_PIN_NUMBER,
```

```
12.             CTS_PIN_NUMBER,//串口管脚配置
13.             APP_UART_FLOW_CONTROL_DISABLED,//流控设置
14.             false,
15.             UART_BAUDRATE_BAUDRATE_Baud115200//波特率
16.         };//设置串口参数
17.
```

RX_PIN_NUMBER、TX_PIN_NUMBER、RTS_PIN_NUMBER、CTS_PIN_NUMBER 四个配置串口的硬件管脚，带流控下四个管脚都需要宏定义 IO 端口。如果是不带流控的模式下，只需要配置 RX_PIN_NUMBER 和 RTX_PIN_NUMBER 两个管脚。

设置 APP_UART_FLOW_CONTROL_DISABLED 关掉流控。

设置中断优先级为低。

设置 UART_BAUDRATE_BAUDRATE_Baud115200 波特率为 115200。

那么这些参数如何赋值给串口的寄存器？这就需要深入 APP_UART_FIFO_INIT 内部研究了，进入函数内部：

```
01. #define APP_UART_FIFO_INIT(P_COMM_PARAMS, RX_BUF_SIZE, TX_BUF_SIZE,
    EVT_HANDLER, IRQ_PRIO, ERR_CODE) \
02.     do
03.     {
04.         app_uart_buffers_t buffers;
05.         static uint8_t          rx_buf[RX_BUF_SIZE];
06.         static uint8_t          tx_buf[TX_BUF_SIZE];
07.         buffers.rx_buf          = rx_buf;
08.         buffers.rx_buf_size = sizeof (rx_buf);
09.         buffers.tx_buf          = tx_buf;
10.         buffers.tx_buf_size = sizeof (tx_buf);
11.         ERR_CODE = app_uart_init(P_COMM_PARAMS, &buffers, EVT_HANDLER,
    IRQ_PRIO);
12.     }
```

函数内部申请两个软件 BUF 缓冲空间提供给 RX 和 TX。然后调用函数 app_uart_init 进行串口的初始化。继续深入 app_uart_init 函数内部，这段函数是官方给的驱动组件库，读者不需要单独配置或者修改，只需要直接调用：

```
01. uint32_t app_uart_init(const app_uart_comm_params_t * p_comm_params,
02.                                     app_uart_buffers_t *        p_buffers,
03.                                     app_uart_event_handler_t event_handler,
04.                                     app_irq_priority_t          irq_priority)
05. {
06.     uint32_t err_code;
07.
08.     m_event_handler = event_handler;
09.
10.     if (p_buffers == NULL)
11.     {
12.         return NRF_ERROR_INVALID_PARAM;
```

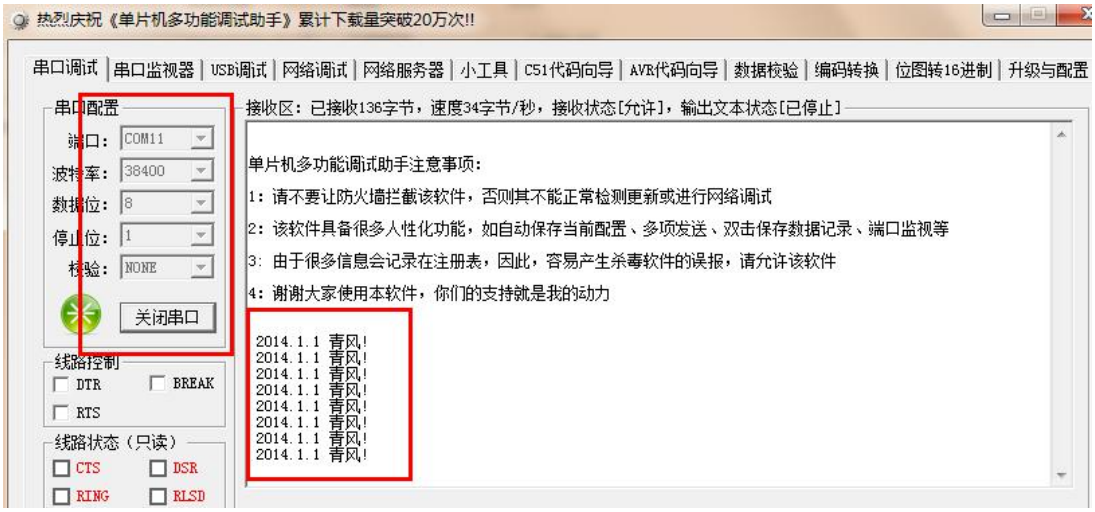
```
13.         }
14.
15.         // 配置 RX buffer 缓冲.
16.         err_code = app_fifo_init(&m_rx_fifo, p_buffers->rx_buf, p_buffers->rx_buf_size);
17.         if (err_code != NRF_SUCCESS)
18.         {
19.             // Propagate error code.
20.             return err_code;
21.         }
22.
23.         // 配置 TX buffer 缓冲
24.         err_code = app_fifo_init(&m_tx_fifo, p_buffers->tx_buf, p_buffers->tx_buf_size);
25.         if (err_code != NRF_SUCCESS)
26.         {
27.             // Propagate error code.
28.             return err_code;
29.         }
30.       //配置串口的管脚，波特率，流控等特性
31.         nrf_drv_uart_config_t config = NRF_DRV_UART_DEFAULT_CONFIG;
32.         config.baudrate = (nrf_uart_baudrate_t)p_comm_params->baud_rate;
33.         config.hwfc = (p_comm_params->flow_control
34.           ==APP_UART_FLOW_CONTROL_DISABLED) ?
35.                     NRF_UART_HWFC_DISABLED : NRF_UART_HWFC_ENABLED;
36.         config.interrupt_priority = irq_priority;
37.         config.parity = p_comm_params->use_parity ? NRF_UART_PARITY_INCLUDED :
    NRF_UART_PARITY_EXCLUDED;
38.         config.pselcts = p_comm_params->cts_pin_no;
39.         config.pselrts = p_comm_params->rts_pin_no;
40.         config.pselrxd = p_comm_params->rx_pin_no;
41.         config.pseltxd = p_comm_params->tx_pin_no;
42.
43.         err_code = nrf_drv_uart_init(&config, uart_event_handler);
44.
45.         if (err_code != NRF_SUCCESS)
46.         {
47.             return err_code;
48.         }
49.         nrf_drv_uart_rx_enable();
50.         return nrf_drv_uart_rx(rx_buffer,1);
51. }
```

那么主函数就是十分的简单了，直接调用我们串口的驱动库函数进行配置，输出采用 printf 函数打印输出，代码如下所示：

```
01. /******************** (C) COPYRIGHT 2017 青风电子 ********************
```

```
02.  * 文件名   ：main
03.  * 实验平台：青云 nRF52832 开发板
04.  * 描述     ：串口输出
05.  * 作者     ：青风
06.  * 店铺     ：qfv5.taobao.com
07. ********************************************************************/
08. int main(void)
09. {
10.     LEDS_CONFIGURE(LEDS_MASK);
11.     LEDS_OFF(LEDS_MASK);
12.     uint32_t err_code;
13.     const app_uart_comm_params_t comm_params =
14.       {
15.           RX_PIN_NUMBER,
16.           TX_PIN_NUMBER,
17.           RTS_PIN_NUMBER,
18.           CTS_PIN_NUMBER,//串口管脚配置
19.           APP_UART_FLOW_CONTROL_DISABLED,//流控设置
20.           false,
21.           UART_BAUDRATE_BAUDRATE_Baud115200//波特率
22.       };//设置串口参数
23.
24.     APP_UART_FIFO_INIT(&comm_params,      //串口参数
25.                        UART_RX_BUF_SIZE,//RX 缓冲大小
26.                        UART_TX_BUF_SIZE,//TX 缓冲大小
27.                        uart_error_handle,//错误处理
28.                        APP_IRQ_PRIORITY_LOW,//中断优先级
29.                        err_code);//配置串口
30.
31.     APP_ERROR_CHECK(err_code);
32.
33.   while (1)
34.     {
35.       LEDS_INVERT(LEDS_MASK);
36.      printf(" 2017.10.1  青风!\r\n");//答应输出
37.        nrf_delay_ms(500);
38.    }
39. }
```

实验下载到青云 nRF52832 开发板后连接 usb 转串口端，如下图所示，然后打开串口调试助手，实验现象如下，指示灯闪亮表示串口输出进行中，打开串口调试助手，输出要求的内容：



6.4.2.2 串口输入与回环

上一节通过 printf 打印输出数据，这一节讲使用官方组件库来输入和输出数据。对应串口输出和输入，官方的组件库提供了两个函数，分别为:

```
uint32_t app_uart_get(uint8_t * p_byte);
/**@功能为从 UART 串口获取数据.
 *
 * @details This function will get the next byte from the RX buffer. If the RX buffer is empty
 *          an error code will be returned and the app_uart module will generate an event upon
 *          reception of the first byte which is added to the RX buffer.
 *
 * @参数：输出 p_byte                    指针指向下一个接收字节存放的地址。
 *返回值：
 * @参数 NRF_SUCCESS                    如果收到成功收到字节，则返回成功。
 * @参数 NRF_ERROR_NOT_FOUND  如果再 RX buffer 中没有发现收到数据，则返回 NRF_ERROR_NOT_FOUND。
 */
```

```
uint32_t app_uart_put(uint8_t byte);
 /**@功能：通过串口输出字节
  *
  * @参数[in] byte     串口传输的字节
  *  返回值：
  * @参数  NRF_SUCCESS       如果通过 TX 缓冲把字节发送出去，则返回成功。
  * @参数  NRF_ERROR_NO_MEM     如果在 TX 缓冲中没有更多的空间。常用在流控控制中。
  * @参数 NRF_ERROR_INTERNAL     如果串口驱动报错。
  */
```

使用上面两个函数，下面写一个串口回环测试，通过串口助手发送数据，通过串口接收后，发回串口助手。下面串口回环测试代码如下：

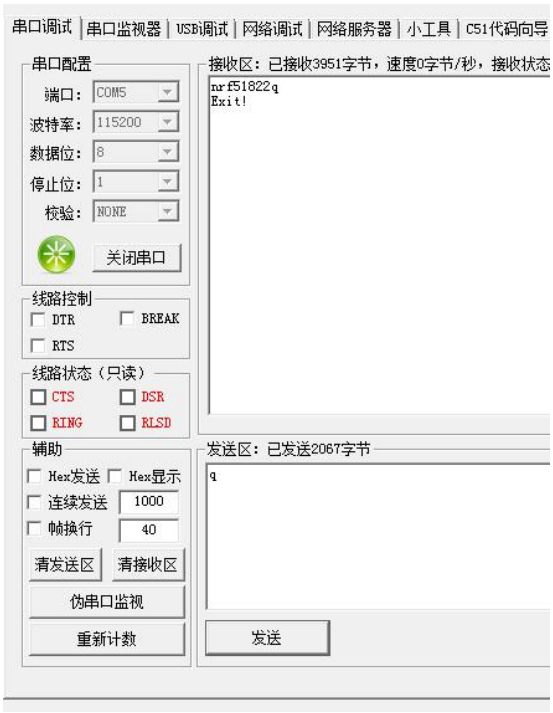
```
40. static void uart_loopback_test()
41. {
42. 
43.     while (true)
44.         {
45.             uint8_t cr;
46.             while(app_uart_get(&cr) != NRF_SUCCESS);//接收数据
47.             while(app_uart_put(cr) != NRF_SUCCESS);//把接收的数据发出
48. 
49.             if (cr == 'q' || cr == 'Q')
50.             {
51.                 printf(" \n\rExit!\n\r");//结束
52.                 while (true)
53.                 {
54.                     // Do nothing.
55.                 }
56.             }
57.         }
58. 
59. }
```

主函数的编写主要就是配置串口参数，设串口管脚，比特率，流控等参数设置：

```
60. int main(void)
61. {
62.         LEDS_CONFIGURE(LEDS_MASK);
63.         LEDS_OFF(LEDS_MASK);
64.         uint32_t err_code;
65.         const app_uart_comm_params_t comm_params =
66.             {
67.                 RX_PIN_NUMBER,//RX 管脚
68.                 TX_PIN_NUMBER,//TX 管脚
69.                 RTS_PIN_NUMBER,//RTS 管脚
70.                 CTS_PIN_NUMBER,//CTS 管脚
```

```
71.             APP_UART_FLOW_CONTROL_DISABLED,//关掉流控
72.             false,
73.             UART_BAUDRATE_BAUDRATE_Baud115200//设置比特率
74.         };
75.
76.     APP_UART_FIFO_INIT(&comm_params,
77.                        UART_RX_BUF_SIZE,//设置 RX 缓冲
78.                        UART_TX_BUF_SIZE,//设置 TX 缓冲
79.                        uart_error_handle,
80.                        APP_IRQ_PRIORITY_LOW,
81.                        err_code);
82.
83.     APP_ERROR_CHECK(err_code);
84.     while (1)
85.     {
86.         uart_loopback_test();//调用回环
87.     }
88. }
```

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。在发送区写入任何字符或者字符串，比如发送 nrf52832，点击发送后，接收端会接收到相同的字符或者字符串，如图接收到了 nrf52832。当输入字符 'q' 或者 'Q'，会输出 Exit!结束测试。



6.5 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：

6.5.1 原理分析

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。

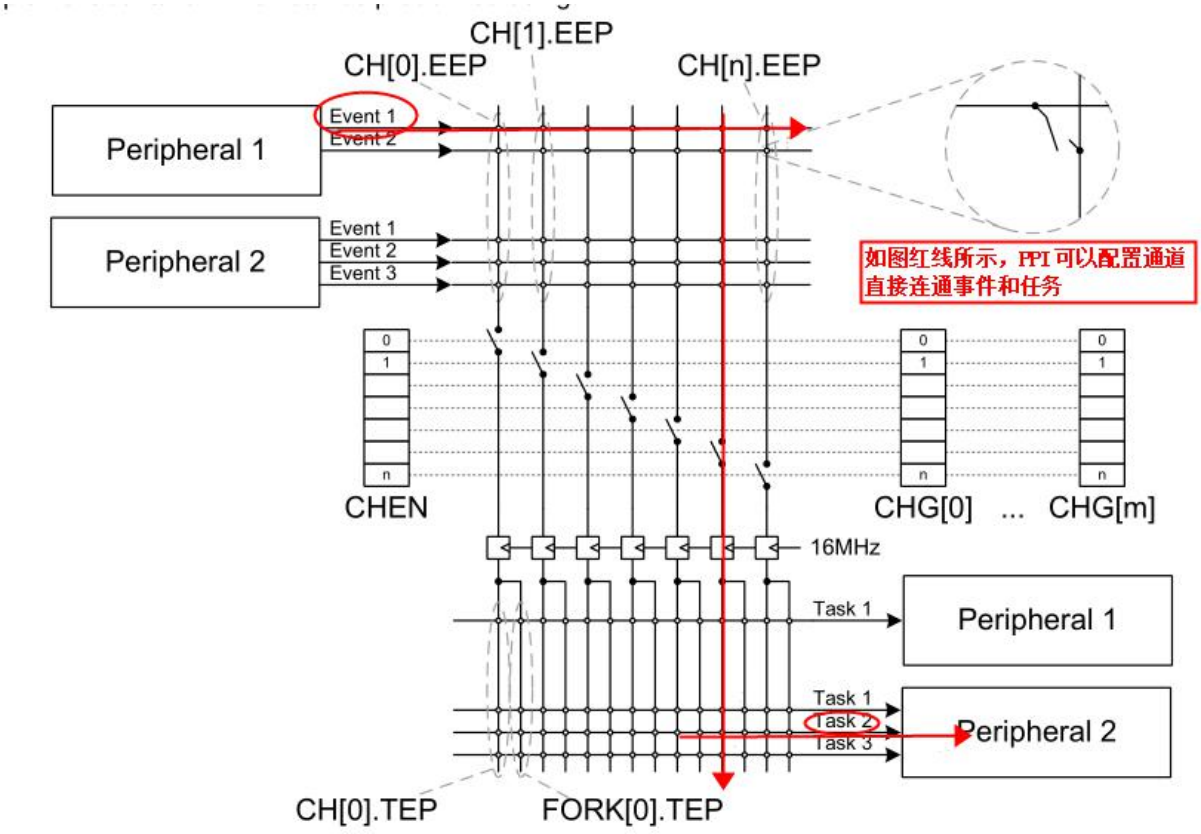
PPI 的两端一端连接的是事件端点（EEP），一端连接的是任务端点（TEP）。因此 PPI 可以通过一个外设上发生的事件自动的触发另一个外设上的任务。首先外设事件需要通过与事件相关的寄存器地址连接到一个事件端点（EEP），另一端外设任务事件也需要通过此任务相关的任务寄存器地址连接到一个任务端点（TEP），当两端连接好后就可以通过 PPI 自动触发了。

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。



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

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。

6.5.2 应用实例编写

6.5.2.1 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 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。

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
14.       NRF_TIMER0->MODE =TIMER_MODE_MODE_Timer;//设置为定时模式
04.       NRF_TIMER0->BITMODE = 3;       //32bit
05.
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：

```
01. #define PWM_OUT1                LED_0
02. #define PWM_OUT2                LED_1
03.
04. void gpiote_init(void){
05.       NRF_GPIOTE->CONFIG[0] = ( 3 << 0 )            //作为 task 模式
06.                               | ( PWM_OUT1 << 8) //设置 PWM 输出引脚
07.                               | ( 3 << 16 )       //设置 task 为翻转 PWM 引脚的电平
08.                               | ( 1 << 20);       //初始输出电平为高
09.    NRF_GPIOTE->CONFIG[1] = ( 3 << 0 )             //作为 task 模式
10.                               | ( PWM_OUT2 << 8) //设置 PWM 输出引脚
11.                               | ( 3 << 16 )       //设置 task 为翻转 PWM 引脚的电平
12.                               | ( 1 << 20);       //初始输出电平为高
13. }
```

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：

```
14. int main(void){
15.
16.       gpiote_init();
17.       ppi_set();
18.       timer0_init();
19.       while(1);
20. }
```

6.5.2.2 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：

```
●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);

/**
 * 函数功能：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 imer/PWM 正在使用或者初始化失
败 */
```

```
● ret_code_t app_pwm_channel_duty_set(app_pwm_t const * const p_instance,
                                      uint8_t channel, app_pwm_duty_t duty);

/**
 * 设置 PWM 的占空比的值.
 *
 * A duty cycle change requires one full PWM clock period to finish.
 * If another change is attempted for any channel of given instance before
 * the current change is complete, the new attempt will result in the error
 * 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_LOW},
    .num_of_channels = 2,
    .period_us= period_in_us
    }
```

结构体主要包括：管脚、管脚极性、通道数量、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：

```
    // Initialize timer:初始化定时器
    nrf_timer_frequency_t    timer_freq = pwm_calculate_timer_frequency(p_config->period_us);//估算定时器频率
    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
{
```

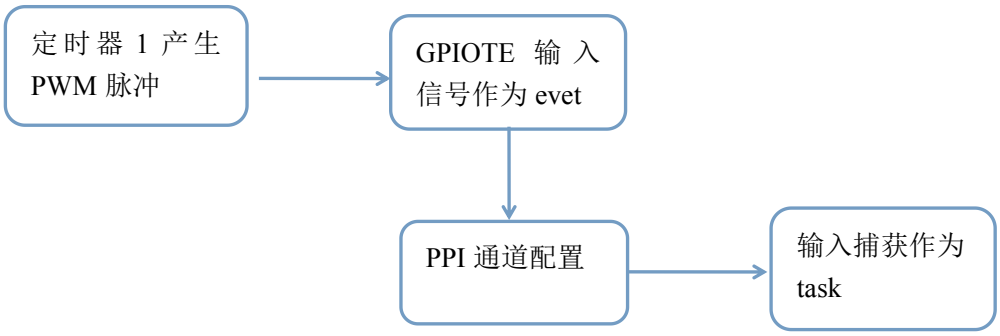
```
07.         ready_flag = true;
08.  }
09.
10.  int main(void)
11.  {
12.         ret_code_t err_code;
13.
14.         /* 2-个频道的 PWM, 200Hz（5000us=5ms), 通过 开发板 LED 管脚输出. */
15.         app_pwm_config_t   pwm1_cfg = APP_PWM_DEFAULT_CONFIG_2CH(5000L, BSP_LED_0,
    BSP_LED_1);
16.
17.         /* 切换两个频道的极性 */
18.         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。

6.6 定时器输入捕获

6.6.1 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。输入脉冲信号则通过处理器产生 PWM 来实现。



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：

下面介绍本例将使用的几个重要的组件库函数：

●uint32_t nrf_drv_ppi_channel_alloc(nrf_ppi_channel_t * p_channel);

```
/**功能：分配 PPI 通道的函数，用于分频未使用的 PPI 通道.
 *
 * 参数：  p_channel                指向被分配的 PPI 通道的指针
 *
 * 返回值  NRF_SUCCESS              如果这个通道被成功分配
 * 返回值  NRF_ERROR_NO_MEM         如果被使用的是一个无效的通道
 */
```

●uint32_t nrf_drv_ppi_channel_assign(nrf_ppi_channel_t channel, uint32_t eep, uint32_t tep);

```
/**功能：  用于在 PPI 通道两端分频的 task  事件和 event 事件.
 *
 * 参数：   channel                 用于分配端点地址的 PPI 通道
 * 参数：   eep                     Event  端点的地址
 * 参数：   tep                     Task 端点的地址
 *
 * 返回值      NRF_SUCCESS                 如果通道被成功分配
 * 返回值      NRF_ERROR_INVALID_STATE  如果通道不分配给用户
 * 返回值      NRF_ERROR_INVALID_PARAM  如果通道不是可由用户配置的
```

```
●ret_code_t nrf_drv_gpiote_in_init(nrf_drv_gpiote_pin_t    pin,
                                   nrf_drv_gpiote_in_config_t     const * p_config,
                                   nrf_drv_gpiote_evt_handler_t    evt_handler);

/**
 * 功能：  初始化 GPIOTE 输入端口.
 * 详细描述：  输入管脚能够在两种模式下工作:
 * - lower accuracy but low power (high frequency clock not needed)
 * - higher accuracy (high frequency clock required)
 *
 * The initial configuration specifies which mode is used.
 * If high-accuracy mode is used, the driver attempts to allocate one
 * of the available GPIOTE channels. If no channel is
 * available, an error is returned.
 * 参数：pin           管脚
 * 参数：p_config     初始化配置
 * 参数：evt_handler  当配置发送后的回调函数.
 *
 * 返回值   NRF_SUCCESS                  如果初始化成功.
 * 返回值   NRF_ERROR_INVALID_STATE  如果驱动未初始化或者端口正在被使用
 * 返回值   NRF_ERROR_NO_MEM          如果没有 GPIOTE 通道是可用的
 */
```

6.6.2 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，输入事件设置为高精度*/
} 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，高精度感应，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(0);
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. }
```

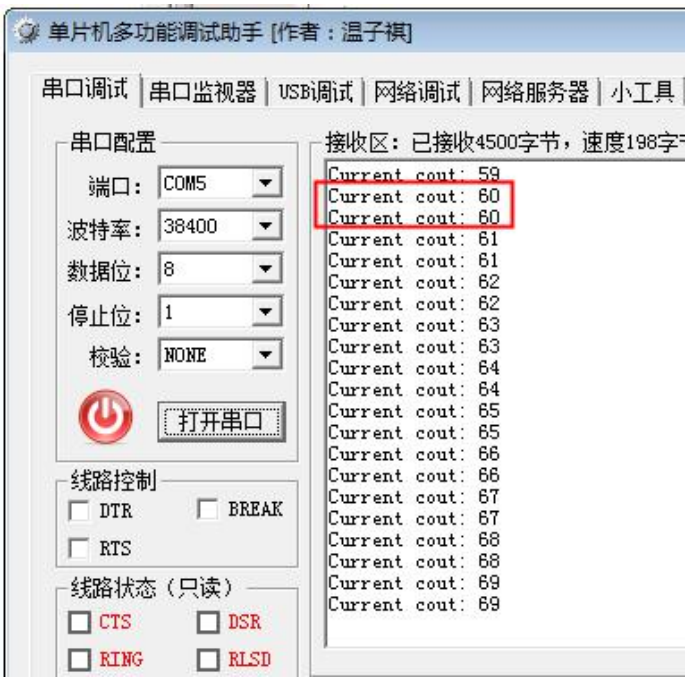
主函数调用之前初始化函数，同时需要使用串口输出计数值，因此还需要对串口进行配置，然后调用定时器比较器捕获功能，使用函数：

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. {
```

```
    timer0_init(); //初始化定时器 0
     gpio_init();
    ppi_init();       //PPI 的初始化
    PWM_OUT(10);
    uint32_t err_code;
    const app_uart_comm_params_t comm_params =
      {
          RX_PIN_NUMBER,
          TX_PIN_NUMBER,
          RTS_PIN_NUMBER,
          CTS_PIN_NUMBER,
          APP_UART_FLOW_CONTROL_ENABLED,
          false,
          UART_BAUDRATE_BAUDRATE_Baud38400
      };
    APP_UART_FIFO_INIT(&comm_params,
                       UART_RX_BUF_SIZE,
                       UART_TX_BUF_SIZE,
                       uart_error_handle,
                       APP_IRQ_PRIORITY_LOW,
                       err_code);
    APP_ERROR_CHECK(err_code);
    // 开功耗
    NRF_POWER->TASKS_CONSTLAT = 1;
    // 开始定时器
    nrf_drv_timer_enable(&timer0);
    //定时器计数器捕获的值输出
    while (true)
    {
        printf("Current cout: %d\n\r",
(int)nrf_drv_timer_capture(&timer0,NRF_TIMER_CC_CHANNEL2));
        nrf_delay_ms(1000);//延迟 1s
    }
}
```

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：



6.7 RTC 实时时钟

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

6.7.1 原理分析

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

●RTC 时钟源

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

●溢出分频率和分频

RTC 的计数器 COUNTER 增量的频率按照下面公式计算：

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

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

●COUNTER 寄存器

●溢出功能

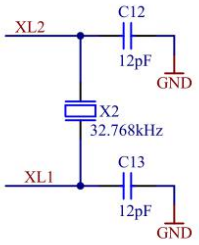
●TICK 事件

●事件控制

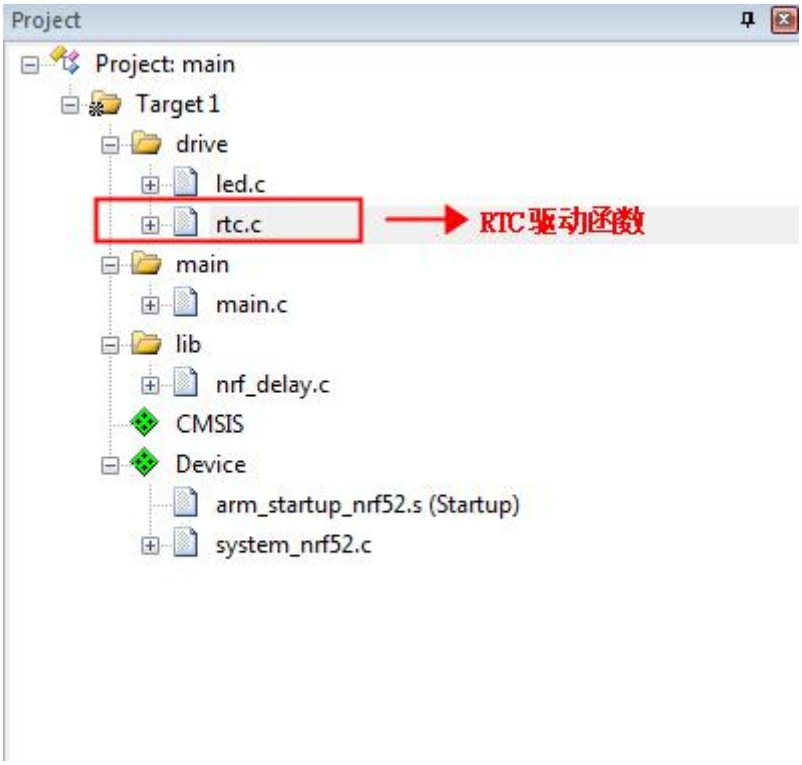
6.7.2 应用实例编写

6.7.2.1 TICK 与比较应用

如下图所示： 青云 QY-nRF52832 开发板上，通过管脚 P0.27 和管脚 P0.26 连接低速外部晶振，晶振大小为 32.768KHZ。



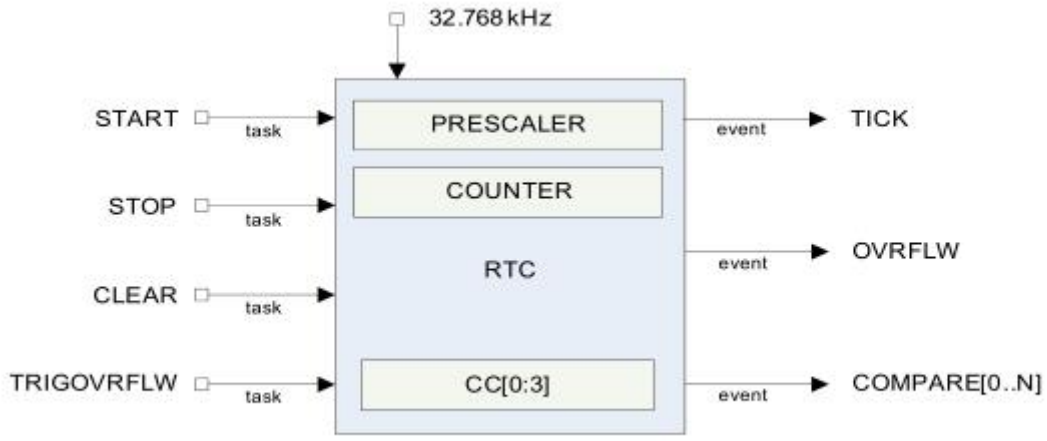
在代码文件中，实验五建立了一个演示历程，我们打开看看需要那些库文件。打开 user 文件夹中的 RTC 工程：



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

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

下面我们就结合寄存器来详细分析下 RTC 的设置：



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

```
/**  启动 LFCLK 晶振功能*/
void lfclk_config(void)
{
    NRF_CLOCK->LFCLKSRC = (CLOCK_LFCLKSRC_SRC_Xtal <<
                                    CLOCK_LFCLKSRC_SRC_Pos);//设置 32K 时钟源
    NRF_CLOCK->EVENTS_LFCLKSTARTED   = 0;//关 32K 震荡事件
    NRF_CLOCK->TASKS_LFCLKSTART      = 1;//开 32K 震荡任务
    while (NRF_CLOCK->EVENTS_LFCLKSTARTED == 0)
    {
    }
    NRF_CLOCK->EVENTS_LFCLKSTARTED = 0;
}
```

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

上面红色框框的几个寄存器就是代码里定义的寄存器。

TASKS_LFCLKSTART 开低速时钟振荡。时钟源的选择：

```
#define CLOCK_LFCLKSRC_SRC_RC (0UL) /*!< 内部 32KiHz RC 时钟. */
#define CLOCK_LFCLKSRC_SRC_Xtal (1UL) /*!< 外部 32KiHz 振荡.l. */
#define CLOCK_LFCLKSRC_SRC_Synth (2UL) /*!< 内部 32KiHz 从 HFCLK 系统时钟产生. */
```

我们选择外部低速时钟振荡作为时钟源。这里就把低速时钟源设置完成了。接下来进行 RTC 的配置，我们实验的要求是实现 8Hz 的计数频率，也就是计数时间 125ms 控制 led 翻转一次，翻转 24 次接近 3000ms，也就是 3s 的时候进行模拟比较报警，报警控制另一个 led 灯点亮报警。

这个是下面代码设置要用的几个寄存器，解释很清楚，关键是搞清楚怎么用，我们之间对着代码段分析：

```
01.    void rtc_config(void)
02. {
03.        NVIC_EnableIRQ(RTC0_IRQn);                  // 使能 RTC 中断.
04.        NRF_RTC0->PRESCALER = COUNTER_PRESCALER;
05.    //12 位预分频器的计数器频率（ 32768 / （预分频器+1 ） ）
06.    //当 RTC 停止时候才能设置
07.    // 设置预分频值 prescaler to a TICK of RTC_FREQUENCY.
08.        NRF_RTC0->CC[0]    = COMPARE_COUNTERTIME * RTC_FREQUENCY;
09.    // 设置比较寄存器值 Compare0 after approx COMPARE_COUNTERTIME seconds.
10.
11.        // 启用 Tick 事件和节拍中断
12.        NRF_RTC0->EVTENSET          = RTC_EVTENSET_TICK_Msk;
13.        NRF_RTC0->INTENSET          = RTC_INTENSET_TICK_Msk;
14.
15.        // 启用事件比较匹配和比较匹配中断
16.        NRF_RTC0->EVTENSET          = RTC_EVTENSET_COMPARE0_Msk;
```

```
17.     NRF_RTC0->INTENSET          = RTC_INTENSET_COMPARE0_Msk;
18. }
19.
```

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

第 4 行~第 8 行：我们设计 RTC 的计数频率，根据手册给出的公式：

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

根据这个公式，下面举两个例子：

1. 设置计数频率为 100 Hz (10 ms 一个计数周期)

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

RTC 的时钟频率 fRTC = 99.9 Hz

10009.576 μs 一个计数周期

2. 设置计数频率为 8 Hz (125ms 一个计数周期)

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

RTC 的时钟频率 fRTC = 8 Hz

125 ms 一个计数周期

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

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

```
NRF_RTC0->PRESCALER = COUNTER_PRESCALER;
#define  COUNTER_PRESCALER  ((LFCLK_FREQUENCY/RTC_FREQUENCY) - 1)
#define LFCLK_FREQUENCY  (32768UL) //
#define RTC_FREQUENCY       (8UL)
```

这个参数是更加上面举的例子 2 里

代入公式 PRESCALER = round (32.768 kHz / 8 Hz) - 1 = 4095 来设置这个 RTC 预分频计数器

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

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

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

ID	Field	Value	Description
A			Compare value

设置完成后，使能计数器中断和比较中断：

```
01.     // Enable TICK event and TICK interrupt:启用 Tick 事件和节拍中断
```

```
02.       NRF_RTC0->EVTENSET          = RTC_EVTENSET_TICK_Msk;
03.       NRF_RTC0->INTENSET          = RTC_INTENSET_TICK_Msk;
04.
05.       // Enable COMPARE0 event and COMPARE0 interrupt:启用事件比较匹配和比较匹配中断
06.       NRF_RTC0->EVTENSET          = RTC_EVTENSET_COMPARE0_Msk;
07.       NRF_RTC0->INTENSET          = RTC_INTENSET_COMPARE0_Msk;
```

并且设置中断功能，中断做的内容就比较简单了，判断中断发生后翻转 LED 灯，两种中断事件，一个是 RTC 滴答事件，一个是 RTC 比较事件。

```
01.    void RTC0_IRQHandler()
02. {
03.       if ((NRF_RTC0->EVENTS_TICK != 0) &&
04.           ((NRF_RTC0->INTENSET & RTC_INTENSET_TICK_Msk) != 0))
05.       {
06.           NRF_RTC0->EVENTS_TICK = 0;
07.           LED1_Toggle();
08.       }
09.
10.       if ((NRF_RTC0->EVENTS_COMPARE[0] != 0) &&
11.           ((NRF_RTC0->INTENSET & RTC_INTENSET_COMPARE0_Msk) != 0))
12.       {
13.           NRF_RTC0->EVENTS_COMPARE[0] = 0;
14.           LED2_Toggle();
15.       }
16. }
17.
```

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

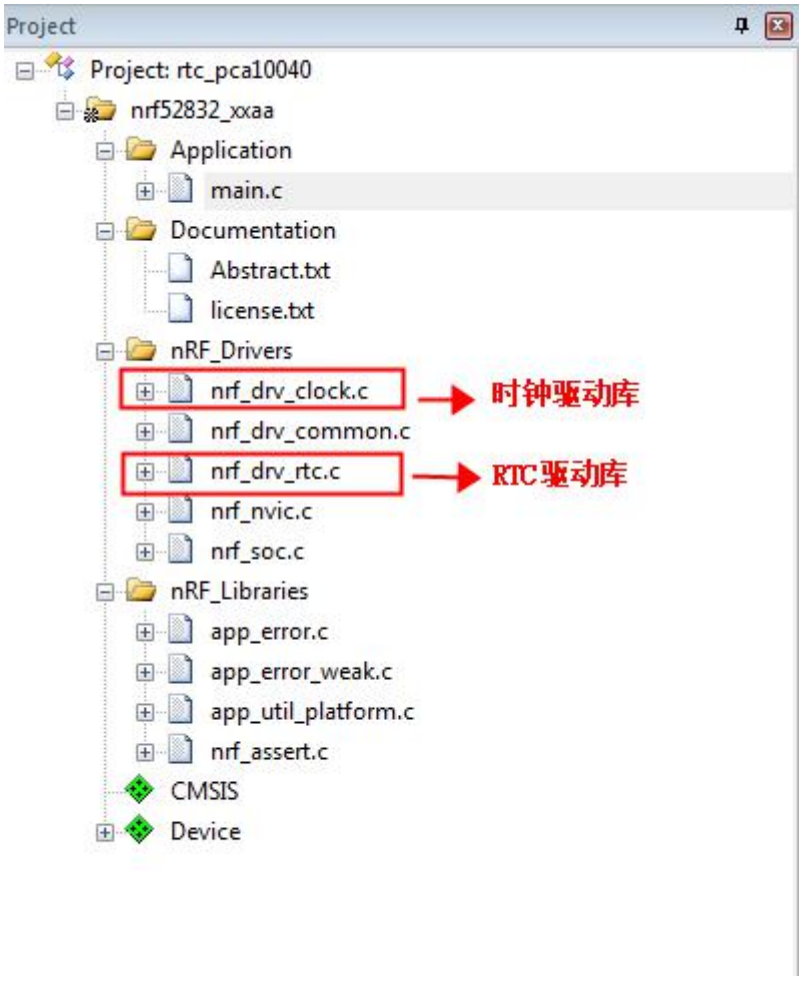
```
01. /******************** (C) COPYRIGHT 2012  青风电子  ********************
02.   * 文件名    ：main
03.   * 描述      ：
04.   * 实验平台：青风 nrf52832 开发板
05.   * 描述      ：RTC
06.   * 作者      ：青风
07.   * 店铺      ：qfv5.taobao.com
08. *********************************************************************/
09. #include "nrf52.h"
10. #include   "led.h"
11. #include   "rtc.h"
12.
13. int main(void)
14. {
15.     LED_Init();
```

```
16.     LED1_Close();
17.     LED2_Close();
18.     lfclk_config();//启动内部 LFCLK 晶振功能
19.     rtc_config();//配置 RTC
20.
21.     NRF_RTC0->TASKS_START = 1;//开启 rtc
22.     while (1)
23.       {
24.           // Do nothing.
25.       }
26. }
```

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

6.7.2.2 RTC 组件库的使用

官方 SDK 中，提供了关于 RTC 的驱动组件库文件，下面我们来演示如果采用 RTC 组件库进行工程的编写。首先建立工程目录树如下图所示，工程中加入两个关键的库文件：nrf_drv_clock.c 和 nrf_drv_rtc.c，这两个驱动文件，一个为时钟的驱动库，一个是 RTC 的驱动库，我们只需要在 main.c 主函数中编写自己的应用。



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

●ret_code_t nrf_drv_rtc_init(nrf_drv_rtc_t const * const p_instance,
nrf_drv_rtc_config_t const * p_config,
nrf_drv_rtc_handler_t handler);

```
/**功能: 初始化 RTC 驱动。初始化后，实例处于电源关闭状态
 *
 * 参数：  p_instance          指向本实例的指针
 * 参数：  p_config            初始化配置. 如果使用的是默认配置就是 NULL.
 * 参数：  handler             用户的事件处理程序
 *
 * 返回值   NRF_SUCCESS                  如果初始化成功
 * 返回值   NRF_ERROR_INVALID_PARAM      如果没有处理被提供.
 * 返回值   NRF_ERROR_INVALID_STATE      如果实例已经初始化了.
 */
```

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

```
/*RTC 驱动实例配置结构体*/
typedef struct
{
    uint16_t prescaler;          /*分频值*/
    uint8_t  interrupt_priority; /*中断优先级*/
    uint8_t  tick_latency;       /*滴答下中断处理程序的最大长度 (最大 7.7 ms). */
    bool     reliable;           /**< 标记. */
} nrf_drv_rtc_config_t;
```

●void nrf_drv_rtc_tick_enable(nrf_drv_rtc_t const * const p_instance, bool enable_irq);

```
/**功能:使能 RTC 的 tick 功能.
 *
 * 参数：  p_instance          指向本实例的指针
 * 参数：  enable_irq          真（True）的就使能中断，假（False）的就关闭中断
*/
```

●void nrf_drv_rtc_enable(nrf_drv_rtc_t const * const p_instance);

```
/**@功能:使能 RTC 驱动实例
 *
 * 参数：p_instance     指向实例的指针.
 */
```

```
●ret_code_t nrf_drv_rtc_cc_set(nrf_drv_rtc_t const * const p_instance,
                               uint32_t channel,
                               uint32_t val,
                               bool enable_irq);

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

介绍完基本函数，下面开始编写程序。首先要求配置时钟，RTC 选取的时钟为低速时钟，首先调用函数 nrf_drv_clock_init 进行时钟配，时钟配置的结构体为 nrf_drv_clock_config_t，这个结构体是时钟初始化参数结构体，这个参数是在没有使用协议栈状态下，同时 RC 低频时钟被选择下使用的。结构体参数如下：

```
typedef struct
{
    nrf_drv_clock_lf_cal_interval_t    cal_interval;        /*校准时间间隔*/
    uint8_t                            interrupt_priority;  /*时钟中断优先级*/
} nrf_drv_clock_config_t;
```

如果使用默认参数，则为 NULL。默认时钟配置参数为：

```
#define NRF_DRV_CLOCK_DEAFULT_CONFIG                                    \
        {
            .cal_interval          = CLOCK_CONFIG_LF_RC_CAL_INTERVAL,
            .interrupt_priority = CLOCK_CONFIG_IRQ_PRIORITY,
        }
```

最后发出低速时钟请求，具体代码如下：

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

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

```
01. static void rtc_config(void)//RTC 配置
```

```
02. {
03.     uint32_t err_code;
04.     //初始化 RTC
05.     err_code = nrf_drv_rtc_init(&rtc, NULL, rtc_handler);
06.     APP_ERROR_CHECK(err_code);
07.     //使能滴答事件和中断
08.     nrf_drv_rtc_tick_enable(&rtc,true);
09.    //设置当 COMPARE_COUNTERTIME 秒后比较频道去触发中断
10.
    err_code=nrf_drv_rtc_cc_set(&rtc,0,COMPARE_COUNTERTIME*RTC0_CONFIG_FREQUENCY,
11.                                true);
12.     APP_ERROR_CHECK(err_code);
13.     //开 RTC 电源
14.     nrf_drv_rtc_enable(&rtc);
15. }
```

第 5 行，对 RTC 初始化，在程序开头先通过一个申明使用的&rtc 是 RTC0：const nrf_drv_rtc_t rtc = NRF_DRV_RTC_INSTANCE(0)，那么初始化 nrf_drv_rtc_init 就是配置这个 RTC0，配置参数为 NULL，也就是使用默认配置，RTC 中断回调操作为 rtc_handler。那么库组件中给出的默认配置如下：

```
#define NRF_DRV_RTC_DEFAULT_CONFIG(id)
{
    .prescaler    = (uint16_t)(RTC_INPUT_FREQ / CONCAT_3(RTC,id,_CONFIG_FREQUENCY))-1,
    //分频值
    .interrupt_priority = CONCAT_3(RTC, id, _CONFIG_IRQ_PRIORITY),
    //中断优先级
    .reliable    = CONCAT_3(RTC, id, _CONFIG_RELIABLE),
    //标记
    .tick_latency = RTC_US_TO_TICKS(NRF_MAXIMUM_LATENCY_US,
                    CONCAT_3(RTC, id, _CONFIG_FREQUENCY)),
  //滴答下中断处理程序的最大长度
}
```

其中分频值是根据 RTC 的计数频率的公式：$PRESCALER = (32.768kHZ / f_{RTC}) - 1$，RTC_INPUT_FREQ 为 32768，CONCAT_3(RTC,id,_CONFIG_FREQUENCY)表示三个参数组合：RTC、id=0、_CONFIG_FREQUENCY 表示把 RTC0 计数频率配置为 8HZ，注意，由于分频器是 12bit，最低频率只能达到 8HZ。

第 8 行：设置 RTC 滴答 tick，并且使能滴答 tick 中断，那么在 RTC 的计数频率下，计数一次就发生一次中断，因此 125ms 触发一次中断。

第 10 行：设置 RTC 比较中断，当 RTC 的计数值达到了比较寄存器内的值的时候触发 RTC 比较中断，其中比较值设置为：COMPARE_COUNTERTIME*RTC0_CONFIG_FREQUENCY=3*8，相当于计数 24 次触发一次比较中断，时间为 125ms*24=3s。

第 14 行：使能 RTC，开 RTC 电源，RTC 开始运行。

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

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

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

```
12. int main(void)
13. {
14.     leds_config();//led 配置
15.     lfclk_config();//时钟配置
16.     rtc_config();//RTC 配置
17.
18.     while (true)
19.     {
20.
21.     }
22. }
```

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

6.8 SAADC 采集

ADC 为 Analog-to-Digital Converter 的缩写，指模/数转换器或者模数转换器 。是指将连续变化的模拟信号转换为离散的数字信号的器件。真实世界的模拟信号，例如温度、压力、声音或者图像等，需要转换成更容易储存、处理和发射的数字形式。模/数转换器可以实现这个功能。在 nrf52832 中的 ADC 为一个逐次逼近（successive-approximation）模拟数字转换器。具体的属性如下所示（参考手册 P345 页）：

1: 8/10/12 分辨率，采用过采样可达到 14 位分辨率。

2：多达 8 个输入通道：

单端输入时时有 1 个通道，2 个通道组成差分输入。

单端和差分输入时可以配置成扫描模式。

3：满量程输入范围为 0 和 VDD

4：可以通过软件触发采样任务启动采样，也可以使用低功耗 32.768Khz 的 RTC 定时器 H 或者更加精确的的 1/16Mhz 定时器通过 PPI 来触发采样任务，使得 SAADC 具有非常灵活的采样频率。

5：单次的采集模式只使用一个采集通道。

6：扫描模式是按照顺序采样一系列通道。频道直接的采样延迟位 t ack + t conv。用户通过配置 t ack 可以使频道直接的间隔时间不同

7：可以通过 EasyDMA 可以直接将采样的结果保存到 RAM 内。

8：中断发生在单次采样和换成满事件时。

9： Samples stored as 16-bit 2’ s complement values for differential and single-ended sampling

10：无需外部定时器就可以实现连续采样。

11：可以配置通道输入负载电阻。

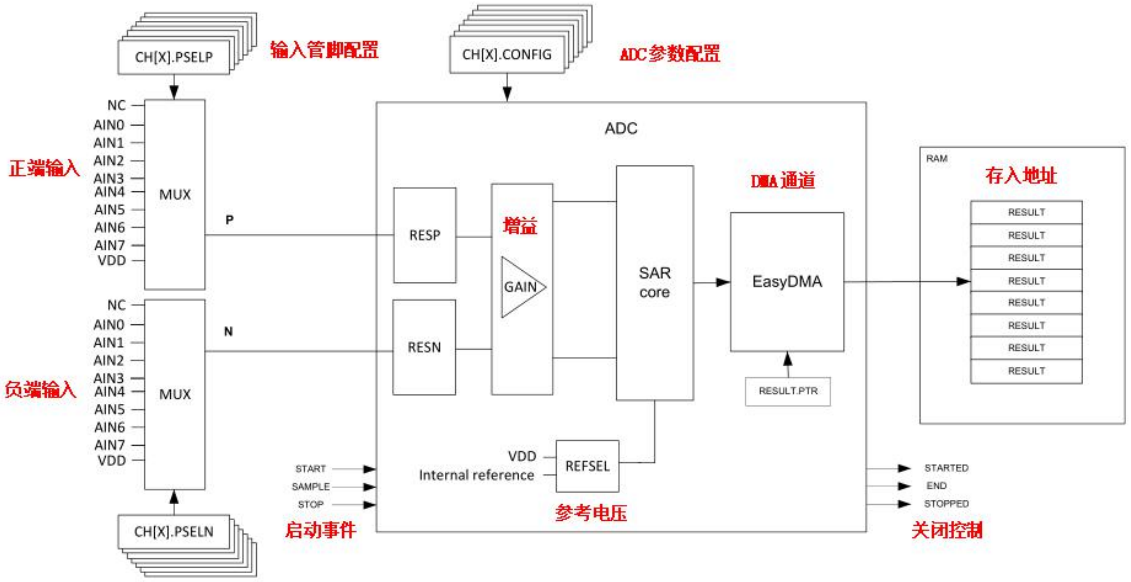
6.8.1 原理分析

下面来分析下 nrf52832 内部 ADC 的基本原理，其内部 SAADC 支持多达到 8 个外部模拟输入通道。我们需要关心的 ADC 的几个关键参数：

1：采样模式

nrf52832 内部 ADC 的采样信号输入可以通过 8 个外部模拟输入通道进行采集，但是其 ADC 内部事件上有 16 个通道和 VDD，如下图所示。其中 8 个通道为正端输入（N），另外 8 个通道为负端输入（P）。因此信号采样的模式可以分为单独输入和差分输入两种方式。

Channel input	Source	Connectivity
CH[n].PSELP	AIN0...AIN7	Yes(any)
CH[n].PSELP	VDD	Yes
CH[n].PSELN	AIN0...AIN7	Yes(any)
CH[n].PSELN	VDD	Yes



默认状态下，ADC 的配置模式为单端输入，因为在配置寄存器 CH[n].CONFIG 中设置的 MODE 为 0。需要配置为差分输入的时候，把 MODE 设置为 1。单端模式的时候，ADC 内部把负极接入到地，如下表所示：

F	RW	MODE		Enable differential mode
		SE	0	Single ended, PSELN will be ignored, negative input to ADC shorted to GND
		Diff	1	Differential

而差分模式则是把负极通过负向端输入，通过计算两端的差值来换算出采样结果。具体计算公式我们等下来讨论。

2：信号增益

如上面的 ADC 内部结构图，采样数据通过采集输入端进入后会通过一个增益 GAIN 对信号进行放大，这个增益在配置寄存器 CH[n].CONFIG 中设置的 GAIN 位进行设置，如下表所示，可以设置为几个值：1/6、1/5、1/4、1/3、1/2、1、2、4 这些参数。

C	RW	GAIN		Gain control
		Gain1_6	0	1/6
		Gain1_5	1	1/5
		Gain1_4	2	1/4
		Gain1_3	3	1/3
		Gain1_2	4	1/2
		Gain1	5	1
		Gain2	6	2
		Gain4	7	4

3：参考电压：

ADC 的参考电压可以采用两种方式，一个是内部参考电压，为 0.6V 大小。另一种采用 VDD/4 为参考电压。通过配置寄存器 CH[n].CONFIG 中设置的 RESEL 位进行配置，如下表所示：

D	RW	REFSEL		Reference control
		Internal	0	Internal reference (0.6 V)
		VDD1_4	1	VDD/4 as reference

输入采样电压范围：当采样内部电压作为参考时，范围为$\pm 0.6\ V$；当使用 VDD 电压作为参考电压时，输入范围为$\pm VDD/4$。Gain 增益参数可以来调整输入范围，如下图所示：

```
Input range = (+- 0.6 V or +-VDD/4)/Gain
```

比如，如果选择 VDD 作为参考电压，信号输入采样单端输入，同时增益为 1/4，那么输入信号范围为：

```
Input range = (VDD/4)/(1/4) = VDD
```

如果选择内部电压作为参考，信号为单端输入，放大增益为 1/6，那么输入电压范围为：

```
Input range = (0.6 V)/(1/6) = 3.6 V
```

但是要注意，AIN0-AIN7 输入范围不能超过 VDD，低于 VSS。

4：采样精度：

ADC 的分辨率对采样结果也是至关重要的，nrf52832 内部 ADC 可以配置为多种采样分辨率，一般情况下可以设置为 8/10/12 位，采用过采样可达到 14 位分辨率。通过设置 RESOULUTION 寄存器来配置，如下表所示：

Bit number				31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Id				A A A
Reset 0x00000001				0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
Id	**RW**	**Field**	**Value Id**	**Value**	**Description**
A	RW	VAL			Set the resolution
			8bit	0	8 bit
			10bit	1	10 bit
			12bit	2	12 bit
			14bit	3	14 bit

通过这几个步骤，采集结果就可以出来了，那么输出结果按照下面公式进行计算：

$$RESULT = [V(P) - V(N)] * GAIN/REFERENCE * 2^{(RESOLUTION - m)}$$

公式中：

V(P)为：ADC 正端输入

V(N)为：ADC 负端输入

GAIN 为：为增益值

REFERENCE 为：为参考电压

RESOULUTION 为：采样精度

M 为：采样模式，当单端输入的时候为 0，差分输入的时候为 1。

5：工作模式：

SAADC 有三种工作模式，分别为：单次转换模式，连续转换模式，扫描模式。

5.1 单次转换模式 One-shot mode：

要配置 nRF52832 的 SAADC 为单次转换模式，通过配置 CH[n].PSELP,CH[n].PSELN 和 CH[n].CONFIG 寄存器，来使能一个 ADC 通道，使得 ADC 工作于单次模式。当触发采样任务后，ADC 开始采样输入电压，采样时间通过 CH[n].CONFIG.TACQ 来设置，当 DONE 事件发生表示一次采集完成。

在没有过采样发生的时候，RESULTDONE event 事件等同于 DONE 事件。在实际采样数据通过 EasyDMA 保存到 RAM 之前，这两个事件都会发生。

5.2 连续转换模式 Continuous mode：

连续采样模式能够通过内部定时器实现定时采样，或者触发 SAMPLE 任务通过 PPI 链接一个通用寄存器来实现。

SAMPLERATE 寄存器能够被用于用本地的定时器代替独立的 SAMPLE tasks。当设置 SAMPLERATE.MODE 位设置为 Timers, 单次的 SAMPLE task 任务来启动 SAADC。一个 STOP task 停止采样。在 SAMPLERATE.CC 来控制采样率.

回到普通采样，设置 SAMPLERATE.MODE 返回 Task。如下图所示：

Bit number				31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Id				B A A A A A A A A A A
Reset 0x00000000				0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Id	**RW**	**Field**	**Value Id**	**Value**	**Description**
A	RW	CC		[80..2047]	Capture and compare value. Sample rate is 16 MHz/CC
B	RW	MODE			Select mode for sample rate control
			Task	0	Rate is controlled from SAMPLE task
			Timers	1	Rate is controlled from local timer (use CC to control the rate)

采样的频率，SAMPLERATE.CC 内定时时间越短，频率越快。同时，这个还和通道数量有关系，公式如下所示：

$$f_{SAMPLE} < 1/[CHANNELS \times (t_{ACQ} + t_{conv})]$$

在没有过采样发生的时候，RESULTDONE event 事件等同于 DONE 事件。在实际采样数据通过 EasyDMA 保存到 RAM 之前，这两个事件都会发生。

5.3 扫描采样

频道被使能如果 CH[n].PSELP 被设置。如果超过 1 个通道被使能，那么 ADC 就进入扫描模式。区别与单次模式，就是频道大于 1。

在扫描模式下，一个 SAMPLE task 任务触发每个被使能的通道进行转换，所有频道转换需要的时间按下面公式计算:

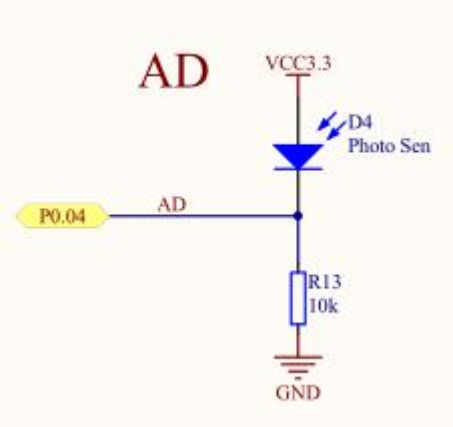
$$Total\ time < Sum(CH[x].t_{ACQ}+t_{CONV}),\ x=0..enabled\ channels$$

在没有过采样发生的时候，RESULTDONE event 事件等同于 DONE 事件。在实际采样数据通过 EasyDMA 保存到 RAM 之前，这两个事件都会发生。

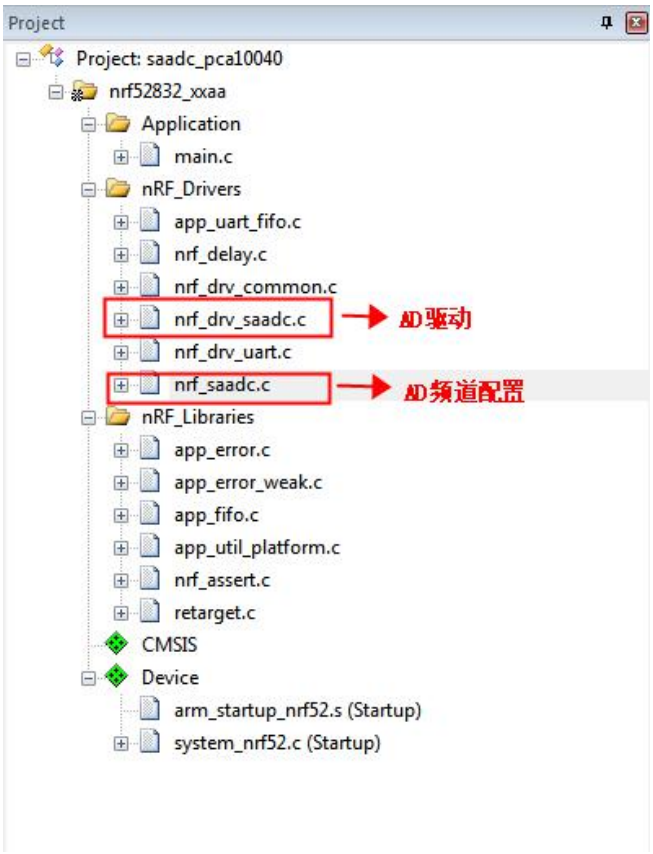
6.8.2 应用实例编写

6.8.2.1 ADC 的单次采样

如下图所示：青云 QY-nRF52832 开发板上，光敏电阻通过管脚 P0.04 接入到 nrf52832 中，P0.04 作为 ADC 采样管脚。



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 arm5 文件夹中的工程项目如下：



在工程中，两个和 ADC 相关的驱动文件分别为：nrf_drv_saadc.c 和 nrf_saadc.c 两个驱动文件，其中 nrf_drv_saadc.c 为 adc 的驱动函数集合，nrf_saadc.c 为 adc 频道配置函数。这两个函数需要加入工程项目中，同时注意配置驱动路径。

下面我们首先来进行 saadc 的初始化配置，设置代码如下 ：

```
16.    void saadc_init(void)
17. {
18.        ret_code_t err_code;
19.      //adc 通道配置
20.        nrf_saadc_channel_config_t channel_config =
21.        NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_AIN2);
22.      //adc 初始化
23.        err_code = nrf_drv_saadc_init(NULL, saadc_callback);
24.        APP_ERROR_CHECK(err_code);
25.      //adc 通道初始化
26.        err_code = nrf_drv_saadc_channel_init(0, &channel_config);
27.        APP_ERROR_CHECK(err_code);
28. }
29.
```

第 5 行：通过设置一个结构体，设置配置函数，官方给了两个结构体来配置单端输入和差分输入。在函数头文件 nrf_drv_saadc.h 文件中采用：

NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_SE：表示单端输入配置。

NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_DIFFERENTIAL：表示差分输入配置。

比如单端输入配置结构体如下表示所示：

```
01. #define NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_SE(PIN_P) \
02. {                                                      \
03.     .resistor_p = NRF_SAADC_RESISTOR_DISABLED,         \
04.     .resistor_n = NRF_SAADC_RESISTOR_DISABLED,         \
05.     .gain       = NRF_SAADC_GAIN1_6,                   \
06.     .reference  = NRF_SAADC_REFERENCE_INTERNAL,        \
07.     .acq_time   = NRF_SAADC_ACQTIME_10US,              \
08.     .mode       = NRF_SAADC_MODE_SINGLE_ENDED,         \
09.     .pin_p      = (nrf_saadc_input_t)(PIN_P),          \
10.     .pin_n      = NRF_SAADC_INPUT_DISABLED             \
11. }
```

上面的参数配置代码解释如下：

第 3 行：正端输入：SAADC 的旁路电阻关

第 4 行：负端输入：SAADC 的旁路电阻关

第 5 行：增益：SAADC 的增益为 1/6

第 6 行：参考电压值：采用芯片内部参考电压

第 7 行：采样时间：10us

第 8 行：模式：单端输入

第 9 行：正端输入引脚：输入管脚

第 10 行：负端输入引脚：输入管脚关闭。

根据硬件配置，管脚 P0.04 是可以配置为 AD 输入管脚 AIN2，可以查看数据手册 P27 页管脚配置，如下图所示：

5	P0.03	Digital I/O	General purpose I/O pin.
	AIN1	Analog input	SAADC/COMP/LPCOMP input.
6	P0.04	Digital I/O	General purpose I/O pin.
	AIN2	Analog input	SAADC/COMP/LPCOMP input.
7	P0.05	Digital I/O	General purpose I/O pin.
	AIN3	Analog input	SAADC/COMP/LPCOMP input.
8	P0.06	Digital I/O	General purpose I/O pin.

```
30.     //adc 初始化
31.     err_code = nrf_drv_saadc_init(NULL, saadc_callback);
```

nrf_drv_saadc_init 函数对 saadc 进行初始化，第一样形参 NULL,表示默认使用 nrf_drv_config.h 配置文件里的设置（在 SDK12 版本开始使用 sdk_config.h 配置文件代替），如下图所示：

```
387 /* SAADC */
388 #define SAADC_ENABLED 1
389
390 #if (SAADC_ENABLED == 1)
391 #define SAADC_CONFIG_RESOLUTION         NRF_SAADC_RESOLUTION_10BIT
392 #define SAADC_CONFIG_OVERSAMPLE         NRF_SAADC_OVERSAMPLE_DISABLED
393 #define SAADC_CONFIG_IRQ_PRIORITY       APP_IRQ_PRIORITY_LOW
394 #endif
```

默认设置首先需要使能 SAADC 设置 SAADC_ENABLED 为 1。ADC 的分辨率为 10bit，不发送过采样，中断优先级设为低。

第二个形参为 saadc_callback，设置为 saadc 回调函数，回调中断函数可以为空，什么都不执行。如果有中断任务需要执行，可以在 saadc_callback 内写中断函数。

```
32.     err_code = nrf_drv_saadc_channel_init(0, &channel_config);
33.     APP_ERROR_CHECK(err_code);
```

nrf_drv_saadc_channel_init 函数初始化 ADC 的通道函数，第一个形参设置为通道值，当设置为 0 的时候，表示选择通道 0。第二个形参为前面配置的通道参数结构体作为指针调入。

那么 adc 配置好后，直接可以在主函数里进行调用，同时采用串口输出，代码如下所示：

```
int main(void)
{    nrf_saadc_value_t   saadc_val;
    float   val;  //保存 SAADC 采样数据计算的实际电压值
    uart_config();//配置串口

    printf("\n\rSAADC HAL simple example.\r\n");
    saadc_init();//saadc 初始化
    while(1)
    {
        //启动一次 ADC 采样。
            nrf_drv_saadc_sample_convert(0,&saadc_val);
            //串口输出 ADC 采样值。
            val = saadc_val * 3.6 /1024;
            printf(" %.3fV\n",   val);
            //延时 300ms，方便观察 SAADC 采样数据
            nrf_delay_ms(500);

    }
}
```

主函数中，因为我们没有采用中断采集，那么就通过一个循环扫描采集数据，在 while 循环下延迟 500ms 采集一次电压值。

实验下载到青云 nRF52832 开发板后，通过串口来观察采集的电压大小，为了方便计算，我们可以根据前面的计算公式计算出来电压，

$$RESULT = [V(P) - V(N)] * GAIN/REFERENCE * 2^{(RESOLUTION - m)}$$

VP 为 要采样的电压，VN 为 0，GAIN 为 1/6，REFERENCE 为 0.6，RESOLUTION 为 10，m 为 0，RESULT 是 saadc_val 结果。是那么公式可以化为：

val = saadc_val * 3.6 /1024;

的实验现象如下，可以和万用表对比测量结果：

6.8.2.2 ADC 的差分采样

在 ADC 采样中，单端输入容易受到外界信号的干扰，而采样差分输入，两个输入信号的相减的差值作为最后的结果，可以有效的互相抵消外界干扰。对比上面单端输入，改变如下位置：

```
01. void saadc_init(void)
02. {
03.     ret_code_t err_code;
04. //配置 ADC 输入频道（要改动的）
05.  nrf_saadc_channel_config_t channel_config =
    NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_DIFFERENTIAL
06.                         (NRF_SAADC_INPUT_AIN2,NRF_SAADC_INPUT_AIN0);
07.   //初始化 ADC
08.     err_code = nrf_drv_saadc_init(NULL, saadc_callback);
09.     APP_ERROR_CHECK(err_code);
10.   //初始化 ADC 通道配置
11.     err_code = nrf_drv_saadc_channel_init(0, &channel_config);
12.     APP_ERROR_CHECK(err_code);
13. }
```

用结构体 NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_DIFFERENTIAL 表示差分输入配置。同时配置两个输入通道 NRF_SAADC_INPUT_AIN2 和 NRF_SAADC_INPUT_AIN0，两个输入管脚更具实际需要进行选择，对照手册查找管脚数。配置结构体如下：

```
01. #define NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_DIFFERENTIAL(PIN_P, PIN_N) \
02. {                                                                       \
03.     .resistor_p = NRF_SAADC_RESISTOR_DISABLED,                          \
04.     .resistor_n = NRF_SAADC_RESISTOR_DISABLED,                          \
05.     .gain       = NRF_SAADC_GAIN1_6,                                    \
06.     .reference  = NRF_SAADC_REFERENCE_INTERNAL,                             \
```

```
07.     .acq_time     = NRF_SAADC_ACQTIME_10US,                    \
08.     .mode         = NRF_SAADC_MODE_DIFFERENTIAL,               \
09.     .pin_p        = (nrf_saadc_input_t)(PIN_P),                \
10.     .pin_n        = (nrf_saadc_input_t)(PIN_N)                 \
11. }
```

解释配置如下：

第 3 行：正端输入：SAADC 的旁路电阻关

第 4 行：负端输入：SAADC 的旁路电阻关

第 5 行：增益：SAADC 的增益为 1/6

第 6 行：参考电压值：采用芯片内部参考电压

第 7 行：采样时间：10us

第 8 行：模式：单端输入

第 9 行：正端输入引脚： 正端输入管脚

第 10 行：负端输入引脚：负端输入管脚。

根据硬件配置，管脚 P0.04 是可以配置为 AD 输入管脚 AIN2，管脚 P0.02 是可以配置为 AD 输入管脚 AIN0。

可以查看数据手册 P27 页管脚配置，如下图所示：

4	P0.02	Digital I/O	General purpose I/O pin.
	AIN0	Analog input	SAADC/COMP/LPCOMP input.
5	P0.03	Digital I/O	General purpose I/O pin.
	AIN1	Analog input	SAADC/COMP/LPCOMP input.
6	P0.04	Digital I/O	General purpose I/O pin.
	AIN2	Analog input	SAADC/COMP/LPCOMP input.
7	P0.05	Digital I/O	General purpose I/O pin.
	AIN3	Analog input	SAADC/COMP/LPCOMP input.
8	P0.06	Digital I/O	General purpose I/O pin.

主函数可以不做任何变化，在主函数里扫描采集数据。实验下载到青云 nRF52832 开发板后，通过串口来观察采集的电压大小，为了方便计算，我们可以根据前面的计算公式计算出来电压，

$$RESULT = [V(P) - V(N)] * GAIN/REFERENCE * 2^{(RESOLUTION - m)}$$

VP 为 要正端采样的电压，VN 为负向端采样电压，GAIN 为 1/6，REFERENCE 为 0.6，RESOLUTION 为 10，m 为 0，RESULT 是 saadc_val 结果。是那么公式可以化为：

差分电压 VP-VN=val = saadc_val * 3.6 /1024;

打开串口助手，输出的实验现象如下，可以和万用表对比测量结果：

6.8.2.3 EasyDMA 之单缓冲中断采样

一：原理分析：

官方 SDK 中，对 ADC 采样会提供软件缓冲存放转换，缓冲寄存器满了后就会触发中断，在中断内读取缓冲寄存器的值。也就是称为 EasyDMA 方式，这种方式的采样速度大大提高。当 ADC 的采样任务一旦被触发，那么 ADC 的转换结果可以通过 EasyDMA 存储到在 RAM 内的结果缓冲内。

结果缓冲 buffer 的地址位于 RESULT.PTR 寄存器中，RESULT.PTR 寄存器是双缓冲，当 STARTED 时间产生，下一个 START task 产生，该缓冲能够立即更新和做好数据准备。

在官方库函数中，提供函数 nrf_drv_saadc_buffer_convert 实现该功能，下面通过对该函数的详细介绍，让大家理解采用 EasyDMA 方式如何进行配置的。

:采用该函数的时候首先需要在主函数文件中定义几个宏定义参数，分别如下：

```
//设置缓冲的数量，决定要填满几个缓冲后启动中断
#define   SAMPLES_IN_BUFFER   5
//设置缓冲的组数，这里也就是对应使用的通道的数量，本例是单通道，我们只定义一个数值
static nrf_saadc_value_t          m_buffer_pool[SAMPLES_IN_BUFFER];
//采样转换次数
static uint32_t                    m_adc_evt_counter;
```

对应函数 ret_code_t nrf_drv_saadc_buffer_convert(nrf_saadc_value_t * p_buffer, uint16_t size)这样一个完整定义的函数，其中两个形势参数是必须详细理解的。

◆第一形参：nrf_saadc_value_t * p_buffer，在 adc 初始化设置，我们设置如下所示：

```
err_code = nrf_drv_saadc_buffer_convert(m_buffer_pool,SAMPLES_IN_BUFFER);
    APP_ERROR_CHECK(err_code);
```

也就是说，第一个形参用的 m_buffer_pool，这个 m_buffer_pool 是一个数组，数组的长度为 SAMPLES_IN_BUFFER 这么长。这样一个数组，对应一个 adc 的转换通道，你有几个通道，你就是设置几个数组的长度加倍，例如你如果用 2 个通道的，可以把数组设置为：

m_buffer_pool[SAMPLES_IN_BUFFER*2]

这样一二个数组长度放置两个通道的数据。

◆第二个形参 uint16_t size，这个表示数组大小，也就是在 RAM 内给的缓存大小，如果大小为 N，表示 N 个字节大小的缓存，当然这个 N 并不是没有限制的，它取决与寄存器 RESULT.MAXCNT 内的设置。

这两个才是实际是在函数内部的 nrf_saadc_buffer_init 中调用了，这个函数直接配置的是寄存器，非常方便大家理解：

```
__STATIC_INLINE void nrf_saadc_buffer_init(nrf_saadc_value_t * buffer, uint32_t num)
{
    NRF_SAADC->RESULT.PTR = (uint32_t)buffer;
    NRF_SAADC->RESULT.MAXCNT = num;
}
```

RESULT.PTR 寄存器：

Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Id	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Reset 0x00000000	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Id	RW	Field	Value Id	Value	Description
A	RW	PTR			Data pointer

RESULT.MAXCNT 寄存器：

Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Id																	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Reset 0x00000000	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Id	RW	Field	Value Id	Value	Description
A	RW	MAXCNT			Maximum number of buffer words to transfer

那么对的 EasyDMA 采样方式整个过程可以这样表述出来：

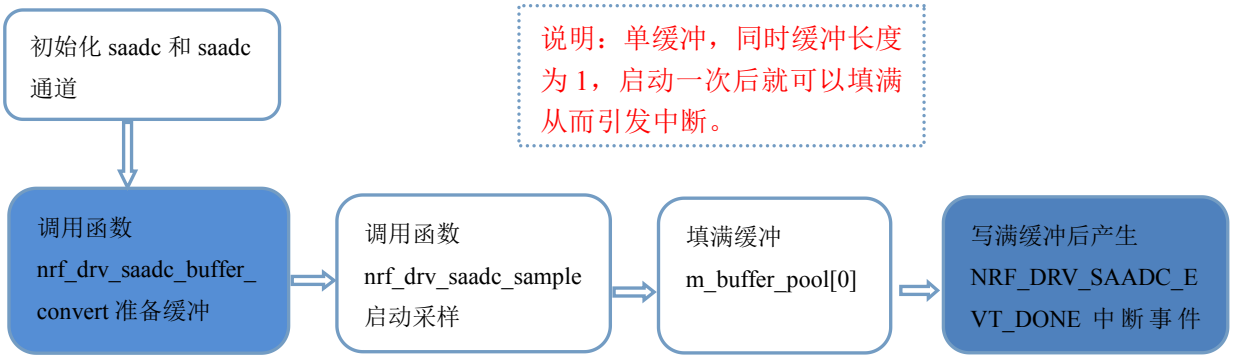
1：EasyDMA 作为 adc 采样的专用通道，负责把采样的数据发送给 RAM 内的结果寄存器 RESULT。

2：根据结果寄存器 RESULT 内的 RESULT.PTR 寄存器和 RESULT.MAXCNT 寄存器，分别决定了数据指针和这个数据指针的大小。

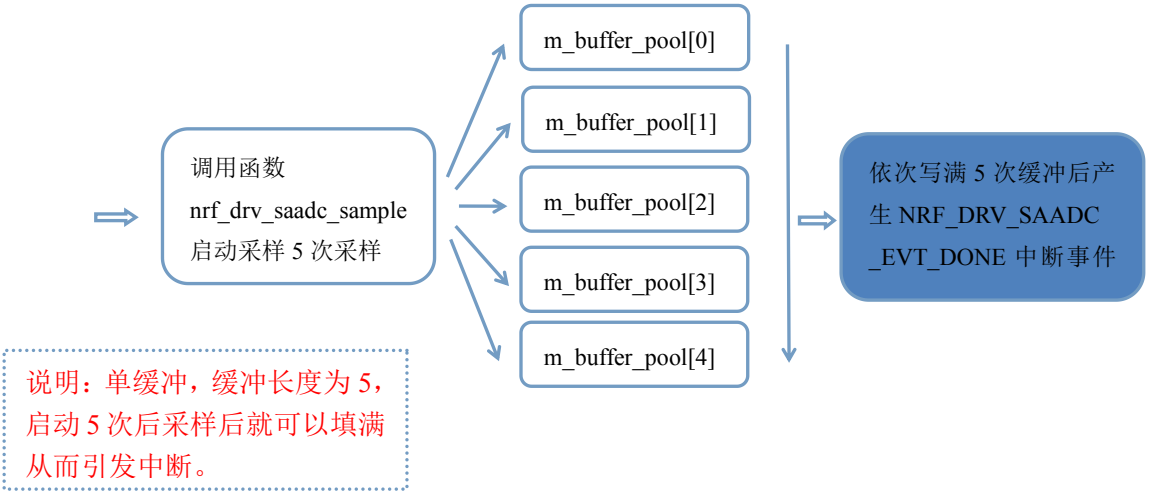
3：只有填满这个数据指针内所有的空间，才能触发中断把转换数据读出，因此整个转换次数=通道数*buff 缓冲大小。

例 1：单通道采集

① 假设采样数据缓冲为 1，通道为 1：

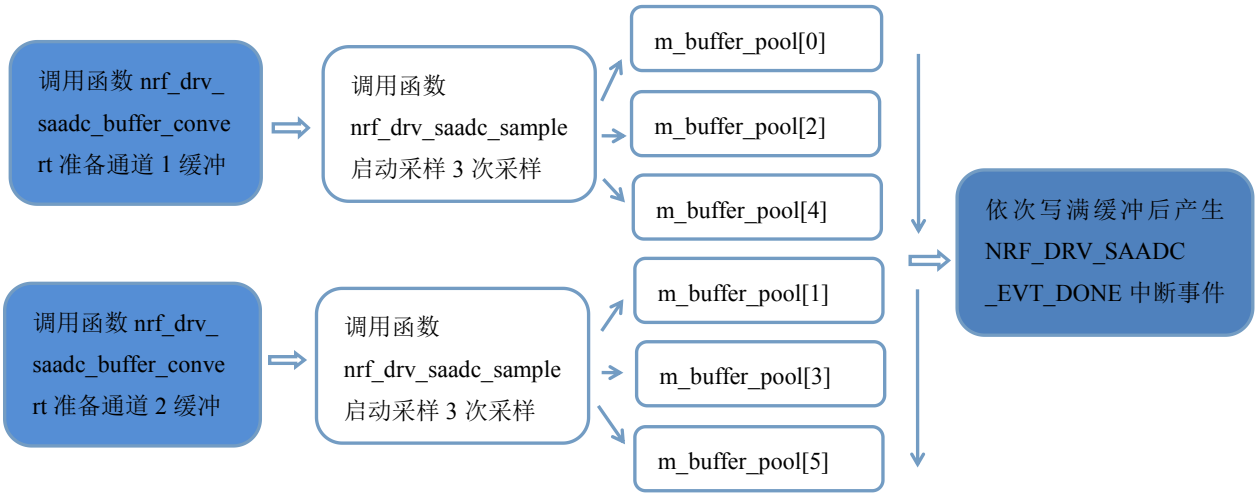


② 假设采样数据缓冲为 5，通道为 1：



例 2：多通道采集

③ 假设采样数据缓冲为 3，通道为 2：



二：实例编程

例 1：单通道采集

已单端输入为例，文件开头做一个宏定义：

```
//定义缓冲大小为 1.
#define SAMPLES_IN_BUFFER 1
//定义缓冲数值
static nrf_saadc_value_t          m_buffer_pool[SAMPLES_IN_BUFFER];
//定义采样次数
static uint32_t                   m_adc_evt_counter;
```

下面我们首先来进行 saadc 的初始化配置，设置代码，初始化通道的部分和前面的配置相同，只是需要多添加缓冲配置函数，如下所示：

```
 void saadc_init(void)
{
    ret_code_t err_code;
   //adc 通道配置
    nrf_saadc_channel_config_t channel_config =
    NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_AIN2);
   //adc 初始化
    err_code = nrf_drv_saadc_init(NULL, saadc_callback);
    APP_ERROR_CHECK(err_code);
   //adc 通道初始化
    err_code = nrf_drv_saadc_channel_init(0, &channel_config);
    APP_ERROR_CHECK(err_code);
    //添加缓冲配置函数：
     err_code = nrf_drv_saadc_buffer_convert(m_buffer_pool,SAMPLES_IN_BUFFER);
    APP_ERROR_CHECK(err_code);
}

```

添加单缓冲如红色字体显示内容，之后在主函数中启动采样，调用 nrf_drv_saadc_sample()函数，开始采样。

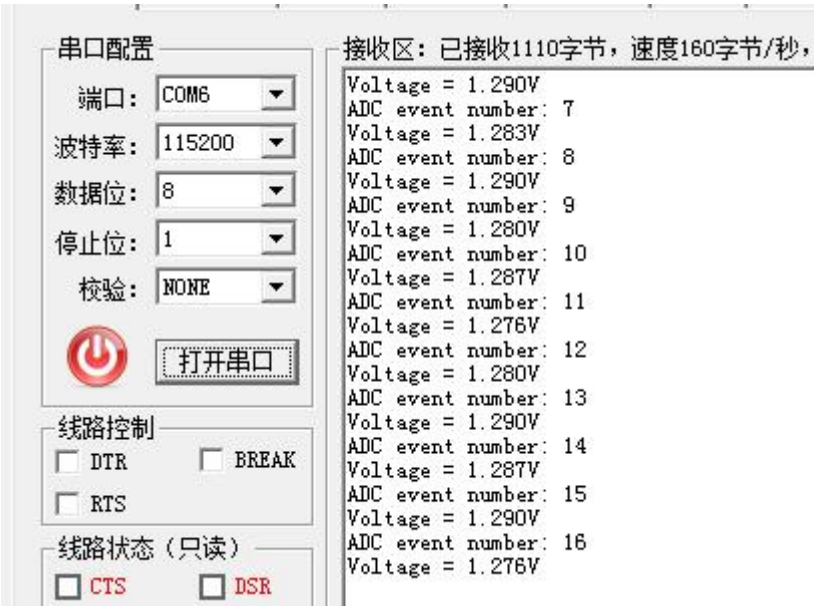
```
int main(void)
{
    uart_config();
    printf("\n\rSAADC HAL simple example.\r\n");
    saadc_init();

    while(1)
    {
   //启动一次 ADC 采样。
            nrf_drv_saadc_sample();
            //启动一次 ADC 采样。
            //延时 300ms，方便观察 SAADC 采样数据
```

```
35.                 nrf_delay_ms(300);
36.         }
37. }
```

启动后，触发 adc 采样中断，在中断中进行判断是否缓冲填满，缓冲填满后会启动 NRF_DRV_SAADC_EVT_DONE 事件，同时串口输出转换后的采样电压，内容如下所示：

```
01. void saadc_callback(nrf_drv_saadc_evt_t const * p_event)
02. {
03.         float    val;
04.           //判断是否发送填满缓冲事件，如何发送表示本次采样完成
05.         if (p_event->type == NRF_DRV_SAADC_EVT_DONE)
06.         {
07.                 ret_code_t err_code;
08.               //设置好缓存，为下次转换预备缓冲
09.                 err_code =
    nrf_drv_saadc_buffer_convert(p_event->data.done.p_buffer,SAMPLES_IN_BUFFER);
10.                 APP_ERROR_CHECK(err_code);
11.
12.                 int i;
13.               //打印输出采样次数
14.                 printf("ADC event number: %d\r\n",(int)m_adc_evt_counter);
15.                 for (i = 0; i < SAMPLES_IN_BUFFER; i++)
16.                 {
17.             //打印输出，通过前面的转换公式进行电压的转换计算。
18.                     val = p_event->data.done.p_buffer[i] * 3.6 /1024;
19.                           printf("Voltage = %.3fV\r\n", val);
20.                 }
21.               //采样次数加 1
22.                 m_adc_evt_counter++;
23.         }
24. }
```

打开串口助手，输出的实验现象如下，可以和万用表对比测量结果，同时采样次数也跟着计数：



如果我们改变下最开头的宏定义，把缓冲大小改成 5，那么根据上面的分析，要采样 5 次才会有事件输出，如下变动：

```
38. //定义缓冲大小为 5
39. #define SAMPLES_IN_BUFFER 1
40. //定义缓冲数值
41. static nrf_saadc_value_t          m_buffer_pool[SAMPLES_IN_BUFFER];
42. //定义采样次数
43. static uint32_t                   m_adc_evt_counter;
```

打开串口助手，输出的实验现象如下，采样次数每次都会依次输出 5 个采样电压值，如下所示：

例 2：多通道采集

下面来实现多通道的采集，首先假设我们才有 2 个输入通道，每个通道分配 3 个缓冲大小，首先头文件设置如下：

```
//定义缓冲大小为 6
#define SAMPLES_IN_BUFFER  6
//定义缓冲数值
static nrf_saadc_value_t          m_buffer_pool[SAMPLES_IN_BUFFER];
//定义采样次数
static uint32_t                   m_adc_evt_counter;
```

2 通道，每个通道 3 个缓冲，所以定义的缓冲总数为 2*3=6，然后再 adc 的定义初始化，初始化中需要定义两个通道，如下所示：

```
void saadc_init(void)
{
    ret_code_t err_code;
  //配置通道 0，输入管脚为 AIN2
    nrf_saadc_channel_config_t channel_0_config =
        NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_AIN2);
  //配置通道 1，输入管脚为 AIN0
     nrf_saadc_channel_config_t channel_1_config =
        NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_AIN0);
    //adc 初始化
    err_code = nrf_drv_saadc_init(NULL, saadc_callback);
    APP_ERROR_CHECK(err_code);
    //adc 通道初始化，带入前面的通道配置结构体
    err_code = nrf_drv_saadc_channel_init(0, &channel_0_config);
    APP_ERROR_CHECK(err_code);
    err_code = nrf_drv_saadc_channel_init(1, &channel_1_config);
    APP_ERROR_CHECK(err_code);
   //添加缓冲配置函数：
   err_code = nrf_drv_saadc_buffer_convert(m_buffer_pool,SAMPLES_IN_BUFFER);
    APP_ERROR_CHECK(err_code);
}
```

对比单通道配置和多通道配置，区别就如上红色标示部分，需要配置多个通道，进行多通道的初始化过程。配置好后，主函数的设置没有任何变化，直接调用启动 adc 采样函数，代码如下：

```
int main(void)
{
    uart_config();
    printf("\n\rSAADC HAL simple example.\r\n");
    saadc_init();//调用 adc 初始化函数
    while(1)
    {
   //启动一次 ADC 采样。
            nrf_drv_saadc_sample();
            //启动一次 ADC 采样。
            //延时 300ms，方便观察 SAADC 采样数据
            nrf_delay_ms(300);
    }
```

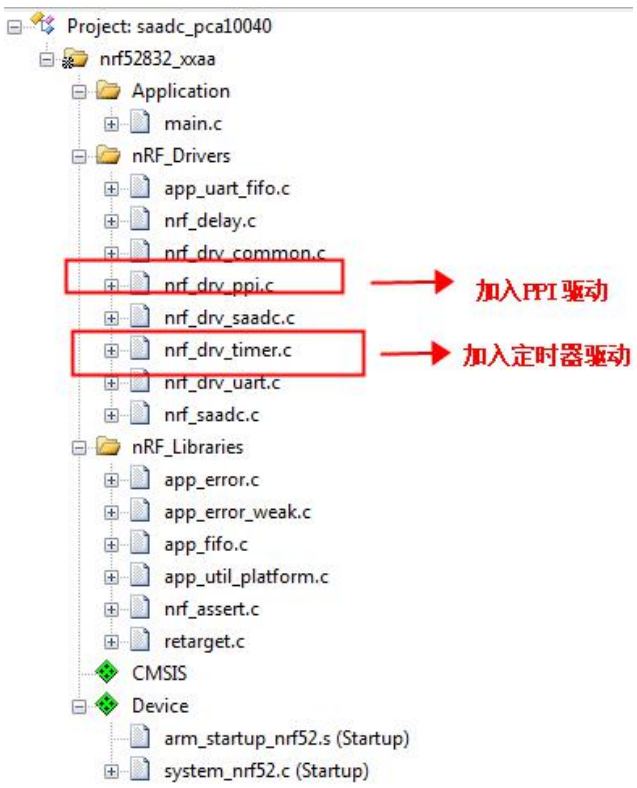
```
14. }
```

打开串口助手，输出的实验现象如下，缓冲大小为 6，所以采样次数每次都会依次输出 6 个采样电压值，如下所示，而两个通道的采样值依次交错输出：

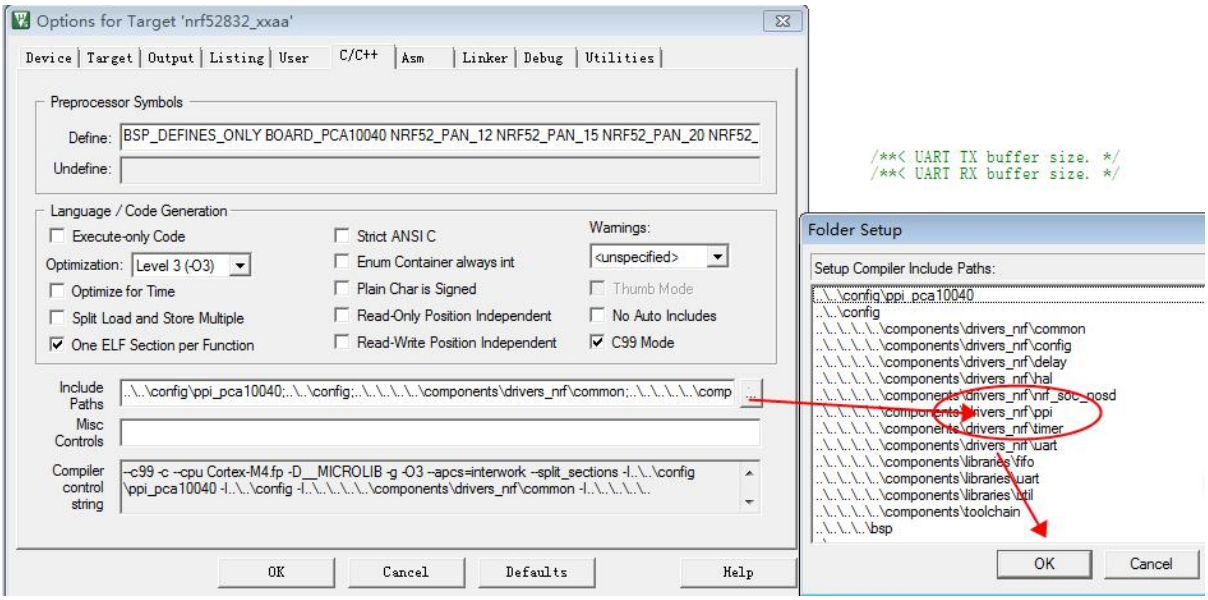
6.8.2.4 PPI 启动双缓冲中断采样

最后我们来讨论一个综合应用，结合了前面的定时器和 PPI 的功能，减少 CPU 的参与，提供转换效率的方法，也就是这讲的基于 PPI 启动的双缓冲中断采样法。我们前面的几种方法，都需要在主函数 main 中启动 adc 的采样，事件上是一种占用了 CPU 的，为了提搞我们的系统工作效率，我们把启动 adc 采样事件的这个工作丢给定时器和 PPI 通道来完成，CPU 不需要去参与。同时我们还采样官方推荐的双缓冲方式来存储与输出采样结果。

作为一个综合应用，我们用到了前面讲的 PPI 和定时器的知识，大家需要提前温习下前面的内容，首先是工程目录树如下：



图中，需要在之前的工程目录中添加 PPI 的驱动文件和定时器的驱动文件，然后再 main 主函数的头文件中调用#include "nrf_drv_ppi.h"和#include "nrf_drv_timer.h"，然后再路径中添加这两个启动的路径：



完成上面的工作后，我们来配置代码。配置代码要实现几个功能：

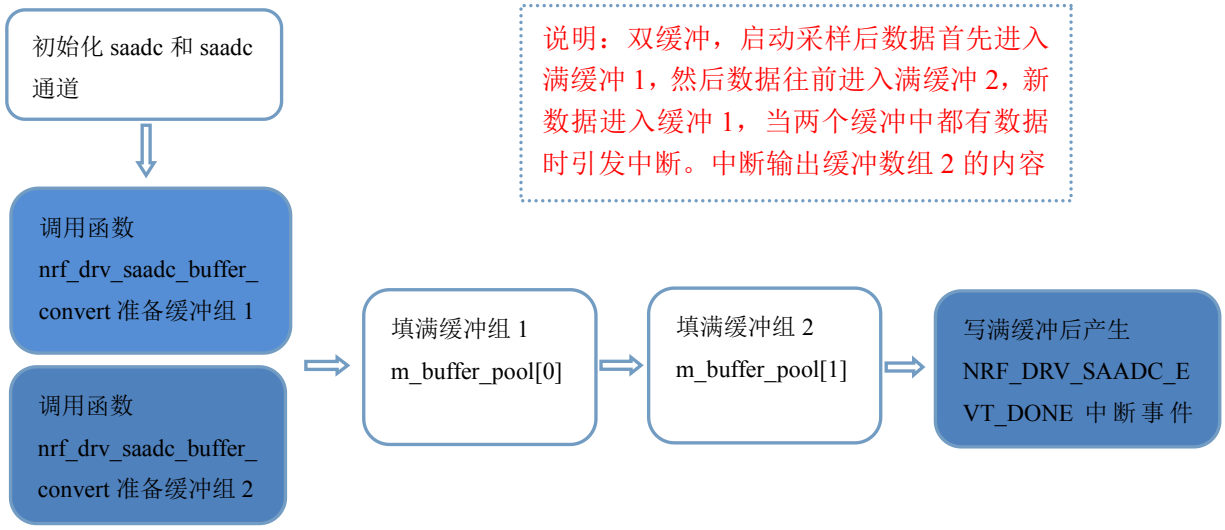
1：adc 的初始化，配置好 adc 的通道，缓冲大小和缓冲个数。

2：启动 adc 采样，本讲的核心是把启动工作丢给 PPI 和定时器处理，所以这里好配置定时器定时事件和 PPI 的触发通道。

3：最后就是 adc 采样完成后触发中断，中断中输出采集的数据。

下面就按照这 3 步来配置代码。

首先谈一下双缓冲 adc 的设置，双缓冲是在前面单缓冲的基础上实现的一个工作机制，也就是把缓冲的数组变成两个，数据依次进入缓冲数组 1 和缓冲数组 2，当两个数组内都有数据就会触发中断事件发生，中断后输出缓冲数组 2 内的内容：

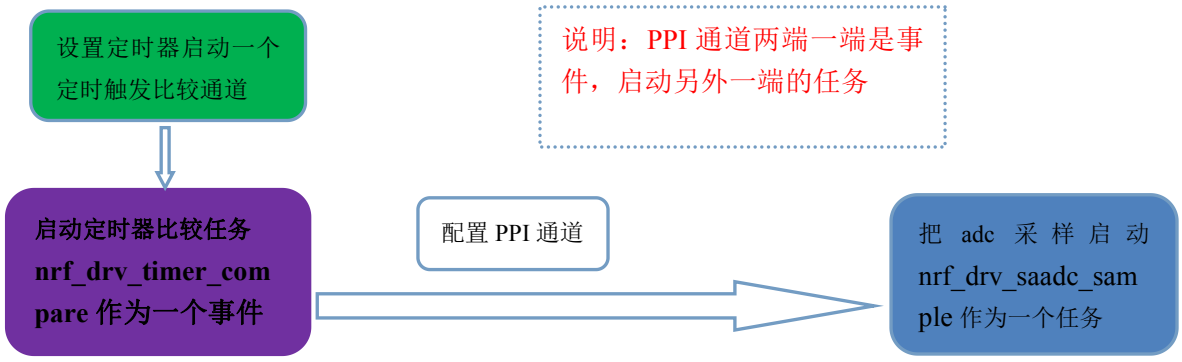


那么代码配置和单缓冲的区别如下所示，多一个缓冲数组配置：

```
15. void saadc_init(void)
16. {
17.     ret_code_t err_code;
18.     nrf_saadc_channel_config_t channel_config =
19.      //配置通道参数
    NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_AIN2);
20.     //adc 初始化
21.     err_code = nrf_drv_saadc_init(NULL, saadc_callback);
22.     APP_ERROR_CHECK(err_code);
23.     //带入通道配置参数
24.     err_code = nrf_drv_saadc_channel_init(0, &channel_config);
25.     APP_ERROR_CHECK(err_code);
26.     //配置第一个缓冲
27.     err_code = nrf_drv_saadc_buffer_convert(m_buffer_pool[0],SAMPLES_IN_BUFFER);
28.     APP_ERROR_CHECK(err_code);
29.     //配置第二个缓冲
30.     err_code = nrf_drv_saadc_buffer_convert(m_buffer_pool[1],SAMPLES_IN_BUFFER);
31.     APP_ERROR_CHECK(err_code);
32. }
33.
```

再谈下第二个问题，adc 的采样启动，如何通过 PPI 启动 adc 的采样。前面的学习我们知道 PPI 实际上是一个通道，通道不能启动任何外设，打个比方：比如你在马路上，并不能自动的走，马路

只是一个通道，要车或者步行才能动。PPI 一样的是这个功能，我们要通过其他的外设来启动 adc 采集，一般采用定时器定时来启动 adc，原理如下图所示：



其基本原理还是比较简单的，我们首先配置一个定时器，定一个时间，比如 400ms，触发一次定时器比较发生。然后设置一个 PPI 通道，把定时器比较这个作为一个事件，作为 PPI 通道的一端。另外一端把启动 adc 的采样作为任务，作为 PPI 的另外一端。当 400ms 后，会通过定时器比较事情启动 adc 的采样开始。代码具体配置如下

```
34. //使能 PPI 通道
35. void saadc_sampling_event_enable(void)
36. {
37.     ret_code_t err_code = nrf_drv_ppi_channel_enable(m_ppi_channel);
38.     APP_ERROR_CHECK(err_code);
39. }
40.
41. void saadc_sampling_event_init(void)
42. {
43.     ret_code_t err_code;
44.     err_code = nrf_drv_ppi_init();
45.     APP_ERROR_CHECK(err_code);
46.     //定时器初始化
47.     err_code = nrf_drv_timer_init(&m_timer, NULL, timer_handler);
48.     APP_ERROR_CHECK(err_code);
49.
50.     //设置 每 400ms 发送一次 m_timer 比较事件
51.     uint32_t ticks = nrf_drv_timer_ms_to_ticks(&m_timer, 400);
52.   //设置定时 ，捕获/比较通道，比较值，清除比较任务，关掉比较器中断
53.     nrf_drv_timer_extended_compare(&m_timer, NRF_TIMER_CC_CHANNEL0, ticks,
    NRF_TIMER_SHORT_COMPARE0_CLEAR_MASK, false);
54.     nrf_drv_timer_enable(&m_timer);
55.
56.    //是设置 PPI 两端的通道，一个作为任务，一个作为事件
57.     uint32_t timer_compare_event_addr = nrf_drv_timer_compare_event_address_get(&m_timer,
    NRF_TIMER_CC_CHANNEL0);
58.     uint32_t saadc_sample_event_addr = nrf_drv_saadc_sample_task_get();
```

```
59.         //分频一个 PPI 通道
60.         err_code = nrf_drv_ppi_channel_alloc(&m_ppi_channel);
61.         APP_ERROR_CHECK(err_code);
62.         //分频 PPI 通道地址，一端是比较事件，一端是 adc 采样事件
63.         err_code = nrf_drv_ppi_channel_assign(m_ppi_channel, timer_compare_event_addr,
    saadc_sample_event_addr);
64.         APP_ERROR_CHECK(err_code);
65. }
```

第三步就是 adc 中断触发后进行数据输出，这个和前面的 adc 中断采集的内容一致，没有做任何变化，代码如下：

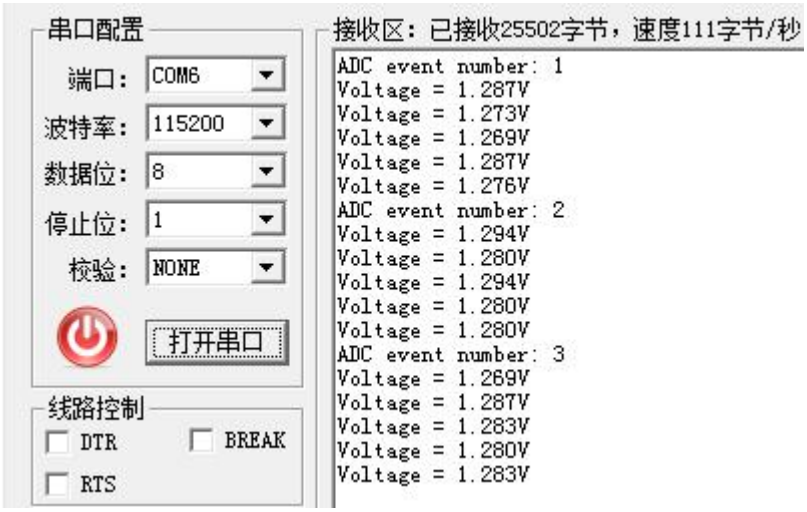
```
66. //adc 中断输出
67. void saadc_callback(nrf_drv_saadc_evt_t const * p_event)
68. {           float    val;
69.         if (p_event->type == NRF_DRV_SAADC_EVT_DONE)
70.         {
71.             ret_code_t err_code;
72.          //设置好缓存，为下次转换预备缓冲
73.             err_code = nrf_drv_saadc_buffer_convert(p_event->data.done.p_buffer,
    SAMPLES_IN_BUFFER);
74.             APP_ERROR_CHECK(err_code);
75.
76.             int i;
77.              //打印输出采样次数
78.             printf("ADC event number: %d\r\n",(int)m_adc_evt_counter);
79.              //输出采样值
80.             for (i = 0; i < SAMPLES_IN_BUFFER; i++)
81.             {
82. //                  printf("%d\r\n", p_event->data.done.p_buffer[i]);
83.                                 val = p_event->data.done.p_buffer[i] * 3.6 /1024;
84.                   printf("Voltage = %.3fV\r\n", val);
85.             }
86.             m_adc_evt_counter++;
87.         }
88. }
```

在主函数中，CPU 得到了解放，可以不做任何操作，初始化完成后直接等待结果输出，这种方式比较适合移植到协议栈下，代码如下：

```
89. int main(void)
90. {
91.         uart_config();
92.
93.         printf("\n\rSAADC HAL simple example.\r\n");
```

```
94.         saadc_sampling_event_init();
95.   //adc 初始化
96.         saadc_init();
97.         saadc_sampling_event_enable();
98.
99.         while(1)
100.         {
101.
102.         }
103. }
```

打开串口助手，输出的实验现象如下，缓冲大小为 5，如下所示，每次采样输出 5 个采样值：



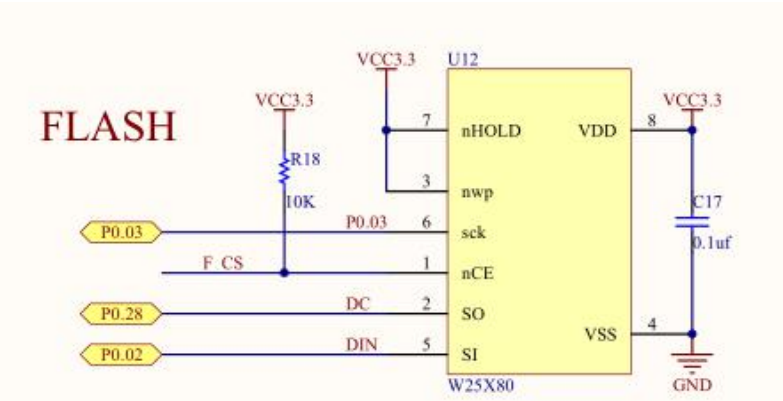
6.9 SPI 读写外部 FLASH

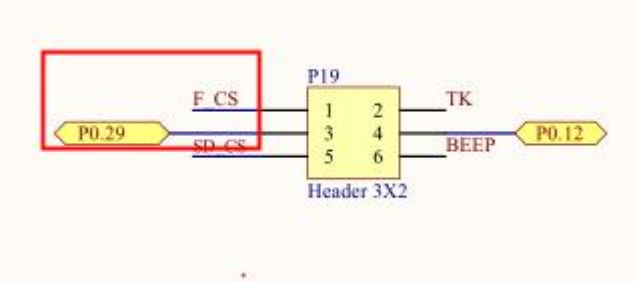
6.9.1 原理分析

SPI(Serial Peripheral Interface--串行外设接口)总线系统是一种同步串行外设接口，它可以使 MCU 与各种外围设备以串行方式进行通信以交换信息。SPI 有三个寄存器分别为：控制寄存器 SPCR，状态寄存器 SPSR，数据寄存器 SPDR。外围设备包括 FLASHRAM、网络控制器、LCD 显示驱动器、A/D 转换器和 MCU 等。SPI 总线系统可直接与各个厂家生产的多种标准外围器件直接接口，该接口一般使用 4 条线：串行时钟线（SCLK）、主机输入/从机输出数据线 MISO、主机输出/从机输入数据线 MOSI 和低电平有效的从机选择线 SS(有的 SPI 接口芯片带有中断信号线 INT、有的 SPI 接口芯片没有主机输出/从机输入数据线 MOSI)。 本实验通过 SPI 读写串行 FLASH ,串行 FLASH 采样 W25X16。

6.9.2 硬件准备：

硬件配置入下图所示，在 flash w25q16 和 SD 卡共用一个 SPI 接口,通过 SPI 的片选段进行选择，电路图如下所示：





用跳线帽把端口 P0.29 和 FLASH 的片选端 F_CS 短接，如上图所示，那么整个端口分配如下：

MISO:端口 P0.28

MOSI:端口 P0.02

SCK: 端口 P0.03

F_CS:端口 P0.29

解释：CS:FLASH 片选信号引脚。

SCK:FLASH 时钟信号引脚。

MISO:FLASH 主入从出引脚。

MOSI：FLASH 主出从进引脚。

硬件按照如上方式连接后，下面来配置驱动程序。

6.9.3 应用实例编写

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 arm4 文件夹中的工程项目如下：

在工程中，两个和本项目相关的驱动文件分别为：nrf_drv_spi.c 和 w25q16.c 两个驱动文件，其中 nrf_drv_spi.c 为 spi 硬件的驱动函数集合，w25q16.c 这个文件是基于 spi 硬件的 FLASH 驱动文件。这两个函数需要加入工程项目中，同时注意配置驱动路径。其中 w25q16.c 文件是我们需要单独编写的。下面重点就是关于该驱动的编写：

下面我们首先来进行 spi 的初始化配置，设置代码如下 ：

```
#define SPI_INSTANCE  0 /**< SPI instance index. */
#define SPI_CS_PIN    29  /**< SPI CS Pin.*/

static volatile bool spi_xfer_done;  //SPI 数据传输完成标志
static const nrf_drv_spi_t spi = NRF_DRV_SPI_INSTANCE(SPI_INSTANCE);  /**< SPI 使用的配置端 */

static uint8_t       spi_tx_buf[256];     /**< TX buffer 缓冲. */
static uint8_t       spi_rx_buf[256];     /**< RX buffer 缓冲. */
/** SPI 中断处理函数
 */
void spi_event_handler(nrf_drv_spi_evt_t const * p_event)
{
    spi_xfer_done = true;
}
//SPI 初始化端口
```

```
16. void hal_spi_init(void)
17. {
18.  nrf_drv_spi_config_t spi_config = NRF_DRV_SPI_DEFAULT_CONFIG(SPI_INSTANCE);
19.    spi_config.ss_pin = SPI_CS_PIN;
20.    APP_ERROR_CHECK(nrf_drv_spi_init(&spi, &spi_config, spi_event_handler));
21.
22. }
```

第 1 行：设置 SPI_INSTANCE 位使用了第几个 SPI 端口。

第 2 行：宏定义 SPI_CS_PIN 为 FLASH 的从设备的片选端口。

第 4 行：宏定义一个标志位：spi_xfer_done，SPI 触发中断事件的时候会被置位

第 5 行：配置 SPI 端口参数。

NRF_DRV_SPI_INSTANCE(SPI_INSTANCE)，其中 SPI_INSTANCE 第一行被定义为 0，那么表示为：NRF_DRV_SPI_INSTANCE(0)，该函数定义的 4 个结构体，如下所示

```
#define NRF_DRV_SPI_INSTANCE(id)                                    \
{                                                                   \
    .p_registers   = NRF_DRV_SPI_PERIPHERAL(id),                    \
    .irq           = CONCAT_3(SPI, id, _IRQ),                       \
    .drv_inst_idx = CONCAT_3(SPI, id, _INSTANCE_INDEX), \
    .use_easy_dma = CONCAT_3(SPI, id, _USE_EASY_DMA)      \
}
```

第一句 NRF_DRV_SPI_PERIPHERAL(id)判断是否使能了了 EASY_DMA,如果使能了就配置的是 SPIM0 寄存器，如果没有使能，配置的是 SPI0 寄存器。

```
#define NRF_DRV_SPI_PERIPHERAL(id)                \
    (CONCAT_3(SPI, id, _USE_EASY_DMA) == 1 ? \
        (void *)CONCAT_2(NRF_SPIM, id)          \
      : (void *)CONCAT_2(NRF_SPI, id))
```

第二句，优先级采用 SPI0_IRQ 的定义；

第三句，设置 SPI 的驱动引索；

第四句，设置了 SPI 如果要使用 EASY_DMA 为 SPI0_USE_EASY_DMA

上面的二~四这三个定义在 Config.h 文件中定义配置了：

```
207 #define SPI0_ENABLED 1
208
209 #if (SPI0_ENABLED == 1)
210 #define SPI0_USE_EASY_DMA 0
211
212 #define SPI0_CONFIG_SCK_PIN          3
213 #define SPI0_CONFIG_MOSI_PIN         2
214 #define SPI0_CONFIG_MISO_PIN         28
215 #define SPI0_CONFIG_IRQ_PRIORITY     APP_IRQ_PRIORITY_HIGH
216
217 #define SPI0_INSTANCE_INDEX 0
218 #endif
```

第 7~8 行：设置 TX buffer 缓冲和 RX buffer 缓冲的大小。

第 11~14 行：设置 SPI 的中断处理函数 spi_event_handler，当发生中断后，标志位：spi_xfer_done 会被置位 1。

第 18~20 行：通过函数 nrf_drv_spi_init 把配置的 SPI 产生，导入到&spi 指针中，同时这中断处理函数。其中配置 spi 的基本参数如下：

```
#define   NRF_DRV_SPI_DEFAULT_CONFIG(id)                                \
```

```
{                                                                   \
    .sck_pin          = CONCAT_3(SPI, id, _CONFIG_SCK_PIN),         \
    .mosi_pin         = CONCAT_3(SPI, id, _CONFIG_MOSI_PIN),        \
    .miso_pin         = CONCAT_3(SPI, id, _CONFIG_MISO_PIN),        \
    .ss_pin           = NRF_DRV_SPI_PIN_NOT_USED,                   \
    .irq_priority = CONCAT_3(SPI, id, _CONFIG_IRQ_PRIORITY), \
    .orc              = 0xFF,                                       \
    .frequency        = NRF_DRV_SPI_FREQ_500K,                      \
    .mode             = NRF_DRV_SPI_MODE_0,                         \
    .bit_order        = NRF_DRV_SPI_BIT_ORDER_MSB_FIRST,            \
}
```

前四个.sck_pin ， .mosi_pin ，.miso_pin ，.ss_pin 配置 SPI 管脚，第 5 个参数配置 SPI 优先级，第六个参数配置接收后自动发送的参数，第七个参数配置 SPI 时钟频率，第八个参数配置 SPI 的模式。第八个参数配置发送 BIT 高位在先还是地位在先。

初始化后，开始编写 读和写 W25X16 的代码，时序关系我们需要参考 w25x16 的数据手册：

首先通过 SPI 接口发送 0xFF 字节，同时接收数据，实现读取一个字节：

```
uint8_t SpiFlash_ReadOneByte(void)
{
    uint8_t len = 1;

    spi_tx_buf[0] = 0xFF;//发送的数据
  spi_xfer_done = false;
  APP_ERROR_CHECK(nrf_drv_spi_transfer(&spi, spi_tx_buf, len, spi_rx_buf, len));//发送数据
  while(!spi_xfer_done)
        ;
    return (spi_rx_buf[0]);//接收返回
```

nrf_drv_spi_transfer 函数为 SPI 传送函数，表示如下所示：

```
ret_code_t nrf_drv_spi_transfer(nrf_drv_spi_t const * const p_instance,
                                uint8_t const * p_tx_buffer,
                                uint8_t           tx_buffer_length,
                                uint8_t         * p_rx_buffer,
                                uint8_t           rx_buffer_length);
```

形参 1: nrf_drv_spi_t const * const p_instance：表示前面配置的 SPI 通道参数指针

形参 2: const * p_tx_buffer：表示 SPI 发送缓冲

形参 3: tx_buffer_length：表示 SPI 发送缓冲的长度

形参 4: * p_rx_buffer：表示 SPI 接收缓冲

形参 5: rx_buffer_length:表示 SPI 接收缓冲的长度

通过 SPI 接口发送任意的 uint8_t Dat 字节，函数如下所示：

```
void SpiFlash_WriteOneByte(uint8_t Dat)
{
    uint8_t len = 1;

```

```
    spi_tx_buf[0] = Dat;
    spi_xfer_done = false;
    APP_ERROR_CHECK(nrf_drv_spi_transfer(&spi, spi_tx_buf, len, spi_rx_buf, len));
    while(!spi_xfer_done)
        ;
}
```

通过 SPI 接口向 W25Q16 写命令，时序见 W25Q16 的参考手册，函数如下所示：

```
uint8_t SpiFlash_Write_CMD(uint8_t *CMD)
{
    uint8_t len = 3;

    spi_tx_buf[0] = *CMD;
    spi_tx_buf[1] = *(CMD+1);
    spi_tx_buf[2] = *(CMD+2);
    spi_xfer_done = false;
    APP_ERROR_CHECK(nrf_drv_spi_transfer(&spi, spi_tx_buf, len, spi_rx_buf, len));
    while(!spi_xfer_done)
        ;
    return RET_SUCCESS;
}
```

实现了以上三个几次操作后，就可以实现 FLASH 的页写，页擦除等操作，这里就不累述了。

主函数简单验证下功能，通过按键 1 按下写入一个 good 数据，然后读出来，对比写入和读出数据是否一致，判断是否正确，现象通过串口输出实现观察。

```
int main(void)
{
    uint8_t i;

    nrf_gpio_cfg_output(LED_1);//配置 P0.21 为输出，驱动指示灯 D1
    nrf_gpio_pin_set(LED_1);    //D1 初始状态设置为熄灭

    nrf_gpio_range_cfg_input(BUTTON_START,BUTTON_STOP,NRF_GPIO_PIN_PULLUP);
    //配置 P0.17~P0.20 为输入
    uart_init();        //串口初始化
    hal_spi_init();   //SPI 初始化
    nrf_delay_ms(100);
    printf("...start\r\n");
    nrf_delay_ms(800);
    while (true)
    {
        if(nrf_gpio_pin_read(BUTTON_1) == 0) //按键 S1 按下
        {
            nrf_delay_ms(10);                    //延时去抖动
            if(nrf_gpio_pin_read(BUTTON_1) == 0)//确认按键 S1 按下
```

```
21.                         {
22.                             nrf_gpio_pin_clear(LED_1);
23.                           SPIFlash_Erase_Sector(0,0);      //写之前必须先擦除
24.                            nrf_delay_ms(100);
25.                           SpiFlash_Write_Page(Tx_Buffer,0x00,5);//写入 5 个字节数据
26.                         for(i=0; i<5;i++)printf(" %c",(uint8_t)Tx_Buffer[i]);
27.                            //串口打印从 FLASH 读出的数据
28.                       for(i=0; i<5;i++)Rx_Buffer[i] = 0; //清零 Flash_WR_Buf
29.                           SpiFlash_Read(Rx_Buffer,0x00,5);          //读出 5 个字节数据
30.                           printf("Read data = ");
31.                           for(i=0; i<5;i++)printf(" %c",(uint8_t)Rx_Buffer[i]);
32.                        //串口打印从 FLASH 读出的数据
33.                             printf("\r\n");   //回车换行
34.                       while(nrf_gpio_pin_read(BUTTON_1) == 0);//等待按键释放
35.                       nrf_gpio_pin_set(LED_1);
36.                         }
37.             }
38.
```

6.9.4 实验现象

打开串口调试助手，设置为 115200 波特率，下载程序后，按下按键 1，如果输出如下，表示写入成功：