第六章 GPIO 端口的应用

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 nrf52832 处理器 GPIO 性能分析

6.1.1 GPIO 端口资源描述

GPIO 称为输入输出端口，根据封装最大具有 32 个 I/O 口，可以通过 P0 这样一个端口访问和控制多达 32 个端口。而且每个端口都可以独立访问。其特点如下：

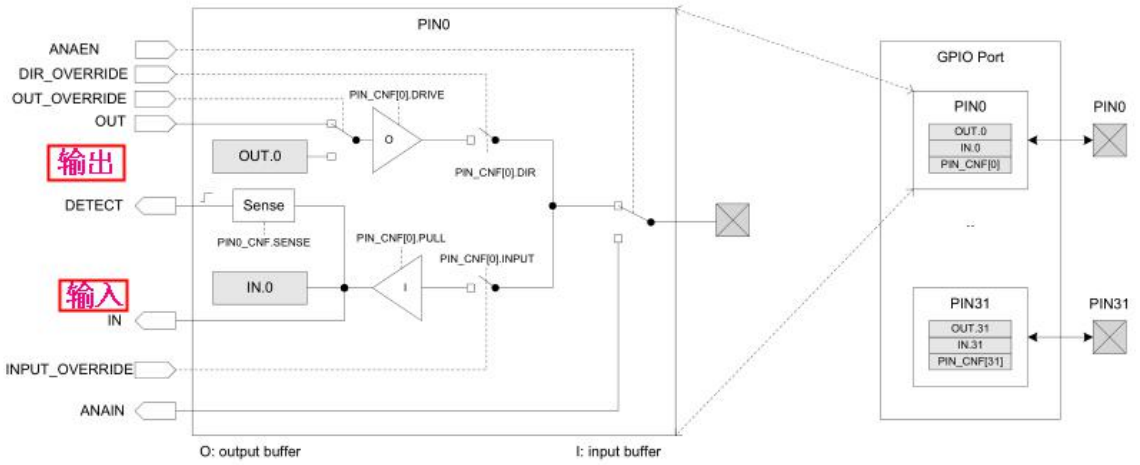
- 最大 32 个 GPIO，分别为 P0.0~P0.31
- 具有 8 个带有模拟通道的 GPIO 口，可以用于 SAADC、COMP 和 LPCOMP 输入
- 可以配置的输入驱动强度；内部具有上拉和下拉电阻
- 可以从所有的引脚上的高电平或者低电平触发唤醒
- 任何引脚的状态变化都可以触发中断
- PPI 任务/事件系统可以使用所有引脚
- 可以通过 PPI 和 GPIOTE 通道控制一个或多个 GPIO 输出
- 所有引脚都可以单独映射到外设接口上，以实现布局灵活性
- 在 SENSE 信号上捕获的 GPIO 状态变化可以由 LATCH 寄存器存储

GPIO 端口外设最多可实现 32 个引脚。这些引脚中的每一个都可以在 PIN_CNF [n]寄存器（n = 0..31）中单独配置。可以通过这些寄存器配置以下参数：

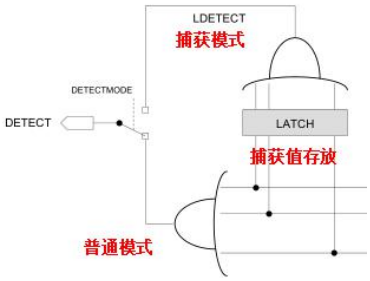
- 方向
- 驱动强度
- 启用上拉和下拉电阻
- 引脚感应
- 输入缓冲区断开
- 模拟输入（适用于所选引脚）

nrf52832 芯片内核为 ARM Cortex M4，其 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。nRF52832 的 IO 管脚可以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：



当在任何这样配置的引脚上检测到正确的电平时，感测机制将 DETECT 信号设置为高电平。每个引脚都有一个单独的 DETECT 信号，DETECTMODE 寄存器定义的默认行为是来自 GPIO 端口中所有引脚的 DETECT 信号被合并到一个普通的 DETECT 信号。如果在启用感测机制时满足 PIN_CNF 寄存器中配置的感测条件，则检测将立即变为高电平。如果在启用感测机制之前 DETECT 信号为低，这将触发 PORT 事件。PORT 事件在后面 GPIOE 中再详细讲解。

6.1.2 GPIO 软件编程

实际 nRF52832 中所包含的寄存器是非常的简单的，使用下面表进行的说明：

寄存器名称	地址偏移	R/W	功能描述
OUT	0x504	读/写	设置端口输出
OUTSET	0x508	读/写	置位端口输出高电平，写 0 无效
OUTCLR	0x50C	读/写	置位端口输出低电平，写 0 无效
IN	0x510	只读	设置端口输入
DIR	0x514	读/写	设置端口方向
DIRSET	0x518	读/写	置位端口为输入，写 0 无效
DIRCLR	0x51C	读/写	置位端口为输出，写 0 无效
LATCH	0x520	读/写	传感锁存寄存器：指示哪些 GPIO 引脚符合 PIN_CNF[n] .SENSE 寄存器中设置的条件
DETECTMODE	0x524	读/写	传感模式选择
PIN_CNF[n]n=0~31	0x700~0x77C	读/写	对应端口号 0 到 31 的端口设置

通过寄存器配置 GPIO 端口的输出十分简单，官方提供了一个库，对寄存器进行了封装。我们可以很简单的进行调用，下面将结合寄存器和官方的库函数进行说明。

1：GPIO 端口状态的设置：

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。对比数据手册上关于 PIN_CNF[n]寄存器的描述：

位数	Field	Value ID	Value	描述
第 0 位	DIR	Input	0	设置为输入引脚
		Output	1	设置为输出引脚
第 1 位	INPUT	Connect	0	连接输入缓冲
		Disconnect	1	断开输入缓冲
第 2~3 位	PULL	Disable	0	没有上下拉
		Pulldown	1	开启内部下拉
		Pullup	3	开启内部上拉
第 8~10 位	DRIVE	S0S1	0	标准 '0'，标准 '1'
		H0S1	1	高驱动 '0'，标准 '1'
		S0H1	2	标准 '0'，高驱动 '1'
		H0H1	3	高驱动'0'，高驱动'0'
		D0S1	4	断开'0'，标准'0'（通常用于有线或连接）
		D0H1	5	断开'0'，高驱动'1'（通常用于有线或连接）
		S0D1	6	标准'0'。断开'1'（通常用于有线和连接）
		H0D1	7	高驱动'0'，断开'1'（通常用于有线和连接）
第 16~17 位	SENSE	Disable	0	关闭感应
		High	2	高电平感应
		Low	3	低电平感应

●第 0 位为 DIR 位，设置 IO 管脚为输入引脚或者输出引脚。当为 0 的时候为输入硬件，当设置为 1 的时候为输出管脚。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 口方向的结构体 nrf_gpio_port_dir_t，这里完全是对照参考手册进行编写的 ：

```
01. #define   GPIO_PIN_CNF_DIR_Input      (0UL) /*配置管脚为输入管脚 */
02. #define   GPIO_PIN_CNF_DIR_Output     (1UL) /*配置管脚为输出管脚*/
```

```
03.
04. typedef enum
05. {
06. NRF_GPIO_PIN_DIR_INPUT    = GPIO_PIN_CNF_DIR_Input,   //输入
07. NRF_GPIO_PIN_DIR_OUTPUT = GPIO_PIN_CNF_DIR_Output //输出
08. } nrf_gpio_port_dir_t;
```

●第 1 位为 INPUT 为，如果设置了 IO 端口为输入端口，该位用于设置输入是否接入输入缓冲，由于复位默认为 1，所有默认是断开连接输入缓冲的。在"nrf_gpio.h 库文件中找到设置 INPUT 的结构体：

```
01. #define GPIO_PIN_CNF_INPUT_Connect (0UL)       /*连接输入缓冲 */
02. #define GPIO_PIN_CNF_INPUT_Disconnect (1UL)   /*断开输入缓冲 */
03. typedef enum
04. {
05.   NRF_GPIO_PIN_INPUT_CONNECT= GPIO_PIN_CNF_INPUT_Connect,     //连接输入缓冲
    NRF_GPIO_PIN_INPUT_DISCONNECT = GPIO_PIN_CNF_INPUT_Disconnect //断开输入缓冲
06. }   nrf_gpio_pin_input_t;
```

●第 2~3 位为输入是否开上拉电阻。当设置为 0 的时候没有上下拉，设置为 1 的时候为开下拉，设置为 3 的时候为开上拉。在"nrf_gpio.h 库文件中找到设置 PULL 的结构体如下所示：

```
01. #define GPIO_PIN_CNF_PULL_Disabled (0UL) /*  没有上下拉*/
02. #define GPIO_PIN_CNF_PULL_Pulldown (1UL) /*开下拉  */
03. #define GPIO_PIN_CNF_PULL_Pullup (3UL) /*开上拉*/
04.
05. typedef enum
06. {
07.       NRF_GPIO_PIN_NOPULL     = GPIO_PIN_CNF_PULL_Disabled, //关闭上下拉
08.       NRF_GPIO_PIN_PULLDOWN = GPIO_PIN_CNF_PULL_Pulldown, //端口 pull-down 使能
09.       NRF_GPIO_PIN_PULLUP     = GPIO_PIN_CNF_PULL_Pullup,     //端口 pull-up 使能
10. }   nrf_gpio_pin_pull_t;
```

●第 8~10 位为 DRIVE 位，该位为设置 IO 端口输出的驱动强度的，对应输出 IO 端口输出强度可以编程进行配置。

比如把 DRIVE 位设置为 0，为 S0S1 方式。这种方式下当 IO 端口输出为 0 的时候，IO 端口输出强度为标准驱动能力，当 IO 端口输出为 1 的时候，IO 端口输出强度也为标准驱动能力。

GPIO 端口的所谓的标准驱动能力、高驱动能力在，在芯片手册 P149 页 IO 口电气特性有说明，如下表所示：

Symbol	Description	Min.	Typ.	Max.	Units
V_{IH}	输入电压高	0.7×VDD		VDD	V
V_{IL}	输入低电压	VSS		0.3×VDD	V
$V_{OH,SD}$	输出高电压，标准驱动，0.5 毫安，VDD≥1.7	VDD-0.4		VDD	V
$V_{OH,HDH}$	输出高电压，高驱动器，5 毫安，VDD> = 2.7 V	VDD-0.4		VDD	V
$V_{OH,HDL}$	输出高电压，高驱动器，3 毫安，VDD> = 1.7 V	VDD-0.4		VDD	V
$V_{OL,SD}$	输出低电压，标准驱动，0.5 毫安，VDD≥1.7	VSS		VSS + 0.4	V
$V_{OL,HDH}$	输出低电压，高驱动器，5 毫安，VDD> = 2.7 V	VSS		VSS + 0.4	V
$V_{OL,HDL}$	输出低电压，高驱动器，3 毫安，VDD> = 1.7 V	VSS		VSS + 0.4	V

输出高电压就表示输出逻辑为’1’，输出低电压就表示输出逻辑为’0’。标准驱动能力和高驱

动能力的驱动电流值是不同的。因此在不同的 DRIVE 位配置之下，输出高低电压的驱动标准都是有区别的，读者可以根据上表配置自己所需的驱动能力。在"nrf_gpio.h 库文件中找到设置 DRIVE 的结构体如下所示：

```
01. typedef enum
02. {
03. NRF_GPIO_PIN_S0S1 = GPIO_PIN_CNF_DRIVE_S0S1, //标准 '0'，标准 '1'
04. NRF_GPIO_PIN_H0S1 = GPIO_PIN_CNF_DRIVE_H0S1, //高驱动 '0'，标准 '1'
05. NRF_GPIO_PIN_S0H1 = GPIO_PIN_CNF_DRIVE_S0H1, //标准 '0'，高驱动 '1'
06. NRF_GPIO_PIN_H0H1 = GPIO_PIN_CNF_DRIVE_H0H1, //高驱动'0'，高驱动'0'
07. NRF_GPIO_PIN_D0S1 = GPIO_PIN_CNF_DRIVE_D0S1, //断开'0'，标准'0'（通常用于有线或连接）
08. NRF_GPIO_PIN_D0H1 = GPIO_PIN_CNF_DRIVE_D0H1, //断开'0'，高驱动'1'（通常用于有线或
    连接）
09. NRF_GPIO_PIN_S0D1 = GPIO_PIN_CNF_DRIVE_S0D1, //标准'0'。断开'1'（通常用于有线和连接）
10. NRF_GPIO_PIN_H0D1 = GPIO_PIN_CNF_DRIVE_H0D1, //高驱动'0'，断开'1'（通常用于有线和
    连接）
11. } nrf_gpio_pin_drive_t;
```

特别注意：数据手册上特别提醒蓝牙无线电性能参数，比如灵敏度，可能会受到高频数字 I/O 的影响。因此接近无线电电源和天线引脚接收器端的 GPIO 端口，设计的时候注意驱动能力和数字频率都不要太高。因此开发板上的 QFN48 封装芯片建议如下管脚配置为低驱动能力和低频 I/O：

27	P0.22	低驱动，低频 I/O
28	P0.23	
29	P0.24	
37	P0.25	
38	P0.26	
39	P0.27	
40	P0.28	
41	P0.29	
42	P0.30	
43	P0.31	

●第 16~17 位位 SENSE，该位为感应设置为，可以设置感应外部信号，常用与 GPIOE 唤醒中使用，在"nrf_gpio.h 库文件中找到设置 DRIVE 的结构体如下所示：

```
01. #define GPIO_PIN_CNF_SENSE_Disabled (0UL) /*关闭感应能力 */
02. #define GPIO_PIN_CNF_SENSE_High (2UL) /*设置高电平感应 */
03. #define GPIO_PIN_CNF_SENSE_Low (3UL) /*设置低电平感应*/
04.
05. typedef enum
06. {
07. NRF_GPIO_PIN_NOSENSE = GPIO_PIN_CNF_SENSE_Disabled, //关闭感应能力
08. NRF_GPIO_PIN_SENSE_LOW = GPIO_PIN_CNF_SENSE_Low, //设置低电平感应
09. NRF_GPIO_PIN_SENSE_HIGH = GPIO_PIN_CNF_SENSE_High, //设置高电平感应
10. } nrf_gpio_pin_sense_t;
```

关于 PIN_CNF[n]寄存器的描述就讲到这里，相信大家对 GPIO 管脚的配置有了初步的了解。

上面所有的参数都在库函数中封装了一个 nrf_gpio_cfg 函数，该函数就是来配置 PIN_CNF[n]寄存器的，通过调用这个函数容易就配置出 GPIO 的端口状态，代码如下所示：

```
01. __STATIC_INLINE void nrf_gpio_cfg(
02.         uint32_t                pin_number,
03.         nrf_gpio_pin_dir_t      dir,
```

```
04.       nrf_gpio_pin_input_t input,
05.       nrf_gpio_pin_pull_t    pull,
06.       nrf_gpio_pin_drive_t drive,
07.       nrf_gpio_pin_sense_t sense) //GPIO 端口状态的配置
08. {      //配置为那个端口
09.       NRF_GPIO_Type * reg = nrf_gpio_pin_port_decode(&pin_number);
10.        //配置对应端口的状态，包含了：
11.       reg->PIN_CNF[pin_number] = ((uint32_t)dir << GPIO_PIN_CNF_DIR_Pos)//方向
12.                                  | ((uint32_t)input << GPIO_PIN_CNF_INPUT_Pos)//输入缓冲
13.                                  | ((uint32_t)pull << GPIO_PIN_CNF_PULL_Pos)//上拉配置
14.                                | ((uint32_t)drive << GPIO_PIN_CNF_DRIVE_Pos)//驱动能力配置
15.                                 | ((uint32_t)sense << GPIO_PIN_CNF_SENSE_Pos);//感应能力
16. }
```

2：GPIO 端口状态配置的引申函数：

由 nrf_gpio_cfg 函数可以引申出库函数中多个驱动函数，比如 GPIO 的输出函数：

```
01. __STATIC_INLINE void nrf_gpio_cfg_output(uint32_t pin_number)
02. {
03.       nrf_gpio_cfg(
04.             pin_number,
05.             NRF_GPIO_PIN_DIR_OUTPUT,
06.             NRF_GPIO_PIN_INPUT_DISCONNECT,
07.             NRF_GPIO_PIN_NOPULL,
08.             NRF_GPIO_PIN_S0S1,
09.             NRF_GPIO_PIN_NOSENSE);
10. }
```

比如 GPIO 的输入函数：

```
01. __STATIC_INLINE void nrf_gpio_cfg_input(uint32_t pin_number, nrf_gpio_pin_pull_t pull_config)
02. {
03.       nrf_gpio_cfg(
04.             pin_number,
05.             NRF_GPIO_PIN_DIR_INPUT,
06.             NRF_GPIO_PIN_INPUT_CONNECT,
07.             pull_config,
08.             NRF_GPIO_PIN_S0S1,
09.             NRF_GPIO_PIN_NOSENSE);
10. }
```

由 nrf_gpio_cfg 函数引申出的函数比较多，这里就不一一的展开了，读者可以具体查看库文件，下面我们就在应用中展开说明如何进行使用。

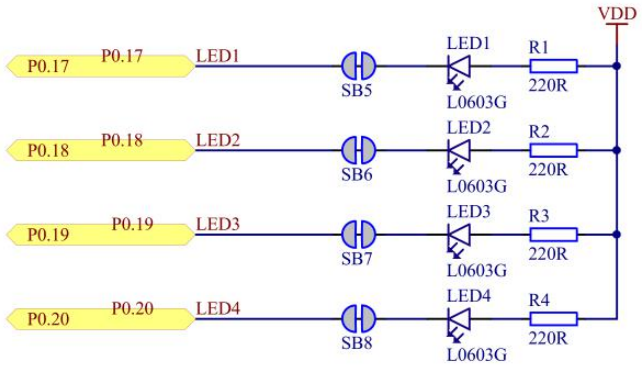
6.2 GPIO 输出应用

6.2.1 点亮第一个 LED 灯

6.2.1.1 硬件设计

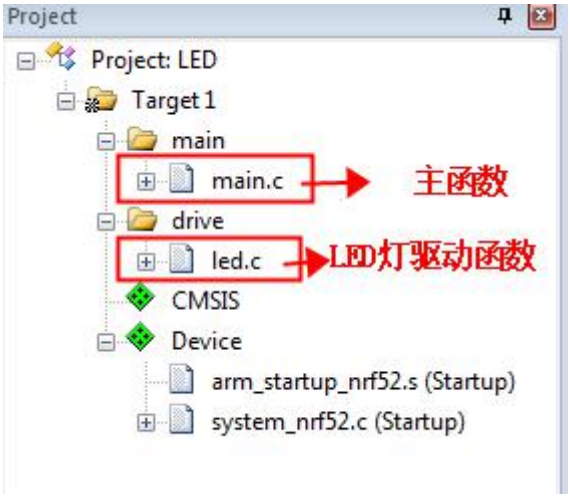
对于一个处理器来说，最简单的控制莫过于通过 I/O 端口输出的电平控制设备，本节就讲述一个经典的 LED 灯控制来开启 nRF52832 系列处理器的开发之旅。

硬件方面青风 nRF52832 开发板上，通过管脚 P0.17 到管脚 P0.20 连接 4 个 LED 灯，LED 另一端通过 220 欧电阻接到电源 VDD 上去，VDD 选择 3.3v 电源。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。



6.2.1.2 工程的搭建与编写

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 drive 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：

```
01. void LED_Init(void)
02. {
03.    nrf_gpio_cfg_output(LED_0);
04.    nrf_gpio_cfg_output(LED_1);
05.    nrf_gpio_cfg_output(LED_2);
06.    nrf_gpio_cfg_output(LED_3);//配置 IO 端口为输出状态
07. }
```

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：

```
01. void LED1_Open(void)//LED1 灯开灯
02. {
03.    nrf_gpio_pin_clear(LED_0);
04. }
05.
06. void LED1_Close(void)//LED1 灯关灯
07. {
08.  nrf_gpio_pin_set(LED_0);
09. }
10. void LED1_Toggle(void)//LED1 灯翻转
11. {
12.    nrf_gpio_pin_toggle(LED_0);
13. }
```

上面的代码分别调用了如下几个组件库的函数：

nrf_gpio_pin_clear(uint32_t pin_number)

nrf_gpio_pin_set (uint32_t pin_number));

nrf_gpio_pin_toggle(uint32_t pin_number));

这几个组件库的函数的封装非常简单，我们可以深入到函数内部，看看其是如何封装寄存器的操作的，比如我们打开 nrf_gpio_pin_clear(uint32_t pin_number)函数内部，会发现最终是封装的如下代码：

```
p_reg->OUTCLR = clr_mask;
```

也就是说要 IO 管脚输出为 0，实际上就是直接设置 OUTCLR 寄存器为高。同理 nrf_gpio_pin_set 函数就是设置 OUTSET 寄存器。nrf_gpio_pin_toggle 是同时设置了 OUTCLR 和 OUTSET 寄存器了。

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. {
```

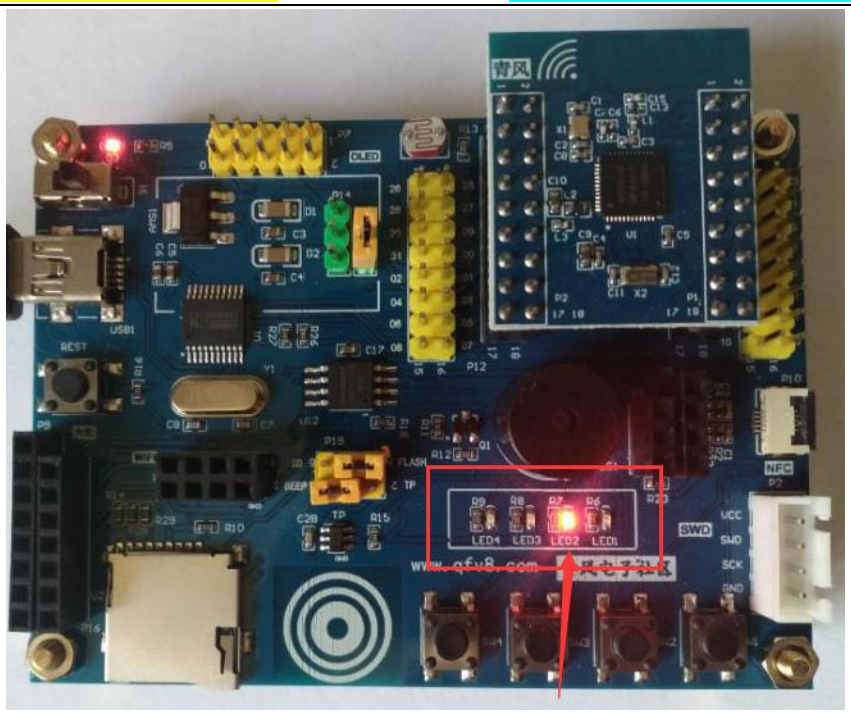
```
    //初始化 led 灯
    LED_Init();
    while(true)
    {
        LED1_Open();
        LED2_Close();
    }
}
```

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：

```
#include "nrf_delay.h"
#include "nrf_gpio.h"
#include "led.h"

int main(void)
{
    //初始化 led 灯
    LED_Init();
    while(true)
    {
        LED1_Open();
        LED2_Close();
        nrf_delay_ms(500); //延迟 500ms
        LED2_Open();
        LED1_Close();
        nrf_delay_ms(500);
    }
}
```

编译代码后，用 keil 把工程下载到青风 QY-nRF52832 蓝牙开发板上。运行后的效果如下图所示，LED 开始闪烁：

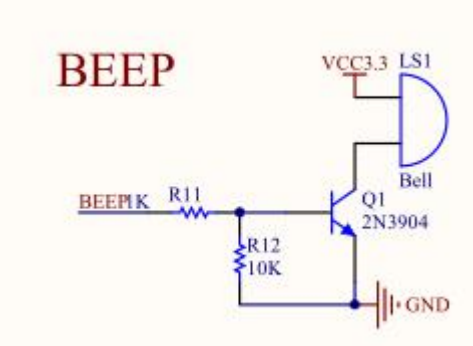


6.2.2 有源蜂鸣器的驱动

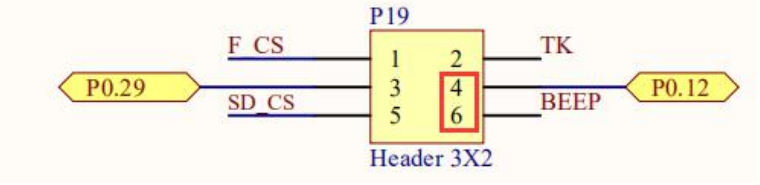
6.2.2.1 硬件设计

蜂鸣器常用的分为：有源蜂鸣器与无源蜂鸣器。这里的“源”不是指电源，而是指震荡源。也就是说，有源蜂鸣器内部带震荡源，所以只要一通电就会叫。而无源内部不带震荡源，所以如果用直流信号无法令其鸣叫，必须用 2K~5K 的方波去驱动它。有源蜂鸣器往往比无源的贵，就是因为里面多个震荡电路。

有源蜂鸣器的设计非常简单，如下上图所示：



1) 电路介绍：蜂鸣器一端接 VCC3.3V 电源，一端接 PNP 三极管 2N3904 的集电极。三极管的发射极接地。基极通过 R11 电阻后 BEEP 端接到 GPIO 端口。BEEP 端需要把 P19 上的跳线帽 4-6 进行短接。



2) 驱动原理：当 P0.12 输出高电平，三极管 PNP 基极会上电，三级管导通，此时蜂鸣器一端接地，一端接 VCC，此时形成一个电势差，就有电流通过蜂鸣器。蜂鸣器就会被驱动鸣叫。当 P0.12 输出低电平，PNP 三极管会被截止，三极管没有电流通过，停止鸣叫。

6.2.2.2 程序的编写

通过上面的分析，有源蜂鸣器的驱动实际上非常简单，直接对 GPIO 端口输出高电平，就可以使得蜂鸣器鸣叫。其基本原理和 LED 灯的驱动相同，因此我们可以在 led 灯的演示实例中进行编写。其工程目录树不变化，只需要在驱动文件中加入 BEEP 的驱动函数就可以了。

配置代码如下：

1.首先采用函数 nrf_gpio_cfg_output 设置 GPIO 为普通驱动能力的输出 GPIO；

2.打开蜂鸣器采用组件库函数 nrf_gpio_pin_set 函数，关闭蜂鸣器采用 nrf_gpio_pin_clear 函数；

3.主函数循环开关蜂鸣器，中间进行 800ms 的延迟；

具体代码实现如下所示：

```
#define      BEEP        12

   void BEEP_Init(void)
{
   //配置蜂鸣器驱动 GPIO
   nrf_gpio_cfg_output(BEEP);

}

void BEEP_Open(void)
{
   nrf_gpio_pin_set(BEEP);

}

void   BEEP_Close(void)
{
      nrf_gpio_pin_clear(BEEP);
}
```

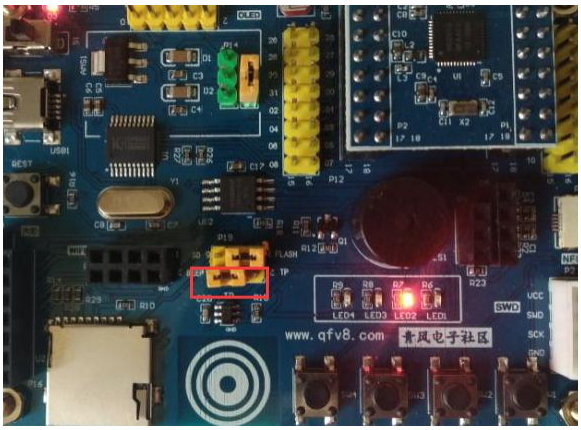
ii 主函数实现列表，注意头文件的引用：

```
#include <stdbool.h>
#include <stdint.h>
#include "nrf_delay.h"
#include "nrf_gpio.h"
```

```
#include "led.h"

int main(void)
{
    // 初始化 LED 灯
    LED_Init();
    BEEP_Init();//初始化蜂鸣器 IO 端口状态
    // 循环打开和关闭蜂鸣器
    while(true)
    {
        LED1_Toggle();
        BEEP_Open();
        nrf_delay_ms(800);
        BEEP_Close();
        LED1_Toggle();
       nrf_delay_ms(800);
    }
}
```

编译代码后，用 keil 把工程下载到青风 nRF52832EK 蓝牙开发板上。同时把 P19 上的跳线帽 4-6 进行短接如下图所示，运行后的效果如下图所示，LED 开始闪烁，同时蜂鸣器每隔 800ms 会鸣叫一次：



6.3 GPIO 输入应用

6.3.1 GPIO 输入扫描流程

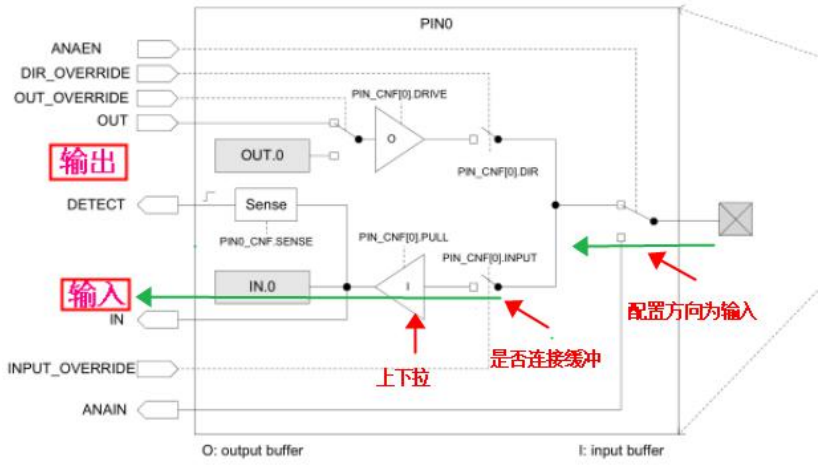
GPIO 的输入配置首先需要配置 PIN_CNF[n]寄存器的三个域，如 GPIO 结构体图所示；

1：PIN_CNF[n].DIR：该位决定 IO 端口方向为输入。

2：PIN_CNF[n].INPUT：该位决定是否接入缓冲。输入缓冲的作用和连接一个电阻类似，降低电压波动幅度过大对输入管脚的影响。

3：PIN_CNF[n].DIR：该位决定是否进行上下拉。上下拉的作用主要就是用于输入端口的电平维持，

保存信号稳定，方便处理器进行检测。



在 nrf52xx 的组件库中，提供了一个 API 函数 nrf_gpio_cfg_input 来完成 IO 口输入状态的设置，该函数就是调用前面讲解 nrf_gpio_cfg 函数，代码如下所示：

```
01.     __STATIC_INLINE void nrf_gpio_cfg_input(uint32_t pin_number, nrf_gpio_pin_pull_t
    pull_config)
02. {
03.     nrf_gpio_cfg(
04.         pin_number,//配置 IO 端口号
05.         NRF_GPIO_PIN_DIR_INPUT,//配置为输入
06.         NRF_GPIO_PIN_INPUT_CONNECT,//默认上接输入缓冲
07.         pull_config,
08.         NRF_GPIO_PIN_S0S1,
09.         NRF_GPIO_PIN_NOSENSE);
10. }
```

这个函数中形参为 pin_number 和 pull_config，也就是 IO 端口号和上下拉配置，可以调用该动态的进行修改这两个参数。而接输入缓冲被默认使能，如果开发者不需要接输入缓冲，可以对函数内部进行修改。

第二步，对输入信号进行判断，寄存器 IN 中提供了该判断功能，寄存器描述如下所示：

位数	Field	Value ID	Value	描述
第 0~31 位	IN[n] n=0~31	Low High	0 1	输入引脚位低电平 输入引脚为高电平

在 nrf52xx 的组件库中，对 IN 寄存器进行了封装，封装成库函数 nrf_gpio_pin_read 用于判断 IO 断开的状态，然后把对应状态进行返回。内部封装如下图所示：

```
__STATIC_INLINE uint32_t nrf_gpio_pin_read(uint32_t pin_number)
{
    NRF_GPIO_Type * reg = nrf_gpio_pin_port_decode(&pin_number);
    return ((nrf_gpio_port_in_read(reg) >> pin_number) & 1UL);
}

__STATIC_INLINE uint32_t nrf_gpio_port_in_read(NRF_GPIO_Type const * p_reg)
{
    return p_reg->IN;
}
```

nrf_gpio_pin_read 函数最后返回的就是 pin_number 管脚对应的 IN 寄存器的值，通过判断该值可以确定 IO 口的输入状态，比如下面一句：

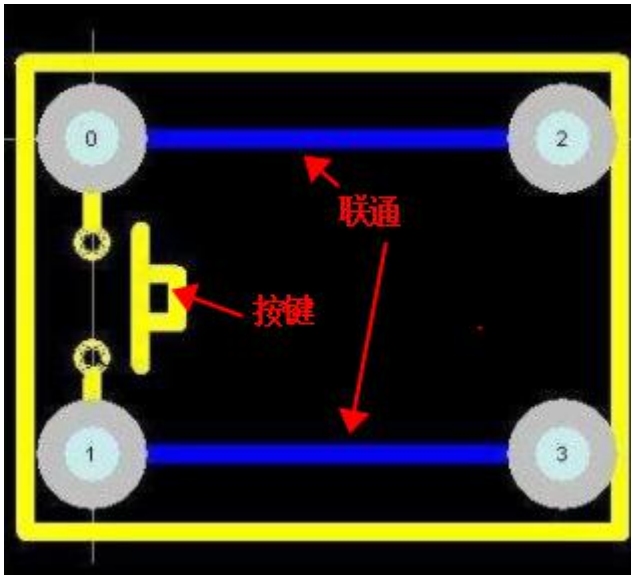
```
if(nrf_gpio_pin_read(13)== 0 )
```

表示判断 13 管脚是否输入为低电平信号。这里就基本描述了 IO 端口输入如何实现了。

6.3.2 机械按键输入扫描

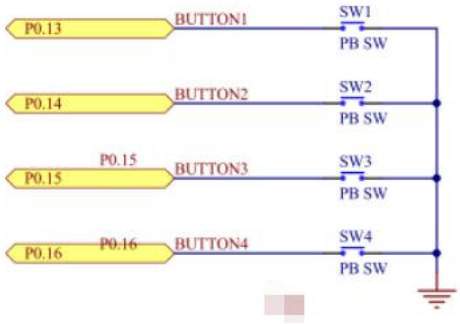
6.3.2.1 硬件设计

机械按键也称为又叫轻触按键，是一种电子开关。开发板上采用了四脚按键，其工作原理由常开触点、常闭触点组合而成。在四脚按键开关中，常开触点的作用，就是当压力向常开触点施压时，这个电路就呈现接通状态；当撤销这种压力的时候，就恢复到了原始的常闭触点，也就是所谓的断开。这个施压的力，就是用我们的手去开按钮、关按钮的动作。其截个图如下所示：



图中 0 和 2 管脚，1 和 3 管脚联通，当按下按键按钮后，0 就和 1 管脚接通，释放按键后，0 和 1 管脚断开。因此轻触按键 0 管脚可以接到地，1 管脚接 GPIO 输入。当按键被按下后，1 管脚会被拉低，作为一个低电平输入。

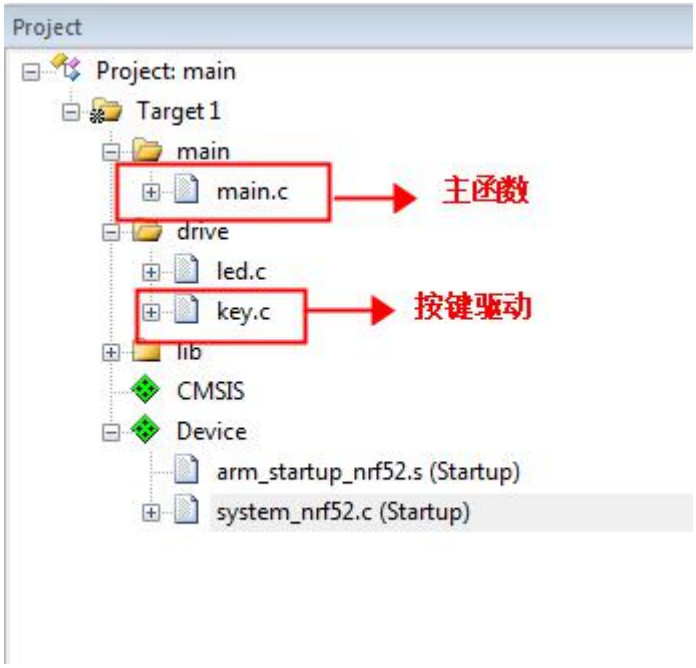
开发板上采用四个轻触按键的方式，四个轻触按键分别连接 P0.13、P0.14、P0.15、P0.16 四个 IO 端口上，轻触按键另外一端接地，电路如图所示：



6.3.2.2 程序的编写

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状态维持能力，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。

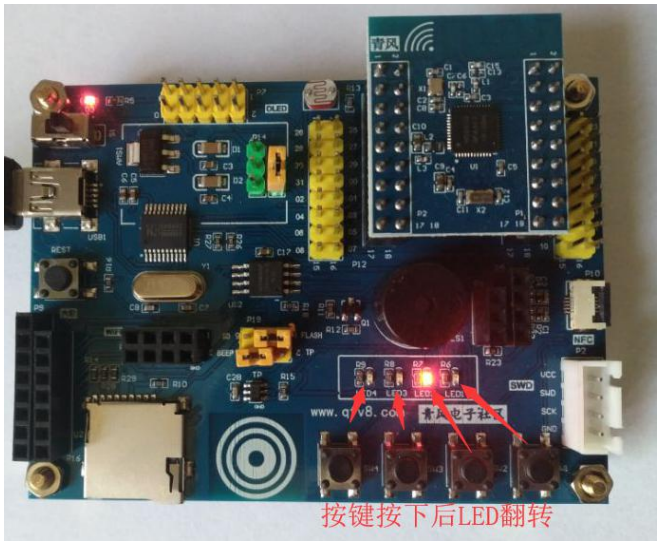
按键扫描时通过函数 nrf_gpio_pin_read 读取管脚状态，判断 IO 是否被拉低，如果被拉低就可以说明按键被按下了。机械按键最多问题就是抖动，这里同时为了防止按键抖动，加入一个软件延迟函数去抖。

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，使用 KEIL 下载到青云 nRF52832EK 开发板后的实验现象如下：下载后按下按键 1，LED1 灯翻转,下载后按下按键 2，LED2 灯翻转。



6.3.3 电容触摸按键的应用

6. 3. 3. 1 硬件设计

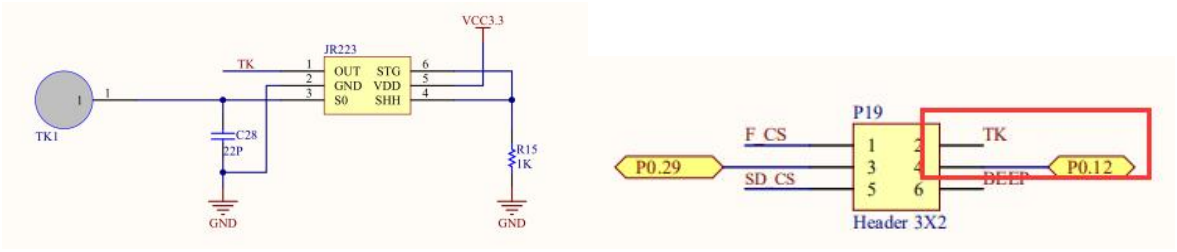
电容触摸按键可以穿透绝缘材料 20mm 以上，准确无误地侦测到手指的有效触摸。并保证了产品的灵敏度、稳定性、可靠性等不会因环境条件的改变或长期使用而发生变化，并具有防水和强抗干扰能力，超强防护，超强适应温度范围。电容式触摸按键没有任何机械部件，不会磨损，无限寿命，减少后期维护成本。电容式触摸按键面板图案、按键大小、形状任意设计，字符、商标、透视窗 LED 透光等任意搭配，外型美观、时尚，不褪色、不变形、经久耐用。因此在一下家电、手持设备上为了防止进水、腐蚀等情况发生，常常采用电容触摸按键。

电容触摸按键的触摸信号检测最简单的方式就是采用触摸芯片。开发板上采用 JR223 单键触摸式芯片。该芯片是电容式触摸按键专用检测传感器 IC。采用最新一代电荷检测技术，利用操作者的手指与触摸按键焊盘之间产生电荷电平来进行检测,通过监测电荷的微小变化来确定手指接近或者触摸到感应表面。对应 JR223 的芯片管脚表述如下列表：

管脚	名称	功能
1	OUT	触摸芯片输出端，接处理器的输入端
2	GND	接地端

3	SO	接触摸 TOUCH 信号
4	SLH	高低电平选择输出：SHL=0 高电平输出；SHL=1 低电平输出
5	VDD	电源，范围 2.0v~5.5v
6	STG	触发模式选择：STG=0 直接模式；STG=1 触发模式

电路设计如下图所示：管脚 1 接 TK 通过 P19 接 2 和 4 管脚，连接芯片 P0.12；VDD 接 3.3v 电源；STG 接高电平，设置为触发模式；SDH 接低电平，设置为高电平输出。触摸板 TK1 要接电容 C28 进行灵敏度调节，电容范围为 0~50p。



6.3.3.2 程序的编写

通过上面的分析，电容触摸按键的驱动实际上非常简单，直接对 GPIO 端口输入高电平，就可以被处理检测到。其基本原理和机械按键的驱动相同，只是检测电压为高电平。因此我们可以在按键扫描的的演示实例中进行编写。其工程目录树不变化，只需要在驱动文件中加入 TOUCH 的驱动函数就可以了。配置方式和机械按键相似，检测按键是否按下的时候判断 IO 口输入是否为高电平，代码具体如下所示：

```
01.  nrf_gpio_cfg_input(12,NRF_GPIO_PIN_PULLUP);//初始化 TOUCH 的管脚为输入
02.
03. uint8_t   TCH_Down(void)//电容触摸按键检测
04. {
05.          /*检测是否有按键按下  */
06.       if( nrf_gpio_pin_read(TCH)== 1 )
07.    {
08.           /*延时消抖*/
09.         Delay(10000);
10.            if(nrf_gpio_pin_read(TCH)== 1 )
11.                    {
12.                        /*等待按键释放  */
13.                        while(nrf_gpio_pin_read(TCH)== 1);
14.                        return     0    ;
15.                    }
16.            else
17.                    return 1;
18.       }
19. else
20.       return 1;
```

```
21. }
22.
23. int main(void)
24. {
25.     LED_Init();//led 初始化
26.     KEY_Init();//按键初始化
27.     LED1_Open();
28.     while(1)
29.     {
30.         if( TCH_Down()== 0)//判定电容触摸按键是否按下
31.  {
32.          LED4_Toggle();//led 灯翻转
33.  }
34.     }
35. }
```

实验编译后，使用 KEIL 下载到青云 nRF52832EK 开发板后的实验现象如下：短接 P19 接 2 和 4 管脚，下载后触摸电容触摸按键区域，LED41 灯翻转。