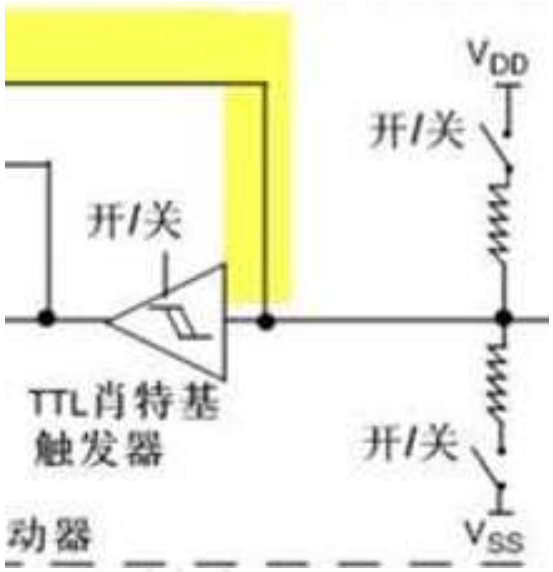
通用输出输出

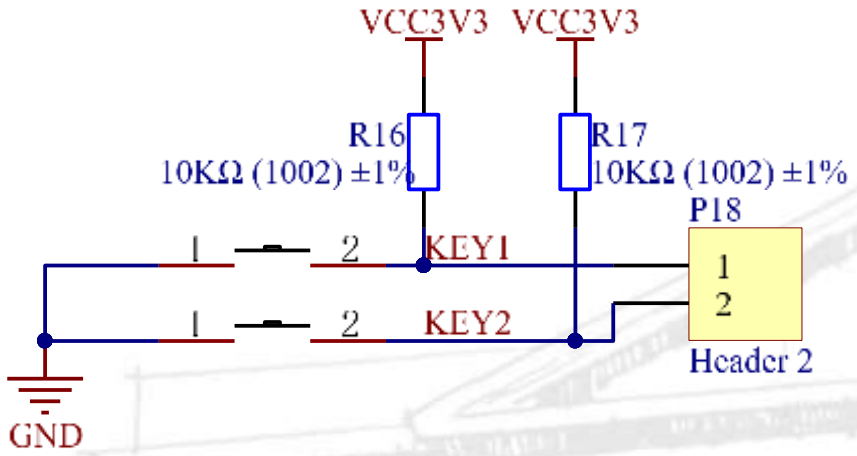
- 通用输入输出端口的基本概念
- STM32F401RB的GPIO
- 控制寄存器的访问
- 控制寄存器的功能
- 数据寄存器的功能与操作
- GPIO模块的使能与使用
- STM32CubeIDE初始化GPIO，并用HAL库操作GPIO
- GPIO模块的应用1 数码管
- GPIO模块的应用2 矩阵键盘
- 矩阵键盘和数码管的应用

- **通用输入输出接口（GPIO）**
  - GPIO=General Purpose Input Output，通用输入输出。有时候简称为“IO口”
  - MCU的最基本外设接口
- **端口（Port）：**
  - CPU和外设进行通信的媒介。STM32系列MCU的每个端口有16个引脚。
  - STM32F401RB有5个端口(Port A, Port B, Port C, Port D, Port H)
- **引脚（管脚,Pin）：**
  - MCU与外界连接的独立的导线，隶属于某一个端口
  - 对其中一个引脚进行操作的时候不会影响到其他引脚
  - 多种功能可以设置，可设置为数字输入或者数字输出，部分引脚还可以用作模拟输入和外设中断输入。

➢引脚的输入方式：

- **浮空输入**（上下两个电阻（上拉、下拉）上电开关都打开，输入信号必须有确定的电平，不能悬空）
- **上拉输入**（上面的电阻上电开关闭合，悬空时为高电平）
- **下拉输入**（下面的电阻上电开关闭合，悬空时为低电平）
- **模拟输入**（上下两个电阻上电开关都打开，接模拟信号）

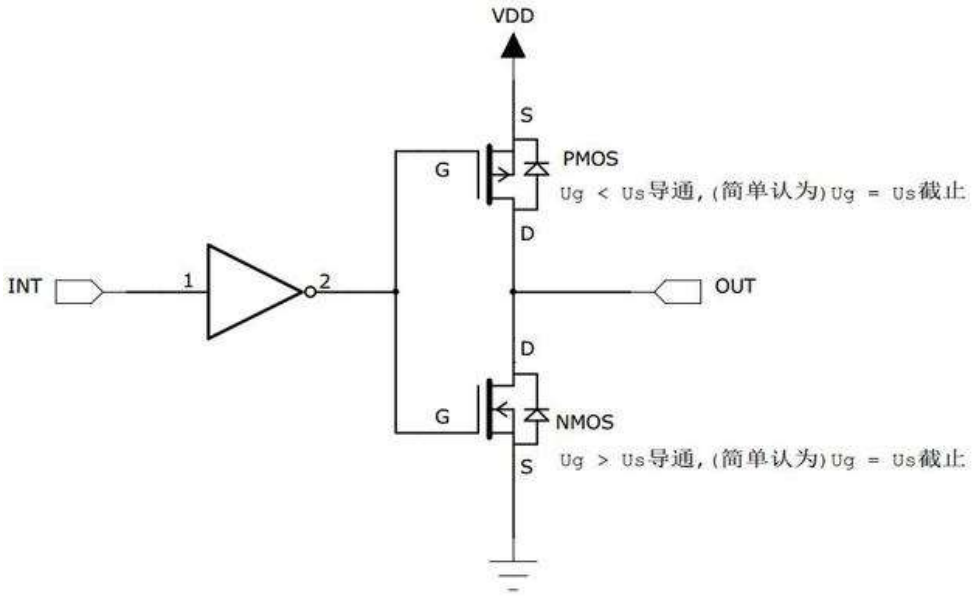




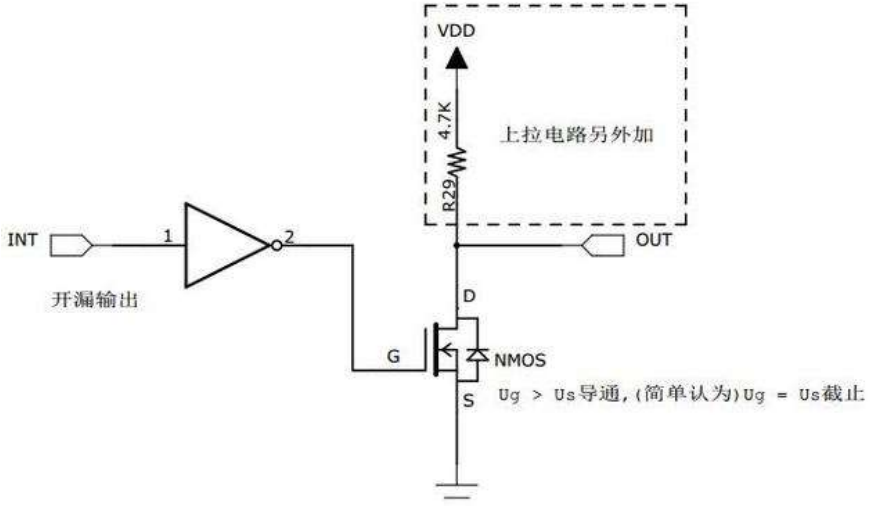
STM32F401RB开发板的简单按键，已经配置了上拉电阻

➢引脚的输出方式：

- **推挽输出**（高电平VDD，低电平GND）
- **开漏输出**（需要上拉电阻才能工作，高电平可以为VDD以下的任意电压，低电平GND，可用于5V系统与3.3V系统连接）



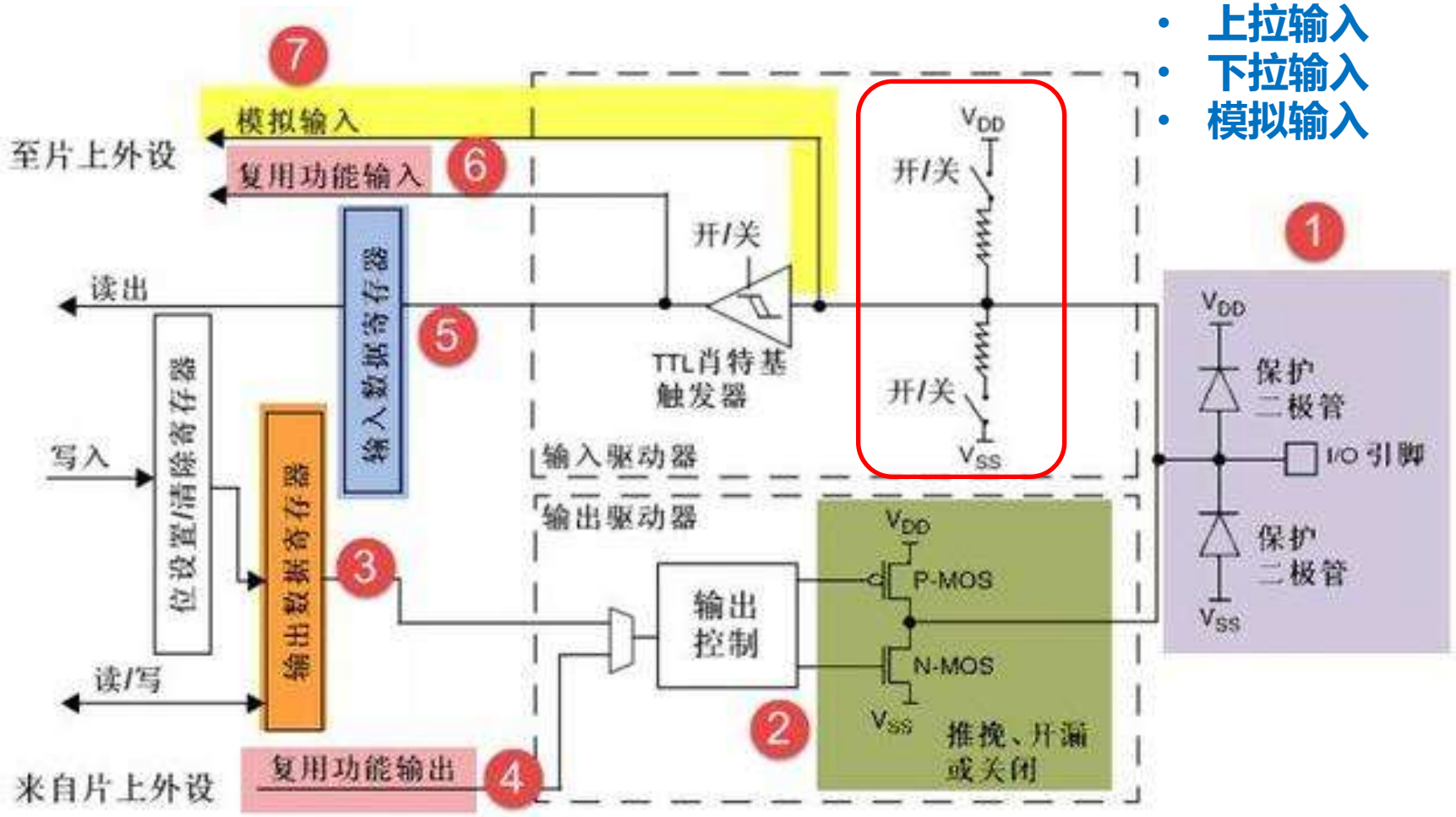
推挽输出



开漏输出

输入电路：

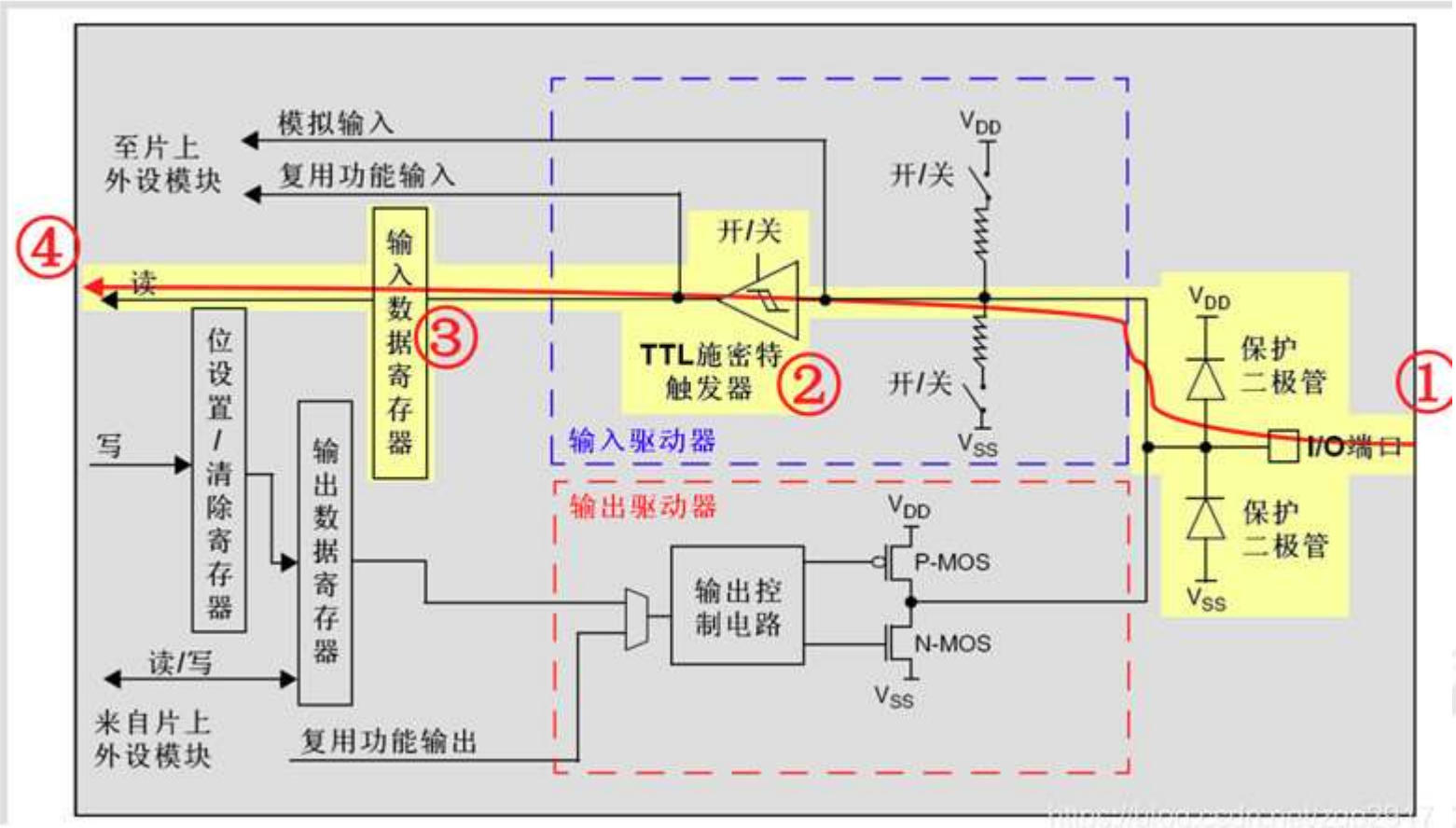
- 浮空输入
- 上拉输入
- 下拉输入
- 模拟输入



输出电路：

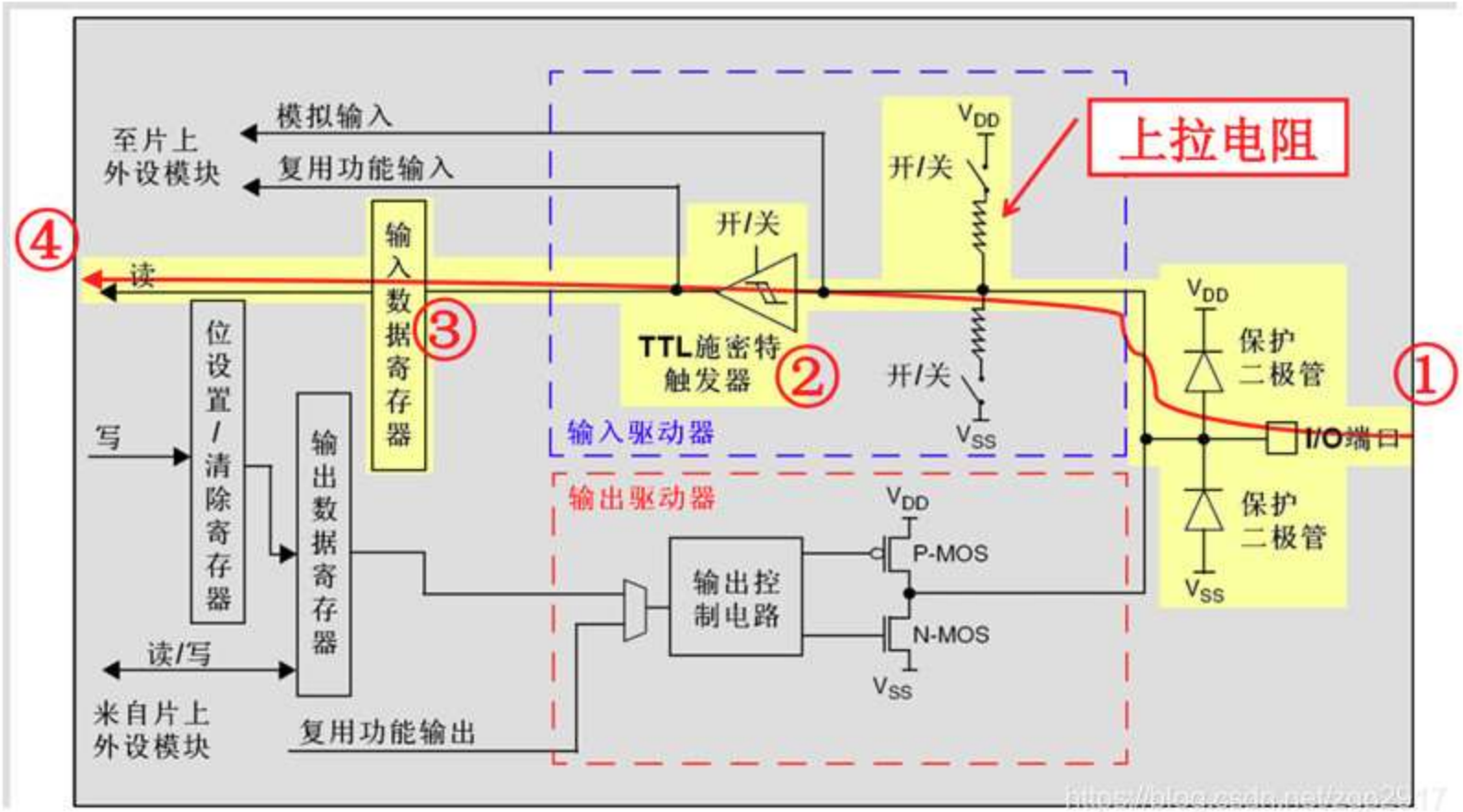
- 开漏输出
- 推挽输出

浮空输入模式



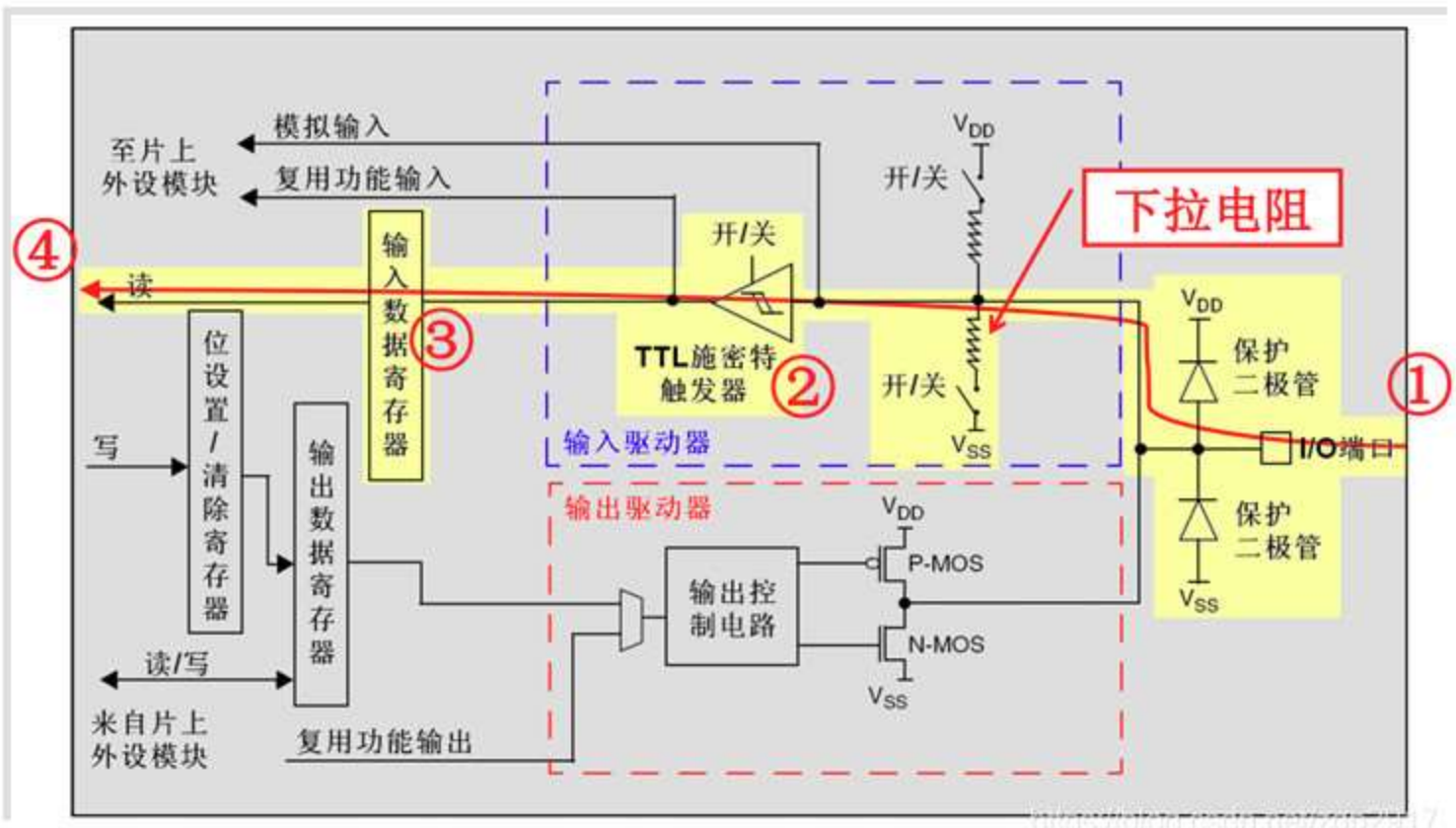
浮空输入模式下，I/O端口的电平信号直接进入输入数据寄存器。也就是说，I/O的电平状态是不确定的，完全由外部输入决定；如果在该引脚悬空（在无信号输入）的情况下，读取该端口的电平是不确定的。

上拉输入模式



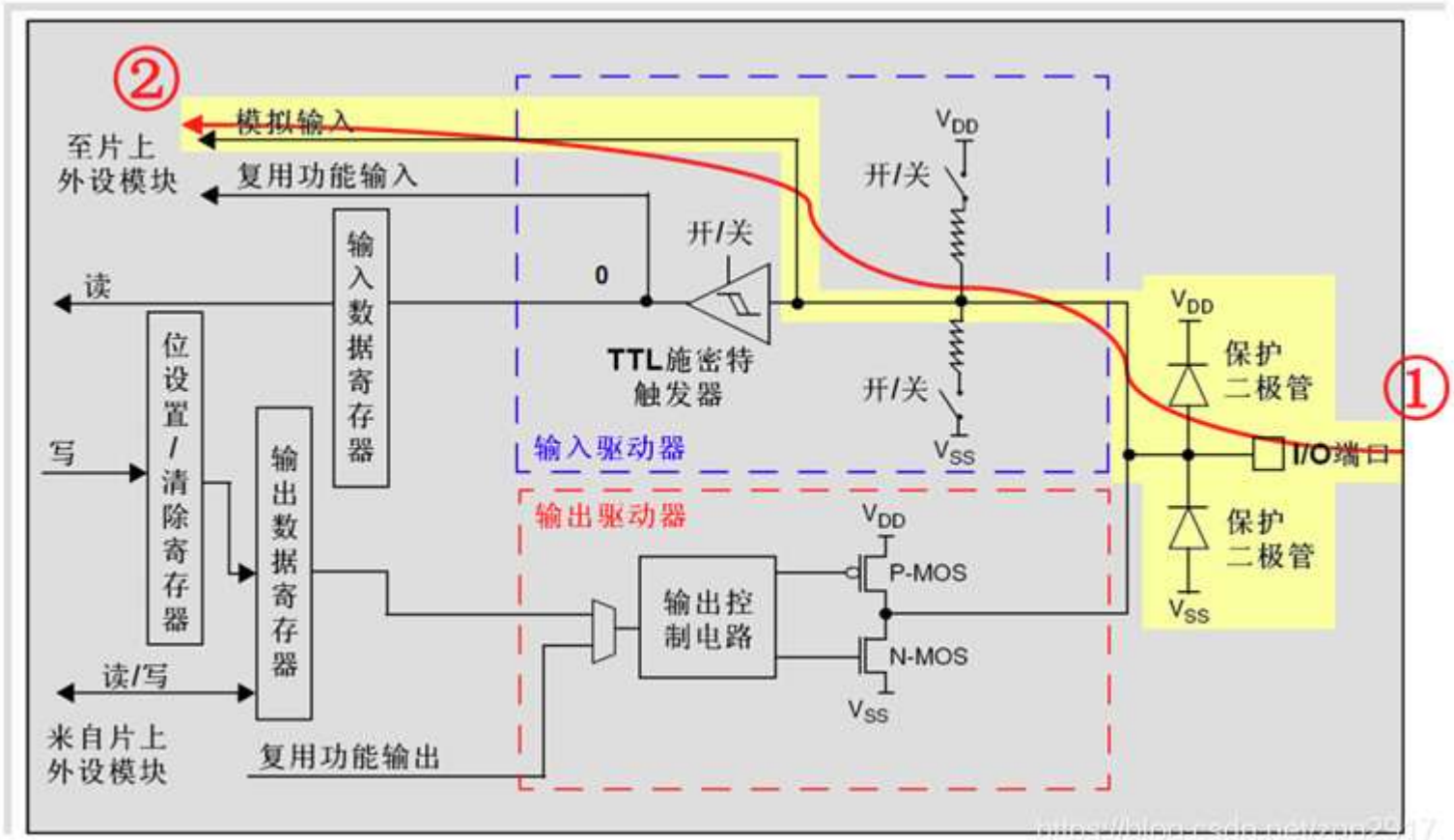
上拉输入模式下，I/O端口的电平信号直接进入输入数据寄存器。但是在I/O端口悬空（在无信号输入）的情况下，输入端电平可以保持在高电平；并且在I/O端口输入为低电平的时候，输入端的电平也还是低电平。

下拉输入模式



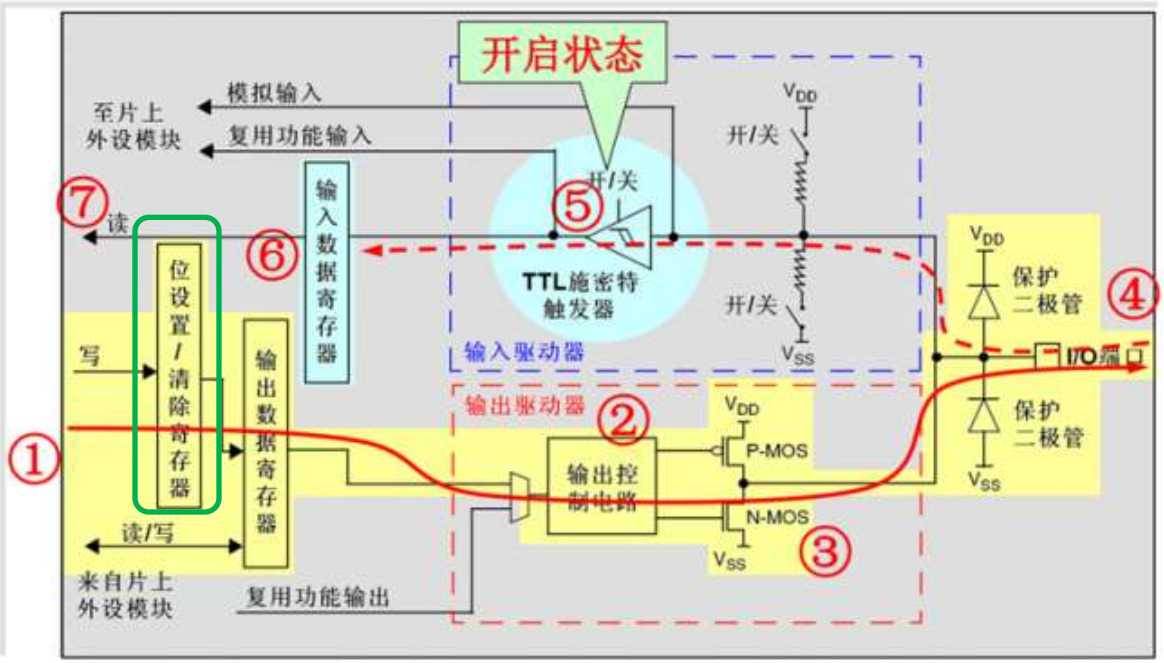
下拉输入模式下，I/O端口的电平信号直接进入输入数据寄存器。但是在I/O端口悬空（在无信号输入）的情况下，输入端电平可以保持在低电平；并且在I/O端口输入为高电平的时候，输入端的电平也还是高电平。

模拟输入模式



模拟输入模式下，I/O端口的模拟信号（电压信号，而非电平信号）直接模拟输入到片上外设模块，比如ADC模块等等。

推挽输出模式



推挽输出模式下，通过设置位设置/清除寄存器或者输出数据寄存器的值，途经P-MOS管和N-MOS管，最终输出到I/O端口。

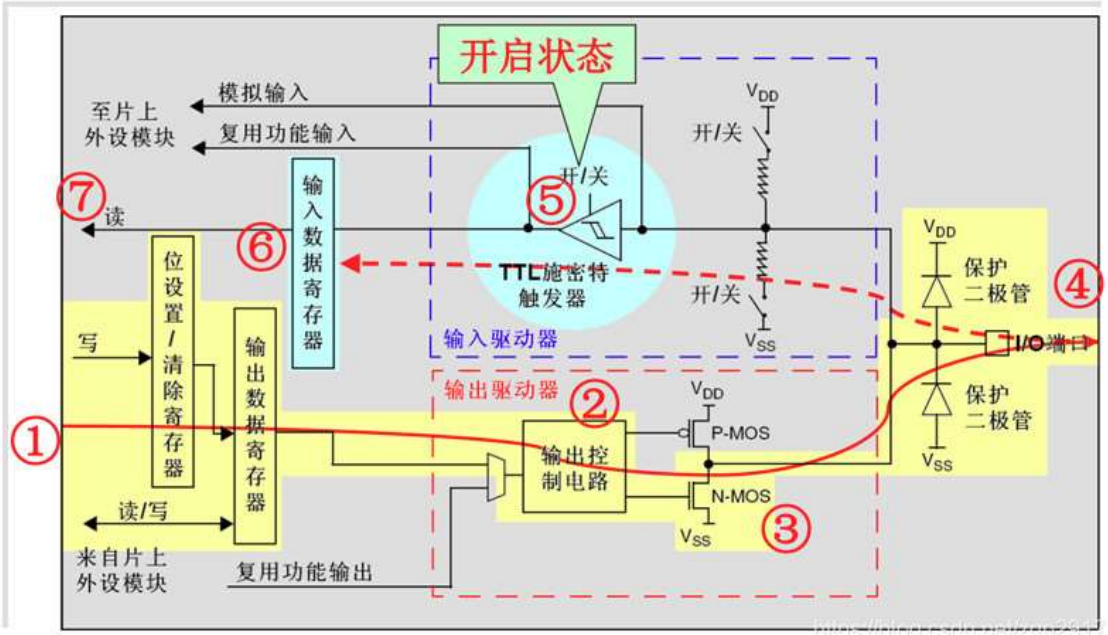
注意：

1、P-MOS管和N-MOS管，当设置输出值为高电平的时候，P-MOS管处于开启状态，N-MOS管处于关闭状态，此时I/O端口的电平就由P-MOS管决定：高电平；

2、当设置输出值为低电平的时候，P-MOS管处于关闭状态，N-MOS管处于开启状态，此时I/O端口的电平就由N-MOS管决定：低电平；

3、I/O端口的电平也可以通过输入电路进行读取；注意，此时I/O端口的电平一定是输出的电平。

开漏输出模式



开漏输出模式下，通过设置位设置/清除寄存器或者输出数据寄存器的值，途经N-MOS管，最终输出到I/O端口。

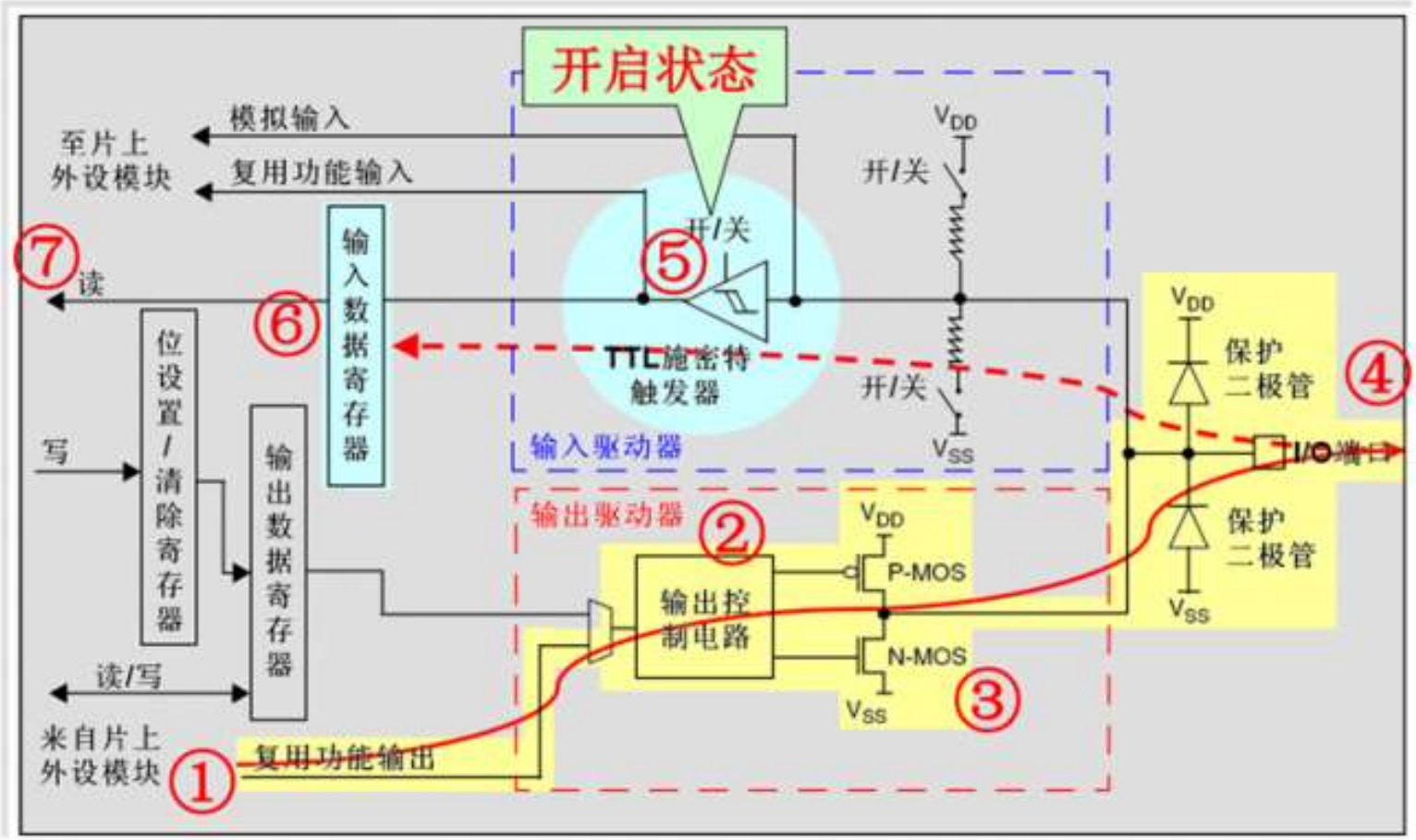
注意：

1、N-MOS管，当设置输出的值为高电平的时候，N-MOS管处于关闭状态，I/O端口的电平就不会由输出的高低电平决定，而是由I/O端口外部的上拉或者下拉决定；

2、当设置输出的值为低电平的时候，N-MOS管处于开启状态，此时I/O端口的电平就是低电平。同时，I/O端口的电平也可以通过输入电路进行读取；

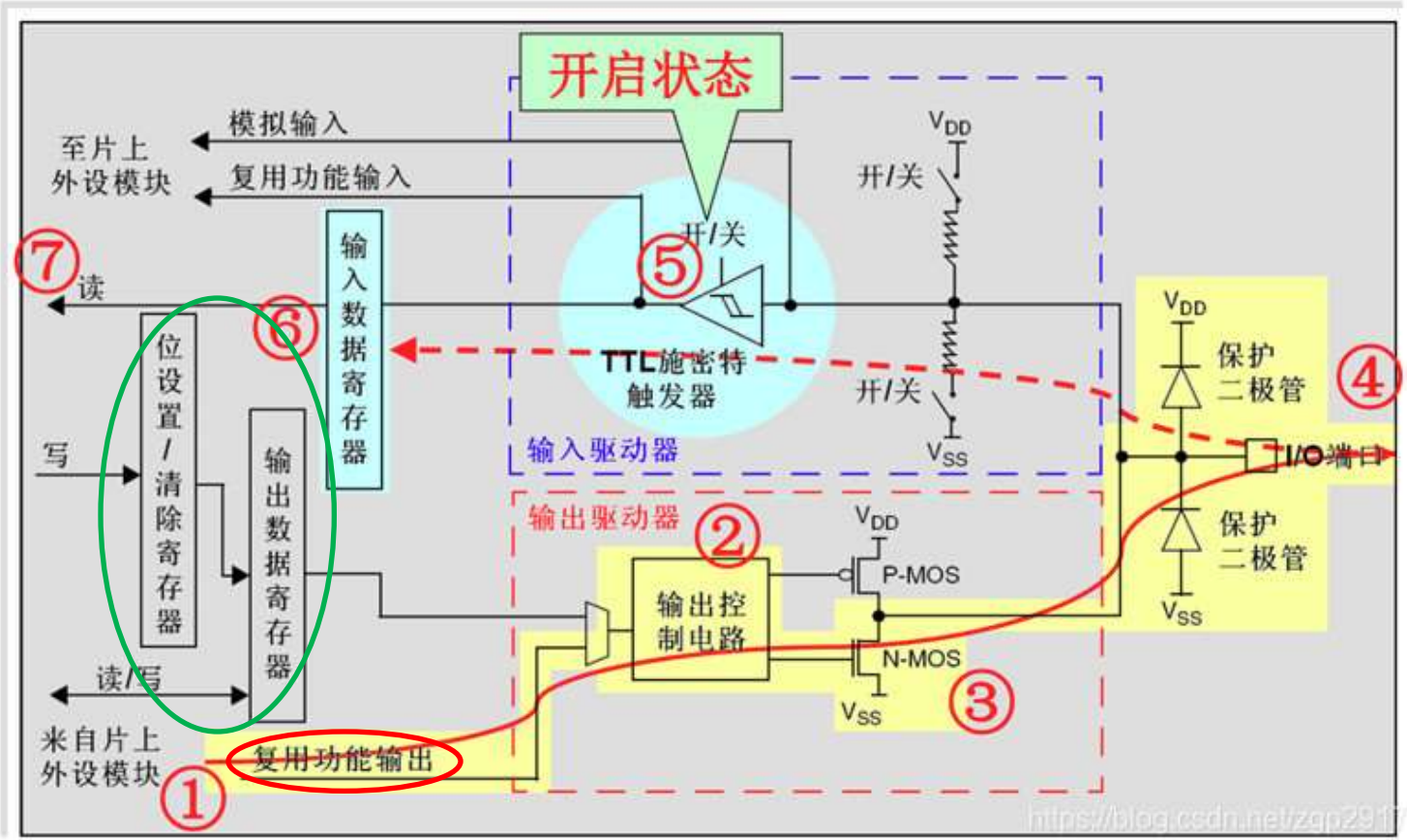
3、I/O端口的电平不一定是输出的电平。

推挽复用输出模式



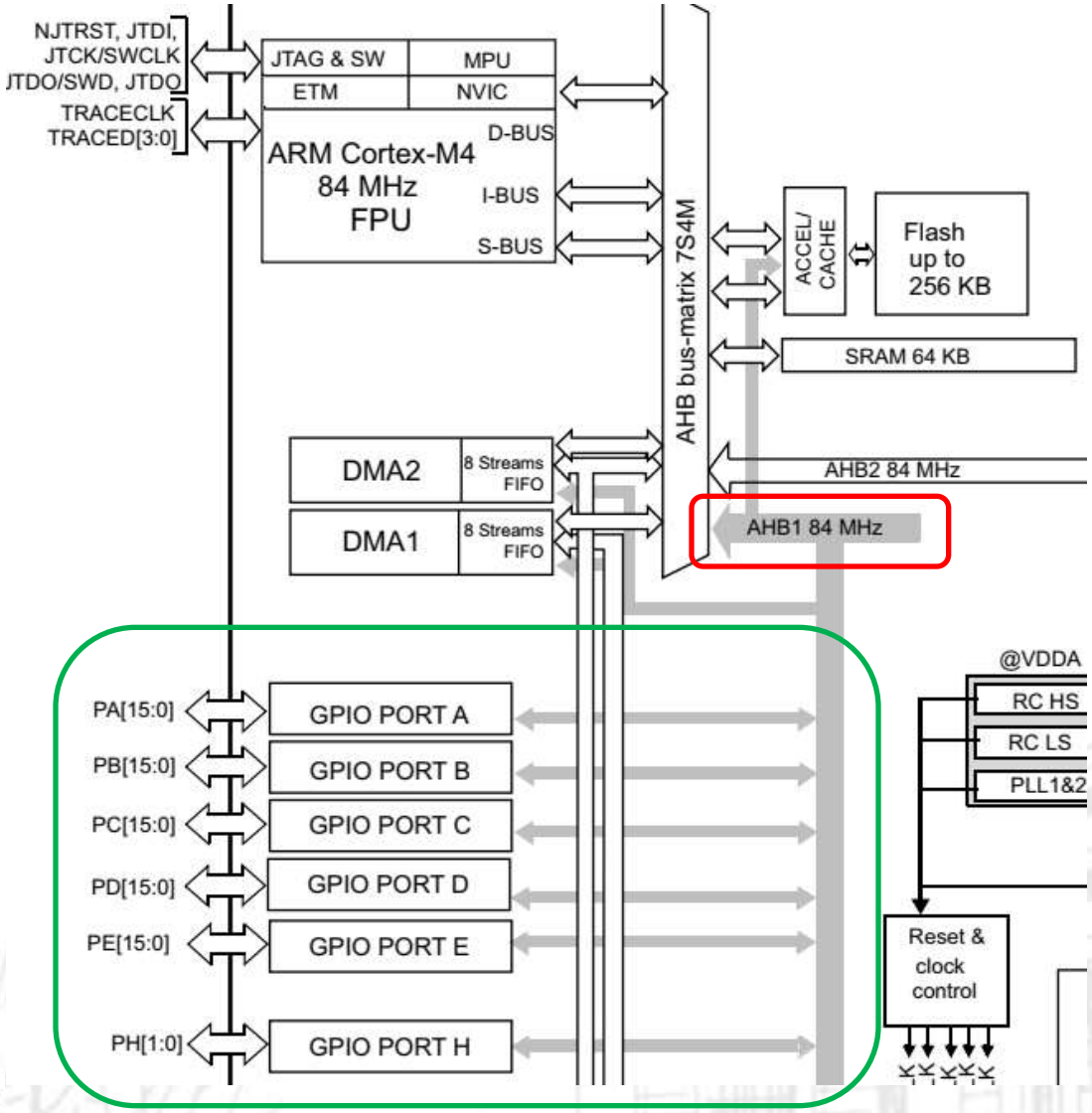
推挽复用输出模式，与推挽输出模式很是类似。只是输出的高低电平的来源，不是让CPU直接写输出数据寄存器，取而代之利用片上外设模块的复用功能输出来决定的。

开漏复用输出模式



开漏复用输出模式，与开漏输出模式类似。只是输出的高低电平的来源，不是让CPU直接写输出数据寄存器，而代之以片上外设模块的复用功能输出来决定的。

STM32F401的GPIO控制寄存器组挂载在AHB1总线上



➢外设寄存器

- 控制引脚的具体功能
- 外设寄存器可以通过系统总线访问，在系统总线上，有确定的地址。
- 端口的控制寄存器组的地址，由基地址和偏移地址组成
- 实际地址为：**基地址+偏移地址**

➢各个端口寄存器组在AHB1上的基地址

0x4002 3800 - 0x4002 3BFF	RCC	AHB1
0x4002 3000 - 0x4002 33FF	CRC	
0x4002 1C00 - 0x4002 1FFF	GPIOH	
0x4002 1000 - 0x4002 13FF	GPIOE	
0x4002 0C00 - 0x4002 0FFF	GPIOD	
0x4002 0800 - 0x4002 0BFF	GPIOC	
0x4002 0400 - 0x4002 07FF	GPIOB	
0x4002 0000 - 0x4002 03FF	GPIOA	

- 各个端口配置寄存器组的基地址不同
- 负责控制具体功能的偏移地址是相同的

Offset	Register	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
0x00	GPIOA_ MODER	MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]		MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
	Reset value	0	0	0	0	1	1	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
0x00	GPIOB_ MODER	MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]		MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
0x00	GPIOx_MODER (where x = C..E and H)	MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]		MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
	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
0x04	GPIOx_ OTYPER (where x = A..E and H)	Reserved																OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	GPIOx_ OSPEEDR (where x = C..E and H)	OSPEEDR15[1:0]		OSPEEDR14[1:0]		OSPEEDR13[1:0]		OSPEEDR12[1:0]		OSPEEDR11[1:0]		OSPEEDR10[1:0]		OSPEEDR9[1:0]		OSPEEDR8[1:0]		OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1[1:0]		OSPEEDR0[1:0]	
	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
0x08	GPIOA_ OSPEEDER	OSPEEDR15[1:0]		OSPEEDR14[1:0]		OSPEEDR13[1:0]		OSPEEDR12[1:0]		OSPEEDR11[1:0]		OSPEEDR10[1:0]		OSPEEDR9[1:0]		OSPEEDR8[1:0]		OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1[1:0]		OSPEEDR0[1:0]	
	Reset value	0	0	0	0	1	1	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
0x08	GPIOB_ OSPEEDR	SPEEDR15[1:0]		SPEEDR14[1:0]		SPEEDR13[1:0]		SPEEDR12[1:0]		SPEEDR11[1:0]		SPEEDR10[1:0]		SPEEDR9[1:0]		SPEEDR8[1:0]		SPEEDR7[1:0]		SPEEDR6[1:0]		SPEEDR5[1:0]		SPEEDR4[1:0]		SPEEDR3[1:0]		SPEEDR2[1:0]		SPEEDR1[1:0]		SPEEDR0[1:0]	

➢外设寄存器的访问

- 在stm32f401xc.h文件中，使用C语言实现了一种简便的寄存器访问方式。
- stm32f401xc.h文件中定义了一个结构类型，名称为PIO_TypeDef。
- 依据偏移地址，地址由小到大定义各个寄存器（功能）。寄存器数据类型都是32位无符号整形。

```
typedef struct
{
  __IO uint32_t MODER;    /*!< GPIO port mode register,                Address offset: 0x00      */
  __IO uint32_t OTYPER;   /*!< GPIO port output type register,         Address offset: 0x04      */
  __IO uint32_t OSPEEDR;  /*!< GPIO port output speed register,        Address offset: 0x08      */
  __IO uint32_t PUPDR;    /*!< GPIO port pull-up/pull-down register,   Address offset: 0x0C      */
  __IO uint32_t IDR;      /*!< GPIO port input data register,          Address offset: 0x10      */
  __IO uint32_t ODR;      /*!< GPIO port output data register,         Address offset: 0x14      */
  __IO uint32_t BSRR;     /*!< GPIO port bit set/reset register,       Address offset: 0x18      */
  __IO uint32_t LCKR;     /*!< GPIO port configuration lock register,  Address offset: 0x1C      */
  __IO uint32_t AFR[2];   /*!< GPIO alternate function registers,      Address offset: 0x20-0x24 */
} GPIO_TypeDef;
```

➢外设寄存器的访问

– stm32f401xc.h文件中定义了GPIOA模块的基地址

```
#define GPIOA_BASE            (AHB1PERIPH_BASE + 0x0000UL)
#define AHB1PERIPH_BASE       (PERIPH_BASE + 0x00020000UL)
#define PERIPH_BASE           0x40000000UL /*!< Peripheral base address in the alias region
*/
```

– stm32f401xc.h文件中定义了一个宏GPIOA，将GPIOA模块的基地址转换为GPIO_TypeDef类型的指针。

```
#define GPIOA                 ((GPIO_TypeDef *) GPIOA_BASE)
```

– 宏GPIOA就是一个GPIO_TypeDef类型的指针，其指向GPIOA模块的基地址。通过该结构体指针，可以方便的访问GPIOA模块的各个寄存器

- GPIOA->MODER
- GPIOA->IDR
- GPIOA->ODR

➢模式寄存器 GPIOx_MODER

- 共有32位，可以控制16个引脚的工作模式，包括输入、输出、外设复用、模拟信号四种模式。
- 每两位控制一个引脚，MODER0[1:0]两位控制PA0引脚，MODER1[1:0]两位控制PA1引脚。
  - 00代表为数字输入模式
  - 01代表为数字输出模式
  - 10代表外设复用模式
  - 11代表模拟信号模式

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

➢输出模式寄存器GPIOx_OTYPER

- 32位寄存器，控制了16个引脚的输出模式，每一位控制一个引脚，因此只有低16位有效，高16位无效。
- OT0控制PA0引脚，OT1控制PA1引脚。
- 如果OT0为0，代表推挽输出，如果OT0为1，代表开漏输出。
- 只有在GPIOx_MODER寄存器里被设置为输出的引脚，才受GPIOx_OTYPER寄存器的控制。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

GPIOA->OTYPER=0x01;将PA0引脚设置为开漏输出，其他引脚为推挽输出

➢ 上拉电阻与下拉电阻配置寄存器：GPIOx_PUPDR

– 32位，可以控制16个引脚。

– 每两位控制一个引脚。如PUPDR0控制PA0， PUPDR1控制PA2 。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- 00代表不使用上拉或者下拉
- 01代表打开上拉电阻
- 10代表打开下拉电阻
- 11代表无效

➢数据输入寄存器GPIOx_IDR

- 32位只读寄存器，低16位有效。
- 每一位代表一个引脚当前的电平状态
- 0代表低电平，1代表高电平。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

➢数据输出寄存器GPIOx_ODR

- 32位寄存器，低16位有效。
- 改变该寄存器的可以改变引脚上的电平状态
- 0代表低电平，1代表高电平。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

➢ 数据输出位操作寄存器GPIOx_BSRR（注意区别于GPIOx_ODR）

- 由于操作GPIOx_ODR寄存器会改变某些位可能会不经意的影响其他位的状态，因此提供了数据输出位操作寄存器GPIOx_BSRR。
- 32位，操作16个引脚。该寄存器的位分为两组，BRn（高16位）和BSn （低16位），n=0-15。
- 给BRn的一个位或几个位赋1，可以使对应的引脚，如PAn输出低电平，GPIOx_ODR寄存器的位也会相应的改变。
- 给BSn的一个位或几个位赋1，可以使对应的引脚，如PAn输出高电平，GPIOx_ODR寄存器的位也会相应的改变。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

- 使PA5输出高电平，而不影响其他的位：GPIOA->BSRR=(1<<5);
- 使PA5输出低电平，而不影响其他的位：GPIOA->BSRR=(1<<5)<<16;

GPIO模块的使能

- 所有的外设在使用前，都要使能它的时钟，不然无法对它的寄存器进行读写。
- 系统控制模块中的RCC_AHB1ENR寄存器控制了AHB1总线上各个外设的时钟，只有打开GPIO模块的时钟，GPIO模块才能工作

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									DMA2EN	DMA1EN	Reserved				
									rw	rw					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Reserved				GPIOH EN	Reserved		GPIOEEN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
			rw					rw			rw	rw	rw	rw	rw

- 在stm32f4xx_hal_rcc.h文件中，提供了一个宏定义__HAL_RCC_GPIOA_CLK_ENABLE()来完成这个任务。

```
#define __HAL_RCC_GPIOA_CLK_ENABLE()   do { \
                                        __IO uint32_t tmpreg = 0x00U; \
                                        SET_BIT(RCC->AHB1ENR, RCC_AHB1ENR_GPIOAEN);\
                                        /* Delay after an RCC peripheral clock enabling */ \
                                        tmpreg = READ_BIT(RCC->AHB1ENR, RCC_AHB1ENR_GPIOAEN);\
                                        UNUSED(tmpreg); \
                                          } while(0U)
```

➢使用GPIO模块的步骤：

- 1. RCC_AHB1ENR寄存器使能时钟
- 2. GPIOx_MODER 寄存器设置输入还是输出
- 3.如果设置为输出，使用GPIOx_OTYPER设置推挽输出还是开漏输出
- 4.如果设置为输入，则使用GPIOx_PUPDR设置上拉电阻和下拉电阻。
- 5.使用GPIOx_IDR、GPIOx_ODR、GPIOx_BSRR（引脚位操作）寄存器读写数据。

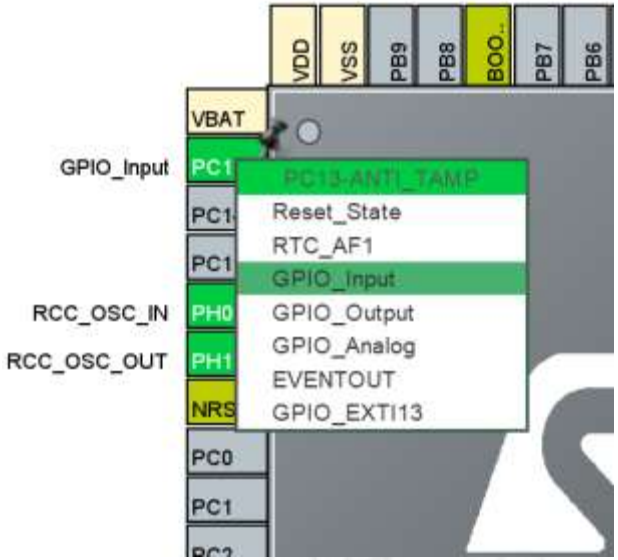
➢练习

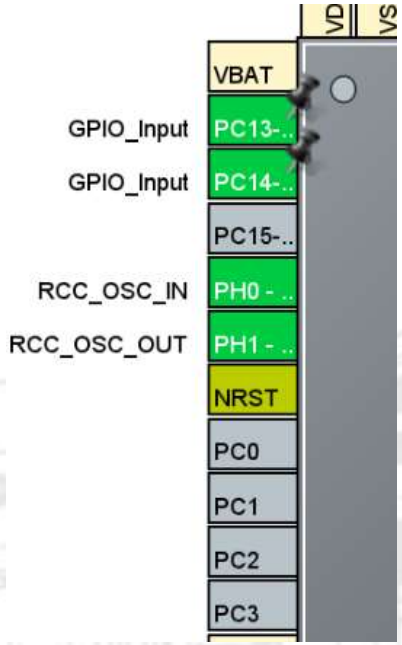
- 将KEY1和KEY2分别连接到PC13和PC14号引脚
- 将LED0和LED1分别连接到PC0和PC1号引脚。
- 编写程序，对GPIO进行初始化。
- 在主循环中，读取PC13和PC14引脚的状态，
  - 如果PC13引脚为高电平，则key1变量为1，否则为0。
  - 如果PC14引脚为高电平，则key2变量为1，否则为0。
- 以一定的频率（2Hz），使LED0和LED1闪烁。

- STM32CubeIDE图形化初始化GPIO，并用HAL库操作GPIO：
  - 使用直接操作寄存器的方式操作GPIO虽然简单直接，但是可读性不强。编程需要经常对照手册，编程效率不高。
  - STM32CubeIDE提供了图形化界面，可以方便的初始化GPIO。同时提供了HAL库函数，以较高的可读性编写代码。
  - HAL库是ST公司目前主力推的开发方式，全称就是Hardware Abstraction Layer（硬件抽象层），可以大大增加硬件的可读性，并且将功能代码与底层驱动分离

➢STM32CubeIDE图形化初始化GPIO：

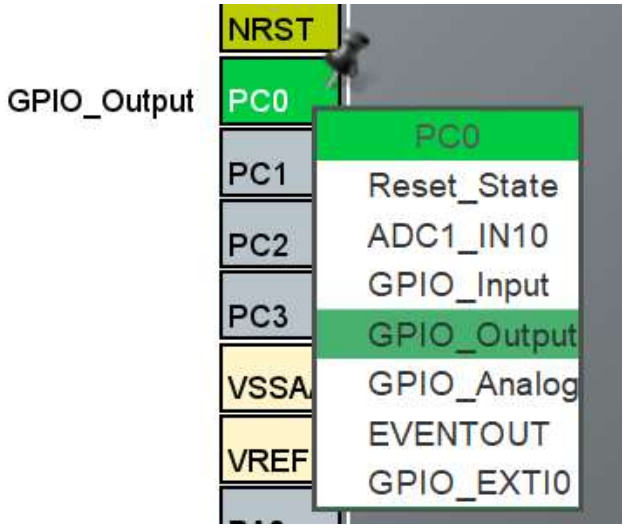
- 从HelloWorld项目复制一个新项目，命名为GPIO_HAL，并完成相应的修改。
- 打开配置文件GPIO_HAL.ioc
- 点击右侧芯片引脚配置图中的PC13引脚，在弹出的选项中选择GPIO_Input，可以将PC13引脚设置为数字输入。
- 同理，将PC14号引脚也设置为输入。

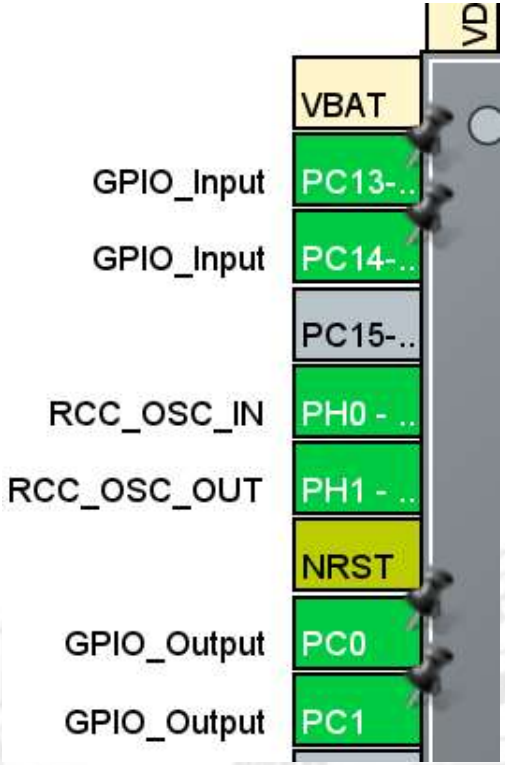




➢STM32CubeIDE图形化初始化GPIO：

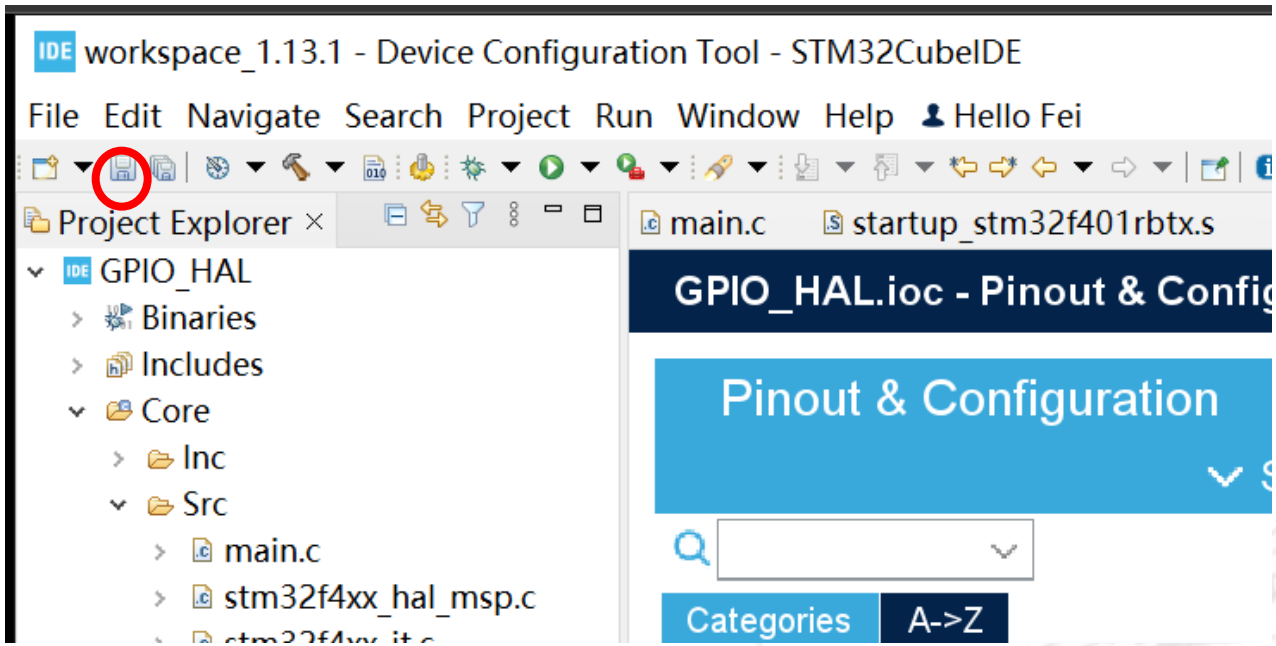
- 点击PC0引脚，在弹出的选项中选择GPIO_Output，可以将PC0设置为输出。
- 同理，将PC1引脚也设置为输出。





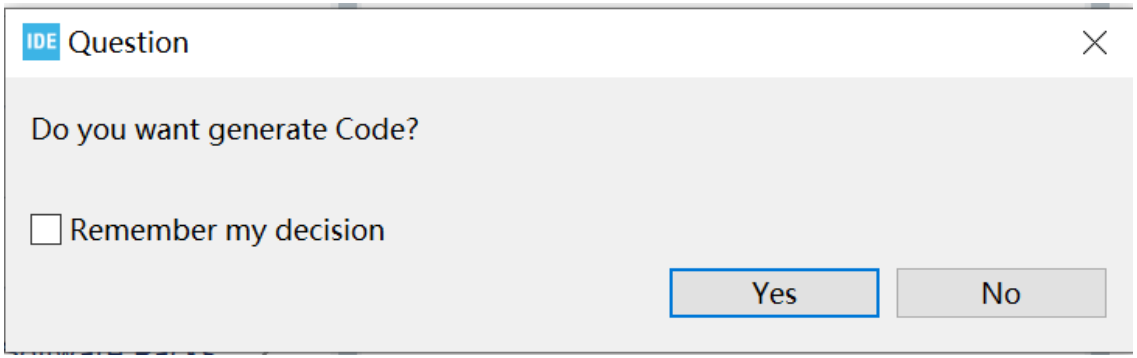
➢STM32CubeIDE图形化初始化GPIO：

– 完成后点击保存按钮 保存配置

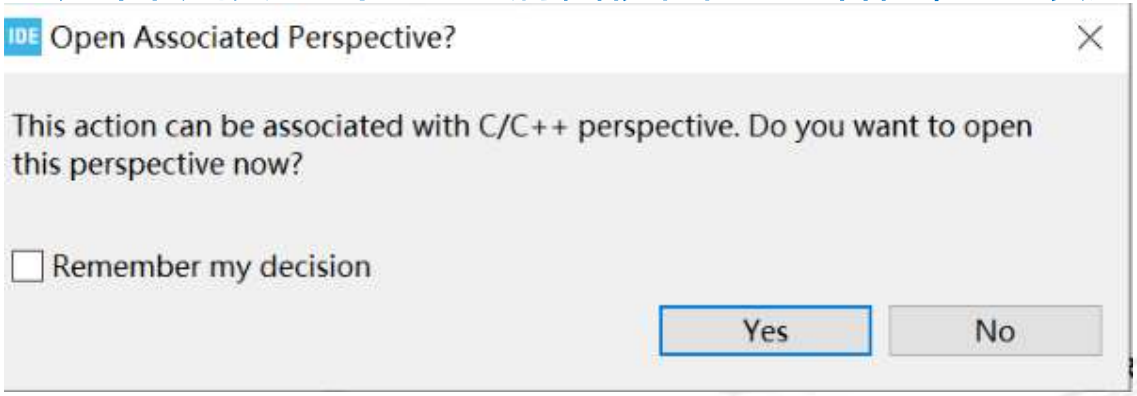


➢ STM32CubeIDE图形化初始化GPIO：

– 会弹出是否生成代码的对话框，点Yes



– 会弹出是否切换到C/C++编辑视图的对话框，继续点Yes



– 软件会重新生成配置代码。

– 在规定的位置/* USER CODE BEGIN XXX*/与/* USER CODE END XXX*/之间书写的代码，会被删除。

➢STM32CubeIDE图形化初始化GPIO：

– MX_GPIO_Init函数内增加了初始化PC13、PC14、PC0、PC1的代码

```
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
/* USER CODE BEGIN MX_GPIO_Init_1 */
/* USER CODE END MX_GPIO_Init_1 */

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1, GPIO_PIN_RESET);

  /*Configure GPIO pins : PC13 PC14 */
  GPIO_InitStruct.Pin = GPIO_PIN_13|GPIO_PIN_14;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  /*Configure GPIO pins : PC0 PC1 */
  GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}
```

调用HAL_GPIO_Init函数，初始化PC13、PC14、PC0、PC1。

➢ STM32CubeIDE图形化初始化GPIO：

- HAL_GPIO_Init函数是HAL库提供给用户的函数
- 以**HAL_开头**
- 第二个字段GPIO为该函数操作的模块名称
- 第三个字段描述的是该函数的功能，如Init为初始化的功能。

Function name

void HAL_GPIO_Init (GPIO_TypeDef * GPIOx, GPIO_InitTypeDef * GPIO_Init)

```
typedef struct
{
  uint32_t Pin;       /*!< Specifies the GPIO pins to be configured.
                           This parameter can be any value of @ref GPIO_pins_define */
  uint32_t Mode;      /*!< Specifies the operating mode for the selected pins.
                           This parameter can be a value of @ref GPIO_mode_define */
  uint32_t Pull;      /*!< Specifies the Pull-up or Pull-Down activation for the selected pins.
                           This parameter can be a value of @ref GPIO_pull_define */
  uint32_t Speed;     /*!< Specifies the speed for the selected pins.
                           This parameter can be a value of @ref GPIO_speed_define */
  uint32_t Alternate;  /*!< Peripheral to be connected to the selected pins.
                            This parameter can be a value of @ref GPIO_Alternate_function_selection */
}GPIO_InitTypeDef;
```

- 第二个参数为一个GPIO_InitTypeDef类型的结构体指针，里面存储了GPIO的配置信息。
- GPIO_InitStruct在MX_GPIO_Init函数的开始位置定义，函数退出后销毁

➢STM32CubeIDE图形化初始化GPIO：

- Pin指定要初始化的引脚号
- 端口号已经由HAL_GPIO_Init的第一个参数，GPIOx指定

```
#define GPIO_PIN_0                 ((uint16_t)0x0001)  /* Pin 0 selected    */
#define GPIO_PIN_1                 ((uint16_t)0x0002)  /* Pin 1 selected    */
#define GPIO_PIN_2                 ((uint16_t)0x0004)  /* Pin 2 selected    */
#define GPIO_PIN_3                 ((uint16_t)0x0008)  /* Pin 3 selected    */
#define GPIO_PIN_4                 ((uint16_t)0x0010)  /* Pin 4 selected    */
#define GPIO_PIN_5                 ((uint16_t)0x0020)  /* Pin 5 selected    */
#define GPIO_PIN_6                 ((uint16_t)0x0040)  /* Pin 6 selected    */
#define GPIO_PIN_7                 ((uint16_t)0x0080)  /* Pin 7 selected    */
#define GPIO_PIN_8                 ((uint16_t)0x0100)  /* Pin 8 selected    */
#define GPIO_PIN_9                 ((uint16_t)0x0200)  /* Pin 9 selected    */
#define GPIO_PIN_10                ((uint16_t)0x0400)  /* Pin 10 selected   */
#define GPIO_PIN_11                ((uint16_t)0x0800)  /* Pin 11 selected   */
#define GPIO_PIN_12                ((uint16_t)0x1000)  /* Pin 12 selected   */
#define GPIO_PIN_13                ((uint16_t)0x2000)  /* Pin 13 selected   */
#define GPIO_PIN_14                ((uint16_t)0x4000)  /* Pin 14 selected   */
#define GPIO_PIN_15                ((uint16_t)0x8000)  /* Pin 15 selected   */
#define GPIO_PIN_All               ((uint16_t)0xFFFF)  /* All pins selected */
```

- **STM32CubeIDE图形化初始化GPIO：**
  - Pin指定要初始化的引脚号
  - Pin可以是GPIO_PIN_x，也可以是他们的组合

```
GPIO_InitStruct.Pin = GPIO_PIN_13|GPIO_PIN_14;
```

➢STM32CubeIDE图形化初始化GPIO：

– Mode用于选择以上指定的引脚的输入输出方式

```
#define  GPIO_MODE_INPUT        MODE_INPUT                  /*!< Input Floating Mode                   */
#define  GPIO_MODE_OUTPUT_PP    (MODE_OUTPUT | OUTPUT_PP) /*!< Output Push Pull Mode                 */
#define  GPIO_MODE_OUTPUT_OD    (MODE_OUTPUT | OUTPUT_OD) /*!< Output Open Drain Mode                */
#define  GPIO_MODE_AF_PP        (MODE_AF | OUTPUT_PP)      /*!< Alternate Function Push Pull Mode     */
#define  GPIO_MODE_AF_OD        (MODE_AF | OUTPUT_OD)      /*!< Alternate Function Open Drain Mode    */
#define  GPIO_MODE_ANALOG       MODE_ANALOG
```

– HAL_GPIO_Init函数根据这个参数设置引脚的输入、输出、推挽、开漏等寄存器

```
typedef struct
{
  uint32_t Pin;       /*!< Specifies the GPIO pins to be configured.
                           This parameter can be any value of @ref GPIO_pins_define */
  uint32_t Mode;      /*!< Specifies the operating mode for the selected pins.
                           This parameter can be a value of @ref GPIO_mode_define */
  uint32_t Pull;      /*!< Specifies the Pull-up or Pull-Down activation for the selected pins.
                           This parameter can be a value of @ref GPIO_pull_define */
  uint32_t Speed;     /*!< Specifies the speed for the selected pins.
                           This parameter can be a value of @ref GPIO_speed_define */
  uint32_t Alternate;  /*!< Peripheral to be connected to the selected pins.
                            This parameter can be a value of @ref GPIO_Alternate_function_selection */
}GPIO_InitTypeDef;
```

➢STM32CubeIDE图形化初始化GPIO：

– Pull用于选择上拉电阻或下拉电阻

```
#define  GPIO_NOPULL        0x00000000U   /*!< No Pull-up or Pull-down activation  */
#define  GPIO_PULLUP        0x00000001U   /*!< Pull-up activation                  */
#define  GPIO_PULLDOWN      0x00000002U   /*!< Pull-down activation                */
```

– 第四个参数Speed用于设置引脚的翻转速度，此处使用默认设置。

– 第五个参数Alternate用于设置引脚的复用功能，此处不复用。

```
typedef struct
{
  uint32_t Pin;        /*!< Specifies the GPIO pins to be configured.
                            This parameter can be any value of @ref GPIO_pins_define */
  uint32_t Mode;       /*!< Specifies the operating mode for the selected pins.
                            This parameter can be a value of @ref GPIO_mode_define */
  uint32_t Pull;       /*!< Specifies the Pull-up or Pull-Down activation for the selected pins.
                            This parameter can be a value of @ref GPIO_pull_define */
  uint32_t Speed;      /*!< Specifies the speed for the selected pins.
                            This parameter can be a value of @ref GPIO_speed_define */
  uint32_t Alternate;  /*!< Peripheral to be connected to the selected pins.
                             This parameter can be a value of @ref GPIO_Alternate_function_selection */
}GPIO_InitTypeDef;
```

➢STM32CubeIDE图形化初始化GPIO：

– HAL_GPIO_Init函数

```
if(((GPIO_Init->Mode & GPIO_MODE) == MODE_OUTPUT) || \
          (GPIO_Init->Mode & GPIO_MODE) == MODE_AF)
      {
        /* Check the Speed parameter */
        assert_param(IS_GPIO_SPEED(GPIO_Init->Speed));

        /* Configure the IO Output Type */
        temp = GPIOx->OTYPER;
        temp &= ~(GPIO_OTYPER_OT_0 << position) ;
        temp |= (((GPIO_Init->Mode & OUTPUT_TYPE) >> OUTPUT_TYPE_Pos) << position);
        GPIOx->OTYPER = temp;
       }
      /* Configure IO Direction mode (Input, Output, Alternate or Analog) */
      temp = GPIOx->MODER;
      temp &= ~(GPIO_MODER_MODER0 << (position * 2U));
      temp |= ((GPIO_Init->Mode & GPIO_MODE) << (position * 2U));
      GPIOx->MODER = temp;
```

- 根据Mode参数这是MODER寄存器，确定引脚是输入还是输出。
- 如果是输出，再根据Mode参数来确定推挽还是开漏。其操作方式是先读后写，与我们直接操作寄存器的方式类似。

➢使用HAL库函数读/写GPIO：

– HAL_GPIO_ReadPin函数

```
GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
  GPIO_PinState bitstatus;

  /* Check the parameters */
  assert_param(IS_GPIO_PIN(GPIO_Pin));

  if((GPIOx->IDR & GPIO_Pin) != (uint32_t)GPIO_PIN_RESET)
  {
    bitstatus = GPIO_PIN_SET;
  }
  else
  {
    bitstatus = GPIO_PIN_RESET;
  }
  return bitstatus;
}
```

- 第一个参数为GPIO端口的基地址
- 第二个参数为引脚号
- 引脚如果为高电平，这返回GPIO_PIN_SET
- 引脚如果为低电平，则返回GPIO_PIN_RESET。
- 该函数运行一次只能读取一个引脚的状态。

➢使用HAL库函数读写GPIO：

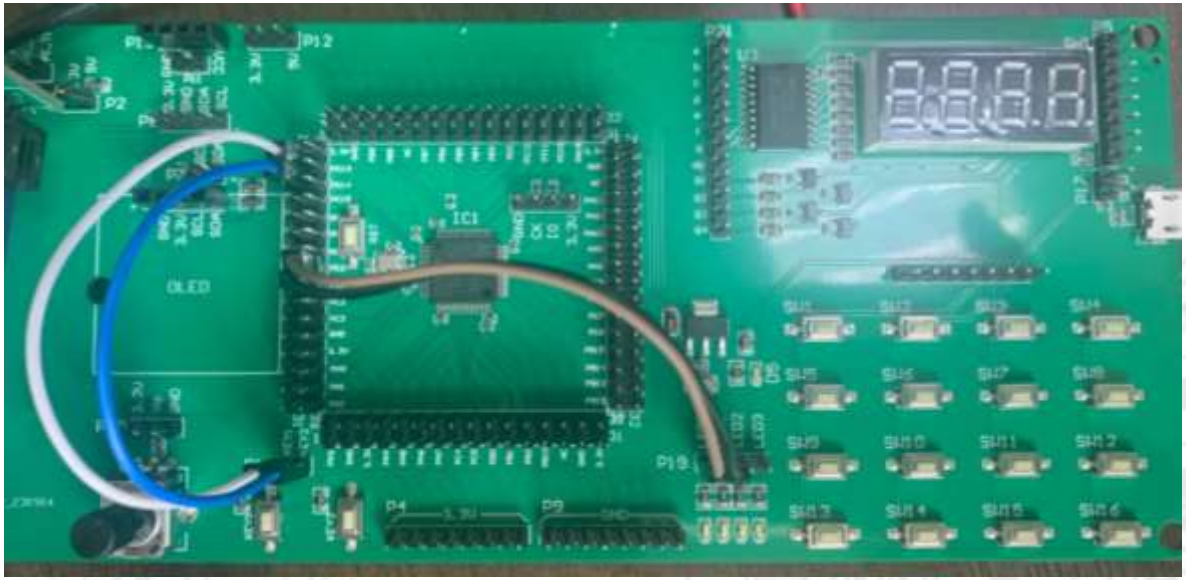
– HAL_GPIO_WritePin函数

```
void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, GPIO_PinState
PinState)
{
  /* Check the parameters */
  assert_param(IS_GPIO_PIN(GPIO_Pin));
  assert_param(IS_GPIO_PIN_ACTION(PinState));

  if(PinState != GPIO_PIN_RESET)
  {
    GPIOx->BSRR = GPIO_Pin;
  }
  else
  {
    GPIOx->BSRR = (uint32_t)GPIO_Pin << 16U;
  }
}
```

- 第一个参数为GPIO端口的基地址
- 第二个参数为引脚号，可以是GPIO_PIN_x（x=0-16）中的一个，也可以是他们的组合。
- 第三个参数是要设置的引脚的状态，GPIO_PIN_RESET可以把这些引脚设置为低电平，GPIO_PIN_SET可以把这些引脚设置为高电平。
- HAL_GPIO_WritePin函数可以同时设置多个引脚的状态。

➢练习

- 将KEY1和KEY2分别连接到PC13和PC14号引脚
- 将LED0和LED1分别连接到PC0和PC1号引脚
- 使用器件配置界面初始化端口，并编写程序
- 在主循环中，使用HAL库函数读取PC13和PC14引脚的状态
- 如果PC13引脚为高电平，则key1变量为1，否则为0
- 如果PC14为高电平，则key2变量为1，否则为0
- 以一定的频率（2Hz），使LED0和LED1闪烁。



➢使用HAL库函数读/写GPIO：

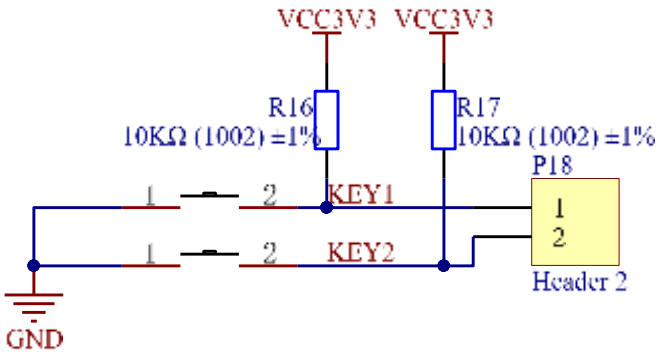
– 在helloworld项目的基础上，修改主循环代码：

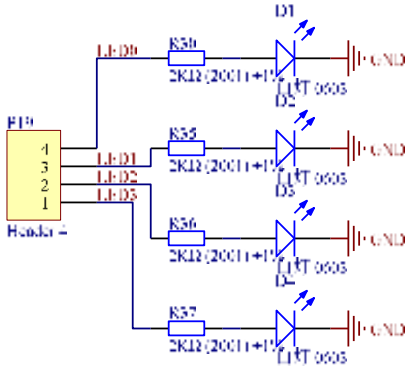
```
/* Infinite loop */
 /* USER CODE BEGIN WHILE */
 while (1)
 {
         g++;
         l++;
         HAL_Delay(500);

         key1=HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13);
         key2=HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_14);
         if(g&1){
                 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, GPIO_PIN_SET);
                 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_1, GPIO_PIN_RESET);
         }else{
                 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, GPIO_PIN_RESET);
                 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_1, GPIO_PIN_SET);
         }
   /* USER CODE END WHILE */
```

- 小节
  - 通用输入输出端口的基本概念
  - STM32F401RB的GPIO
  - 控制寄存器的访问
  - 控制寄存器的功能
  - 数据寄存器的功能与操作
  - GPIO模块的使能与使用
  - STM32CubeIDE初始化GPIO，并用HAL库操作GPIO

➢练习

– 简单按键的电路和LED的电路如图所示





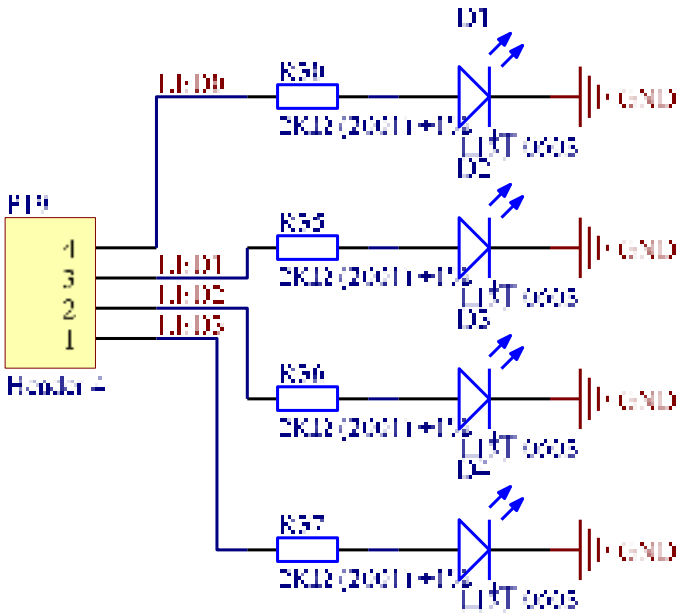
– 将KEY1连接到PC13，KEY2连接到PC14。需要用PC13和PC14感知按键的状态。利用图形界面设置PC13和PC14引脚，应该设置为输入还是输出？

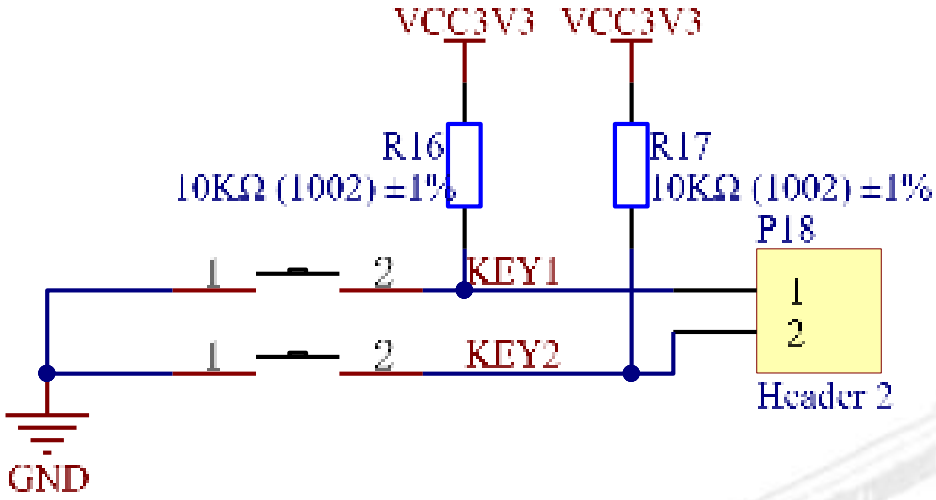
– 将LED0和LED1连接到PC0和PC1用于控制灯的亮灭，利用图形界面设置PC13和PC14引脚，应该设置为输入还是输出？

– 始化代码MX_GPIO_Init代码执行完毕后，GPIOC模块的MODER寄存器的值是什么？没有按键按下时，GPIOC->IDR寄存器的值是什么？KEY1按键按下时，GPIOC->IDR寄存器的值是什么？

- 常用的IO驱动电路
- 简单按键开关
- 矩阵键盘
- 继电器
- LED灯
- 数码管
- 传感器输入

➢开发板上的按键和LED灯





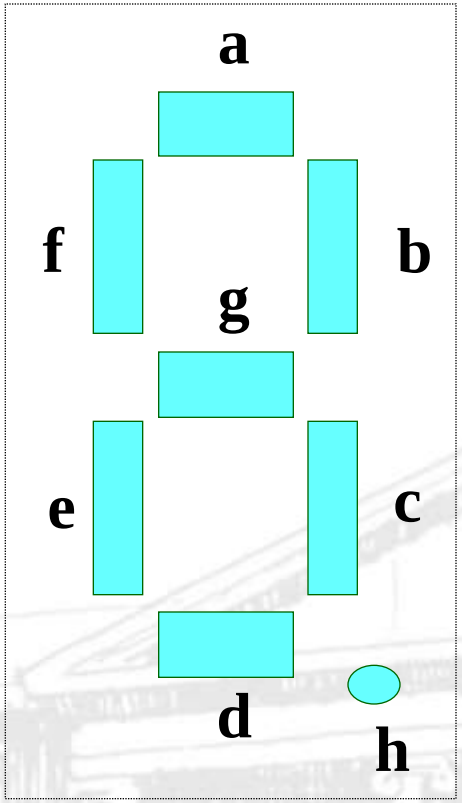
➢数码管

– 发光二极管LED是最简单的显示设备，由7段LED就可以组成的LED数码管，一般还有一个LED代表小数点

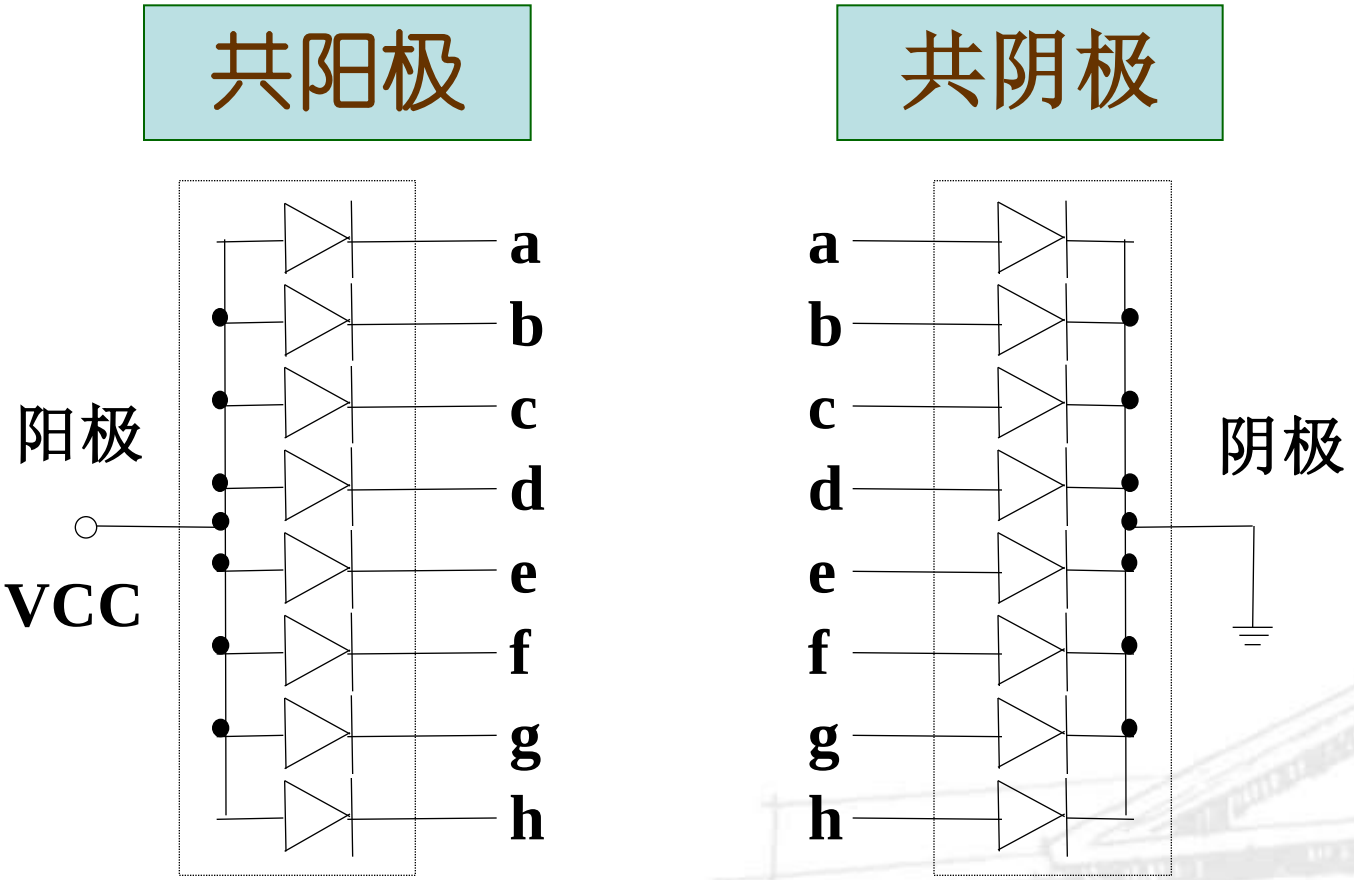
– LED数码管广泛用于微机系统，作为显示设备。

➢LED数码管的结构

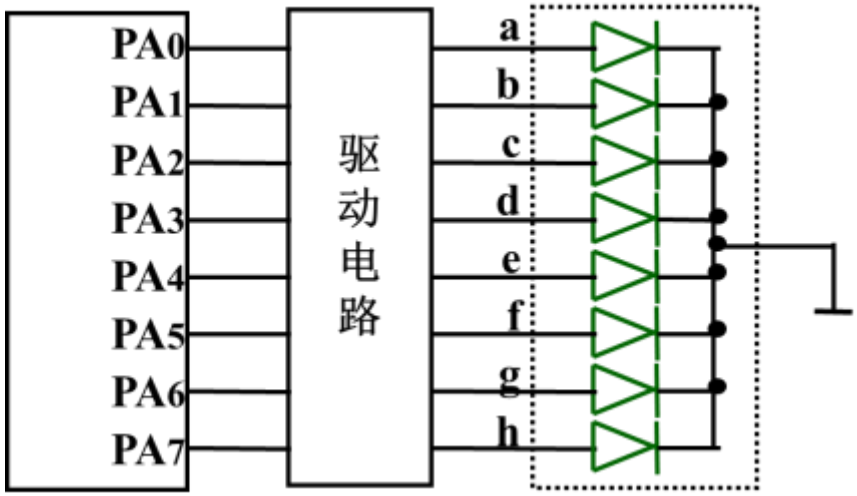
- 主要部分是7段发光管
- 顺时针分别称为a、b、c、d、e、f、g
- 有的产品还附带有一个小数点h
- 通过7个发光段的不同组合
  - 主要显示0～9
  - 也可以显示A～F（实现16进制数的显示）
  - 还可以显示个别特殊字符，如 - 、P 等



➢LED数码管的结构



➢驱动电路

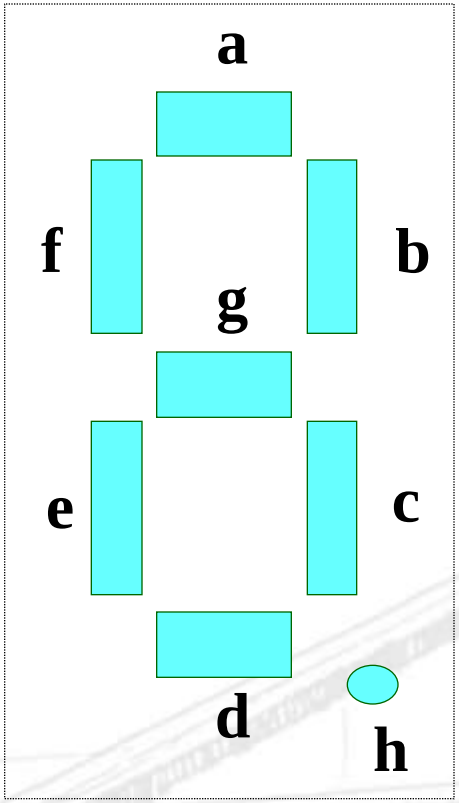

PA0
PA1
PA2
PA3
PA4
PA5
PA6
PA7
驱动电路
a
b
c
d
e
f
g
h


R0 200
SEG0
SEG1
SEG2
SEG3
SEG4
SEG5
SEG6
SEG7
10
11
13
14
15
16
17
18

➢数码管的显示

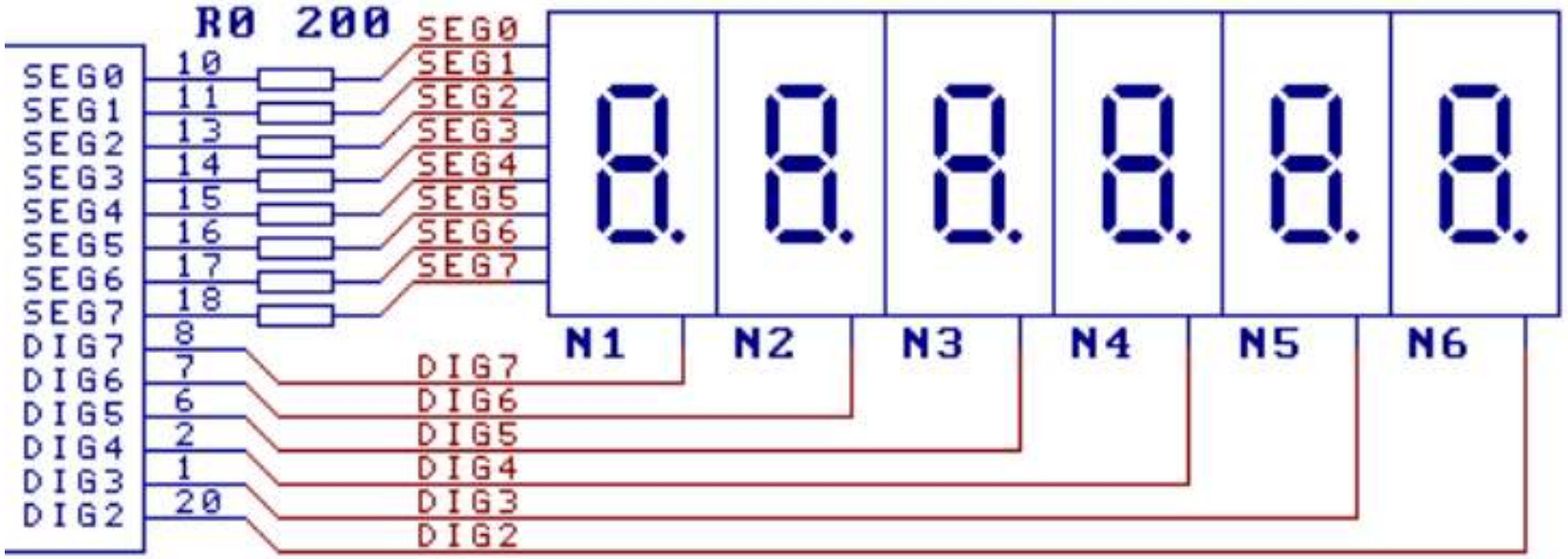
– 通过不同的编码，显示不同的数组和字符：

显示字形	共阳极字段码 gfedcba	共阴极字段码 gfedcba	显示字形	共阳极字段码 gfedcba	共阴极字段码 gfedcba
0	C0H	3FH	9	90H	6FH
1	F9H	06H	A	88H	77H
2	A4H	5BH	b	83H	7CH
3	B0H	4FH	C	C6H	39H
4	99H	66H	d	A1H	5EH
5	92H	6DH	E	86H	79H
6	82H	7DH	F	8EH	71H
7	F8H	07H	“熄灭”	FFH	00H
8	80H	7FH			



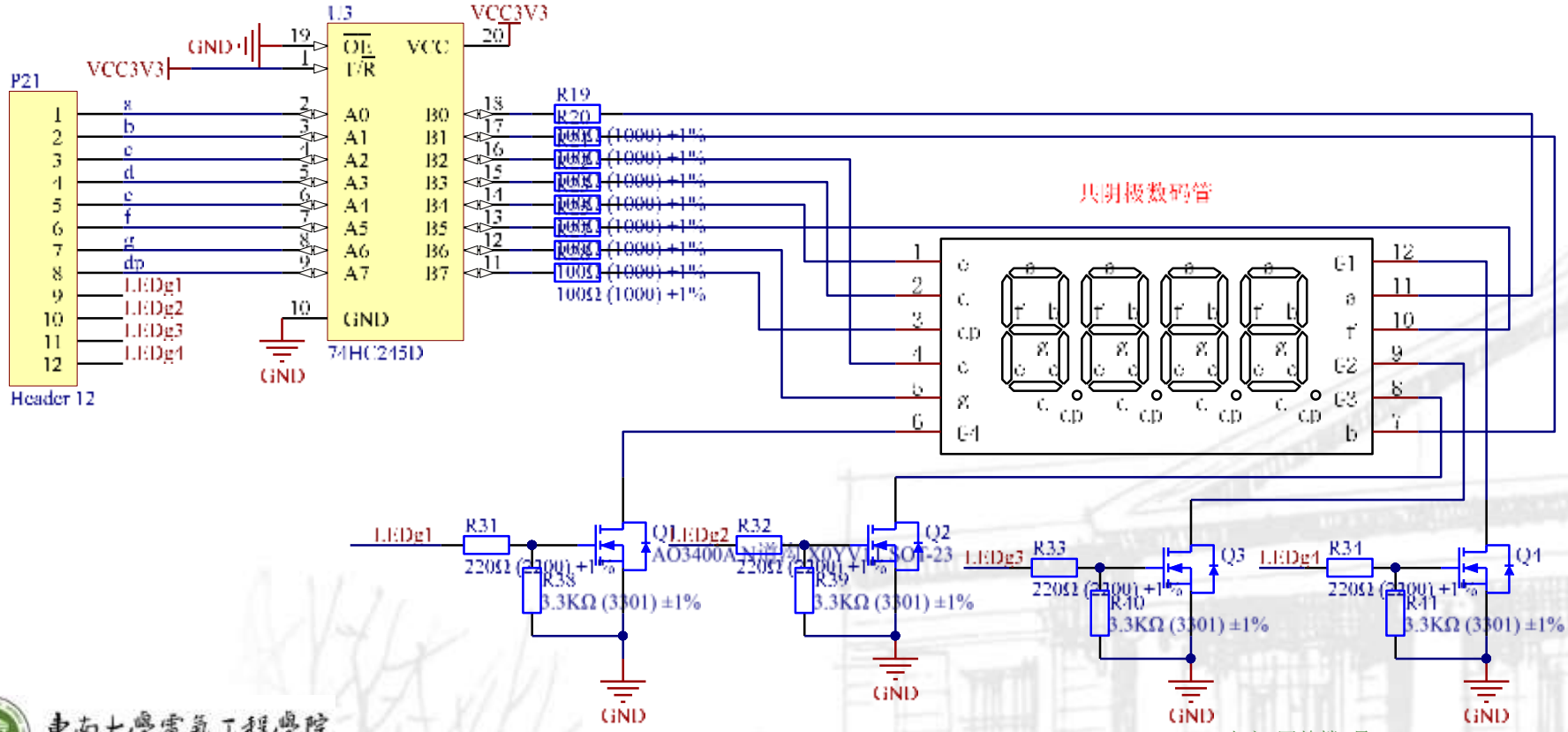
➢多个LED数码管的显示

- 共阴极连接，轮流控制N1到N6的阳极为高电平，每次只显示一个数字。
- 快速轮流显示N1到N6，就可以利用视觉暂留现象，以为6个数字同时显示了。

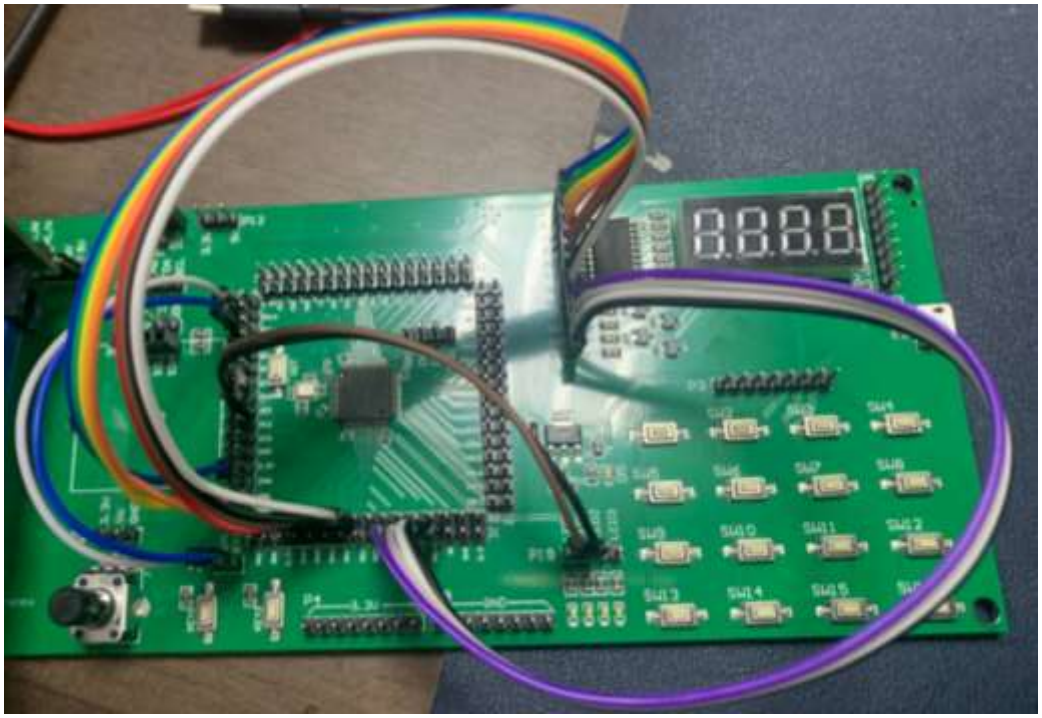


➢开发板上的数码管电路（四位数码管）

- 阳极由74HC245缓冲器经过限流电阻驱动。
- 阴极由一个MOS管控制，MOS管导通，该位的数码管能显示数字。MOS管不导通，则该位的数码管不显示数字。
- 轮流导通MOS管，可以实现多位显示的功能。
- 所有的控制信号用过P21排针与MCU相连。
- 在使用数码管之前，需要先用线将PB21的引脚与MCU的引脚连起来。



➢开发板上的数码管电路（四位数码管）

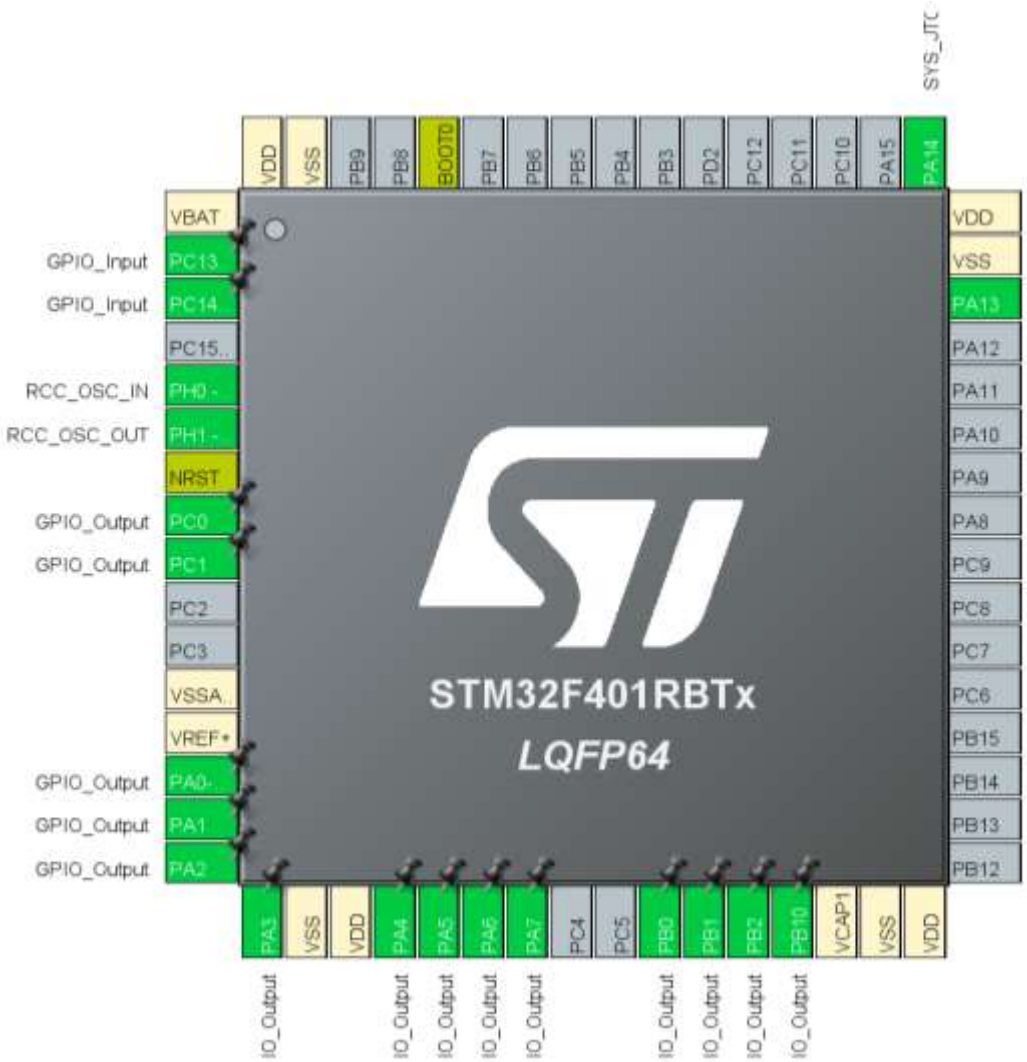


使用PA0、PA1、PA2、PA3、PA4、PA5、PA6、PA7分别控制a、b、c、d、e、f、g、dq。

使用PB0、PB1、PB2、PB10控制四个MOS管。

➢开发板上的数码管电路（四位数码管）

– 正确配置引脚



➢开发板上的数码管电路（四位数码管）

– 修改配置，并生成代码后，在/* USER CODE BEGIN PV */和/* USER CODE END PV */之间定义一个数组，用于方便显示数码管上的数字。

```
const uint8_t  leddata[]={
                          0x3F,  //"0"
                          0x06,  //"1"
                          0x5B,  //"2"
                          0x4F,  //"3"
                          0x66,  //"4"
                          0x6D,  //"5"
                          0x7D,  //"6"
                          0x07,  //"7"
                          0x7F,  //"8"
                          0x6F,  //"9"
                          0x77,  //"A"
                          0x7C,  //"B"
                          0x39,  //"C"
                          0x5E,  //"D"
                          0x79,  //"E"
                          0x71,  //"F"
                          0x76,  //"H"
                          0x38,  //"L"
                          0x37,  //"n"
                          0x3E,  //"u"
                          0x73,  //"P"
                          0x5C,  //"o"
                          0x40,  //"-"
                          0x00  //熄灭
};
```

➢开发板上的数码管电路（四位数码管）

– 要想在第一个LED数码管上显示数字，需要控制PB0引脚，通过开通MOS管，将第一个数码管所有的LED的阴极连接到0V。同时关闭其他数码管阴极的MOS管

```
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET);
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_RESET);
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_2, GPIO_PIN_RESET);
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, GPIO_PIN_RESET);
```

– 然后给GPIOA的ODR寄存器赋值，点亮数码管上的LED等，显示数字：

```
GPIOA->ODR=leddata[5];
```

开发板上的数码管电路（四位数码管）

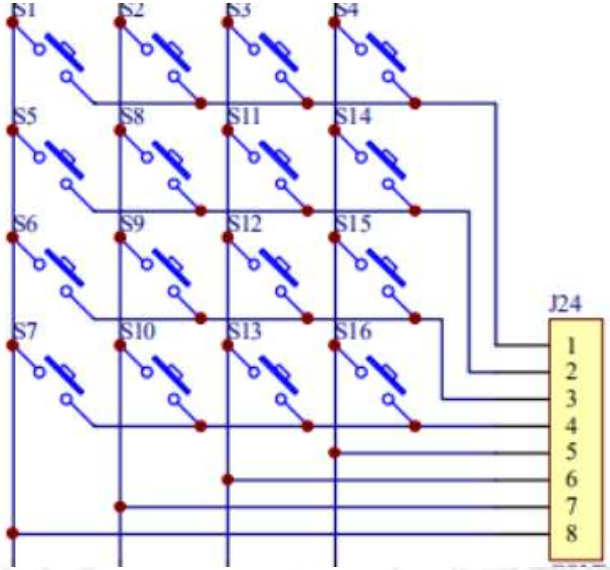
- 如果要在四个数码管上显示数字，需要将要显示的数字轮流输出出来，如要显示1234在四个数码管上。
- 在主循环中编辑代码：

```
/* USER CODE BEGIN WHILE */
while (1)
{

    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_2, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, GPIO_PIN_RESET);
    GPIOA->ODR=leddata[4];
    HAL_Delay(1);

    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_SET);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_2, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, GPIO_PIN_RESET);
    GPIOA->ODR=leddata[3];
    HAL_Delay(1);

    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_2, GPIO_PIN_SET);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, GPIO_PIN_RESET);
    GPIOA->ODR=leddata[2];
    HAL_Delay(1);

    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_2, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, GPIO_PIN_SET);
    GPIOA->ODR=leddata[1];
    HAL_Delay(1);

/* USER CODE END WHILE */
```

➢矩阵键盘

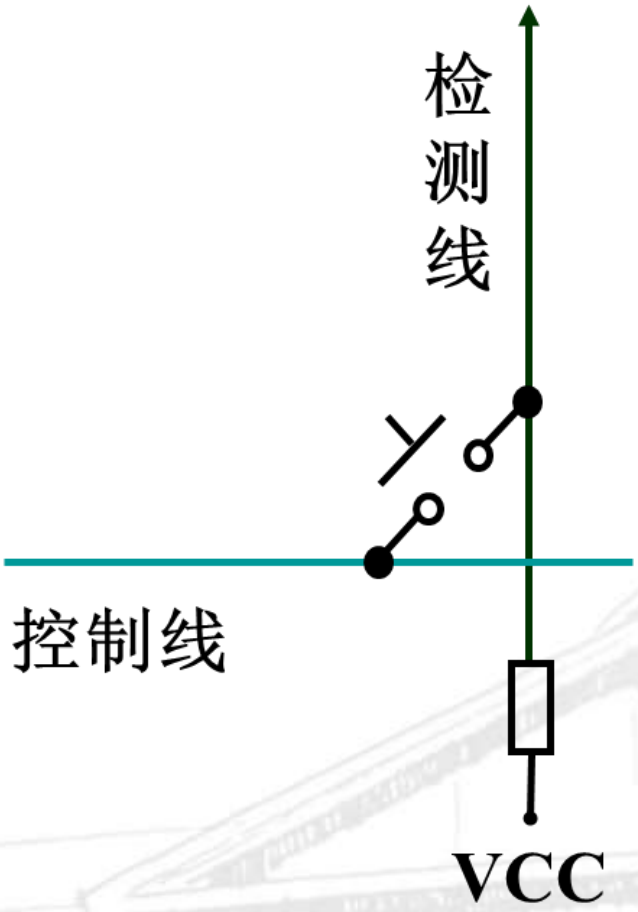

1
2
3
A
4
5
6
B
7
8
9
C
*
0
#
D


C4
C3
C2
C1
R1
R2
R3
R4
S13
S14
S15
S16


S1
S2
S3
S4
S5
S8
S11
S14
S6
S9
S12
S15
S7
S10
S13
S16
J24
1
2
3
4
5
6
7
8

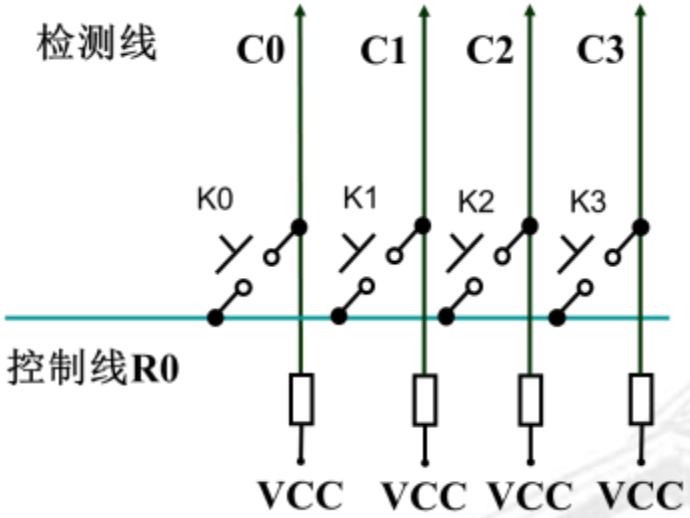
➢矩阵键盘 按键检测

- 行线作为控制线，列线为检测线。
- 用上拉电阻把检测线上拉至VCC
- 按键按下时，它所在的控制线与检测线相短接。
- 如果在控制线上输出低电平。按键没有按下时，检测线上的电压为高电平。按键按下时，检测线上的电压为低电平。
- 如果在控制线上输出高电平。那么无论有没有按键按下，检测线都是高电平



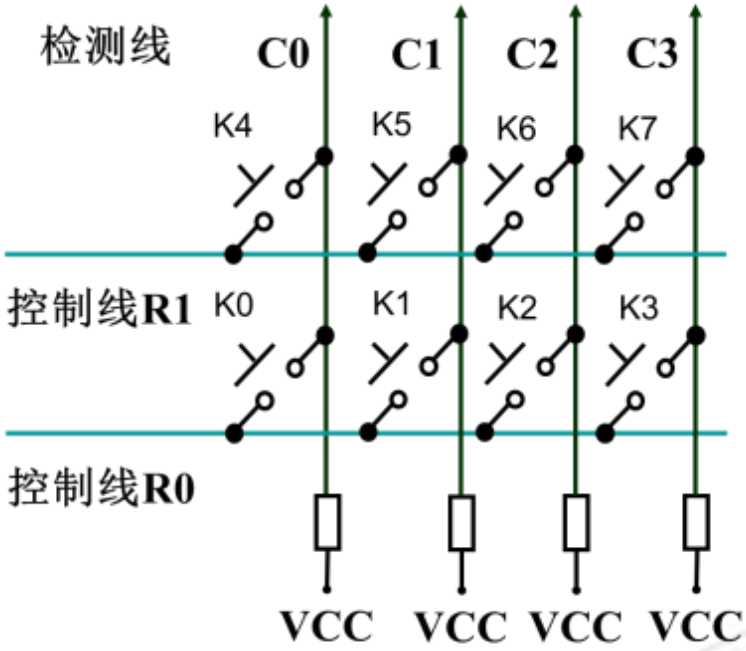
➢矩阵键盘 按键检测

- 行线作为控制线，列线为检测线。
- 用上拉电阻把检测线上拉至VCC
- 按键按下时，它所在的控制线与检测线相短接。
- 如果在控制线上输出低电平。按键没有按下时，检测线上的电压为高电平。按键按下时，检测线上的电压为低电平。
- 如果在控制线上输出高电平。那么无论有没有按键按下，检测线都是高电平
- 如果一行上有不同的按键按下，那么四个检测线上的电平也会有所不同。因此我们可以分辨按键在哪一个列。



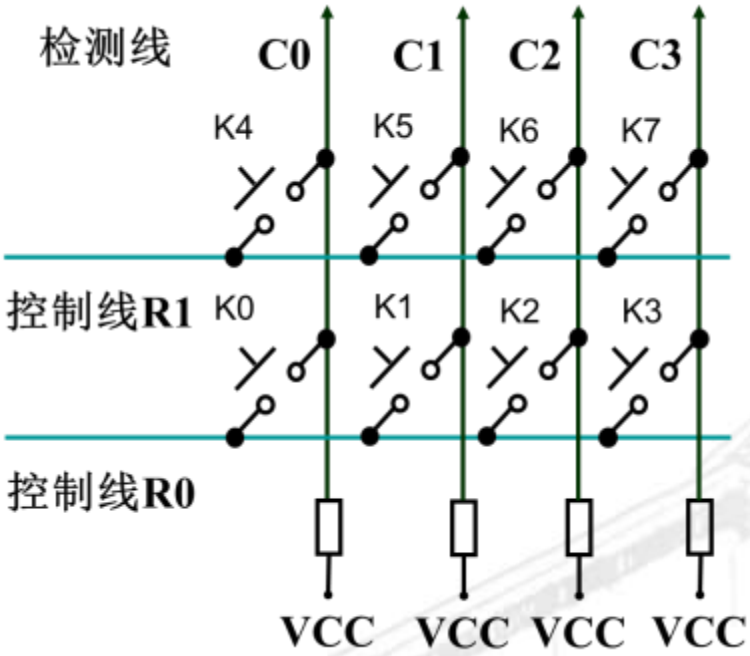
➢矩阵键盘 按键检测

- 多个行：
- 如果控制线R0和R1都输出电平，那么如果检测到C1为低电平，还是无法区分是K1按下还是K5按下



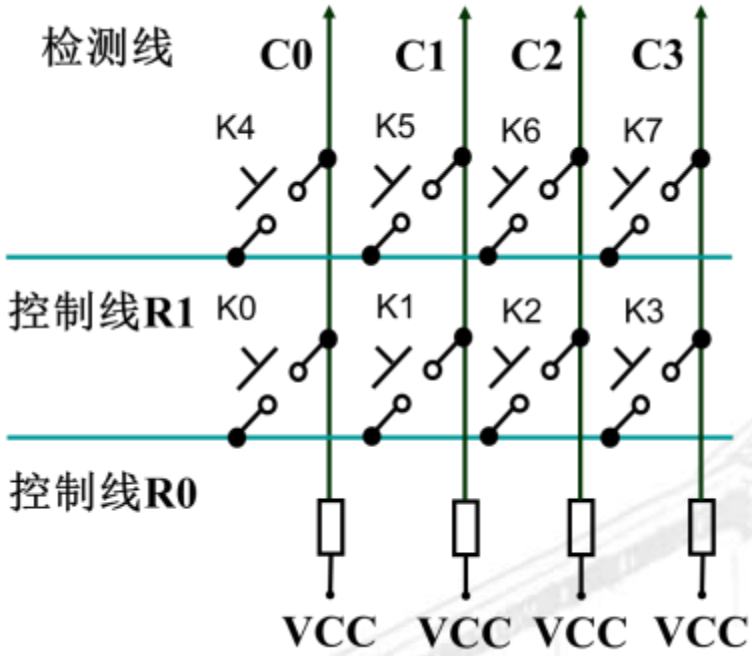
➢扫描法：

- 先把R0设置输出低电平，R1设置输出高电平。如果C0-C3上检测到了低电平，说明按键在R0行。通过C0-C3的电平，可以判断是K0-K3中的哪个按键被按下。
- 然后把R0设置输出高电平，R1设置输出低电平。如果C0-C3上检测到了低电平，说明按键在R1行。通过C0-C3的电平，可以判断是K4-K7中的哪个按键被按下。
- 如果把R2、R3行画出来，也是以此类推。
- 扫描法是最常用的方法。



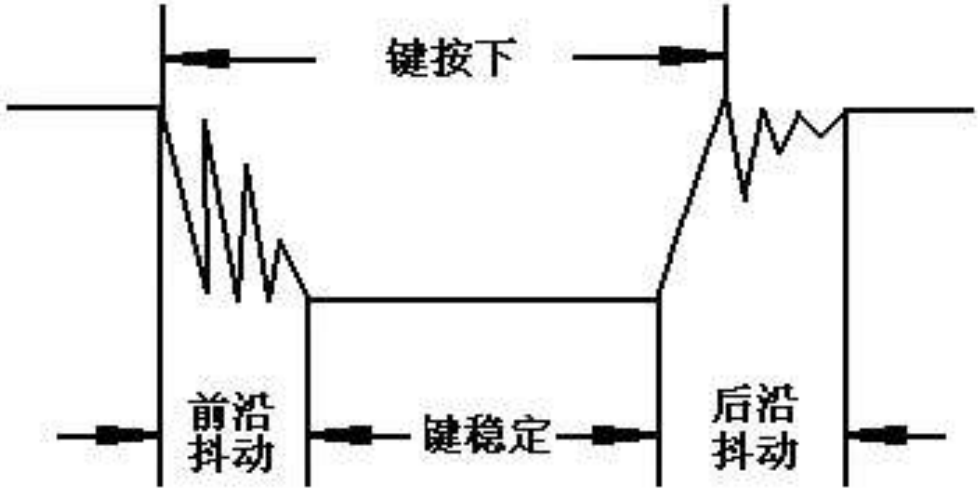
➢反转法：

- 先把R0、R1都设置为输出、然后输出低电平。
- 把C0、C1、C2、C3 都设置为输入，上拉。
- 如果C0-C3上检测到低电平，可以通过C0-C3的电平，判断按键在哪一列 M;
- 然后把C0、C1、C2、C3都设置为输出、然后输出低电平。
- 把R0、R1、都设置为输入，上拉。
- 如果R0-R1上检测到低电平，可以通过R0-R1的电平，判断按键在哪一行 N;
- 这样就可以检测出按键位于N行M列。



➢ 按键的抖动

- 机械按键存在抖动现象
- 当按下或释放一个键时，往往会出现按键在闭合位置和断开位置之间跳几下才稳定到闭合状态
- 抖动的持续时间通常不大于10ms。



- **采用硬件消抖电路或软件延时方法解决**

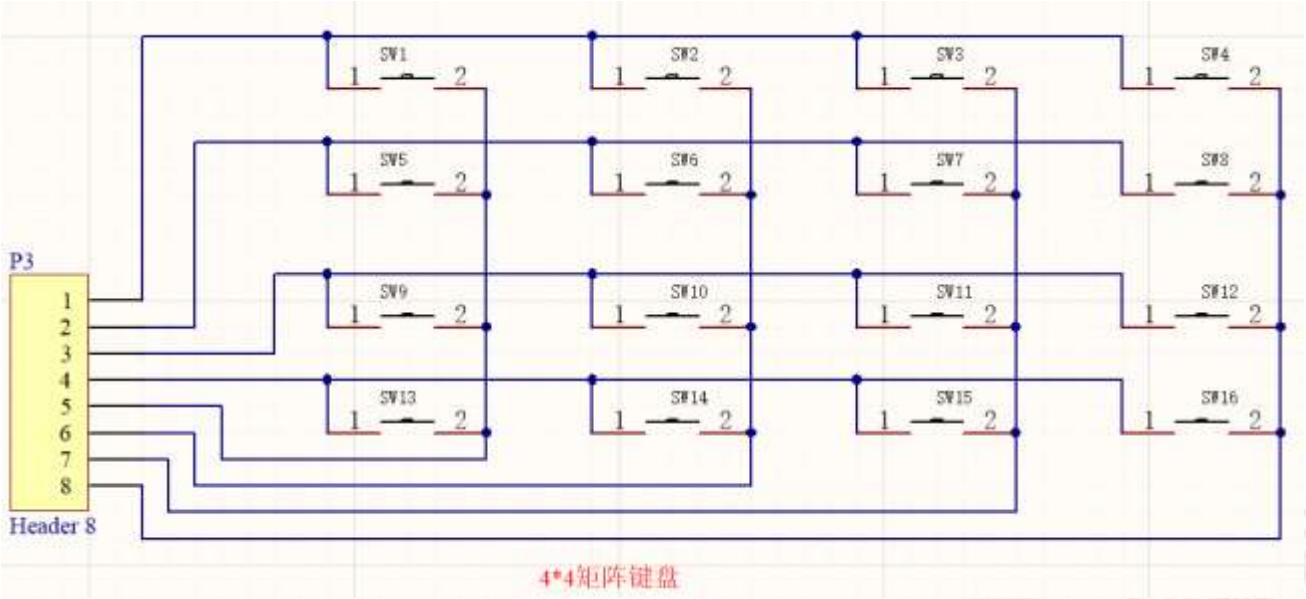
重键

- 两个或多个键同时闭合。
- 出现重键时，读取的键值必然出现一个以上的0，需要根据应用选择合适的办法应对重键。
  - 简单情况：不予识别，认为是错误的按键
  - 通常情况：只承认先识别出来的键
  - 正常的组合键：都识别出来

➢实验套件底板上的矩阵键盘

– 只将行线和列线引出来。没有连接上拉电阻

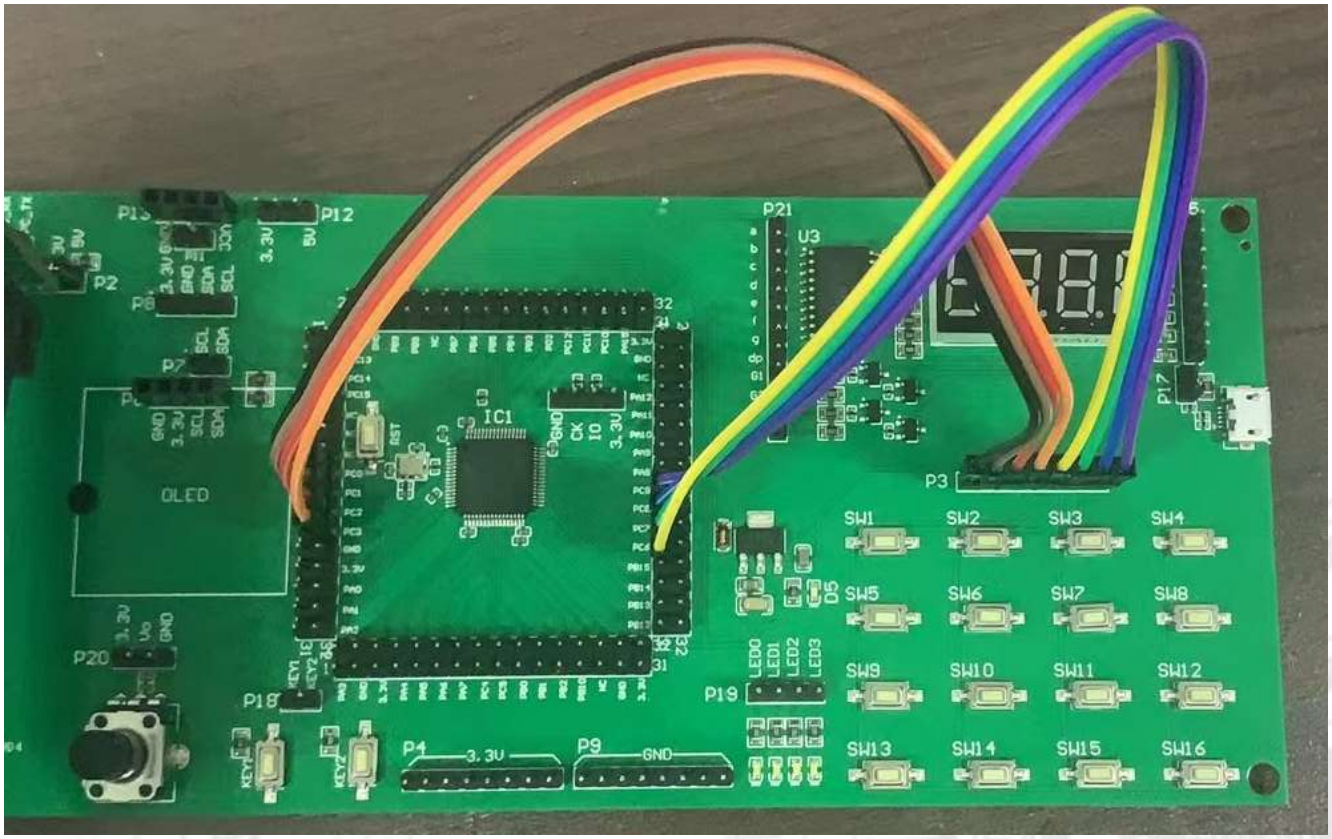
– 由于STM32的引脚自带上拉电阻，无需外接上拉电阻。



➢实验套件底板上的矩阵键盘

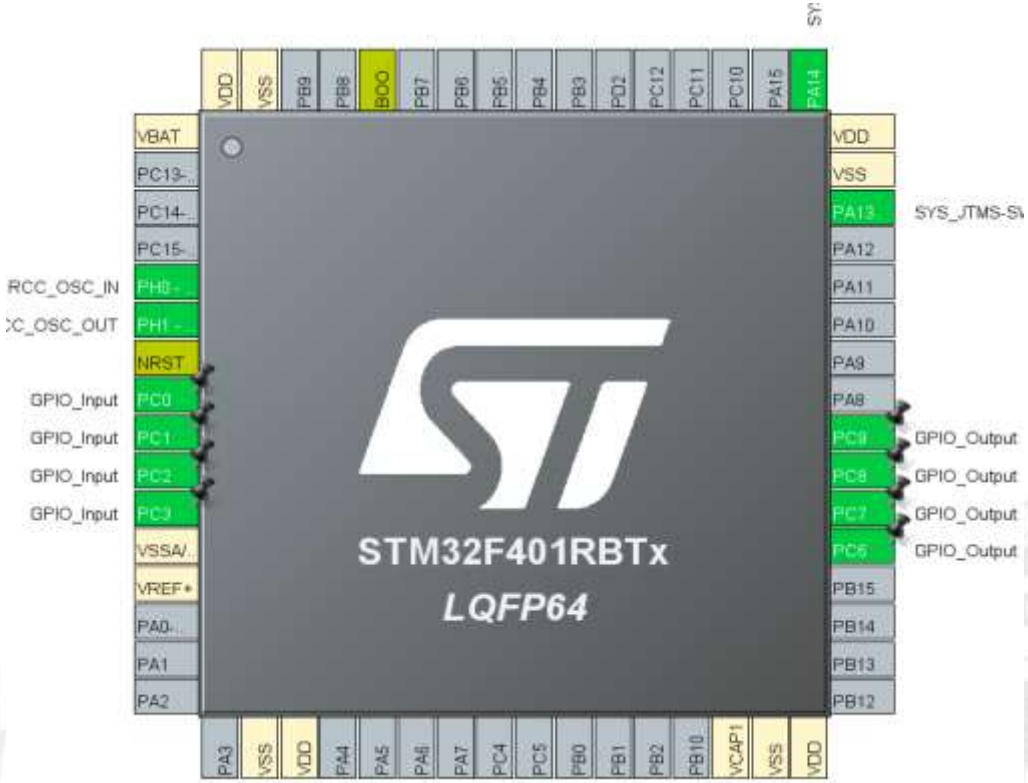
– 在使用键盘之前，首先将键盘的接口与MCU的引脚连接起来。

– 假设键盘的行线连接到PC0、PC1、PC2、PC3四个引脚，键盘的列线连接到PC6、PC7、PC8、PC9四个引脚



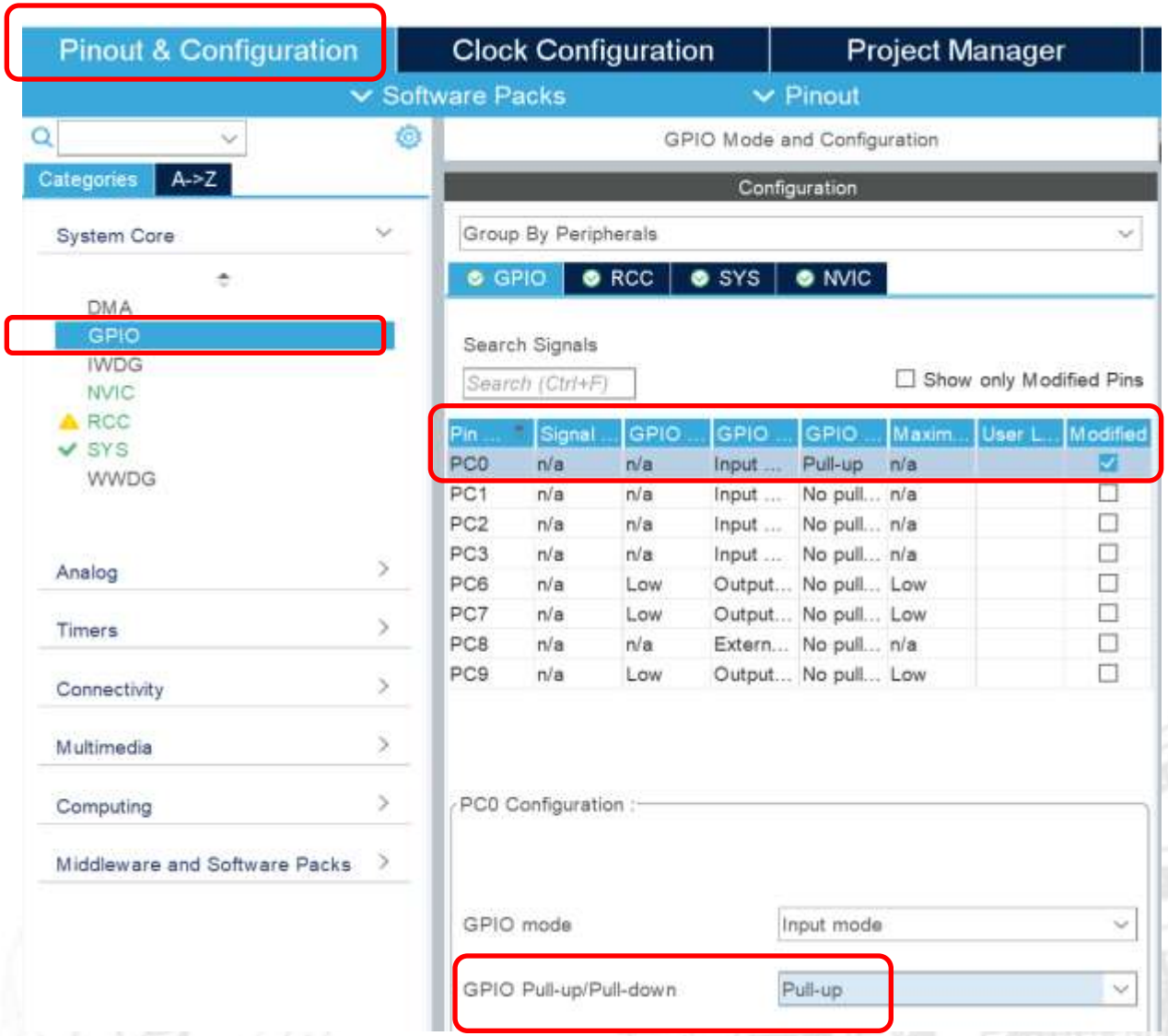
➢实验套件底板上的矩阵键盘

- 连接好硬件后，使用器件参数配置工具，修改MCU的配置，将行线（PC0、PC1、PC2、PC3）当作检测线，启用上拉电阻。列线（PC6、PC7、PC8、PC9）当作控制线。
- 首先在helloworld项目的基础上，将PC0、PC1、PC2、PC3设置为输入，将PC6、PC7、PC8、PC9设置为输出。



- 实验板上的矩阵键盘
  - 在左侧的Pinout & Configuration窗口里点击System Core -> GPIO
  - 在GPIO的配置界面里点击PC0，然后在下面出现的PC0 Configuration窗口中，将GPIO Pull-up/Pull-down选项改为Pull-up。
  - 然后在GPIO的配置界面里点击PC1，然后在下面出现的PC1 Configuration窗口中，将GPIO Pull-up/Pull-down选项改为Pull-up。
  - 根据相同的方法，启用PC2、PC3的上拉电阻。
  - 配置完成后，保存并生成代码。

➢实验板上的矩阵键盘


Pinout & Configuration
Clock Configuration
Project Manager
Software Packs
Pinout
Categories
A->Z
System Core
DMA
GPIO
IWDG
NVIC
RCC
SYS
WWDG
Analog
Timers
Connectivity
Multimedia
Computing
Middleware and Software Packs
GPIO Mode and Configuration
Configuration
Group By Peripherals
GPIO
RCC
SYS
NVIC
Search Signals
Search (Ctrl+F)
Show only Modified Pins
Pin ... Signal ... GPIO ... GPIO ... GPIO ... Maxim... User L... Modified
PC0 n/a n/a Input ... Pull-up n/a
PC1 n/a n/a Input ... No pull... n/a
PC2 n/a n/a Input ... No pull... n/a
PC3 n/a n/a Input ... No pull... n/a
PC6 n/a Low Output... No pull... Low
PC7 n/a Low Output... No pull... Low
PC8 n/a n/a Extern... No pull... n/a
PC9 n/a Low Output... No pull... Low
PC0 Configuration :
GPIO mode
Input mode
GPIO Pull-up/Pull-down
Pull-up

➢实验板上的矩阵键盘

– 在/* USER CODE BEGIN PV */和/* USER CODE END PV */之间**定义扫描程序用到的全局变量，方便调试。**

```
/* USER CODE BEGIN PV */
int32_t keyValue=0;
int32_t row,col,rowLineState;
  /* USER CODE END PV */
```

– 然后在/* USER CODE BEGIN WHILE */和/* USER CODE END WHILE */之间添加**扫描程序**的代码。

```
/* USER CODE BEGIN WHILE */
 HAL_GPIO_WritePin(GPIOC,GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9,GPIO_PIN_RESET);
 while (1)
 {

                while((GPIOC->IDR & 0x0f) != 0x0f){
                                int32_t NewkeyValue=0;
                                HAL_Delay(10);
                                row=0;
                                col=0;
                                NewkeyValue=0;
                                for(col=0;col<4;col++){
                                                if(col==0){
                                                                HAL_GPIO_WritePin(GPIOC,GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9,GPIO_PIN_SET);
                                                                HAL_GPIO_WritePin(GPIOC, GPIO_PIN_6, GPIO_PIN_RESET);
                                                }
                                                if(col==1){
                                                                HAL_GPIO_WritePin(GPIOC,GPIO_PIN_6|GPIO_PIN_8|GPIO_PIN_9,GPIO_PIN_SET);
                                                                HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7, GPIO_PIN_RESET);
                                                }
                                                if(col==2){
                                                                HAL_GPIO_WritePin(GPIOC,GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_9,GPIO_PIN_SET);
                                                                HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_RESET);
                                                }
                                                if(col==3){
                                                                HAL_GPIO_WritePin(GPIOC,GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8,GPIO_PIN_SET);
                                                                HAL_GPIO_WritePin(GPIOC, GPIO_PIN_9, GPIO_PIN_RESET);
                                                }
                                                HAL_Delay(1);
                                                rowLineState=GPIOC->IDR&0x0f;
                                                for(row=0;row<4;row++){
                                                                if( (rowLineState | 0x0e) == 0x0e ){
                                                                                NewkeyValue=row*4+col+1;
                                                                                break;
                                                                }
                                                                rowLineState>>=1;

                                                }
                                                if(NewkeyValue) break;
                                }
                                if(NewkeyValue) {
                                                keyValue=NewkeyValue;
                                                break;
                                }

                }
                HAL_GPIO_WritePin(GPIOC,GPIO_PIN_6|GPIO_PIN_7|GPIO_PIN_8|GPIO_PIN_9,GPIO_PIN_RESET);


    /* USER CODE END WHILE */
```

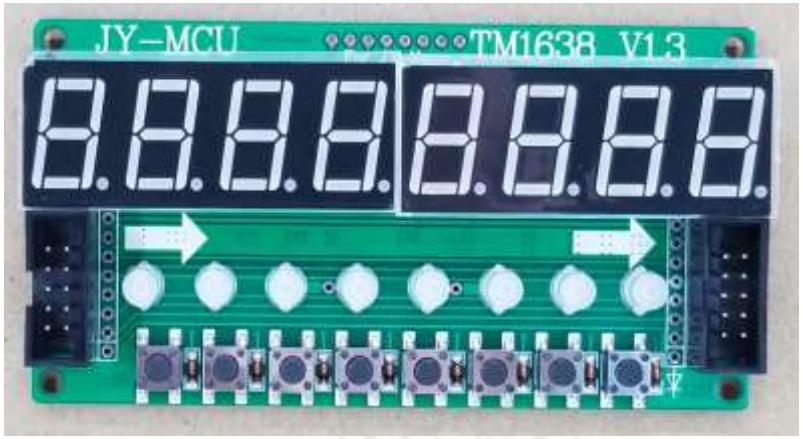
➢实验板上的矩阵键盘

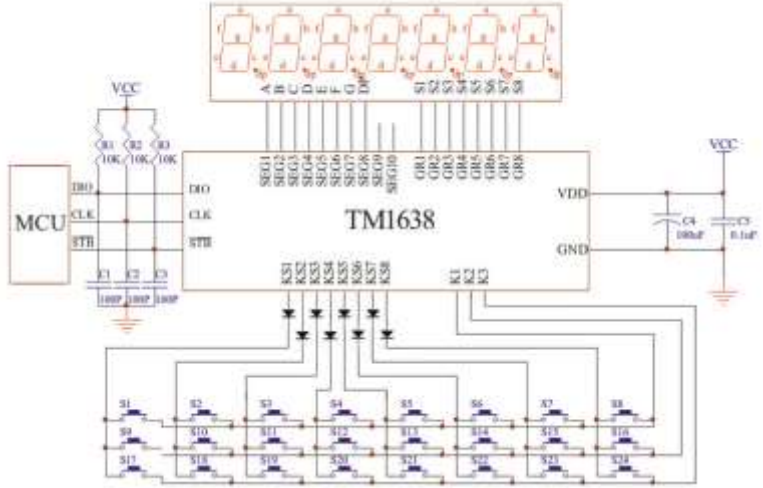
- 进入主循环之前，将列线（控制线）全部拉低。
- 如果没有按键按下，则行线上都是高电平。如果有按键按下，则行线上会出现低电平。使用while((GPIOC->IDR & 0x0f) != 0x0f){语句，检测行线是否全是高电平。一旦发现行线不全是高电平，则进入扫描程序。
- 进入扫描程序后，需要延时一段时间用于消抖，然后开始逐列扫描。
- 将要扫描的列置低电平，其他列置高电平。延时1ms待电平稳定。然后读取行线的值，存在rowLineState变量中。
- 如果被扫描的行里，有按键按下，那么rowLineState变量的低四位中，一定存在0。使用if((rowLineState | 0x0e) == 0x0e)语句检测rowLineState变量的最低位是否为零。然后使用rowLineState >>=1;语句，将rowLineState变量的第二位移动至最低位，再检测最低位是否为零，重复4次，就可以找出rowLineState的0在哪一位。
- 检测到rowLineState变量存在0后，使用NewkeyValue=row*4+col+1;语句标记该按键所在的行和列，形成该按键的编号，然后退出扫描程序。将检测到的新按键值NewkeyValue赋值给keyValue。
- 如果rowLineState变量不存在0，则继续下一行的扫描。
- 最后将所有控制线拉低，方便下一次检测。
- 可以使用现场表达式观察keyValue变量，查看按键值。

➢矩阵键盘和数码管的应用

- 矩阵键盘和数码管，即使使用了扫描功能，依旧占用了过多的引脚。
- 在嵌入式系统中，主控MCU一般不会使用宝贵的引脚资源直接控制矩阵键盘和数码管。
  - 通过异步串行通信端口控制的显示模块，除电源VCC和GND之外，只需要两个引脚

  - 通过专用芯片TM1638控制的数码管和键盘模块，与主控MCU之间，通过IIC总线通信，除电源VCC和GND之外，只需要两个引脚





➢作 业：

– 数码管和键盘都用到了扫描的方式，如果同时使用这两个模块，引脚能否复用?