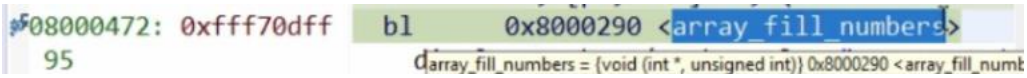
18.single stepping

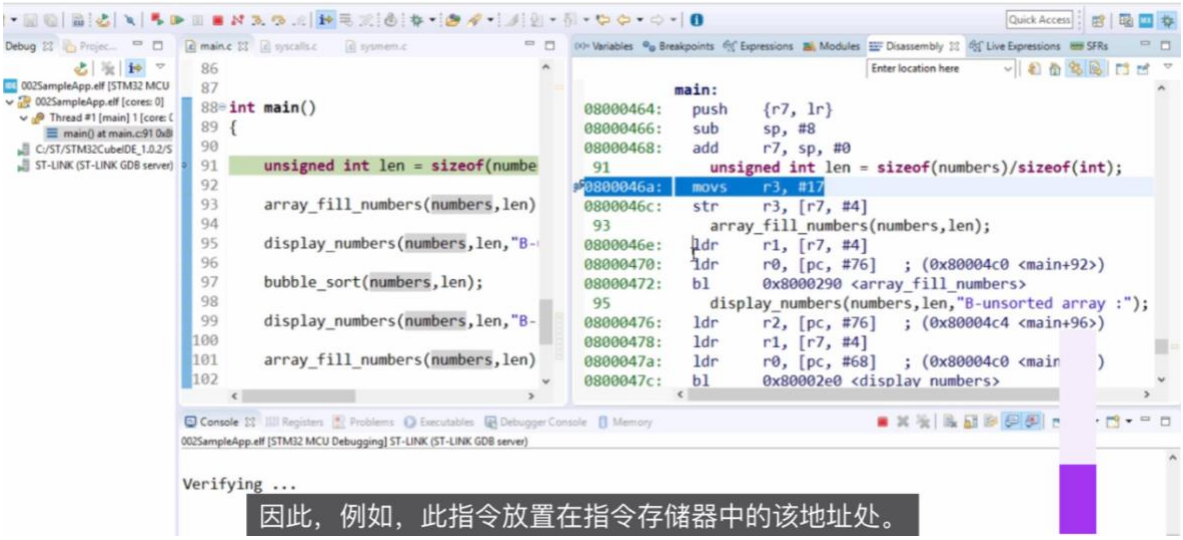
Debug 有三个选项，step into, step over 和 step return

Step into 是单步执行，step over 是跳过一个函数的结果，直接执行下一个语句，不过如果已经进入函数的内部的话，那么 step into 和 step over 的效果是一样的，这个时候如果想立刻退出当前函数，用 step out 或者 step return。

19.disassembly and register window



Bl 是指 array_fill_number 存储在 0x8000290 这个地址里



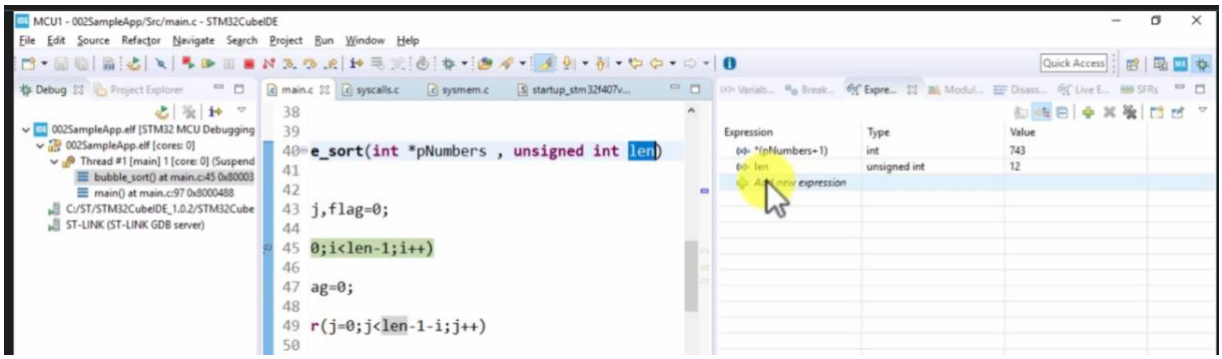
这个 0800046a 是指这个指令的地址(flash 的地址)，window-showview-register 可以看寄存器的值

20.breakpoints

可以最多设置 5 个断点（硬件要求），然后通过运行 resume 可以直接跳到下一个断点

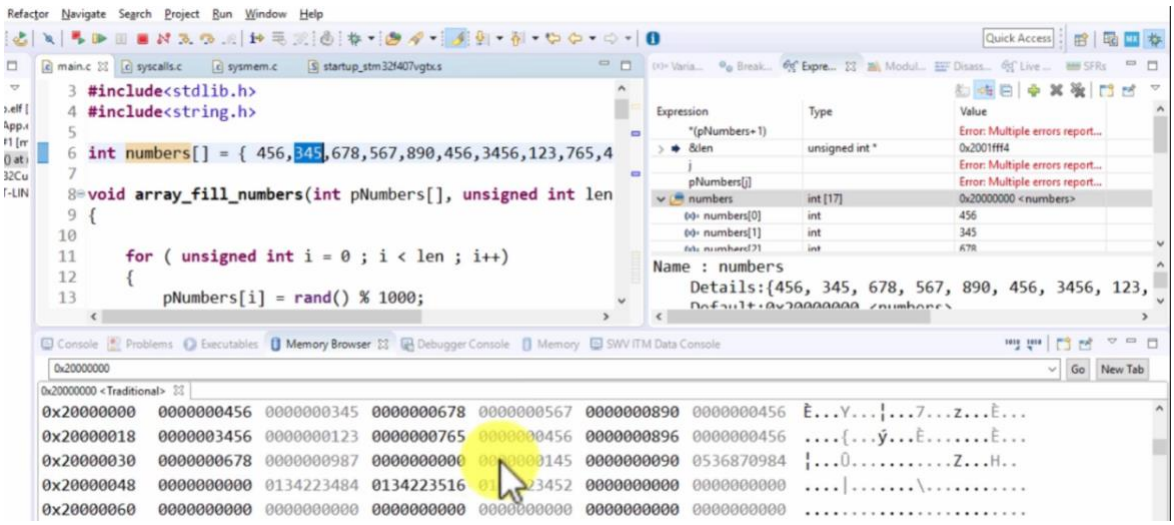
21.expression and variable windows

可以从 window-showview-variable 获得 debug 的当前函数的各种变量，也可以通过 expression 执行一些简单的操作，比如看看这个 pnumbers 下一个指针的值之类的，编写短表达式并获得值。Expression 是当前函数的堆栈框架。



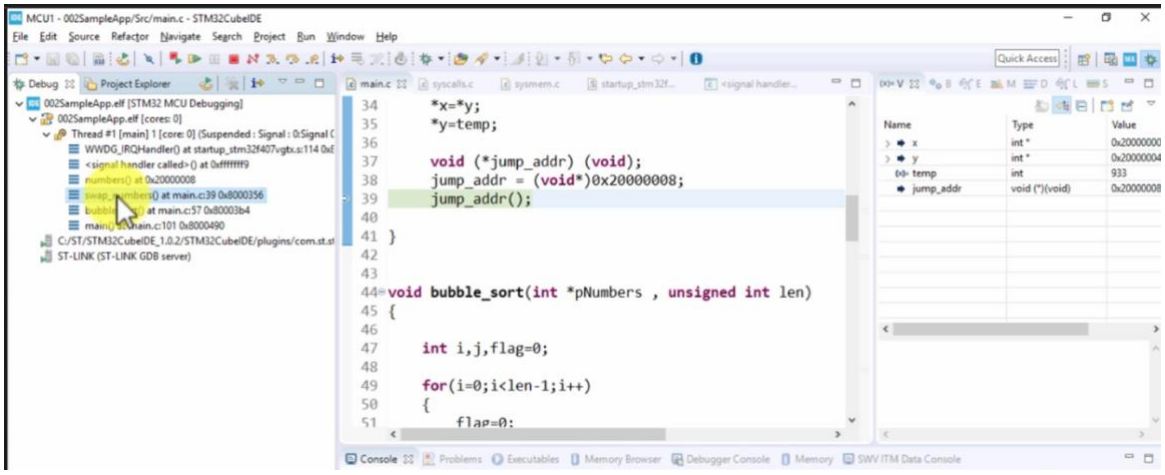
22.memory browser window

可以在这里看到地址的对应值。



23.call stack and fault analyzer

当程序出错时，通过下图左边的鼠标处的 stack 一个个排查大约是哪里出错误了，如果有这个 signal handler caller 也可以打开 fault analyzer(window-老地方)看看原因



24.Data watch-points

可以在 breakpoint 里设置一个 data watch point，来监视变量或者地址的值什么时候改变了

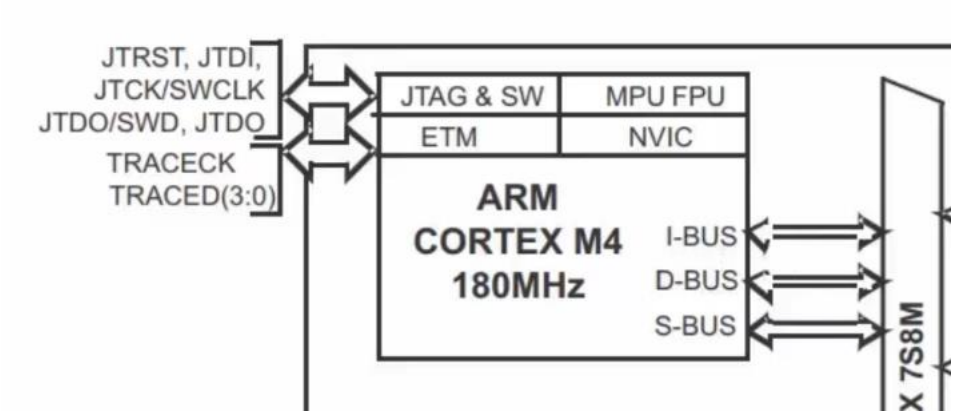
25.SFR windows

可以看各种外围寄存器比如 gpio，spi 的值

26.other basic feature

代码提示: hold left ctrl 然后按 space， ctrl+o 表示所有函数的列表

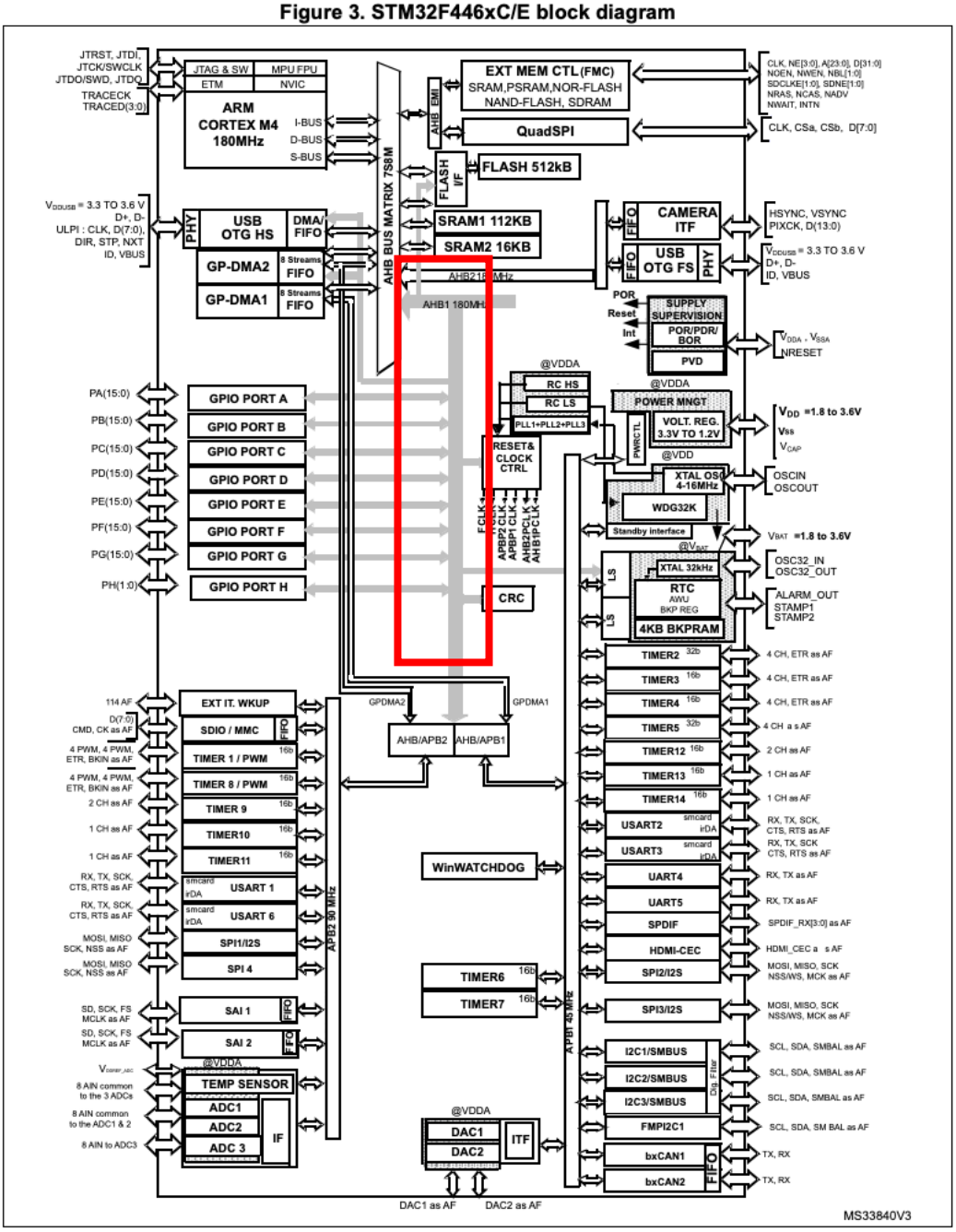
30.MCU bus interfaces



I bus 是用于执行指令的，D bus 是用于传输数据，S bus 是系统总线，I 和 D 必须连到 flash。
Flash 要保存指令，const data，向量表，这三个都叫 AHB-Lite bus

```
notes about MCU bus interfaces
==============================================
if the instructions are present in between the memory locations 0x00000000 to 0x1FFFFFFC then
the Cortex Processor will fetch the instructions using ICODE interface.

if the instructions are present outside of 0x00000000 to 0x1FFFFFFC then processor fetches the instructions
over the sytem bus.

if the data is present in between the memory locations 0x00000000 to 0x1FFFFFFF, then processor fetches th
data over D-CODE bus interface.

if the data is present outside 0x00000000 to 0x1FFFFFFF memory locations then , the data will be fetched
over the sytem bus.
```

Figure 3. STM32F446xC/E block diagram



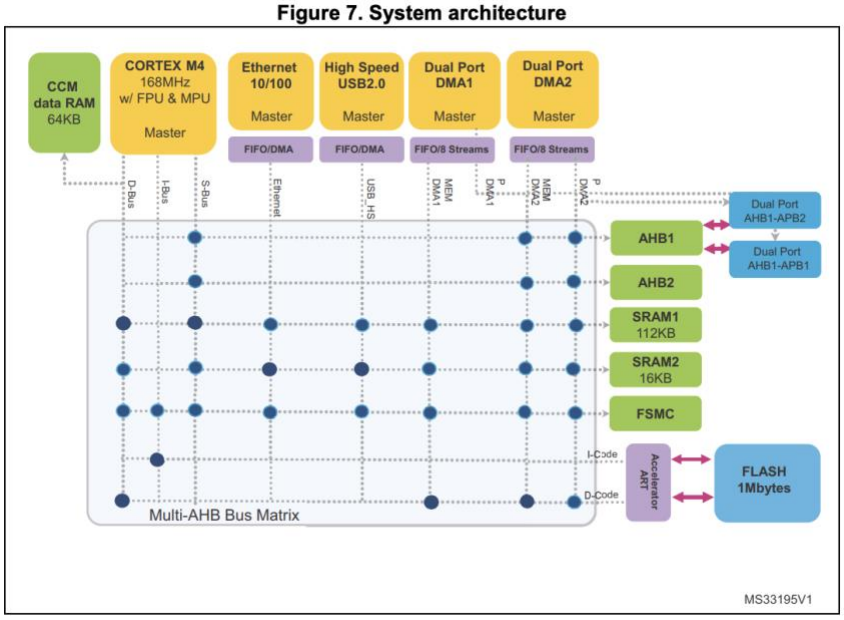
这里的 AHB1 和 AHB2 实际上都是 system bus，因为连的外设地址都在 0x1FFFFFF 的外面，SRAM 也是连接到 system bus

```
1. Is it true that, System Bus is not connected to FLASH memory ?
True

2. Processor can fetch instructions from SRAM over i-code bus  T/F?
False

3. System Bus can operate at the speed up to 180MHz T/F?
True ..

4. SRAMs are connected to System Bus T/F ?
True
```

```
7. Processor can fetch instructions as well as data simultaneously from SRAM T/F ?
False.

8. Processor can fetch instructions as well as data simultaneously from FLASH T/F ?
True. icode and dcode.
```

33.Bus Matrix

Figure 7. System architecture



Bus Matrix 就是执行谁可以连接到谁的，执行这些规则的复杂电子电路。

34. MCU clocking system

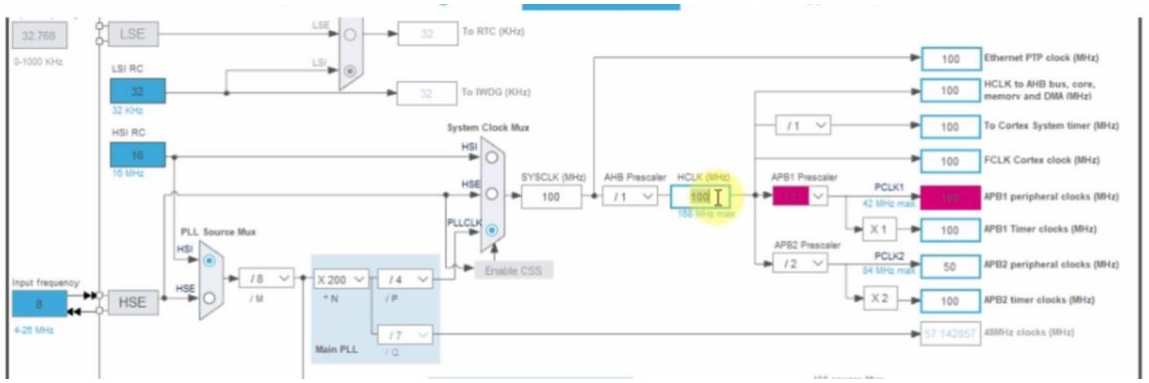
MCU needs clock， and clock comes from clock source.第一种水晶是外部设备，而二三都是 MCU 内部的东西

```
Clock
=================
Sources :
1) Crystal Oscillator (external to the MCU)
2) The RC Oscillator  (Internal to the MCU)
3) The PLL (Phase Locked Loop) (Internal to the MCU)
```

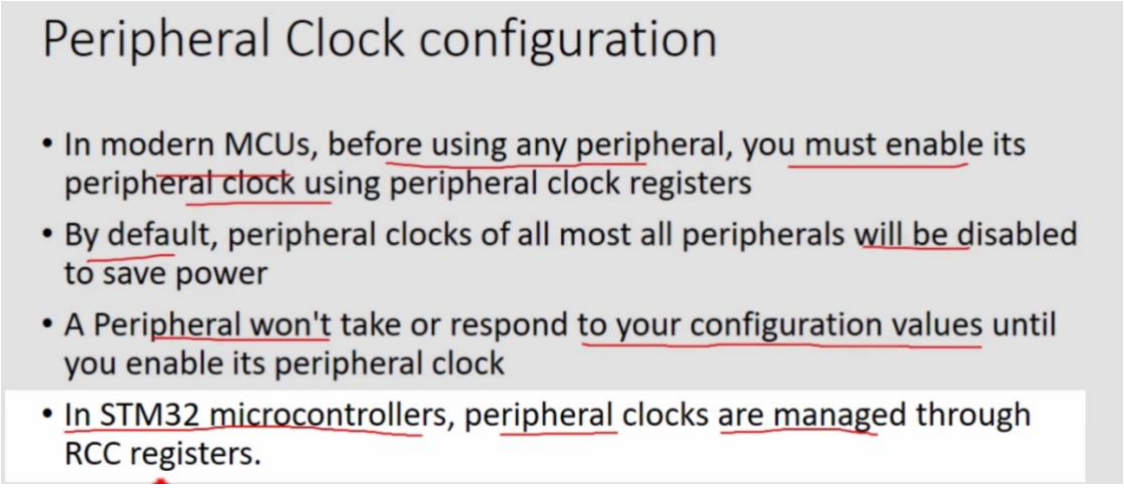
HSE 就是 high speed external，crystal oscillator

```
Summary of HSE
================================
HSE can be provided to the MCU via a crystal or external source
(from another ckt or from another MCU)

On the STM32-DISC board, HSE is 8MHz provided by the onboard crystal.

on NUCLEO board, HSE is of 8MHz pulled from ST-LINK Ckt
```

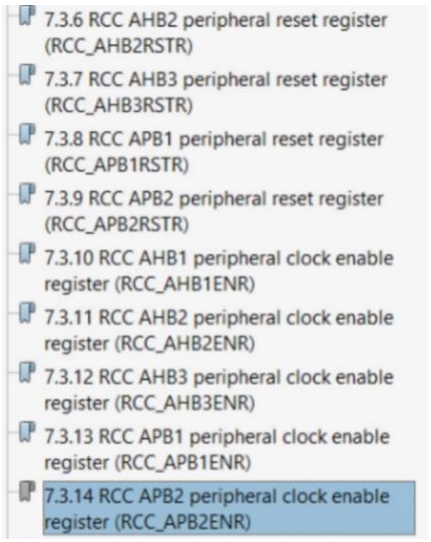
HSI 是 high speed internal，是默认的时钟源，用的系统内部的 RC
PLL 引擎是锁相环，是利用乘法系统将时钟频率提高，高于 HIS 或者 HSE



37.peripheral clock configuration
在使用任何外设存储器之前，必须 enable 他的时钟，在 STM32 里，这些时钟都通过 RCC 寄存器管理
RCC: reset and clock control



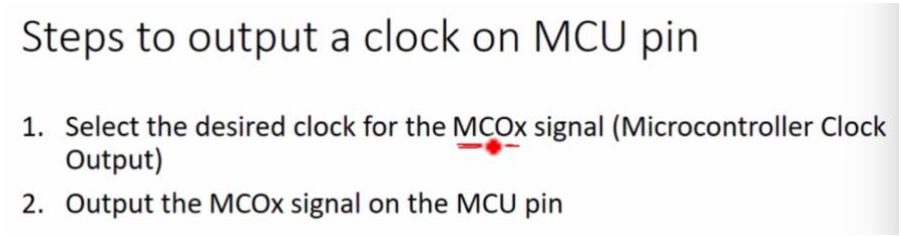
为了启用时钟，你必须先确定这个外围设备连接在哪一条总线上，然后去 RCC 寄存器，去搜索寄存器，
找到这个总线的 RCC 寄存器，然后再去 enable 那个外围设备的 clock

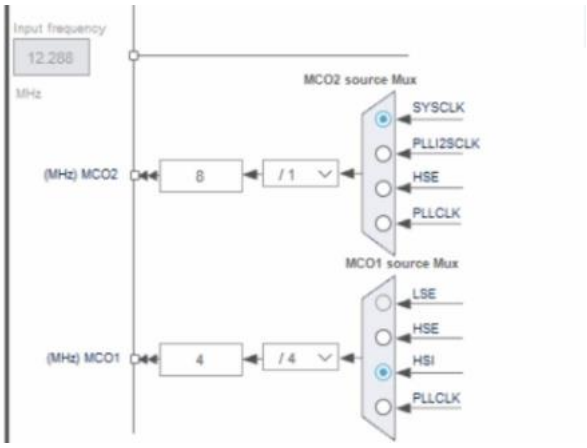



```c
#define ADC_BASE_ADDR               0x40012000UL

#define ADC_CR1_REG_OFFSET          0x04UL

#define ADC_CR1_REG_ADDR            (ADC_BASE_ADDR + ADC_CR1_REG_OFFSET )

#define RCC_BASE_ADDR              0x40023800UL

#define RCC_APB2_ENR_OFFSET        0x44UL

#define RCC_APB2_ENR_ADDR          (RCC_BASE_ADDR + RCC_APB2_ENR_OFFSET )

int main(void)
{
    uint32_t *pAdcCr1Reg =   (uint32_t*) ADC_CR1_REG_ADDR;
    uint32_t *pRccApb2Enr =  (uint32_t*) RCC_APB2_ENR_ADDR;

    //1.Enable the peripheral clock for ADC1
    *pRccApb2Enr |= ( 1 << 8);

    //2. modify the ADC cr1 register
    *pAdcCr1Reg |= ( 1 << 8);

    for(;;);
}
```

38.HSI measurement

测量 HIS 的频率并把频率输出到一个引脚上，MCO 代表微控制器时钟输出信号，是一个信号，不是一个外围设备





在 RCC_CFGR 里可以配置上图的选项，让输出 MCO1 的信号
可以通过下表把 MCO1 的信号映射到 PA8(GPIO A 的第八个引脚)，然后根据封装确定是第几个引脚

Table 9. Alternate function mapping

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10
		SYS	TIM1/2	TIM3/4/5	TIM8/9/10/11	I2C1/2/3	SPI1/SPI2/I2S2/I2S2ext	SPI3/I2Sext/I2S3	USART1/2/3/I2S3ext	UART4/5/USART6	CAN1/2 TIM12/13/14	OTG_FS/OTG_HS
Port A	PA0	-	TIM2_CH1_ETR	TIM 5_CH1	TIM8_ETR	-	-	-	USART2_CTS	UART4_TX	-	-
	PA1	-	TIM2_CH2	TIM5_CH2	-	-	-	-	USART2_RTS	UART4_RX	-	-
	PA2	-	TIM2_CH3	TIM5_CH3	TIM9_CH1	-	-	-	USART2_TX	-	-	-
	PA3	-	TIM2_CH4	TIM5_CH4	TIM9_CH2	-	-	-	USART2_RX	-	-	OTG_HS_ULPI_D0
	PA4	-	-	-	-	-	SPI1_NSS	SPI3_NSS I2S3_WS	USART2_CK	-	-	-
	PA5	-	TIM2_CH1_ETR	-	TIM8_CH1N	-	SPI1_SCK	-	-	-	-	OTG_HS_ULPI_CK
	PA6	-	TIM1_BKIN	TIM3_CH1	TIM8_BKIN	-	SPI1_MISO	-	-	-	TIM13_CH1	-
	PA7	-	TIM1_CH1N	TIM3_CH2	TIM8_CH1N	-	SPI1_MOSI	-	-	-	TIM14_CH1	-
	PA8	MCO1	TIM1_CH1	-	-	[illegible]	-	-	USART1_CK	-	-	OTG_FS_SOF

然后可以用 USB logic analyzer(也是个硬件，通过 usb 连接)来替代示波器去查看信号(www.saleae.com),399 美元

Exercise :
HSE measurement (Discovery)

1. Enable the HSE clock using HSEON bit (**RCC_CR)**
2. Wait until HSE clock from the external crystal stabilizes (only if crystal is connected) (indicates if the high-speed external oscillator is stable or not)
3. Switch the systemclock to HSE (**RCC_CFGR)**
4. Do MCO1 settings to measure it

42. MCU vector table

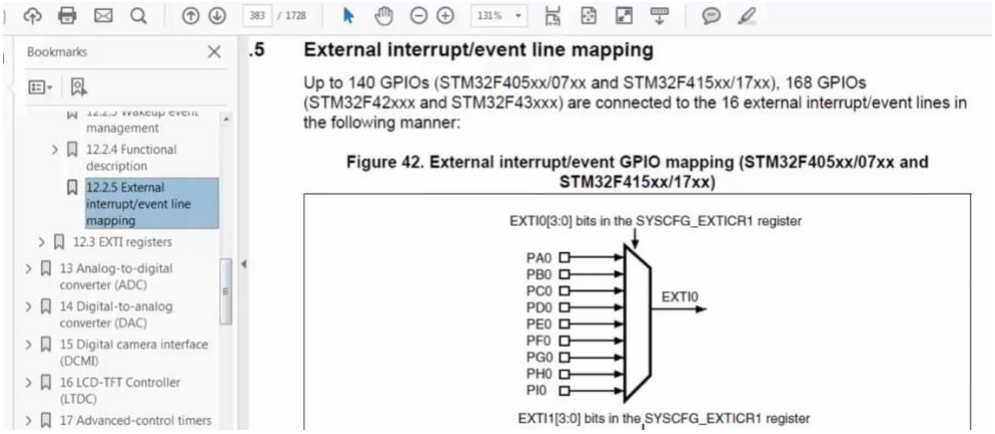
Positio	Priorit	Type of priority	Acronym	Description	Address
	-	-	-	Reserved	0x0000 0000
	-3	fixed	Reset	Reset	0x0000 0004
	-2	fixed	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000 0008
	-1	fixed	HardFault	All class of fault	0x0000 000C
	0	settable	MemManage	Memory management	0x0000 0010
	1	settable	BusFault	Pre-fetch fault, memory access fault	0x0000 0014
	2	settable	UsageFault	Undefined instruction or illegal state	0x0000 0018
	-	-	-	Reserved	0x0000 001C - 0x0000 002B

第一列的 position 又叫 IRQ number,

43.understanding MCU interrupt Design

```
Some peripherals deliver their interrupt to the NVIC over the EXTI
line.

some peripherals deliver their interrupt directly to the NVIC

this is the design of ST. you may find some other design in TI.
```

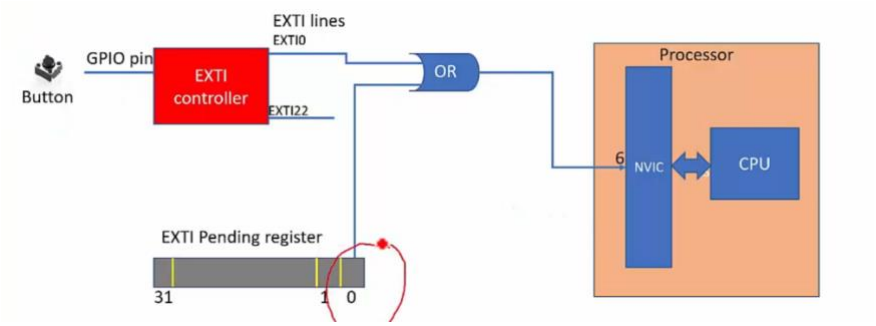
GPIO 传递中断的方式: PA0 就是 GPIO A 的第 0 个引脚(引脚 23)



Summary: Button interrupt

- How does a button issue interrupt to the processor in STM32?

1. The button is connected to a GPIO pin of the microcontroller
2. The GPIO pin should be configured to input mode
3. The link between a GPIO port and the relevant EXTI line must be established using the SYSCFG_EXTICRx register.
4. Configure the trigger detection (falling/rising/both)for relevant EXTI line (This is done via EXTI controller registers)5.Implement the handler to service the interrupt.



如果按下按钮，EXTI controller 和 EXTI pending register 都变 1，所以 NVIC 变 1 触发中断，中断结束以后，EXTI controller 会自动清除但是 pending register 不会，所以要在中断程序里向 pending register 写入 1 注意不是 0，(写入 1 会让他变 0)让中断消失。

47.volatile

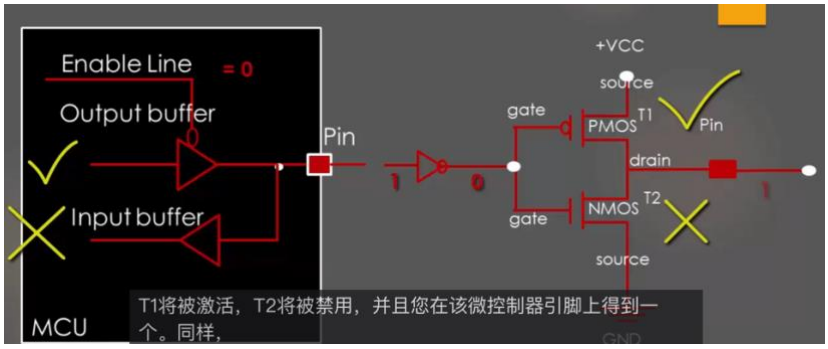
每当处理内存位置时，最好将 volatile 关键字和用于访问该内存位置的指针变量一起使用

Uint32_t volatile *p

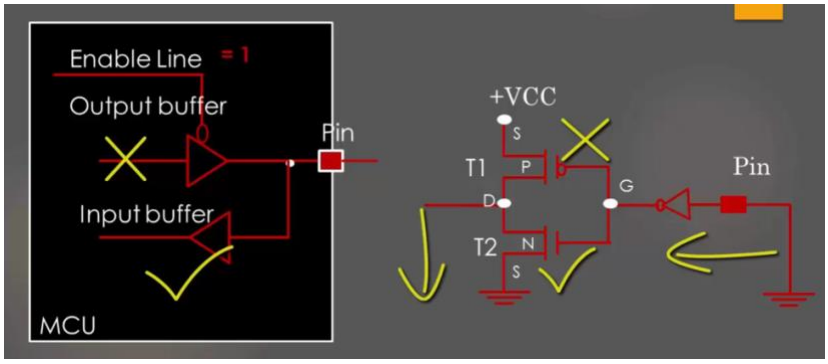
49.GPIO behind the scene

Gpio 内部是下图的样子，如果使能是 0，那么启用 output buffer， enable line is configured by GPIO

Control register



output buffer 如左图

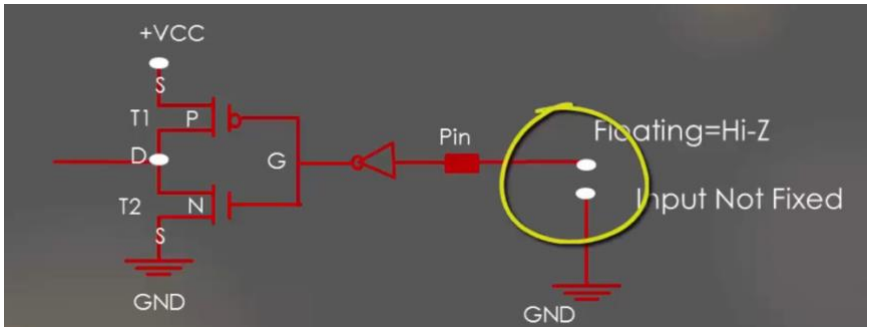


input buffer 如左图

50.GPIO input mode with high impedance

High impedance 就是浮动引脚，不和高电压也不和低电压连接来保持引脚浮动，这样会导致电流泄漏，

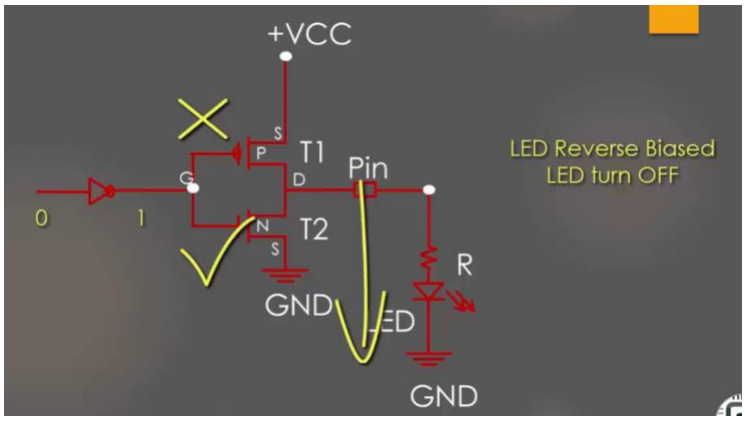
可以通过上拉/下拉电阻让 pin 处于高/低电平,输入模式必须接上拉电阻



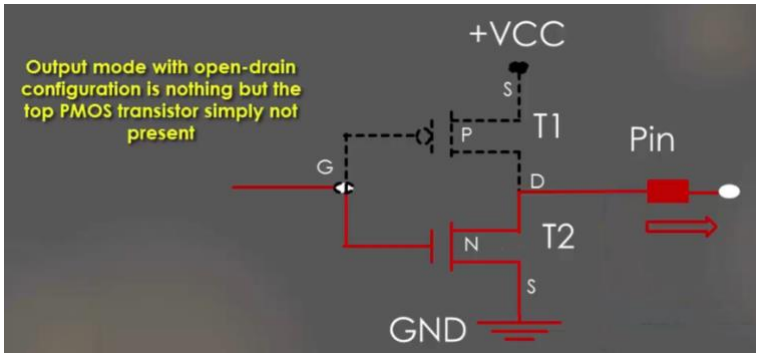
53.GPIO output mode

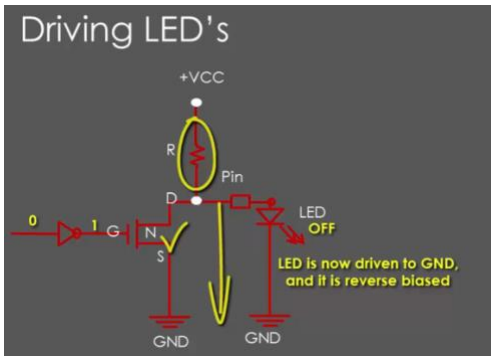
GPIO output 有两种模式，有 push pull state 和 open drain state

Push pull state

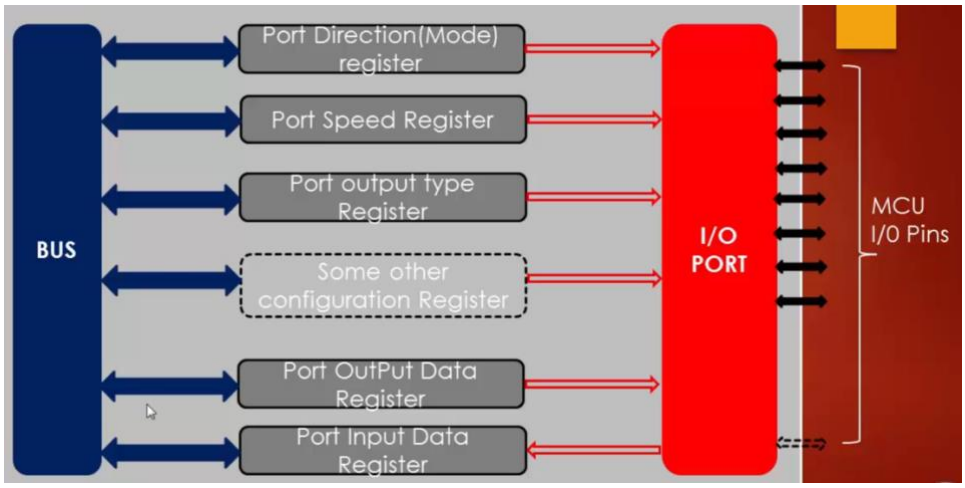


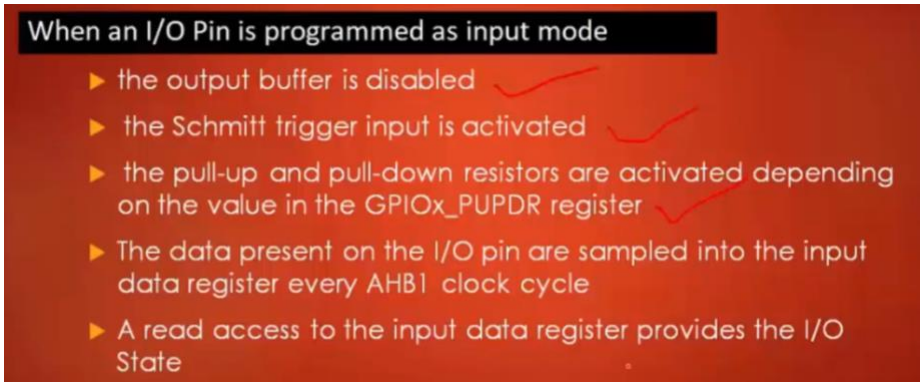
Open drain state，就是上面的二极管不用了，因为不能提供高电平，所以必须接上拉电阻，为了输出高电平



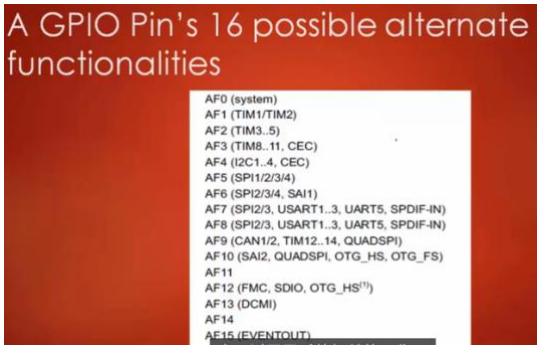


55.GPIO 都通过系统总线与 MCU 相连，同时有多个 register





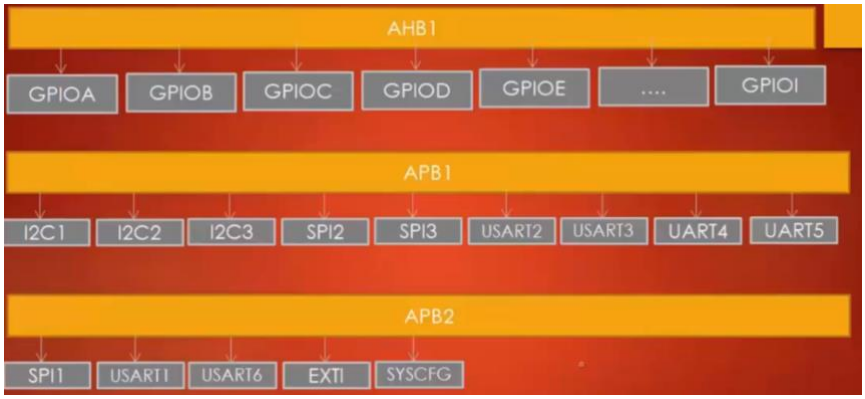
68.Alternate functionality of GPIO pin

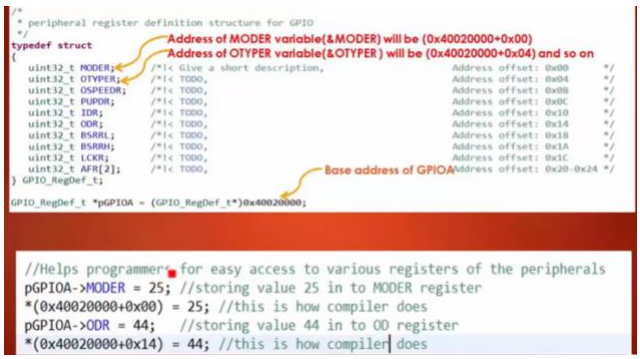


75.include path settings
在 properties 的 c/c++ build 的 settings 的 compiler 里的 include path 里可以选择头文件的位置

ROM 就是 system memory(和 flash 不一样)

首先需要建立是 stm32 的头文件，把 SRAM1，FLASH，ROM，AHB，APB 的基地址都整理出来，
然后在整理下图的每一个外围设备的每一个基准地址





在每一个外设里建立一个结构体，结构体里包含他的寄存器，注意这里面每个寄存器都是 32 位的正好差四个字节，而实际上 datasheet 里这些寄存器也是差四个字节，所以要注意顺序。
同时记得每一个元素要用 volatile

```
235 #define GPIOA_REG_RESET()          do{ (RCC->AHB1RSTR |= (1 << 0)); (RCC->AHB1RSTR &= ~(1 << 0)); }while(0)
236
237 //some generic macros
238
239 #define ENABLE              1
240 #define DISABLE             0
241 #define SET                 ENABLE
242 #define RESET               DISABLE
```

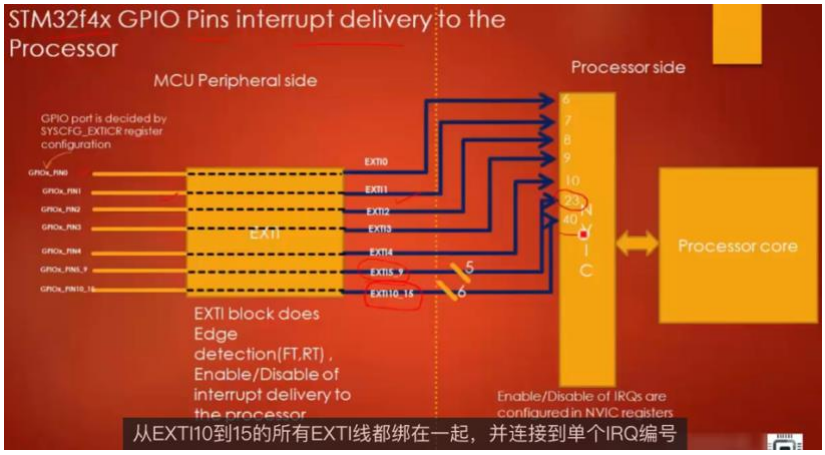
do .... while .. condition zero loop :

This is a technique in 'C' programming to execute multiple 'C' statements using single 'C' macro

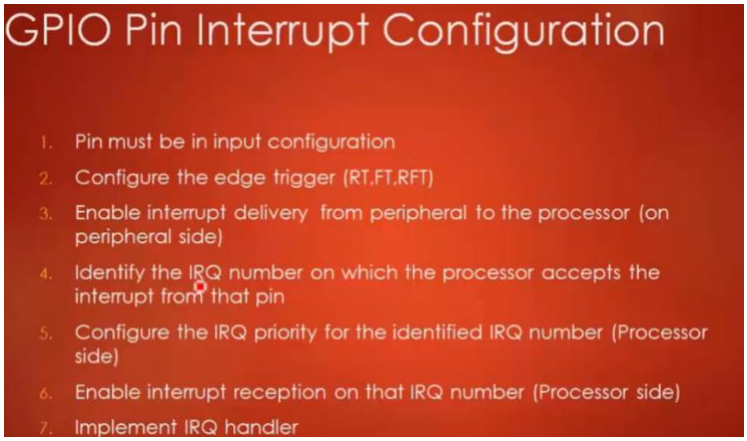
87.STM32 头文件

https://github.com/niekiran/MasteringMCU/blob/master/Resources/Source_code/Workspace/stm32f4xx_drivers/drivers/inc/stm32f407xx.h

108.GPIO pin interrupt configuration



NVIC 上的数字是实际中断时候的 IRQ number



第 2 条，必须配置 EXTI 的 RTSR 和 FTSR 这两个 register。之后必须使用 SYSCFG_EXTICR 决定 GPIO port

第 3 条，enable the exit interrupt delivery using IMR register(mask 指的是遮蔽，default 是遮蔽，要设置成不遮蔽的)，configure GPIO port selection in SYSCFG_EXTICR(P246)，第四步 enable ISER register

GPIO 的 IRQ handler 里面需要清除 EXTI 的 pending register

```
433 void GPIO_IRQHandling(uint8_t PinNumber)
434 {
435     //clear the exti pr register corresponding to the pin number
436     if(EXTI->PR & ( 1 << PinNumber))
437     {
438         //clear
439         EXTI->PR |= ( 1 << PinNumber);
440     }
441
442 }
```

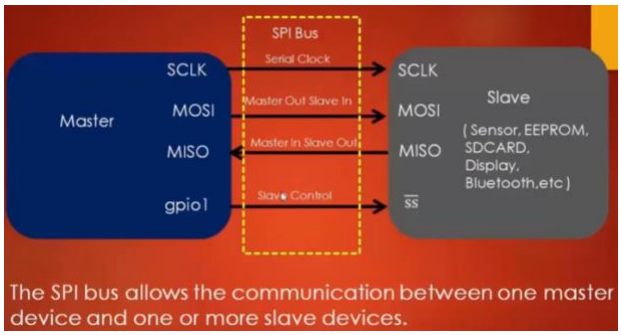
118. STM32 pin specification

V_{dd} and V_{ss}

- Vdd is the main power supply of the microcontroller. Provided externally through Vdd pins of the microcontroller
- Standard operating voltage of Vdd is: 1.8 V ≤ Vdd ≤ 3.6 V
- Max voltage which can be applied on any Vdd pin of the microcontroller is 4V (check AMR tables in the datasheet)
- Vss is a ground reference of the microcontroller and should be maintained at 0V
- Minimum value which can be applied to Vss is -0.3V(no less than this)

121.introduction to SPI Bus

SPI: serial peripheral interface 串行外围接口



1. Four I/O pins are dedicated to SPI communication with external devices.
2. MISO: Master In / Slave Out data. In the general case, this pin is used to transmit data in slave mode and receive data in master mode
3. MOSI: Master Out / Slave In data. In the general case, this pin is used to transmit data in master mode and receive data in slave mode.
4. SCK: Serial Clock output pin for SPI master and input pin for SPI slaves.
5. NSS: Slave select pin. Depending on the SPI and NSS settings, this pin can be used to select an individual slave device for communication

ss(slave select)引脚只有在有多个从机的时候才有意义，选择哪一个从机进行通信

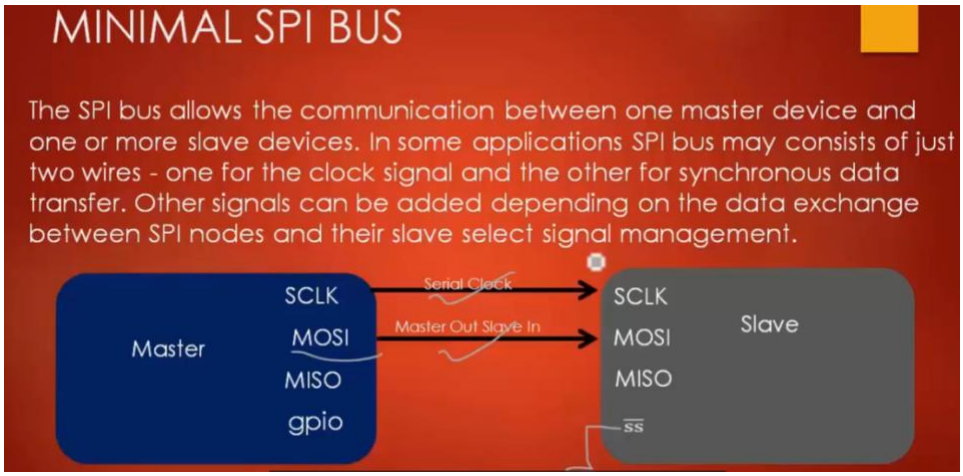
在选择之后，从机的 ss 会被拉至 0 或者接地，只有 ss 是 0 之后，才会激活数据通讯，之后为了发送数据，数据会和时钟一起发送。

122.SPI comparison

Protocol	Type	Max distance(ft.)	Max Speed (bps)	Typical usage
USB 3.0	dual simplex serial	9 (typical) (up to 49 with 5 hubs)	5 G	Mass storage, video
USB 2.0	half duplex serial	16 (98 ft. with 5 hubs)	1.5M, 12M, 480M	Keyboard, mouse, drive, speakers, printer, camera
Ethernet	serial	1600	10G	network communications
I2C	synchronous serial	18	3.4 M in High-speed mode.	Microcontroller communications
RS-232	asynchronous serial	50–100	20k	Modem, mouse, instrumentation
RS-485	asynchronous serial	4000	10M	Data acquisition and control systemsSPI
SPI	synchronous serial	10	fPCLK/2	

如果 SPI 的时钟是 50MHz，所以 max speed 是 25mbps

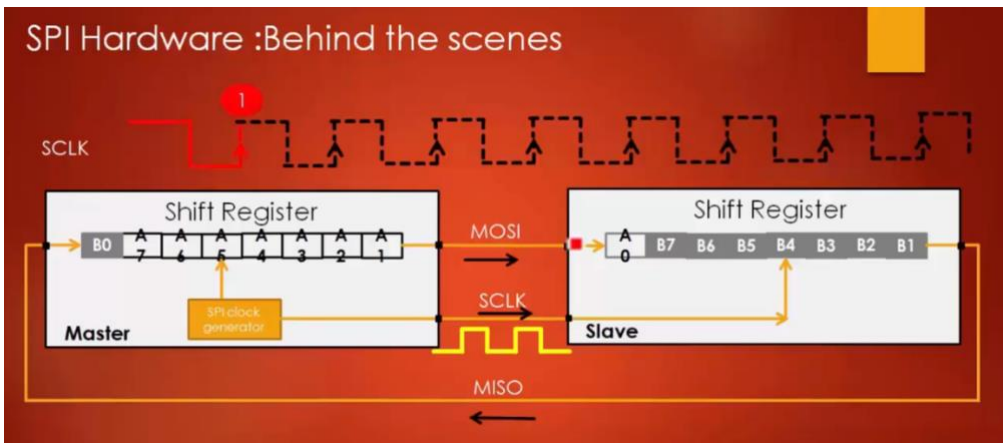
124.SPI minimum bus configuration



如果只有一个从机，而且他只接收数据的话，可以只用两条线(注意 ss 必须接地)

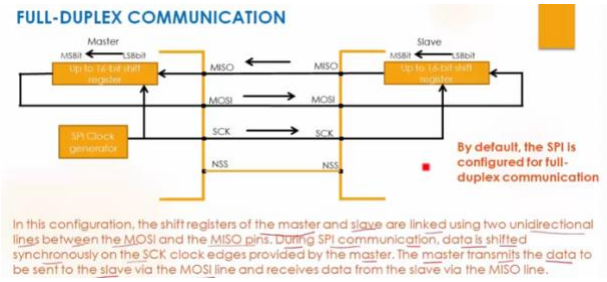
125.SPI principle

SPI 通讯使用了移位寄存器，每一个时钟周期移动一位，同步通讯，主机的 A0 通过 MOSI 移动到从机的同时，从机的 B0 也通过 MISO 传入到主机。8 个时钟周期以后，传输完毕。

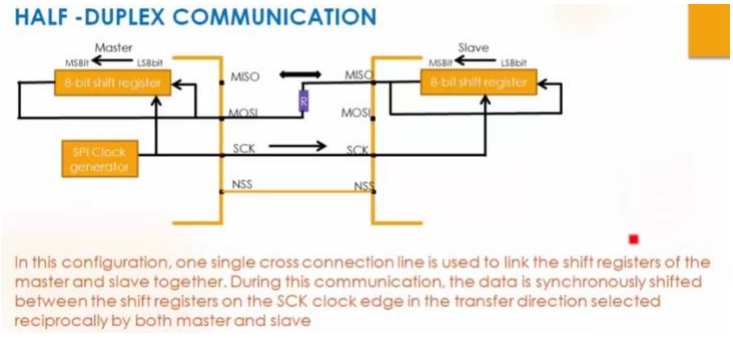


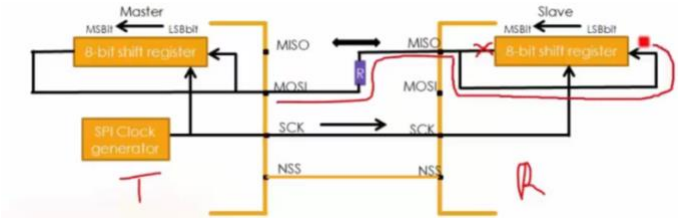
126.SPI bus configuration:

SPI 有三种通讯方式，第一种叫 full duplex，全双工通讯(by default)



第二种叫 half-duplex 半双工通讯

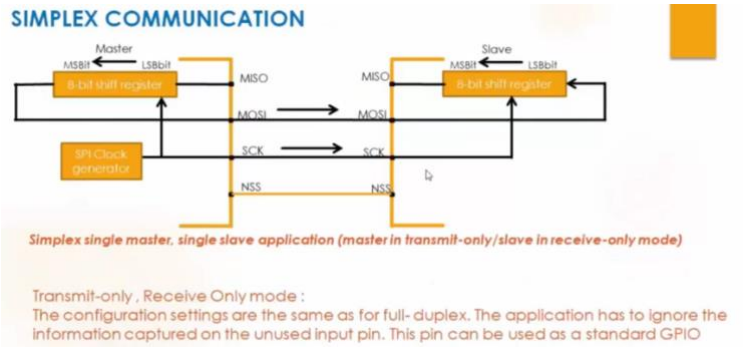




发送的时候，沿这条路线发送

当 master 想发送数据的时候，master 必须处于发送模式，slave 要处于接收模式，(可以在程序中进行配置)，也可以反过来，master 接收，slave 发送

第三种叫 simplex communication，单工通讯。Master 只能发送，slave 只能接收

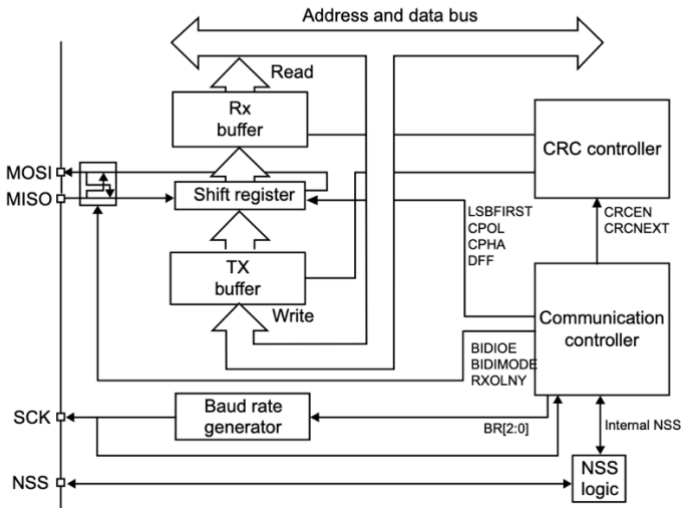


127.SPI functional block diagram

Reference manual 是第 848 页，这里的 address and data bus 就是 APB,读取数据的时候，实际是 shift register 把数据写入 Rx buffer 里，然后读的是 buffer 的数据。具体是收到一个完整的数据帧以后，数据就会移到 rx buffer，同时，用户收到中断(rx buffer 满)可以读入数据。写入的时候是往 tx buffer 里面写，然后 shift register 空闲后，就自动读入。那数据什么时候写入缓冲区?只要 tx 缓冲区为空。就会收到一个中断。TX_buffer 和 Rx_buffer 都实际与

SPI_data register 有关

Figure 303. SPI block diagram

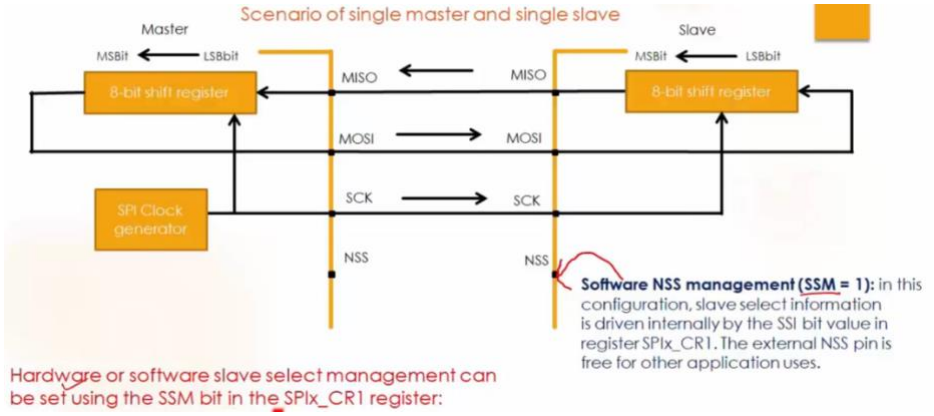


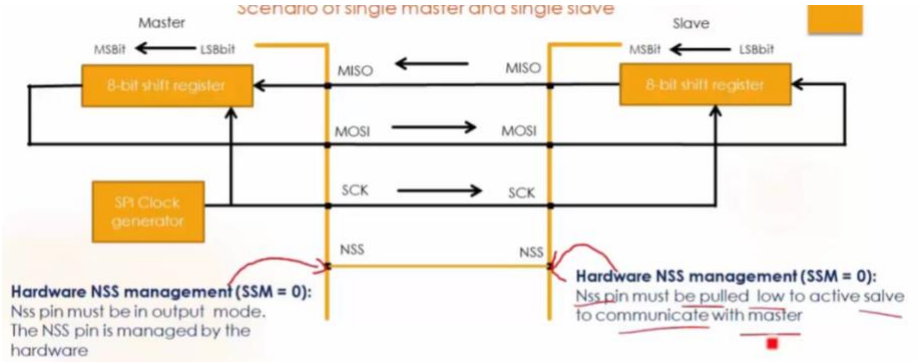
128. NSS settings in STM32 master and slave modes

当 device 是 master 的时候，NSS 只用于 output，本课程不讲 input （多主机模式），slave 的话，NSS 只有 input

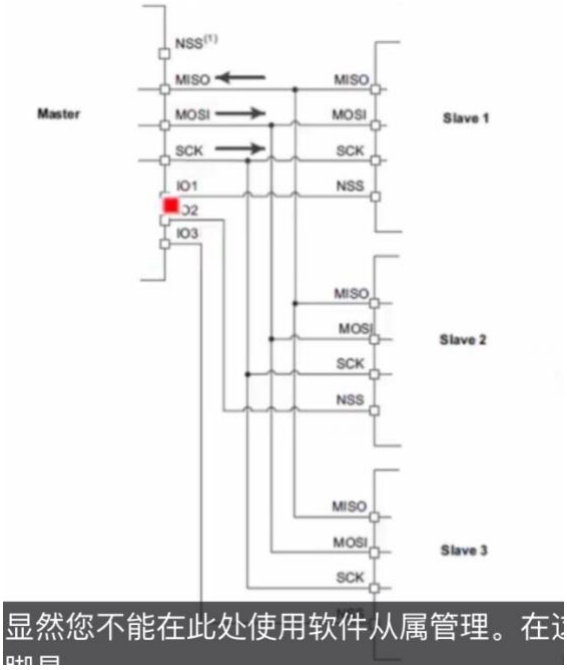
129.SPI hardware and software slave management

SPIx_CRI 当 SSM=1(选择软件从管理，这样的话，SSI 的值就会驱动这个引脚)而且 SSI=0(接地，如果 SSI 等于 1，那么 NSS 就是 1),那么 slave 就被内部接地，NSS 引脚就可以用作他用。当 SSM 为 0 的时候，SSI 的值没有意义，硬件管理，这样的话，NSS pin 必须接地。SSM=1 无需使用 NSS pin



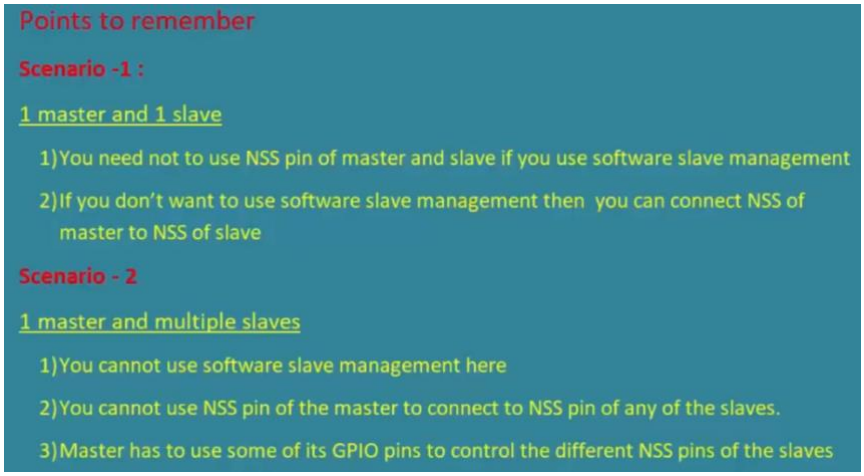


Hardware or software slave select management can be set using the SSM bit in the SPIx_CR1 register:

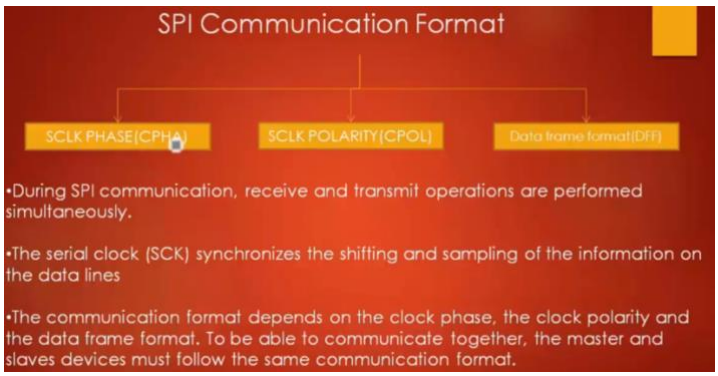


显然您不能在此处使用软件从属管理。在这

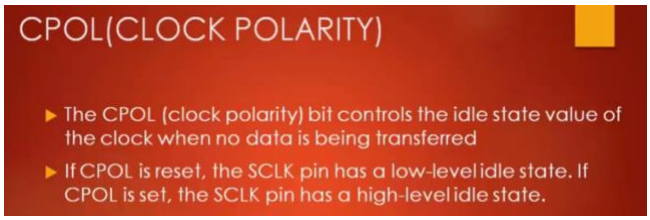
一主机多从机的话，只能使用硬件管理，因为要先确定和哪个从机通讯。同时主机的 NSS 不使用，而且必须接地。

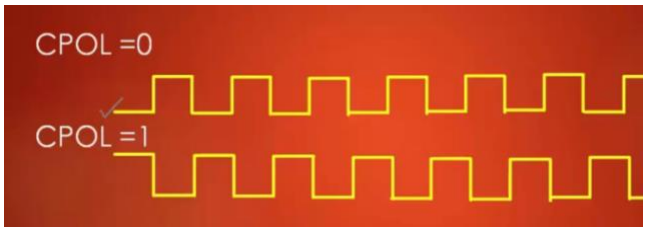


130.SPI CPOL and CPHA discussion

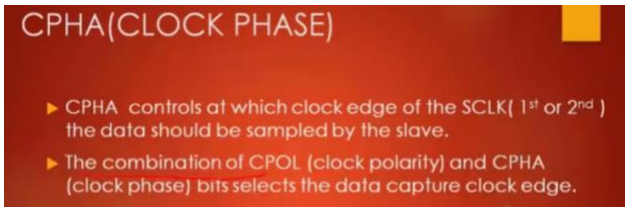


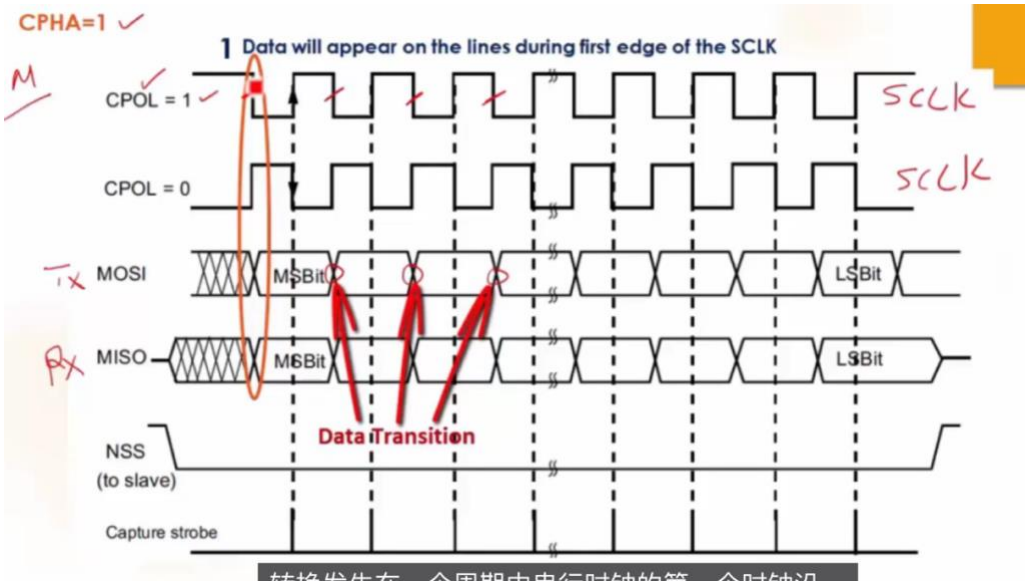
SCLK Phase：串行时钟相位，SCLK POLARITY:串行时钟极性，data frame format 数据帧格式





CPOL 就是决定用哪种时钟格式，为 0 时，如果没有通讯，那么为低电平



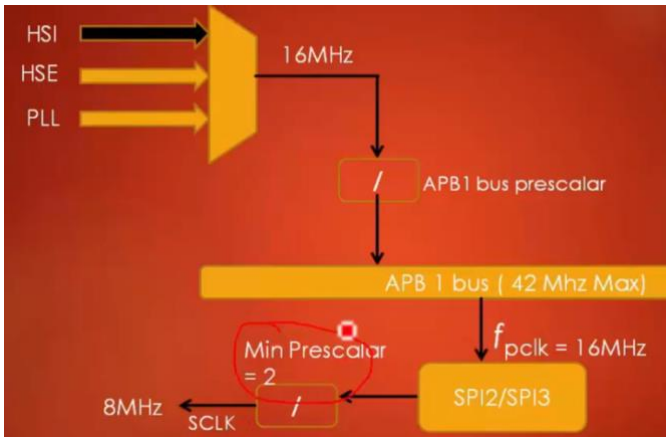


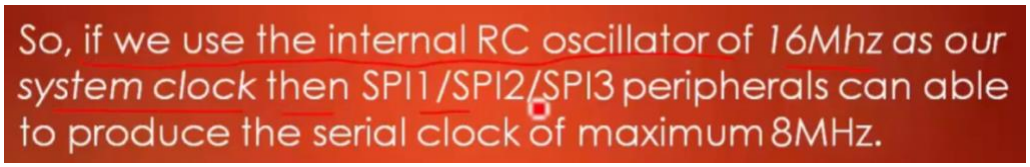
CPHA 就是决定什么时候传输数据，为 1，就在第一个沿传输数据，第二个沿 slave 和 master 采样数据(capture the data); CPHA 为 0 就是第二个沿传输数据，第一个沿采样数据

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

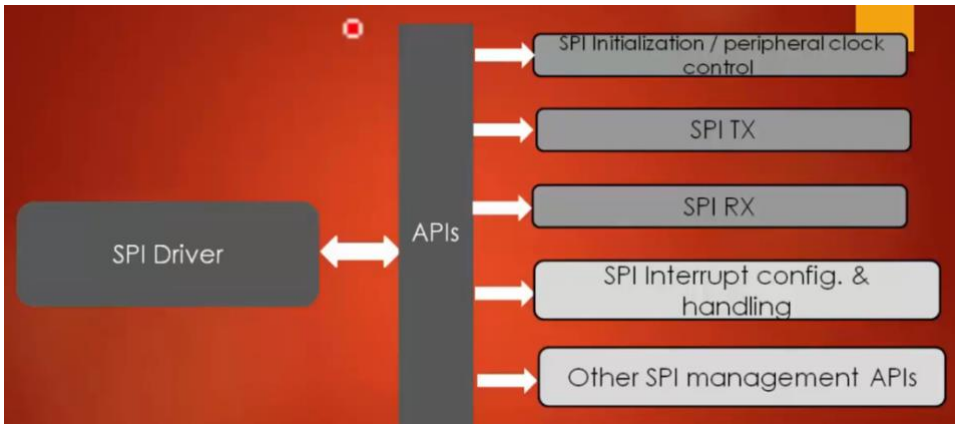
132.SPI peripheral of microcontroller

SPI1 是连接在 APB2 上，SPI2,3 连接在 APB1 上(8MHz,如果用 16MHz 的 HIS 的话)





134.SPI API requirements and configuration



SPI_deviceMode:配置的是 master 还是 slave，SPI_BusConfig 是配置的是全双工还是半双工，
SPI_DFF:data format 一次传送 8 位，还是 16 位数据
CPHA 和 CPOL 是配置时钟，SSM:从属管理，软件还是硬件。Speed 配置速度

代码部分

135.update MCU specific header file

```
 *  Configuration structure for SPIx peripheral
 */
typedef struct
{
    uint8_t SPI_DeviceMode;
    uint8_t SPI_BusConfig;
    uint8_t SPI_SclkSpeed;
    uint8_t SPI_DFF;
    uint8_t SPI_CPOL;
    uint8_t SPI_CPHA;
    uint8_t SPI_SSM;
}SPI_Config_t;


/*
 *Handle structure for SPIx peripheral
 */
typedef struct
{
    SPI_RegDef_t    *pSPIx;   /*!< This holds the base address of SPIx(x:0,1,2) peripheral >*/
    SPI_Config_t    SPIConfig;
}SPI_Handle_t;
```

SPI Configuration Structure

SPI Handle Structure

```
198 typedef struct
199 {
200     __vo uint32_t CR1;        /*!< TODO,
201     __vo uint32_t CR2;        /*!< TODO,
202     __vo uint32_t SR;         /*!< TODO,
203     __vo uint32_t DR;         /*!< TODO,
204     __vo uint32_t CRCPR;      /*!< TODO,
205     __vo uint32_t RXCRCR;     /*!< TODO,
206     __vo uint32_t TXCRCR;     /*!< TODO,
207     __vo uint32_t I2SCFGR;    /*!< TODO,
208     __vo uint32_t I2SPR;      /*!< TODO,
209 } SPI_RegDef_t;
```

同时还得定义宏包括 enable SPI1_PCLK_EN(),以及定义 IRQ 编号

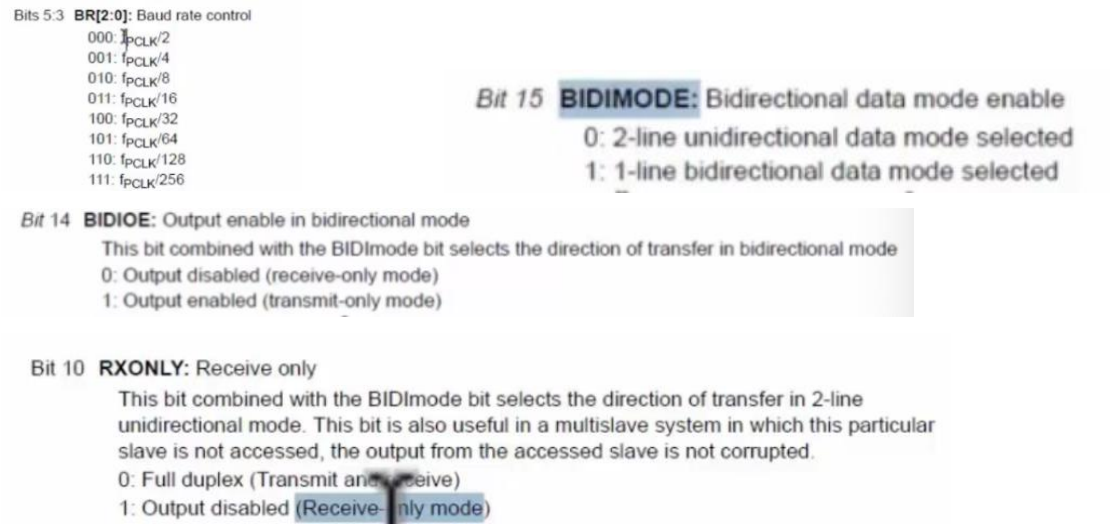
136.SPI adding API prototypes to driver header file
1.peripheral clock setup; 2.Init and De init; 3.Data send and Receive (block and unblock); 4. IRQ configuration and ISR handling

137.implementation of SPI peripheral clock control API
就是选择使用哪个 SPI1234 后，启用对应的时钟(先启用时钟，才能让 spi 工作)

```
24 void SPI_PeriClockControl(SPI_RegDef_t *pSPIx, uint8_t EnorDi)
25 {
26
27     if(EnorDi == ENABLE)
28     {
29         if(pSPIx == SPI1)
30         {
31             SPI1_PCLK_EN();
32         }else if (pSPIx == SPI2)
33         {
34             SPI2_PCLK_EN();
35         }else if (pSPIx == SPI3)
36         {
37             SPI3_PCLK_EN();
38         }else if (pSPIx == SPI4)
39         {
40             SPI4_PCLK_EN();
41         }
42     }
43     else
```

138.SPI user configuration options writing and register bit definition macros
为了配置 init 函数，需要首先定义宏，SPI_Device_mode 有 master 和 slave(在 SPI control register1 里配置)；SPI_BusConfig 有全双工，半双工， simplex_RX(也在 SPI CR1 里配置，半双工的话 BIDIMODE 为 1，全双工是 0，半双工的话 BIDIOE 必须一个 0 一个 1，全双工的话，必须配置 bit10，bit10 为 1 的话就是 simplex 模式，只接收)
Sclk_clock_speed(在 SPI control register1 里配置)，配置波特率(下图)，DFF 有 8bits 和 16bits(SPI_CR1 配置)， CPOL 和 CPHA 都有 low 和 high(SPI_CR1 配置)，SSM 分为软件控制和硬件控制。(SPI_CR1 配置)

Debug 知识点：device_mode 是 master 的话，会产生时钟，如果是 slave 的话，不会产生时钟



然后 init 函数就是配置这些参数，注意写代码的时候，移动多少位的多少位最好用宏代替，这样方便阅读

```
void SPI_Init(SPI_Handle_t *pSPIHandle)
{
```

141.implement of SPI send data API(阻塞式发送)
注意下图，有错误，16bit 的时候，一次要向 data register 发送 2byte(16bit)的数据,然后 len--两次

注意只有 Tx_buffer(TX_buffer 和 Rx_buffer 的写入或读取都是操作 SPI_data register)为空的时候才能写入数据，判断 buffer 是否为空，可以看 SPI_status register
当写入 SPI_data register 的时候，实际就是写入 TX_buffer,当读取 SPI_data register 的时候，就是读取 RX_buffer.不过在写入或读入之前，必须检查 status register 的 TXe 或者 RXne

```
void SPI_SendData(SPI_RegDef_t *pSPIx,uint8_t *pTxBuffer, uint32_t Len)
{
```

*pSPIx 是 SPI 的地址，pTxBuffer 是写入内容的首地址，len 是数据长度。注意定义的时候 pTxBuffer 是 8 位，所以传输 16 位的时候要长度-2，同时传的时候先变成 uint16_t,这样解引用就一下出 16bit 的数据

```
    //1. wait until TXE is set
    while(SPI_GetFlagStatus(pSPIx,SPI_TXE_FLAG)  == FLAG_RESET );

    //2. check the DFF bit in CR1
    if( (pSPIx->CR1 & ( 1 << SPI_CR1_DFF) ) )
    {
        //16 bit DFF
        //1. load the data in to the DR
        pSPIx->DR =   *((uint16_t*)pTxBuffer);
        Len--;
        Len--;
        (uint16_t*)pTxBuffer++;
    }else
    {
        //8 bit DFF
        pSPIx->DR =   *pTxBuffer;
        Len--;
        pTxBuffer++;
    }
```

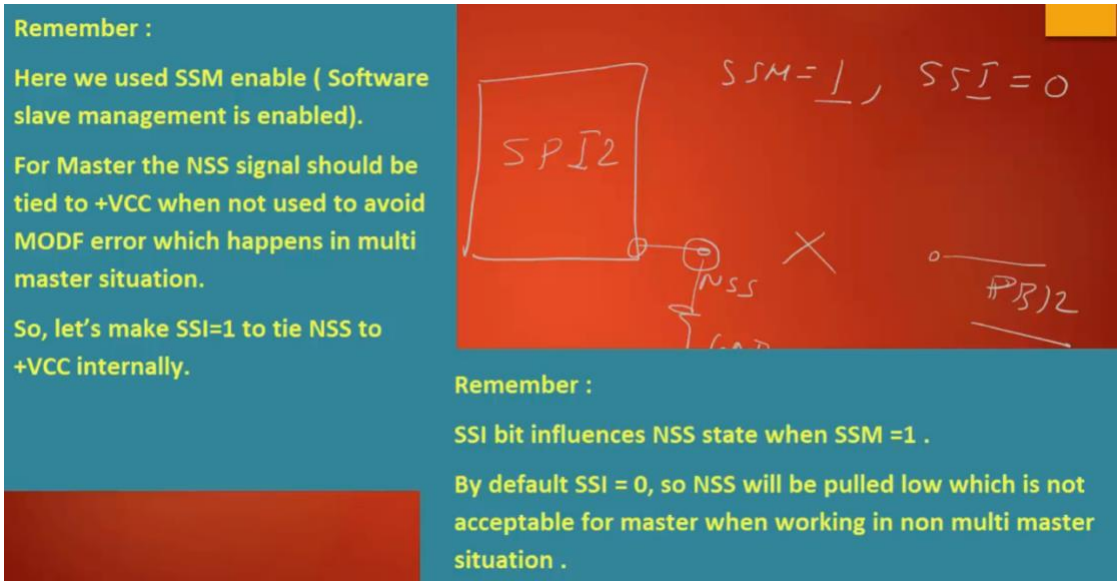
146.find out the PGIO pins over which SPI2 can communicate
首先必须找到 data sheet 去配置 GPIO 口，让他们处于 spi 的替代功能(alternate function mapping)

```
* PB14 --> SPI2_MISO
* PB15 --> SPI2_MOSI
* PB13 -> SPI2_SCLK
* PB12 --> SPI2_NSS
* ALT function mode : 5
```

Spi 都用 push-pull 结构，初始化这 4 个引脚

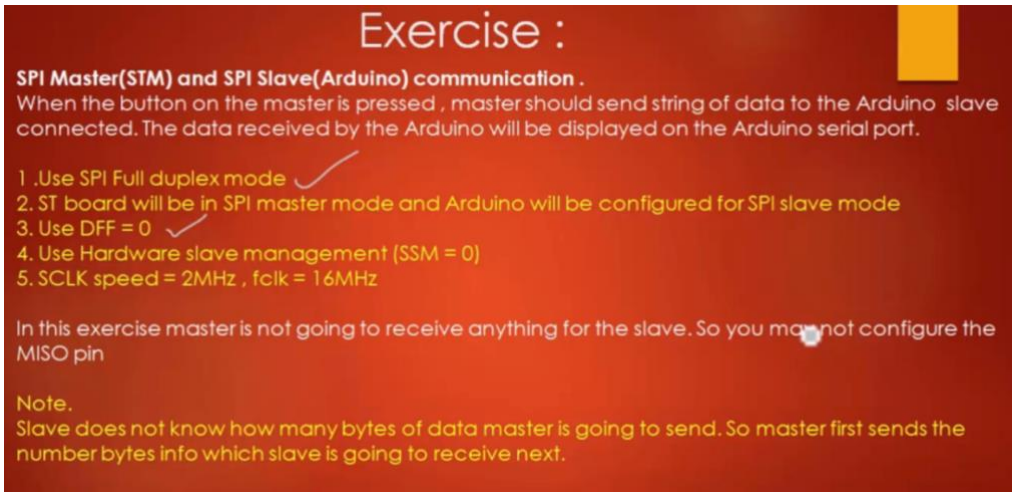
```
char user_data[] = "Hello world";

//this function is used to initialize the GPIO pins t
SPI2_GPIOInits();

//This function is used to initialize the SPI2 periph
SPI2_Inits();

SPI_SendData(SPI2,user_data,strlen(user_data));
```

发送数据的时候，长度直接用 strlen，user_data 要写前缀 unit8_t,
最后，在初始化结束之后，必须最后配置 spi_CR1 来 enable SPI(函数 SPI_PeripheralControl)
150.exercise

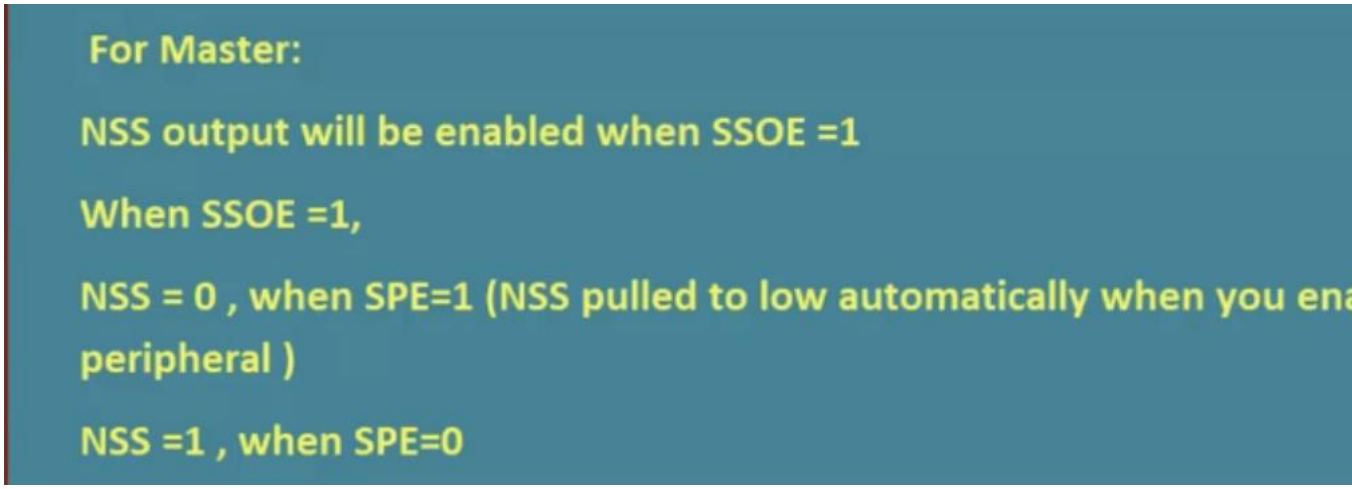



```
//Disable the SPI2 peripheral
SPI_PeripheralControl(SPI2,DISABLE);
```

Debug 注意，master 在不使用 NSS 的时候要把 NSS 置 1，然后在传输完数据之后，必须关闭 SPI 通讯

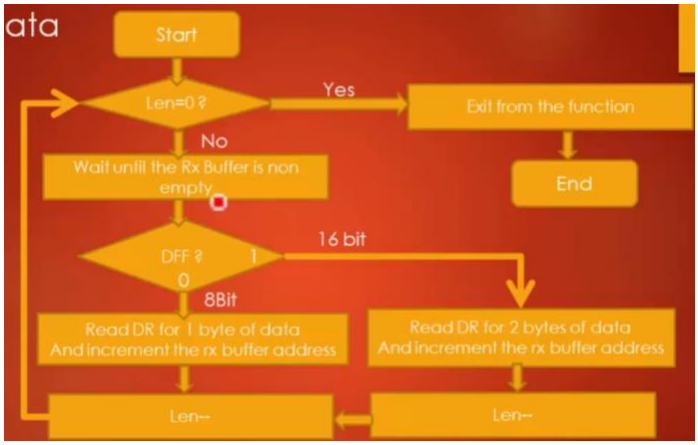


SSM 等于 0(硬件控制)的时候: 需要把 SSOE 设置为 1, SSOE 是 CR2 的 register 的位, NSS=0, 从机开始工作



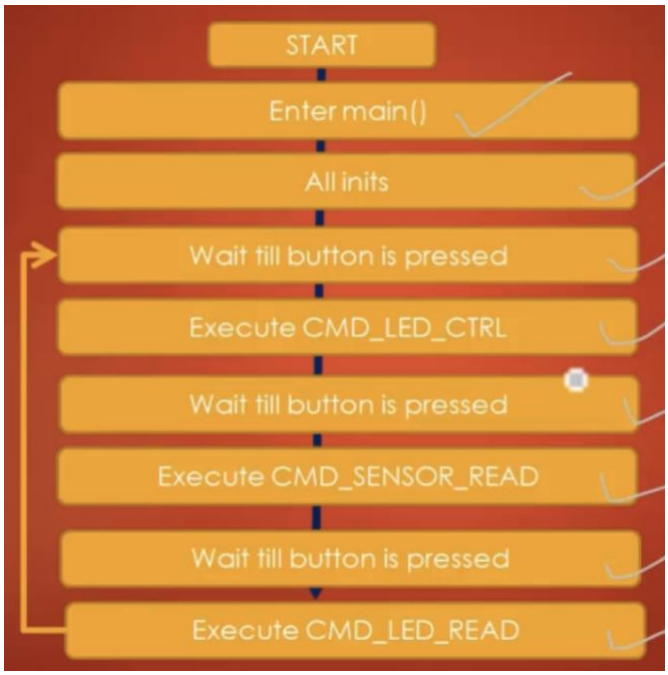
在关闭 spi 通讯之前，必须先去 status register 确定 spi 不忙才行（通讯完毕）

156.implementation of SPI data receive API

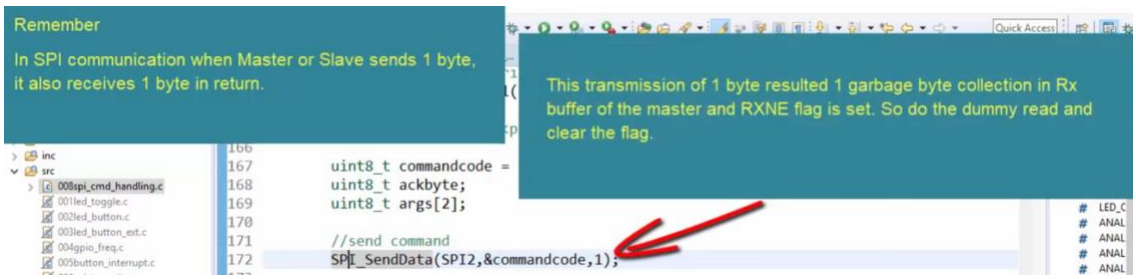


158.SPI command and response based communication(STM32 为 master，Arduino 为 slave)

STM32 向 Arduino 首先发送 command，然后，如果 arduino 支持这个 command，他就会返回 ACK



In SPI communication, when master or slave sends 1 byte, it also receives 1 byte in return.



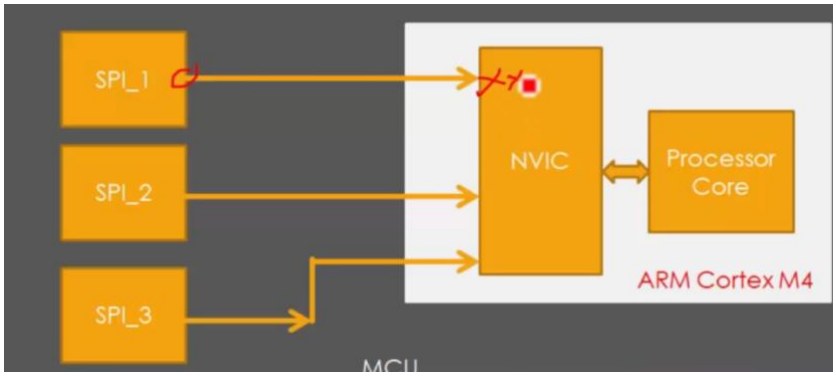
```
170                 //enable the SPI2 peripheral
171                 SPI_PeripheralControl(SPI2,ENABLE);
172
173             //1. CMD_LED_CTRL   <pin no(1)>     <value(1)>
174
175                 uint8_t commandcode = COMMAND_LED_CTRL;
176                 uint8_t ackbyte;
177                 uint8_t args[2];
178
179                 //send command
180                 SPI_SendData(SPI2,&commandcode,1);
181
182                 //do dummy read to clear off the RXNE
183                 SPI_ReceiveData(SPI2,&dummy_read,1);
184
185
186                 //Send some dummy bits (1 byte) fetch the response from the slave
187                 SPI_SendData(SPI2,&dummy_write,1);
188
189                 //read the ack byte received
190                 SPI_ReceiveData(SPI2,&ackbyte,1);
191
192                 if( SPI_VerifyResponse(ackbyte))
193                 {
194                         args[0] = LED_PIN;
195                         args[1] = LED_ON;
196
197                         //send arguments
198                         SPI_SendData(SPI2,args,2);
199                         // dummy read
200                         SPI_ReceiveData(SPI2,args,2);
201                         printf("COMMAND_LED_CTRL Executed\n");
202                 }
203                 //end of COMMAND_LED_CTRL
```

这里第 180 行，stm32 先发送命令，但是 SPI 通讯里，每发送一个 byte 的数据，就会收到一个 byte 的数据(spi 规定)，所以会收到一个 byte 的 data，第 183 行要先读取这一个 byte 的 dummy data。然后 arduino 收到命令后，会发送 ACK 命令，但是 slave 不会主动发送，这个 ack 命令会存在移位寄存器里，为了获得这个返回命令，stm32 需要在发送一个 byte 的 dummy 命令，之后就会读取这一个 byte 的返回命令。然后，在发送命令行的参数。
就是说如果 master 想发送数据，在发送数据之后必须进行一次 dummy read，如果 master 想接收数据，必须先进行一次 dummy write。在读取数据之前，也可以加入一个小的 delay，让 slave 能够准备好数据

162.SPI interrupting the processor

SPI interrupt requests

Interrupt event	Event flag	Enable Control bit
Transmit Tx buffer ready to be loaded	TXE	TXEIE
Data received in Rx buffer	RXNE	RXNEIE
Master Mode fault event	MODF	ERRIE
Overrun error	OVR	
CRC error	CRCERR	
TI frame format error	FRE	

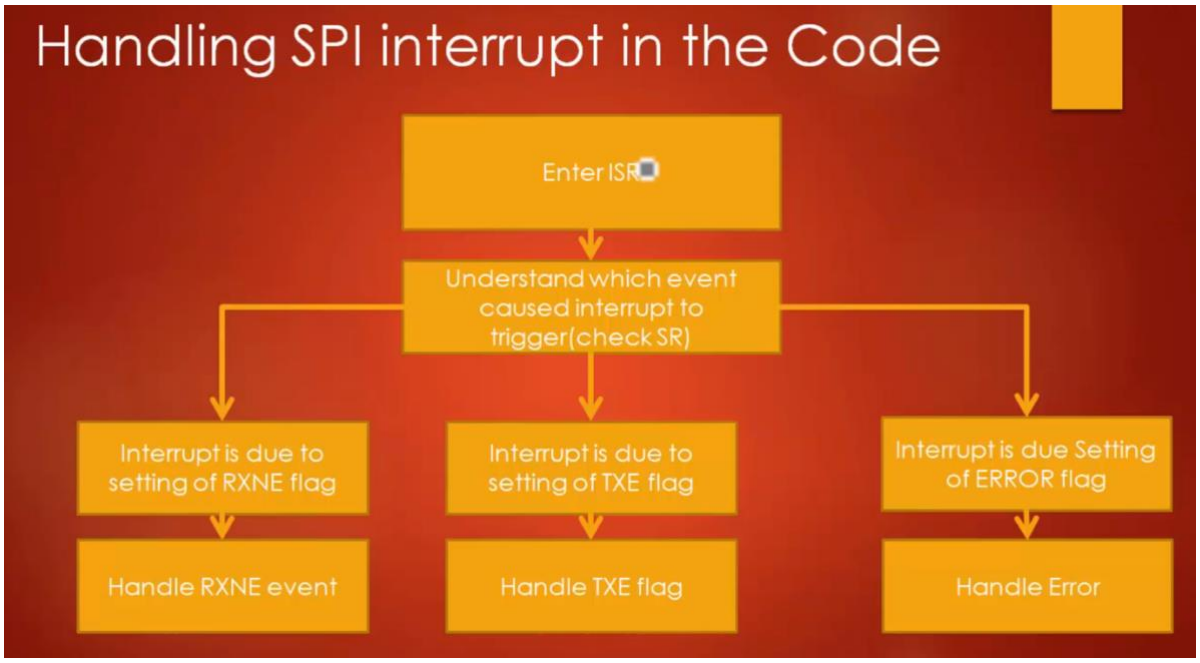


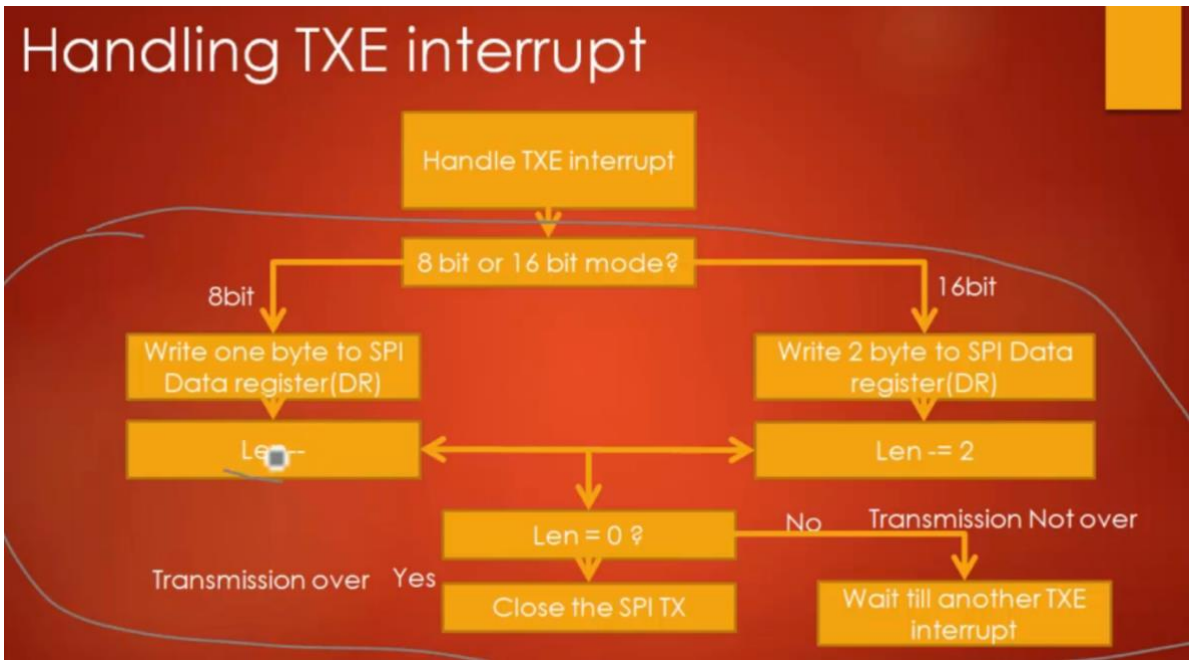

```
 * IRQ Configuration and ISR handling
 */
void SPI_IRQInterruptConfig(uint8_t IRQNumber, uint8_t EnorDi);
void SPI_IRQPriorityConfig(uint8_t IRQNumber, uint32_t IRQPriority);
void SPI_IRQHandling(SPI_Handle_t *pHandle);
```

API to send data with interrupt mode

```
uint8_t SPI_SendDataWithIT(SPI_Handle_t *pSPIHandle,uint8_t *pTxBuffer, uint8_t Len)
{

    //1 . Save the Tx buffer address and Len information in some global variables

    //2.  Mark the SPI state as busy in transmission so that
    //    no other code can take over same SPI peripheral until transmission is over

    //3. Enable the TXEIE control bit to get interrupt whenever TXE flag is set in SR

    //4. Data Transmission will be handled by the ISR code ( will implement later)
```

TXEIE 在 SPI_CR2 里





每个中断传输一个字节，close SPI TX 就是把 TXEIE 位清 0，然后把 TX buffer 变 null， txlen=0，TXstate 变 ready。