5.Communication using SPI

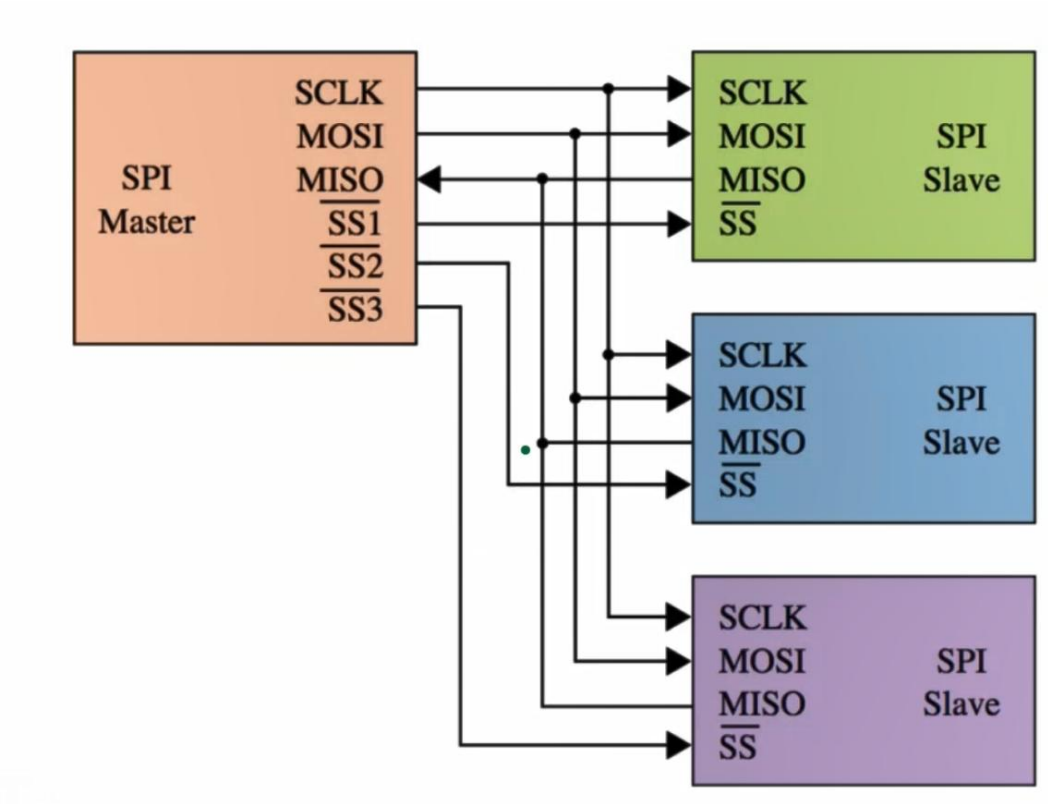
5.1.Lý thuyết chung

SPI dùng 4 chân để giao tiếp giữa 2 thiết bị.
MASTER có thể giao tiếp với nhiều thiết bị SLAVE
Điều kiện để thực hiện giao tiếp được:

- Chân SS được kéo xuống 0
- Sau đó master phát xung SCK đồng bộ truyền nhận, lập trình truyền ở cạnh lên hoặc cạnh xuống của xung.
- Master truyền Data đi trên chân chân MOSI (Master out Slave in)
- Master nhận data phản hồi trên chân MISO (Master in, Slave out)

Trạng thái IDLE : chân SS = 0 và chân SCK =0.
Các kiểu truyền dữ liệu :

- Song công, data truyền 2 chiều trên đường dây MOSI va MISO bởi cả master và slave
- Bán song công, chỉ truyền từ master tới slave



Đối với MASTER :
SCK, MOSI, CS là Output => Cấu hình Mode output kiểu Push Pull
MISO là input => Chọn Mode input kiểu In-floating (điện áp thả nổi). thường đường đây được kéo lên mức 3v hay 5v bởi điện trở kéo để clear mức 1.
Với SLAVE ngược lại :
SCK, MOSI, CS là Input => Chọn Mode input : In-Floating

MISO là output => Mode output, kiểu Push Pull
Hàm Truyền dữ liệu
Hàm truyền dữ liệu sẽ lần lượt truyền 8 bit trong byte dữ liệu

- Kéo CS = 0.
- Kiểm tra Clock() = 1 ??
- Chỉ khi CS = 0 & SCK =1 thì đọc data trên chân MOSI ghi vào biến
- Dịch 1 bit
- Kiểm tra CS = 1 => dừng đọc ghi

Hàm nhận dữ liệu
Hàm nhận dữ liệu sẽ lần lượt nhận 8 bit dữ liệu từ hàm truyền

- Kiểm tra CS = 0 ?? .
- Kiểm tra Clock() = 1 ??
- Chỉ khi CS = 0 & SCK =1 thì đọc data trên chân MOSI ghi vào biến
- Dịch 1 bit
- Kiểm tra CS = 1 => dừng đọc ghi

5.2.Viết chương trình truyền dữ liệu giữa 2 MCU stm32, truyền mảng giữ liệu datatrans[]={3, 6, 9, 369, 999}

Phần cứng và thư viện cần dùng gồm

- SPI (truyền nhận dữ liệu)
- GPIO (config 4 chân cho SPI). Note: GPIO là cổng giao tiếp ra thế giới bên ngoài duy nhất của STM32. Sử dụng GPIO A thì cấp Clock qua bus APB2Periph
- TIMER (để tạo hàm delay và xung đồng bộ clock())

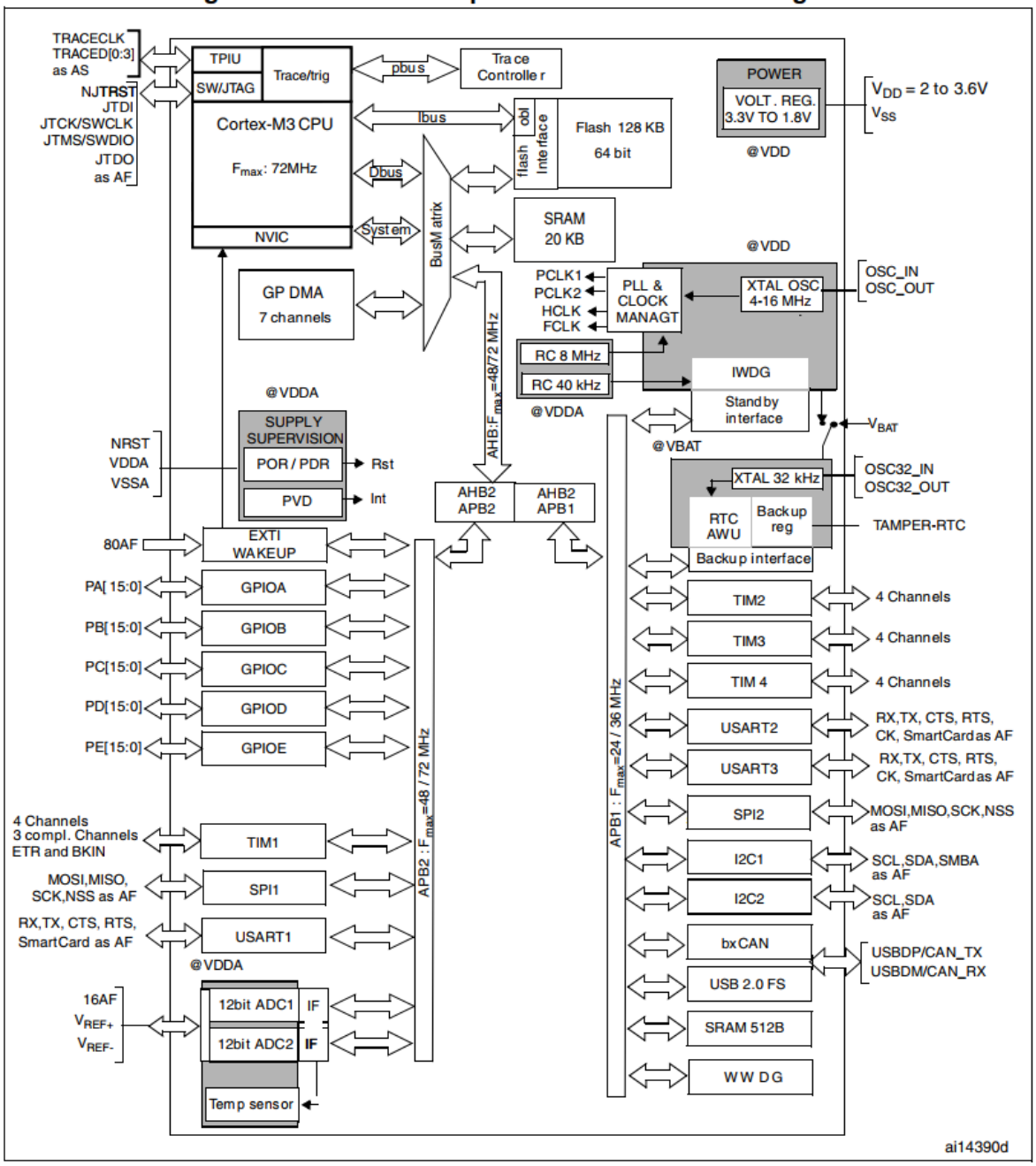
Có thể viết bằng Hardware và Software.
Các hàm đã có sẵn trong thư viện của vi điều khiển, để viết chương trình cần đi từ
Mỗi loại ngoại vi được định nghĩa thành 1 struct, nên search từ khóa “Typedef struct” để tìm hiểu cách tổ chức của các truct GPIO, TIMER. Truyền tham số và set thông số cho struct.

5.3. Lập trình truyền dữ liệu bằng HARDWARE

STM32 có 2 phần cứng SPI: SPI1 và SPI2

- SPI1 trên bus APB2, SPI 2 trên bus APB1.
- Chân MOSI, MISO, SCK, NSS hoạt đông ở mode AF (Alternative Function)

Figure 1. STM32F103xx performance line block diagram



Kiểm tra trong “pin definitions” table .Khi cấp clock cho SPI thì 4 chân đầu ra sẽ được kết nối mặc định đến các chân nào. như bảng dưới là NSS, SCK, MISO, MOSI lần lượt nối ra PA4,5,6,7 của GPIOA.

E4	E3	-	C2	18	27	-	V_{SS_4}	S	-	V_{SS_4}	-	-
F4	H3	-	D2	19	28	-	V_{DD_4}	S	-	V_{DD_4}	-	-
G3	M3	14	H3	20	29	11	PA4	I/O	-	PA4	SPI1_NSS(9)/ USART2_CK(9)/ ADC12_IN4	-
H3	K4	15	F4	21	30	12	PA5	I/O	-	PA5	SPI1_SCK(9)/ ADC12_IN5	-
J3	L4	16	G4	22	31	13	PA6	I/O	-	PA6	SPI1_MISO(9)/ ADC12_IN6/ TIM3_CH1(9)	TIM1_BKIN
K3	M4	17	H4	23	32	14	PA7	I/O	-	PA7	SPI1_MOSI(9)/ ADC12_IN7/ TIM3_CH2(9)	TIM1_CH1N
G4	K5	-	H5	24	33		PC4	I/O	-	PC4	ADC12_IN14	-

Nếu bật cùng lúc nhiều ngoại vi như ADC, SPI, USART cần remap lại các chân thì ta dùng thanh ghi AFIO_MAPR.

Ví dụ set bit 0 của thanh ghi lên 1 thì ta đã remap chân của SPI 1

0: No remap (NSS/PA4, SCK/PA5, MISO/PA6, MOSI/PA7)

1: Remap (NSS/PA15, SCK/PB3, MISO/PB4, MOSI/PB5)

9.4.2 AF remap and debug I/O configuration register (AFIO_MAPR)

Address offset: 0x04

Reset value: 0x0000 0000

Memory map and bit definitions for low-, medium- high- and XL-density devices:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					SWJ_CFG[2:0]			Reserved			ADC2_ETRGREG_REMAP	ADC2_ETRGINJ_REMAP	ADC1_ETRGREG_REMAP	ADC1_ETRGINJ_REMAP	TIM5CH4_IREMAP
					w	w	w				rw	rw	rw	rw	rw
PD01_REMAP	CAN_REMAP[1:0]		TIM4_REMAP	TIM3_REMAP[1:0]		TIM2_REMAP[1:0]		TIM1_REMAP[1:0]		USART3_REMAP[1:0]		USART2_REMAP	USART1_REMAP	I2C1_REMAP	SPI1_REMAP
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 3 **USART2_REMAP:** USART2 remapping

This bit is set and cleared by software. It controls the mapping of USART2 CTS, RTS,CK,TX and RX alternate functions on the GPIO ports.
0: No remap (CTS/PA0, RTS/PA1, TX/PA2, RX/PA3, CK/PA4)
1: Remap (CTS/PD3, RTS/PD4, TX/PD5, RX/PD6, CK/PD7)

Bit 2 **USART1_REMAP:** USART1 remapping

This bit is set and cleared by software. It controls the mapping of USART1 TX and RX alternate functions on the GPIO ports.
0: No remap (TX/PA9, RX/PA10)
1: Remap (TX/PB6, RX/PB7)

Bit 1 **I2C1_REMAP:** I2C1 remapping

This bit is set and cleared by software. It controls the mapping of I2C1 SCL and SDA alternate functions on the GPIO ports.
0: No remap (SCL/PB6, SDA/PB7)
1: Remap (SCL/PB8, SDA/PB9)

Bit 0 **SPI1_REMAP:** SPI1 remapping

This bit is set and cleared by software. It controls the mapping of SPI1 NSS, SCK, MISO, MOSI alternate functions on the GPIO ports.
0: No remap (NSS/PA4, SCK/PA5, MISO/PA6, MOSI/PA7)
1: Remap (NSS/PA15, SCK/PB3, MISO/PB4, MOSI/PB5)

Tương tự các ngoại vi khác, các tham số SPI được cấu hình trong **Struct SPI_InitTypeDef:**

spi_master_hw.c | stm32f10x_rcc.c | stm32f10x_rcc.h | stm32f10x_gpio.c | stm32f10x_gpio.h | stm32f10x_spi.c | stm32f10x_spi.h

```
45 /**
46   * @brief  SPI Init structure definition
47   */
48
49 typedef struct
50 {
51   uint16_t SPI_Direction;           /*!< Specifies the SPI unidirectional or bidirectional data mode.
52                                         This parameter can be a value of @ref SPI_data_direction */
53
54   uint16_t SPI_Mode;                /*!< Specifies the SPI operating mode.
55                                         This parameter can be a value of @ref SPI_mode */
56
57   uint16_t SPI_DataSize;            /*!< Specifies the SPI data size.
58                                         This parameter can be a value of @ref SPI_data_size */
59
60   uint16_t SPI_CPOL;                /*!< Specifies the serial clock steady state.
61                                         This parameter can be a value of @ref SPI_Clock_Polarity */
62
63   uint16_t SPI_CPHA;                /*!< Specifies the clock active edge for the bit capture.
64                                         This parameter can be a value of @ref SPI_Clock_Phase */
65
66   uint16_t SPI_NSS;                 /*!< Specifies whether the NSS signal is managed by
67                                         hardware (NSS pin) or by software using the SSI bit.
68                                         This parameter can be a value of @ref SPI_Slave_Select_management */
69
70   uint16_t SPI_BaudRatePrescaler;   /*!< Specifies the Baud Rate prescaler value which will be
71                                         used to configure the transmit and receive SCK clock.
72                                         This parameter can be a value of @ref SPI_BaudRate_Prescaler.
73                                         @note The communication clock is derived from the master
74                                               clock. The slave clock does not need to be set. */
75
76   uint16_t SPI_FirstBit;            /*!< Specifies whether data transfers start from MSB or LSB bit.
77                                         This parameter can be a value of @ref SPI_MSB_LSB_transmission */
78
79   uint16_t SPI_CRCPolynomial;       /*!< Specifies the polynomial used for the CRC calculation. */
80 }SPI_InitTypeDef;
81
82 /**
83   * @brief  I2S Init structure definition
```

- SPI_Mode: Quy định chế độ hoạt động của thiết bị SPI (Master or Slave)

```
#define SPI_Mode_Master                  ((uint16_t)0x0104)
#define SPI_Mode_Slave                   ((uint16_t)0x0000)
#define IS_SPI_MODE(MODE) (((MODE) == SPI_Mode_Master) || \
                           ((MODE) == SPI_Mode_Slave))
```

- SPI_Direction: Quy định kiểu truyền của thiết bị (FullDuplex, RxOnly,RX,TX)

```
127 #define SPI_Direction_2Lines_FullDuplex ((uint16_t)0x0000)
128 #define SPI_Direction_2Lines_RxOnly     ((uint16_t)0x0400)
129 #define SPI_Direction_1Line_Rx          ((uint16_t)0x8000)
130 #define SPI_Direction_1Line_Tx          ((uint16_t)0xC000)
131 #define IS_SPI_DIRECTION_MODE(MODE) (((MODE) == SPI_Direction_2Lines_FullDuplex) || \
132                                      ((MODE) == SPI_Direction_2Lines_RxOnly) || \
133                                      ((MODE) == SPI_Direction_1Line_Rx) || \
134                                      ((MODE) == SPI_Direction_1Line_Tx))
```

- SPI_BaudRatePrescaler: Hệ số chia clock cấp cho Module SPI (2,4,8,16,…,256-chia hệ số theo hàm mũ của 2)

```
203 #define SPI_BaudRatePrescaler_2         ((uint16_t)0x0000)
204 #define SPI_BaudRatePrescaler_4         ((uint16_t)0x0008)
205 #define SPI_BaudRatePrescaler_8         ((uint16_t)0x0010)
206 #define SPI_BaudRatePrescaler_16        ((uint16_t)0x0018)
207 #define SPI_BaudRatePrescaler_32        ((uint16_t)0x0020)
208 #define SPI_BaudRatePrescaler_64        ((uint16_t)0x0028)
209 #define SPI_BaudRatePrescaler_128       ((uint16_t)0x0030)
210 #define SPI_BaudRatePrescaler_256       ((uint16_t)0x0038)
```

- SPI_CPOL: Cấu hình cực tính (Polary) của SCK . Có 2 chế độ:
- SPI_CPOL_Low: Cực tính mức 0 khi SCK không truyền xung.
- SPI_CPOL_High: Cực tính mức 1 khi SCK không truyền xung.

(Thường chọn mức CPOL mức 0)

```
167 #define SPI_CPOL_Low                    ((uint16_t)0x0000)
168 #define SPI_CPOL_High                   ((uint16_t)0x0002)
169 #define IS_SPI_CPOL(CPOL) (((CPOL) == SPI_CPOL_Low) || \
170                            ((CPOL) == SPI_CPOL_High))
```

- SPI_CPHA: Cấu hình hoạt động ở pha (phase) nào của SCK. Có 2 chế độ:
- SPI_CPHA_1Edge: Tín hiệu truyền đi ở cạnh xung đầu tiên (pha 1 lên).
- SPI_CPHA_2Edge: Tín hiệu truyền đi ở cạnh xung thứ hai (pha 2 xuống)

```
75 /** @defgroup SPI_Clock_Phase
76   * @{
77   */
78
79 #define SPI_CPHA_1Edge                  ((uint16_t)0x0000)
80 #define SPI_CPHA_2Edge                  ((uint16_t)0x0001)
81 #define IS_SPI_CPHA(CPHA) (((CPHA) == SPI_CPHA_1Edge) || \
82                            ((CPHA) == SPI_CPHA_2Edge))
```

- SPI_DataSize: Cấu hình số bit truyền. 8 hoặc 16 bit.

```
155 #define SPI_DataSize_16b                ((uint16_t)0x0800)
156 #define SPI_DataSize_8b                 ((uint16_t)0x0000)
157 #define IS_SPI_DATASIZE(DATASIZE) (((DATASIZE) == SPI_DataSize_16b) || \
158                                    ((DATASIZE) == SPI_DataSize_8b))
```

- SPI_FirstBit: Cấu hình **chiều truyền** của các bit là MSB hay LSB.

Ví dụ truyền data 8 bit data = 0x80 = 0b1000 0000. MSB = Most Significant Bit = 1 : Bit có trọng số cao nhất(bít đầu bên trái) , LSB = Least Significant bit = 0 (bit có trọng số thấp nhất (bit đầu bên phải).

```
/** @defgroup SPI_MSB_LSB_transmission
  * @{
  */

#define SPI_FirstBit_MSB                 ((uint16_t)0x0000)
#define SPI_FirstBit_LSB                 ((uint16_t)0x0080)
#define IS_SPI_FIRST_BIT(BIT) (((BIT) == SPI_FirstBit_MSB) || \
                               ((BIT) == SPI_FirstBit_LSB))
```

- SPI_CRCPolynomial: Cấu hình số bit CheckSum cho SPI.

```
/** @defgroup SPI_CRC_polynomial
  * @{
  */

#define IS_SPI_CRC_POLYNOMIAL(POLYNOMIAL) ((POLYNOMIAL) >= 0x1)
```

- SPI_NSS: Cấu hình chân SS là điều khiển bằng thiết bị hay phần mềm.

```
#define SPI_NSSInternalSoft_Set          ((uint16_t)0x0100)
#define SPI_NSSInternalSoft_Reset        ((uint16_t)0xFEFF)
#define IS_SPI_NSS_INTERNAL(INTERNAL) (((INTERNAL) == SPI_NSSInternalSoft_Set) || \
                                       ((INTERNAL) == SPI_NSSInternalSoft_Reset))
```

- Hàm SPI_I2S_SendData(SPI_TypeDef* SPIx, uint16_t Data), tùy vào cấu hình datasize là 8 hay 16 bit sẽ truyền đi 8 hoặc 16 bit dữ liệu. Hàm nhận 2 tham số là bộ SPI sử dụng và data cần truyền.
  Bản chất việc truyền là chính là ghi Data vào thanh ghi DR

```
void SPI_I2S_SendData(SPI_TypeDef* SPIx, uint16_t Data)
{
  /* Check the parameters */
  assert_param(IS_SPI_ALL_PERIPH(SPIx));

  /* Write in the DR register the data to be sent */
  SPIx->DR = Data;
}
```

- Hàm SPI_I2S_ReceiveData(SPI_TypeDef* SPIx) trả về giá trị đọc được trên SPIx. Hàm trả về 8 hoặc 16 bit data
  Bản chất hàm nhận là đọc Data từ thanh ghi DR ra.

```
uint16_t SPI_I2S_ReceiveData(SPI_TypeDef* SPIx)
{
  /* Check the parameters */
  assert_param(IS_SPI_ALL_PERIPH(SPIx));

  /* Return the data in the DR register */
  return SPIx->DR;
}
```

- Hàm SPI_I2S_GetFlagStatus(SPI_TypeDef* SPIx, uint16_t SPI_I2S_FLAG) trả về giá trị 1 cờ trạng thái trong thanh ghi của SPI.
  Trong phần cứng SPI có 1 bộ đệm Transmit Buffer chứa data trước khi truyền đi chính là thanh ghi DR, nếu dữ liệu chưa truyền đi mà ta ghi data mới vào thì data cũ sẽ bị mất đi, bởi vậy phải kiểm tra cờ trạng thái xem thanh ghi DR còn trống không trước khi truyền. Các cờ thường được dùng:

- SPI_I2S_FLAG_TXE: Cờ báo truyền, cờ này sẽ set lên 1 khi truyền xong data trong buffer.
- SPI_I2S_FLAG_RXNE: Cờ báo nhận, cờ này set lên 1 khi nhận xong data.
- SPI_I2S_FLAG_BSY: Cờ báo bận,set lên 1 khi SPI đang bận truyền nhận.

```
748 /**
749   * @brief  Checks whether the specified SPI/I2S flag is set or not.
750   * @param  SPIx: where x can be
751   *   - 1, 2 or 3 in SPI mode
752   *   - 2 or 3 in I2S mode
753   * @param  SPI_I2S_FLAG: specifies the SPI/I2S flag to check.
754   *   This parameter can be one of the following values:
755   *     @arg SPI_I2S_FLAG_TXE: Transmit buffer empty flag.
756   *     @arg SPI_I2S_FLAG_RXNE: Receive buffer not empty flag.
757   *     @arg SPI_I2S_FLAG_BSY: Busy flag.
758   *     @arg SPI_I2S_FLAG_OVR: Overrun flag.
759   *     @arg SPI_FLAG_MODF: Mode Fault flag.
760   *     @arg SPI_FLAG_CRCERR: CRC Error flag.
761   *     @arg I2S_FLAG_UDR: Underrun Error flag.
762   *     @arg I2S_FLAG_CHSIDE: Channel Side flag.
763   * @retval The new state of SPI_I2S_FLAG (SET or RESET).
764   */
765 FlagStatus SPI_I2S_GetFlagStatus(SPI_TypeDef* SPIx, uint16_t SPI_I2S_FLAG)
766 {
767   FlagStatus bitstatus = RESET;
768   /* Check the parameters */
769   assert_param(IS_SPI_ALL_PERIPH(SPIx));
770   assert_param(IS_SPI_I2S_GET_FLAG(SPI_I2S_FLAG));
771   /* Check the status of the specified SPI/I2S flag */
772   if ((SPIx->SR & SPI_I2S_FLAG) != (uint16_t)RESET)
773   {
774     /* SPI_I2S_FLAG is set */
775     bitstatus = SET;
776   }
777   else
778   {
779     /* SPI_I2S_FLAG is reset */
780     bitstatus = RESET;
781   }
782   /* Return the SPI_I2S_FLAG status */
783   return  bitstatus;
784 }
```

5.4.Lập trình bằng Software:

#Include: Thư viện

```
#include "stm32f10x.h" // Thư viện chung vkd
#include "stm32f10x_gpio.h" // Thư viện gpio giao tiếp với tb bên ngoài
#include "stm32f10x_rcc.h"// Thư viện rcc cấp xung clock cho ngoại vi hoạt động
#include "stm32f10x_tim.h"// Thư viện timer để tạo hàm timer và delay
```

#Define: cổng giao tiếp cho SPI => chọn GPIOA, pin CS, SCK, MOSI, MISO chọn pin 4,5,6,7 , Cập clock cho SPI qua RCC_APB2Periph_GPIOA

Viết hàm:

- RCC_Config() : cấp clock cho ngoại vi hoạt động gồm TIMER, GPIO, SPI
- TIMER(): để tạo hàm delayms, hoặc tạo clock()
- Delayms(): tạo độ trễ ms
- Clock(): tạo xung đồng bộ, ví dụ : tạo xung clock mức 1 trong 4 us, mức 0 : 4us
- GPIO_Config() : Cấu hình 4 chân spi, chọn mode, speed cho các chân, cho vào cùng mảng với các chân cùng mode, speed để dễ thao tác.
- SPI_Idle(): quy đinh mức logic 4 chân SPI khi ở trạng thái Idle.
- SPI_Master_trans() : Hàm truyền mảng dữ liệu, truyền theo từng bit. Hàm này cần thực hiện kéo CS = 0, kiểm tra Clock = 1, dịch 1 bit, kiểm tra CS=1 dừng truyền.

Giải thích từng dòng code

Code chương trình MASTER truyền dữ liệu.

```
// Chương Trình MASTER truyền mảng dữ liêu Datatrans[] từ MASTER truyền qua SPI.
// 1. include thư viện phần cứng sử dụng, gpio, rcc, timer, spi.

#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_tim.h"
#include "stm32f10x_spi.h"

//2. Định nghĩa chân cho SPI, chọn cổng ra GPIOA, các chận PA4,5,6,7 đại diện cho 4 chân SPI
#define SPI_CS GPIO_Pin_4
#define SPI_SCK GPIO_Pin_5
#define SPI_MISO GPIO_Pin_6
#define SPI_MOSI GPIO_Pin_7
#define SPI_GPIO GPIOA
#define SPI_RCC RCC_APB2Periph_GPIOA // Dùng SPI, GPIOA nên cấp clock qua bus APB2Periph

void RCC_config(){
  RCC_APB2PeriphClockCmd(SPI_RCC,ENABLE);          // Cấp clock cho GPIOA
```

```
   RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE);  // Cấp Clock cho timer 2, nằm trên bus APB1Periph
}

void GPIO_config(){
   GPIO_InitTypeDef GPIO_Struct;
   GPIO_Struct.GPIO_Pin = SPI_CS | SPI_SCK | SPI_MOSI;  // Vs Master Các chân CS, SCK, MOSI đều là ouput, cùng mode và speed nên ghép chúng vào 1 cùng mảng pin
   GPIO_Struct.GPIO_Mode = GPGPIO_Mode_Out_PP;         // chế độ thả chôi điện áp
   GPIO_Struct.GPIO_Speed = GPIO_Speed_50MHz;          // tốc độ truyền 50Hz
   GPIO_Init(SPI_GPIO,&GPIO_Struct);                   // Nạp các cấu hình trên cho biến GPIO_struct

   GPIO_Struct.GPIO_Pin = SPI_MISO;                    // Chân MISO là Input và khác mode nên cấu hình riêng
   GPIO_Struct.GPIO_Mode = GPIO_Mode_IN_FLOATING;
   GPIO_Struct.GPIO_Speed = GPIO_Speed_50MHz;
   GPIO_Init(SPI_GPIO,&GPIO_Struct);                   // Nạp cấu hình cho riêng chân MISO.

// Để config cho GPIO trước tiên cần vào thư viện gpio.h, tìm từ khóa "typedef struct" xem cách cấu trúc set các thông số.
//Trong Struct GPIO cần cấu hình Pin, Mode, Speed

}


void SPI_Idle(){
   GPIO_WriteBit(SPI_GPIO, SPI_SCK, 0);  //bitreset = 0
   GPIO_WriteBit(SPI_GPIO, SPI_CS, 1);   // bitset = 1
   GPIO_WriteBit(SPI_GPIO, SPI_MISO, 0);
   GPIO_WriteBit(SPI_GPIO, SPI_MOSI, 0);
// KHi các chân ở trạng thái nghỉ Idle, cần reset các chân về mức logic nghỉ.
}

void TIM_config(){
   TIM_TimeBaseInitTypeDef TIM_Struct;
   TIM_Struct.TIM_ClockDivision = TIM_CKD_DIV1;      // phép chia nhỏ xung clock. ở đây devided to 1.
   TIM_Struct.TIM_CounterMode = TIM_CounterMode_Up;
   TIM_Struct.TIM_Period = 0xFFFF;
   TIM_Struct.TIM_Prescaler = 7200 - 1;              //0.1ms
   TIM_TimeBaseInit(TIM2, &TIM_Struct);              // Nạp cấu hình cho TIM_Struct
   TIM_Cmd(TIM2,ENABLE);                             // Đối với TIMER cần hàm này để bật timer 2 cho phép hoạt động.
// Vào thư viện tim.h search từ khóa "typedef struct" để xem các cài đặt thông số cho timer struct.
}
void delay_ms(uint16_t time){
   TIM_SetCounter(TIM2,0);
```

```
    while(TIM_GetCounter(TIM2) < time * 10);
}

void Clock()
{
    GPIO_WriteBit(SPI_GPIO, SPI_SCK, Bit_SET); // ghi bit 1 lên chân SCK và delay 4 ms
    delay_ms(4);
    GPIO_WriteBit(SPI_GPIO, SPI_SCK, Bit_RESET); // Ghi bit 0 lên chân SCK và delay 4
ms
    delay_ms(4);
// Hàm clock để đồng bộ tín hiệu giữa 2 MCU
}

void SPI_Master_Transmit(uint8_t u8Data){   //ob1001 0000
    uint8_t u8Mask = 0x80;                  // ob1000 0000
    uint8_t tempData;                       // Tạo biến chứa dữ liệu tạm thời
    GPIO_WriteBit(SPI_GPIO, SPI_CS, Bit_RESET); // Ghi chân CS xuống mức thấp =0
    delay_ms(1);
    for(int i=0; i<8; i++){                 // Vòng lặp for thực hiện 8 lần để truyền 8 bit
        tempData = u8Data & u8Mask;             // mặt nạ u8Mask chỉ có bit MSB =1, các bit còn
lại =0 nên phép and với u8Mask chỉ có vị trí bit MSB giữ nguyên giá trị,còn lại =0
        if(tempData){
            GPIO_WriteBit(SPI_GPIO, SPI_MOSI, Bit_SET);
            delay_ms(10);
        } else{
            GPIO_WriteBit(SPI_GPIO, SPI_MOSI, Bit_RESET);
            delay_ms(10);
        }
        u8Data<<=1;                         // Dịch 1 bit dữ liêu sang trái để tiếp tục phép & mặt nạ
u8Mask
        Clock();                        // Sau khi dịch 1 bit thì truyền 1 clock đi
    }
    GPIO_WriteBit(SPI_GPIO, SPI_CS, Bit_SET);   // Ghi chan CS về mức thấp =0, kết thúc
truyền
    delay_ms(10);
    // Nếu tempdata ở mức cao thực hiện ghi chân MOSI lên 1 (BIT_SET), ngược lại ghi cahan
MOSI về 0 (BIT_RESET)
    // mỗi bước ghi thực hiện delay khoảng 10ms
    // Dịch u8Data 1 bit sang trái. để thực hiện phép & với bit MSB trong u8Mask.
}
uint8_t Datatrans[] = {1, 3, 9, 10, 15, 19, 90}; //data
int main(){
    RCC_config();
    GPIO_config();
    TIM_config();
    SPI_Idle();

    while(1){
```

```
        for(int i = 0; i < sizeof(Datatrans)/sizeof(Datatrans[0]); i++){
            SPI_Master_Transmit(Datatrans[i]);
            delay_ms(1000);
        }
    }
}
```

Code Chương trình SLAVE nhận dữ liệu từ Master truyền sang

```
// Chương Trình SLAVE nhận mảng dữ liêu Datatrans[] từ MASTER truyền qua SPI.
// 1. include thư viện phần cứng sử dụng, gpio, rcc, timer, spi.

#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_tim.h"
#include "stm32f10x_spi.h"

//2. Định nghĩa chân cho SPI, chọn cổng ra GPIOA, các chận PA4,5,6,7 đại diện cho 4 chân SPI
#define SPI_CS GPIO_Pin_4
#define SPI_SCK GPIO_Pin_5
#define SPI_MISO GPIO_Pin_6
#define SPI_MOSI GPIO_Pin_7
#define SPI_GPIO GPIOA
#define SPI_RCC RCC_APB2Periph_GPIOA // Dùng SPI, GPIOA nên cấp clock qua bus APB2Periph

void RCC_config(){
    RCC_APB2PeriphClockCmd(SPI_RCC,ENABLE);             // Cấp clock cho GPIOA
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE);  // Cấp Clock cho timer 2, nằm trên bus APB1Periph
}

void GPIO_config(){
    GPIO_InitTypeDef GPIO_Struct;
    GPIO_Struct.GPIO_Pin = SPI_CS | SPI_SCK | SPI_MOSI;  // Vs SLave Các chân CS, SCK, MOSI đều là input, cùng mode và speed nên ghép chúng vào 1 cùng mảng pin
    GPIO_Struct.GPIO_Mode = GPIO_Mode_IN_FLOATING;       // chế độ thả chôi điện áp
    GPIO_Struct.GPIO_Speed = GPIO_Speed_50MHz;           // tốc độ truyền 50Hz
    GPIO_Init(SPI_GPIO,&GPIO_Struct);                    // Nạp các cấu hình trên cho biến GPIO_struct

    GPIO_Struct.GPIO_Pin = SPI_MISO;                     // Chân MISO là output của Slave và khác mode nên cấu hình riêng
    GPIO_Struct.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Struct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(SPI_GPIO,&GPIO_Struct);                    // Nạp cấu hình cho riêng chân MISO.
```

```
// Để config cho GPIO trước tiên cần vào thư viện gpio.h, tìm từ khóa "typedef struct" xem cách cấu trúc set các thông số.
//Trong Struct GPIO cần cấu hình Pin, Mode, Speed

}


void SPI_Idle(){
   GPIO_WriteBit(SPI_GPIO, SPI_SCK, 0);  //bitreset = 0
   GPIO_WriteBit(SPI_GPIO, SPI_CS, 1);   // bitset = 1
   GPIO_WriteBit(SPI_GPIO, SPI_MISO, 0);
   GPIO_WriteBit(SPI_GPIO, SPI_MOSI, 0);
// KHi các chân ở trạng thái nghỉ Idle, cần reset các chân về mức logic nghỉ.
}

void TIM_config(){
   TIM_TimeBaseInitTypeDef TIM_Struct;
   TIM_Struct.TIM_ClockDivision = TIM_CKD_DIV1;      // phép chia nhỏ xung clock. ở đây devided to 1.
   TIM_Struct.TIM_CounterMode = TIM_CounterMode_Up;
   TIM_Struct.TIM_Period = 0xFFFF;
   TIM_Struct.TIM_Prescaler = 7200 - 1;              //0.1ms
   TIM_TimeBaseInit(TIM2, &TIM_Struct);              // Nạp cấu hình cho TIM_Struct
   TIM_Cmd(TIM2,ENABLE);                             // Đối với TIMER cần hàm này để bật timer 2 cho phép hoạt động.
// Vào thư viện tim.h search từ khóa "typedef struct" để xem các cài đặt thông số cho timer struct.
}
void delay_ms(uint16_t time){
   TIM_SetCounter(TIM2,0);
   while(TIM_GetCounter(TIM2) < time * 10);
}

void Clock()
{
   GPIO_WriteBit(SPI_GPIO, SPI_SCK, Bit_SET);
   delay_ms(4);
   GPIO_WriteBit(SPI_GPIO, SPI_SCK, Bit_RESET);
   delay_ms(4);
// Hàm clock để đồng bộ tín hiệu giữa 2 MCU
}

uint8_t SPI_Slave_Receive(void) {
   uint8_t dataReceive = 0x00;                       // khởi tạo biến dataReceive gán giá trị dầu =0x00

   while (GPIO_ReadInputDataBit(SPI_GPIO, SPI_CS));       // Vong lap while điểm tra trạng thái chan CS liên tục, nếu CS ở mức cao =1 thì vòng lặp tiếp tục đến khi CS = 0 mức Low, nó sé thoát khỏi vòng lặp
```

```
    for (int i = 0; i < 8; i++) {                       // Thực hiện vòng lặp for 8 lần để ghi 8 bit dữ liệu
        while (!GPIO_ReadInputDataBit(SPI_GPIO, SPI_SCK));  // trang thái Idle, SCK = 0 mức thấp. Đợi khi nào SCK =1 mức cao=> !SCK = 0 => thoát vòng lặp, thực hiện nhận dữ liệu

        if (GPIO_ReadInputDataBit(SPI_GPIO, SPI_MOSI)) {    // Kiên tra chân MOSI ở mức cao thì thực hiện phép OR dataReceive với 1. ghi bit 1 vào biến
            dataReceive |= 1;
        }
        dataReceive <<= 1;                              // Dịch 1 bit để đợi ghi bit Data tiếp theo
        while (GPIO_ReadInputDataBit(SPI_GPIO, SPI_SCK));   // SCK đang mưc cao =1, Đợi khi nào SCK về mức thấp =0. Tức là đợi hết xung 1 của clock Nó sẽ thoát vòng lặp while.
    }
    while (!GPIO_ReadInputDataBit(SPI_GPIO, SPI_CS));       // Đợi CS lên mức cao, kết thúc vòng lặp, kết thúc nhận dữ liệu

    return dataReceive;
    // Hàm nhận SPI_Slave_Receive() có kiểu trả về 8 bit (uint8_t)
    // Hàm Receive hoạt động với điều khiên chân CS = 0, SCK = 1. dùng while để kiểm tra điều kiện 2 chân này.
    // Vòng lặp while(x), mặc địch tham số trong vòng lặp ở mức cao. chỉ thoát ra vòng lặp này tham số x có giá trị mức thấp = 0.
    // ví dụ while(1){} sẽ là vòng lặp vô tận
    // Hàm điều kiện if(x): nếu không có điều kiện cụ thể, mặc định x là mưc cao = 1 thì thực hiện điều kiện trong hàm.
}

uint8_t ReceiveData;

int main(){
    RCC_config();
    GPIO_config();
    TIM_config();
    SPI_Idle();

    while(1)
    {
        ReceiveData = SPI_Slave_Receive();

    }
}
```