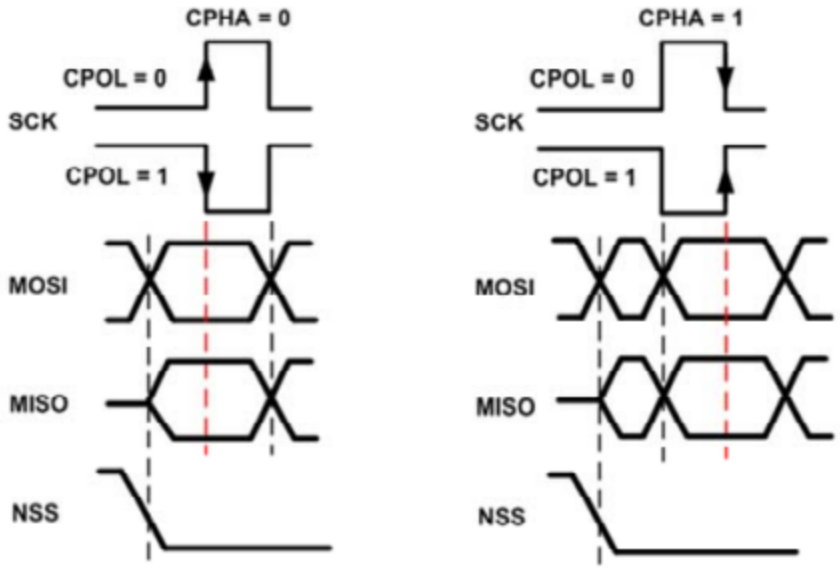
Interface serial de periféricos (SPI)



Implementación del spi con lineas digitales:

```
///  A partir de aqui la configuracion de spi por puerto
//-------------------------------------------------------------------------//
//FUNCTION NAME:SPI_CS_1
//FUNCTION     :Pone el bit SPI_CS en 1
//INPUT        :none
//OUTPUT       :none
//-------------------------------------------------------------------------//
void SPI_CS_1(void)
{
    GPIO_SetBits(GPIOB, GPIO_Pin_11);
    return;
}

//-------------------------------------------------------------------------//
//FUNCTION NAME:SPI_CS_0
//FUNCTION     :Pone el bit SPI_CS en 0
//INPUT        :none
//OUTPUT       :none
//-------------------------------------------------------------------------//
void SPI_CS_0(void)
{
    GPIO_ResetBits(GPIOB, GPIO_Pin_11);
    return;
}

//-------------------------------------------------------------------------//
//FUNCTION NAME:SPI_MOSI_1
```

```
//FUNCTION     :Pone el bit SPI_MOSI en 1
//INPUT        :none
//OUTPUT       :none
//-----------------------------------------------------------------------------//
void SPI_MOSI_1(void)
{
    GPIO_SetBits(GPIOB, GPIO_Pin_15);
    return;
}

//-----------------------------------------------------------------------------//
//FUNCTION NAME:SPI_MOSI_0
//FUNCTION     :Pone el bit SPI_MOSI en 0
//INPUT        :none
//OUTPUT       :none
//-----------------------------------------------------------------------------//
void SPI_MOSI_0(void)
{
    GPIO_ResetBits(GPIOB, GPIO_Pin_15);
    return;
}

//-----------------------------------------------------------------------------//
//FUNCTION NAME:SPI_MISO_
//FUNCTION     :Lee el bit SPI_MISO
//INPUT        :none
//OUTPUT       :1 si el bit es 1 o cero lo contrario
//-----------------------------------------------------------------------------//
uint8_t SPI_MISO_(void)
{
    if(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_14)== 1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

//-----------------------------------------------------------------------------//
//FUNCTION NAME:SPI_CK_1
//FUNCTION     :Pone el bit SPI_CK en 1
//INPUT        :none
//OUTPUT       :none
//-----------------------------------------------------------------------------//
void SPI_CK_1(void)
{
    GPIO_SetBits(GPIOB, GPIO_Pin_13);
    return;
```

```
}

//------------------------------------------------------------------------------//
//FUNCTION NAME:SPI_CK_0
//FUNCTION     :Pone el bit SPI_CK en 0
//INPUT        :none
//OUTPUT       :none
//------------------------------------------------------------------------------//
void SPI_CK_0(void)
{
    GPIO_ResetBits(GPIOB, GPIO_Pin_13);
    return;
}

//------------------------------------------------------------------------------//
//FUNCTION NAME: single_read
//FUNCTION     : read data from register
//INPUT        :regAddr: register address
//OUTPUT       :register value
//------------------------------------------------------------------------------//
uint8_t lora_single_read(uint8_t regAddr)
{
  uint8_t SPICount;                                       // Counter used to clock out the
data
  uint8_t SPIData;

  SPI_CS_1(); // Make sure we start with active-low CS high
  SPI_CK_0(); // and CK low
  SPIData = regAddr;                                        // Preload the data to be sent
with Address and Data

  SPI_CS_0(); // Set active-low CS low to start the SPI cycle
  spi_write(SPIData & 0x7f);
                                                           // and loop back to send the
next bit
  SPI_MOSI_0(); // Reset the MOSI data line

  SPIData = 0;
  for (SPICount = 0; SPICount < 8; SPICount++)                // Prepare to clock in the
data to be read
  {
    SPIData <<=1;                                           // Rotate the data
    SPI_CK_1(); // Raise the clock to clock the data out
    SPIData += SPI_MISO_();                                    // Read the data bit
    SPI_CK_0(); // Drop the clock ready for the next bit
  }                                                         // and loop back
  SPI_CS_1(); // Raise CS

  return ((uint8_t)SPIData);                                // Finally return the read data
}
```

```
//-------------------------------------------------------------------------//
//FUNCTION NAME: sin_Spi1_init
//FUNCTION     : inicializacion de pines que emulan el spi
//INPUT        :none
//OUTPUT       :none
//-------------------------------------------------------------------------//
void sin_Spi1_init(void)
{
    GPIO_InitTypeDef GPIO_Struct;

    //configuración de los pines de salida
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB, ENABLE);

    // GPIOB PIN13 (SPI2_SCK), PIN15(SPI2_MOSI) Y PIN11 (SPI2_NSS) - salidas
    GPIO_Struct.GPIO_Pin = GPIO_Pin_11 | GPIO_Pin_13 | GPIO_Pin_15;
    GPIO_Struct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Struct.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOB, &GPIO_Struct);
    // GPIOB PIN14 (SPI2_MISO) entrada
    GPIO_Struct.GPIO_Pin = GPIO_Pin_14;
    GPIO_Struct.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOB, &GPIO_Struct);

    // GPIOB PIN4 (GDO0) entrada, GPIOB PIN3 (GDO2) entrada
    GPIO_Struct.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_4;
    GPIO_Struct.GPIO_Mode = GPIO_Mode_IPU;
    GPIO_Init(GPIOB, &GPIO_Struct);

     // configuracion de interrupcion del pin GDO0
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOB, GPIO_PinSource4);// GDO0;
    EXTI_InitTypeDef EXTI_InitStruct;
    EXTI_InitStruct.EXTI_Line = EXTI_Line4;
    EXTI_InitStruct.EXTI_LineCmd = ENABLE;
    EXTI_InitStruct.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStruct.EXTI_Trigger = EXTI_Trigger_Rising;
    EXTI_Init(&EXTI_InitStruct);

    NVIC_InitTypeDef NVIC_Struct;
    NVIC_Struct.NVIC_IRQChannel = EXTI4_IRQn;
    NVIC_Struct.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_Struct.NVIC_IRQChannelSubPriority = 0;
    NVIC_Struct.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_Struct);

    return;
}

void spi_write(uint8_t One)
{
```

```
    uint8_t SPI_Count;
    uint8_t SPI_Data;

    SPI_Data = One;
    for (SPI_Count = 0; SPI_Count < 8; SPI_Count++)
    {
        if((SPI_Data & 0x80)== 0x80)
        {
            SPI_MOSI_1();
        }
        else
        {
            SPI_MOSI_0();
        }
        SPI_CK_1();
        SPI_CK_0();
        SPI_Data <<= 1;
    }

    return ;
}

//-------------------------------------------------------------------------//
//FUNCTION NAME: single_write
//FUNCTION     : write data to register
//INPUT        :regAddr: register address; regData: register value
//OUTPUT       :none
//-------------------------------------------------------------------------//
void lora_single_write(uint8_t regAddr, uint8_t regData)
{
    unsigned char SPIData; // Define a data structure for the SPI data

    SPI_CS_1(); // Make sure we start with active-low CS high
    SPI_CK_0(); // and CK low

    SPIData = regAddr; // Preload the data to be sent with Address
                       // Set active-low CS low to start the SPI cycle
                       // Although SPIData could be implemented as an "int",
                       // resulting in one
                       // loop, the routines run faster when two loops
                       // are implemented with
                       // SPIData implemented as two "char"s.
    SPI_CS_0();
    spi_write(SPIData | 0x80);
    SPIData = regData; // Preload the data to be sent with Data
    spi_write(SPIData & 0xff);

    SPI_CS_1();
    SPI_MOSI_0();
    return;
```

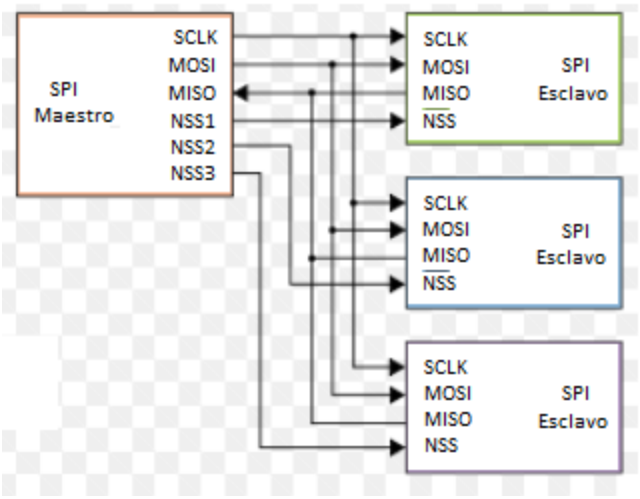
```
}

//-------------------------------------------------------------------------//
//FUNCTION NAME:lora_write_buffer
//FUNCTION     : write burst data to register
//INPUT        :addr: register address; buffer:register value array; num:number to write
//OUTPUT       :none
//-------------------------------------------------------------------------//
void lora_write_buffer(uint8_t addr, uint8_t *buffer, uint8_t num)
{
    uint8_t i, temp;
    temp = addr;
    SPI_CS_1();
    SPI_CK_0();
    SPI_CS_0();
    spi_write(temp | 0x80);
    for (i = 0; i < num; i++)
    {
        spi_write(buffer[i] & 0xff);
    }
    SPI_CS_1();
    SPI_MOSI_0();
    return;
}

//-------------------------------------------------------------------------//
//FUNCTION NAME:lora_read_buffer
//FUNCTION     : read burst data from register
//INPUT        :addr: register address; buffer:array to store register value; num: number to
read
//OUTPUT       :none
//-------------------------------------------------------------------------//
void lora_read_buffer(uint8_t addr, uint8_t *buffer, uint8_t num)
{
   uint8_t i,temp;
   uint8_t SPICount;
   uint8_t SPIData;

   temp = addr;
   SPI_CS_1();
   SPI_CK_0();
   SPI_CS_0();
   spi_write(temp & 0x7f);
   SPI_MOSI_0();
   for(i=0;i<num;i++)
   {
       SPIData = 0;
       for(SPICount = 0; SPICount < 8; SPICount++)
       {
            SPIData <<=1;
```

```
            SPI_CK_1();
            SPIData += SPI_MISO_();
            SPI_CK_0();
         }
        buffer[i]= SPIData;
   }
   SPI_CS_1();
   return;
}
```

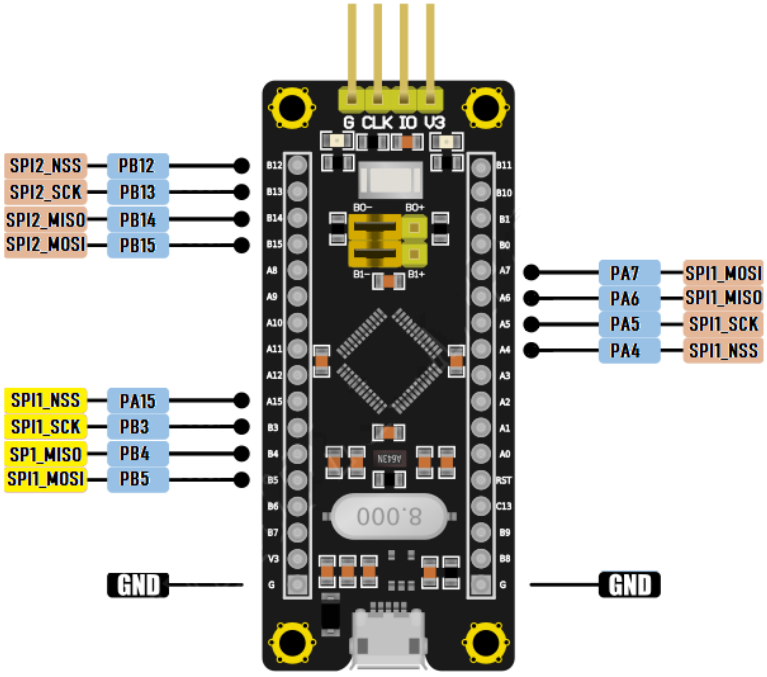
La interface serial de periféricos, SPI, es un módulo de comunicaciones sincrónico que es más rápido que el módulo Uart o el I2C, pues puede transmitir datos a una velocidad de hasta 18 Mbits/s en el Stm32f103x, y se usa para intercambiar datos entre un microcontrolador y otro microcontrolador o un periférico. La comunicación puede ser half/ full- duplex, bidireccional y funciona entre un maestro y un esclavo. En el caso del maestro, este provee el reloj al esclavo. Otra de las características del SPI es que se puede conectar a varios periféricos o microcontroladores en paralelo. La comunicación SPI está dirigida a periféricos con alta transferencia de datos, como módulos de radio, pantallas gráficas, entre otras.

El SPI se conecta a periféricos externos mediante cuatro líneas:

- MISO: (Master Input Slave Output) Por este pin el dato entra al maestro y sale del esclavo.
- MOSI: (Master Output Slave Input) Por este pin el dato sale del maestro y entra al esclavo.
- SCK: En este pin sale la señal de reloj en el lado del maestro y entra en el lado del esclavo.
- NSS: Este pin es de salida en el lado maestro y sirve para seleccionar el periférico esclavo en el caso que hayan varios periféricos conectados como esclavos a un maestro. El pin NSS como entrada se puede controlar por cualquier pin GPIO del maestro.



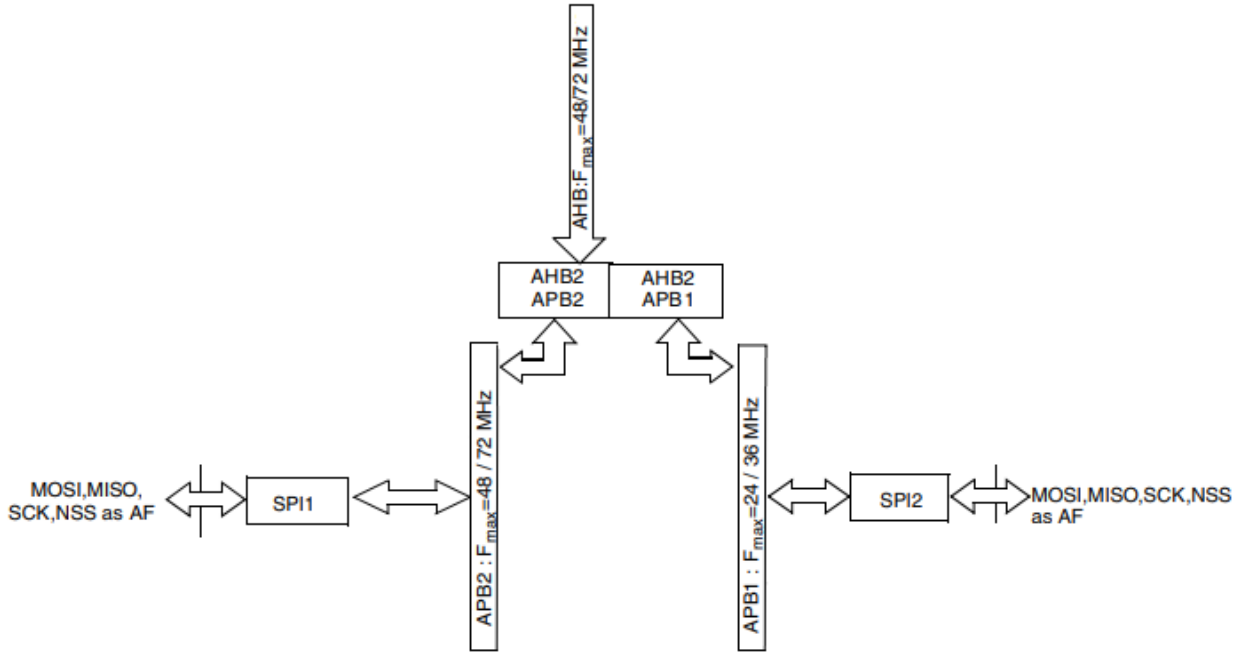
Los pines MOSI están conectados juntos como los pines MISO, como lo muestra la figura. La comunicación siempre la inicia el maestro y cuando el dato va desde el maestro hasta el esclavo por el pin MOSI, el esclavo responde por el pin MISO, de esta forma la comunicación es full duplex. El Stm32f103x tiene dos puertos SPI, SPI1 y SPI2. Los pines para el SPI1 y SPI2 se muestran en la siguiente figura. Nótese que en amarillo están los valores de NSS,SCK, MOSI y MISO de SPI1 remapeados a otros pines.



Función Alterna	SPI1_REMAP = 0	SPI1_REMAP = 1
SPI1_NSS	PA4	PA15
SPI1_SCK	PA5	PB3
SPI1_MISO	PA6	PB4
SPI1_MOSI	PA7	PB5

El remapeo de las señales de SPI1 se hace con la función `GPIO_PinRemapConfig(GPIO_Remap_SPI1, ENABLE)`. Para volver a los pines iniciales por defecto, se usa la función `GPIO_PinRemapConfig(GPIO_Remap_SPI1, DISABLE)`.

El STM32F103C8T6 tiene los módulos SPI1 y SPI2 en diferentes buses.



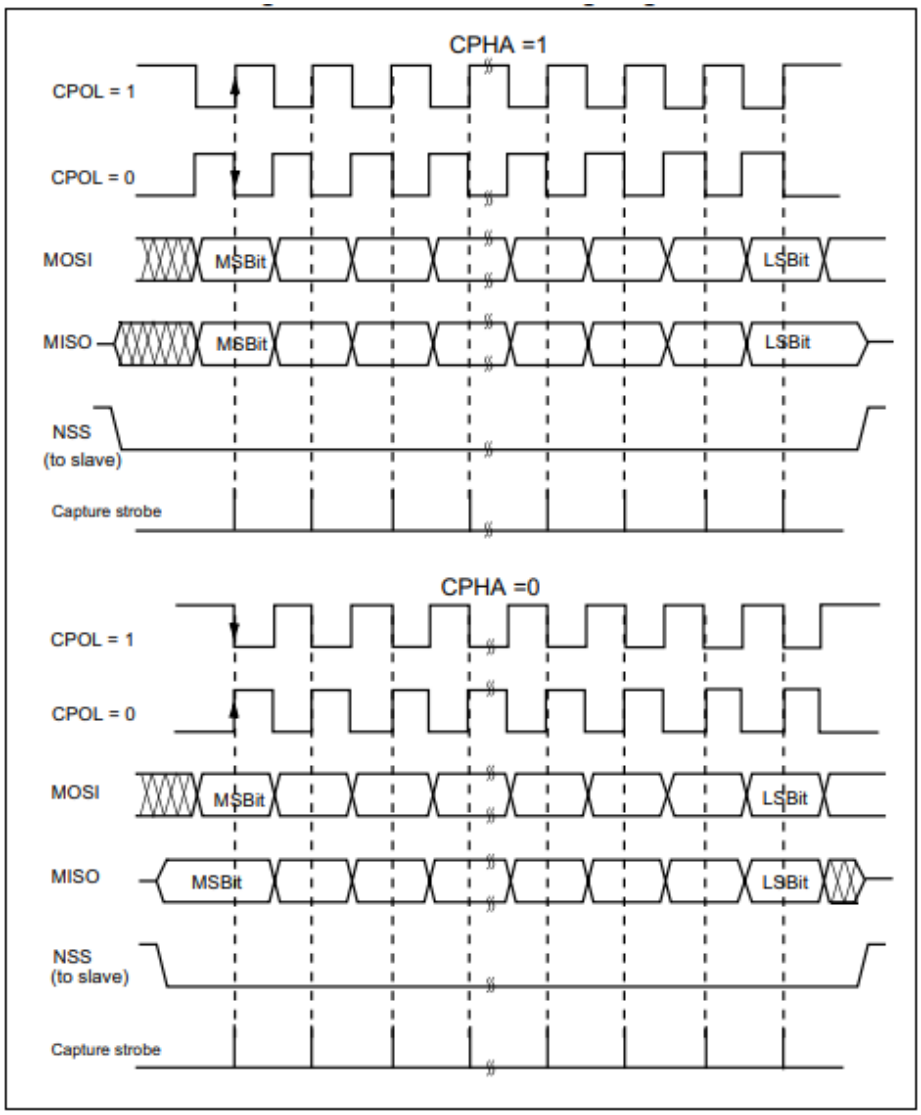
El SPI1 está conectado al bus de alta velocidad APB2, mientras que el SPI2 está conectado al de baja velocidad APB1, así que la habilitación del reloj para cada uno se hace con la función: `RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE)`, para el SP1 y `RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE)`, para el SP2.

Cada SPI tiene tres bits de prescaler, o divisor de frecuencias, que produce hasta ocho frecuencias diferentes. El dato se puede configurar a 8 o 16 bits. Los dos módulos SPI pueden ser atendidos por el DMA para hacer la comunicación más rápida sin el procesador.

Para la configuración del SPI, se usa primero una inicialización de una estructura de tipo SPI_InitTypeDef con la función SPI_StructInit(&SPI_InitStruct). Esta función rellena las variables dentro de la estructura con la siguiente información:

- `SPI_Direction = SPI_Direction_2Lines_FullDuplex`: Indica si la comunicación es unidireccional o bidireccional. Se puede modificar con `SPI_Direction_2Lines_RxOnly`, `SPI_Direction_1Line_Rx` o `SPI_Direction_1Line_Tx`.
- `SPI_Mode = SPI_Mode_Slave`: Indica el modo de operación y puede ser `SPI_Mode_Master` o `SPI_Mode_Slave`.
- `SPI_DataSize = SPI_DataSize_8b`: Indica el tamaño del dato en bits. Se puede modificar por `SPI_DataSize_16b`.
- `SPI_CPOL = SPI_CPOL_Low`: Indica la polaridad del reloj. También puede ser `SPI_CPOL_High`.
- `SPI_CPHA = SPI_CPHA_1Edge`: Indica el borde del pulso del reloj para captura de bits. Puede también ser `SPI_CPHA_2Edge`.
- `SPI_NSS = SPI_NSS_Hard`: Indica si la señal de NSS se maneja por el pin NSS o por software con el bit SSI. Se puede modificar con `SPI_NSS_Soft`.
- `SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_2`: Indica el divisor de reloj con el que se configura el reloj del SPI. El número `2` se puede cambiar también por `4, 8, 16, 32, 64, 128 o 256`.
- `SPI_FirstBit = SPI_FirstBit_MSB`: Indica si la transferencia de datos empieza por el bit más significativo del dato o el menos significativo. Se puede cambiar por `SPI_FirstBit_LSB`.
- `SPI_CRCPolynomial = 7`: Indica el polinomio usado para el cálculo del byte de seguridad CRC.

Una vez se hagan los cambios en las variables de la estructura, se puede inicializar el SPI con la función `SPI_Init(SPI1, &SPI_InitStruct)`, para el SPI1 o hacerlo para el SPI2. Una descripción sobre el comportamiento de la comunicación spi con respecto a la configuración de SPI_CPOL y SPI_CPHA se muestra en la siguiente figura.



La librería SPL tiene un gran repertorio de funciones en el SPI. Aquí la descripción de cada una de ellas.

- Aunque la estructura ya tiene todos los parámetros para la inicialización del SPI, después de la inicialización algunos de estos parámetros se pueden cambiar, como el tamaño del dato con la función `SPI_DataSizeConfig(SPI1, SPI_DataSize_8b)`.
- Luego de la inicialización del SPI, se debe habilitar para empezar a hacer la comunicación usando la función `SPI_Cmd(SPI1, ENABLE)`.
- Después de estar habilitado el SPI, se pueden enviar datos con la función `SPI_I2S_SendData(SPI1, dato)`, donde `dato` es un byte o dos, según lo configurado como tamaño del dato.
- Para saber si llegó un dato del periférico, se usa la función `FlagStatus= SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE)`. Esta misma función sirve para ver otros estados de las banderas, como `SPI_I2S_FLAG_TXE`, que indica si ya se puede transmitir, además de otras banderas de indicación de error.

- Para recibir datos se usa la función `SPI_I2S_ReceiveData(SPI1)`.
- Para borrar las banderas de indicación, se usa la función `SPI_I2S_ClearFlag(SPI1, SPI_I2S_FLAG_RXNE)` con el tipo de bandera que se quiere borrar. El mismo uso de mirar y modificar banderas, pero en el caso de interrupción, se usan las funciones `ITStatus= SPI_I2S_GetITStatus(SPI1, SPI_I2S_IT_RXNE)` y `SPI_I2S_ClearITPendingBit(SPI1, SPI_I2S_IT_RXNE)`, respectivamente, para la bandera de recepción de datos o cualquier otra referida a comunicación SPI. A propósito de interrupciones, se puede habilitar la interrupción por uno cualquiera de los eventos de recepción de datos (`SPI_I2S_IT_RXNE`), transmisión lista para enviar datos (`SPI_I2S_IT_TXE`) o indicación de error (`SPI_I2S_IT_ERR`) con la función `SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_RXNE, ENABLE).`
- La llegada de datos o la transmisión de datos masivos se puede programar para que los datos fluyan entre la memoria y el periférico sin intervención del procesador por medio de la función `SPI_I2S_DMACmd(SPI1, SPI_I2S_DMAReq_Rx, ENABLE)`, para lectura de datos, o usando el parámetro `SPI_I2S_DMAReq_Tx`, para escritura.
- Si en la inicialización del SPI estaba el manejo del pin NSS por software, se puede usar la función `SPI_NSSInternalSoftwareConfig(SPI1, SPI_NSSInternalSoft_Set)` para poner en uno este pin, o `SPI_NSSInternalSoftwareConfig(SPI1, SPI_NSSInternalSoft_Reset)` para poner ese mismo pin en cero internamente. Si se quiere que este mismo estado salga al pin NSS, se puede usar la función `SPI_SSOutputCmd(SPI1, ENABLE).`
- Si en la configuración del SPI se escogió comunicación bidireccional, se puede cambiar la dirección de recepción por transmisión con la función `SPI_BiDirectionalLineConfig(SPI1, SPI_Direction_Tx).`
- En cuanto al manejo de los códigos de seguridad CRC, la librería SPL provee algunas funciones para su manipulación. Por ejemplo, se puede conocer el polinomio en la configuración del SPI con la función `polinomio= SPI_GetCRCPolynomial(SPI1).` También se puede obtener el valor CRC de recepción o transmisión del SPI con la función `valor_crc= SPI_GetCRC(SPI1, SPI_CRC_Rx)` o `valor_crc= SPI_GetCRC(SPI1, SPI_CRC_Rx)`, respectivamente. También se puede habilitar o deshabilitar el cálculo del código CRC de los datos transmitidos con la función `SPI_CalculateCRC(SPI1, ENABLE)`, o transmitir el código CRC con la función `SPI_TransmitCRC(SPI1).`
- En caso que se quiera reiniciar la operación del SPI con sus valores por defecto que tendría después de un reset, se puede usar la función `SPI_I2S_DeInit(SPI1).`

Como ejemplo de operación se va a usar el SPI1 como maestro y el SPI2 como esclavo. Adicionalmente se va a usar el Uart1 para controlar la comunicación entre los dos módulos SPI. La idea es que todo lo que se escribe en el terminal con el Uart, se envíe por SPI1 a SPI2, luego se lea lo que llega en SPI2 y se envíe por Uart1. Las conexiones de los pines deben ser:

```
SPI1_NSS (PA4)   -->  No conectar
SPI1_SCK (PA5)   -->  SPI2_SCK (PB13)
SPI1_MISO (PA6)  <--  SPI2_MISO (PB14)
SPI1_MOSI (PA7)  -->  SPI2_MOSI (PB15)
SPI2_NSS (PB12)  -->  GND
```

Para la configuración del SPI1 como maestro se tiene en cuenta la frecuencia, que para este caso se programa en 7 Mps de un reloj en SysClk igual a 56 Mhz, con PCLK2 con la misma

frecuencia. Así que el prescaler debe ser igual a 8. El pin NSS se programa por software y el modo debe ser maestro.

```
void Spi1_Maestro_Conf(void)
{
    //habilitación del reloj para spi1, funciones alternas y puerto A
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1 | RCC_APB2Periph_AFIO | RCC_APB2Periph_GPIOA, ENABLE);
    // Configuración de los pines del spi1
    GPIO_InitTypeDef GPIO_Struct;
    // GPIOA PIN5 (SPI1_SCK), PIN7(SPI1_MOSI) Y PIN4 (SPI1_NSS) función alterna- salidas
    GPIO_Struct.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_7 | GPIO_Pin_4;
    GPIO_Struct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Struct.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOA, &GPIO_Struct);

    // GPIOA PIN6 (SPI1_MISO) funcion alterna- entrada
    GPIO_Struct.GPIO_Pin = GPIO_Pin_6;
    GPIO_Struct.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_Struct);

    // configuración de SPI1 como maestro
    SPI_InitTypeDef SPI_InitStruct;
    // configuración de SPI1 como maestro con una frecuencia de 7 Mhz, maestro y NSS por software
    SPI_StructInit(&SPI_InitStruct);
    SPI_InitStruct.SPI_BaudRatePrescaler=SPI_BaudRatePrescaler_8; // 56Mz/ 8= 7 Mbps
    SPI_InitStruct.SPI_Mode= SPI_Mode_Master;
    SPI_InitStruct.SPI_NSS= SPI_NSS_Soft;
    SPI_Init(SPI1, &SPI_InitStruct);

    // habilitación spi1
    SPI_Cmd(SPI1, ENABLE);
    return;
}
```

La configuración del SPI2 como esclavo a la misma frecuencia, 7Mhz. PCLK1 tiene la mitad de la frecuencia de PCLK2, pues no soporta más de 36Mhz, luego el prescaler debe ser igual a 4, o sea, 28Mhz/ 4= 7Mhz.

```
void Spi2_Esclavo_Conf(void)
{
    //habilitación del reloj para spi2, funciones alternas y puerto B
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO | RCC_APB2Periph_GPIOB, ENABLE);
    // Configuración de los pines del spi2
    GPIO_InitTypeDef GPIO_Struct;
    // GPIOB PIN14 (SPI2_MISO) función alterna- salida
    GPIO_Struct.GPIO_Pin = GPIO_Pin_14;
    GPIO_Struct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Struct.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOB, &GPIO_Struct);

    // GPIOB PIN12 (SPI2_NSS), PIN13 (SPI2_SCK) y PIN15 (SPI2_MOSI) funcion alterna- entradas
    GPIO_Struct.GPIO_Pin = GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_15;
    GPIO_Struct.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOB, &GPIO_Struct);

    // configuración del SPI2 como esclavo
    SPI_InitTypeDef SPI_InitStruct;
```

```
    // configuración de SPI2 como esclavo con una frecuencia de 7 Mhz, maestro y NSS por software
    SPI_StructInit(&SPI_InitStruct);
    SPI_InitStruct.SPI_BaudRatePrescaler=SPI_BaudRatePrescaler_4; // 28Mz/ 4= 7 Mbps
    SPI_InitStruct.SPI_Mode= SPI_Mode_Slave;
    SPI_InitStruct.SPI_NSS= SPI_NSS_Soft;
    SPI_Init(SPI2, &SPI_InitStruct);

    // habilitación SPIs
    SPI_Cmd(SPI1, ENABLE);
    SPI_Cmd(SPI2, ENABLE);
    return;
}
```

Dentro del while(1) se recibe el caracter del Uart1, se envía por Spi1 al Spi2, se cambian los caracteres a mayúscula y luego se envía de nuevo al Uart1.

```
while(1)
  {
      // espera por caracter desde terminal uart
      while(USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET);
      dato= USART_ReceiveData(USART1);
      // el dato que lea del uart, lo manda a spi1
      SPI_I2S_SendData(SPI1, dato);
      // espera por dato que llegue a spi2
      while(SPI_I2S_GetFlagStatus(SPI2, SPI_I2S_FLAG_RXNE)== RESET);
      dato= SPI_I2S_ReceiveData(SPI2);
      // conversión de los caracteres a mayúsculas
      if((dato> 96) && (dato< 123))
      {
          dato= dato- 32;
      }

      // dato leído de spi2 lo envía de nuevo al uart
      USART_SendData(USART1, dato);
  }
```