# 18. Serial communication - SPI

*Some sensors implement SPI (Serial Peripheral Interface) protocol for data transfer. An example of communication between a microcontroller and an accelerometer sensor using the SPI interface will be demonstrated in this example.*

## 18.1. The SPI protocol and hardware

The SPI protocol defines a bus with four wires (four signals) and a common ground. There is one master device controlling the activity on the bus, and one active slave device. The slave is active only when the signal Slave Select (SS) enables it; this signal is provided by the master. There can be more than one slave connected to the SPI bus, but each slave requires its own Slave Select (SS1, SS2, …) signal, see Fig. 18.1. Data gets transferred serially bit-by-bit. There are two signals to carry information, one from master to slave (MOSI, Master Output Slave Input, driven by master), and one for the opposite direction (MISO, Master Input Slave Output, driven by slave). The last signal SCLK (Serial CLocK) assures the time synchronization between master and slave, and is always driven by master. There are streamlined versions of the SPI bus using only one signal to transfer data, but the direction of data must be reversed on request; we will not use this kind of data transfer in this example.

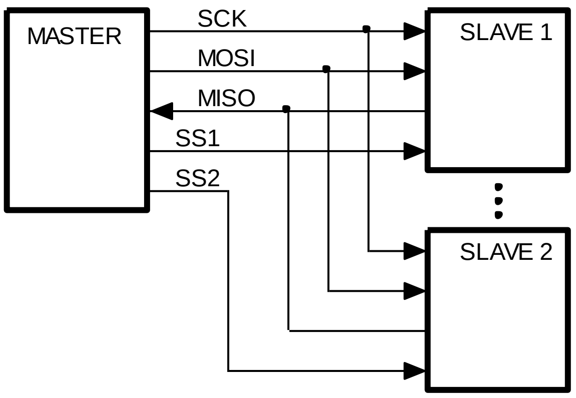

*Fig. 18.1: The SPI bus can be used to transfer data between the master and the slave using four wires.*

The speed of data transfer is higher than with I$^2$C bus, since the slave is selected using a hardware signal and there is no need to transfer the address of the slave. However, this requires multiple Slave Select signals if more than one slave is connected to the bus, and therefore more pins at the microcontroller. The speed of transfer can also be higher due to the device outputs which are able to force signals low or high, contrary to the open-drain outputs used at I$^2$C which can force signals low only. The logic levels are depend on the power supply for the devices connected to the SPI bus. The achievable speed of transmission conforms to the slowest device on the bus, as with I$^2$C bus.

The SPI protocol is less strict that the I2C protocol due to the fact that it was implemented first by several different companies and standardized only later. Variants of clock polarities, edge synchronizations, and even number of bits per transfer are used, and the designer should adopt its hardware to the SPI devices used. The microcontroller used here can implement some possible variants of the standard, and the variant implemented in the accelerometer LIS3LV02DQ is one of them.

The timing diagram of the required signals for the communication with the accelerometer chip is given in Fig. 18.2, the writing from master to slave being shown in the upper half. Slave select signal SS must first be forced low by the master, then a series of 2 times eight clock pulses (low-to-high transitions) are issued by the master at the SCK line. After this the signal SCK stays high, followed by the signal slave select SS. The value of the signal MOSI is clocked into the slave on positive edge of the clock signal SCK, and the MOSI signal can change either before or after the clocking edge, see the datasheet on LIS3LV02DQ for details. The first eight bits start with a bit to define either writing to slave (low) or reading from slave (high); this bit is low here. The next bit is fixed to zero, followed by six bits of address. This is the address to be used by the slave device to select one of its internal locations for writing, not the address of the slave on the bus, as with I$^2$C bus. Here, with SPI bus, the device is selected using the Slave Select signal! Next eight bits are simply the byte as it is supposed to be written into the slave. In the diagram address 0x21 is selected for writing, and value 0x40 is written. The signal MISO is not used during writing, and is not shown.

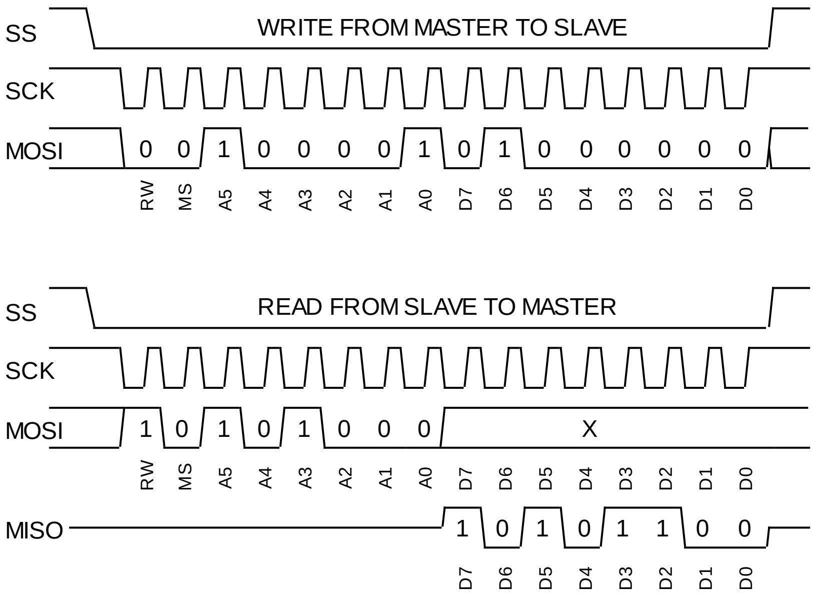
SS
WRITE FROM MASTER TO SLAVE
SCK
MOSI
0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0
RW MS A5 A4 A3 A2 A1 A0 D7 D6 D5 D4 D3 D2 D1 D0
SS
READ FROM SLAVE TO MASTER
SCK
MOSI
1 0 1 0 1 0 0 0
X
RW MS A5 A4 A3 A2 A1 A0 D7 D6 D5 D4 D3 D2 D1 D0
MISO
1 0 1 0 1 1 0 0
D7 D6 D5 D4 D3 D2 D1 D0

*Figure 18.2: SPI bus signals during the writing into the slave (top), and reading from the slave (bottom)*

The timing diagram for reading is given in the same figure, bottom half. The Slave Select signal and the 16 rising edges of the clock are the same as for writing, signals MOSI and MISO differ. The master first sends a command to read from the slave by forcing the first bit of the first byte at MOSI line high, and then pulling the same line low during the second bit. Next come six bits of address within the slave, here 0x28. After the first byte the MOSI signal is not important anymore. It can be left floating as shown in the figure, but it can also have any other logic value; its value will be ignored by the slave. However, clock pulses are still coming, and the slave now drives the MISO signal returning the byte to be read, shown as 0xAC. This sequence should be accepted by the SPI block within the microcontroller and combined into a byte.

## 18.2. The SPI protocol - writing

The Discovery board exposes pins associated with SPI block 2, making them available to the user. Additionally, the BaseBoard provides two connectors for SPI slaves both having separate, hardware defined Slave Select signals; the one we will use in this example is taken from port B, pin 12. Signals SCK, MOSI and MISO are available as alternate function at pins 13 to 15. It is convenient to summarize the steps for writing into:

1. Enable the Slave Select signal (port B, pin 12) by pulling it low.
2. Wait some time.
3. Send address and data from master to the slave using the SPI2 block; this is done by a call to a CMSIS function “SPI_I2S_SendData()”, which simply writes 16 bits into the data register DR of

the SPI2 block. Do not miss to keep the MSB of the 16 bit low, since this indicates "write" to the slave. The CMSIS function can be replaced by a hard-coded write into register SPI2->DR, as shown in this example.
4. Wait some time for all bits to be transferred to slave.
5. Disable the Slave Select signal (port B, pin 12) by pulling it high.

These steps are implemented in the function "LIS_write()" below:

```
void LIS_write (char address, char data)        {
  GPIOB->BSRRH = BIT_12;                                                       // step 1
  for (int i = 0; i < 150; i++)   {};           // waste time                  // step 2
  SPI2->DR = ((unsigned short)(address & 0x3f) << 8) + (unsigned short)(data);         // step 3
  for (int i = 0; i < 1200; i++)  {};           // waste time                  // step 4
  GPIOB->BSRRL = BIT_12;                                                       // step 5
}
```

Step 3 here may need additional discussion. The address must be a six bit value. However, since the 'address' is an argument to this function, an unwary user might specify more than six bits, and it is best to clip it to six bits by AND-ing it with 0x3f. This address must come to upper eight positions of the data register, so the software converts is into unsigned short integer, and shifts it for eight places. Finally, the 'data' is added to lower eight positions of the final value to be written into the 'SPI2->DR'.

The waiting in steps 2 and 4 is done by an empty loop, and is not very efficient and precise.

## 18.3. The SPI protocol – reading

Similarly, the steps required for reading from the slave can be summarized into:

1. Enable the Slave Select signal (port B, pin 12) by pulling it low.
2. Wait some time.
3. Send address from master to slave using the SPI2 block, the address must occupy the upper 8 bits of the short integer passed to a CMSIS function call "SPI_I2S_SendData()", which simply writes 16 bits into the data register DR of the SPI2 block. Do not miss to keep the MSB of the 16 bit high, since this indicates to read from the slave. The CMSIS function can be avoided by a hard-coded write into the register SPI2->DR.
4. Wait some time for all bits to be transferred to slave.
5. Disable the Slave Select signal (port B, pin 12) by pulling it high.
6. Read the content of the data register SPI2->DR in the SPI2 block, it contains the value returned by the slave. A CMSIS function "SPI_I2S_ReceiveData()" could be used instead of the hard-coded read from the register.

These steps are implemented in the function "LIS_read()" below:

```
short LIS_read (char address) {
  GPIOB->BSRRH = BIT_12;                                                       // step 1
  for (int i = 0; i < 150; i++)   {};           // waste time                  // step 2
  SPI2->DR = ((unsigned short )(address & 0x3f) << 8) + 0x8000;                // step 3
  for (int i = 0; i < 1200; i++)  {};           // waste time                  // step 4
  GPIOB->BSRRL = BIT_12;                                                       // step 5
  return (SPI2->DR & 0xff);                                                    // step 6
}
```

The step 3 might need additional discussion. The address of the register within the SPI slave must be reside in the upper eight bits of the unsigned short integer to be written into the register SPI2-DR, and this is accomplished by shifting it to the left for eight places. However, the MSB of the shifted value must he high to signal reading from the SPI slave, and this is achieved by OR-ing the result of shifting with 0x8000.

The value obtained from the SPI slave is available in the SPI2->DR register, and only lower eight bits are to be used; those are isolated by an AND function and argument of 0xff.

Once we have functions to write and read the SPI bus we are ready to see the program to communicate with the accelerometer and pass results on the screen. The listing is given below.

```
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_spi.c"
#include "LCD2x16.c"
#include "dd.h"

int main ()  {
  SPI2init();                                  // SPI2 init                                // 8
  LCD_init();                                  // Init LCD
  LCD_string("SPI demo Z", 0);                 // write title
  LCD_string("X        Y", 0x40);              // write title

  LIS_write (0x20,0xc7);                                                                   // 13

  while (1) {
    short xLSB = LIS_read(0x28); short xMSB = LIS_read(0x29) << 8;                         // 16
    short yLSB = LIS_read(0x2a); short yMSB = LIS_read(0x2b) << 8;                         // 17
    short zLSB = LIS_read(0x2c); short zMSB = LIS_read(0x2d) << 8;                         // 18
    LCD_sInt16((xLSB + xMSB), 0x41, 0x01);      // show acc x                              // 19
    LCD_sInt16((yLSB + yMSB), 0x4a, 0x01);      // show acc y                              // 20
    LCD_sInt16((zLSB + zMSB), 0x0a, 0x01);      // show acc z                              // 21
    for (int i = 0; i < 10000000; i++)   {};    // waste time
  };
}
```

The listing starts with the inclusion of necessary source files, and the execution starts with the function “main()”. A function to configure the SPI2 block is called in line 8, the content of this function will be discussed last. The function for the configuration and initialization of the LCD screen is called next, followed by two commands to write introductory strings to the LCD.

The accelerometer sensor is turned on and configured by a single write in line 13. As in the previous chapter: value 0xc7 must be written to sensor at address 0x20 in order for sensor to operate.

The execution proceeds with the loop, where the content of six registers within the sensor are read periodically using the derived function for reading the SPI bus, these are lines 16 to 18. Next the results obtained are combined and written to the LCD screen at appropriate positions.

The loop terminates with an empty ‘for’ loop to waste some time and to reduce the frequency of results being written to the LCD.

Lines 16 to 18 need some discussion. The datasheet of the accelerometer specifies that the result of the measurement is available in registers 0x28 (LSB) and 0x29 (MSB) for the acceleration in x-axis. Registers 0x2a (LSB) and 0x2b (MSB) hold the result for y-axis, and registers 0x2c (LSB) and 0x2d (MSB) hold the result for z-axis. These bytes are read from the relevant registers one by one and stored in six short integers ‘xLSB’, ‘xMSB’, ‘yLSB’... These integers are then combined to form complete results for accelerations in each of the three axes.

The configuration procedure for the SPI block 2 is discussed last. Its listing is given below, and it consists of two parts. The main part is called “SPI2init()”, and is used to configure the SPI block. However, the SPI signals must be assigned to pins on the microprocessor, and this is done in the function “GPIOBinit_SPI2()” which is called from the “SPI2init()”.

```
void GPIOBinit_SPI2(void)   {
GPIO_InitTypeDef          GPIO_InitStructure;
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);                                    // 3
  GPIOB->BSRRL = BIT_14 | BIT_13 | BIT_12 | BIT_11;                                        // 5

  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;                   // 7
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF;                                              // 8
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;                                          // 9
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;                                             // 10
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;                                          // 11
  GPIO_Init(GPIOB, &GPIO_InitStructure);                                                     // 12

  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_11 | GPIO_Pin_12; // PB12, PB12: SPI_SEL          // 14
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_OUT;                                             // 15
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_UP;                                              // 16
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;                                             // 17
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;                                          // 18
  GPIO_Init(GPIOB, &GPIO_InitStructure);                                                     // 19

  GPIO_PinAFConfig(GPIOB, GPIO_PinSource13, GPIO_AF_SPI2);  // PB13:SPI2_SCK                 // 21
  GPIO_PinAFConfig(GPIOB, GPIO_PinSource14, GPIO_AF_SPI2);  // PB14:SPI2_MISO                // 22
  GPIO_PinAFConfig(GPIOB, GPIO_PinSource15, GPIO_AF_SPI2);  // PB15:SPI2_MOSI                // 23
}

void SPI2init(void) {       //
SPI_InitTypeDef   SPI_InitStructure;
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2,  ENABLE);                                      // 3
  GPIOBinit_SPI2();     // GPIO clock enable, digital pin definitions                        // 4

  SPI_InitStructure.SPI_Direction       = SPI_Direction_2Lines_FullDuplex;                   // 6
  SPI_InitStructure.SPI_Mode            = SPI_Mode_Master;                                   // 7
  SPI_InitStructure.SPI_DataSize        = SPI_DataSize_16b;                                  // 8
  SPI_InitStructure.SPI_CPOL            = SPI_CPOL_High;                                     // 9
  SPI_InitStructure.SPI_CPHA            = SPI_CPHA_2Edge;                                    // 10
  SPI_InitStructure.SPI_NSS             = SPI_NSS_Soft;                                      // 11
  SPI_InitStructure.SPI_BaudRatePrescaler       = SPI_BaudRatePrescaler_64;                  // 12
  SPI_InitStructure.SPI_FirstBit        = SPI_FirstBit_MSB;                                  // 13
  SPI_InitStructure.SPI_CRCPolynomial   = 7;                                                 // 14
  SPI_Init(SPI2, &SPI_InitStructure);                                                        // 15

  SPI_Cmd(SPI2, ENABLE);                                                                     // 17
}
```

The configuration of the SPI block is initiated by the call of the function “SPI2init()”. The function “SPI_Init()” from a CMSIS library will be used for the configuration as described in the source file “stm32f4xx_spi.c”. This function requires the use of the data structure ‘SPI_InitStructure’, which is defined in the header file “stm324xx_spi.h”, lines 54 to 85.

The clock for the block SPI2 is enabled in line 3, then the function that deals with pin configuration at port B is called; this will be discussed later. The data structure is initialized in lines 6 to 14 as follows.

- Member ‘.SPI_Direction’ defines the number of wires used for the transmission and the direction of data transfer, see the Reference manual RM0090, pg. 863 for details. We are going to use three lines, one for the clock signal, one to send data from the microcontroller to the accelerometer chip, and one to receive data into the microcontroller from the accelerometer chip. The synonymous for this in CMSIS is the keyword ‘SPI_Direction_2Lines_FullDuplex’. Alternative keywords are listed in the header file, lines 151 to 154.

- Member '.SPI_Mode' defines the mode of operation for the block SPI; it can be either master or slave. In our case the microcontroller is taking the complete control over the SPI bus, and this is accomplished by making the SPI block the master of the bus.
- Member '.SPI_DataSize' defines the length of the message sent in one package. It can be either 8 or 16 bits, here we opt for 16 bits by initializing the member with the keyword 'SPI_DataSize_16b'.
- Members '.SPI_CPOL' and '.SPI_CPHA' define the relation of the polarity of the clock signal SCL and the edge of this signal to be used for clocking of data, see the details in the Reference manual RM0090, pages 857 and 858. The sensor used in our experiment requires signals as given in Fig. 18.2, and this is accomplished by initializing these two members as 'SPI_CPOL_High' and 'SPI_CPHA_2Edge'.
- Member '.SPI_NSS' defines the generation of the slave select signal. We intend to generate this signal by software, and this is accomplished by initializing this member with 'SPI_NSS_Soft'.
- Member '.SPI_BaudRatePrescaler' defines the speed of transmission. We arbitrary select one of the division ratios in this experiment as 'SPI_BaudRatePrescaler_64'. Note that the slave select signal is managed by software and a software delay is used to align the select signal. Using different value for the prescaler requires the change in the delay loops as well.
- Member '.SPI_FirstBit' defines the order the bits are sent to the slave. The chain can start with the MSB or the LSB, and the accelerometer chip used in this experiment requires MSB to be sent first.
- Member '.SPI_CRCPolynomial' is not used here since we do not use any error checking. Nevertheless, this member is initialized to 7.

Following the initialization of data structure the block gets configured by a call in line 15. The configuration is followed by a call in line 17 that enables the block SPI for the use.

The complete list of special functions that can be assigned to pins is given in table 9, pg. 60 and on, in the datasheet of the microcontroller (DM00037051). Following this table the SPI signals for block 2 are available at port B, pins 12 to 15; these pins need to be configured before use, and this is done in function "GPIOBinit_SPI2". The function starts with the definition of the data structure and enabling of the clock for port B. Since the initial state of all lines is supposed to be high all pins are initialized to logic high in line 5.

Pins 13, 14 and 15 are used for SPI signals SCK, MISO and MOSI respectively. These pins are designated as outputs (lines 7 to 12), and alternate functions are selected for them (lines 21 to 23). The rest of the initialization of the data structure and function call itself follow the usual pattern for pin configuration.

There are two slave select pins 11 and 12; pin 12 is used in this example since only one of the SPI connectors is utilized. These pins are designated as outputs in lines 14 to 19.

Finally the alternate function for pins 13 to 15 is selected in lines 21 to 23.