SPI

Communication

Protocol

(STM32F401)

Team 7	
Mohamed Magdy	20011701
Walaa AbdElaziz	20012193
Manar Fawzy	20011989
Menna Mohamed	20012011
Nansy Mohamed	20012081
Eslam Elsayed	20010292
Abanob samir	20010002
Mahmoud Hesham	20011836

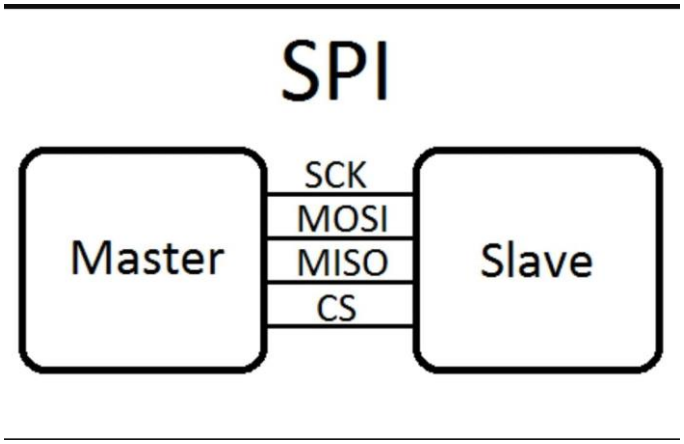
Chapter (1)

Introduction:

the Serial Peripheral Interface (SPI) is a synchronous serial communication protocol used to exchange data between microcontrollers and peripheral devices, such as sensors, memory chips, display controllers, and more. The STM32F401 is a microcontroller from STMicroelectronics that supports SPI communication.

Features :

Hardware support: The STM32F401 microcontroller has multiple SPI interfaces (SPI1, SPI2, etc.) that support full-duplex communication.



Configurability :

-configurable as a master or slave device.

-Configurable data frame size (8-bit, 16-bit, or 32-bit)

-Configurable clock polarity and phase (CPOL and CPHA) for compatibility with various SPI devices

Clock control :

-Configurable clock prescaler to adjust the SPI clock frequency.

-Support for various clock sources, allowing flexibility in setting the clock speed.

Data handling:

-FIFO (First In, First Out) buffers for both transmission and reception.

-Support for simultaneous transmit and receive operations.

GPIO_PINS:

Flexible GPIO pin configuration to assign SPI signals to specific pins (SCK, MISO, MOSI) based on the chosen SPI interface.

Power management:

power-saving modes to reduce energy consumption when SPI is idle.

Peripheral support:

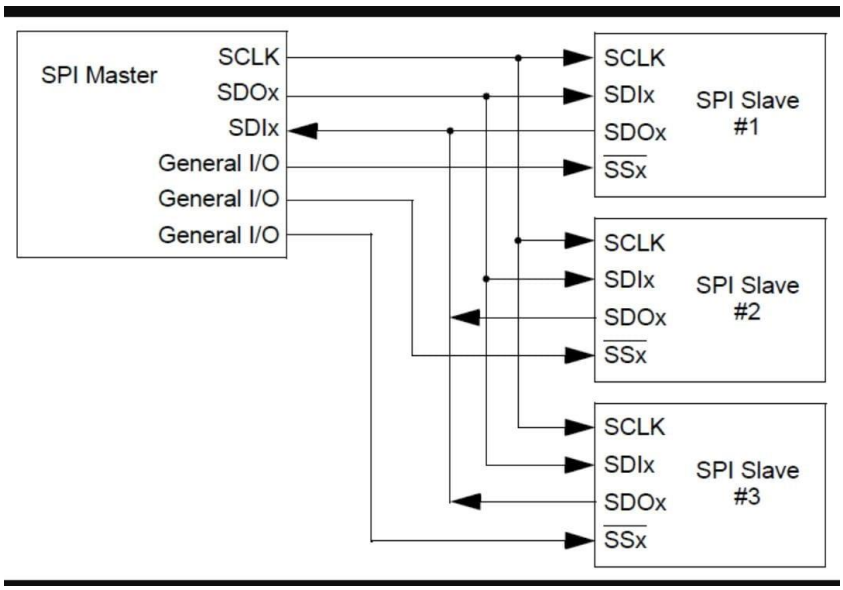
Compatibility with a wide range of SPI-compatible devices such as sensors, displays, memory chips, and other microcontroller.

Throughput: the STM32F401 microcontroller supports SPI communication. Its SPI peripheral can operate at different clock speeds, typically configurable up to a few tens of MHz or more, depending on the specific STM32F401 variant and the system configuration.

Throughput = (SPI Clock Frequency) / (Number of bits per transfer)

Software and hardware slave selection :

In STM32 microcontrollers, such as the STM32F401, you can communicate using the Serial Peripheral Interface (SPI) protocol by directly manipulating the registers without relying on the Hardware Abstraction Layer (HAL) provided by STMicroelectronics



overrun conditions in the SPI protocol might occur for various reasons:

-data over writing .

-interrupt or DMA handling

-FIFO overflow

CRC and ethernet polynomial

for SPI communication with CRC in STM32F401, you typically use a polynomial for CRC calculation. However, the specific polynomial used can vary based on the requirements of your application. The polynomial is often defined in terms of its bit representation

For Ethernet CRC calculation, the commonly used polynomial is often represented as 0x04C11DB7 for Ethernet CRC-32. However, it's essential to note that this polynomial might not directly correlate to the SPI protocol in STM32F401.

the CRC polynomial used for SPI in the STM32F401 and its configuration registers are defined in the STM32F401 reference manual provided by STMicroelectronics. You should refer to the "SPIx_CR1" and "SPIx_CRCPOLY" registers in the reference manual to find the exact polynomial and CRC-related configurations.

The frame structure in SPI typically consists of the following components:

1. Clock Polarity (CPOL): Determines the idle state of the clock signal (high or low).

2. Clock Phase (CPHA): Defines when the data is sampled or changed concerning the clock signal.

3. Data Size: Indicates the number of bits per transfer.

4. Master/Slave Configuration: Specifies the device as either a master or a slave.

5. Bit Order: Defines whether the Most Significant Bit (MSB) or the Least Significant Bit (LSB) is transmitted first.

6. Chip Select (CS): Used to select the device the STM32 is communicating with in a multi-device setup.

Relation between SPI and I2S in STM32F401:

• While SPI and 12s are both serial communication protocols, they serve different purposes.

• In some microcontrollers, including the STM32 series, there might be hardware modules that support both SPI and I2S communication. However, they are distinct protocols designed for different types of applications.

In the STM32F401 microcontroller, the SPI peripheral modules can be used for general-purpose synchronous serial communication with various devices, while the 125 peripheral is more suitable for high-quality audio data transmission.

It's essential to configure the STM32F401's hardware peripherals and pins correctly to use either SPI or 125 communication based on the requirements of your specific application, as these protocols have different configurations and functionalities.

If you need to use SPI or I2S in your STM32F401 project, you'll typically configure the appropriate peripheral and pins, set the desired parameters (like clock speed, data format, etc.), and handle the data transmission/reception accordingly. Both protocols require distinct configurations and handling due to their specific purposes.

common ICs that utilized the spi protocol:

1. Microcontrollers and Microprocessors: Many microcontrollers and microprocessors, such as those from Atmel (now Microchip), STMicroelectronics, Texas Instruments, and others, often have SPI interfaces for communication with peripheral devices.

2. Sensors: Accelerometers, gyroscopes, temperature sensors, pressure sensors, and other types of sensors often use SPI for interfacing with microcontrollers or other main processing units.

3. Display Drivers: ICs used in TFT LCDs (Thin Film Transistor Liquid Crystal Displays), OLED (Organic Light Emitting Diode) displays, and other types of displays often employ SPI for communication.

4. Memory Devices: Serial Flash memories, EEPROMs (Electrically Erasable Programmable Read- Only Memory), and some types of FRAM (Ferroelectric RAM) use SPI for interfacing with the main controller.

5. Communication Modules: Various communication modules like Wi-Fi modules, Bluetooth modules, Ethernet controllers, and RF transceivers utilize SPI for communication with the main system.

6. Real-Time Clocks (RTCS): Some RTC ICs use SPI to communicate with microcontrollers or other control systems to provide accurate timekning functions.

↓ 7. Digital-to-Analog Converters (DACs) and alog-to-Digital Converters (ADCS): Some DACs and ADCs use SPI for configuring settings and transferring data between the IC and the controlling system

8. Motor Drivers: Certain motor driver ICs and controllers incorporate SPI for configuration and control purposes.

These are just a few examples; the SPI protocol is versatile and is employed in a wide range of ICs across different industries and applications due to its simplicity, flexibility, and efficiency in serial data transmission.

Chapter (2)

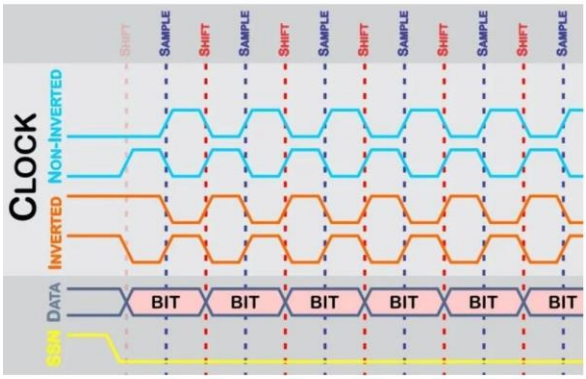
1. SPI modes of transmission:-

*Clock Polarity and Phase.

Clock transitions govern the shifting and sampling of data. SPI has four modes (0,1,2,3) that correspond to the four possible clocking configurations.

Each transaction begins when the slave-select line is driven to logic low (slave select is typically an active-low signal). The exact relationship between the slave-select, data, and clock lines depends on how the clock polarity (CPOL) and clock phase (CPHA) are configured. With non-inverted clock polarity (i.e., the clock is at logic low when slave select transitions to logic low):

• **Mode 0**: Clock phase is configured such that data is sampled on the rising edge of the clock pulse and shifted out on the falling edge of the clock pulse. This corresponds to the first blue clock trace in the above diagram. Note that data must be available before the first rising edge of the clock.



Bits that are sampled on the rising edge of the clock cycle are shifted out on the falling edge of the clock cycle, and vice versa.

• **Mode 1**: Clock phase is configured such that data is sampled on the falling edge of the clock pulse and shifted out on the rising edge of the clock pulse. This corresponds to the second blue clock trace in the above diagram.

SPI Mode	Clock Polarity	Clock Edge/Phase	
	CPOL/CKP	CPHA	NCPHA/CKE
0	0	0	1
1	0	1	0
2	1	0	1
3	1	1	0

• **Mode 2**: Clock phase is configured such that data is sampled on the falling edge of the clock pulse and shifted out on the rising edge of the clock pulse. This corresponds to the first orange clock trace in the above diagram. Note that data must be available before the first falling edge of the clock.

• **Mode 3**: Clock phase is configured such that data is sampled on the rising edge of the clock pulse and shifted out on the falling edge of the clock pulse. This corresponds to the second orange clock trace in the above diagram.

2. SPI transaction handling modes:- polling , interrupts and DMA

* Interrupt Transactions:-

_ Interrupt transactions blocks the transaction routine until the transaction completes, thus allowing the CPU to run other tasks.

_ An application task can queue multiple transactions, and the driver automatically handles them one by one in the interrupt service routine (ISR). It allows the task to switch to other procedures until all the transactions are complete.

* Polling Transactions:-

✓• Polling transactions do not use interrupts. The routine keeps polling the SPI Host's status bit until the transaction is finished.

✓• be blocked by the queue. At this point, they need to wait for the ISR to run twice before the transaction is finished. Polling transactions save time otherwise spent on queue handling and context switching, which results in smaller transaction duration. The disadvantage is that the CPU is busy while these transactions are in progress.

✓• The spi_device_polling_end() routine needs an overhead of at least 1 µs to unblock other tasks when the transaction is finished. It is strongly recommended to wrap a series of polling transactions using the functions spi_device_acquire_bus() and spi_device_release_bus() to avoid the overhead. For more information.

3. SPI module block diagram brief explanation

SCK or SCLK : Serial Clock, clock generated by master.

MOSI or SDO : Master Output Slave Input, data output from master.

MISO or SDI : Master Input Slave Output, data output from slave.

SS : Slave Select, often an active low input of a slave.

When multiple slave devices are used, an independent slave select line is connected from master to each slave device as below.

Master will select only one slave at a time. Mostly slave devices will be equipped with tri-state outputs. So when they are not selected, their output lines appears disconnected.

Clock Polarity (CKP) = 0

This means that the base value of clock is zero. Which implies idle state is 0 and active state is 1.

*Clock Edge (CKE) = 0

Data transmission. occurs during idle to active clock state, ie LOW to HIGH transition.

*Data transmission occurs during active to idle clock state, ie HIGH to LOW transition.

Clock Polarity (CKP) = 1

This means that the base value of clock is one. Which implies idle state is 1 and active state is 0.

*Clock Edge (CKE) = 0

Data transmission occurs during idle to active clock state, ie HIGH to LOW transition.

*Clock Edge (CKE) = 1

Data transmission occurs during active to idle clock state, ie LOW to HIGH transition.

Spi control registers:

SPI Control Register 1

	7	6	5	4	3	2	1	0
$00D0	SPIE	SPE	SWOM	MSTR	CPOL	CPHA	SSOE	LSBF

Fig.3: SPI Control Register 1, SP0CR1

SPIE: Serial Peripheral Interrupt Enable

SPE: Serial Peripheral System Enable

SWOM: Port S Wired-OR Mode (affects PS[4:7] pins)
0–Normal CMOS outputs 1–Open-drain outputs

MSTR: Master/Slave Mode Select 0–Slave mode 1–Master mode

CPOL: Clock Polarity

CPHA: Clock Phase

SSOE: Slave Select Output Enable (master mode only)

LSBF: Least-Significant Bit Enable

SPI Control Register 2

	7	6	5	4	3	2	1	0
$00D1	0	0	0	0	PUPS	RDS	0	SPC0

Fig.5: SPI Control Register 2, SP0CR2

PUPS: Pull-Up Port S Enable
0 – No internal pull-ups on port S
1 – All port S input pins have an active pull-up device

RDS: Reduce Drive of Port S
0 – Normal port S output drivers
1 – Reduced drive capability on all port S outputs for lower power and less noise

SPC0: Serial Pin Control 0
0 – Normal operation
1 – Bidirectional mode

Spi status registers:

SPI Status Register

	7	6	5	4	3	2	1	0
$00D3	SPIF	WCOL	0	MODF	0	0	0	0

Fig.4: SPI Status Register, SP0SR

SPIF: SPI Transfer Complete Flag
0 – Cleared by SP0SR read with SPIF set, followed by SP0DR access
1 – Set upon completion of data transfer between processor and external device

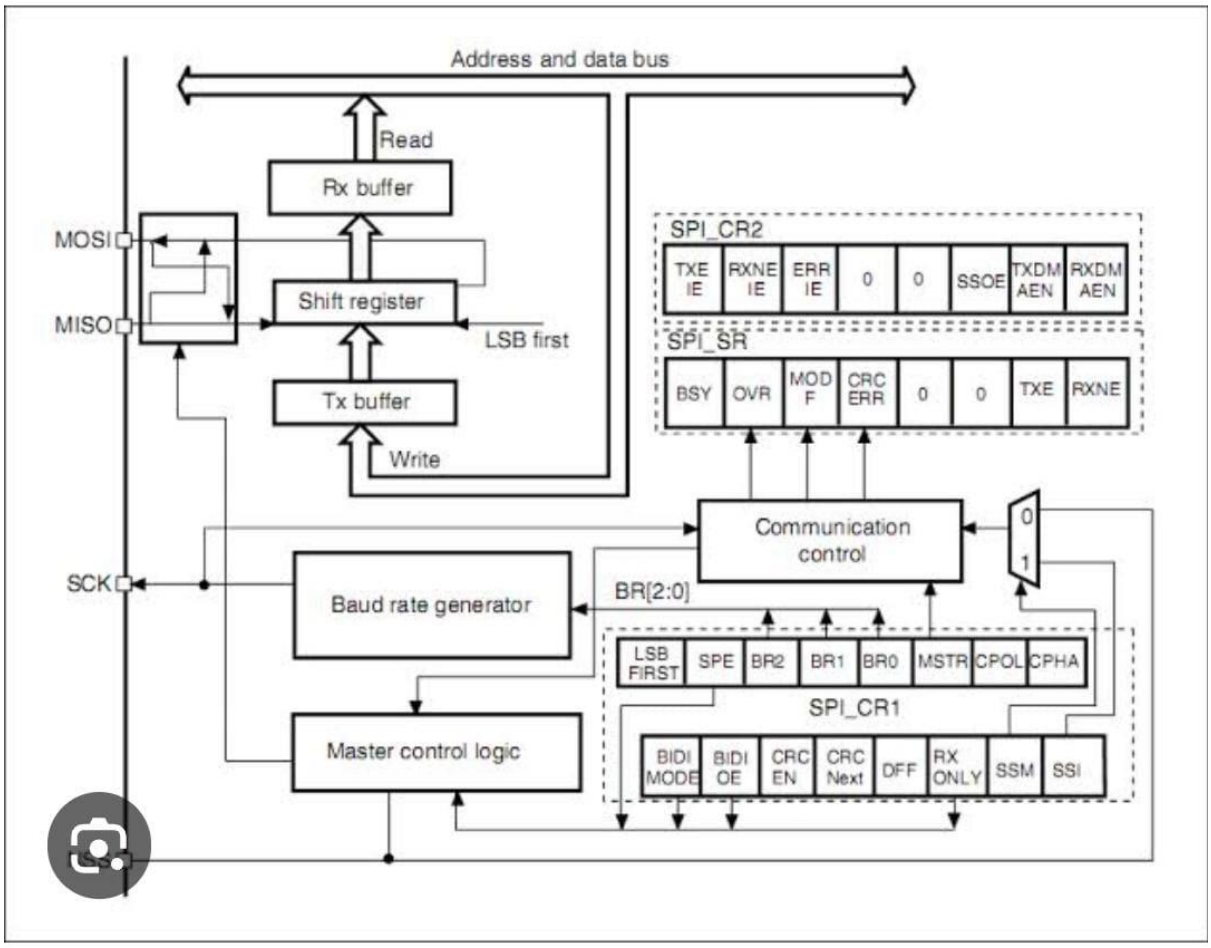
WCOL: Write Collision
0 – No write collision
1 – Write collision

MODF: Mode Fault
0 – No mode fault
1 – Mode fault

Registers or presale of frequency

SPR[2:0]	Divisor	Frequency (*E*=8 MHz)
000	2	4.0 MHz
001	4	2.0 MHz
010	8	1.0 MHz
011	16	500 KHz
100	32	250 KHz
101	64	125 KHz
110	128	62.5 KHz
111	256	31.25 KHz

The architecture:



void begin(void)

Description:

This function initializes the microcontroller, system clock, peripherals, and communication interfaces. It is typically called in the main function of the program before starting the main loop.

Parameters: None.

Return value: None.

Source:

```
void begin ()

{ enable_SPI _clock();
  SPI_GPIO_config();
  slaveManagementMode();
  SPI_enable();
}
// USED FUCTIONS
/*
enable_SPI _clock(){
RCC->APB2ENR |= (1u<<12) // enable clk of spi1
}
SPI_GPIO_config(){
// Configure GPIO pins for SPI
   GPIOA->MODER |= GPIO_MODER_MODER5_1 |
GPIO_MODER_MODER6_1 | GPIO_MODER_MODER7_1;  // AF
mode for SCK, MISO, MOSI
   GPIOA->AFR[0] |= (0x5 << 20) | (0x5 << 24) | (0x5 << 28);
// AF5 for SPI1}




slaveManagementMode(){
```

```
  SPI1->CR1 = SPI_CR1_MSTR | SPI_CR1_SSM | SPI_CR1_SSI;
// Master mode, software slave management
}
SPI_enable()
{
   SPI1->CR1 |= SPI_CR1_SPE;}
*/
```

void setDataMode (int mode, int clock)

Description:

This function configures the serial data mode and clock mode for the SPI device.

Parameters:

mode: a member of the enumeration spiDataMode. The enumeration members define the serial data mode as follows:

SPI_MODE_0: The data is captured on the rising edge of the clock and data is propagated on the falling edge of the clock.

SPI_MODE_1: The data is captured on the falling edge of the clock and data is propagated on the rising edge of the clock.

SPI_MODE_2: The data is captured on the falling edge of the clock anddata is propagated on the rising edge of the clock.

SPI_MODE_3: The data is captured on the rising edge of the clock and data is propagated on the falling edge of the clock.

clockMode: a member of the enumeration spiClockMode. The enumeration members define the clock mode as follows:

Return value: None.

Source:

```
void setDataMode(int mode, int clockMode){

  if(mode == 8)
 clear_bit(SPI1_CR1 ,11) // clear bit 11
     else if (mode == 16)
set_bit(SPI1_CR1 ,11);  // set bit 11

     switch (clockMode)
     {
```

```
      case 0:
       clear_bit(SPI1_CR1 , 1); CPOL
       clear_bit(SPI1_CR1 , 0); // (CPHA)
      break;

      case 1:
       clear_bit(SPI1_CR1 , 1);  // CPOL
       set_bit(SPI1_CR1 , 0); // (CPHA)
      break;

      case :
       set_bit(SPI1_CR1 , 1); // CPOL
       clear_bit(SPI1_CR1 , 0); // (CPHA)
      break;

      case 3:
       set_bit(SPI1_CR1 ,1); // (CPOL)
       set_bit(SPI1_CR1 ,0); // (CPHA)
      break;
     }
  }
```

void setClockDivider(unsigned int divider)

Description: This function sets the clock divider value for the selected frequency. The divider value is used to scale the frequency of the main system clock.

Parameters: freq_divider: the desired frequency divider value. The main system clock will be divided by this value.

Return value: NONE

Source:

```
 void setClockDivider(unsigned int divider){
switch (divider)
      {
       case 2:              // setprescaler to 2
        clear_bit(SPI1_CR1 ,3);
        clear_bit(SPI1_CR1 ,4);
        clear_bit(SPI1_CR1 ,5);
       break;

       case 4:       // setprescaler to 4
        set_bit(SPI1_CR1 ,3);
        clear_bit(SPI1_CR1 ,4);
        clear_bit(SPI1_CR1 ,5);
       break;

       case 8:                    // setprescaler to 8
        clear_bit(SPI1_CR1 ,3);
        set_bit(SPI1_CR1 ,4);
        clear_bit(SPI1_CR1 ,5);
       break;

       case 16:                   // setprescaler to 16
        set_bit(SPI1_CR1 ,3);
        set_bit(SPI1_CR1 ,4);
```

```
        clear_bit(SPI1_CR1 ,5);
       break;

       case 32:                  // setprescaler to 32
        clear_bit(SPI1_CR1 ,3);
        clear_bit(SPI1_CR1 ,4);
        set_bit(SPI1_CR1 ,5);
       break;

       case 64:                  // setprescaler to 64
        set_bit(SPI1_CR1 ,3) ;
      clear_bit(SPI1_CR1 ,4);
        set_bit(SPI1_CR1 ,5);
       break;

       case 128:                    // setprescaler to 128
        clear_bit(SPI1_CR1 ,3);
        set_bit(SPI1_CR1 ,4);
        set_bit(SPI1_CR1 ,5);
       break;

       case 256:                  // setprescaler to 256
        set_bit(SPI1_CR1 ,3);
        set_bit(SPI1_CR1 ,4);
        set_bit(SPI1_CR1 ,5);
       break;

     }
  }
```

void setBitOrder(int byteOrder)

Description:

The setBitOrder(byteOrder) function sets the bit order used by the shift registers. The available options are either MSBFIRST (most significant bit first) or LSBFIRST (least significant bit first).

Parameters:

byteOrder: the desired bit order, either MSBFIRST or LSBFIRST.

Return value: none

Source:

```
void setBitOrder(int byteOrder){
    switch (byteOrder)
    {
      case 0:
       set_bit(SPI1_CR1 ,7); //LSB transmitted first
      break;
      case 1:
       clear_bit(SPI1_CR1 ,7); //MSB transmitted first
       break;
    }
  }
```

int transfer(int data)

Description:

The transfer(data) function transfers a given amount of data between two internal devices. The data can be transferred in different directions based on the value of the direction parameter.

Parameters:

data: This parameter specifies the data to be transferred. It can be an array of bytes or a pointer to the memory location containing the data.

Return Value: integer

Source:

```
int transfer(int data)
 { int receivedData ;
   set_bit(SPI1_CR1 ,10);

   while (~(SPI1_SR & (1 << 1))) {/* Wait until transmit buffer is
empty (TXE flag set)*/}
   while ((SPI1_SR & (1 << 7))) {/* Wait fot busy bit to reset */}

   // Write data to be sent into the data register
   SPI1_DR = data;

   // Wait until RX buffer is full (wait for receive data)
   while (!(SPI1_SR & (1 << 0))) {/* Wait until receive buffer is full
(RXNE flag set)*/}
   while ((SPI1_SR & (1 << 7))) {/* Wait fot busy bit to reset */}
```

```
    // Read received data from data register
     receivedData = SPI1_DR;

    // Return received data
    return receivedData ;
}
```

void end(void)

Description:

This function initializes the microcontroller, system clock, peripherals, and communication interfaces. It is typically called in the main function of the program before starting the main loop.

Parameters: None.

Return value: None.

Source:

```
void end() {
   // Wait until RXNE = 1 to receive the last data
   while (!(SPI1_SR & (1 << 0))) {/* Wait until RXNE flag is set (last
data received)*/}

   // Wait until TXE = 1
   // Wait until BSY = 0
   while (SPI1_SR & (1 << 7)) {/* Wait until BSY flag is cleared (SPI
not busy)*/}
  // Disable SPI (SPE = 0)
    clear_bit(SPI1_CR1 ,6);

   // Disable peripheral clock
   clear_bit(RCC_APB2ENR ,12); // Disable SPI1 clock
}
```

void beginTransaction(int byteOrder,int dataMode,unsigned int baudRate, int clockMode)

Description:

This function completes the current transaction by either committing the changes or rolling back the transaction.

Parameters:

-Byte order

- dataMode

- baudRate

- clockMode

.

Return value: none

Source:

```
void beginTransaction(int byteOrder,int dataMode,unsigned int
baudRate, int clockMode){


 //enable SPI1 clock
     set_bit(RCC_APB2ENR ,12);
 // Set byte order (MSB or LSB first)
     setBitOrder(byteOrder);

 // set data mode (8_bit or 16_bit mode) and clock mode
```

```
    setDataMode(dataMode,clockMode)

   //setting baud rate
    setClockDivider(baudRate); // Set BR bits to divide the clock

}
```

void endTransaction(void)

Description:

This function ends the transaction with the slave device and resets the bus to idle state.

Parameters: None.

Return value: None.

Source:

```
void endTransaction(){
    // Wait until RXNE = 1 to receive the last data
   while (!(SPI1_SR & (1 << 0))) {/* Wait until RXNE flag is set (last data received)*/}

   // Wait until TXE = 1
   while (!(SPI1_SR & (1 << 1))) {/* Wait until TXE flag is set*/}

   // Wait until BSY = 0
   while (SPI1_SR & (1 << 7)) {/* Wait until BSY flag is cleared (SPI not busy)*/}
  // Disable SPI (SPE = 0)
    clear_bit(SPI1_CR1 ,6);

 }
```

CHAPTER(4)

Simulation: