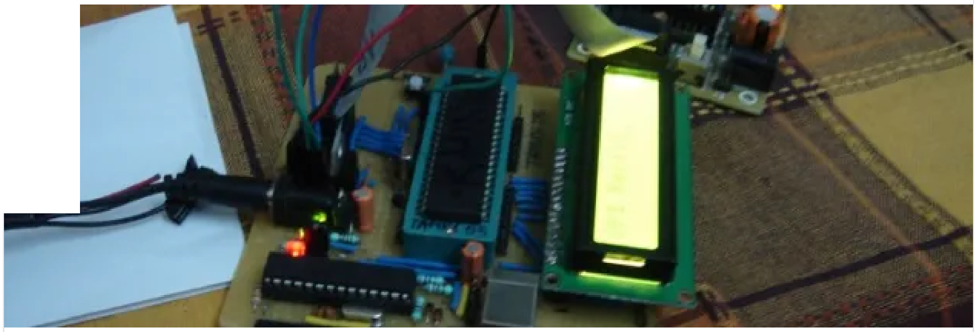
sted by Yash on Nov 26, 2013 in Atmel AVR, Microcontrollers | 97 comments



# The SPI of the AVR

Continuing with the series of tutorials on Serial Communication, here is another one, and much awaited, the Serial Peripheral Interface (SPI) of AVR! Before proceeding ahead, I would suggest you to read Mayank's tutorial on the basics of SPI.

## Contents

- SPI – Basics Revisited
  - Advantages of SPI
  - Master and Slave
  - Pin Description

Search maxEmbedded

Search

Popular Recent Random

MON 20 The ADC of the AVR
Posted by Mayank in Atmel AVR, Microcontrollers

FRI 24 AVR Timers – TIMER0
Posted by Mayank in Atmel AVR, Microcontrollers

TUE 06 RF Module Interfacing without Microcontrollers
Posted by Mayank in Electronics

THU 16 LCD Interfacing with AVR
Posted by Mayank in Atmel AVR, Microcontrollers

FRI 10 I/O Port Operations in AVR
Posted by Mayank in Atmel AVR, Microcontrollers

Browse maxE by Categories

- Register Descriptions
  - SPCR, SPSR, SPDR
- Data Modes
- Slave Select
- SPI Coded!
  - Enabling SPI on Master
  - Enabling SPI on Slave
  - Sending and Receiving Data
- Problem Statement
  - Methodology
  - Hardware Connections
  - Full Code
  - Video
- Using Interrupts
- Summary

## Serial Peripheral Interface (SPI) – Basics Revisited

Here we will discuss some basics of Serial Peripheral Interface (SPI, pronounced *spy* or *ess-pee-eye*). Mayank has already dealt with the basics of SPI and SPI bus transactions in the previous tutorial, but I will go over some of the nitty-gritties here again.

Serial Peripheral Interfacing is one of the most used serial communication protocols, and very simple to use! As a matter of fact, I find this one much simpler than USART! ;)

Since SPI has been accepted as a *de facto* standard, it is available in almost all architectures, including 8051, x86, ARM, PIC, AVR, MSP etc., and is thus widely used. This means that there shouldn't be any portability issues and you can connect devices of two different architectures together as well!

So here are the most popular **applications of SPI**:

1. Wired transmission of data (though the first preference is mostly USART, but SPI *can* be used when we are using multiple slave or master systems, as addressing is much simpler in SPI).
2. Wireless transmissions through Zigbee, 2.4GHz etc.
3. Programming your AVR chips (Yes! They are programmed through the SPI! You'll would have read about it in Mayank's Post on SPI).
4. It is also used to talk to various peripherals – like sensors, memory devices, real time clocks, communication protocols like Ethernet, etc.

### Advantages of SPI

SPI uses 4 pins for communications (which is described later in this post) while the other communication protocols available on AVR use lesser number of pins like 2 or 3. Then why does one use SPI? Here are some of the advantages of SPI:

transmit data through SPI as compared to I2C and UART!)
2. Full duplex communication
3. Less power consumption as compared to I2C
4. Higher hit rates (or throughput)

And a lot more!

But there are some disadvantages as well, like higher number of wires in the bus, needs more pins on the microcontroller, etc.

## Master and Slave

In SPI, every device connected is either a *Master* or a *Slave*.

The *Master* device is the one which initiates the connection and controls it. Once the connection is initiated, then the *Master* and one or more *Slave(s)* can transmit and/or receive data. As mentioned earlier, this is a full-duplex connection, which means that *Master* can send data to *Slave(s)* and the *Slave(s)* can also send the data to the *Master* at the same ime.

Shares

As I said earlier, SPI uses 4 pins for data communication. So let's move on to he pin description.

## Pin Description

The SPI typically uses 4 pins for communication, wiz. MISO, MOSI, SCK, and SS. These pins are directly related to the SPI bus interface.

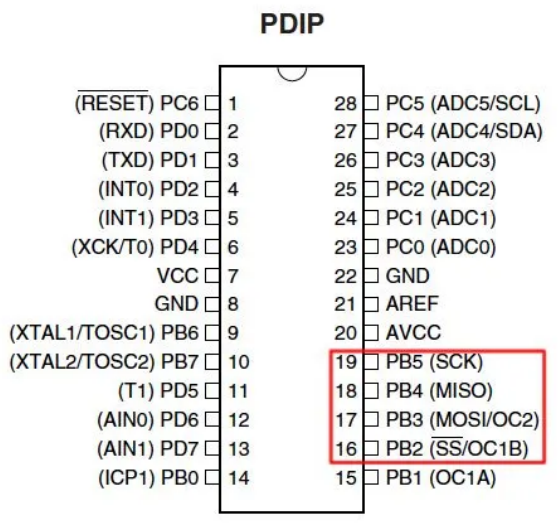
1. **MISO** – MISO stands for Master In Slave Out. MISO is the input pin for *Master* AVR, and output pin for *Slave* AVR device. Data transfer from *Slave* to *Master* takes place through this channel.
2. **MOSI** – MOSI stands for Master Out Slave In. This pin is the output pin for *Master* and input pin for *Slave*. Data transfer from *Master* to *Slave* takes place through this channel.
3. **SCK** – This is the SPI clock line (since SPI is a synchronous communication).
4. **SS** – This stands for *Slave Select*. This pin would be discussed in detail later in the post.

Now we move on to the SPI of AVR!

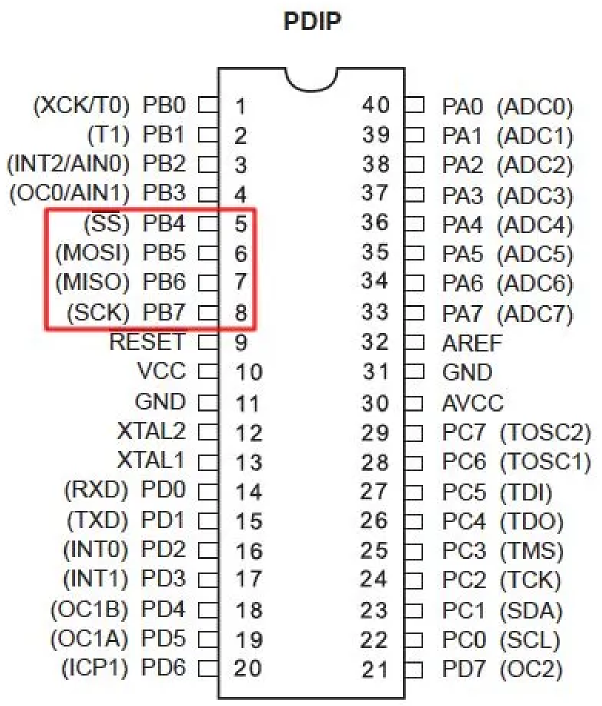
## The SPI of the AVR

The SPI of AVRs is one of the most simplest peripherals to program. As the AVR has an 8-bit architecture, so the SPI of AVR is also 8-bit. In fact, usually the SPI bus is of 8-bit width. It is available on PORTB on all of the ICs, whether 28 pin or 40 pin.

Some of the images used in this tutorial are taken from the AVR datasheets.



SPI pins on 28 pin ATmega8



SPI pins on 40 pin ATmega16/32

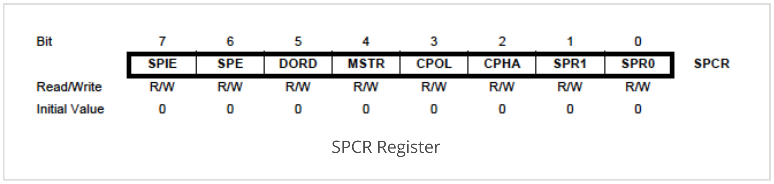
## Register Descriptions

The AVR contains the following three registers that deal with SPI:

1. **SPCR – SPI Control Register** – This register is basically the master register i.e. it contains the bits to initialize SPI and control it.
2. **SPSR – SPI Status Register** – This is the status register. This register is used to read the status of the bus lines.
3. **SPDR – SPI Data Register** – The SPI Data Register is the read/write register where the actual data transfer takes place.

## The SPI Control Register (SPCR)

As is obvious from the name, this register controls the SPI. We will find the bits that enable SPI, set up clock speed, configure master/slave, etc. Following are the bits in the SPCR Register.



| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | SPIE | SPE | DORD | MSTR | CPOL | CPHA | SPR1 | SPR0 | SPCR |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

SPCR Register

**Bit 7: SPIE – SPI Interrupt Enable**
The SPI Interrupt Enable bit is used to enable interrupts in the SPI. Note that global interrupts must be enabled to use the interrupt functions. Set this bit to '1' to enable interrupts.

**Bit 6: SPE – SPI Enable**
The SPI Enable bit is used to enable SPI as a whole. When this bit is set to 1, the SPI is enabled or else it is disabled. When SPI is enabled, the normal I/O unctions of the pins are overridden.

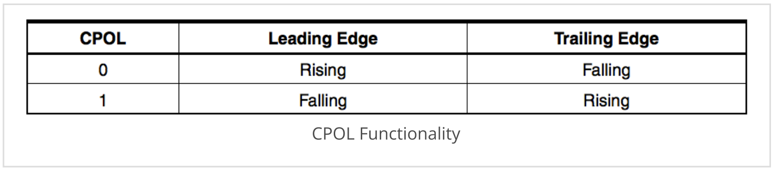
Shares

**it 5: DORD – Data Order**
ORD stands for Data ORDer. Set this bit to 1 if you want to transmit LSB irst, else set it to 0, in which case it sends out MSB first.

**it 4: MSTR – Master/Slave Select**
his bit is used to configure the device as *Master* or as *Slave*. When this bit is et to 1, the SPI is in *Master* mode (i.e. clock will be generated by the articular device), else when it is set to 0, the device is in SPI *Slave* mode.

**it 3: CPOL – Clock Polarity**
his bit selects the clock polarity when the bus is idle. Set this bit to 1 to nsure that SCK is HIGH when the bus is idle, otherwise set it to 0 so that CK is LOW in case of idle bus.

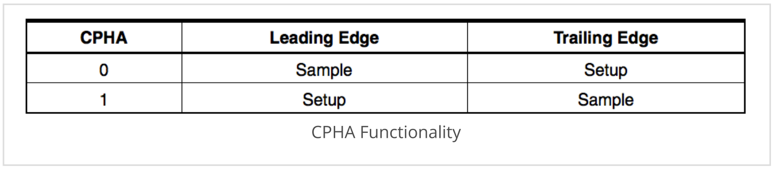
his means that when CPOL = 0, then the leading edge of SCK is the rising dge of the clock. When CPOL = 1, then the leading edge of SCK will actually be the falling edge of the clock. Confused? Well, we will get back to it a little later in this post again.



| CPOL | Leading Edge | Trailing Edge |
|---|---|---|
| 0 | Rising | Falling |
| 1 | Falling | Rising |

CPOL Functionality

**Bit 2: CPHA – Clock Phase**
This bit determines when the data needs to be sampled. Set this bit to 1 to sample data at the leading (first) edge of SCK, otherwise set it to 0 to sample data at the trailing (second) edge of SCK.

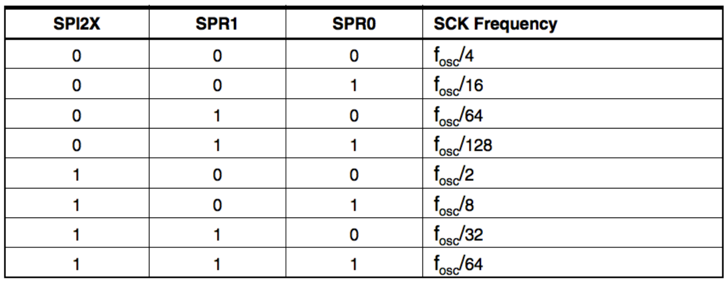


| CPHA | Leading Edge | Trailing Edge |
|---|---|---|
| 0 | Sample | Setup |
| 1 | Setup | Sample |

CPHA Functionality

These bits, along with the SPI2X bit in the SPSR register (discussed next), are used to choose the oscillator frequency divider, wherein the f$_{OSC}$ stands for internal clock, or the frequency of the crystal in case of an external oscillator.

The table below gives a detailed description.



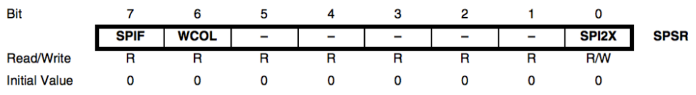
| SPI2X | SPR1 | SPR0 | SCK Frequency |
|---|---|---|---|
| 0 | 0 | 0 | $f_{osc}/4$ |
| 0 | 0 | 1 | $f_{osc}/16$ |
| 0 | 1 | 0 | $f_{osc}/64$ |
| 0 | 1 | 1 | $f_{osc}/128$ |
| 1 | 0 | 0 | $f_{osc}/2$ |
| 1 | 0 | 1 | $f_{osc}/8$ |
| 1 | 1 | 0 | $f_{osc}/32$ |
| 1 | 1 | 1 | $f_{osc}/64$ |

Frequency Divider

## The SPI Status Register (SPSR)

The SPI Status Register is the register from where we can *get* the status of he SPI bus and interrupt flag is also set in this register. Following are the its in the SPSR register.

Shares



| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | SPIF | WCOL | – | – | – | – | – | SPI2X | SPSR |
| Read/Write | R | R | R | R | R | R | R | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

SPSR Register

**3it 7: SPIF – SPI Interrupt Flag**
The SPI Interrupt Flag is set whenever a serial transfer is complete. An nterrupt is also generated if SPIE bit (bit 7 in SPCR) is enabled and global nterrupts are enabled. This flag is cleared when the corresponding ISR is executed.

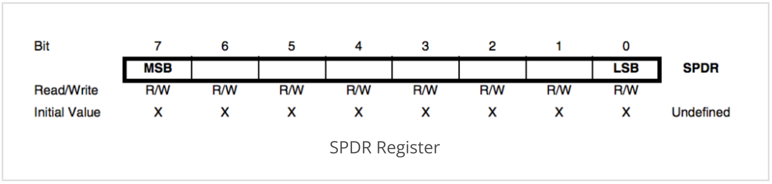
**3it 6: WCOL – Write Collision Flag**
The Write COLlision flag is set when data is written on the SPI Data Register (SPDR, discussed next) when there is an impending transfer or the data lines are busy.

This flag can be cleared by first reading the SPI Data Register when the WCOL is set. Usually if we give the commands of data transfer properly, this error does not occur. We will discuss about how this error can be avoided, in the later stages of the post.

**Bit 5:1**
These are reserved bits.

**Bit 0: SPI2x – SPI Double Speed Mode**
The SPI double speed mode bit reduces the frequency divider from 4x to 2x, hence doubling the speed. Usually this bit is not needed, unless we need very specific transfer speeds, or very high transfer speeds. Set this bit to 1 to enable SPI Double Speed Mode. This bit is used in conjunction with the SPR1:0 bits of SPCR Register.

The SPI Data register is an 8-bit read/write register. This is the register from where we read the incoming data, and write the data to which we want to transmit.



SPDR Register

The 7th bit is obviously, the Most Significant Bit (MSB), while the 0th bit is the Least Significant Bit (LSB).

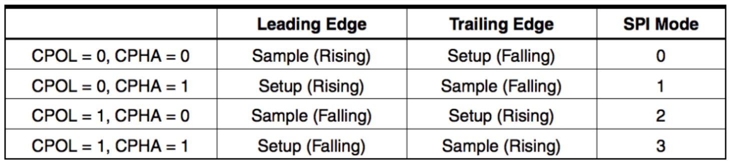
Now we can relate it to bit 5 of SPCR – the DORD bit. When DORD is set to 1, then LSB, i.e. the 0th bit of the SPDR is transmitted first, and vice versa.

## Data Modes

The SPI offers 4 data modes for data communication, wiz SPI Mode 0,1,2 and 3, the only difference in these modes being the clock edge at which data is sampled. This is based upon the selection of CPOL and CPHA bits.

Shares

The table below gives a detailed description and you would like to refer to this for a more detailed explanation and timing diagrams.



| | Leading Edge | Trailing Edge | SPI Mode |
|---|---|---|---|
| CPOL = 0, CPHA = 0 | Sample (Rising) | Setup (Falling) | 0 |
| CPOL = 0, CPHA = 1 | Setup (Rising) | Sample (Falling) | 1 |
| CPOL = 1, CPHA = 0 | Sample (Falling) | Setup (Rising) | 2 |
| CPOL = 1, CPHA = 1 | Setup (Falling) | Sample (Rising) | 3 |

SPI Data Modes

## The Slave Select (SS') Pin

As you would see in the next section, the codes of SPI are fairly simple as compared to those of UART, but the major headache lies here: the SS' pin!

SS' (means SS complemented) works in active low configuration. Which means to select a particular slave, a LOW signal must be passed to it.

***When set as input, the SS' pin should be given as HIGH (Vcc) on as Master device, and a LOW (Grounded) on a Slave device.***

When as an output pin on the *Master* microcontroller, the SS' pin can be used as a GPIO pin.

The SS pin is actually what makes the SPI very interesting! But before we proceed, one question is that why do we need to set these pins to some value?

The answer is, that when we are communicating between multiple devices working on SPI through the same bus, the SS' pin is used to select the slave to which we want to communicate with.

Let us consider the following two cases to understand this better:

microcontroller. Since we want only a specific slave to receive the data, the master microcontroller would give a low signal to the SS' pin of that specific microcontroller, and hence only that slave microcontroller would receive data.

2. **When there are multiple masters and a single slave.**
A similar setup as above can be used in this case as well, the difference being that the SS' lines of all the masters is controlled by the slave, while the slave SS' line is always held low. The slave would select the master through which it has to receive data by pulling its SS' high.Alternatively, a multiplexed system can be used where each master microcontroller can control every other master microcontroller's SS' pin, and hence when it has to transmit data, it would pull down every other master microcontroller's SS' Pin, while declaring its own SS' as output.

You can also refer to this if you are still confused regarding Slave Select.

Shares

## SPI Coded!

Jp till now, we only discussed about the advantages, uses, and register lescription, hardware connections etc. of the SPI. Now lets see how we Code it!

### Enabling SPI on Master

```
// Initialize SPI Master Device (with SPI interrupt)
void spi_init_master (void)
{
    // Set MOSI, SCK as Output
    DDRB=(1<<5)|(1<<3);

    // Enable SPI, Set as Master
    // Prescaler: Fosc/16, Enable Interrupts
    //The MOSI, SCK pins are as per ATMega8
    SPCR=(1<<SPE)|(1<<MSTR)|(1<<SPR0)|(1<<SPIE);

    // Enable Global Interrupts
    sei();
}
```

In the SPI Control Register (`SPCR`), the `SPE` bit is set to 1 to enable SPI of AVR. To set the microcontroller as Master, the `MSTR` bit in the `SPCR` is also set to 1. To enable the SPI transfer/receive complete interrupt, the `SPIE` is set to 1.

In case you don't wish to use the SPI interrupt, do not set the `SPIE` bit to 1, and do not enable the global interrupts. This will make it look somewhat like this-

```
// Initialize SPI Master Device (without interrupt)
void spi_init_master (void)
{
    // Set MOSI, SCK as Output
    DDRB = (1<<5)|(1<<3);

    // Enable SPI, Set as Master
}
```

When a microcontroller is set as Master, the Clock prescaler is also to be set using the `SPRx` bits.

## Enabling SPI on Slave

```
// Initialize SPI Slave Device
void spi_init_slave (void)
{
    DDRB = (1<<6);     //MISO as OUTPUT
    SPCR = (1<<SPE);   //Enable SPI
}
```

For setting an microcontroller as a slave, one just needs to set the `SPE` Bit in the `SPCR` to 1, and direct the MISO pin (PB4 in case of ATmega16A) as OUTPUT.

## Sending and Receiving Data

Shares

```
//Function to send and receive data for both master and slave
unsigned char spi_tranceiver (unsigned char data)
{
    // Load data into the buffer
    SPDR = data;

    //Wait until transmission complete
    while(!(SPSR & (1<<SPIF) ));

    // Return received data
    return(SPDR);
}
```

The codes for sending and receiving data are same for both the slave as well as the master. To send data, load the data into the SPI Data Register (`SPDR`), and then, wait until the `SPIF` flag is set. When the `SPIF` flag is set, the data to be transmitted is already transmitted and is replaced by the received data. So, simply return the value of the SPI Data Register (`SPDR`) to receive data. We use the return type as `unsigned char` because it occupies 8 bits and its value is in the range 0-255.

# Problem Statement

Enough of reading, time to get your hands dirty now! Get your hardware toolkit ready and open up your software. Let's demonstrate the working of SPI practically.

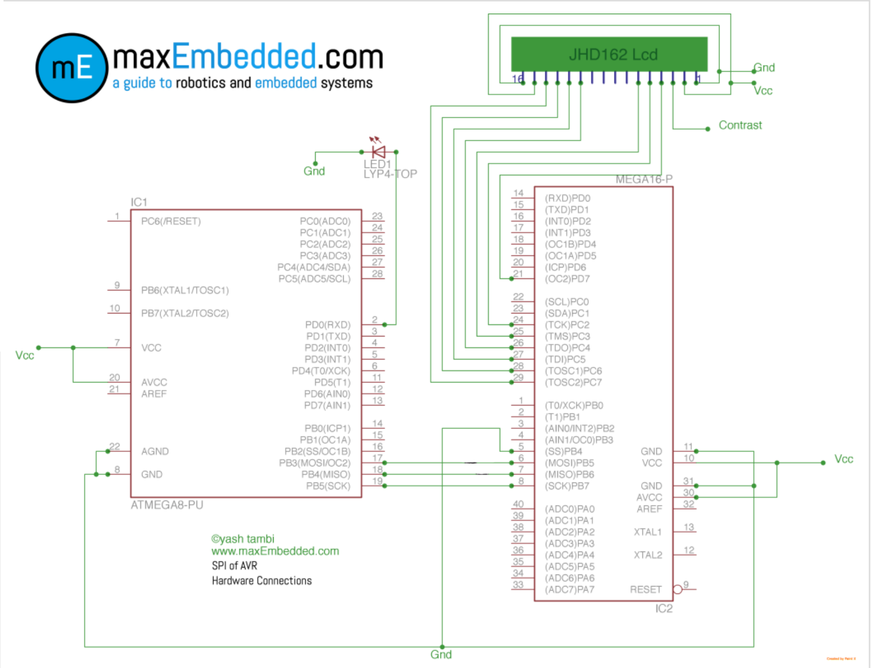
Let's assume a problem statement. Say the given problem statement is to send some data from *Master* to *Slave*. The *Slave* in return sends an acknowledgement (ACK) data back to the *Master*. The *Master* should check for this ACK in order to confirm that the data transmission has completed. This is a typical example of full duplex communication. While the *Master* sends the data to the *Slave*, it receives the ACK from the *Slave* simultaneously.

We would use the primary microcontroller (ATmega8 in this case) as the *Master* device, and a secondary microcontroller (ATmega16 in this case) as the *Slave* device. A counter increments in the *Master* device, which is being sent to the *Slave* device. The *Master* then checks whether the received data is the same as ACK or not (ACK is set as 0x7E in this case). If the received data is the same as ACK, it implies that data has been successfully sent and received by the *Master* device. Thus, the *Master* blinks an LED connected to it as many number of times as the value of the counter which was sent to the *Slave*. If the *Master* does not receive the ACK correctly, it blinks the LED for a very long time, thus notifying of a possible error.

On the other hand, *Slave* waits for data to be received from the *Master*. As soon as data transmission begins (from *Master* to *Slave*, the *Slave* sends ACK (which is 0x7E in this case) to the *Master*. The *Slave* then displays the received data in an LCD.

## Hardware Connections

Hardware connections are simple. Both the MOSI pins are connected together, MISO pins are connected together and the SCK pins are also connected together. The SS' pin of the slave is grounded whereas that of master is left unconnected. And then we have connected the LCD to the slave as well. Check Mayank's tutorial on LCD interfacing. We also connect an LED to the master to demonstrate the SPI interrupt. Here are the schematics–



SPI Hardware Connections (Click to Enlarge)

And here is how my final set up looks like–

Final Setup

## Full Code

The codes for the *Master* and *Slave* are given below. The codes are well commented, so it should be easy to understand what is going on in the code. In case something doesn't make sense, feel free to drop in a comment below. The full comments can viewed by scrolling the code sideways. You can also find the code in the AVR code gallery.

### Master Code

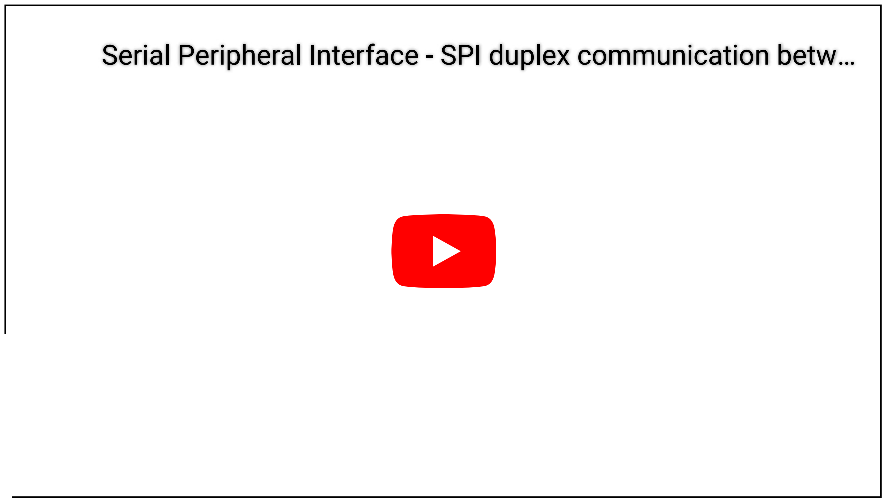
```
#ifndef F_CPU
#define F_CPU 16000000UL
#endif

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

#define ACK 0x7E
#define LONG_TIME 10000

//Initialize SPI Master Device
void spi_init_master (void)
{
    DDRB = (1<<5)|(1<<3);              //Set MOSI, SCK as Outp
    SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0); //Enable SPI, Set as
                                       //Prescaler: Fosc/16, E
}

//Function to send and receive data
unsigned char spi_tranceiver (unsigned char data)
{
    SPDR = data;                       //Load data into the bu
    while(!(SPSR & (1<<SPIF) ));       //Wait until transmiss
    return(SPDR);                      //Return received data
}

//Function to blink LED
void led_blink (uint16_t i)
{
    //Blink LED "i" number of times
    for (; i>0; --i)
    {
        PORTD|=(1<<0);
        _delay_ms(100);
    }
}

//Main
int main(void)
{
    spi_init_master();                  //Initialize SPI Maste
    DDRD |= 0x01;                       //PD0 as Output

    unsigned char data;                 //Received data storec
    uint8_t x = 0;                      //Counter value which

    while(1)
    {
        data = 0x00;                    //Reset ACK in "data"
        data = spi_tranceiver(++x);     //Send "x", receive AC
        if(data == ACK) {               //Check condition
            //If received data is the same as ACK, blink LED '
            led_blink(x);
        }
        else {
            //If received data is not ACK, then blink LED for
            led_blink(LONG_TIME);
        }
        _delay_ms(500);                 //Wait
    }
}
```

## Slave Code

```
#ifndef F_CPU
#define F_CPU 16000000UL
#endif

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include "lcd.h"

#define ACK 0x7E

void spi_init_slave (void)
{
    DDRB=(1<<6);                                  //MISO as OU
    SPCR=(1<<SPE);                                //Enable SPI
}

//Function to send and receive data
unsigned char spi_tranceiver (unsigned char data)
{
    SPDR = data;                                  //Load data
    while(!(SPSR & (1<<SPIF) ));                  //Wait unti
    return(SPDR);                                 //Return rec
}

int main(void)
{
    lcd_init(LCD_DISP_ON_CURSOR_BLINK);           //Initialize
    spi_init_slave();                             //Initialize
    unsigned char data, buffer[10];
    DDRA  = 0x00;                                 //Initialize
    PORTA = 0xFF;                                 //Enable Pul
    while(1)
    {
        lcd_clrscr();                             //LCD Clear
        lcd_home();                               //LCD move c
        lcd_puts("Testing");
        itoa(data, buffer, 10);                    //Convert ir
        lcd_puts(buffer);                          //Display re
        _delay_ms(20);                             //Wait
    }
}
```

## Video

Here is short demonstration of how this setup operates.


Serial Peripheral Interface - SPI duplex communication betw...

Shares

## Using Interrupts

n case you are interested in using the SPI Interrupt of the AVR, you should keep in mind the following things–

- Be sure to include `#include <avr/interrupt.h>` header.
- Set the `SPIE` bit to 1 in the `SPCR` register.
- Enable global interrupts using `sei()`.

Next thing is to write an Interrupt Service Routine (ISR), which can be written something like this–

```
// SPI Transmission/reception complete ISR
ISR(SPI_STC_vect)
{
    // Code to execute
    // whenever transmission/reception
    // is complete.
}
```

If you want to know what are interrupts, may be this little introduction might be useful.

## Summary

Let's have a look what we have learnt in this post–

- SPI is a serial communication protocol where all the devices are classified as either *Master* or *Slave*. It requires four pins to communicate – MISO,