## Using hardware and software to make new stuff

### Search

type, hit enter

### Categories

- 3D Printing (1)
- Affinity Photo (1)
- Aide-memoir (1)
- Analog (1)
- Ansible (3)
- Dates to Remember (3)
- Docker (1)
- Electronics (144)
- ESP8266 (12)
- FPGA (4)
- Garden (1)
- General (8)
- Home Built CPU (6)
- Informatica (1)
- Internet of Things (8)
- iOS (2)
- KiCad (4)
- MSP430 (2)
- Netduino (49)
- NuttX (9)
- Photography (3)
- Pico (10)
- Raspberry Pi (17)
- Silverlight (6)
- Software Development (88)
- STM32 (5)
- STM8 (54)
- Tips (5)

### Archives

- July 2024
- June 2024
- May 2024
- April 2024
- March 2024
- February 2024
- January 2024
- December 2023
- November 2023
- October 2023

- April 2023
- February 2020
- January 2020
- July 2019
- February 2018
- July 2017
- June 2017
- April 2017
- March 2017
- February 2017
- October 2016
- September 2016
- July 2016
- June 2016
- May 2016
- April 2016
- March 2016
- January 2016
- December 2015
- November 2015
- October 2015
- August 2015
- June 2015
- May 2015
- April 2015
- March 2015
- February 2015
- January 2015
- December 2014
- October 2014
- September 2014
- August 2014
- July 2014
- June 2014
- May 2014
- March 2014
- February 2014
- January 2014
- December 2013
- November 2013
- October 2013
- September 2013
- August 2013
- July 2013
- June 2013
- May 2013
- April 2013
- March 2013
- January 2013
- November 2012
- October 2012
- September 2012
- August 2012
- July 2012
- June 2012
- May 2012

- September 2011
- July 2011
- June 2011
- May 2011
- April 2011
- March 2011
- February 2011
- January 2011
- December 2010
- April 2010

## STM8S SPI Slave Device

For the next few posts I will be taking a look at SPI and how to use this to allow communication between two devices.

For some background reading I suggest that you visit Wikipedia and read the article on SPI. This post will assume that you are familiar with the material in that article.

In the first of the series we are going to be implementing a simple byte transfer between two devices, namely:

- Netduino Plus
- STM8S Discovery board

This scenario will require that SPI on the STM8S operates in slave mode as SPI on the Netduino Plus can only operate as a SPI master device. The Netduino family of products was chosen as the master because the SPI implementation is quick to setup and use. This means that we can be sure that any problems which arise during development are highly likely to be related to the STM8S code.

The problem definition is as follows:

- Configure the Netduino Plus as a SPI master device
- Send a repeated pattern of bytes over SPI to a listening slave device
- Configure the STM8S to operate as a SPI slave device
- Read the bytes from the SPI bus (MOSI) and send them back out on the bus (MISO)

We will also add some debugging code to allow us to connect a logic analyser to the STM8S and verify the data which is being received.

## SPI Master – Netduino Plus Code

The initial version of this application will use a low bus speed for the SPI communication. By using a low speed we will reduce the influence of transmission errors and also allow the code to be debugged and logic errors eliminated without worrying too much about errors introduced through timing issues.

Our simple application looks like this:

```
public class Program
{
    /// <summary>
    /// SPI object.
    /// </summary>
    private static SPI spi = null;

    public static void Main()
    {
        config = new SPI.Configuration(SPI_mod: SPI.SPI_module.SPI1,                /
                                       ChipSelect_Port: Pins.GPIO_PIN_D10,           /
                                       ChipSelect_ActiveState: false,                /
                                       ChipSelect_SetupTime: 0,
                                       ChipSelect_HoldTime: 0,
                                       Clock_IdleState: false,                       /
                                       Clock_Edge: true,                             /
                                       Clock_RateKHz: 10);
        spi = new SPI(config);

        byte[] buffer = new byte[1];
        while (true)
        {
            for (byte counter = 0; counter < 255; counter++)
            {
                buffer[0] = counter;
                spi.Write(buffer);
                Thread.Sleep(200);
            }
        }
    }
}
```

The configuration of the SPI bus is as follows:

- Chip select is digital pin 10
- Chip select is low when the bus is active
- Clock is active low (Clock Polarity – CPOL)
- Data will be valid on the rising clock edge (Clock Phase – CPHA)

It is important to note that the sampling settings must be duplicated on the slave device.

Once the SPI bus is configured, the application continuously loops outputting the bytes 0 to 254 on the SPI bus with a 200ms pause between each byte.

## SPI Slave – STM8S

With the exception of the clock speed, we now need to configure the STM8S as a slave device using the same settings as the SPI master device.

### The Registers

#### SPI_CR1_CPOL – Clock Polarity

The first setting we will consider is the clock polarity (CPOL). This is controlled by the CPOL bit in the CR1 register. This is defined as:

| Setting | Description |
|---|---|
| 0 | Clock is low when idle |

**SPI_CR1_CPHA – Clock Phase**

The clock phase determines when the data is ready to be sampled, i.e. on the rising or falling clock edge.

| Setting | Description |
|---|---|
| 0 | Data is ready to be sampled on the rising edge of the clock |
| 1 | Data is ready to be sampled on the falling edge of the clock |

We will be sampling on the first clock transition, on the rising edge.

**SPI_CR1_SPE – Enable or Disable SPI**

This register determines if SPI is enabled or disabled. Setting this register to 0 disables SPI, setting it to 1 enables SPI.

**SPI_ICR_TXIE and SPI_ICR_RXNE – Interrupt Enable/Disable**

These two registers determine if the SPI interrupts will be triggered on transmit (TXIE) or receive (RXIE). Setting a bit to 0 will disable the interrupt, setting it to 1 will enable the interrupt.

**SPI_DR – Data Register**

The data register is used in two contexts, when data has been received and to transmit data. Reading this register will retrieve data from the receive buffer. Setting this register will load the specified value into the transmit buffer.

**SPI_SR_RXNE – Receive Buffer Not Empty**

This bit in the status register indicates if the receive buffer is empty. You can check this value before you read the SPI_DR register to determine if there is any data waiting to be read.

## STM8S Code

The first thing we will need to do is to initialise the SPI bus matching the settings of the SPI master:

```
void InitialiseSPIAsSlave()
{
    SPI_CR1_SPE = 0;                    //  Disable SPI.
    SPI_CR1_CPOL = 0;                   //  Clock is low when idle.
    SPI_CR1_CPHA = 0;                   //  Sample the data on the rising edge
    SPI_ICR_TXIE = 1;                   //  Enable the SPI TXE interrupt.
    SPI_ICR_RXIE = 1;                   //  Enable the SPI RXE interrupt.
    SPI_CR1_SPE = 1;                    //  Enable SPI.
}
```

This code not only matches the master settings but also enables the generation of interrupts for transmit empty and receive not empty. These interrupts are handled by the following code:

```
#pragma vector = SPI_TXE_vector
__interrupt void SPI_IRQHandler(void)
{
```

```
            //
            //  Output some debug information.
            //
            OutputStatusCode(SC_OK);
            BitBang(byte);
        }
    }
```

This method checks the SPI_SR_RXNE flag to determine if the receive buffer is not empty. If there is data ready for processing then the data is retrieved and then transmitted back to the master.

Note that this method offers a naive approach to sending and receiving data, something we will overcome in subsequent posts.

We have also provided two methods for debugging, one will output a status code; the second will output a single byte by bit banging the data using two pins on an output port. We will need to use these methods with care when we start to look at higher transmission speeds. The two debug methods need to operate at speeds which allow the interrupt service routine to complete before the next interrupt is ready to be generated.

## Hardware Setup

The connections between the two devices are straight forward. The following pins should be connected:

| Pin Description | Netduino Pin | STM8S Discovery Pin |
|---|---|---|
| MISO | D12 | PC7 |
| MOSI | D11 | PC6 |
| SCLK | D13 | PC5 |
| Chip Select | D10 | PE5 |
| GND | GND | GND |

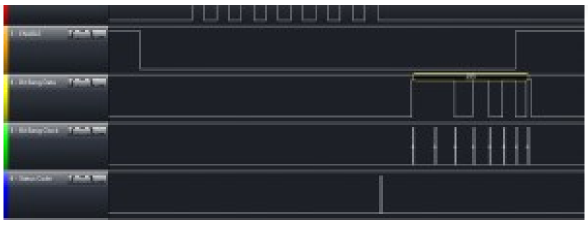
In addition to the above connections between the two boards we have three pins defined for debugging/diagnostics.

Port D, pin 2 will be used to output a status code. The code will be output as a series of high/low transitions.

Port D, pins 4 (clock) and 5 (data) will output debug/diagnostic data. This will be output in a similar form to the SPI data being transmitted on the SPI bus. This form has been chosen as it allows a logic analyser to be used to interpret the data.

## Results

If we connect the two devices and hook up a logic analyser we get output similar to the following:



Logic Analyser Output

The top four traces represent the data which is being transmitted by the Netduino and the STM8S on the SPI bus along with the clock and select control signals. The labelling on the traces indicate which signal is being shown.

The traces labelled 4 and 5 show the data which has been output from the *BitBang* method.

The final trace shows the status code.

If we read this trace from left to right we can see that the Netduino master output the byte 213 on MOSI. At the same time, the STM8S is sending the byte 212 back to the master on the MISO line. This difference of one is caused by the fact that the transmission from the STM8S is always one behind the transmission by the master to the slave.

Traces 4 and 5 confirm that the STM8S has in fact received the byte 213.

Trace 6 shows that we have a status code of 1 – a single pulse – showing that the application has not detected an error condition.

We see that the first thing that happens is that the transmission of data starts on the MOSI and MISO lines simultaneously. When the data transmission has completed we have in interrupt generated and the status code of 1 is output. Finally the application copies the data received onto the diagnostic output.

## Conclusion

This application represents the first step on the road to building a faster application capable of transmitting and receiving greater amounts of data.

As usual, the source code for this application is available for download (STM8S SPI Slave and Netduino SPI Master).

### Source Code Compatibility

| System | Compatible? |
|---|---|
| STM8S103F3 (Breadboard) | ✗ |
| Variable Lab Protomodule | ✗ |
| STM8S Discovery | ✓ |

Wednesday, November 14th, 2012 at 8:36 pm • Electronics, Netduino, STM8 • RSS 2.0 feed Both comments and pings are currently closed.

Comments are closed.

## Pages

- About
- Making a Netduino GO! Module

## Support This Site

Buy me a coffee

## Currently Reading