DESIGN NOTE #035

AUTHOR: ZOLTAN MOLNAR
KEYWORDS: SPI, HIGH SPEED, MASTER, SLAVE, COMMUNICATION

Using the SPI as High Speed, Bi-directional Data Bus with Automatic Master/Slave Switch

Introduction

While the UART offers asynchronouse communication support, there are some situations (dual controller systems), where high speed (Mbps) communication between two microcontrollers is desirable.

For this kind of applications the SPI bus seems to be a good solution because its high speed, but there are some limitations such as:

- The SPI interface is a synchronouse data bus, where data transfers are initiated by the SPI Bus Master
- Data transfers can't be initiated by the bus slave
- Data transfer from the Slave to the Master is possible only while the Master is writing into the Slave.

This design note shows how to achieve high speed, asynchronouse, half-duplex, interrupt driven communication with the hardware SPI, between two AVR microcontrollers.

Overview

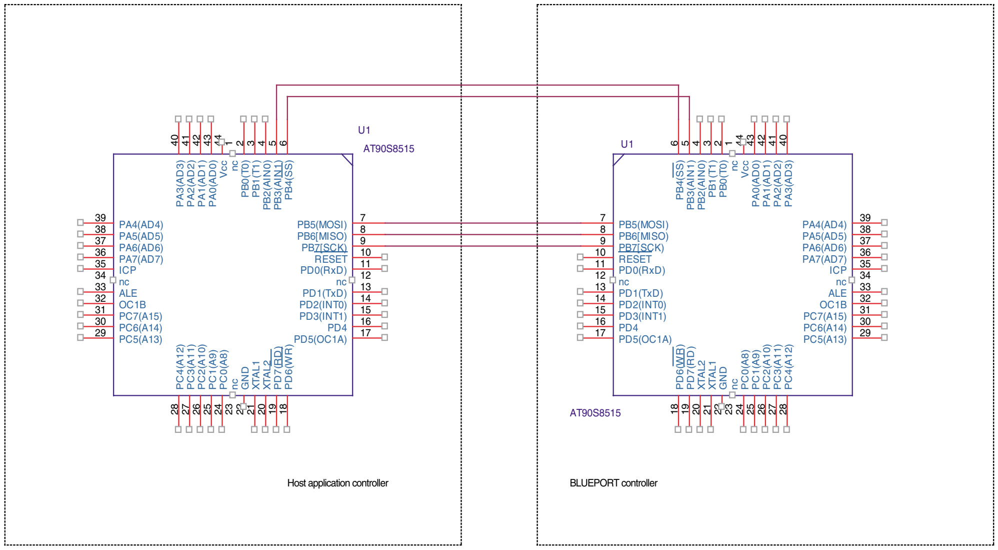
To send data from the SPI Slave to the SPI Master, a Master/Slave switch is performed whenever one of the SPI devices want to transmit data to the other. The communication is half-duplex.

This solution takes full use of the capability of the AVR's SPI port wich will switch to Slave mode if in Master mode the $\overline{SS}$ pin is driven low.

The SPI interconnection between the two controllers is shown in Figure 1 bellow.

The two AVRs have the $\overline{SS}$ pin of the other AVR wired to an output pin. Initially one of the controllers is in Master mode, the other in Slave mode. The Master controller has its $\overline{SS}$ pin configured as output, so Master/Slave switch is not possible. The Master controller will keep in low the $\overline{SS}$ pin of the Slave controller selecting as active SPI Slave. The Slave controller keeps the $\overline{SS}$ pin of the Master high till it is in Slave mode. After transmitting the first message from the Master to the Slave, the Master controller will configure its $\overline{SS}$ pin as input, enabling Master/Slave role switch when the Slave controller drives low the $\overline{SS}$ pin of the Master.

Figure 1. SPI Interconnection between Two Controllers



The Master/Slave switch mechanism is interrupt driven, so no additional load is put on the MCU (no waiting loops). A data transmission is initiated by the putSPI(u08 data) API that will check if the SPI port is in Master mode. If so putSPI(u08 data) will put one data byte into SPDR.

If the SPI port is in Slave mode, putSPI(u08 data) will initialize a Master/Slave transition in the other controller by driving its $\overline{SS}$ pin low.

Untill a controller is switching mode, the SPI interface is disabled. When the controller is in Slave mode, it will drive high the $\overline{SS}$ pin of the controller initiating the role switch. This is noticed, the SPI interface re-enabled in Master mode and data transmission started.

While the controller is in Master mode and transmitting a message its $\overline{SS}$ pin is configured as output, so no accidental Master/Slave switch can happen during the transfer. Master/Slave switch is enabled only after transmitting a whole message. While in Master mode and transmitting the SPI transmission complete interrupt will fetch data from the source FIFO to the SPDR till the message is transmitted.

If not transmitting and controller is in Master mode, a transmission complete interrupt will signal a Master/Slave role switch request, so the SPI interface will be disabled, reconfigured as Slave and re-enabled.

If the SPI interface is in Slave mode, a transmission complete interrupt will signal the reception of a message byte, so the content of the SPDR will be stored in the SPI receive FIFO.

This mechanism enables that high speed, asynchronous, half-duplex bi-directional data transmission to be performed through the synchronous SPI bus.

If the $\overline{SS}$ Selection pin of the AVR is wired as an external interrupt source, the same mechanism can be used even with microcontrollers that doesn't have the AVR SPI port abilities.

Code

Here is an example application where both of the microcontrollers uses its UART to receive data wich is transmitted through the SPI to the other controller. Data received from the SPI will be transmitted through the UART, achieving UART to UART communication. This sample application can serve as backbone for half-duplex protocol translators or other similar applications.

The bi-directional SPI communication library has the following API:

- Void SPImaster(void) – set up SPI&port pins for Master mode.
- Void SPIslave(void) – set up SPI&port pins for Slave mode.
- Void initSPI_RX(t_fifo* pfifo) – initialize interrupt based reception into a FIFO.
- Void putSPI(u08 data) – send one byte through the SPI and perform automatic Master/Slave switch if required (controllers SPI is in Slave mode).

The application uses two other librarys: A circular FIFO and an UART communication library.

The circular FIFO library offers dynamic allocation of circular FIFOs, commonly used in communication applications.

The UART communication library supports interrupt based data reception into a circular FIFO, interrupt based transmission from a circular FIFO as well polled mode byte transmission.

The application code is the same in both microcontrollers, to learn more about the FIFO and the UART communication library and to download the full source code please visit http://ww.microhardsoft.igo.com

```
/***************************************************************************
   Sample application showing asynchronouse, half-duplex SPI communication.
***************************************************************************/
#include <io.h>
#include <interrupt.h>
#include "global.h"
#include "hardware.h"
#include "fifo.h"
#include "uart.h"
#include "spi_bidir.h"

//=============================================================================
t_fifo HWUARTrx_fifo;                        //hardware UART receive FIFO
t_fifo SPIrx_fifo;                           //hardware SPI receive FIFO

//=============================================================================
int main (void)
{
 /*set up port pin directions and levels*/
 outp(0xFF,DDRB);
 outp(0xFF,PORTB);
 outp(0xFF,DDRD);
 outp(0xFF,PORTD);

 //init. hardware UART receive FIFO
 if(!initFIFO(&HWUARTrx_fifo, 100))         //try to create and initialize a 100 byte circular fifo
 {  for(;;){}   }                           //trap memory allocation errors

 //init. hardware SPI receive FIFO
 if(!initFIFO(&SPIrx_fifo, 100))            //try to create and initialize a 100 byte circular fifo
 {  for(;;){}   }                           //trap memory allocation errors

 setupUART0(57600,&HWUARTrx_fifo);          //initialize UART speed and receive FIFO
 initSPI_RX(&SPIrx_fifo);                   //initialize SPI receive FIFO
 SPImaster();                               //initialize SPI master mode

 sei();                                     //enable interrupts

 //MAIN program loop
 for(;;)
 {
   if(!HWUARTrx_fifo.isempty)               //check if there is data in HWUART input FIFO
   {
    u08 data=getFIFO(&HWUARTrx_fifo);       //if it is, extract byte/byte
    putSPI(data);                           //send through SPI
   }
```

```
   if(!SPIrx_fifo.isempty)                //check if there is data in SPI input FIFO
   {
    u08 data=getFIFO(&SPIrx_fifo);        //if it is, extract byte/byte
    putch0(data);                         //send through UART
   }
 }

 return 0;
}
```

Here is the code for the SPI driver:

```
/****************************************************************************
   Header file for bi-directional SPI communication support.
****************************************************************************/
#ifndef SPI_H
#define SPI_H

/*init. SPI, master mode*/
extern void SPImaster(void);

/*init. SPI slave mode*/
extern void SPIslave(void);

/*Init. SPI receive FIFO*/
extern void initSPI_RX(t_fifo* pfifo);

/*Send a byte through the SPI, perform master/slave switch if SPI is in slave mode*/
extern void putSPI(u08 data);

#endif

/****************************************************************************
   Implementation of bi-directional SPI communication library.
****************************************************************************/
#include <io.h>
#include <sig-avr.h>
#include "global.h"
#include "hardware.h"
#include "fifo.h"
#include "spi_bidir.h"

#define SPIport PORTB                     //SPI interface port pin levels
#define SPIpins PINB                      //SPI interface port pin levels
#define SPIdir DDRB                       //SPI interface port pin directions

#if defined (__AVR_ATmega103__) || defined (__ATmega103__)    /*on ATMEGA103 SPI is on PB.0-4*/
 #define SS 0                             //SPI SS pin
```

```
 #define SSsel 4                          //SS pin for SPI peer

 #define SPImask 0x1F

 #define SPIdir_S 0xf8
 #define SPIlvl_S 0xff

 #define SPIdir_M 0xf6
 #define SPIlvl_M 0xef
#else                                     /*on regular AVRs SPI is on PB.3-7*/
#define SS 4                              //SPI SS pin
 #define SSsel 3                          //SS pin for SPI peer

 #define SPImask 0xf8

 #define SPIdir_S 0x4f
 #define SPIlvl_S 0xff

 #define SPIdir_M 0xaf
 #define SPIlvl_M 0xf7
#endif

t_fifo* pSPIrx_fifo;                      //pointer to hardware SPI receive FIFO
u08 SPImode;                              //previouse SPI state: if "1" master/if "0" slave
volatile u08 SPIstatus;

/*init. SPI, master mode*/
void SPImaster(void)
{
 u08 temp=inp(SPIdir);                    //prepare pin directions for SPI master mode
 temp=temp | SPImask;
 temp=temp & SPIdir_M;
 outp(temp,SPIdir);

 temp=inp(SPIport);                       //prepare pin levels for SPI master mode
 temp=temp | SPImask;
 temp=temp & SPIlvl_M;
 outp(temp,SPIport);

 loop_until_bit_is_set(SPIpins,SS);       //wait till SPI peer is slave
 outp(0xF8,SPCR);                         //set up SPI control register
 SPImode=1;                               //signal master
}

/*init. SPI slave mode*/
void SPIslave(void)
{
 u08 temp=inp(SPIdir);                    //prepare pin directions for SPI slave mode
```

```
 temp=temp | SPImask;
 temp=temp & SPIdir_S;
 outp(temp,SPIdir);

 temp=inp(SPIport);                      //prepare pin levels for SPI slave mode
 temp=temp | SPImask;
 temp=temp & SPIlvl_S;
 outp(temp,SPIport);

 outp(0xE8,SPCR);                        //set up SPI control register
 SPImode=0;                              //signal slave
}

/*Init. SPI receive FIFO*/
void initSPI_RX(t_fifo* pfifo)
{ pSPIrx_fifo=pfifo; }

/*Send a byte through the SPI, perform master/slave switch if SPI is in slave mode*/
void putSPI(u08 data)
{
 if(bit_is_clear(SPCR,MSTR))             //verify that we are in slave mode
 {
  cbi(SPCR,SPIE);                        //disable SPI interrupts
  cbi(SPCR,SPE);                         //disable SPI interface -> PortB, pin SS becomes a gen. input pin
  cbi(SPIport,SSsel);                    //SS pin of master to low -> init. master -> slave transition
  while(bit_is_clear(SPIpins,SS))
  {};                                    //wait till SS is driven high by former master -> now its slave
  SPImaster();                           //became an SPI master
 }

 outp(data,SPDR);                        //send first data byte
 SPIstatus=1;                            //we are transmitting through SPI
 while(SPIstatus){};                     //wait till byte is transferred
}

/*interrupt handler for SPI transmission complete interrupt.
Handles transmit/receive/master-to-slave transition. */

SIGNAL(SIG_SPI)
{
 if(bit_is_set(SPCR,MSTR))               //transmit complete interrupt condition
 {
  SPIstatus=0;                           //byte is sent through SPI
 }
 else                                    //receive complete or master-to-slave interrupt
 {
  if(!SPImode)                           //receive interrupt
  {
```

```
   u08 SPItemp=inp(SPDR);                    //load received character
   if(!(pSPIrx_fifo->isfull))
   {
    cbi(PORTD,LED);
    putFIFO(pSPIrx_fifo,SPItemp);
   }
  }
  else                                       //master to slave transition
  {
   cbi(SPCR,SPIE);                           //disable SPI interrupts
   cbi(SPCR,SPE);                            //disable SPI interface
   SPIslave();                               //became an SPI slave
  }
 }
}
```