IMPLEMENTATION OF SSI MASTER INTERFACE
APPLICATION NOTE

Application Note

12 August 2013

AMERICA
FRABA Inc.
1800 East State Street, Suite 148
Hamilton, NJ 08609-2020, USA
T +1-609-750-8705, F +1-609-750-8703
www.posital.com, info@posital.com

EUROPE
POSITAL GmbH
Carlswerkstrasse 13c
D-51063 Köln, Germany
T +49 221 96213-0, F +49 221 96213-20
www.posital.eu, info@posital.eu

ASIA
FRABA Pte. Ltd.20 Kallang Avenue
#01-00 Pico Creative Centre
Singapore 339411, Singapore
T +65 65148880, F +65 62711792
www.posital.sg, info@posital.sg

IMPLEMENTATION OF SSI MASTER INTERFACE

APPLICATION NOTE

For additional information about our products click on the links below:

IXARC Rotary Encoders

TILTIX Inclinometers

LINARIX Linear Sensors

Table of Contents

IMPLEMENTATION OF SSI MASTER INTERFACE
APPLICATION NOTE

IMPLEMENTATION OF SSI MASTER INTERFACE
APPLICATION NOTE

Table of Figures

IMPLEMENTATION OF SSI MASTER INTERFACE
APPLICATION NOTE

Introduction

Figure 1: SSI Logo

This application note describes in detail on how to implement a synchronous serial interface (SSI) master using an Atmel ATmega88 controller.

The following sections will cover a brief description about the SSI protocol and discuss the hardware and software implementations in detail. For better understanding, hardware examples and illustrations of software implementation with I/O ports and SPI interface of the ATmega 88 have been used.

Although, the knowledge of the ATmega88 is helpful for programming, it is not very necessary to understand the underlying concepts about the synchronous serial interface implementation explained in this document.

IMPLEMENTATION OF SSI MASTER INTERFACE
APPLICATION NOTE

1. SSI Theory

Synchronous Serial Interface (**SSI**) is a widely used serial interface standard for industrial applications especially, rotary encoders. It is a point-to-point connection from a master (e.g. PLC, microcontroller or other control systems) to a slave (e.g. the rotary encoder). In this type of interface, the position data is continually updated by the sensor and made available to the output register.

Data is shifted out when the sensor receives a pulse train from the controller. When the least significant bit (LSB) is transmitted, the sensor holds the data constant for a certain period of time. When this time has elapsed, the new position data is updated to the output register continuously once again.

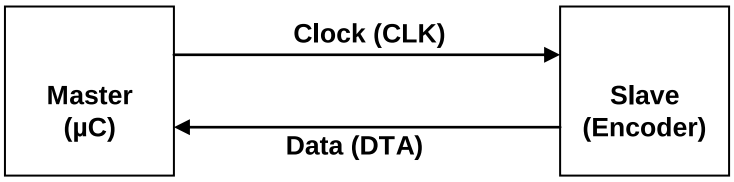


Figure 1.1: Simple SSI Block Diagram

The Clock (CLK) and Data (DTA) signals are transmitted according RS-422 standards. This standard also known as, TIA/EIA-422-B, is an industry standard specifying the electrical characteristics of a differential voltage driven transmission circuit. The advantage of differential signalling is the improved resistance to electromagnetic interference (EMI), especially in industrial environments and on larger signal line (Transmission) lengths.

SSI Data is transferred in a single data word with the most significant bit (MSB) first. Rotary encoders normally use 13 bits of data for transmitting the angle within one revolution (singleturn). If the revolutions are also counted (multiturn), a 25 bit word is used.

Conventionally, the SSI interface of the slave would have been implemented using a parallel load shift register in conjunction with a retriggerable mono flop to freeze the value, while a transfer is in progress. But nowadays, the interface is commonly integrated into FPGAs, PLDs or customized ASICs.

An ideal SSI timing diagram can be found below, in figure 1.2.

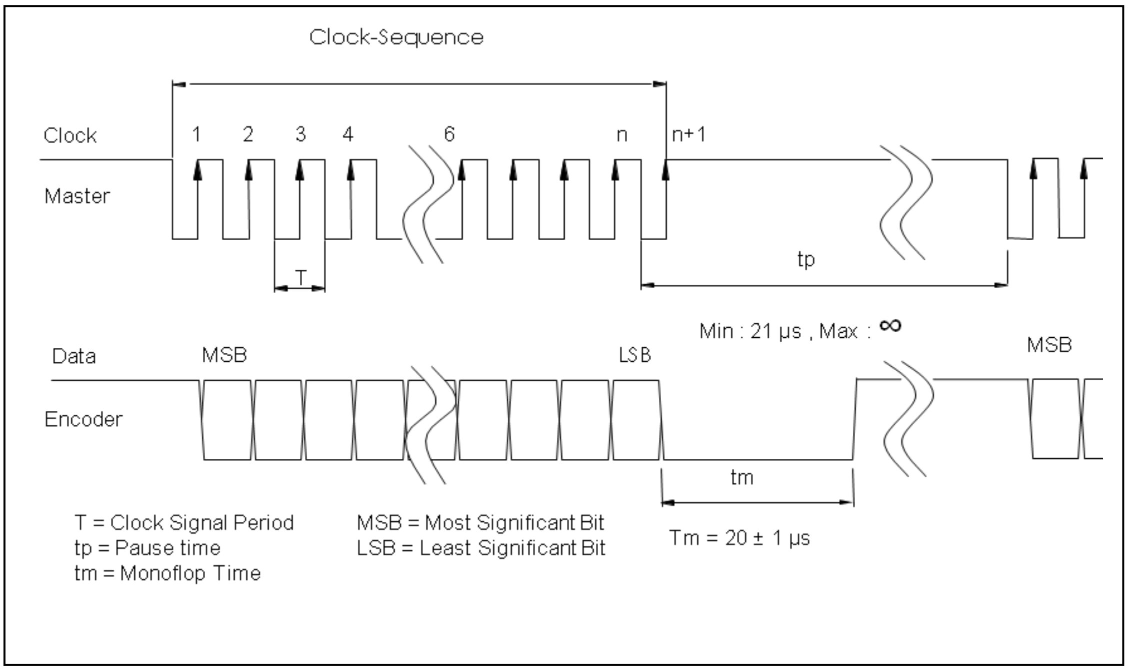


Figure 1.2: SSI Timing Diagram

The time ‘tm’ represents the transfer timeout. This is the time required by the encoder to recognize that a transfer is complete. ‘tp’ is called the pause time or the time delay between two consecutive clock sequences. It should always be greater than 21 µs, a maximum time is not defined.

In idle state, encoder data line stays HIGH. After the first falling edge of the clock, the the position value of the encoder is still held constant with the Data Level still remaining in HIGH state.. With the first rising edge, the first bit, the MSB is transmitted each rising clock edge will trigger the transmit of a bit. Finally when the LSB is transferred (end of transmission) an additional rising clock will set the data output to LOW level. This will be held low for 20 ±1 µs (monoflop time). After the time is over the encoder will start to update the position value continuously and the data line is set to HIGH state. The next transmission is started with a train of clock pulses.

The maximum clock frequency can be to 2MHz or higher (period of 500ns). The minimum clock frequency is 50 KHz. This value is determined by the timeout definition. For example, a timeout time of 20 ±1 µs corresponds to 50 KHz.

Most SSI-devices implement multiple transmissions. Multiple transmission, also known as ringshift or double transmission, can be used to improve transfer safety by repeatedly reading the same data word. The encoder will not update the data word before SSI timeout occurs. If the encoder is continuously clocked, it leads to multiple transmissions of the same position data without updating. The data words can be compared inside the SSI Master to recognize transmission errors normally two transmissions are enough to ensure a safe transmission.

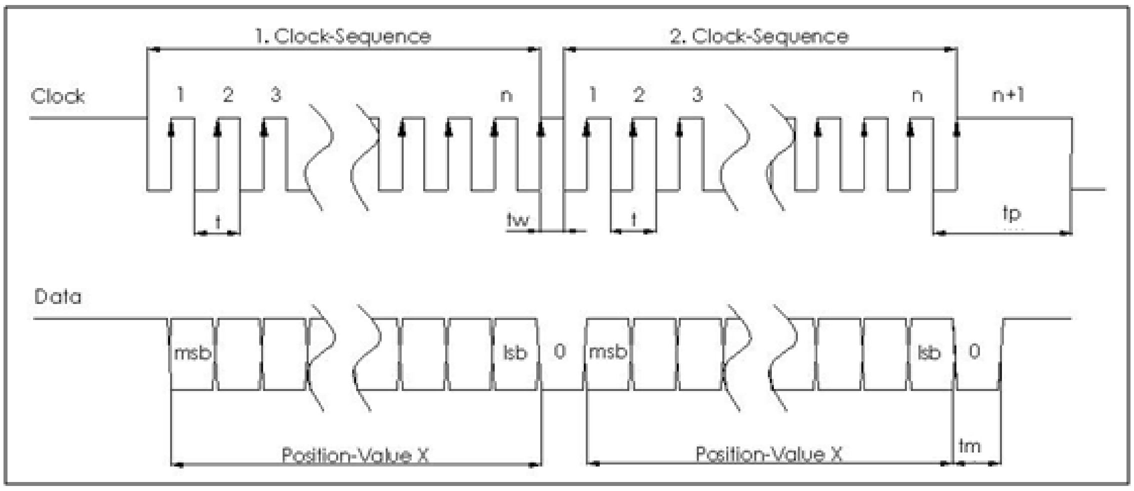


Figure 1.3: Multiple Transmissions in SSI Interface

However, after n clocks (where ‘n’ is the resolution of the encoder), the following rising clock cycle (n+1) will set the data output to LOW level. If the master continues providing a clock signal, without waiting the transfer timeout,, the encoder repeats the data word starting with the MSB. ‘tw” should always be maintained less than 19 µs.

Note that no particular start or stop sequence is required. The master simply starts clocking and stops when all necessary bits have been transferred. The clock rate should be more than the minimum clock rate of 80 KHZ and should not exceed 2MHz. The transfer pause between consecutive transfers has to be taken into account for updating the next position value. A running transmission can be interrupted at any time by just stopping the clock. The Slave than will recognize it after the tm time and just start to update its value.

To understand the SSI interface based transfer more clearly, a real world illustration has been used.

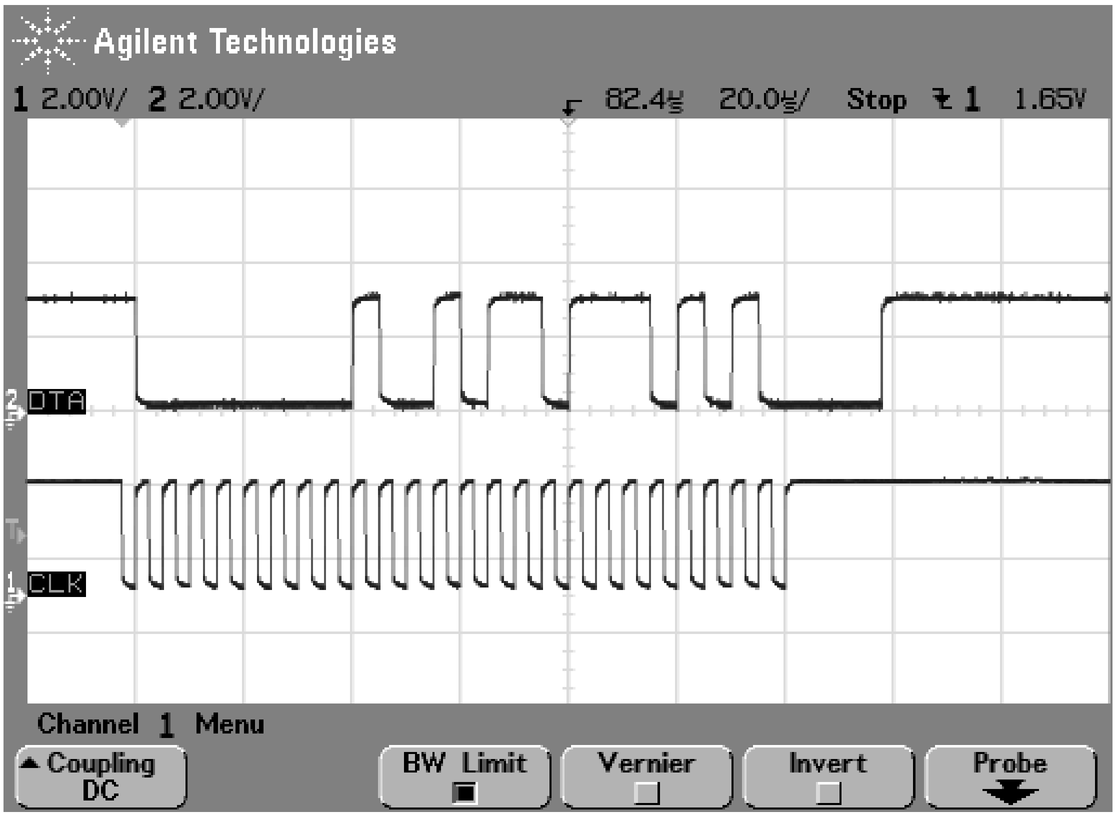


Figure 1.4: Real World SSI Transfer

The above figure shows a real world example of a single transfer. The data word transferred is binary 0000 0000 1001 0110 1110 1010 or hex 0x00096EA. The interpretation of this value is device and sometimes configuration specific.

Now we can clearly see that the data transmission stays HIGH until the first rising edge. At the first rising edge, DTA (the data transmission line) starts to transmit the data. Similarly, the transmission of data is completed by the last but one transmission edge (‘n$^{th'}$ rising edge) and the next rising edge of the clock sets the DTA to LOW. Since the last bit transferred is 0, the timeout of the signal, 20µs is clearly visible.

2. SSI Hardware

The hardware is fairly simple. Just use any RS-422 transceiver to generate the differential signals from/to the microcontroller port pins.

Simple SSI Master Implementation

The following example uses the MAX1486 transceiver from MAXIM. RS-422 transceivers are available from many companies (Linear Technologies, National Semiconductor, to name just two), in various shapes and flavours (3.3V; 5V, single transceiver, multiple transceivers in a single chip etc).

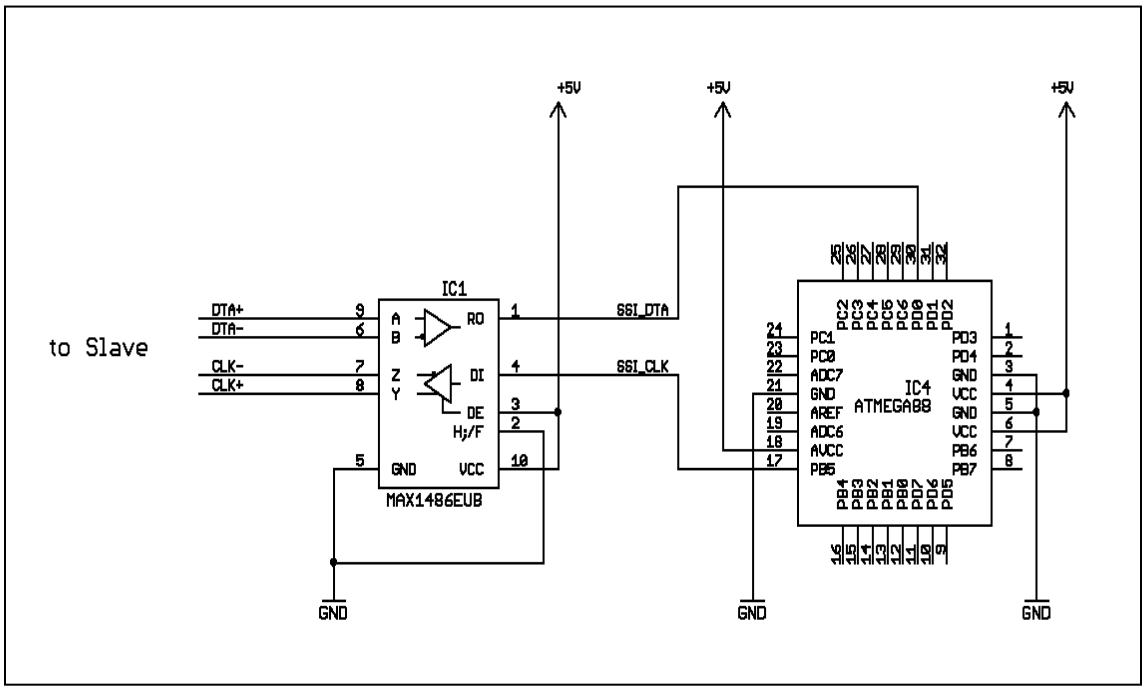


Figure 2.1: Simple SSI Master

IMPLEMENTATION OF SSI MASTER INTERFACE
APPLICATION NOTE

SSI Interface with Opto-coupler (Galvanic Insulated)

A different implementation uses a galvanic insulated master. This insulation isolates functional sections of the system. It is an effective method of breaking ground loops by preventing unwanted current from travel-ling between two units sharing a ground conductor.

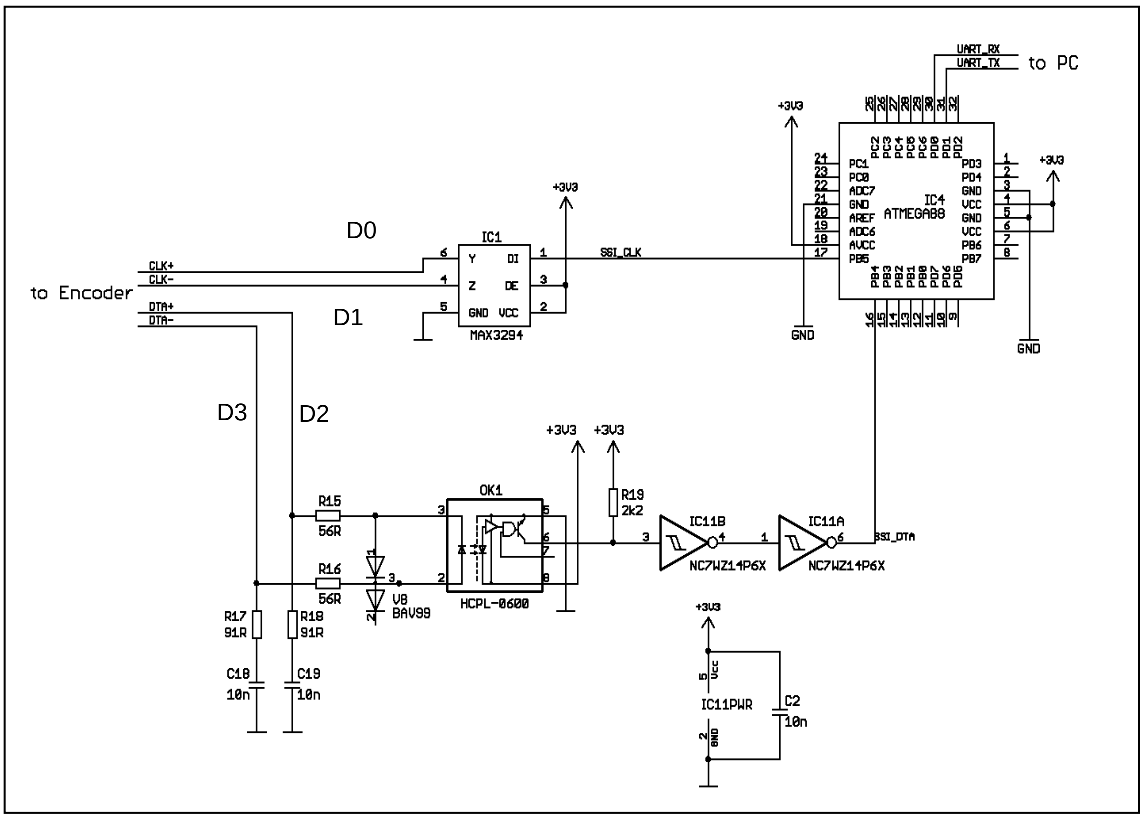


Figure 2.2: SSI with Opto-coupler (Galvanic Insulated)

The above implementation in figure 2.2 uses an opto-coupler together with a Schmitt-trigger buffer for the receiver stage and an RS-422 level translator for the transmitter. This version uses a 3.3 V supply instead of 5 V.

The encoder contains a similar RS-422 transceiver stage. On the encoder side, CLK+ and CLK- are in-puts, and DTA+ and DTA- are outputs.

Illustration of SSI Transfers using differential signals

Figure 2.3, shows DTA and `CLK` at the microcontroller (Ch #1 and Ch #2) and the differential signals to the decoder (D1, D2) and from the encoder (D3, D4).

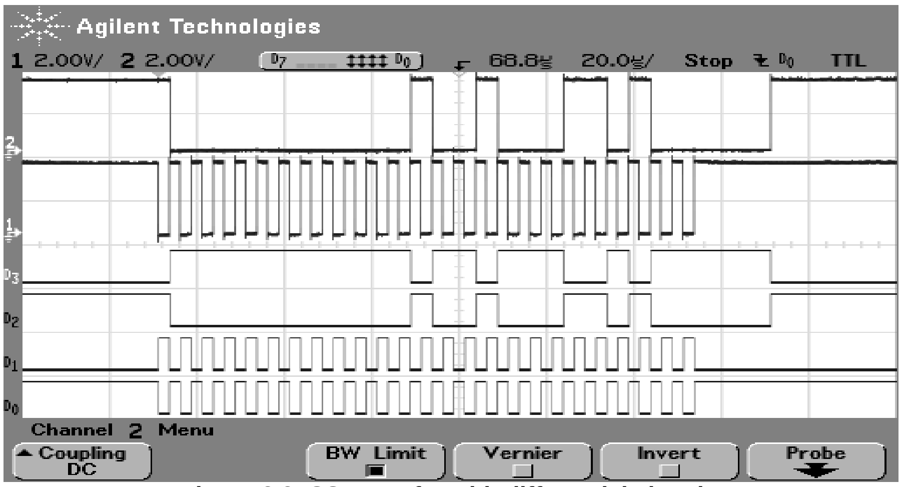


Figure 2.3: SSI transfer with differential signals

When no encoder is connected, the receiver signals (`DTA+` and `DTA-`) will be open. With the circuit shown in Figure 2.2, the following signals result:

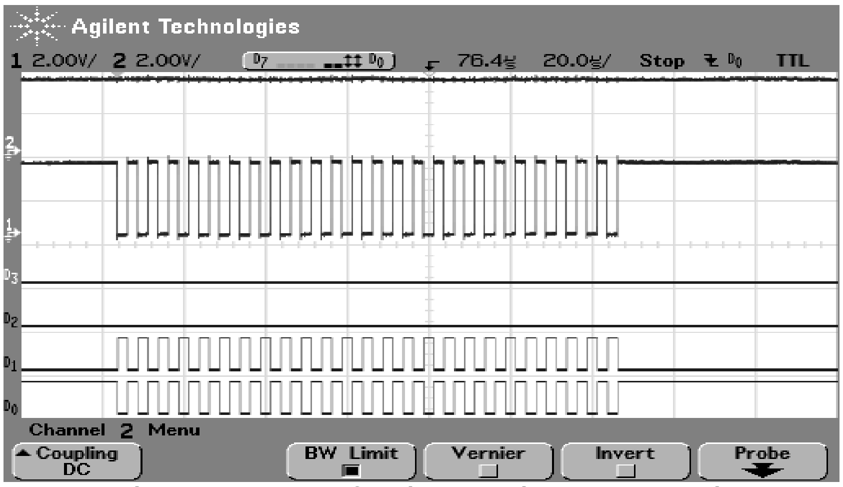


Figure 2.4: SSI transfer when encoder not connected

The clock signal from the SSI master is present as usual, but both input lines (D2 and D3) are low. Since this means the LED inside the opto-coupler is not driven, the pull-up resistor at the output of the opto-coupler will return a high signal to the microcontroller (Channel #2).

3. SSI Software

3.1 SSI Interface Using I/O Ports

The following example routine reads a 25 bit SSI word and returns it as 32bit value:

```
Uint32 pinToggleReadSSI ( void )

{
    Uint8  bit_count;
    Uint32 u32result = 0;
    Uint8  u8portdata;

    for (bit_count=0; bit_count<25; bit_count++)
    {
        // falling edge on clock port
        SSI_CLK_PORT &= ~(1 << SSI_CLK_BIT);

        // left-shift the current result
        u32result = (u32result << 1);

        // read the port data
        u8portdata = SSI_DTA_PORT;

        // rising edge on clock port, data changes
        SSI_CLK_PORT |= (1 << SSI_CLK_BIT);

        // evaluate the port data (port set or clear)
        if ( (u8portdata & (1 << SSI_DTA_BIT)) != 0)
        {
            // bit is set, set LSB of result
            u32result = u32result | 0x01;
        }  // if
    } // for
    return u32result;
}
```

Figure 3.1: Code Example for Reading SSI Data by pin toggling

SSI_CLK_PORT is used as an output, SSI_DTA_PORT as an input.

This codes results in the following transmission shown in **Error! Reference source not found.**:

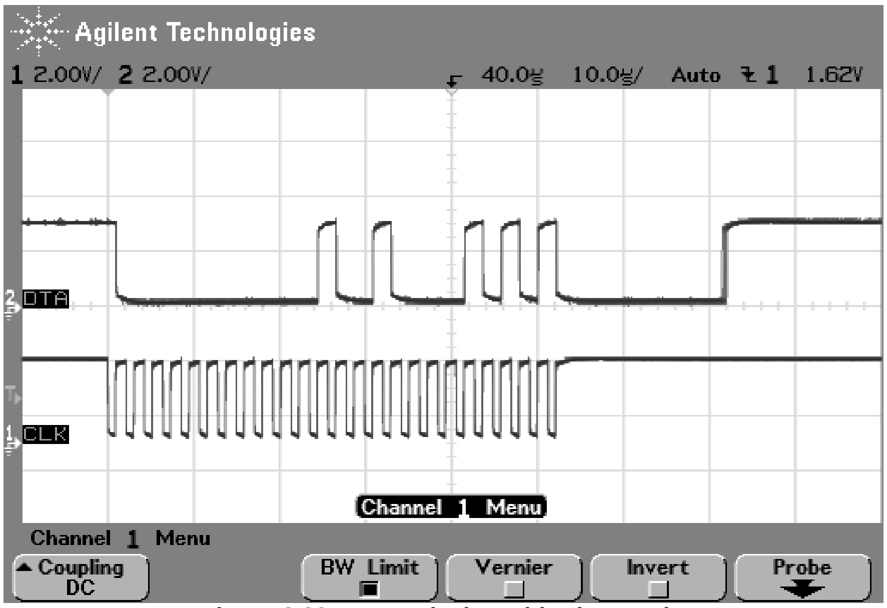


Figure 3.22: Transmission with Pin Toggle

Only the `CLK+` and `DTA+` signals are shown. The routine above does simply read one data word from the encoder. The wait for the timeout is not handled here, but in the main routine:

```
// forever loop
for (;;)
{
        // get the SSI word
        u32ssiResult = readSSI_pinToggle();

        // do some processing here

        // delay at least 25µs for SSI timeout
        for (i=0; i<1000; i++)
        {
            asm( " nop " );  // prevent the optimizer from removing the loop
        }
}
```

Figure 3.3: Code Example: Main routine contains delay

Depending on the execution time of your processing, the delay may or may not be necessary. For example, if you retransmit the word by a serial port to a PC, this transmission usually takes way longer than 25µs and no delay loop is required.

3.2 Reading SSI with the SPI Port

Using pin toggling is not a particular efficient method. An alternative is the use of the synchronous peripheral interface (SPI) found on most microcontrollers. This interface provides the means to serially read and write data synchronous to a clock signal. For the ATmega88 used in this example, the code could look like this:

```
/*FH**************************************************************************
* Name:         spiInit
* Parameters:   -
* Return value: -
* Description:  init SPI as master for use with SSI
****************************************************************************/
void spiInit( void )
{
    // configure SCK, MOSI and Slave Select as output
    DDRB = (1 << SPI_SCK_BIT) | (1 << SPI_MOSI_BIT) | (1 << SPI_SS_BIT);

    // configure SPI as master, with CLK idle high
    SPCR = (1 << SPE) | (1 << MSTR) | (1 << CPOL);
}
/*FH**************************************************************************
* Name:         readSSI_SPI
* Parameters:   -
* Return value: value read from SSI
* Description:  read a 25 Bit SSI word using the SPI interface
****************************************************************************/
Uint32 spiReadSSI( void )
{
    Uint8  u8byteCount;
    Uint8  u8data;
    Uint32 u32result = 0;

    for (u8byteCount=0; u8byteCount<4; u8byteCount++)
    {
        // send a dummy byte, read the result
        SPDR = 0xFF;                          // send 0xFF as dummy
        u32result <<= 8;                      // left shift the result so far
        while ( (SPSR & (1 << SPIF)) == 0);   // wait until transfer complete
        u8data = SPDR;                        // read data from SPI register
        u32result |= u8data;                  // and ‘or’ it with the result word
    }

    u32result >>= 7;                           // throw aways the LSBs

    return u32result;
}
```

Figure 3.4: Code Example: Using the SPI to read SSI data

This code example contains two functions, first `spiInit()`, which is used at start-up to initialize the SPI of the ATmega, second the actual read function, `spiReadSSI()` to read the SSI data word from the encoder. Figure 3.5 shows the transfer.

A total of 32 bits are transferred. This is the result of the SSI data word having 25 bits of data, but SPI only providing the means to transfer 8 bit per transfer. This is made up for by the higher transfer rate.

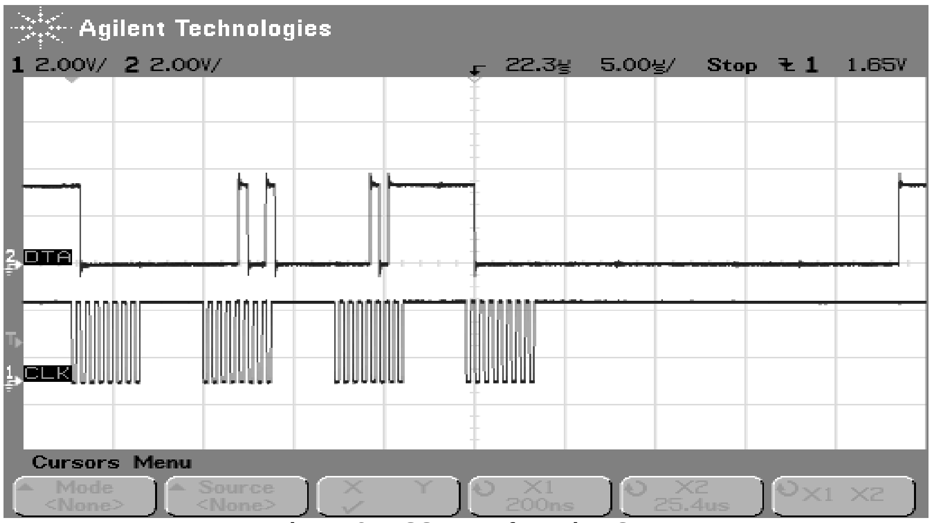


Figure 3.5: SSI transfer using SPI

Compared to the pin toggling transfer mode, the data transfer is almost twice as fast, even though a total of 32 bits are read.

Another side effect of reading 32 bits is that the resulting data word is left aligned, i.e. the first 25 bits contain the actual data. The next 7 bits contain the beginning of the data word again ("cyclic transmission of same data"), and either have to be masked, or eliminated by cyclic shifting of the data word, as done in the example above. However, depending on the further evaluation of the data word, this may not always be necessary in every application.

3.3 Evaluating the Data Word

Now that the data word has successfully reached the microcontroller, it needs to be evaluated further.
A standard encoder delivers a data word consisting of a "multi turn" part, containing the number of full 360° turns, and a "single turn" part, containing a value for the current angle. Multi turn and single turn are often shortened to "MT" and "ST". The first bit (bit 24) is always 1.

In the following examples we use an encoder that provides 12 bits of single turn and 12 bits of multi turn data:

Bi	24	23	…	12	11	…	0
	1	Multi Turn Value			Single Turn Value		
	1	MS	…	LSB	MS	…	LSB

Table 3.1: Data Format 12 Bit Multi Turn, 12 Bit Single Turn

3.3.1 Separating Singleturn and Multiturn values

To separate the two values, we need to mask out the multi turn part and right shift and mask the leading '1'.

```
    Uint32  u32ssiResult;
    Uint16  u16singleTurn;
    Uint16  u16multiTurn;

    // get the SSI word
    u32ssiResult = pinToggleReadSSI();

        // extract single and multi turn values from the data word
    u16singleTurn = u32ssiResult & 0x0FFF;
    u16multiTurn  = (u32ssiResult >> 12) & 0x0FFF;
```

Figure 3.6: Code Example 4.4: Extracting Multi Turn and Single Turn values

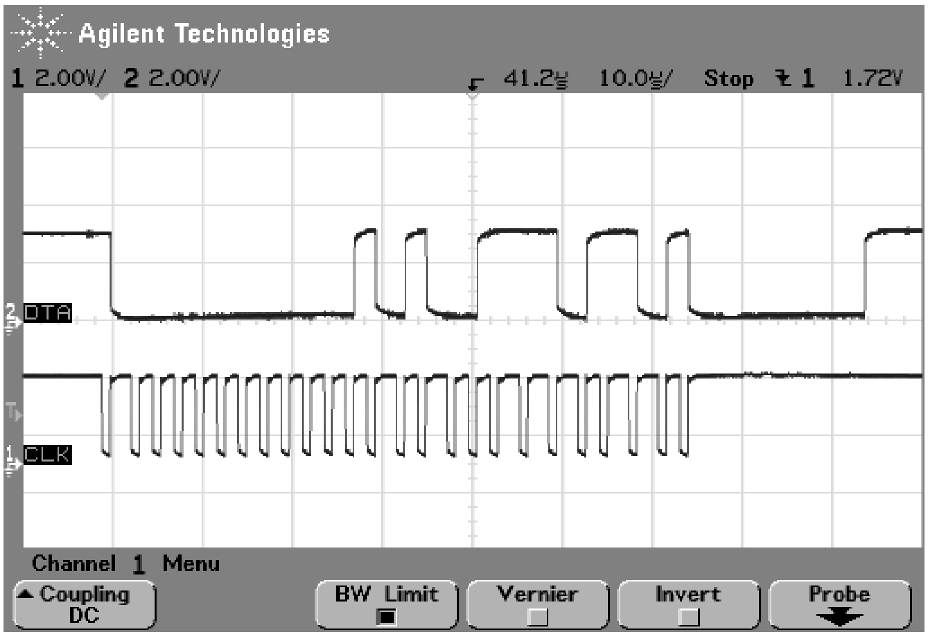


Figure 3.7: Data Transfer (port toggling method)

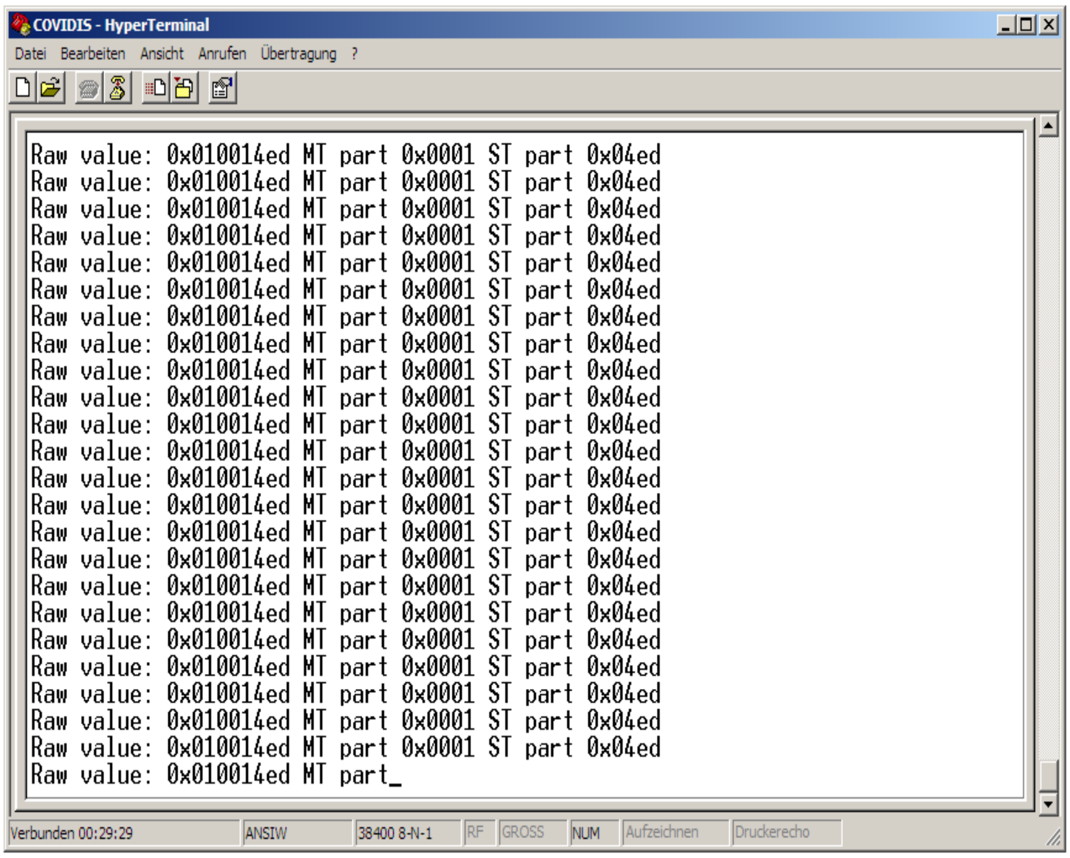
Figure 3.7 shows the transfer of an angle value using the pin toggling method. The value transferred can be read by looking at the DTA signal at the falling edges of the CLK.

For the signal above, this gives:

Bit	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	Multi Turn Value												Single Turn Value											
Bin	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	0	1	1	0	1
Hex	0x	0x0				0x0				0x1				0x4				0xE				0xD			

Table 3.2: Transferred Value

Figure 3.8, shows the evaluation results of this transfer, when send to the PC via the serial port. Since all 32 bits are shown, the total hex word is `0x010014ED`.



```
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed
Raw value: 0x010014ed MT part_
```

Figure 3.8: Screenshot of Evaluation Result.

3.3.2 Calculating an Angle from the Single Turn Value

The single turn value as a hexadecimal number is not very intuitive. To get to an angle in degree, we need to apply the rule of three:

$$Angle = 360 * \frac{Value}{Range}$$

Angle is the result, *value* is the value read from the encoder, and *ranges* the single turn value range. In the above example with 12 bit single turn resolution and a single turn value of 0x4ED (hex 0x4ED=1261 decimal), the result rounded to two decimals would be:

$$Angle = 360° * \frac{0x4ED}{2^{12}} = 360° * \frac{1261}{4096} = 110.83°$$

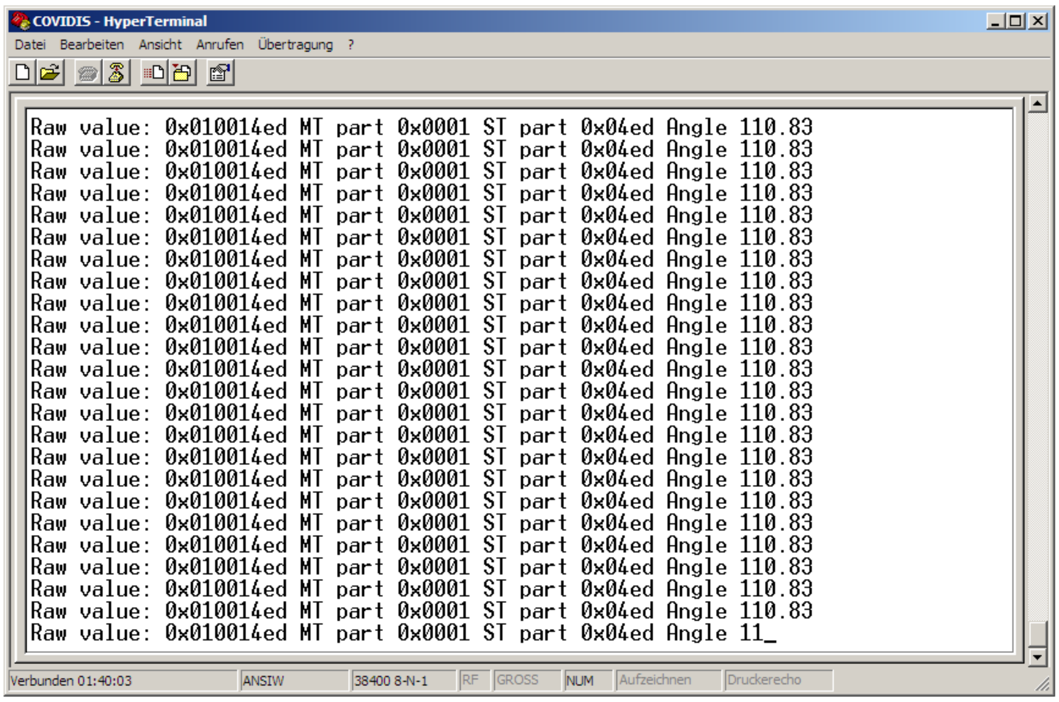
The code for this is straightforward when using floating point arithmetic:

```
uint32_t  u32ssiResult;
uint16_t  u16aux;
uint16_t  u16singleTurn;
uint16_t  u16multiTurn;
double    dAngle;

u32ssiResult = pinToggleReadSSI();

// extract single and multiturn values from the data word
u16singleTurn = u32ssiResult & 0x0FFF;
u16multiTurn  = (u32ssiResult >> 12) & 0x0FFF;
```

Figure 3.9: Code Example: Extracting Multi Turn and Single Turn values



```
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 110.83
Raw value: 0x010014ed MT part 0x0001 ST part 0x04ed Angle 11_
```

Figure 3.10: Screenshot with Angle Result

The code example in Figure 3.9 generates the above sequence. We can continuously monitor the angular value of the encoder.

Appendix a: Complete Software

In addition to the functions already shown above, this code contains the routines used to write the data from the encoder to the UART port of the ATmega88. This data was then captured using Microsoft's Hyperterminal software.

```
/*MH***************************************************************************
*
* Module desc:  SSI demo main file
*               (c) Copyright Fraba-Posital 2010
* History:
*               15Mar2010pme: created
*
******************************************************************************/


/*IN**************************************************************************
* Include files  *************************************************************
******************************************************************************/
#include <avr/io.h>
#include <stdio.h>
#include <stdint.h>


/*TD**************************************************************************
* Type definitions************************************************************
******************************************************************************/
/* add the missing type definitions for standard types */
typedef unsigned char       uint8_t;
typedef signed char         int8_t;
/*
// these are defined in stdint.h
typedef unsigned short      uint16_t;
typedef signed short        int16_t;
*/
typedef unsigned long       uint32_t;
typedef signed long         int32_t;


/*LC**************************************************************************
* Local constants and macros *************************************************
******************************************************************************/
#define USE_PIN_TOGGLING


// defines, so we can change the SSI ports as necessary
#define SSI_CLK_BIT     5
#define SSI_CLK_PORT    PORTB
#define SSI_CLK_DDR     DDRB

#define SSI_DTA_PORT    PIND
#define SSI_DTA_BIT     0
#define SSI_DTA_DIR     DDRD

#define SPI_MOSI_BIT    3
#define SPI_MISO_BIT    4
#define SPI_SS_BIT      2
#define SPI_SCK_BIT     5

// RS232 pins
#define RS232_CTS   4 /* Port D4 */
#define RS232_RXD   0 /* Port D0 */
#define RS232_TXD   1 /* Port D1 */

// RS232 baud rate (assuming 8.0MHz internal RC oscillator)
#define UART_BAUDRATE_9k6     51  // UBRR0L = 51; // 8.0e6/(16*9600)-1;      9600 Baud
#define UART_BAUDRATE_19200   25  // UBRR0L = 25; // 8.0e6/(16*19200)-1;   19200 Baud
#define UART_BAUDRATE_38400   12  // UBRR0L = 12; //                       38400 Baud
```

```
/*--------------------------- C O D E A R E A ------------------------------*/


/*FH*************************************************************************
* Name:         pinToggleReadSSI
* Parameters:   -
* Return value: value read from SSI
* Description:  read a 25 Bit SSI word using pin toggling
****************************************************************************/
uint32_t pinToggleReadSSI( void )
{
    uint8_t  bit_count;
    uint32_t u32result = 0;
    uint8_t  u8portdata;

    for (bit_count=0; bit_count<25; bit_count++)
    {
        // falling edge on clock port
        SSI_CLK_PORT &= ~(1 << SSI_CLK_BIT);

        // left-shift the current result
        u32result = (u32result << 1);

        // read the port data
        u8portdata = SSI_DTA_PORT;

        // rising edge on clock port, data changes
        SSI_CLK_PORT |= (1 << SSI_CLK_BIT);

        // evaluate the port data (port set or clear)
        if ( (u8portdata & (1 << SSI_DTA_BIT)) != 0)
        {
            // bit is set, set LSB of result
            u32result = u32result | 0x01;
        }  // if
    } // for
    return u32result;
}  // pinToggleReadSSI


/*FH*************************************************************************
* Name:         spiInit
* Parameters:   -
* Return value: -
* Description:  init SPI as master for use with SSI
****************************************************************************/
void spiInit( void )
{
    // configure SCK, MOSI and Slave Select as output
    DDRB = (1 << SPI_SCK_BIT) | (1 << SPI_MOSI_BIT) | (1 << SPI_SS_BIT);

    // configure SPI as master, with CLK idle high
    SPCR = (1 << SPE) | (1 << MSTR) | (1 << CPOL);
}  // spiInit
```

```
/*FH*************************************************************************
* Name:         spiReadSSI
* Parameters:   -
* Return value: value read from SSI
* Description:  read a 25 Bit SSI word using the SPI interface
****************************************************************************/
uint32_t spiReadSSI( void )
{
    uint8_t  u8byteCount;
    uint8_t  u8data;
    uint32_t u32result = 0;

    for (u8byteCount=0; u8byteCount<4; u8byteCount++)
    {
        // send a dummy byte, read the result
        SPDR = 0xFF;                           // send 0xFF as dummy
        u32result <<= 8;                       // left shift the result so far
        while ( (SPSR & (1 << SPIF)) == 0);    // wait until transfer complete
        u8data = SPDR;                         // read data from SPI register
        u32result |= u8data;                   // and 'or' it with the result word
    }

    u32result >>= 7;                           // throw aways the LSBs

    return u32result;
}  // spiReadSSI


/*FH*************************************************************************
* Name:         rs232Init
* Parameters:   -
* Return value: -
* Description:  initialize rs232 port pins and peripheral
****************************************************************************/
void rs232Init( void )
{
    // enable the port pullups for RS232
    PORTD |= (1 << RS232_RXD) | (1 << RS232_TXD) | (1 << RS232_CTS);

    // set port directions
    DDRD |= (1 << RS232_TXD);

    // enable rs232 port
    UBRR0L = UART_BAUDRATE_38400; // Set Baudrate
    UCSR0A = 0x40;                // clear TXCE bit, set everything else to 0
    UCSR0B = 0x18;                // enable receiver and transmitter
    UCSR0C = 0x86;                // no parity, 8bits

}  // rs232Init


/*FH*************************************************************************
* Name:         rs232send
* Parameters:   cbuffer - pointer to 0-terminated string (char buffer)
* Return value: -
* Description:  sends the string by rs232
****************************************************************************/
void rs232send( char *cbuffer )
{
    while (*cbuffer!= 0)
    {
        UDR0 = *cbuffer;
        cbuffer++;
        asm( " wdr" );
        while ( ( UCSR0A & ( 1<< UDRE0)) == 0 );
    }
}  // rs232send


/*FH*************************************************************************
* Name:         main
* Parameters:   -
* Return value: -
* Description:  entry point and main loop
*************************************************************************/
int main()
{
    int i;
    uint32_t  u32ssiResult;
    uint16_t  u16aux;
    uint16_t  u16singleTurn;
    uint16_t  u16multiTurn;
    char      cBuffer[32];
    double    dAngle;

    // init the rs232 interface to the PC
    rs232Init(); asm (" wdr ");
    rs232send( "Hello\n" ); asm (" wdr ");

#ifdef USE_PIN_TOGGLING
    // enable clock output, set to high
    SSI_CLK_DDR   |= (1 << SSI_CLK_BIT);    // CLK is output
    SSI_CLK_PORT  |= (1 << SSI_CLK_BIT);    // set to high (idle state)
#else
    spiInit();
#endif

    // forever loop
    for (;;)
    {

        // get the SSI word
#ifdef USE_PIN_TOGGLING
        u32ssiResult = pinToggleReadSSI();
#else
        u32ssiResult = spiReadSSI();
#endif

        // extract single and multiturn values from the data word
        u16singleTurn = u32ssiResult & 0x0FFF;
        u16multiTurn  = (u32ssiResult >> 12) & 0x0FFF;

        // calculate the single turn angle
        dAngle = (double) u16singleTurn;    // make the value floating point
        dAngle = 360.0 * dAngle / 4096.0;   // calculate actual angle


        // send the entire unmodified 32bit word to the PC in hex format
        rs232send( "Raw value: " );
        u16aux = (u32ssiResult >> 16);
        sprintf( cBuffer, "0x%04x", u16aux );
        rs232send( cBuffer );
        u16aux = (u32ssiResult  & 0xFFFF);
        sprintf( cBuffer, "%04x ", u16aux );
        rs232send( cBuffer );

        rs232send( "MT part " );
        sprintf( cBuffer, "0x%04x ", u16multiTurn );
        rs232send( cBuffer );

        rs232send( "ST part " );
        sprintf( cBuffer, "0x%04x ", u16singleTurn );
        rs232send( cBuffer );

        // we do not have sprintf for float, so first print the integer part
        u16aux = (uint16_t) dAngle;
        rs232send( "Angle " );
        sprintf( cBuffer, "%u", u16aux );
        rs232send( cBuffer );

        // then calculate two decimal places of the fractional part
        dAngle = (dAngle - u16aux) * 100;
        u16aux = (uint16_t) dAngle;
        sprintf( cBuffer, ".%u", u16aux );
        rs232send( cBuffer );
        // send CR/LF
        rs232send( "\r\n" );

        // delay at least 25µs for SSI timeout
        for (i=0; i<1000; i++)
        {
            asm( " nop " );  // prevents the optimizer from removing the loop
        }

    }

}


/****************************************************************************
* End of source module
****************************************************************************/
```

Appendix B: References

[1] Atmel, ATmega88 datasheet

[2] Maxim, Max1486 datasheet

History of Changes

- March 15th , 2010: First Version
- July 9th,2010 : Second Version
- August 2013 : minor changes and updated links