AVR320: Software SPI Master

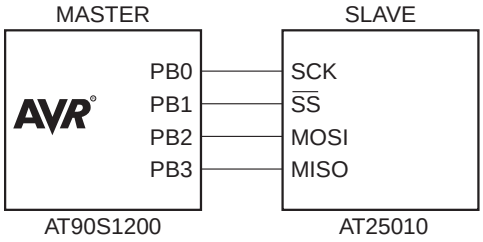
Features

- **Up to 444K Bytes/Sec Throughput @ 10 MHz**
- **Directly Supports Large Block Writes**
- **Easily Expandable for Multiple SPI Slaves**
- **Operates in SPI Mode 0**
- **16-bit Data, Easily Modified to 8-bit Operation**
- **No Interrupts Required**
- **Low Register Count; Only 3 Needed (2 if 8-bit Mode)**
- **Small Code Size; 35 Words, Including Initialization**
- **Interfaces to Atmel AT25xxx Serial EEPROM**

Introduction

The Synchronous Peripheral Interface (SPI) standard is gaining rapidly in popularity. It allows faster communication than I^2C and can be implemented using fewer gates which means lower silicon costs. Although the larger AVR family members include a fully-functional SPI interface, the smaller members do not. This application note describes a set of low-level routines for software implementation of the SPI protocol, in Master Mode (all communications originate from the AVR). These can be used as the basis for communicating with Atmel's 25XXX family of serial EEPROM memories, as well as a host of other peripheral IC's such as display drivers.

Figure 1. Connection between the AVR MCU and the slave device shown as a serial EEPROM

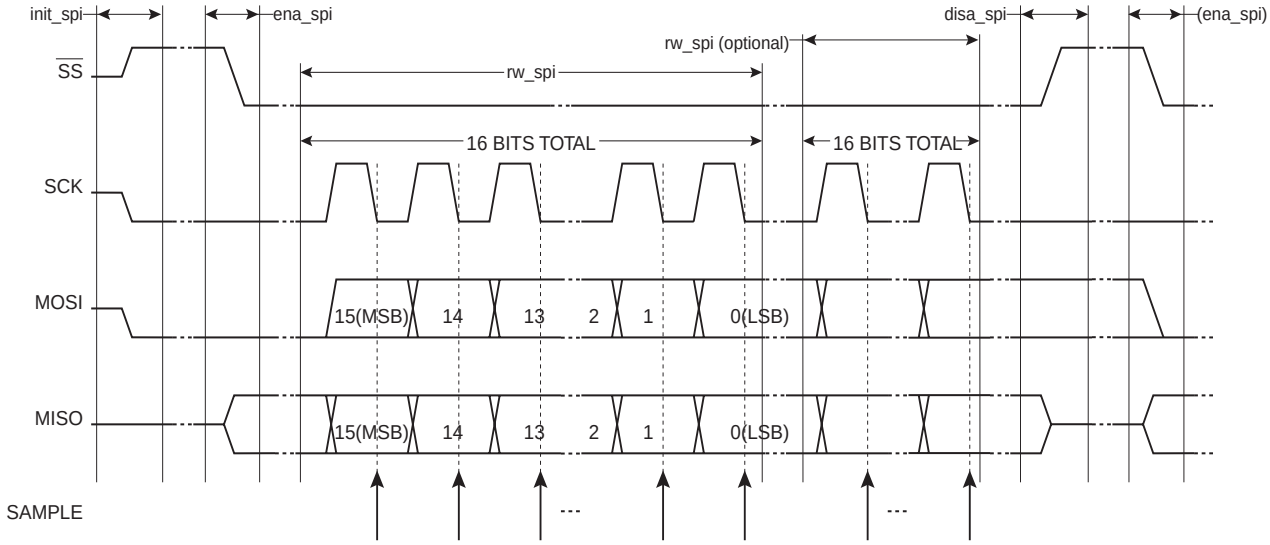


Software SPI Master

Application Note

Rev. 1108A–09/98

Figure 2. SPI Transmission Timing



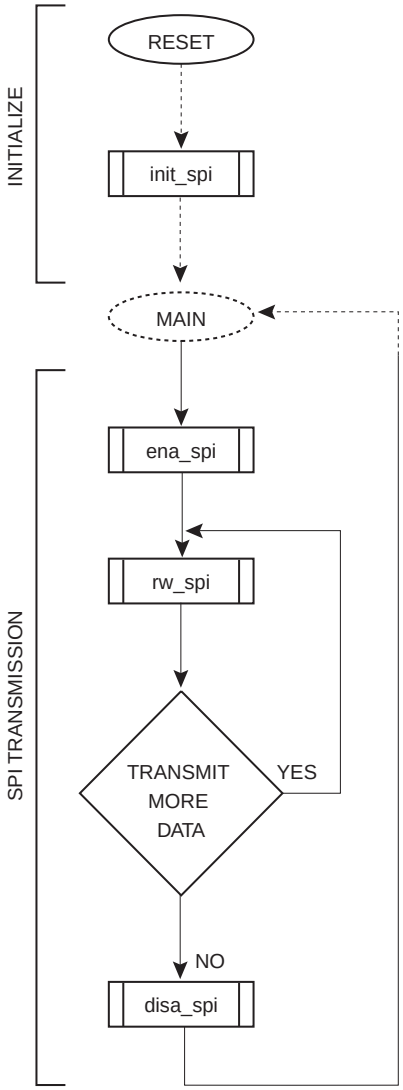
Theory of Operation

SPI Mode 0 implies the following conditions:

1. The SCK signal is low when idle,
2. MOSI data (from the AVR) must be stable at some time (setup) before the rising edge of the clock,
3. MISO data (from the slave) is valid at some time (t_{VALID}) after the falling edge of the clock,
4. Data is always sent MSB first.

These routines have been written to meet those standards, with special attention being paid to the boundary conditions, such as t_{VALID} and t_{SETUP}. These timings are detailed later in this application note.

Figure 3. SPI Master Flowchart



Subroutine Description

init_spi:

This routine initializes the SPI port lines. The macros will need to be modified if Port B is not used; otherwise, simply changing the Port Definition .EQU's for the SCK, MOSI, MISO and NSS (not-Slave-Select) is adequate. This routine has no entry requirements and returns nothing.

ena_spi:

This routine makes sure that SCK is low before setting $\overline{SS}$ to the active state. This routine has no entry requirements and returns nothing.

disa_spi:

This routine brings the $\overline{SS}$ signal high (inactive). It should be called when a transmission sequence is complete, to prevent spurious clocking of the SPI slaves. It also has no entry requirements and returns nothing.

rw_spi:

This routine sends/receives either an 8-bit or 16-bit data word, depending on whether the user has modified the source code. It leaves the SCK signal low upon exit, and does not modify the $\overline{SS}$ signal; therefore, many back-to-back writes are possible by simply calling this routine repeatedly. Entry requirements are simply that the spi_lo (and spi_hi, if used in 16-bit mode) are initialized with the data to be sent, prior to calling this routine. Upon exit, the same register(s) contain the data received from the SPI slave.

Conversion to 8-bit Operation

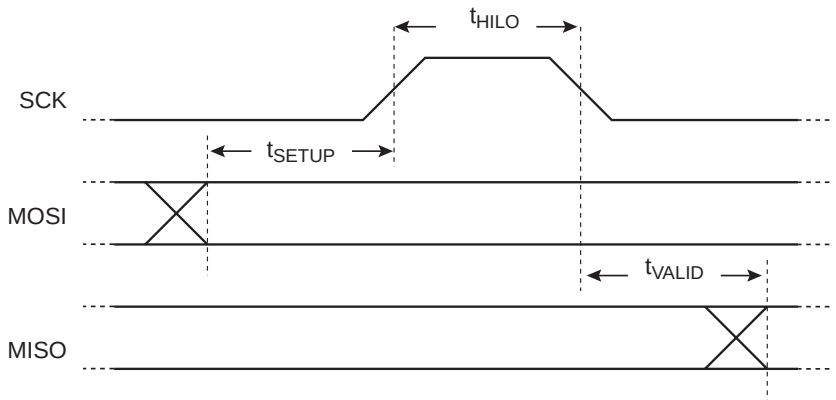
There are only two lines of code that require modification in order for the appliction note to work with 8-bit data words. Both of the changes are in the rw_spi routine. First, the bit counter initialization value must be changed from 16 to 8; secondly, the line "rol spi_hi" must be commented out. No other changes are necessary, except that the spi_hi register is no longer needed and is never used.

Macros

Macros are used to make the code more readable. It is assumed that Port B will be used for the SPI interface; if this is not the case, then the macros themselves must be modified to reflect the correct port. Pins for the 4 SPI signals are listed under "PORT DEFINITIONS" as .EQU's and are easily modified.

The delay function macros deserve some additional explanation. To help conserve registers, this routine gets double-duty out of the TEMP register by keeping a 5-bit bit-counter value and the delay counter for measuring the SCK high and low time. The latter are kept in the uppermost bits. By simply incrementing the upper 3 bits and watching for a rollover, we can track time without affecting the lowest 5 bits. Note that, in reality, we are subtracting values rather than adding; the end result is the same, except that Carry is cleared (rather than set) when the value of the upper 3 bits rolls from 7 to 0.

Figure 4. Setup & Hold Timing



Delay from MOSI Update to SCK Rising Edge

The delay period from updating MOSI to the rising edge of SCK is a critical period, as it is actually the data setup time (t_{SETUP}) for the connected peripheral. In this routine, the amount of setup time is 2 clock cycles if MOSI is changing to low, 3 if changing to high (this skew allows for different hi/lo drive levels as well). At a 10 MHz clock, this equates to 200-300 ns. If more time is required, NOPs should be placed immediately before the sck_hi statement.

SCK Duty Cycle

SCK will spend most of its time low, due to the fact that it is low during the “overhead” portions of the send loop. However, if the connected peripheral can handle a faster rate, the high-time delay (t_{HILO}) can be reduced to match the minimum SCK-high specification.

With a delay value of 1 being used, the routine requires approximately 22.5 AVR clocks per data bit, and the SCK-high time is exactly 4 clocks. At 10 MHz, this would correspond with an equivalent SCK-high spec of 400 ns, and an overall throughput of 444K bytes/sec.

The low-time delay must be set to meet the “SCK fall to MISO (slave data out) Valid” period (t_{VALID}), which is determined by the peripheral. With a delay value of 1, this routine will have 3 AVR clocks delay before latching the MISO signal into the pin’s synchronization register. Again, at 10 MHz, this is a 300 ns period, and can be adjusted upward by increasing the delay value.