AVR 8-bit Microcontrollers

AVR319: Using the USI Module for SPI Communication on tinyAVR and megaAVR Devices

APPLICATION NOTE

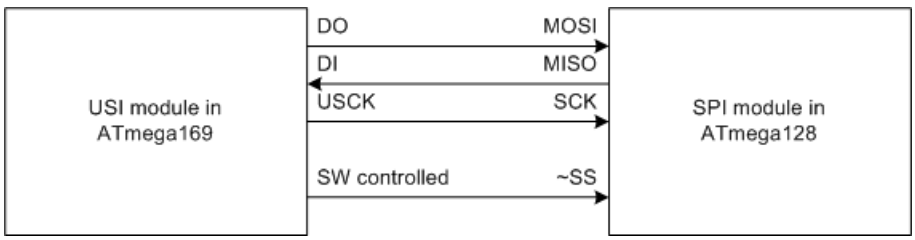
Introduction

The Serial Peripheral Interface (SPI) allows high-speed synchronous data transfer between an Atmel® AVR® device and peripheral devices, or between several AVR devices. The strength of the SPI bus includes ease of use, high communication speed, and a vast amount of peripheral devices supporting it.

The Universal Serial Interface (USI) module on devices like Atmel ATmega169, ATtiny26, and ATtiny2313 has a dedicated Three-wire mode. The USI provides the basic hardware resources needed for synchronous serial communication. Combined with a minimum of control software, the USI allows higher transfer rates, less CPU load, and in general uses less code space than solutions based on software only.

This application note describes an SPI interface implementation, in form of a full-featured driver and an example of usage for this driver. The driver handles transmission according to SPI Modes 0 and 1.

Figure -1. Example Application Setup



Feature

- C-code driver for SPI master and slave
- Use the USI module
- Supports SPI Mode 0 and 1

Table of Contents

1. Theory

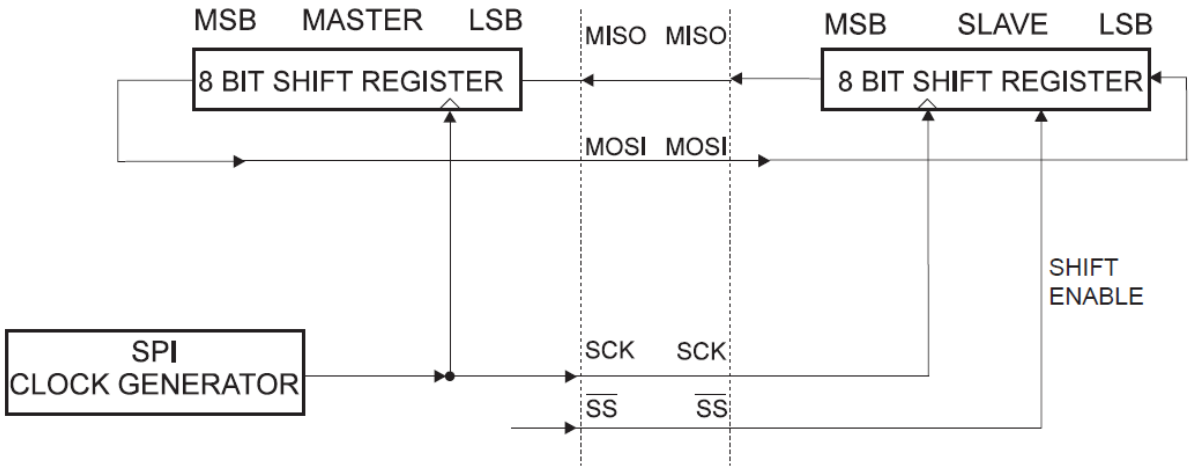
This chapter gives a short description of the SPI interface and the USI module. For more detailed information, refer to the datasheets.

2. Serial Peripheral Interface

The Serial Peripheral Interface allows high-speed synchronous data transfer between an AVR device and peripheral devices, or between several AVR devices.

The interconnection between Master and Slave devices with SPI is shown in the figure below. The system consists of two shift registers and a master clock generator. The SPI Master initiates the communication cycle when pulling low the Slave Select (SS) pin of the desired Slave. Master and Slave prepare the data to be sent in their respective shift registers, and the Master generates the required clock pulses on the SCK line to interchange data.

Figure 2-1. SPI Master-slave Interconnection



3. SPI Data Modes

The four combinations of SCK phase and polarity with respect to serial data are determined by clock phase (CPHA) and clock polarity (CPOL) settings. The SPI data transfer formats are shown in the figures below. Data bits are shifted out and latched in on opposite edges of the SCK signal, ensuring sufficient time for data signals to stabilize.

Figure 3-1. SPI Transfer Format with CPHA = 0

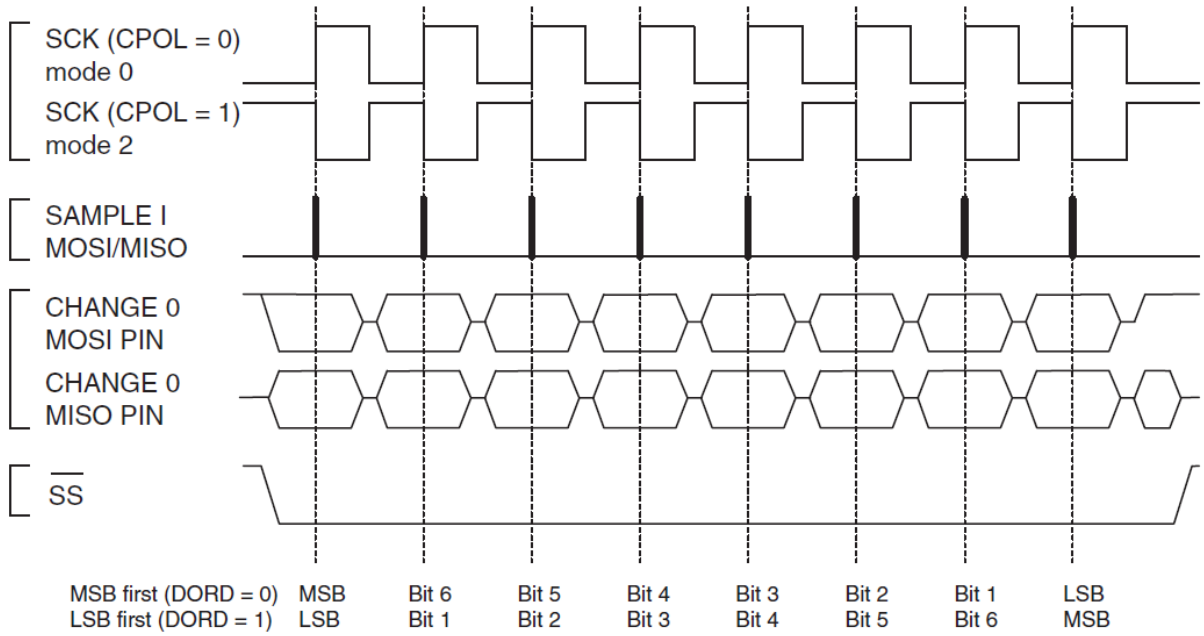
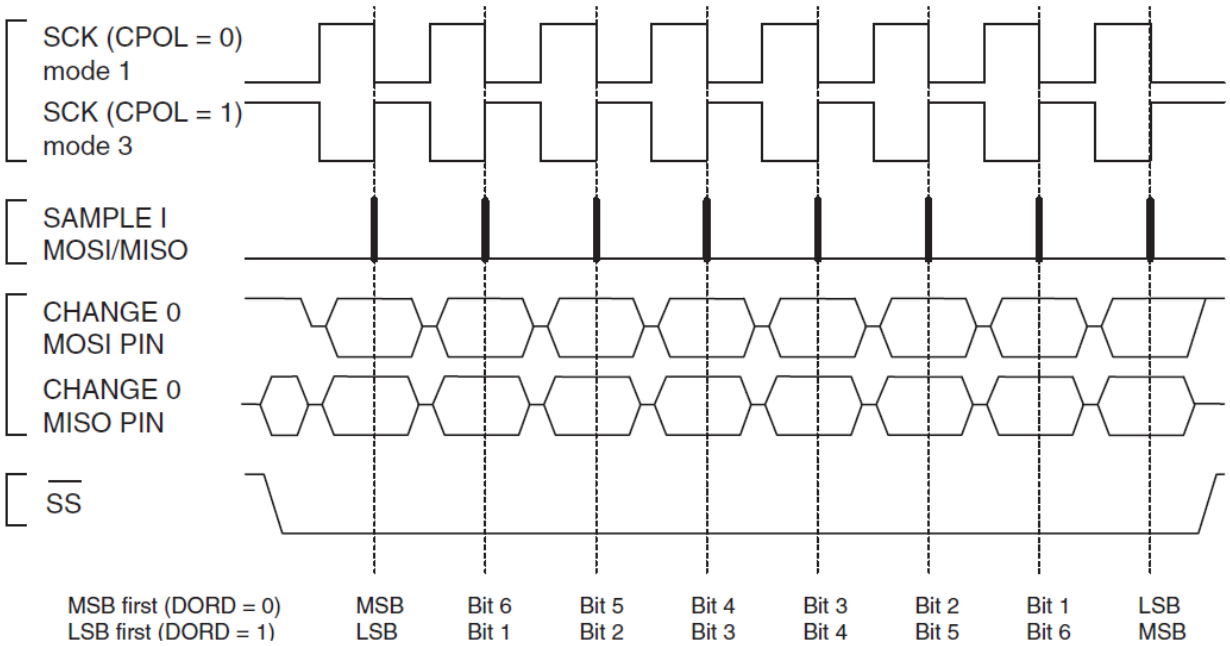


Figure 3-2. SPI Transfer Format with CPHA = 1



4. Universal Serial Interface

The Universal Serial Interface provides the basic hardware resources needed for synchronous serial communication. The main features of the USI are:

- Two-wire Synchronous Data Transfer
- Three-wire Synchronous Data Transfer
- Data Received Interrupt
- Wake-up from Idle Mode

The USI Three-wire mode is compliant with the Serial Peripheral Interface (SPI) Mode 0 and 1, but does not have the Slave Select (SS) pin functionality. However, this feature can be implemented in software if necessary. The first figure below shows the USI module block diagram, and the second figure below shows the module in Three-wire mode.

Figure 4-1. Universal Serial Interface, Block Diagram

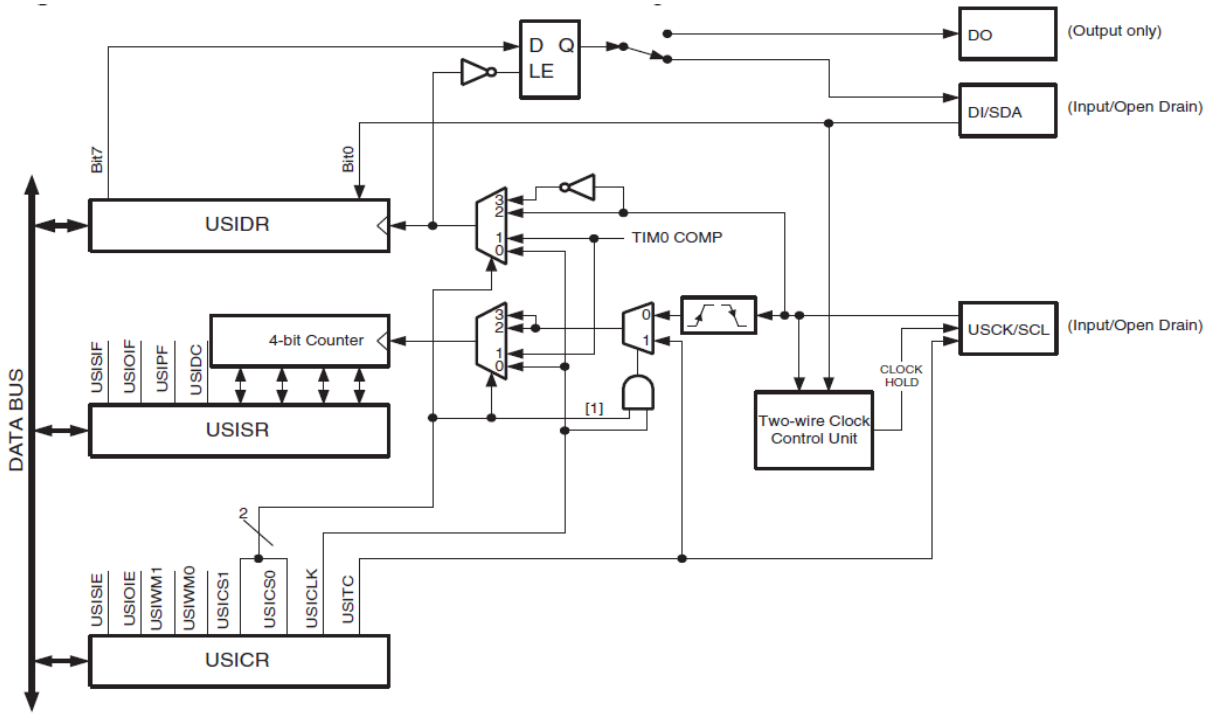
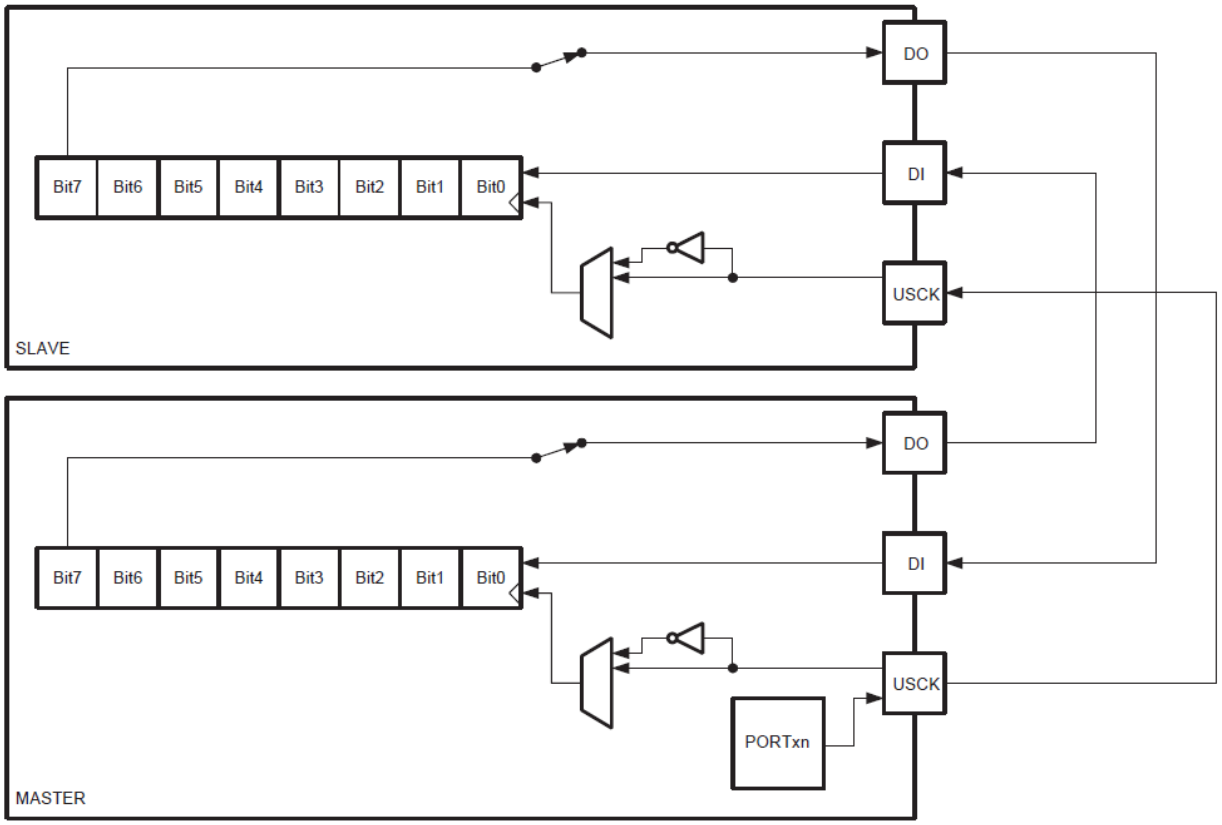


Figure 4-2. Three-wire Mode Operation, Simplified Diagram



The USI Data Register (USIDR) is an 8-bit Shift Register that contains the incoming and outgoing data. The register has no buffering so the data must be read as quickly as possible to ensure that no data is lost. The USI Status Register (USISR) contains a 4-bit counter. Both the shift register and the counter are clocked simultaneously by the same clock source. This allows the counter to count the number of bits received or transmitted and sets a flag, alternatively generates an interrupt, when the transfer is complete. The clock can be selected to use three different sources; The USCK pin, Timer/Counter0 Compare Match, or from software.

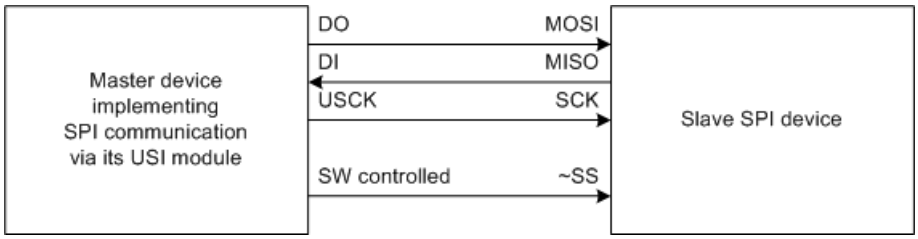
The figure above shows two USI units operating in Three-wire mode, one as Master and one as Slave. The two shift registers are interconnected in such way that after eight USCK clocks, the data in each register are interchanged. The same clock also increments the USI's 4-bit counter. The Counter Overflow Interrupt Flag, or USIOIF, can therefore be used to determine when a transfer is completed. The clock is generated by the Master device software by toggling the USCK pin via the PORT Register or by writing a one to the USITC bit in USICR.

It is the master device's responsibility to give the slave device time to prepare its next byte before starting a new transfer.

5. Implementation

This application note describes the implementation of an SPI driver for both master and slave communication. An example setup with the USI module as an SPI master is shown in the figure below. The driver is written as a standalone driver that easily can be included into the main application. Use the code as an example, or customize it for own use. All relevant functions and global variables are prefixed with ‘spiX_’, so a quick search-and-replace is enough to rename the driver interface in case of naming conflicts.

Figure 5-1. USI Module Setup as SPI Master



The driver uses the USI module and the USI counter overflow interrupt. Therefore, interrupts must be enabled to be able to use the driver. In master mode the driver also uses Timer/Counter0. The T/C0 compare match interrupt is used to generate the master clock signal.

The driver interface consists of these functions:

- *spiX_initmaster*, which initializes the driver in master mode
- *spiX_initslave*, which initializes the driver in slave mode
- *spiX_put*, which starts a transfer in master mode, or prepares a byte in slave mode
- *spiX_get*, which returns the last incoming byte
- *spiX_wait*, which waits for a transfer to finish

Note that the Slave Select (SS) line of the SPI bus must be controlled manually in software if required by the slave device. When using the USI as an SPI slave, you also need to watch the incoming SS line with, for instance, an interrupt line if required. The driver in this application note does not use the SS line.

The following global variables are also available:

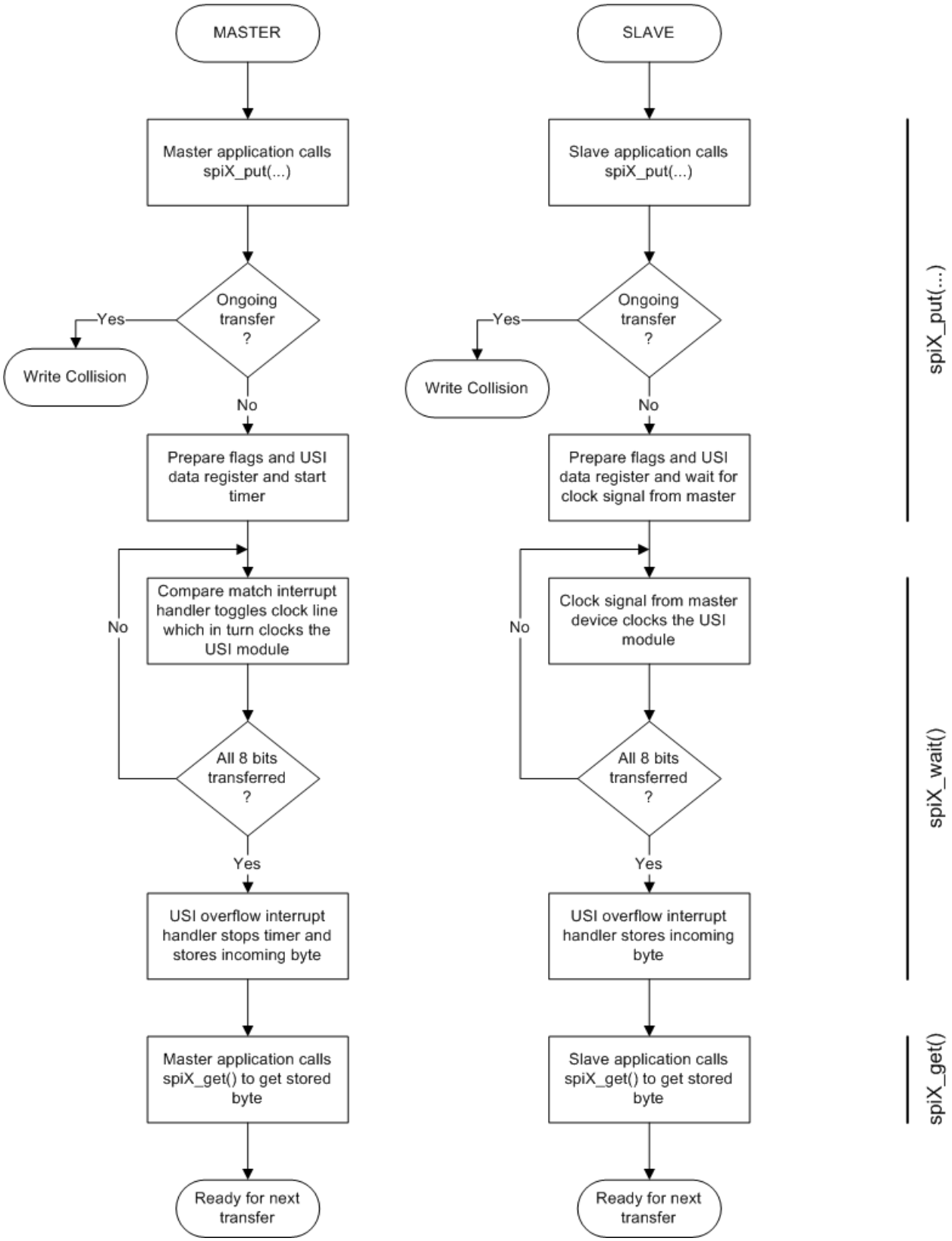
- *spiX_status*, which contains the driver status flags
- *storedUSIDR*, which contains the last incoming byte. This should be accessed with the spiX_get() function only.

The functions are documented in the source code. In addition, the simplified flowchart for a typical byte transfer session is given in the figure below. The example source code accompanying this application note shows an implementation of such a session.

Note that the part of the flowchart below grouped under the *spiX_wait()* function is performed by interrupt handlers and not the function itself. The function just waits for the transfer complete flag to be set.

The example code is written for Atmel START. It can be downloaded from "BROWSE EXAMPLES" entry of Atmel START for both Atmel Studio 7 and IAR IDE. Double click the downloaded .atzip file and the project will be imported to Atmel Studio 7. To import the project in IAR, refer "Atmel START in IAR" , select Atmel Start Output in External Tools -> IAR.

Figure 5-2. One Byte Transfer in Master and Slave Mode



6. Literature References

- ATmega169 Datasheet
- Application Note AVR310: Using the USI Module as a I^2C Master

7. Revision History

Doc Rev.	Date	Comments
2582B	08/2016	The example firmware is ported to Atmel START.
2582A	04/2009	Initial document release.

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**

 / Rev.: Atmel-2582B-Using-the-USI-Module-for-SPI-Communication-on-tinyAVR-and-megaAVR-Devices_AVR319_Application Note-08/2016