Software SPI Master Implementation

Features

- Configurable number of bits to transfer
- Supports high baud rate (1M bits/sec with a 32 MHz master clock)
- MOSI line setting synchronized with the falling edge of the clock
- MISO line sampling synchronized with the rising edge of the clock
- Memory size: less than 0.5K bytes (90 Code Words - 10 Data Words in the stack, plus the relevant buffers)

Introduction

The need for a simple and cost effective serial link for use in consumer, telecommunications and industrial electronics, has led to the development of the Serial Peripheral Interface (SPI). Today the SPI is implemented in a large number of peripheral devices and microcontrollers. The AT91M40400 does not have dedicated hardware for the SPI, but because of the high processing speed and flexible Timer Counter (TC), an effective software SPI implementation can easily be performed. The AT91M40400 is capable of a high transfer speed (1M bits/sec with a 32 MHz master clock).

Theory of Operation

The SPI is a synchronous serial interface consisting of one transmit data line (MOSI), one receive data line (MISO), and one clock (SCK) line. One or several chip selects (CS) are used to select the device to be interfaced.

The SPI is a master/slave interface. The master drives the CS, the SCK and the MOSI lines while the slave(s) drive the MISO line. Transmission and reception are performed at the same time. Both the master and the slave(s) set the data bit to transmit on one edge of the clock, and sample the received data bit on the other edge of the clock. The corresponding edge depends on the application, and is programmable. The Chip Select must select, by windowing the packet, the slave for the complete transfer. The number of bits transferred depends on the application, and the MSB is always transferred first. When the transfer is complete (all data bits are transferred), the level (inactive polarity) of the clock signal depends on the application and is programmable.

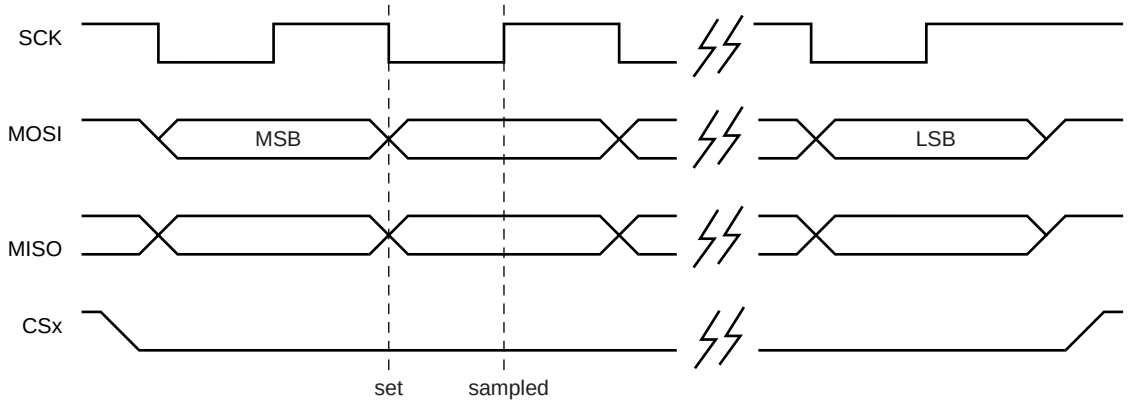
In this application note it is assumed that the data is set on the falling edge and sampled on the rising edge of the clock, and the inactive level of the clock is 1 (one). On both sides, master and slave, the first bit is set on the first falling edge of the SCK signal following the high-to-low transition of the chip select, and sampled on the rising edge. When the transfer is complete, all the signals must be left at a high level.

AT91
ARM Thumb® Microcontrollers

Application Note

Rev. 1155A–08/98

Figure 1. SPI Operation



AT91M40400 Implementation

This application note concerns the master. It describes how to generate the clock signal and manage the MOSI and MISO lines in phase with the edges of SCK. It does not describe the management of the Chip Select which can be performed by using a PIO cleared before the transfer and set after.

In order to operate the SPI, the master must perform the following:

- generate an SCK signal with a square signal (duty cycle = 1)
- set the outgoing data bit on the MOSI line on the falling edge of the SCK
- sample the incoming data bit on the MISO line on the rising edge of the SCK

To meet these requirements, two channels of the Timer Counter (TC) are used and the configuration is as follows:

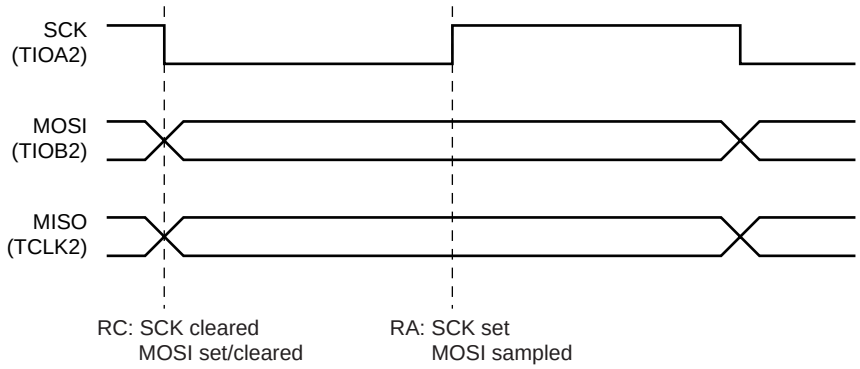
TIOA2	SCK	output
TIOB2	MOSI	output
TCLK2	MISO	input

Channel 2 is used in a double waveform mode to manage the outgoing SCK and MOSI signals. The clock source is MCK/2. The square wave is performed by programming the TC_RC (Register C) with the bit time value, and the TC_RA (Register A) with the half value. The TC_RB (Register B) is unused. TIOA2/SCK is cleared on the software trigger event (for the first bit) and on the RC compare event, and set on the RA compare event. TIOB2/MOSI is set/cleared on the software trigger event (for the first bit) and on the RC compare event.

Channel 1 is used in a single waveform mode (capture also could be) to manage the incoming MISO signal. The clock source is the XC1 input to which TIOA2/SCK is internally connected, and the XC2, to which TCLK2/MISO is internally connected, is used as a burst on the clock. When the incoming bit data is set (1), the clock is enabled and the counter is incremented on the rising edge of SCK. When the incoming bit data is cleared (0), the clock is disabled and the counter is unchanged on the rising edge of SCK.

Figure 2 shows the action of the timer channels on the SPI signals.

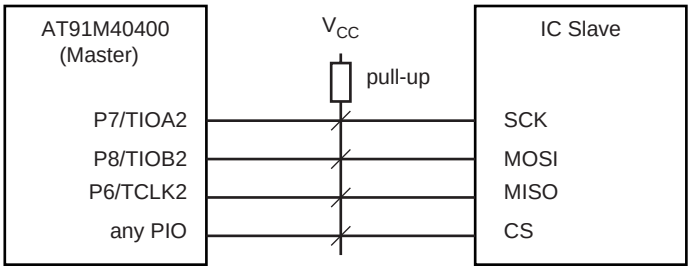
Figure 2. Action of Timer Channels on SPI Signals



Connection

At reset, all PIOs of the AT91M40400 are programmed as input. Therefore, each line must be connected to the supply voltage via a pull-up resistor.

Figure 3. PIO Connection



See the AT91M40400 datasheet for more details concerning the value of the pull-up resistors.

Software SPI

The software SPI described in this application note is written with the 32-bit ARM assembler for performance aims. No more than 400 bytes of code are needed. The RAM usage is limited to the Rx and Tx buffers provided by the application, and to the space necessary to push the registers used.

The implementation of the SPI master device presented in this application note is divided into two modules: an initialization routine (executed to program the timer channel registers for SPI use) and a transfer routine.

Initialization Routine ‘spi_tc_init’

The routine ‘spi_tc_init’ performs the necessary initialization of channels 1 and 2 of the Timer Counter for subsequent use of the routine ‘spi_tc_transfer’ which performs the data bit transfer.

The C prototype of this function is:

```
void spi_tc_init (unsigned int period);
```

“period” is the bit time to be programmed in TC_RC. If null, the default value is used to provide a baud rate equal to the master clock divided by 32. Note that the channel is initialized to be clocked by the master clock divided by 2 (MCLK / 2).

The initialization pseudo code is:

```
Begin
| Configure SPI lines as PIO
| TCC1 and 2 - Disable current mode :
| . disable all interrupt source
| . Disable the clock
| . clear mode and counter registers
| . clear events
| Initialise TCC1 :
| . Enable TIOA2 (SCK) as the channel clock
| . Initialize the Mode Register
| . Enable and trig the clock
| Set Default baud rate (1Mbit/sec @ 32MHz) if applicable
| Initialise TCC2 :
| . Setup the Mode Register
| . Initialize the A, B and C counter registers
| . Enable the clock and trig to set the output signals
| . Disable the clock
| Configure SPI lines as peripheral
End
```

The pins used to emulate the SPI signals (TIOA2, TIOB2 and TCLK2) are first configured as PIO. Then the channels are initialized (channel 2 is started then stopped to configure the TIOA and TIOB lines), and the pins are set as peripherals. This allows the internal variables of the timer channel to be set without having any effect on the signals at reset.

The RC register is programmed with the “period” value. The bit time is obtained by trigging on RC compare. The square wave is obtained by programming the RA register with the half of the period.

Transfer Routine ‘spi_tc_transfer’

The routine ‘spi_tc_transfer’ validates the SCK signal (square wave on TIOA2) and, for each rising edge of SCK, programs the mode register of channel 2 to set/clear the MOSI signal (TIOB2) corresponding to the bit transferred on the next falling edge. It then checks the value of the channel 1 counter to initiate the reception. When all bits are transmitted, it programs the mode register to leave the SCK and MOSI signals set.

The C prototype of this function is as follows:

```
void spi_tc_transfer ( unsigned char *tx, unsigned char *rx, unsigned int size ) ;
```

“tx” is the address of the Tx buffer (bytes to send)

“rx” is the address of the Rx buffer (bytes to be received)

“size” is the byte number to transfer

The transfer is performed byte by byte, from lowest address to highest address and, for each byte, Most Significant Bit (MSB) first. If the bit number is not a boundary of 8, the non significant bits are located in the lowest bits of the last byte.

The transfer pseudo code is:

```
Begin
| Initialize the Tx and Rx masks (0x80808080)
| SCK falling edge at software trig and RC Compare
| Initialize the received byte (0)
| Get first byte to transmit
| TIOB (MOSI) set/cleared at SWTRG following first bit
| Decrement the bit number (1st bit set at SWTRG)
| Read TC2 Status (clear events)
| Enable and trig TC2 Clock
| While (all bits not transferred)
| | If bit before last one
| | | force TIOA2 (SCK) unchanged at RC compare
| | | force next bit sent to be one
| | Endif
| | Prepare next Tx bit : Right Shift Tx Mask
| | If (last bit of the byte)
| | | get next byte
| | EndIf
| | Wait RA compare (rising edge of SCK)
| | TIOB (MOSI) set/cleared at next RC compare following current bit
| | If (TC1 Counter Value modified)
| | | Set Rx Data Bit (current bit indicated by the Rx mask)
| | EndIf
| | Prepare next bit to receive : Right Shift Rx Mask
| | If (last bit of the byte)
| | | store received byte
| | | Initialize the received byte (0)
| | EndIf
| | Decrement the bit number to transfer
| EndWhile
| store last byte (if needed)
| Wait end of last bit ( RC Compare )
```

```
    | Disable TC2 Clock
    End
```

In the first stage, the function prepares the register to transfer whole bits. The first transmitted bit is set on the software trigger of channel 2.

Once the transfer has started and for each remaining bit, the function first waits for the SCK rising edge (RA compare), programs TC_CMR (Channel Mode Register) to set/clear the MOSI signal on the next SCK falling edge (RC compare), checks the state of the MISO signal when the SCK rising edge occurred, then checks for the end of transmission.

Two different masks are used: one for transmission and one for reception. These masks allow the current bit to be isolated, and the next byte to be identified. This avoids having to use a bit per byte counter. These masks are initialized with the value 0x80808080 and are right rotated (ROR) after each bit transfer. The ROR instruction causes bit 0 to be copied in bit 31 (mask is restored for next byte) as well as in the Carry bit of the Current Program Status Register (CPSR). Consequently, the CPSR.C bit is set after every eigth bit and cleared after any other bit. The following action for the next byte can then be conditionally performed (e.g. strCSb).

For transmission, the first bit is set/cleared at the software trigger. For all others bits, two different registers are predefined with the values of TC_CMR to set or clear the MOSI. The current bit value is tested by “ANDing“ the byte in progress with the Tx mask. The two predefined registers are copied into TC_CMR by using instructions with opposite conditional codes (streq and strne). After TC_CMR has been programmed for the last bit, the predefined registers are both forced with the same value which leaves the SCK and MOSI signals set.

For reception, as the MISO line is used to enable the clock, the channel 1 counter value is incremented only if the MISO is set. Therefore, a modified value indicates a 1 (one), and an unmodified value indicates a 0 (zero). As each current received byte is pre-initialized with the “0” value, for each bit, the byte value is conditionally “ORed” (orrne) with the Rx mask only if the channel 1 counter value is modified (MISO = 1).

Performances

Parameter	Value
Code Size	89 words
Register Usage	R0 to R11 and R14
Peripherals Usage	3 I/O Pins, Timer counter 1 and 2
Main Loop Time	between 29 and 31 master clock cycles per bit

Register Usage

Register	Entry Parameter
r0	Address of data to be transmitted
r1	Address of data to be received
r2	Number of bits to transfer

Register	Variable
r3	Current byte sent
r4	Current byte received
r5	Tx transfer bit mask
r6	Rx transfer bit mask
r7	TC2 mode register with BCPC to set TIOB
r8	TC2 mode register with BCPC to clear TIOB
r9	TC2 mode register with TIOA and TIOB set
r10	TC1 counter register
r11	TC block base address
r14	Working register

Tips and Warnings

Memory Accesses

The higher speed can be reached only if the code and the data are located in a 32-bit space accessible with 0 wait states. Therefore, it is preferable to use an on chip memory.

Transfer Size

The SPI routine presented can be reduced in size and in execution time if the application guarantees that the number of bits is less than or equal to 32. In this case, the C prototype of this function becomes :

unsigned rx spi_tc_transfer (unsigned int tx, unsigned int size);

“tx” is the profile to send

“size” is the bit number to transfer

The function returns the received profile.

The Rx and Tx masks can then be calculated with the formulae:

```
mov     r5,#1
mov     r5,r5,lsl r2
mov     r5,r5,rrx
mov     r6,r5
```

The end of transmission can be detected when the carry is set by the Tx mask right shifting:

```
movs    r5, r5, ROR #1
biccs   r7, r7, #ACPC
movcs   r8,r7
```

The end of reception can be detected when the carry is set by the Rx mask right shifting:

```
movs    r6, r6, ROR #1
```

```
        bcc     RepeatBegin
```

Interrupt Management

If the application does not need a high baud rate, the RA compare can be detected by an interrupt, and the bit treatment can then be executed in the Interrupt Service Routine. This allows other processes to be activated while the bits are being transferred.

MISO Sampling

If the application on the slave side guarantees that the MISO line is held for 20 master cycles after the SCK rising edge, this signal can then be sampled by reading the pin state register of the PIO controller. This allows Timer Channel 1 to be freed.

Limitation

If used with a high baud rate, the transfer procedure can not be interrupted. Therefore, it is mandatory to disable the interrupts by setting bits I and F in the CPSR before calling the function “spi_tc_tranfer”.