SPI and Microwire/Plus Virtual Peripheral Implementation

Application Note 20
October 2000

1.0 Introduction

This document outlines the hardware and software needed to do SPI (SPI is a trademark of Motorola Semiconductor) with the SX communications from Ubicom (Virtual Peripheral is a trademark of Ubicom). It also describes the SPI bus and its implementation on the SX device using the concept of Virtual Peripheral. The software modules may be used with other Virtual Peripheral modules from Ubicom and with your own application code.

Synchronous serial interfaces are widely used to provide economical board-level interface between different devices such as microcontrollers, DACs, ADCs and other. Although there is no single standard for the synchronous serial bus, there are industry accepted guidelines based on the two most popular implementations: SPI and Microwire/Plus (Microwire/Plus a trademark of National Semiconductor). SPI and Microwire/Plus are often referred to as a "3-wire" bus. Since both standards define only the communication lines and the clock edge, other parameters vary for different devices. This document describes the operation of SPI and details on how it can be implemented on a SX by the use of SPI Virtual Peripheral software modules.

2.0 General Description

During an SPI transfer, data is simultaneously transmitted (shifted out serially) and received (shifted in serially). All SPI transfers are started and controlled by a master SPI device. A serial clock line synchronizes shifting and sampling of the data on the two serial data lines. Slave select line(s) allow individual selection of slave SPI device(s); slave devices that are not selected do not interfere with SPI bus activities.

The SX SPI implementation uses 8-bit transfers, but not all peripherals use eight bits. Some peripherals use multiples of eight bits, and a few use odd word lengths. When a peripheral uses an odd number of bits, it is usually possible to send it as multiple of eight bits, and the peripheral will ignore the extra bits. In any case, the requirements of each peripheral in the system must be considered.

2.1 SPI Signal Lines

SPI uses a master-slave model and typically has three signal lines:

- Serial data input line (Sdi)
- Serial data output line (Sdo)
- Serial clock line (Sck)

Chip select signals from the master are used to address different slaves on the bus (Figure 2-1). SPI interface defines only the communication lines and the clock edge. Other parameters vary for different devices. Clock frequencies happen to be anywhere from a few Hz to a few MHz.

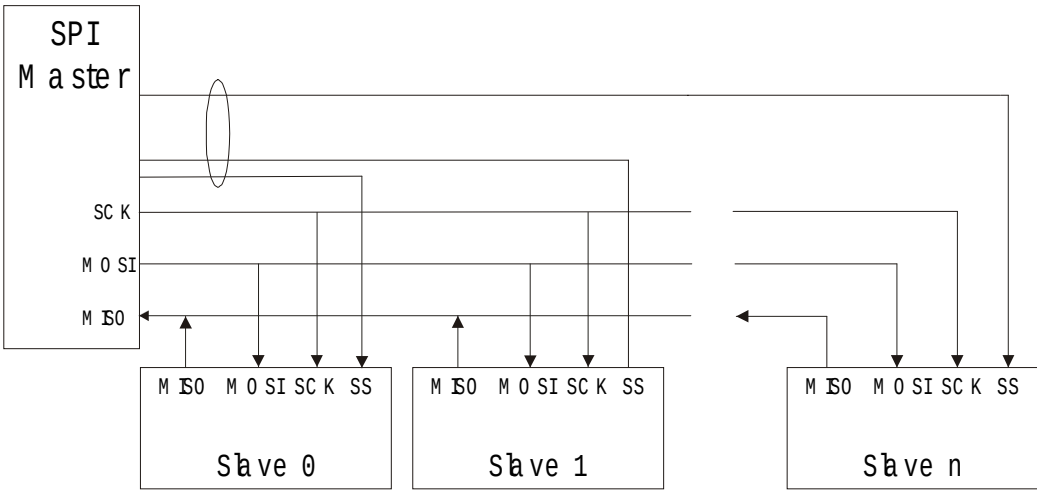

Figure 2-1. Master Slave System Configuration

– **MASTER OUT, SLAVE IN** - The MOSI line is used to transfer data from the master to the slave. The master transfers instructions as addresses and data to the slaves on this line. The name of the pin will depend on which device is referred to; on a master this line is often named serial data out (Sdo) whilst it on a slave is referred to as serial data in (Sdi). These pin-names indicate that the MOSI line only transfers data in one direction; from the master to a slave.

– **MASTER IN, SLAVE OUT** - The MISO line is used to transfer data from the slave to the master. On a slave this line is often named serial data out (Sdo) whilst it on a master referred to as serial data in (Sdi). The MISO line transfers data in the opposite direction to the MOSI line; from a slave device to the master.

– **SERIAL CLOCK** - The Sck line is used to synchronize the communication between a master and a slave. The Sck line is generated by the master device and thus is an input into all slave devices.

– **SLAVE SELECT** - The Slave Select (SS) lines are controlled by the master and used to select slave devices. The SS line must be active prior to data transactions and must stay active for the duration of the transaction. Each slave device requires its own SS input line from the master, and this line is often referred to as chip select (CS).

The timing diagram of signals on these lines can be seen in Figure 2-2.

Only one data line is shown in Figure 2-2 because the timing is the same for the MISO and the MOSI.

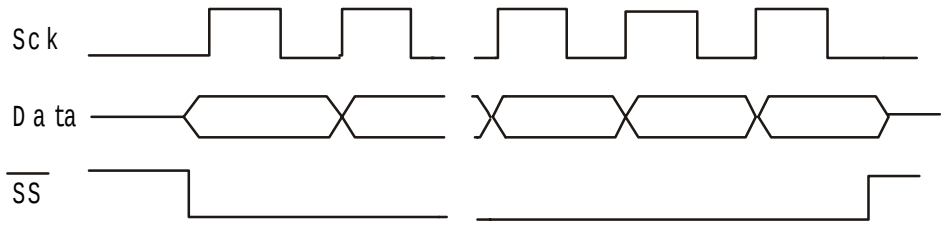

Figure 2-2. Data/Clock Timing Diagram for a Falling Edge Mode (CPOL=0, CPHA=1)

2.2 SPI Modes

There are four different modes, or ways to transmit the data on the SPI bus, determined by the value of CPHA and CPOL.

CPOL - The clock polarity (CPOL) determines the polarity of the clock (Sck) when the bus is idle. When CPOL is 0 the clock is idle low.

CPHA - The clock phase (CPHA) determines whether the data is clocked in or out on the first edge after idle. When CPHA = 0, data in are latched on the first clock edge, while CPHA = 1 means that data out are latched on the first edge after idle.

The four different modes are shown in Table 2-1. Please note that there are some similarities between the four modes; Mode 0 and 3 (00b and 11b), and mode 1 and 2 (01b and 10b) uses the same clock edge to latch data in and out.

Table 2-1. SPI Modes Determined by the CPHA and the CPOL Parameters.

CPHA	CPOL	Description
0	0	Data out on a falling edge and in on a rising edge. Clock is idle low.
0	1	Data out from SPI devices on rising edge and in on falling edge. Clock is idle high.
1	0	Data out from SPI devices on rising edge and in on falling edge. Clock is idle low.
1	1	Data out from SPI devices on falling edge and in on rising edge. Clock is idle high.

The in/out in Figure 2-3 indicates which clock edge data are clocked into and out of the device.

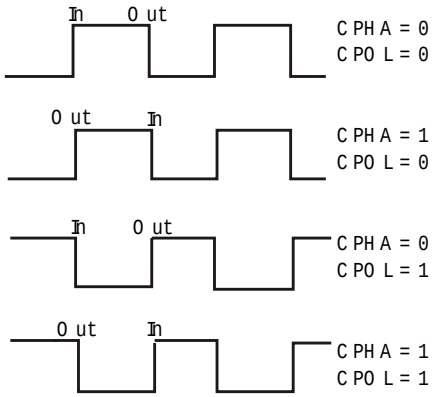

Figure 2-3. Clock Edges Where Data are Latched in/out for the Four Different Modes

2.3 SPI vs. Microwire/Plus

The primary difference between SPI and Microwire/Plus is that SPI supports data locking on both falling and rising edges of the clock signal while the latter is always operating on the rising edge. Many IC manufacturers produce components that are compatible with both SPI and Microwire/Plus.

3.0 Ubicom SPI Virtual Peripheral Implementation

The SX SPI Virtual Peripheral modules are implemented so that the data bits are shifted most significant bit first.

3.1 SPI Master Virtual Peripheral (`spim.src`)

3.1.1 Using the SPI Master Virtual Peripheral

The requirements to be fulfilled in order to use the SPI Master Virtual Peripheral are listed below.

1. Obtain the latest version of `spim.src`.
2. Modify the pin definitions and the port assignment to suit your application.
3. Select the chip you want to use (i.e. SX28AC).
4. Set the phase (CPHA) and polarity (CPOL) to choose SPI mode.
5. Determine the desired SPI bus speed; set the `intPeriod` and/or the repetition rate in the ISR jump table to get correct SPI bus speed.
6. Add your own features.
7. Download the code to the device selected, and the SPI Master is ready to be used.

The flowchart of the SPI Master Virtual Peripheral main program is shown in Figure 3-1.

3.1.2 Function Description

This section describes the interface to the SPI Master Virtual Peripheral and its demo application.

Only the `spimInit`, `spimGetByte` and the `spimSendByte` routine are necessary for running the SPI Master Virtual Peripheral. The rest of the routines are for demo purpose. However, the reason why these routines are included and described here is that they might be useful in an implementation.

3.1.2.1 spimInit

This is the SPI-Master initialization code. Initialization that is specific for the SPI-Master is inserted into this routine.

3.1.2.2 spimGetByte

This routine check if a received byte is available. If not, it waits and a byte returned in the W register when receiving is done.

3.1.2.3 spimSendByte

The `spimSendByte` routine initiates a byte transmission on the master. If there is already a transmission in progress, it will wait before initiating the new transfer.

3.1.2.4 spimWriteAddr

This routine writes the address specified in `spimAddress` to the SPI bus.

3.1.2.5 spimWriteBlock

Writes a block (`SPIM_BLOCK_SIZE`) from `spimRxBank` to the SPI bus. Writing starts from the address specified in `spimAddress`, and ends when the least significant bits of `spimAddress` (LSB) equal `SPIM_BLOCK_SIZE`.

3.1.2.6 spimCheckBlock

Reads a block from EEPROM and checks with `spimRxBank` for errors. Each time a byte error is found, `spimErrorByte` is called.

3.1.2.7 spimErrorByte

If an error is detected on read-back, this routine can be called to log the error as done in the `spimCheckBlock` routine. The address where the last error is detected is stored in `spimLastErrAdd`, the `spimErrCnt` is increased each time this routine is called. This routine could easily be expanded to log the error byte received and the byte value expected etc.

3.1.2.8 spimCSInactive

This routine (not a part of the SPI Virtual Peripheral) waits until read or write is done before chip-select (CS or SS) to EEPROM is set inactive. Delay is added to ensure a minimum CS disable time (`SPIM_CS_DISABLE`).

3.1.3 Flags and Variables

3.1.3.1 SPI Master Flags

`spimStartEn`	-	Tells ISR to start transmission/reception and tells main that TX/RX is in progress.
`spimRxAvailable`	-	Indicates that received byte is available.

3.1.3.2 Variables

`spimTxData`	-	Byte to be transmitted
`spimRxData`	-	Byte received
`spimBitCount`	-	Counts twice the bits sent/received. See `spimTransition`
`spimAddress`	-	16 bit address for reading/writing on the SPI bus
`spimLastErrAdd`	-	16 bit variable containing address where last error was detected during read-back
`spimErrCnt`	-	Counts errors detected during read-back.

3.1.3.3 Constants and Parameters

`SPIM_CPHA1`	-	Clock-phase: Uncomment this declaration if you want to set the CPHA=1 (1 = output data on first edge while 0 = sample data on first edge after idle)
`SPIM_CPOL1`	-	Clock-polarity: Uncomment this declaration if you want to set CPOL=1 (0 = `spimSck` idle low, 1 = `spimSck` idle high)
`SPIM_TRANSITION`	-	(Number of bits to transmit/receive) x 2. This one is set to 16 for byte operation.
`SPIM_CS_DISABLE`	-	Oscillator clock pulses that the CS pin must remain inactive (high)
`SPIM_WIP`	-	"Write in progress flag" position in slave status register
`SPIM_READ_CMD`	-	Read data from memory array command
`SPIM_WRITE_CMD`	-	Write data to memory array command
`SPIM_WREN_CMD`	-	This command enables writing to the slave by setting the write enable bit in the slave status register.
`SPIM_RDSR_CMD`	-	Read status register command
`SPIM_WRSR_CMD`	-	Write status register command
`SPIM_EPROM_SIZE`		Number of bytes in EEPROM to access.
`SPIM_BLOCK_SIZE`		Number of bytes in each block, max. $10 at current implementation. See `spimCheckBlock` and `spimWriteBlock`.

3.1.4 SPI Master Virtual Peripheral Description

The behavior of the SPI Master is controlled by the main program, which calls the subroutines implemented in the SPI Master. These access subroutines set flags and variables that are used in the ISR.

The next chapters will describe the SPI Master Virtual Peripheral and its functionality.

3.1.4.1 SPI Master Main Program

The SPI Master main program is a simple demo program that writes data to a slave EEPROM, reads the data back and checks for error. See flowchart in Figure 3-1 for a more detailed description of the main program.

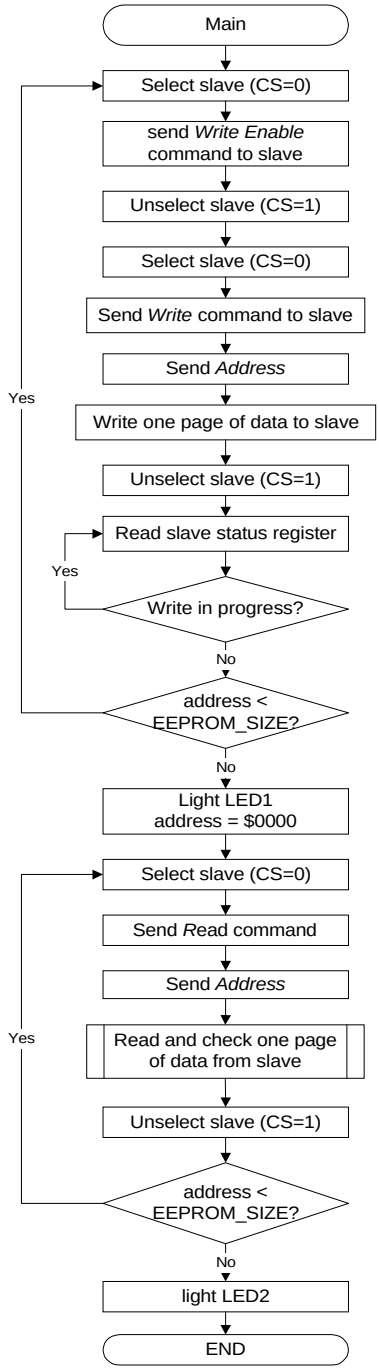

Figure 3-1. Flowchart of SPI Main Program

3.1.4.2 SPI Master ISR

Figure 3-2 shows the ISR routine for the SPI Master when the CPHA is set to "1". If CPHA is set to "0", the "Even *EdgeCount*?" is replaced with "Odd *EdgeCount*?" in the flowchart below.

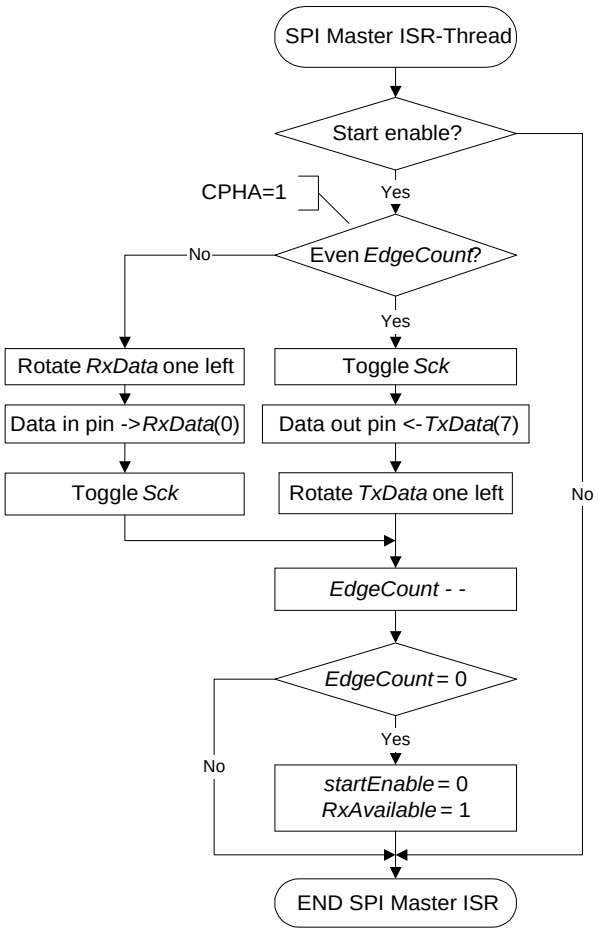

Figure 3-2. SPI Master ISR Routine Shown for CPHA=1

3.2 SPI Slave Virtual Peripheral (`spis.src`)

3.2.1 Using the SPI Slave Virtual Peripheral

The requirements to be fulfilled in order to use the SPI Master Virtual Peripheral are listed below.

1. Obtain the latest version of `spis.src`.
2. Modify the pin definitions and the port assignments to suit your application.
3. Select the correct chip you want to use (i.e. SX28AC).
4. Set the phase (CPHA) and polarity (CPOL) to choose SPI mode.
5. Determine the desired SPI bus speed; set the intPeriod and/or the repetition rate in the ISR jump table to get correct SPI ISR rate.
6. Add your own features.
7. Download the code to the device selected, and the SPI Slave is ready to be used.

3.2.2 Function Description

This section describes the key subroutines of the SPI Slave Virtual Peripheral and its demo application. Only the `spisInit` and the `spisGetByte` routines are necessary for running the SPI Slave Virtual Peripheral. The rest of the routines are mostly for demo purposes. However, some of the demo subroutines can also be found useful in an implementation.

3.2.2.1 spisInit

This is the SPI-Slave specific initialization code. Among other things done in this routine is reading a 16-byte demo-string from program memory (`_demoString`) to the `spisDataBuf` RAM-bank.

3.2.2.2 spisGetByte

This routine checks if a received byte is available. If not it waits and returns a byte in the W register when receiving is done.

3.2.2.3 spisWrSr

When a "write statusregister" command is sent to the slave this routine receives the new status register content (1 byte).

3.2.2.4 spisWrite

When a "write" command is sent to the slave this routine receives a 16-bit address and then the data to be written sequentially until address extends $xxxF or CS=1.

3.2.2.5 spisRead

When a "read" command is sent to the slave this routine receives the address to read from and then clocks the corresponding data out on the `spisSdoPin`.

3.2.2.6 spisWrDi

When a "write disable" command is sent to the slave this routine resets the `spisWriteEnable` flag in the slave status register.

3.2.2.7 spisRdSr

When a "read statusregister" command is sent to the slave this routine clocks the contents of the slave status register out on the `spisSdoPin`.

3.2.2.8 spisWrEn

When a "write enable" command is sent to the slave this routine sets the `spisWriteEnable` flag in the status register.

3.2.3 Flags and Variables

3.2.3.1 SPI Slave Flag

`spisRxAvailable`	-	Indicates that a received byte is available
`spisStatWPEn`	-	"Write protect enable" bit in `spisStatus` register
`spisStatWEL`	-	"Write enable" bit in `spisStatus` register
`spisStatWIP`	-	"Write in progress" bit in `spisStatus` register

3.2.3.2 Variables

`spisStatus`	-	Slave status register
`spisTxData`	-	Byte to be transmitted to the `spisTxBuf` register
`spisRxData`	-	Byte received
`spisBitCount`	-	(Number of bits left to transmit/receive) x 2
`spisPortPrev`	-	Previous state of SPI-Slave port
`spisTemp`	-	Temporary storage used by SPI-Slave ISR
`spisTxBuf`	-	Working register for buffering TX-byte
`spisAddr`	-	16 bit memory address
`SpisDataBuf`	-	16 bytes reserved for EEPROM Demo application

3.2.3.3 Constants and Parameters

`SPIS_CPHA1`	-	Clock-phase: Uncomment this declaration if you want to set the CPHA=1 (1 = outputs data on first edge, while 0 = sample data on first edge after idle)
`SPIS_CPOL1`	-	Clock-polarity: Uncomment this declaration if you want to set CPOL=1 (0 = `spisSck` idle low, 1 = `spisSck` idle high)
`SPIS_TRANSITION`	-	(Number of bits to transmit/receive) x 2
`SPIS_VALID_CMD`	-	Mask incoming command. Check that only the 3 LSb are used.
`SPIS_RANGE`	-	Address mask/range of slave. Set the used bits to zero. I.e. if only 8 bytes is used, set the 3 LSb to zero.

3.2.4 SPI Slave Virtual Peripheral Description

The behavior of the SPI Slave is controlled by the main program, which calls the subroutines implemented in the SPI Master. These access subroutines set flags and variables that are used in the ISR.

3.2.4.1 SPI Slave Main Program

The SPI Slave main program is a simple demo program that makes the SX act as a SPI EEPROM. This SPI slave implementation is an emulation of an EEPROM slave with 16 bytes of memory. The flow chart in Figure 3-3 shows the functionality of the SPI Slave.

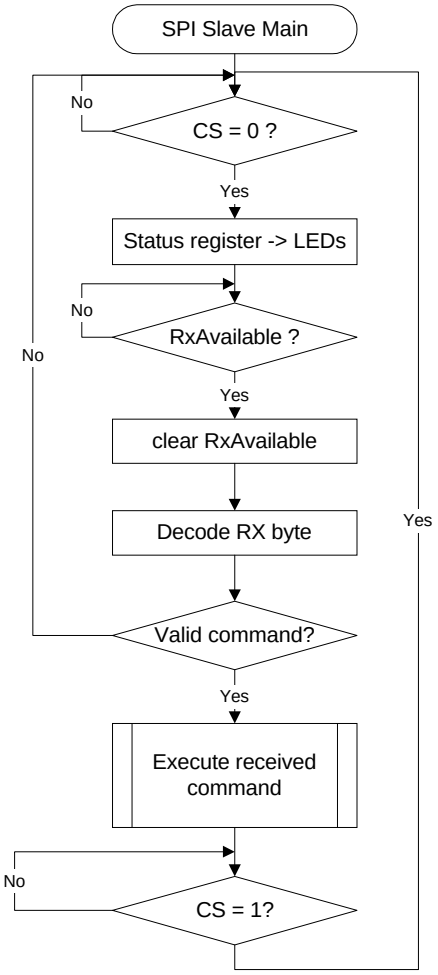

Figure 3-3. SPI Slave Main Flowchart

The subroutines listed in the instruction decoder above are described in section 4.2.2. The commands sent from the master must therefore be a valid number from 1 to 7 and should correspond with the table.

The implementation of the "Execute received command" in Figure 3-3 is shown in the instruction decoder below:

	Command name	Command value
add	pc,w	N/A
jmp	spisIdle	;0
Jmp	spisWrSr	;1
jmp	spisWrite	;2
jmp	spisRead	;3
jmp	spisWrDi	;4
jmp	spisRdSr	;5
jmp	spisWrEn	;6

3.2.4.2 SPI Slave ISR

Figure 3-4 shows the ISR routine for the SPI Slave when the CPHA is set to "1". If CPHA is set to "0", the "Even EdgeCount? " is replaced with "Odd EdgeCount?" in the flowchart below.

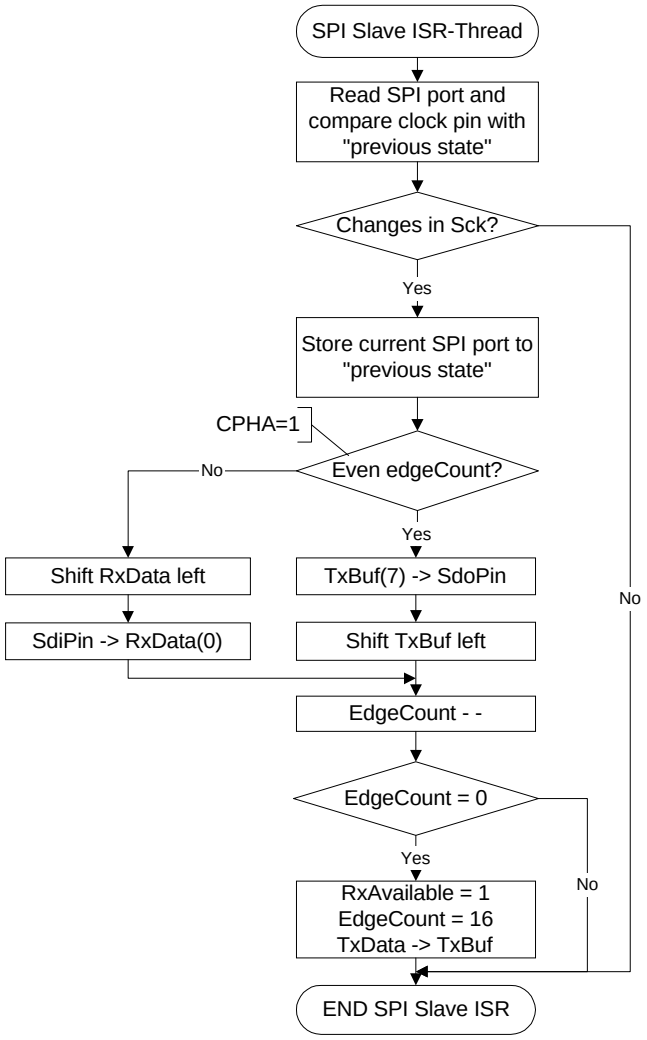

Figure 3-4. SPI Slave ISR Flowchart

3.2.5 Important Considerations

The current implementation does not support tri-state of the MISO line. Thus, the SX slave cannot be used in a multiple slave setup without some minor modifications. This can be solved by setting the MISO pin (`spisSdoPin`) as input when chip select (CS) is inactive.

4.0 Specifications / Features / Characterization

4.1 SPI Master Virtual Peripheral Requirements

Operational mode:	Random read/write of byte or block.
Multithreaded:	Yes
Thread Rate:	1/4
Clock speed:	50 MHz
MIPS usage:	8 MIPS
Max cycles in each thread:	40
Bus oversampling rate:	2
Bus speed at current rate:	99,2kHz
Program Memory usage:	375*) words
RAM usage:	1 byte in bank 1, 10 bytes in bank 2, 2 bit global temp variable.
Pin usage:	4 pins (Sck, Sdo, Sdi and CS)
RTCC setting:	Timer interrupt running every 1.26us à ISR rate = 793.6kHz
Ubicom mnemonics:	Yes

*) The program memory usage is an approximation of the total program memory usage when the demo is included. Approximately 120 words is general code (template) not specific for the SPI Virtual Peripheral modules. Additional 170 words are demo routines and main. This means that approximately 85 words are the SPI master core.

4.2 SPI Slave Virtual Peripheral Requirements

Operational mode:	Random read/write of byte or block.
Multithreaded:	Yes
Thread Rate:	1/4
Clock speed:	50 MHz
MIPS usage:	12 MIPS
Max cycles in each thread:	46
Bus sampling rate:	250kHz
Bus speed at current rate:	max 100kHz
Program Memory usage:	325$) words
RAM usage:	10 bytes in bank 1, 1 bit global temp variable
Pin usage	4 pins (Sck, Sdo, Sdi and CS)
RTCC setting:	Timer interrupt running every 1us à ISR rate = 1MHz.
Ubicom mnemonics:	Yes

$) The program memory usage is an approximation of the total program memory usage when the demo is included. Approximately 120 words is general code (template) not specific for the SPI Virtual Peripheral. Additional 115 words are demo routines and main. This means that approximately 90 words are the SPI slave core.

5.0 Demo Application

The demo applications have been written to give the user the ability to do a functional test of the SPI Virtual Peripheral. The master demo application is written to interface with a SPI EEPROM. The SX slave demo is written to interface with the SX master demo and thus it emulates the most important features of the SPI EEPROM. Because of memory considerations the SPI slave demo only emulates a 16-byte memory.

The master writes a string to the slave device. After the write process it reads back the string stored in the slave and does a byte-by-byte check, to ensure that the write process was successful. If any errors are found the address of the last error and the total number of errors are reported on screen when running the SX Key in debug mode.

By uncommenting a line in the master source code a faulty byte can be written to a predefined memory location. The "watch window" in the SX Key debugger will report the error found during read back.

The default settings on the master have been preset for interfacing with the SX SPI slave demo. To interface with a native SPI EEPROM some modifications have to be made in the constant declarations in the SPI master.

Both the SX slave and the SX master demo require uncommenting the DEMO line in the sourcecode. To run with the SX Key debugger the `sx_key` line has to be uncommented as well.

To follow this demo step by step you need an "SX28AC-52 demoboard revision 2" and an "SX Key revision E/F " debugger. You can run this demo on any SX28AC/52BD system but then your setup may deviate from this description.

5.1 Running the demo with the SX slave.

Connect the master and slave as shown in Figure 5-1.

Ensure that both the SX28AC and the SX52BD have power (jumper JP3 SX28AC and SX52BD to VDD).

Preparing the slave on SX52BD

1. Load the `spis.src` into the SX 48/52 Key editor, and connect the debugger to the SX52BD header on the demo board.
2. Uncomment the DEMO line in the source file.
3. Select assembler by uncommenting the `sx_key` line in the source file.
4. Program the target (Ctrl + P). Oscillator jumpers have to be closed to run the SX without the debugger.
5. Remove the debugger.

Preparing the master on SX28AC

1. Load the `spim.src` in the SX 18/28 Key editor, and connect the debugger to the SX28AC header on the demo board.
2. Uncomment the DEMO line in the source file.
3. Select assembler by uncommenting the `sx_key` line in the source file.
4. Select target SX by uncommenting SX28AC and commenting the SX52BD line in the Source file.
5. Reset the slave with the reset button on the demoboard.
6. Program the SX in debug mode (Ctrl + D) and press run in the "SX Key control panel". Oscillator jumpers have to be open to run the SX trough the debugger.

The SPI master will write a new string to the SX SPI slave and then verify the contents.

The circuit used in the demo is shown in Figure 5-1.

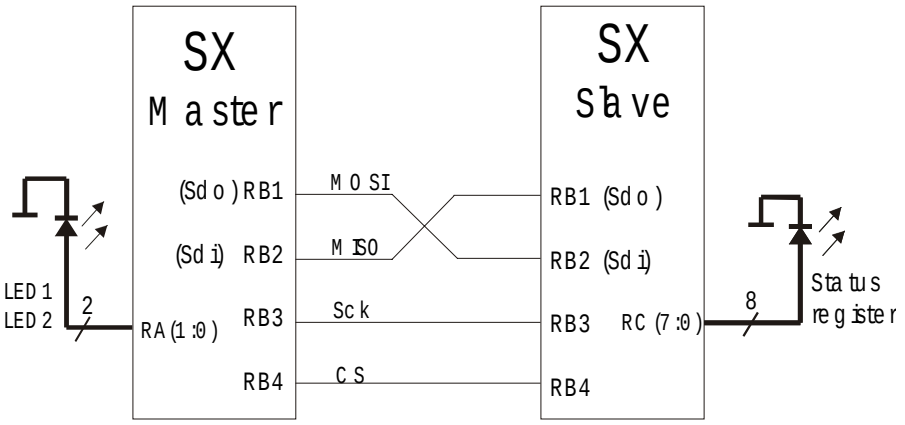

Figure 5-1. SPI Master/Slave Interface

5.2 Running the Demo with the EEPROM.

To run the demo on the SX28AC with a 25LC160 SPI EEPROM as a slave device follow the steps below.

1. Load the `spim.src` in the SX 18/28 Key editor, and connect the debugger to the SX28AC header on the demo board.
2. To run the demo, uncomment the `DEMO` line in the source file.
3. Select assembler by uncommenting the `sx_key` line in the source file.
4. Select target SX by uncommenting SX28AC and commenting the SX52BD line in the Source file.
5. In the code the following constants have to be changed to interface with the EEPROM: `SPIM_EPROM_SIZE = $800` and `SPIM_BLOCK_SIZE = $10`.
6. Program the SX in debug mode (Ctrl + D) and press run in the "SX Key control panel". Oscillator jumpers have to be open to run the SX trough the debugger.

The SPI master will write a string to each page of the EEPROM and then verify the contents.

Other 25xx EEPROM's should work as well but the settings on the master have to be changed to meet the EEPROM specifications.

The circuit used in the demo is shown in Figure 5-2.

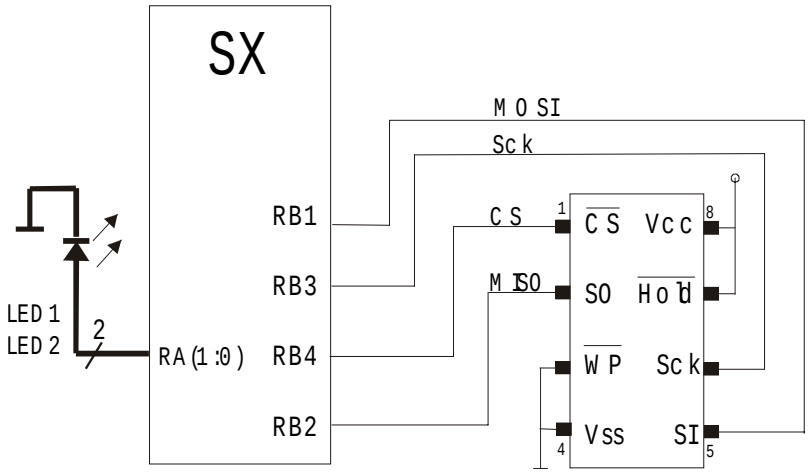

Figure 5-2. SX SPI Master Interfacing to SPI EEPROM Slave

6.0 Test Description

The SPI Virtual Peripheral are tested and verified. This section describes the environment used during the verification and how the test was performed.

6.1 Environment

Oscilloscope	: Tektronix TDS 3014 (100MHz)
Demo board	: Ubicom SX28AC-52 Revision 2.0
Debugger	: Parallax SX key Revision E (2 units)
SPI EEPROM	: 25LC160
Assemblers	: Ubicom SASM version 1.45.5
	: Parallax SX 18/28 Key version 1.09
	: Parallax SX 48/52 Key version 1.19

The binary outputs from SASM were compared and found identical with the Parallax assembler binary outputs. Thus, no special tests were done on binaries generated by SASM.

6.2 Functional Test Description

6.2.1 Master on SX28AC / 52BD Interfacing SPI EEPROM

The schematic of the circuit used in this test is shown in Figure 5-2.

Interfacing SX28AC with SPI EEPROM 25LC160.

1. Load the `spim.src` in SX 18/28 Key editor, and connect the debugger to the SX28AC header on the demo board.
2. Select target SX by uncommenting SX28AC and commenting the SX52BD line in the source file.
3. Ensure that the SX28AC are powered (jumper JP3, VDD - SX28AC).
4. Set SPI mode as listed in Table 6-1.
5. In the source file the following constants have to be changed to interface with the EEPROM: `SPIM_EPROM_SIZE = $800` and `SPIM_BLOCK_SIZE = $10.`
6. Uncommenting the `DEMO` line in the source file.
7. Select assembler by uncommenting the `sx_key` line in the source file.
8. Program the SX in debug mode (Ctrl + D) and press run in the "SX Key control panel". Oscillator jumpers have to be open to run the SX trough the debugger.

The SPI master will write a string to each page of the EEPROM and then verify the contents.

Uncomment the call to the `spimByteError` routine to verify that the `spimCheckBlock` functions properly.

Interfacing SX52BD with SPI EEPROM 25LC160.

1. Load the `spim.src` in SX 48/52 Key editor, and connect the debugger to the SX52BD header on the demo board.
2. Ensure that the SX52BD are powered (jumper JP3, VDD - SX52BD).
3. Repeat steps 4-7 "Interfacing SX28AC with SPI EEPROM 25LC160" above.

The SPI master will write a string to each page of the EEPROM and then verify the contents. Uncomment the call to the `spimByteError` routine to verify that the `spimCheckBlock` functions properly.

6.2.2 Master Virtual Peripheral Interfacing Slave Virtual Peripheral

To verify correct operation of the SPI master and slave, two setups where used.

Before you start you should ensure that both the SX28AC and the SX52BD are powered (jumper JP3 SX28AC and SX52BD to VDD). The schematic of the circuit used is shown in Figure 5-1.

SPI Master on SX52BD and SPI Slave on SX28AC

The setup described below has been repeated for all four SPI modes.

Preparing the slave on SX28AC.

1. Load the `spis.src` in the SX 18/28 Key editor, and connect the debugger to the SX28AC header on the demo board.
2. Select target SX by uncommenting SX28AC and commenting the SX52BD line in the Source file.
3. Set SPI mode as listed in Table 7-2.
4. Uncomment the `DEMO` line in the source file.
5. Select assembler by uncommenting the `sx_key` line in the source file.
6. Program the SX in debug mode (Ctrl + D). Oscillator jumpers have to be open to run the SX through the debugger.

Preparing the master on SX52BD.

1. Load the `spim.src` in the SX 48/52 Key editor, and connect the debugger to the SX52BD header on the demo board.
2. Uncomment the `DEMO` line in the source file.

Set SPI mode as listed in Table 7-2.

1. Select assembler by uncommenting the `sx_key` line in the source file.
2. Program the SX in debug mode (Ctrl + D). Oscillator jumpers have to be open to run the SX trough the debugger.

First start the slave and then start the master by pressing run in the debugger window (note that there is one debugger window for each controller).

The SPI master will write a string to the SX SPI slave and verify the contents.

SPI Master on SX28AC and SPI Slave on SX52BD

Only the default mode (00) was tested for this setup.

Preparing the slave SX52BD.

1. Load the `spis.src` in the SX 48/52 Key editor, and connect the debugger to the SX52BD header on the demo board.
2. Uncomment the `DEMO` line in the source file.
3. Select assembler by uncommenting the `sx_key` line in the source file.
4. Program the SX in debug mode (Ctrl + D). Oscillator jumpers have to be open to run the SX through the debugger.

Preparing the master on SX28AC.

1. Load the `spim.src` in the SX 18/28 Key editor, and connect the debugger to the SX28AC header on the demo board.
2. Select target SX by uncommenting SX28AC and commenting the SX52BD line in the Source file.
3. Uncomment the `DEMO` line in the source file.
4. Select assembler by uncommenting the `sx_key` line in the source file.
5. Program the SX in debug mode (Ctrl + D). Oscillator jumpers have to be open to run the SX through the debugger.

First start the slave and then start the master by pressing run in the debugger window (please note that there is one debugger window for each controller).

The SPI master will write a string to the SX SPI slave and verify the contents.

6.3 Test Results

The criteria for a successful test are that data is both written and read back from the EEPROM without errors.

6.3.1 Master on SX28AC / 52BD Interfacing SPI EEPROM

Table 6-1, test results for SPI master interfacing SPI EEPROM

Table 6-1. Test Results for SPI Master Interfacing SPI EEPROM

Device	CPHA	CPOL	Result
SX28AC	0	0	OK
	1	1	OK
SX52BD	0	0	OK
	1	1	OK

To ensure that the `spimCheckBlock` routine worked, the `spimByteError` was uncommented. In the watch window the following values were obtained:

`spimErrCnt` = 1

`spimLastErrAdd` = $0007

This indicates that one incorrect byte was read back from address $0007 and verifies correct operation of the `spimCheckBlock` routine.

Figure 6-1 shows the start sequence for writing data to the EEPROM. The first byte written is the SPIM_WREN_CMD ($06) to enable writing to the device. CS is toggled and a SPIM_WRITE_CMD ($02) followed by the 16-bit address ($0000) is sent on the MOSI line. No data is present on the MISO line at this time.

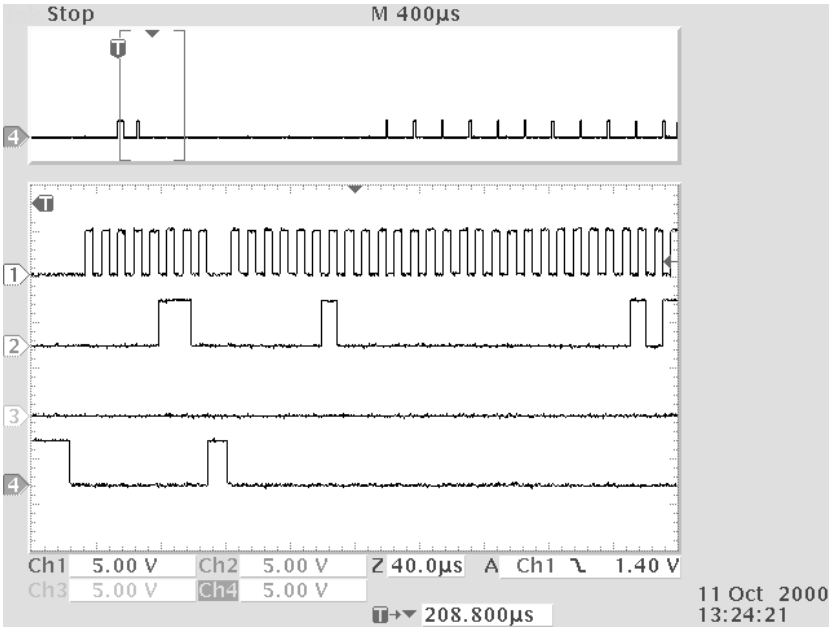

Figure 6-1. Start Sequence for Mode 00 (Signals: Sck, MOSI, MISO and CS)

Figure 6-2 is similar to Figure 6-1 except for the clock is idle high (CPOL=1 and CPHA=1). This shows the similarities between the 00b and the 11b mode mentioned in section 3.2.

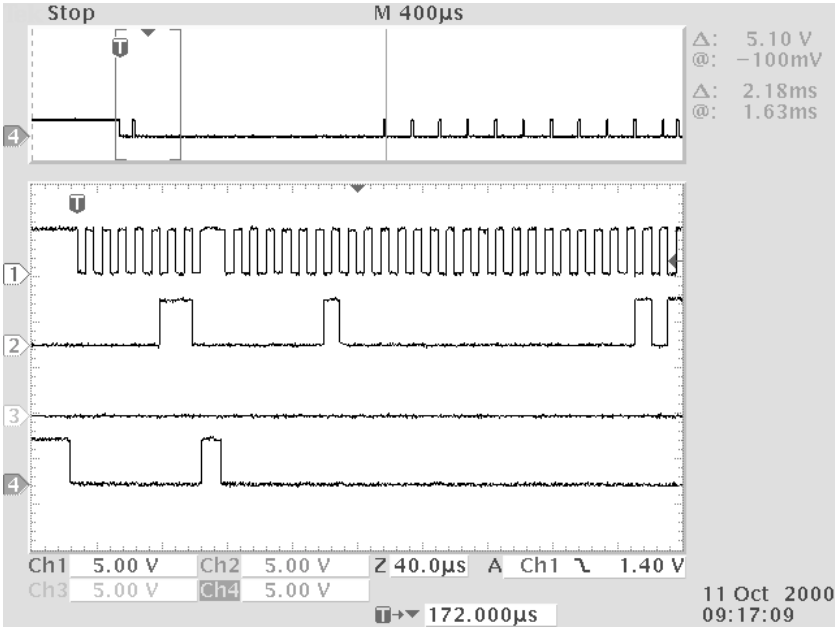

Figure 6-2. Start Sequence for mode 11 (Signals: Sck, MOSI, MISO and CS)

Figure 6-3 shows the end of a block write sequence. After the last byte in the block is written, the master toggles the CS line and starts polling the status register in the slave. The polling of the status register is shown between the cursors in Figure 6-3, and the data returned on the MISO line is $03. This means that the slave is busy with "write in progress" (the WIP bit is bit 0 and WEL is bit 1 in the status register for the 25LC160).

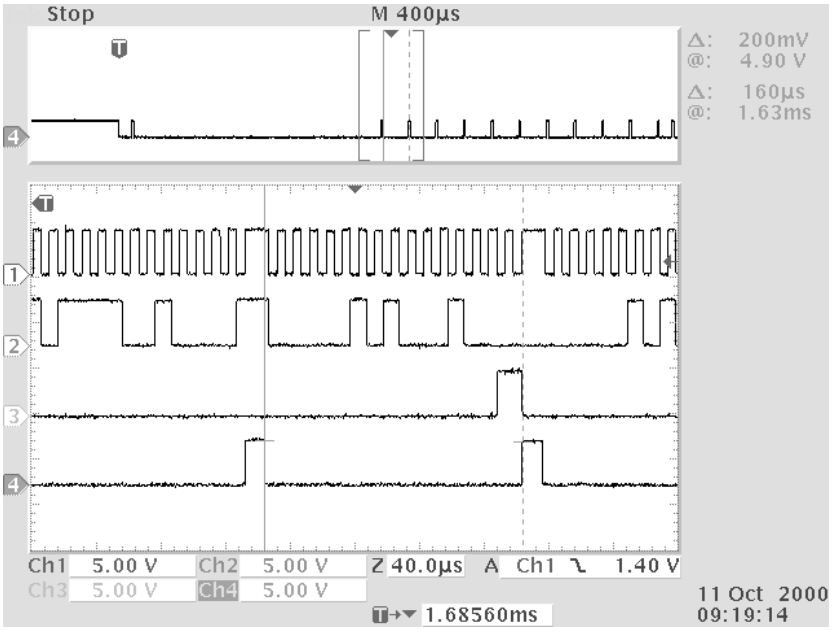

Figure 6-3. End of Write Sequence and Polling Status Register (Signals: Sck, MOSI, MISO and CS)

6.3.1.1 Master Virtual Peripheral Interfacing to Slave Virtual Peripheral

Table 6-2. SPI Master Interface to Slave

Master SX52BD		Slave SX28AC		Results
CPHA	**CPOL**	**CPHA**	**CPOL**	
0	0	0	0	OK
0	1	0	1	OK
1	0	1	0	OK
1	1	1	1	OK
The results below are for documentation only (nice to know).				
0	0	1	1	OK
0	1	1	0	OK
1	0	0	1	Failiure*)
1	1	0	0	Failiure*)

*) The reason why these two modes fail is because the slave is level and not edge triggered.

Table 6-3. SPI Master on SX28AC and Slave on SX52BD

Master SX28AC		Slave SX52BD		Results
CPHA	**CPOL**	**CPHA**	**CPOL**	
0	0	0	0	OK
0	1	0	1	Not tested
1	0	1	0	Not tested
1	1	1	1	Not tested

7.0 References

References and further readings are listed below:

Ref	Title	Author(s)
1	Virtual Peripheral guide v. 1.04 (Template for VP development	Ubicom
2	Schematics for the SX28AC/SX52BD Demoboard.	Ubicom
3	25LC160 SPI EEPROM datasheet	Microchip http://www.microchip.com
4	Motorola M68HC11 Reference Manual (Section 8)	Motorola http://www.motorola.com

Lit #: SXL-AN20-04

Sales and Tech Support Contact Information

For the latest contact and support information on SX devices, please visit the Ubicom website at www.ubicom.com. The site contains technical literature, local sales contacts, tech support and many other features.

1330 Charleston Road
Mountain View, CA 94043
Contact: sales@ubicom.com
http://www.ubicom.com
Tel.: (650) 210-1500
Fax: (650) 210-8715