

General Description

The Nios[®] serial peripheral interface (SPI) module is an Altera[®] SOPC Builder library component included in the Nios development kit. The SPI module is a simple, industry standard communications interface commonly used in embedded systems. Many semiconductor manufacturers sell a variety of sensor, conversion, and control devices that use an SPI. The SOPC Builder SPI library component has available system choices to define device logic and interface signals on the Nios development board. The SPI's Verilog HDL or VHDL source code is available for development and includes the necessary software subroutines for easy system integration.

The Nios SPI peripheral is 3-wired and can be used as either a master or slave device. It allows software to communicate with one or more external devices over an SPI bus. Software controls and communicates with the SPI peripheral through five memory-mapped, 16-bit registers accessed by a standard Nios peripheral bus connection. When configured as a master device, the SPI can interface with up to 16 independent SPI slave devices.

Functional Description

The SPI peripheral extends the conventional protocol by allowing more than one external SPI peripheral to share the same `miso`, `mosi`, and `sclk` pins. Each SPI peripheral instantiated within a Nios system has at least four interface signals (that is, physical I/O pins on the Altera programmable logic device): `miso`, `mosi`, `sclk`, and `ss_n`. Master devices interfacing with multiple SPI slave devices have one `ss_n` signal for each slave device. The function of these signals depends on whether the SPI peripheral is operating as a master or slave device.

Table 1 lists pin names, numbers, and types for both master and slave configurations.

Pin Name	Polarity	Master Device		Slave Device	
		Number of Pins	Pin Type	Number of Pins	Pin Type
miso	High	1	Input	1	Output
mosi	High	1	Output	1	Input
sclk	–	1	Output	1	Input
ss_n	Low	1–16	Output	1	Input

The `miso` (master in, slave out) signal carries synchronous data from the slave to the master device. The `mosi` (master out, slave in) signal carries synchronous data from the master to the slave device. The `sclk` signal is driven by the master, synchronizing all data transfers. Each SPI slave device has one `ss_n` signal. The `ss_n` signal is an active-low slave input/master output pin. Slave devices do not respond to transactions unless their `ss_n` input signal is driven low.



Typical SPI peripherals set their `miso` output pins—when not selected—to high impedance. The Altera-provided SPI slave peripheral drives an undefined value on its `miso` output pin when not selected. If an SPI slave device is connected to an external (off-chip) SPI master device, the select input can be used to control a tri-state pin outside the Nios system module. This option is necessary when multiple slave devices share the same SPI bus.

The SPI peripheral can be used in conjunction with the DMA peripheral to allow streaming data transfers between the SPI and memory.



See the [Nios DMA Data Sheet](#) for details on the DMA library component.

The SPI peripheral consists of these user-visible components:

- A memory-mapped register space (software interface)
- SPI bus interface pins (hardware interface to other SPI devices)

The register control bits determine when data can be read from and written to the memory-mapped registers. All data is transmitted via the SPI bus interface pins, `miso`, `mosi`, `sclk`, and `ss_n`.

Master Mode Operation

The SPI peripheral operates as a master device if it is configured as such in the PTF at design time. In general, the SPI protocol does not support multi-master systems. The master initiates all data transactions, and every transaction is both a receive and transmit operation. The master device transmits a new data bit on the `mosi` pin and the slave device drives a new data bit on the `mis0` pin on each active clock edge.

The SPI peripheral transmitter logic consists of an n -bit (where n is a value from 1 to 16) `txdata` transmit holding register and an n -bit transmit shift register (the number of data bits is determined by the `databits` PTF assignment). Software starts an SPI transaction by writing a transmit-data value to the `txdata` register. The transmit shift register directly feeds the `mosi` data pin. Data is shifted out to `mosi` either least significant bit (LSB) first or most significant bit (MSB) first, depending on the `lsbfirst` PTF assignment. The transmit shift register is loaded automatically from the `txdata` register whenever a serial transmit shift operation is not currently in process.



The transmit shift register and the `txdata` register provide double buffering during data transmission; that is, software can write a new value into the `txdata` register while the previously written character is being shifted out of the transmit shift register.

The SPI peripheral receive logic consists of an n -bit (where n is a value from 1–16) `rxdata` receive holding register and an n -bit receive shift register (the number of data bits is determined by the `databits` PTF assignment). The `rxdata` receive holding register is read directly by software. The receive shift register is fed directly by the `mis0` data pin. The `rxdata` holding register is loaded from the receive shift register automatically every time data is fully received.



The receive shift register and the `rxdata` register provide double buffering during data receiving; that is, the `rxdata` register can hold a previously received data byte while the subsequent data byte is being shifted into the receive shift register.

Software can monitor the status of a master write operation by reading the `trdy`, `tmt`, and `toe` bits in the `status` register. Similarly, software can monitor the status of a slave read operation by reading the `roe` and `rrdy` bits in the `status` register.

Slave Mode Operation

The SPI peripheral slave device behaves similarly to the master device, except it cannot initiate a data transaction.

Before a transaction begins, the slave device polls its `ss_n` pin waiting for the pin to be driven low, which indicates the device is being addressed by an SPI master device. The SPI slave device immediately begins sending the transmit shift register contents to the `mis0` pin. The slave device also reads the receive shift register simultaneously, polling for data on the `mosi` pin. Thus, a read and write transaction are carried out simultaneously.

SPI Registers

Software controls the SPI peripheral by reading from and writing to memory-mapped registers inside the Altera device.

Table 2 shows a register map for SPI master and slave devices with an *n*-bit transmit/receive shift register operating as master and slave devices.

A2..A0	Register Name	R/W	Description/Register Bits										
			15...	10	9	8	7	6	5	4	3	2	1
0	rxdata	RO	rxdata(n-1..0)										
1	txdata	WO	txdata(n-1..0)										
2	status ⁽¹⁾	RW			e	rrdy	trdy	tmt	toe	roe			
3	control	RW		ss0 ⁽²⁾		ie	irrdy	itrdy		itoe	iroe		
4	reserved	–	Present only on master										
5	slaveselect	RW	Slave select mask—present only on master										

Notes

- (1) A write operation to the `status` register clears the `roe`, `toe`, and `e` bits.
- (2) Only for SPI masters.

rxdata Register

Software reads received data from the `rxdata` register. When new data is fully received via the `mis0` input, the `status` register's `rrdy` bit is set to 1 and the data is transferred into the `rxdata` register. When software reads a value from the `rxdata`, the `status` register's `rrdy` bit is cleared. If data is transferred into the `rxdata` register when the `rrdy` bit is set (that is, when software has not retrieved the previous data), a receive-overflow error occurs and the `status` register's `roe` bit is set to 1. New data is always transferred into the `rxdata` register, whether or not software retrieved the previous data.



Writing data to the `txdata` register has no effect.

txdata Register

Software writes data to be transmitted directly into the `txdata` register. Data should not be written to the `txdata` register until the transmitter is ready for new data, as indicated by the `status` register's `trdy` bit. If data is written to the `txdata` register when `trdy` is 0, a `toe` error is generated and the contents of the `txdata` register are undefined. The `trdy` bit is set to 0 whenever software writes data into the `txdata` register. The `trdy` bit is set to 1 when data is transferred from the `txdata` register into the transmitter shift register, indicating an empty `txdata` holding register.

For example, assume that the SPI peripheral is idle and software writes data into the `txdata` holding register. During the idle phase, the `trdy` bit is set to 0, but after the data is transferred into the transmitter shift register, `trdy` is immediately set to 1. Software begins writing a second data byte into the `txdata` register, and again the `trdy` bit is set to 0. However, this time the original data byte is still in the process of being transmitted over the serial `mosi` pin. Accordingly, the `trdy` bit remains at 0 until the transaction cycle ends. When the cycle ends, the second data byte is transferred into the transmitter shift register and the `trdy` bit is again set to 1.

status Register

The `status` register consists of bits that indicate particular conditions inside the SPI peripheral. The `status` register can be read at any time by software, and doing so does not change the value of any of the bits. Each bit is associated with a corresponding interrupt-enable bit in the `control` register, as discussed in “[control Register](#)” on page 7. If a `status` bit's corresponding interrupt-enable bit equals 1 at the same time the `status` bit equals 1, an interrupt request is sent to software.

The status register bits are shown in [Table 3](#):

Table 3. status Register Bits		
Bit Number	Bit Name	Description
3	roe	Receive overrun error.
4	toe	Transmitter overrun error.
5	tmt	Transmitter shift register empty.
6	trdy	Transmitter ready.
7	rrdy	Receive ready.
8	e	Error

roe Bit

The `roe` bit is set to 1 if data is received while the `rxdata` register is full (that is, while the `rrdy` bit is set to 1). If the receive overrun error (ROE) occurs, the new data overwrites the old. The `roe` bit is set to 0 when software performs a write operation to the `status` register.

toe Bit

The `toe` bit is set to 1 when data is written to `txdata` while the register is still full (that is, while the `trdy` bit is set to 0). If the transmitter overrun error (TOE) occurs, the new data is ignored. The `toe` bit is set to 0 when software performs a write operation to the `status` register.

tmt Bit

The `tmt` bit is set to 0 while a transaction is in progress and set to 1 when the shift register is empty.

trdy Bit

The `trdy` bit is set to 1 when the `txdata` register is empty.

rrdy Bit

The `rrdy` bit is set to 1 when the `rxdata` register is full.

e Bit

The `e` bit is set to 1 when either `toe` or `roe` is set. This is a convenience for the programmer. The `e` bit is set to 0 when software performs a write operation to the `status` register.

control Register

The `control` register consists of data bits that control the SPI peripheral's internal operation. Each bit in the `control` register enables an interrupt for the corresponding bit in the `status` register. The value in the `control` register can be read at any time by software.

The `control` register bits are shown in [Table 4](#):

Bit Number	Bit Name	Description
3	<code>iroe</code>	Enable interrupt for a receive overrun error.
4	<code>itoe</code>	Enable interrupt for a transmitter overrun error.
6	<code>itrdy</code>	Enable interrupt for a transmitter ready.
7	<code>irrdy</code>	Enable interrupt for a receiver ready.
8	<code>ie</code>	Enable interrupt for an error.
10	<code>sso</code>	Override <code>SS_n</code> output.

The control bits `iroe`, `itoe`, `itrdy`, `irrdy`, and `ie` allow software to determine which, if any, of the SPI peripheral's internal conditions result in an interrupt request. Each bit in the status register has a corresponding interrupt-enable bit at the same bit position in the control register (see [SPI Register Map on page 4](#)). For example, the status register's bit 7 is `rrdy`, and the control register's corresponding bit 7 is `irrdy` (interrupt-enable, receiver ready). For each status register bit, an interrupt request to software is generated if both the status bit and its corresponding interrupt-enable bit are set to 1.

Bit 10 of the control register, `sso`, does not enable an interrupt source. Instead, it can be used to force the `SS_n` output or outputs active at any time. `sso` can be used to transmit or receive data of arbitrary size on the SPI link. See [Figure 1 on page 8](#) for an example that demonstrates sending 24-bit data over an 8-bit SPI master.



The received data is read, but not saved, which is not strictly necessary, but avoids receive overrun errors.

Figure 1: Code Example for sso Control Register

```

// Force SS_n active:
na_spi_0-> np_spicontrol |= np_spicontrol_sso_mask;

for (i = 0; i < 3; ++i)
{
    // Transmit a byte:
    while (!(na_spi_0->np_spistatus & np_spistatus_trdy_mask));
    na_spi_0->np_spitxdata = data[i];

    // Read and throw away the received data:
    while (!(na_spi_0->np_spistatus & np_spistatus_rrdy_mask));
    na_spi_0->np_spirxdata;
}

// Wait until the last byte is transmitted:
while (!(na_spi_0->np_spistatus & np_spistatus_tmt_mask));

// Release SS_n:
na_spi_0-> np_spicontrol &=~np_spicontrol_sso_mask;

```

slaveselct Register

The `slaveselct` register sets the select bit mask for slave devices driven by the SPI bus master device. The `slaveselct` register is only present when the SPI peripheral is configured as a master device.

The `slaveselct` register can be used to address up to 16 slave devices by setting the corresponding bit. For example, to select slave device 0, bit 0 in the `slaveselct` register must be set to 1.

The SPI master device can also interface to multiple slave devices simultaneously by setting the `slaveselct` bits for each desired device. For example, to select slave devices 1, 5, and 6, the `slaveselct` register bits 1, 5, and 6 must be set to 1.



Exercise caution when selecting multiple slaves simultaneously. Contention may result on the `mis0` pin.

A reset sets bit 0 and clears every other bit of the `slaveselct` register. Thus, after a device reset, slave device 0 is automatically selected.

Software Data Structure

```

typedef volatile struct
{
    int np_spirxdata;           // Read-only, 1-16 bit
    int np_spitxdata;          // Write-only, 1-16 bit
    int np_spistatus;          // Read-only, 9-bit
    int np_spicontrol;         // Read/Write, 9-bit
    int np_spirereserved;      // reserved
    int np_spislaveselct;      // Read/Write, 1-16 bit, master only
} np_spi;

```

Software Subroutines

Table 5 lists the SPI software subroutines available in the Nios library (`lib` directory in the custom SDK) when one or more SPI peripherals are present in the Nios system. These functions are declared in the include file `nios.h`.

<i>Table 5. SPI Software Subroutines</i>	
Subroutine	Description
<code>nr_spi_rxchar</code>	Reads a character from the SPI peripheral whose address is passed as an argument.
<code>nr_spi_txchar</code>	Sends a single character to the SPI peripheral whose address is passed as an argument.

nr_spi_rxchar

This subroutine reads a character from the SPI peripheral whose address is passed as `pSPI`.

Syntax

```
int nr_spi_rxchar(np_spi *pSPI);
```

Parameter

The `pSPI` parameter is a pointer to the SPI peripheral.

nr_spi_txchar

This subroutine sends a single character, `i`, to the SPI peripheral whose address is passed as `pSPI`.

Syntax

```
int nr_spi_txchar(int i, np_spi *pSPI);
```

Parameters

Parameter	Description
<code>i</code>	Character to be sent
<code>pSPI</code>	Pointer to the SPI peripheral

PTF Assignments

Table 6 lists the SPI's PTF parameters. Detailed descriptions follow the table.

Parameter	Section ⁽¹⁾	Type	Allowed Values	Default	Units
clock_freq	S/WSA	Integer	≥ 1	33333000	Hz
ismaster	M/WSA	Boolean	1, 0	1	–
databits	M/WSA	Integer	1 .. 16	8	bits
targetclock	M/WSA	Integer	> 0	128	kHz
numslaves	M/WSA	Integer	1 .. 16	1	bits
clockpolarity	M/WSA	Integer	1, 0	0	–
clockphase	M/WSA	Integer	1, 0	0	–
lsbfirst	M/WSA	Boolean	1, 0	0	–
extradelat	M/WSA	Boolean	1, 0	0	–
targetssdelay	M/WSA	Decimal	≥ 0	100	seconds
delayunits	M/WSA	String	“ns”, “us”, “ms”, “s”	us	–
clockunits	M/WSA	String	“MHz”, “kHz”, “Hz”	kHz	–

Notes

- (1) The **Section** column describes the parameter's location in the PTF:
 S/WSA = SYSTEM/WIZARD_SCRIPT_ARGUMENTS
 M/WSA = MODULE/WIZARD_SCRIPT_ARGUMENTS

clock_freq

The `clock_freq` assignment is the global system clock frequency. This setting is derived from the system PTF, and is not set by SOPC Builder.

ismaster

When `ismaster` is set to 1, the SPI peripheral is a master device. When `ismaster` is set to 0, the SPI peripheral is a slave device.

databits

The `databits` assignment determines the SPI peripheral's send and receive size. That is, it sets the number of bits transmitted or received by the SPI peripheral. The `databits` assignment also determines the `rxdata` and `txdata` register sizes.

For a master and slave SPI to work together, `databits` must have the same value for both peripherals.

targetclock

The `targetclock` assignment determines the SPI peripheral's target clock frequency. The SPI device's actual frequency may not actually be the `targetclock` value. The only possible values for the achieved clock frequency are:

$$(<system\ clock\ frequency>) \div [2, 4, 6, 8, \dots]$$

The actual frequency achieved will not be greater than the `targetclock` value, based on the system frequency specified. For example, if the system clock frequency is 33333000 Hz and the `targetclock` value is 16000000 Hz, then the actual clock frequency is 8333250 Hz, or a clock divisor of 4 (because a clock divisor of 2 would create a clock frequency greater than 16,000,000 Hz).



When `ismaster` is set to 0, this assignment is ignored.

numslaves

The `numslaves` assignment determines the number of slaves a master can address. For every slave specified, a chip select pin is added to the SPI master peripheral.



When `ismaster` is set to 0, this assignment is ignored.

clockpolarity

When `clockpolarity` is 1, the clock's idle state is high. When `clockpolarity` is 0, the clock's idle state is low.

clockphase

When `clockphase` is 0, data is latched on the SPI clock's leading edge, and data changes on the SPI clock's trailing edge. When `clockphase` is 1, data is latched on the SPI clock's trailing edge, and data changes on the SPI clock's rising edge.

Figure 2 through Figure 5 on page 12 illustrate the SPI pins' possible behaviors based on the `clockpolarity` and `clockphase` values.

Figure 2. *clockpolarity = 0, clockphase = 0*



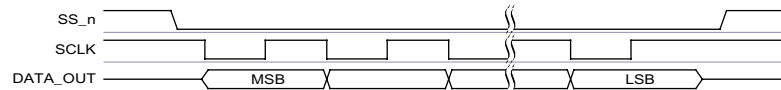
Figure 3. *clockpolarity = 0, clockphase = 1*



Figure 4. *clockpolarity = 1, clockphase = 0*



Figure 5. *clockpolarity = 1, clockphase = 1*



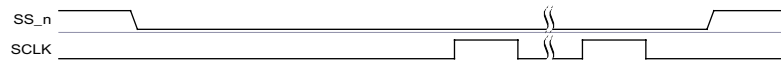
lsbfirst

When `lsbfirst` is set to 1, data is transferred LSB-first. When `lsbfirst` is set to 0, data is transferred MSB-first.

extradelat

When `extradelat` is set to 1, a master SPI peripheral inserts a delay between the falling edge of a chip select to a slave peripheral and the first shift transaction, as shown:

Figure 6. *extradelat=1*



When `extradelay` is set to 1, the `targetssdelay` assignment is enabled.



When `ismaster` is set to 0, this assignment is ignored.

targetssdelay

The `targetssdelay` assignment determines how much extra delay a master SPI peripheral requires between the falling edge of a chip select to a slave peripheral and the first shift transaction. The SPI peripheral, however, utilizes a delay granularity equal to $\frac{1}{2}$ SPI clock period. Thus, the actual `ssdelay` is the `targetssdelay` rounded up to the nearest multiple of $\frac{1}{2}$ the SPI clock period, as shown in the following equation:

$$p = \frac{1}{2} \times \text{sclk_period} ;$$
$$\text{delay} = \left\lceil \frac{\text{targetssdelay}}{p} \right\rceil \times p$$

`sclk_period` is the period of the `sclk` frequency. See [“targetclock” on page 11](#) for details.



When either `ismaster` or `extradelay` are set to 0, this assignment is ignored.

delayunits

The `delayunits` assignment determines the unit of measure to be used for the `targetssdelay` assignment.

clockunits

The `clockunits` assignment determines the unit of measure to be used for the `targetclock` assignment.



101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
<http://www.altera.com>
Applications Hotline:
(800) 800-EPLD
Literature Services:
lit_req@altera.com

Copyright © 2003 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.



I.S. EN ISO 9001