5. SPI Core

5.1. Core Overview

SPI is an industry-standard serial protocol commonly used in embedded systems to connect microprocessors to a variety of off-chip sensor, conversion, memory, and control devices. The SPI core with Avalon interface implements the SPI protocol and provides an Avalon Memory-Mapped (Avalon-MM) interface on the back end.

The SPI core can implement either the master or slave protocol. When configured as a master, the core can control up to 32 independent SPI slaves. The width of the receive and transmit registers are configurable between 1 and 32 bits. Longer transfer lengths can be supported with software routines. The core provides an interrupt output that can flag an interrupt whenever a transfer completes.

5.2. Functional Description

The SPI core communicates using two data lines, a control line, and a synchronization clock:

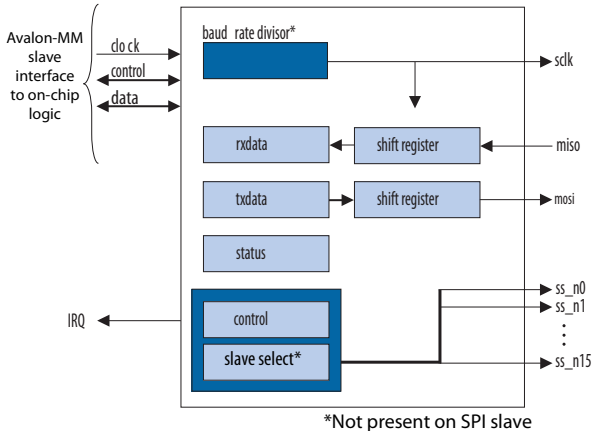
- Master Out Slave In (`mosi`)—Output data from the master to the inputs of the slaves
- Master In Slave Out (`miso`)—Output data from a slave to the input of the master
- Serial Clock (`sclk`)—Clock driven by the master to slaves, used to synchronize the data bits
- Slave Select (`ss_n`)— Select signal (active low) driven by the master to individual slaves, used to select the target slave

The SPI core has the following user-visible features:

- A memory-mapped register space comprised of five registers: `rxdata`, `txdata`, `status`, `control`, and `slaveselect`
- Four SPI interface ports: `sclk`, `ss_n`, `mosi`, and `miso`

The registers provide an interface to the SPI core and are visible via the Avalon-MM slave port. The `sclk`, `ss_n`, `mosi`, and `miso` ports provide the hardware interface to other SPI devices. The behavior of `sclk`, `ss_n`, `mosi`, and `miso` depends on whether the SPI core is configured as a master or slave.

Figure 7. SPI Core Block Diagram (Master Mode)



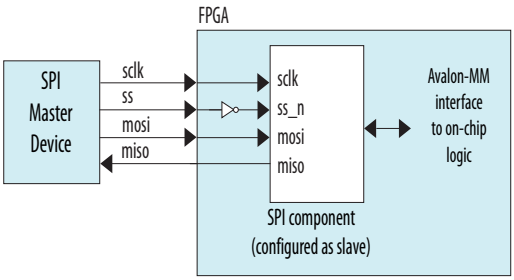
The SPI core logic is synchronous to the clock input provided by the Avalon-MM interface. When configured as a master, the core divides the Avalon-MM clock to generate the SCLK output. When configured as a slave, the core's receive logic is synchronized to SCLK input.

For more details, refer to the "Interval Timer Core" chapter.

5.2.1. Example Configurations

The core block diagram and the SPI core configured as a slave diagram show two possible configurations. In Figure 8 on page 41 the core provides a slave interface to an off-chip SPI master.

Figure 8. SPI Core Configured as a Slave



In the SPI core block diagram, the SPI core provides a master interface driving multiple off-chip slave devices. Each slave device in Figure 8 on page 41 must tristate its `miso` output whenever its select signal is not asserted.

The `ss_n` signal is active-low. However, any signal can be inverted inside the FPGA, allowing the slave-select signals to be either active high or active low.

5.2.2. Transmitter Logic

The core transmitter logic consists of a transmit holding register (`txdata`) and transmit shift register, each n bits wide. The register width n is specified at system generation time, and can be any integer value from 8 to 32. After a master peripheral writes a value to the `txdata` register, the value is copied to the shift register and then transmitted when the next operation starts.

The shift register and the `txdata` register provide double buffering during data transmission. A new value can be written into the `txdata` register while the previous data is being shifted out of the shift register. The transmitter logic automatically transfers the `txdata` register to the shift register whenever a serial shift operation is not currently in process.

In master mode, the transmit shift register directly feeds the `mosi` output. In slave mode, the transmit shift register directly feeds the `miso` output. Data shifts out LSB first or MSB first, depending on the configuration of the SPI core.

5.2.3. Receiver Logic

The core receive logic consists of a receive holding register (`rxdata`) and receive shift register, each n bits wide. The register width n is specified at system generation time, and can be any integer value from 8 to 32. A master peripheral reads received data from the `rxdata` register after the shift register has captured a full n-bit value of data.

The shift register and the `rxdata` register provide double buffering while receiving data. The `rxdata` register can hold a previously received data value while subsequent new data is shifting into the shift register. The receiver logic automatically transfers the shift register content to the `rxdata` register when a serial shift operation completes.

In master mode, the shift register is fed directly by the `miso` input. In slave mode, the shift register is fed directly by the `mosi` input. The receiver logic expects input data to arrive LSB first or MSB first, depending on the configuration of the SPI core.

5.2.4. Master and Slave Modes

At system generation time, the designer configures the SPI core in either master mode or slave mode. The mode cannot be switched at runtime.

5.2.4.1. Master Mode Operation

In master mode, the SPI ports behave as shown in the table below.

Table 14. Master Mode Port Configurations

Name	Direction	Description
`mosi`	output	Data output to slave(s)
`miso`	input	Data input from slave(s)
`sclk`	output	Synchronization clock to all slaves
`ss_nM`	output	Slave select signal to slave M, where M is a number between 0 and 31.

In master mode, an intelligent host (for example, a microprocessor) configures the SPI core using the `control` and `slaveselect` registers, and then writes data to the `txdata` buffer to initiate a transaction. A master peripheral can monitor the status of the transaction by reading the `status` register. A master peripheral can enable interrupts to notify the host whenever new data is received (for example, a transfer has completed), or whenever the transmit buffer is ready for new data.

The SPI protocol is full duplex, so for every transaction both sends and receives data at the same time. The master transmits a new data bit on the `mosi` output and the slave drives a new data bit on the `miso` input for each active edge of `sclk`. The SPI core divides the Avalon-MM system clock using a clock divider to generate the `sclk` signal.

When the SPI core is configured to interface with multiple slaves, the core has one `ss_n` signal for each slave. During a transfer, the master asserts `ss_n` to each slave specified in the `slaveselect` register. Note that there can be no more than one slave transmitting data during any particular transfer, or else there will be a contention on the `miso` input. The number of slave devices is specified at system generation time.

5.2.4.2. Slave Mode Operation

In slave mode, the SPI ports behave as shown in the table below.

Table 15. Slave Mode Port Configurations

Name	Direction	Description
`mosi`	input	Data input from the master
`miso`	output	Data output to the master
`sclk`	input	Synchronization clock
`ss_n`	input	Select signal

In slave mode, the SPI core simply waits for the master to initiate transactions. Before a transaction begins, the slave logic continuously polls the `ss_n` input. When the master asserts `ss_n`, the slave logic immediately begins sending the transmit shift register contents to the `miso` output. The slave logic also captures data on the `mosi` input, and fills the receive shift register simultaneously. After a word is received by the slave, the master must de-assert the `ss_n` signal and reasserts the signal again when the next word is ready to be sent.

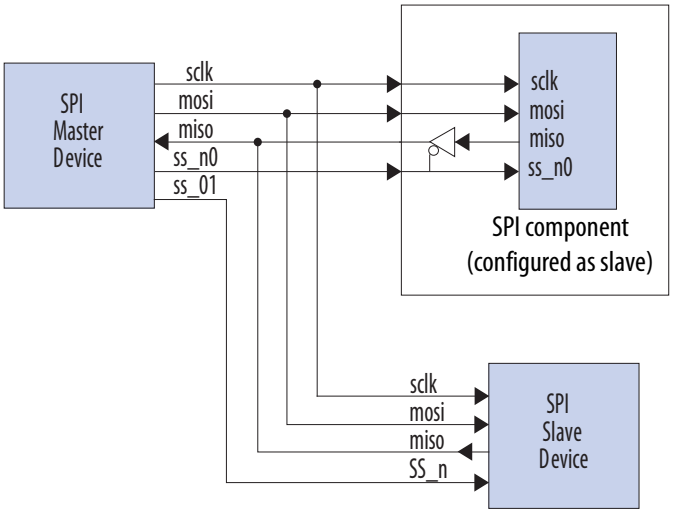
An intelligent host such as a microprocessor writes data to the `txdata` registers, so that it is transmitted the next time the master initiates an operation. A master peripheral reads received data from the `rxdata` register. A master peripheral can enable interrupts to notify the host whenever new data is received, or whenever the transmit buffer is ready for new data.

5.2.4.3. Multi-Slave Environments

When `ss_n` is not asserted, typical SPI cores set their `miso` output pins to high impedance. The provided SPI slave core drives an undefined high or low value on its `miso` output when not selected. Special consideration is necessary to avoid signal

contention on the `miso` output, if the SPI core in slave mode is connected to an off-chip SPI master device with multiple slaves. In this case, the `ss_n` input should be used to control a tristate buffer on the `miso` signal.

Figure 9. SPI Core in a Multi-Slave Environment



5.3. Configuration

The following sections describe the available configuration options.

5.3.1. Master/Slave Settings

The designer can select either master mode or slave mode to determine the role of the SPI core. When master mode is selected, the following options are available: **Number of select (SS_n) signals**, **SPI clock rate**, and **Specify delay**.

5.3.1.1. Number of Select (SS_n) Signals

This setting specifies the number of slaves the SPI master connects to. The range is 1 to 32. The SPI master core presents a unique `ss_n` signal for each slave.

5.3.1.2. SPI Clock (sclk) Rate

This setting determines the rate of the `sclk` signal that synchronizes data between master and slaves. The target clock rate can be specified in units of Hz, kHz or MHz. The SPI master core uses the Avalon-MM system clock and a clock divisor to generate `sclk`.

The actual frequency of `sclk` may not exactly match the desired target clock rate. The achievable clock values are:

<Avalon-MM system clock frequency> / [2, 4, 6, 8, ...]

The actual frequency achieved will not be greater than the specified target value.

When the SPI core is turned on with a synchronizer, the IP clock frequency must be at least six times the SPI clock frequency.

5.3.1.3. Specify Delay

Turning on this option causes the SPI master to add a time delay between asserting the ss_n signal and shifting the first bit of data. This delay is required by certain SPI slave devices. If the delay option is on, you must also specify the delay time in units of ns, µs or ms. An example is shown in below.

Figure 10. Time Delay Between Asserting ss_n and Toggling sclk

The delay generation logic uses a granularity of half the period of sclk. The actual delay achieved is the desired target delay rounded up to the nearest multiple of half the sclk period, as shown in the follow two equations.

Table 16.

p = 1/2 x (period of sclk)

Table 17.

Actual delay = ceiling x (desired delay/ p)

5.3.2. Data Register Settings

The data register settings affect the size and behavior of the data registers in the SPI core. There are two data register settings:

- **Width**—This setting specifies the width of rxdata, txdata, and the receive and transmit shift registers. The range is from 1 to 32.
- **Shift direction**—This setting determines the direction that data shifts (MSB first or LSB first) into and out of the shift registers.

5.3.3. Timing Settings

The timing settings affect the timing relationship between the ss_n, sclk, mosi and miso signals. In this discussion the mosi and miso signals are referred to generically as data. There are two timing settings:

- **Clock polarity**—This setting can be 0 or 1. When clock polarity is set to 0, the idle state for sclk is low. When clock polarity is set to 1, the idle state for sclk is high.
- **Clock phase**—This setting can be 0 or 1. When clock phase is 0, data is latched on the leading edge of sclk, and data changes on trailing edge. When clock phase is 1, data is latched on the trailing edge of sclk, and data changes on the leading edge.

  The following four clock polarity figures demonstrate the behavior of signals in all possible cases of clock polarity and clock phase.

Figure 11. Clock Polarity = 0, Clock Phase = 0

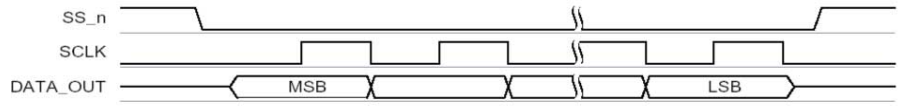


Figure 12. Clock Polarity = 0, Clock Phase = 1

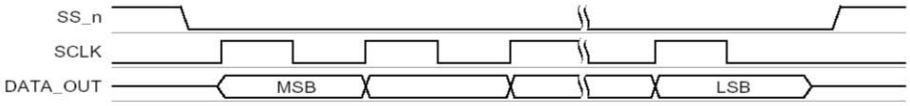


Figure 13. Clock Polarity = 1, Clock Phase = 0

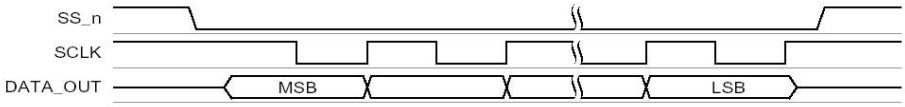
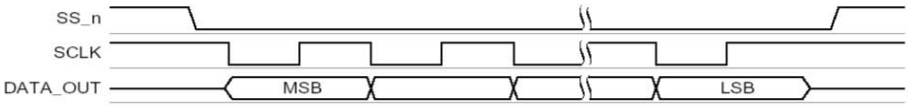


Figure 14. Clock Polarity = 1, Clock Phase = 1



5.3.4. Synchronizer Stages

This setting enables a synchronizer on the SPI interface input. When slave mode is used, you must turn on the synchronizer.

5.4. Software Programming Model

The following sections describe the software programming model for the SPI core, including the register map and software constructs used to access the hardware. For Nios II processor users, Intel provides the HAL system library header file that defines the SPI core registers. The SPI core does not match the generic device model categories supported by the HAL, so it cannot be accessed via the HAL API or the ANSI C standard library. Intel provides a routine to access the SPI hardware that is specific to the SPI core.

5.4.1. Hardware Access Routines

Intel provides one access routine, `alt_avalon_spi_command()`, that provides general-purpose access to the SPI core that is configured as a master.

5.4.1.1. alt_avalon_spi_command()

Prototype:	`int alt_avalon_spi_command(alt_u32 base, alt_u32 slave,`

continued...

	alt_u32 write_length, const alt_u8* wdata, alt_u32 read_length, alt_u8* read_data, alt_u32 flags)
Thread-safe:	No.
Available from ISR:	No.
Include:	<altera_avalon_spi.h>
Description:	This function performs a control sequence on the SPI bus. It supports only SPI masters with data width less than or equal to 8 bits. A single call to this function writes a data buffer of arbitrary length to the `mosi` port, and then reads back an arbitrary amount of data from the `miso` port. The function performs the following actions: (1) Asserts the slave select output for the specified slave. The first slave select output is 0. (2) Transmits write_length bytes of data from wdata through the SPI interface, discarding the incoming data on the `miso` port. (3) Reads read_length bytes of data and stores the data into the buffer pointed to by read_data. The `mosi` port is set to zero during the read transaction. (4) De-asserts the slave select output, unless the flags field contains the value ALT_AVALON_SPI_COMMAND_MERGE. If you want to transmit from scattered buffers, call the function multiple times and specify the merge flag on all the accesses except the last. To access the SPI bus from more than one thread, you must use a semaphore or mutex to ensure that only one thread is executing within this function at any time.
Returns:	The number of bytes stored in the read_data buffer.

5.4.2. Software Files

The core is accompanied by the following software files. These files provide a low-level interface to the hardware.

- `altera_avalon_spi.h`—This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- `altera_avalon_spi.c`—This file implements low-level routines to access the hardware.

5.4.3. Register Map

An Avalon-MM master peripheral controls and communicates with the core via the six 32-bit registers, shown in below in the **Register Map for SPI Master Device** figure. The table assumes an n-bit data width for `rxdata` and `txdata`.

Table 18. Register Map for SPI Master Device

Internal Address	Register Name	Type [R/W]	32-11	10	9	8	7	6	5	4	3	2-0
0	`rxdata` (3)	R	RXDATA (n-1..0)									
1	`txdata` (3)	W	TXDATA (n-1..0)									
2	`status` (1)	R/W			EOP	E	RRDY	TRDY	TMT	TOE	ROE	
3	`control`	R/W		SSO (2)	IEOP	IE	IRRDY	ITRDY		ITOE	IROE	
4	Reserved	—										
5	`slaveselect` (2)	R/W	Slave Select Mask									
6	`eop_value`(3)	R/W	End of Packet Value (n-1..0)									

Reading undefined bits returns an undefined value. Writing to undefined bits has no effect.

5.4.3.1. rxdata Register

A master peripheral reads received data from the `rxdata` register. When the receive shift register receives a full n bits of data, the `status` register's RRDY bit is set to 1 and the data is transferred into the `rxdata` register. Reading the `rxdata` register clears the RRDY bit. Writing to the `rxdata` register has no effect.

New data is always transferred into the `rxdata` register, whether or not the previous data was retrieved. If RRDY is 1 when data is transferred into the `rxdata` register (that is, the previous data was not retrieved), a receive-overrun error occurs and the `status` register's ROE bit is set to 1. In this case, the contents of `rxdata` are undefined.

5.4.3.2. txdata Register

A master peripheral writes data to be transmitted into the `txdata` register. When the `status` register's TRDY bit is 1, it indicates that the `txdata` register is ready for new data. The TRDY bit is set to 0 whenever the `txdata` register is written. The TRDY bit is set to 1 after data is transferred from the `txdata` register into the transmitter shift register, which readies the `txdata` holding register to receive new data.

(1) A write operation to the `status` register clears the ROE, TOE, and E bits.

(2) Present only in master mode.

(3) Bits 31 to n are undefined when n is less than 32.

A master peripheral should not write to the txdata register until the transmitter is ready for new data. If TRDY is 0 and a master peripheral writes new data to the txdata register, a transmit-overrun error occurs and the status register's T0E bit is set to 1. In this case, the new data is ignored, and the content of txdata remains unchanged.

As an example, assume that the SPI core is idle (that is, the txdata register and transmit shift register are empty), when a CPU writes a data value into the txdata holding register. The TRDY bit is set to 0 momentarily, but after the data in txdata is transferred into the transmitter shift register, TRDY returns to 1. The CPU writes a second data value into the txdata register, and again the TRDY bit is set to 0. This time the shift register is still busy transferring the original data value, so the TRDY bit remains at 0 until the shift operation completes. When the operation completes, the second data value is transferred into the transmitter shift register and the TRDY bit is again set to 1.

5.4.3.3. status Register

The status register consists of bits that indicate status conditions in the SPI core. Each bit is associated with a corresponding interrupt-enable bit in the control register, as discussed in the **Control Register** section. A master peripheral can read status at any time without changing the value of any bits. Writing status does clear the R0E, T0E and E bits.

Table 19. status Register Bits

#	Name	Description
3	R0E	**Receive-overrun error** The R0E bit is set to 1 if new data is received while the rxdata register is full (that is, while the RRDY bit is 1). In this case, the new data overwrites the old. Writing to the status register clears the R0E bit to 0.
4	T0E	**Transmitter-overrun error** The T0E bit is set to 1 if new data is written to the txdata register while it is still full (that is, while the TRDY bit is 0). In this case, the new data is ignored. Writing to the status register clears the T0E bit to 0.
5	TMT	**Transmitter shift-register empty** In master mode, the TMT bit is set to 0 when a transaction is in progress and set to 1 when the shift register is empty. In slave mode, the TMT bit is set to 0 when the slave is selected (SS_n is low) or when the SPI Slave register interface is not ready to receive data.
6	TRDY	**Transmitter ready** The TRDY bit is set to 1 when the txdata register is empty.
7	RRDY	**Receiver ready** The RRDY bit is set to 1 when the rxdata register is full.
8	E	**Error** The E bit is the logical OR of the T0E and R0E bits. This is a convenience for the programmer to detect error conditions. Writing to the status register clears the E bit to 0.
9	EOP	**End of Packet** The EOP bit is set when the End of Packet condition is detected. The End of Packet condition is detected when either the read data of the rxdata register or the write data to the txdata register is matching the content of the eop_value register.

5.4.3.4. control Register

The `control` register consists of data bits to control the SPI core's operation. A master peripheral can read `control` at any time without changing the value of any bits.

Most bits (`IROE`, `ITOE`, `ITRDY`, `IRRDY`, and `IE`) in the `control` register control interrupts for status conditions represented in the `status` register. For example, bit 1 of `status` is ROE (receiver-overrun error), and bit 1 of control is `IROE`, which enables interrupts for the ROE condition. The SPI core asserts an interrupt request when the corresponding bits in `status` and `control` are both 1.

Table 20. control Register Bits

#	Name	Description
3	IROE	Setting IROE to 1 enables interrupts for receive-overrun errors.
4	ITOE	Setting ITOE to 1 enables interrupts for transmitter-overrun errors.
6	ITRDY	Setting ITRDY to 1 enables interrupts for the transmitter ready condition.
7	IRRDY	Setting IRRDY to 1 enables interrupts for the receiver ready condition.
8	IE	Setting IE to 1 enables interrupts for any error condition.
9	IEOP	Setting IEOP to 1 enables interrupts for the End of Packet condition.
10	SSO	Setting SSO to 1 forces the SPI core to drive its ss_n outputs, regardless of whether a serial shift operation is in progress or not. The slaveselect register controls which ss_n outputs are asserted. SSO can be used to transmit or receive data of arbitrary size, for example, greater than 32 bits.

After reset, all bits of the `control` register are set to 0. All interrupts are disabled and no `ss_n` signals are asserted.

5.4.3.5. slaveselect Register

The `slaveselect` register is a bit mask for the `ss_n` signals driven by an SPI master. During a serial shift operation, the SPI master selects only the slave device(s) specified in the `slaveselect` register.

The `slaveselect` register is only present when the SPI core is configured in master mode. There is one bit in `slaveselect` for each `ss_n` output, as specified by the designer at system generation time. The `slaveselect` register value is only updated either at the beginning of the actual SPI transmission or when the SSO bit is set from 0 to 1.

A master peripheral can set multiple bits of `slaveselect` simultaneously, causing the SPI master to simultaneously select multiple slave devices as it performs a transaction. For example, to enable communication with slave devices 1, 5, and 6, set bits 1, 5, and 6 of `slaveselect`. However, consideration is necessary to avoid signal contention between multiple slaves on their `miso` outputs.

Upon reset, bit 0 is set to 1, and all other bits are cleared to 0. Thus, after a device reset, slave device 0 is automatically selected.

5.4.3.6. end of packet value Register

The end of packet value register allows you to specify the value of the SPI data word. The SPI data word acts as the end of packet word.

5.5. Example Test Code

```
#include "alt_types.h"
#include "sys/alt_stdio.h"
#include "io.h"
#include "system.h"
#include "sys/alt_cache.h"
#include "altera_avalon_spi.h"
#include "altera_avalon_spi_regs.h"
#include "sys/alt_irq.h"

//This is the ISR that runs when the SPI Slave receives data

static void spi_rx_isr(void* isr_context){

        alt_printf("ISR :) %x \n" ,
IORD_ALTERA_AVALON_SPI_RXDATA(SPI_SLAVE_BASE));

        //This resets the IRQ flag. Otherwise the IRQ will continuously run.
        IOWR_ALTERA_AVALON_SPI_STATUS(SPI_SLAVE_BASE, 0x0);
}

int main()
{
  alt_printf("Hello from Nios II!\n");

  int return_code,ret;
  char spi_command_string_tx[10] = "$HELLOABC*";

  char spi_command_string_rx[10] = "$HELLOABC*";

  //This registers the Slave IRQ with NIOS

  ret =
alt_ic_isr_register(SPI_SLAVE_IRQ_INTERRUPT_CONTROLLER_ID,SPI_SLAVE_IRQ,spi_rx_i
sr,(void *)spi_command_string_tx,0x0);
  alt_printf("IRQ register return %x \n", ret);

  //You need to enable the IRQ in the IP core control register as well.

IOWR_ALTERA_AVALON_SPI_CONTROL(SPI_SLAVE_BASE,ALTERA_AVALON_SPI_CONTROL_SSO_MSK
| ALTERA_AVALON_SPI_CONTROL_IRRDY_MSK);

  //Just calling the ISR to see if the function is OK.
  spi_rx_isr(NULL);

  return_code = alt_avalon_spi_command(SPI_MASTER_BASE,0 ,
                               1, spi_command_string_tx,
                               0, spi_command_string_rx,
                               0);

  return_code = alt_avalon_spi_command(SPI_MASTER_BASE,0 ,
                                 1, &spi_command_string_tx[1],
                                 0, spi_command_string_rx,
                                 0);

  return_code = alt_avalon_spi_command(SPI_MASTER_BASE,0 ,
                                 1, &spi_command_string_tx[2],
                                 0, spi_command_string_rx,
                                 0);

  return_code = alt_avalon_spi_command(SPI_MASTER_BASE,0 ,
                                 1, &spi_command_string_tx[3],
```

```
                                0, spi_command_string_rx,
                                0);

  if(return_code < 0)
                alt_printf("ERROR SPI TX RET = %x \n" , return_code);

  alt_printf("Transmit done. RET = %x spi_rx %x
\n",return_code,spi_command_string_rx[0]);

  //RX is done via interrupts.
  alt_printf("Rx done \n");
  return 0;

}
```

5.6. SPI Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2019.12.16	19.4	Added the following new sections: • *Synchronizer Stages* • *Example Test Code*
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
June 2016	2016.06.17	Updates: • Removed content regarding Avalon-MM flow control • Table 18 on page 48: `eop_value` added • Table 19 on page 49: `EOP` added • Table 20 on page 50: `IEOP` added • end of packet value Register on page 51: New topic
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	Revised register width in transmitter logic and receiver logic. Added description on the disable flow control option. Added R/W column in Table 8–3 .
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. Updated the width of the parameters and signals from 16 to 32.
May 2008	v8.0.0	Updated the description of the `TMT` bit.