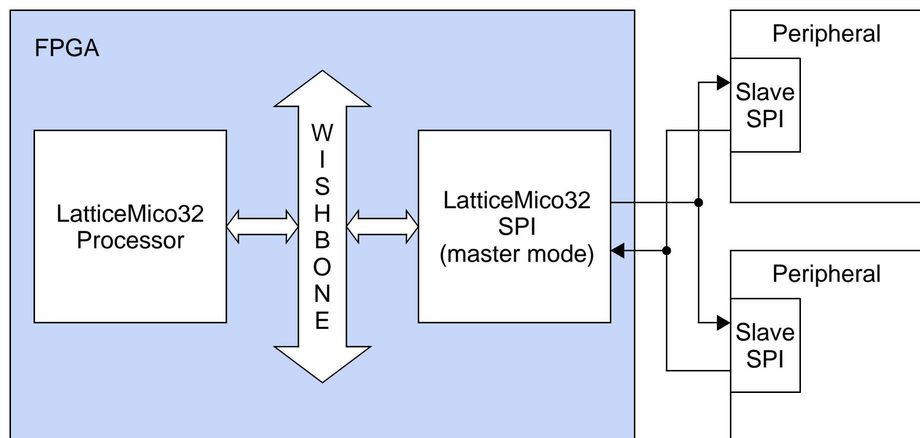
LatticeMico32 SPI

The LatticeMico32 serial peripheral interface (SPI) provides an industry-standard interface between a LatticeMico32 processor and off-chip peripherals, as shown in Figure 1. In master mode, the SPI can be configured to control communication with up to 32 off-chip SPI ports. In slave mode, the SPI supports communications with an off-chip SPI master.

Figure 1: Using LatticeMico32 SPI to Communicate with Peripherals



As a simple serial port, an SPI uses few FPGA resources (about 150 slices) and little board space for wires but runs much slower than a parallel port. The LatticeMico32 SPI uses only three pins (clock, data in, and data out) plus one select for each slave device. An SPI is a good choice for communicating with low-speed devices that are accessed intermittently and transfer data streams rather than reading and writing to specific addresses. An SPI is an especially good choice if you can take advantage of its full-duplex nature, which sends and receives data at the same time.

Version

This document describes the 3.0 version (formerly the 7.0 SP2 version) of the LatticeMico32 SPI.

Features

The LatticeMico32 SPI provides standard, fully configurable SPI ports including:

- WISHBONE B.3 interface
- Slave and master modes. Master mode can control up to 32 slaves.
- Interrupt request to the processor, configurable for a variety of status conditions
- Library of basic data structures and software routines for operating SPIs
- Configurable serial clock (SCLK) frequency
- Configurable timing relationships between data and clock signals, and between data and slave-select signals
- Double-buffered transmission, allowing new data to be written at the same time that previous data is being shifted out
- Receive and transmit registers configurable from 1 to 32 bits wide. Longer transfers can be done with software support.
- Option for least-significant bit or most-significant bit first

For additional details about the WISHBONE bus, refer to the *LatticeMico32 Processor Reference Manual*.

Functional Description

Figure 2 shows the LatticeMico32 SPI configured as a master port, and Figure 3 shows it configured as a slave port.

On the internal side (the left in these diagrams), the SPI has a standard WISHBONE slave bus, which connects the SPI with a LatticeMico32 processor and other on-chip components. From the WISHBONE bus, the SPI appears as a set of addressable registers that can be read or written. Through these registers, the microprocessor can transmit and receive data and control the operation of the SPI.

For a description of the WISHBONE bus, refer to the *LatticeMico32 Processor Reference Manual*.

On the external side (the right in these diagrams), the SPI has a standard master or slave SPI interface:

Figure 2: LatticeMico32 SPI Master Implementation

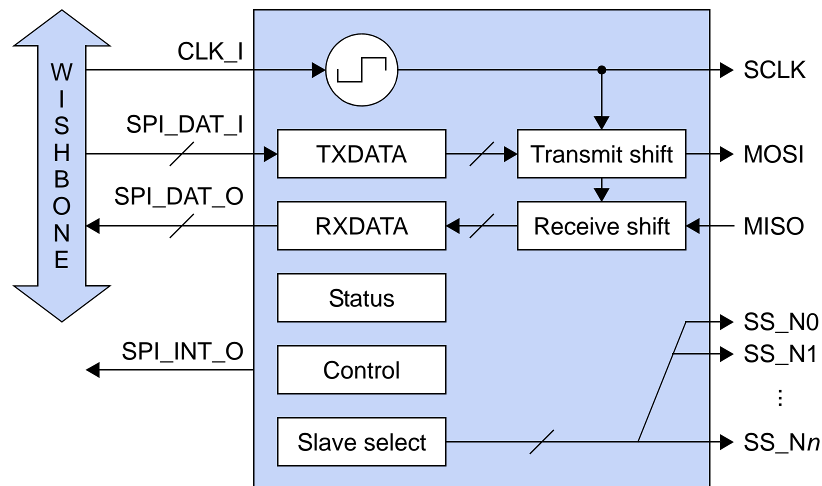
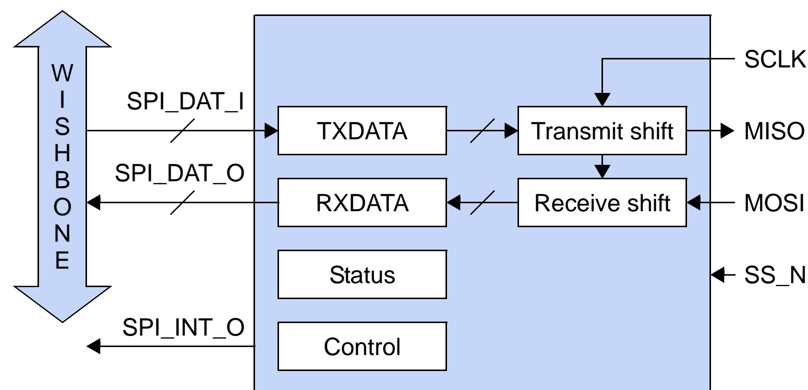


Figure 3: LatticeMico32 SPI Slave Implementation



- SCLK (serial clock) generated by the master SPI to synchronize the data transfers.
- MISO (master in, slave out), which transfers data going to the master SPI from a slave.
- MOSI (master out, slave in), which transfers data going from the master SPI to a slave.
- SS_N (slave select), which is asserted by the master SPI to start a data transfer. In master mode, the SPI has a slave select signal (SS_N0, SS_N1, and so on) for each slave SPI. In slave mode, the SPI has a single SS_N input.

Data Transmit and Receive

Both of the data transmit and receive paths within the LatticeMico32 SPI use two registers: a holding register and a shift register. This double buffering allows the paths to hold one data frame while another is being shifted in or out. The holding registers, TXDATA and RXDATA, are addressable and can be written to or read through the WISHBONE bus.

In the transmit path, TXDATA is written to (or read) through the WISHBONE bus. Writing to TXDATA clears the transmitter ready status bit (TRDY) to 0, blocking any new data until the previous data has moved to the shift register. If no serial transfer is in process, TXDATA immediately moves its data to the shift register and sets TRDY to 1. If a serial transfer is in process, TXDATA holds the new data until the previous data has shifted out. If new data comes in while TRDY is 0, the new data is blocked and the transmit overrun error status bit (TOE) is set.

In the receive path, the shift register, when full, immediately moves its data to the holding register, RXDATA, and sets the receiver ready status bit (RRDY) to 1. RXDATA can be read through the WISHBONE bus. Reading RXDATA clears the RRDY status bit. If new data comes in while RRDY is set, RXDATA is overwritten with the new data and the receive overrun error status bit (ROE) is set.

SPI operations are always full duplex, so every data transfer operation transmits and receives at the same time. If you just want to receive, your software must load TXDATA with appropriate dummy data to transmit. Your software must also use or ignore the received data as appropriate.

The registers can be configured up to 32 bits wide. The transmit and receive logic can be configured to assume the data is either least-significant bit (LSB) first or most-significant bit (MSB) first.

Status and Control

The SPI includes status and control registers. These are mainly used to trigger interrupt requests. The master SPI also has a slave-select register that is used to select a slave SPI and start a data transfer.

To check the status of the SPI, read the status register. It reports conditions such as receive and transmit overrun, transmit shift register empty, and transmitter and receiver ready. For details, see Table 7 on page 10.

To set up an interrupt request on the SPI_INT_O output, set one or more of the interrupt enable bits in the control register. These bits enable interrupt requests for most of the conditions reported in the status register. For details, see Table 8 on page 11.

To clear an interrupt request, clear the associated bit in the status register. Writing any value to the status register clears the overrun error status bits (ROE, TOE, and E). Writing the TXDATA register clears the transmit ready status bit (TRDY). Reading the RXDATA register clears the receiver ready status bit (RRDY).

To start a data transfer, load the slave select register of the master SPI with a slave mask and then load the TXDATA register. Loading TXDATA triggers the next data transfer. The slave select register has one bit for each SS_N output. Setting the bit to 1 asserts the active-low SS_N. For example, a slave mask of 0x00000020 asserts SS_N5, selecting that slave SPI. It is possible to assert more than one slave select signal, but you must take care to prevent contention on the MISO bus.

To have a data transfer longer than the transmit and receive registers, set the SSO bit in the control register to 1. SSO holds the slave select signal after it would normally be de-asserted. The SPIs will continue to exchange data frames until SSO is cleared. SCLK stops toggling between frames. Before clearing SSO to end the data transfer, make sure the transmit shift register is empty by checking the TMT status bit.

A slave SPI cannot start a data transfer. When the SPI's SS_N input goes low, the SPI immediately begins the data transfer.

Clocking Sources

The master SPI generates SCLK to synchronize the data transfers. Each SPI transmits a new data bit with each active edge of SCLK. SCLK is only available during the data transfer, while SS_N is asserted.

SCLK is derived from the system clock, CLK_I, by dividing the frequency. The divisor and other aspects of SCLK can be selected when the SPI is configured. For details, see the following section.

Configuration

The following sections describe the graphical user interface (UI) parameters, the hardware description language (HDL) parameters, and the I/O ports that you can use to configure and operate the LatticeMico32 SPI.

UI Parameters

Table 1 shows the UI parameters available for configuring the LatticeMico32 SPI through the Mico System Builder (MSB) interface.

Table 1: LatticeMico32 SPI UI Parameters

Dialog Box Option	Description	Allowable Values	Default Value
Instance Name	Specifies the name of the SPI instance.	Alphanumeric and underscores	spi
Base Address	Specifies the base address for the device. The minimum byte alignment is 0X80.	0X80000000 – 0XFFFFFF80 If other components are included in the platform, the range of allowable values will vary.	0X80000000
Setting			

Table 1: LatticeMico32 SPI UI Parameters (Continued)

Dialog Box Option	Description	Allowable Values	Default Value
Data Length	Specifies the number of serial data bits per SPI transaction.	1-32 (master) 1-32 (slave)	24 (master) 24 (slave)
Shift Direction	Specifies whether the most significant bit or the least significant bit is first. ◆ If 0, the most significant bit is first. ◆ If 1, the least significant bit is first.	0|1 (master/slave)	1 (master/ slave)
Phase	Specifies the clock phase of the SPI instance: ◆ If 0, data is latched on the leading edge of SCLK, and the data changes on the trailing edge. ◆ If 1, data is latched on the trailing edge of SCLK, and the data changes on the leading edge.	0|1 (master/slave)	1
Polarity	Specifies the polarity of the SPI instance: ◆ If 0, the idle state for SCLK is low. ◆ If 1, the idle state for SCLK is high.	0|1 (master/slave)	1
Master			
Master	Specifes whether the SPI instance is a master or a slave. When selected, the LatticeMico32 SPI acts as a master; otherwise, it acts as a slave.	selected|not selected	not selected
Number of Slaves	Specifies the number of slave devices supported.	1-32 (master)	1
SCLK Rate	Specifies the factor for deriving SCLK from the component input clock (processor clock, CLK_I). SCLK is derived from the following equation: $SCLK = \frac{CLK_I}{2 \times (SCLK_Rate + 1)}$ For example: For $SCLK_Rate = 0$: $SCLK = \frac{CLK_I}{2}$ For $SCLK_Rate = 1$: $SCLK = \frac{CLK_I}{4}$ **Note**: The SCLK Rate value is used as the CLOCK_SEL parameter in the HDL.	0 through $2^{clock_counter_width} - 1$	7

Table 1: LatticeMico32 SPI UI Parameters (Continued)

Dialog Box Option	Description	Allowable Values	Default Value
Clock Counter Width	Sets the range limit for the clock counter. The width should be enough to meet the number of bits required for the slave clock.	1-32	16
Tx Start Delay	Specifies the time delay factor before shifting the first bit of data after the SS_N signal is asserted. The start delay time is derived from the following equation: $Delay = Tx_Start_Delay \times \left(\frac{SCLK_Period}{2}\right)$ **Note**: The Tx Start Delay value is used as the DELAY_TIME parameter in the HDL.	0-63	3
Tx Interframe Pause	Specifies the number of SCLK cycles for which the SS_N signal is held inactive between SPI transmit requests. **Note**: The Tx Interframe Pause parameter is used as the INTERVAL_LENGTH value in the HDL.	0-63	2

HDL Parameters

Table 2 lists the parameters that appear in the HDL.

Table 2: LatticeMico32 SPI HDL Parameters

Parameter Name	Description	Allowable Values
BASE_ADDR	Specifies the base address for the device.	0X80000000 – 0XFFFFFF80
DATA_LENGTH	Specifies the number of serial data bits.	1-32 (master) 1-32 (slave)
SHIFT_DIRECTION	Specifies whether the most significant bit or the least significant bit is first. ◆ If 0, the most significant bit is first. ◆ If 1, the least significant bit is first.	0|1 (master/slave)
CLOCK_PHASE	Specifies the clock phase of the SPI instance: ◆ If 0, data is latched on the leading edge of SCLK, and the data changes on the trailing edge. ◆ If 1, data is latched on the trailing edge of SCLK, and the data changes on the leading edge.	0|1 (master/slave)
CLOCK_POLARITY	Specifies the polarity of the SPI instance: ◆ If 0, the idle state for SCLK is low. ◆ If 1, the idle state for SCLK is high.	0|1 (master/slave)
MASTER	Specifes whether the SPI instance is a master or a slave. A value of 1 defines the LatticeMico32 SPI as a master; otherwise, it acts as a slave.	0|1

Table 2: LatticeMico32 SPI HDL Parameters (Continued)

Parameter Name	Description	Allowable Values
SLAVE_NUMBER	Specifies the number of slave devices supported.	1-32
CLOCK_SEL	Specifies the factor for deriving SCLK from the component input clock (processor clock, CLK_I). SCLK is derived from the following equation: $SCLK = \frac{CLK_I}{2 \times (CLOCK_SEL + 1)}$ For example: For $CLOCK_SEL = 0$: $SCLK = \frac{CLK_i}{2}$ For $CLOCK_SEL = 1$: $SCLK = \frac{CLK_i}{4}$ **Note**: The SCLK Rate value in the UI is used as the CLOCK_SEL parameter.	CLKCNT_WIDTH-1:0
DELAY_TIME	Specifies the time delay factor before shifting the first bit of data after the SS_N signal is asserted. The start delay time is derived from the following equation: $Delay = \mathrm{DELAY_TIME} \times \left(\frac{SCLK_Period}{2}\right)$ **Note**: The Tx Start Delay value in the UI is used as the DELAY_TIME parameter.	0-63
CLKCNT_WIDTH	Sets the range limit for the clock counter. The width should be enough to meet the number of bits required for the slave clock.	1-32
INTERVAL_LENGTH	Specifies the number of SCLK cycles for which the SS_N signal is held inactive between SPI transmit requests. **Note**: The Tx Interframe Pause parameter in the UI is used as the INTERVAL_LENGTH value.	0-63

I/O Ports

The WISHBONE interface supports only classic cycle transfers, which means that the LatticeMico32 System master CTI_O is fixed at 000. The interface does not support cache line wrap; the LatticeMico32 System master BTE_O is fixed at 00. The slave port does not have RTY_O and ERR_O signals. The RTY_O and ERR_O signals are terminated low. The slave adds an interrupt port (SPI_INT_O) to the master (processor). Only a master processor 32-bit operation is supported; S_SEL_I is not supported.

Table 3 describes the input and output ports of the LatticeMico32 SPI.

User Impact of Initial State

At reset and power up, all registers are cleared to 0 except slave-select and TMT and TRDY bits in the status register.

Table 3: LatticeMico32 SPI I/O Port Descriptions

I/O Port	Active	Direction	Initial State	Description
CLK_I	HIGH	I	X	Input clock signal
RST_I	HIGH	I	X	System reset signal
WISHBONE Slave Interface				
SPI_ADR_I	XX	I	X	Slave address bus
SPI_DAT_I	XX	I	X	Slave data input bus
SPI_WE_I	HIGH	I	X	Slave write enable signal
SPI_CYC_I	HIGH	I	X	Slave cycle signal
SPI_STB_I	HIGH	I	X	Slave strobe signal
SPI_SEL_I	HIGH	I	X	Slave select signal
SPI_CTI_I	HIGH	I	X	Slave cycle-type indicator
SPI_BTE_I	HIGH	I	X	Slave burst type
SPI_LOCK_I	HIGH	I	X	Slave bus locked
SPI_DAT_O	XX	O	0	Slave data output bus
SPI_ACK_O	HIGH	O	0	Slave acknowledge signal
SPI_ERR_O	HIGH	O	0	Slave error
SPI_RTY_O	HIGH	O	0	Slave retry
SPI Interface				
MISO	HIGH	I (master) O (slave)	0	Master input slave output
MOSI	HIGH	O (master) I (slave)	0	Master output slave input
SS_N	XX (master) LOW (slave)	O (master) I (slave)	0	Slave select signal (active low)
SCLK	X Set by polarity parameter	O (master) I (slave)	X Set by polarity parameter	Serial clock
Other Auto-Connected Internal Signals				
SPI_INT_O	HIGH	O	0	Slave interrupt request signal

Register Definitions

The LatticeMico32 SPI includes the registers shown in Table 4.

Table 4: Register Map

Register Name	Offset	31-11	10	9	8	7	6	5	4	3	2-0
Receive data	0x00	RXDATA									
Transmit data	0x04	TXDATA									
Status	0x08	Reserved			E	RRDY	TRDY	TMT	TOE	ROE	Res’rv.
Control	0x0C	Res’rv.	SSO	Res’rv.	IE	IRRDY	ITRDY	Res’rv.	ITOE	IROE	Res’rv.
Slave select	0x10	Slave select									

Table 5 through Table 9 provide details about each register in the LatticeMico32 SPI. Writing any value to the status register clears the ROE, TOE, and E bits.

Table 5: RXDATA Register Bit Definition

Register Name	Bit	Access Mode	Description
RXDATA	DATA_LENGTH-1:0	Read only	Data from serial input
Reserved	31:DATA_LENGTH		

Table 6: TXDATA Register Bit Definition

Register Name	Bit	Access Mode	Description
TXDATA	DATA_LENGTH-1:0	Read/write	Data to serial output
Reserved	31:DATA_LENGTH		

Table 7: Status Register Bit Definition

Register Name	Bit	Access Mode	Description
Reserved	2:0		
ROE	3	Read/write	Receive overrun error. 1 indicates that the RXDATA register received new data before the previous data was read. RXDATA was overwritten with the new data; data has been lost.
TOE	4	Read/write	Transmit overrun error. 1 indicates that the TXDATA register received new data before the previous data was moved to the shift register. The new data was not written.
TMT	5	Read/write	Transmit shift register empty. 1 indicates that the transmit shift register is empty and ready to receive the next data frame. 0 indicates that the register is in the process of shifting a data frame out. The TXDATA register holds new data until TMT = 1.

Table 7: Status Register Bit Definition

Register Name	Bit	Access Mode	Description
TRDY	6	Read/write	Transmitter ready. 1 indicates that the TXDATA register is empty and ready to accept new data. 0 indicates that the register has data that has not yet moved to the shift register. New data is blocked until TRDY = 1.
RRDY	7	Read/write	Receiver ready. 1 indicates that the RXDATA register has data and is ready to be read. 0 indicates that the register is empty. Reading the register clears RRDY.
E	8	Read/write	Error. 1 indicates that either the receive overrun error or the transmit overrun error has occurred. As a programming convenience, the E bit is the logical OR of ROE and TOE.
Reserved	31:9		

Table 8: Control Register Bit Definition

Register Name	Bit	Access Mode	Description
Reserved	2:0		
IROE	3	Read/write	Set to 1 to enable interrupt requests for receive overrun errors.
ITOE	4	Read/write	Set to 1 to enable interrupt requests for transmit overrun errors.
Reserved	5		
ITRDY	6	Read/write	Set to 1 to enable interrupt requests for transmitter ready conditions.
IRRDY	7	Read/write	Set to 1 to enable interrupt requests for receiver ready conditions.
IE	8	Read/write	Set to 1 to enable interrupt requests for receive overrun errors and transmit overrun errors.
Reserved	9		
SSO	10	Read/write	In a master SPI, set to 1 to assert the SS_N outputs according to the mask in the slave select register. SSO holds the slave select signal after it would normally be de-asserted. The SPIs continue to exchange data frames until SSO is cleared.
Reserved	31:11		

Table 9: Slave Select Register Bit Definition

Register Name	Bit	Access Mode	Description
slaveselect	SLAVE_NUMBER-1:0	Read/write	Slave select mask
Reserved	31: SLAVE_NUMBER		

The structure shown in Figure 4 depicts the register map layout for the SPI component. The elements are self-explanatory and are based on the register map shown in Table 4. This structure, which is defined in the MicoSPI.h header file, enables you to directly access the SPI registers, if desired. It is used internally by the device driver for manipulating the SPI.

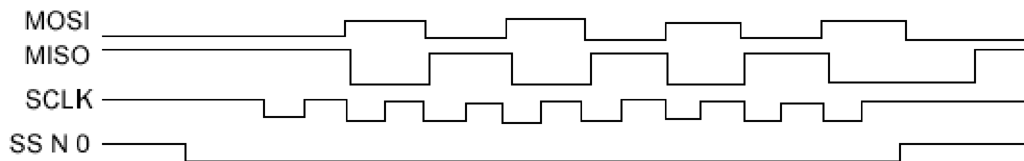
Figure 4: SPI Register Map Structure

```
typedef struct st_MicoSPI{
   volatile unsigned int rx;
   volatile unsigned int tx;
   volatile unsigned int status;
   volatile unsigned int control;
   volatile unsigned int sSelect;
}MicoSPI_t;
```

Timing Diagram

The diagram in Figure 5 shows the timing waveforms for an 8-bit data transfer between an SPI master and an SPI slave.

Figure 5: LatticeMico32 SPI Timing Diagram



EBR Resource Utilization

The LatticeMico32 SPI uses no EBRs.

Software Support

This section describes the software support provided for the LatticeMico32 SPI component. It first describes the LatticeMico32 SPI device driver that directly interacts with a LatticeMico32 SPI instance, then it describes LatticeMico32 SPI services that manage multiple LatticeMico32 SPI instances and any related service. Code examples are provided at the end of this section that show the typical usage of the software.

The support routines are meant for use in a single-threaded environment. If they are used in a multi-tasking environment, you must provide re-entrance protections.

Device Driver

The LatticeMico32 SPI device driver directly interacts with a LatticeMico32 SPI instance. This section describes the type definitions for the SPI device context structure.

This structure, shown in Figure 6, contains SPI component-specific information and is dynamically generated in the DDStructs.h header file. This information is largely filled in by the MSB managed build process, which extracts the SPI component-specific information from the platform definition file. The members should not be manipulated directly, because this structure is for exclusive use by the device driver.

Figure 6: SPI Device Context Structure

```
typedef struct st_MicoSPICtx_t {
const char * name;
unsigned int base;
unsigned int intrLevel;
unsigned int master;
unsigned int slaves;
unsigned int dataLength;
unsigned int shiftDir;
DeviceReg_t lookupReg;
unsigned int control;
void * onRx;
void * onTx;
void * prev;
void * next;
} MicoSPICtx_t;
```

Table 10 describes the parameters of the SPI device context structure shown in Figure 6.

Table 10: LatticeMico32 SPI Context Structure Parameters

Parameter	Data Type	Description
name	const char *	Pointer to the SPI instance name
base	unsigned int	Base address of the SPI instance
intrLevel	unsigned int	The CPU interrupt request to which the SPI instance's interrupt request line is connected
master	unsigned int	Used by the device driver to store information provided by the user
slaves	unsigned int	Used by the device driver to store information provided by the user
dataLength	unsigned int	Width of the SPI instance
shiftDir	unsigned int	Direction of data transfer of the SPI instance is LSB or MSB first

Table 10: LatticeMico32 SPI Context Structure Parameters

Parameter	Data Type	Description
lookupReg	DeviceReg_t	Used by the device driver to register the SPI component instance with the LatticeMico32 lookup service. Refer to the *LatticeMico32 Software Developer User Guide* for a description of the DeviceReg_t data type.
control	unsigned int	Shadow copy of the control register
onRx	void *	Pointer to rrdy interrupt request handler
onTx	void *	Pointer to trdy interrupt request handler
prev	void *	Used by the device driver service to keep track of registered SPI instances
next	void *	Used by the device driver service to keep track of registered SPI instances

Functions

This section describes the implemented device-driver-specific functions.

MicoSPIInit Function

```
void MicoSPIInit(MicoSPICtx_t *ctx)
```

This function initializes a LatticeMico32 SPI instance. It is automatically called as part of the platform initialization for managed builds for each instance of the SPI. It sets the SPI in a known stopped state for future use and registers this SPI instance for the device lookup service.

Table 11 describes the parameter in the MicoSPIInit function syntax.

Table 11: MicoSPIInit Function Parameter

Parameter	Description	Notes
MicoSPICtx_t *	Pointer to a SPI context	For a managed build, the structure referenced is located in DDStructs.c.

MicoSPISetSlaveEnable Function

```
unsigned int MicoSPISetSlaveEnable(MicoSPICtx_t *ctx, unsigned
int mask);
```

This function is used to mask the SS_N output bits of the slave-select register with user-provided data to enable the addressed slaves.

Table 12 describes the parameters in the MicoSPISetSlaveEnable function syntax.

Table 12: MicoSPISetSlaveEnable Function Parameters

Parameter	Description	Notes
MicoSPICtx_t *	Pointer to a SPI context	Pointer to the desired SPI instance's context information
unsigned int	User provided data mask	Used for masking the slave enable outputs

Table 13 shows the values returned by the MicoSPISetSlaveEnable function.

Table 13: Values Returned by MicoSPISetSlaveEnable Function

Return Value	Description
MICOSPI_ERR_SLAVE_DEVICE	Returned if SPI instance is not a master.
0	Success

MicoSPIGetSlaveEnable Function

```
unsigned int MicoSPIGetSlaveEnable(MicoSPICtx_t *ctx, unsigned
int *pMask);
```

This function retrieves the current state of the SS_N output bits.

Table 14 describes the parameters in the MicoSPIGetSlaveEnable function syntax.

Table 14: MicoSPIGetSlaveEnable Function Parameters

Parameter	Description	Notes
MicoSPICtx_t *ctx	Pointer to a SPI context	Pointer to the desired SPI instance's context information
unsigned int *pMask	Pointer to a data storage variable	Used to retrieve the current value of the slave enable outputs

Table 15 shows the values returned by the MicoSPIGetSlaveEnable function.

Table 15: Values Returned by MicoSPIGetSlaveEnable Function

Return Value	Description
MICOSPI_ERR_SLAVE_DEVICE	Returned if SPI instance is not a master
0	Success

MicoSPITxData Function

```
unsigned int MicoSPITxData(MicoSPICtx_t *ctx, unsigned int
data, unsigned int bBlock)
```

This function transmits user-provided data to the addressed slaves. The SPI slave transmits data to the master SPI device as a result of calling this function.

This function writes transmit data to the data-transmit register if the SPI component is ready to accept new data for transmission. The bBlock argument of this function call dictates the behavior if the SPI component is not ready to accept new data for transmission. If bBlock is set to 0—that is, do not block—this function returns without writing data to the data-transmit register and returns a value of MICOSPI_ERR_WOULD_BLOCK. If bBlock is set to a non-zero value—that is, block until completion—this function keeps polling the SPI component's status register until the status indicates that the component is ready to accept new data. This function then writes the data and returns a success status.

Table 16 describes the parameters in the MicoSPITxData function syntax.

Table 16: MicoSPITxData Function Parameters

Parameter	Description	Notes
MicoSPICtx_t *ctx	Pointer to a SPI context	Pointer to the desired SPI instance's context information.
unsigned int data	User provided transmit data	Variable containing the data to be transmitted.
unsigned int bBlock	Block execution flag	Flag to indicate whether to block execution until data is transmitted. If execution is not blocked, loss of transmit data could occur.

Table 17 shows the values returned by the MicoSPITxData function.

Table 17: Values Returned by MicoSPITxData Function

Return Value	Description	Notes
MICOSPI_ERR_WOULD_BLOCK	Returned if SPI instance does not block execution	Indicates a possible loss of transmit data
0	Success	

MicoSPIIsTxDone Function

```
unsigned int MicoSPIIsTxDone(MicoSPICtx_t *ctx)
```

This function checks to see if the shift register is empty.

Table 18 describes the parameter in the MicoSPIIsTXDone function syntax.

Table 18: MicoSPIIsTxDone Function Parameter

Parameter	Description	Note
MicoSPICtx_t *ctx	Pointer to a SPI context	Pointer to the desired SPI instance's context information

Table 19 shows the values returned by the MicoSPIIsTXDone function.

Table 19: Values Returned by MicoSPIIsTxDone Function

Return Value	Description
Non-zero value	Transmission done
0	Transmission pending

MicoSPIRxData Function

```
unsigned int MicoSPIRxData(MicoSPICtx_t *ctx, unsigned int
*pData, unsigned int bBlock)
```

This function retrieves data from the addressed slave ports. It reads data from the receive-data register if the status register indicates that there is data available for reading. The bBlock argument of this function call dictates the behavior if the status register indicates that there is no data available. If bBlock is set to 0—that is, do not block—this function returns immediately with a value of MICOSPI_ERR_WOULD_BLOCK, indicating that there was no data available for reading. If bBlock is set to a non-zero value—that is, block until completion—this function keeps polling the status register until there is data available for reading. The function then reads the data and returns a success status.

Table 20 describes the parameters in the MicoSPIRxData function syntax.

Table 20: MicoSPIRxData Function Parameters

Parameter	Description	Notes
MicoSPICtx_t *ctx	Pointer to a SPI context	Pointer to the desired SPI instance's context information
unsigned int *pData	Pointer to data storage variable	Pointer to a variable for storage of the received data
unsigned int bBlock	Block execution flag	Flag to indicate whether to block execution until data is received

Table 21 shows the values returned by the MicoSPIRxData function.

Table 21: Values Returned by MicoSPIRxData Function

Return Value	Description
MICOSPI_ERR_WOULD_BLOCK	Returned if SPI instance does not block execution
0	Success

MicoSPIEnableTxIntr Function

```
unsigned int MicoSPIEnableTxIntr(MicoSPICtx_t *ctx,
MicoSPIDataHandler_t handler)
```

This function registers the interrupt request handler and enables transmit interrupt requests.

Table 22 describes the parameters in the MicoSPIEnableTxIntr function syntax.

Table 22: MicoSPIEnableTxIntr Function Parameters

Parameter	Description	Notes
MicoSPICtx_t *ctx	Pointer to a SPI context	Pointer to the desired SPI instance's context information
MicoSPIDataHandler_t handler	Structure of transmit data handler	Used to set the address of the transmit data handling interrupt routine

Table 23 shows the values returned by the MicoSPIEnableTxIntr function.

Table 23: Values Returned by MicoSPIEnableTxIntr Function

Return Value	Description
MICOSPI_ERR_INVALID_PARAMETER	Returned if interrupt request handler does not exist
0	Success

MicoSPIDisableTxIntr Function

```
void MicoSPIDisableTxIntr(MicoSPICtx_t *ctx)
```

This function disables transmit interrupt requests.

Table 24 describes the parameter in the MicoSPIDisableTxIntr function syntax.

Table 24: MicoSPIDisableTxIntr Function Parameter

Parameter	Description	Note
MicoSPICtx_t *ctx	Pointer to a SPI context	Pointer to the desired SPI instance's context information

MicoSPIEnableRxIntr Function

```
unsigned int MicoSPIEnableRxIntr(MicoSPICtx_t *ctx,
MicoSPIDataHandler_t handler)
```

This function registers the interrupt request handler and enables receive interrupt requests.

Table 25 describes the parameters in the MicoSPIEnableRxIntr function syntax.

Table 25: MicoSPIEnableRxIntr Function Parameters

Parameter	Description	Notes
MicoSPICtx_t *ctx	Pointer to a SPI context	Pointer to the desired SPI instance's context information
MicoSPIDataHandler_t handler	Structure of receive data handler	Used to set the address of the receive data handling interrupt routine

Table 26 shows the values returned by the MicoSPIEnableRxIntr function.

Table 26: Values Returned by MicoSPIEnableRxIntr Function

Return Value	Description
MICOSPI_ERR_INVALID_PARAMETER	Returned if interrupt request handler does not exist.
0	Success

MicoSPIDisableRxIntr Function

```
void MicoSPIDisableRxIntr(MicoSPICtx_t *ctx)
```

This function disables receive interrupt requests.

Table 27 describes the parameter in the MicoSPIDisableRxIntr function syntax.

Table 27: MicoSPIDisableRxIntr Function Parameters

Parameter	Description	Note
MicoSPICtx_t *ctx	Pointer to a SPI context	Pointer to the desired SPI instance's context information

Services

The SPI device driver registers SPI instances with the LatticeMico32 lookup service, using their instance names for device names and “SPIDevice” as the device type.

For information on the LatticeMico32 lookup service, refer to the *LatticeMico32 Software Developer User Guide.*

Software Usage Examples

This section provides two examples of typical software usage. One demonstrates the polling method, and the other demonstrates the ISR servicing method.

Polling Method

The sample code shown in Figure 7 shows how the LatticeMico32 SPI constantly queries the register in a loop to ascertain whether the register has data to transfer.

Figure 7: Polling Method

```
#include "MicoSPIService.h"

/*
 * Names of the SPI master and slave, as
 * defined in MSB
 */
const char *const SPIM_INSTANCE_NAME = "spim_";
const char *const SPIS_INSTANCE_NAME = "spis_";


int main(void){
    int runs = 0;
    int slave_address = 0x01;
    int slave_txdata = 0x55;
    int master_txdata = ~slave_txdata;
    MicoSPICtx_t *pMaster;
    MicoSPICtx_t *pSlave;

 /* Fetch pointers to master/slave SPI dev-ctx instances */
    pMaster = (MicoSPICtx_t *)MicoGetDevice(SPIM_INSTANCE_NAME);
    pSlave = (MicoSPICtx_t *)MicoGetDevice(SPIS_INSTANCE_NAME);


 /* Make sure pointers are valid */
    if(pMaster == 0){
        printf("Cannot use SPI Master as ctx is unidentified\n");
        return(0);
    }
    if(pSlave == 0){
        printf("Cannot use SPI Slave 0 as ctx is unidentified\n");
        return(0);
    }
```

Figure 7: Polling Method (Continued)

```
 /* MAIN PROGRAM BODY (RUNS INFINITELY) */
   while(1){
       /* Write Slave Data: Block till loaded */
       MicoSPITxData(pSlave,slave_txdata,1);


       /* Enable slave by writing to master */
       MicoSPISetSlaveEnable(pMaster, slave_address);


       /* Check slave enable status. */
       MicoSPIGetSlaveEnable(pMaster, &slave_address);
       if(slave_address != 0x01){
           printf("failed to select internal slave! fatal error\n");
           while(1);
       }


 /* write data targetted for slave */
       MicoSPITxData(pMaster,master_txdata,1);


       /* wait for master's transmission to complete */
       while(1){
           volatile unsigned int iTimeout;
           unsigned int iValue;
           do{
               if(MicoSPIIsTxDone(pMaster)!=0)
                   break;
           }while(1);

/* read data if there's data to read */
           iTimeout = 0;
           do{
               if(MicoSPIRxData(pMaster, &master_txdata, 0) == 0)
                   break;
               if(iTimeout >= 10){
                   pStatus = (volatile unsigned int *)(pMaster->base + 8);
                   iValue = *pStatus;
                   pStatus = (volatile unsigned int *)(pMaster->base + 0);
                   iValue = *pStatus;
                   if(MicoSPIRxData(pSlave, &slave_txdata, 0) != 0){
                       printf("Internal slave did not rx data: hw error\n");
                       while(1);
                   }
                   printf("master failed to rx data within a second %d\n",
                           iTimeout);
                   while(1);
               }
               iTimeout++;
               MicoSleepMilliSecs(100);
           }while(1);
           break;
       }
```

Figure 7: Polling Method (Continued)

```
/* at this point, the slave must also have transmitted data
         * and received data
         */
        if(MicoSPIIsTxDone(pSlave) == 0){
            printf("Internal slave tx not done: hw error\n");
            while(1){};
        }

        if(MicoSPIRxData(pSlave, &slave_txdata, 0) != 0){
            printf("Internal slave did not rx data: hw error\n");
            while(1);
        }


 /* compare data and output results to console */
        if((master_txdata & 0xff) != ((~slave_txdata) & 0xff)){
            printf("data corruption\n");
            printf("master received : 0x%x\n", master_txdata);
            printf("slave received: 0x%x\n", slave_txdata);
            while(1){};
        }
        printf("successfully completed %d iteration\n", ++runs);
    }
}
```

ISR Servicing Method

The example in Figure 8 shows how the LatticeMico32 SPI creates an interrupt request when data is available to be transferred.

Figure 8: ISR Servicing Method

```
#include "MicoSPIService.h"

const char *const SPIM_INSTANCE_NAME = "spim_";
const char *const SPIS_INSTANCE_NAME = "spis_";


/********* Global Variables ***********************/
static int slave_address=0x01;
static int slave_txdata = 0x55;
static int slave_rxdata=0;
static int master_txdata = 0xAA;
static int master_rxdata=0;
static int transfer_complete=0;


// Create Pointers for WB Devices.
MicoSPICtx_t *pMaster;
MicoSPICtx_t *pSlave;
```

Figure 8: ISR Servicing Method (Continued)

```
void OnMasterRx(void){
    /*
     * If you don't want to be interrupted any more, disable
     * the interrupts
     */
    MicoSPIDisableRxIntr(pMaster);
    transfer_complete=1;
    /* read received data here */
    MicoSPIRxData(pMaster, &master_rxdata, 0);
}


void OnMasterTx(void){
    /*
     * If you don't want to be interrupted any more, disable
     * the interrupts
     */
    MicoSPIDisableTxIntr(pMaster);
    transfer_complete=1;


    /*
     * load new tx data or disable tx interrupt; otherwise, this
     * will spin in a loop
     */
     MicoSPITxData(pMaster,master_txdata,1);
     return;
}

 /*************************
 * USER-MAIN ENTRY POINT *
 *************************/
int main(void){
    int runs = 0;


    /* Set Pointers to WB Device Addresses. */
    pMaster = (MicoSPICtx_t *)MicoGetDevice(SPIM_INSTANCE_NAME);
    pSlave = (MicoSPICtx_t *)MicoGetDevice(SPIS_INSTANCE_NAME);


    /* Make sure devices were found and that the pointers aren't null */
    if(pMaster == 0){
        printf("Cannot use SPI Master as ctx is unidentified\n");
        return(0);
    }
    if(pSlave == 0){
        printf("Cannot use SPI Slave as ctx is unidentified\n");
        return(0);
    }
```

Figure 8: ISR Servicing Method (Continued)

```
/* MAIN PROGRAM BODY (REPEATS FOREVER) */
   while(1){

       /* Write Slave Data: Block till loaded */
       MicoSPITxData(pSlave,slave_txdata,1);
       /* Enable slave by writing to master */
       MicoSPISetSlaveEnable(pMaster, slave_address);
       MicoSPIGetSlaveEnable(pMaster, &slave_address);
       if(slave_address != 0x01){
           printf(" failed to select slave!! fatal error\n");
           while(1){};
       }


       /* write first data targeted for slave, block until loaded */
       MicoSPITxData(pMaster,master_txdata,1);
       /* Enable interrupts on master. */
       MicoSPIEnableRxIntr(pMaster, OnMasterRx);
       MicoSPIEnableTxIntr(pMaster, OnMasterTx);


       /* wait for master's transmission complete */
       while(1){
           volatile unsigned int iTimeout;
           unsigned int iValue;
           volatile unsigned int *pStatus;
           pStatus = ((volatile unsigned int *)(pMaster->base + 8));
           iValue = *pStatus;


           /* wait for transmission to complete */
           do{
               /* Check Completion Flag */
               if(transfer_complete!=0)
                   break;
           }while(1);
           transfer_complete=0;
           iTimeout = 0;
```

Figure 8: ISR Servicing Method (Continued)

```
            /* read data if there's data to read */
            do{
                if(MicoSPIRxData(pMaster, &master_rxdata, 0) == 0)
                    break;
                if(iTimeout >= 10){
                    pStatus = ((volatile unsigned int *)(pMaster->base + 8));
                    iValue = *pStatus;
                    if(MicoSPIRxData(pSlave, &slave_rxdata, 0) != 0){
                        printf("Internal slave did not rx data: hw error\n");
                        while(1){};
                    }
                    printf("master failed to rx data within a second %d\n",
                        iTimeout);
                    while(1){};
                }
                iTimeout++;
                MicoSleepMilliSecs(100);
            }while(1);
            break;
        }


        /*
         * at this point, the slave must also have transmitted
         * data and received data
         */
        if(MicoSPIIsTxDone(pSlave) == 0){
            printf("Internal slave tx not done: hw error\n");
            while(1){};
        }
        if(MicoSPIRxData(pSlave, &slave_rxdata, 0) != 0){
            printf("Internal slave did not rx data: hw error\n");
            while(1){};
        }

        /* compare data */
        if((master_rxdata & 0xff) != ((~slave_rxdata) & 0xff)){
            printf("data corruption\n");
            printf("master received : 0x%x\n", master_rxdata);
            printf("slave received: 0x%x\n", slave_rxdata);
            while(1){};
        }
        printf("successfully completed %d iterations\n", ++runs);
        slave_txdata=slave_rxdata;
        master_txdata=master_rxdata;
    }
}
```