Digilent chipKIT SPI Library

Revision: January 18, 2012

1300 NE Henley Court, Suite 3
Pullman, WA 99163
(509) 334 6306 Voice | (509) 334 6300 Fax

Introduction

The Digilent SPI (DSPI) library provides an alternative interface for accessing SPI port to the standard chipKIT SPI library (SPI). The DSPI library provides access to all hardware SPI ports supported by the board in use and provides the ability to perform buffered and interrupt driven transfers.

Serial Peripheral Interface (SPI) is a four wire synchronous serial interface used by many integrated circuits and electronic devices. SPI devices can operate as either master devices or as slave device. The four SPI signals are generally referred to as Slave Select (SS), Master Out, Slave In (MOSI), Master In, Slave Out (MISO), and Serial Clock (SCK). A master device generates SS, MOSI and SCK, and receives MISO. A slave device receives SS, MOSI, and SCK and transmits MISO. The SS signal is used to enable the slave device, and this signal is only significant for slave devices. A master device can use any general purpose I/O pin to generate SS to enable a slave.

The DSPI library only supports operation as an SPI master device.

An SPI transaction begins with the master device bringing SS low. When the slave ses SS go low it becomes enabled and waits for the master to send data. The master shifts data out on MOSI and simultaneously shifts data in on MISO. The slave device receives data from the master on its MOSI pin and simultaneously sends data to the master on its MISO pin. Each time the master sends a byte to the slave, it simultaneously receives a byte from the slave. The master generates the clock signal (SCK) that is used to control the shifting of the data in both the master and the slave.

In general, SPI devices can operate using one of four data transfer modes, and one of two shift directions. The transfer modes specify the idle state of the clock signal (idles high or idles low) and the phase relationship between the active edge of the clock signal and the data. The modes are generally called mode 0 through mode 3. PIC32 microcontrollers support all four transfer modes, but only support shifting the data left (most significant bit first). The DSPI library header file defines symbols used to specify the transfer mode, and default SPI clock rate. Refer to documentation for the SPI slave device being used to determine the transfer mode and shift direction required by the device. Most SPI devices use mode 0 with shift left.

The DSPI library can't be used to access SPI devices that require the data to be shifted to the right (least significant bit first). In this case, the SoftSPI library can be used instead.

The DSPI library defines an object class (DSPI0, DSPI1, etc.) for each hardware SPI port supported by the board in use. These are used to create one or more object instances used to access the SPI ports. The symbol NUM_DSPI_PORTS is defined for each board and can be used to determine the number of DSPI ports available on the board.

To use the DSPI library, an object instance variable of the appropriate DSPIx object class must be created. The object instance is initialized by calling the begin function() and then using other functions to set mode, and clock speed if necessary. Data can then be transferred to a slave device by calling the various data transfer functions.

Defined Interface Symbols

The following symbols are defined in the SoftSPI header file (SoftSPI.h) and can be used with the various configuration functions:

Transfer Mode Values:

- DSPI_MODE0
- DSPI_MODE1
- DSPI_MODE2
- DSPI_MODE3

Default Clock Speed Value:

_DSPI_SPD_DEFAULT - default SPI clock speed, 1Mhz

The SPI port being used is automatically configured for DSPI_MODE0 and with the clock speed set to _DSPI_SPD_DEFAULT when the port is initialized by calling the begin() function.

DSPI Functions

The following describes the various functions defined by the DSPI library for access to hardware SPI ports:

Initialization and Setup Functions

The following functions are used to initialize the SPI port and configure it for operation:

void begin()

Parameters:
None

Return value:
None

This function is used to initialize the SPI port for operation. It sets the transfer mode to DSPI_MODE0 and the clock speed to the default value. The default pin will be used for slave select.

void begin(uint8_t pin)

Parameters:
None

Return value:
None

This function is used to initialize the SPI port for operation. It sets the transfer mode to DSPI_MODE0 and the clock speed to the default value. The specified pin will be used for slave select.

void end()

Parameters:
None

Return value:
None

This function is called when the SPI port is no longer going to be used. It releases the SPI controller and the digital pins so that they can be used for other purposes.

void setSpeed(uint32_t spd)

Parameters:
spd desired SPI clock speed in Hz

Return value:
none

This function is used to set the frequency of the SPI clock. The default value if this function is not called will be _DSPI_SPD_DEFAULT (1Mhz). This clock frequency will be set to the nearest supportable frequency that does not exceed the requested frequency.

void setMode(uint16_t, mod)

Parameters:
mod SPI transfer mode

Return value:
none

This function used to set the SPI transfer mode. The allowed values for mod are: DSPI_MODE0, DSPI_MODE1, DSPI_MODE2, DSPI_MODE3.

void setPinSelect(uint8_t pin)

Parameters:
pin digital pin number of pin to use for slave select

Return value:
none

This is used to specify the digital pin being used for slave select. It is possible for an SPI bus to support multiple slave devices on the bus by using a separate select pin for each slave. In this case, this function can be used to specify which device is being accessed by choosing the select pin currently being used by the DPI object. As part of the initialization for this, all of the pins that are being used for the slave select must be made to be outputs driving high by using the pinMode() and digitalWrite() functions before the begin() function is called.

Data Transfer Functions

The following functions are used to transfer data to and/or from the SPI slave device:

void setSelect(uint8_t sel)

Parameters:

sel	state to set the slave select pin

Return value:
none

This function is used to set the state of the slave select pin. The allowed values for sel are HIGH or LOW. This function is called to set the slave select pin LOW at the beginning of a transfer, and again to set the slave select pin HIGH at the end of a transfer.

uint8_t transfer(uint8_t val)

Parameters:

val	byte to send to the slave device

Return value:
Returns the byte received from the slave device.

This function is used to transfer a single byte to an SPI slave device and simultaneously receive a byte from the slave device.

void transfer(uint16_t cbReq, uint8_t * snd, uint8_t * rcv)

Parameters:

cnt	number of bytes to send/receive
snd	array of bytes to send to the slave
rcv	array to hold bytes received from the slave

Return value:
none

This function is used to send an array of bytes to the slave device and simultaneously receive an array of bytes from the slave device.

void transfer(uint16_t cnt, uint8_t * snd)

Parameters:

cnt	number of bytes to send/receive
snd	array of bytes to send to the slave

Return value:
none

This function is used to send an array of bytes to the slave. Any bytes received from the slave device are ignored.

void transfer(uint16_t cnt, uint8_t pad, uint8_t * rcv)

Parameters:

cnt	number of bytes to send/receive
pad	byte sent repeatedly to slave
rcv	array to hold bytes received from the slave

Return value:
none

This function is used to read an array of bytes from the slave device. The value of pad will be sent repeatedly to cause the bytes to be sent by the slave.

Interrupt Control and Interrupt Data Transfer Functions

The following functions are used to manage transferring data to and/or from an SPI slave device in the background using interrupts. The enableInterruptTransfer() function must be called before any of the interrupt driven data transfer functions are called. The disableInterruptTransfer() function should be called when interrupt driven transfers are no longer going to be performed.

An interrupt driven transfer function will schedule the transfer to occur in the background and then return immediately. The data transfer will not have been completed when the function returns. The transCount() function can be used to determine the number of bytes remaining to be transferred. The transfer is complete when transCount() returns 0.

An interrupt driven transfer can be cancelled before it has completed by calling the cancelIntTransfer() function.

Once an interrupt driven transfer has begun, no other transfers on that SPI port can be performed until it has completed or been cancelled.

The buffers involved in an interrupt driven transfer must remain valid and must not be modified once a transfer has begun until it has completed or been cancelled. In particular, the buffers involved must not be automatic local variables within a function unless control will remain in that function until the

transfer is completed. Automatic local variables within a function are stored on the process stack, and the memory is freed when the function returns. In this case, the DSPI interrupt transfer routines will be reading and writing to undefined memory resulting in unpredictable operation and generally causing the system to crash. In general only static buffers should be used for interrupt transfers.

void enableInterruptTransfer()

Parameters:
None

Return value:
None

This function is used to initialize the SPI port for interrupt operation. It must be called before any interrupt of the interrupt transfer functions can be called.

void disableInterruptTransfer()

Parameters:
None

Return value:
None

This function is used to disable interrupt operation for the SPI port. It can be called when interrupt transfers are no longer going to be used. After calling this function, the interrupt transfer functions cannot be called unless enableInterruptTransfer() has been called again.

void intTransfer(uint16_t cnt, uint8_t * snd, uint8_t rcv)

Parameters:

cnt	number of bytes to send/receive
snd	array of bytes to send to the slave
rcv	array to hold bytes received from the slave

Return value:
none

This function is used to send an array of bytes to the slave device and simultaneously receive an array of bytes from the slave device. This function returns immediately after being called. The data transfer occurs in the background. The snd and rcv buffers should not be modified until the transfer is completed.

void intTransfer(uint16_t cnt, uint8_t * snd)

Parameters:

cnt	number of bytes to send/receive

snd array of bytes to send to the slave

Return value:
none

This function is used to send an array of bytes to the slave. Any bytes received from the slave device are ignored. This function returns immediately after being called. The data transfer occurs in the background. The snd buffer should not be modified until the transfer has completed.

void intTransfer(uint16_t cnt, uint8_t pad, uint8_t rcv)

Parameters:
cnt number of bytes to send/receive
pad byte sent repeatedly to slave
rcv array to hold bytes received from the slave

Return value:
none

This function is used to read an array of bytes from the slave device. The value of pad will be sent repeatedly to cause the bytes to be sent by the slave. The data transfer occurs in the background. The rcv buffer should not be modified until the transfer has completed.

void cancelIntTransfer()

Parameters:
none

Return value:
none

This function can be used to cancel an interrupt transfer before it has completed.

uint16_t transCount()

Parameters:
none

Return value:
Returns the number of bytes remaining to be transferred.

This function can be used to determine if an interrupt transfer is in progress. It returns the number of bytes remaining to be transferred, or 0 if no interrupt transfer is active.

int isOverflow()

Parameters:

none

Return value:
This returns a non-zero value if an overflow error has occurred.

This function will return whether an overflow error has occurred or not. An overflow error occurs when a byte received from a slave has not been read before the next byte arrives.

void clearOverflow()

Parameters:
none

Return value:
none

This function is used to clear the overflow error flag.