SPI Library

A software defined, industry-standard, SPI (serial peripheral interface) component that allows you to control an SPI bus via the xCORE GPIO hardware-response ports. SPI is a four-wire hardware bi-directional serial interface. This component is controlled via C using the XMOS multicore extensions and can either act as SPI master or slave.

The component can be used by multiple tasks within the xCORE device and (each addressing the same or different slaves) and is compatible with other slave devices on the same bus.

Features

- SPI master and SPI slave modes.
- Supports speed of ?? Mbit.
- Multiple slave device support
- All clock polarity and phase configurations supported.

Components

- SPI master (mode 0,1,2,3)
- SPI master, multiple slaves (mode 0,1,2,3)
- SPI slave (mode 0,1,2,3)

Resource Usage

TODO

Software version and dependencies

This document pertains to version 2.0.0 of the SPI library. It is intended to be used with version 13.x of the xTIMEcomposer studio tools.

The library does not have any dependencies (i.e. it does not rely on any other libraries).

Related application notes

The following application notes use this library:

- AN00052 - How to use the SPI master component
- AN00057 - How to use the SPI slave component

1 Hardware characteristics

The SPI protocol requires two wires to be connected to the xCORE device: a clock line and data line as shown in Figure **??**.

SCLK	Clock line, driven by the master
MOSI	Master Output, Slave Input data line, driven by the master
MISO	Master Output, Slave Input data line, driven by the slave
SS	Slave select line, driven by the master

Table 1: SPI data wires

TODO:

- diagrams showing connections of ports to lines (including optional tying down of slave select for single slave)
- mode descriptions (polarity and phase) with timing diags
- description of connecting multiple devices to the same bus

2 Usage

2.1 SPI master synchronous operation

There are two types of interface for SPI master components: synchronous and asynchronous.

The synchronous API provides blocking operation. Whenever a client makes a read or write call the operation will complete before the client can move on - this will occupy the core that the client code is running on until the end of the operation. This method is easy to use, has low resource use and is very suitable for applications such as setup and configuration of attached peripherals.

SPI master components are instantiated as parallel tasks that run in a par statement. For synchronous operation, the application can connect via an interface connection using the spi_master_if interface type:

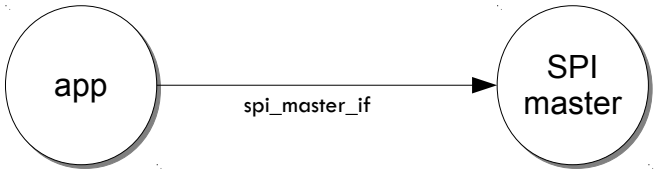

Figure 1: SPI master task diagram

For example, the following code instantiates an SPI master component and connect to it:

```
port p_miso    = XS1_PORT_1A;
port p_ss[1]   = XS1_PORT_1B;
port p_sclk    = XS1_PORT_1C;
port p_mosi    = XS1_PORT_1D;
clock clk_spi  = XS1_CLKBLK_1B;

int main(void) {
  spi_master_if i_spi[1];
  par {
    spi_master(i_spi, 1, p_sclk, p_mosi, p_miso, p_ss, 1, clk_spi);
    my_application(i_spi[0]);
  }
  return 0;
}
```

Note that the connection is an array of interfaces, so several tasks can connect to the same component instance. The slave select ports are also an array since the same SPI data lines can connect to several devices via different slave lines.

The application can use the client end of the interface connection to perform SPI bus operations e.g.:

```
void my_application(client spi_master_if spi) {
  uint8_t val;
  printf("Doing one byte transfer. Sending 0x22.\n");
  spi.begin_transaction(0, 100, SPI_MODE_0);
  val = spi.transfer8(0x22);
  spi.end_transaction()
  printf("Read data %d from the bus.\n", val);
}
```

Here, begin_transaction selects the device 0 and asserts its slave select line. The application can then

transfer data to and from the slave device and finish with end_transaction, which de-asserts the slave select line.

Operations such as spi.transfer8 will block until the operation is completed on the bus. More information on interfaces and tasks can be be found in the XMOS Programming Guide (see XM-004440-PC). By default the SPI synchronous master mode component does not use any logical cores of its own. It is a *distributed* task which means it will perform its function on the logical core of the application task connected to it (provided the application task is on the same tile).

2.2 SPI master asynchronous operation

The synchronous API will block your application until the bus operation is complete. In cases where the application cannot afford to wait for this long the asynchronous API can be used.

The asynchronous API offloads operations to another task. Calls are provide to initiate reads and writes and notifications are provided when the operation completes. This API requires more management in the application but can provide much more efficient operation. It is particularly suitable for applications where the SPI bus is being used for continuous data transfer.

Setting up an asynchronous SPI master component is done in the same manner as the synchronous component:

```
port p_miso    = XS1_PORT_1A;
port p_ss      = XS1_PORT_1B;
port p_sclk    = XS1_PORT_1C;
port p_mosi    = XS1_PORT_1D;
clock clk_spi  = XS1_CLKBLK_1B;

int main(void) {
  i2c_master_async_if i_spi[1];
  par {
    spi_master_async(i_spi, 1, p_sclk, p_mosi, p_miso, p_ss,
                     clk_spi, 100,
    my_application(i_spi[0]);
  }
  return 0;
}
```

The application can use the asynchronous API to offload bus operations to the component. This is done by moving pointers to the SPI slave task to transfer and then retrieving pointers when the operation is complete. For example, the following code repeatedly calculates 100 bytes to send over the bus and handles 100 bytes coming back from the slave:

```
void my_application(client spi_master_async_if spi) {
  uint8_t outdata[100];
  uint8_t indata[100];
  uint8_t * movable buf_in = indata;
  uint8_t * movable buf_out = outdata;

  // create and send initial data
  fill_buffer_with_data(outdata);
  spi.begin_transaction(0, 100, SPI_MODE_0);
  spi.init_transfer_array_8(move(buf_in), move(buf_out), 100);
  while (1) {
    select {
      case spi.operation_complete():
        retreive_transfer_buffers_8(buf_in, buf_out);
        spi.end_transaction();

        // Handle the data that has come in
        handle_incoming_data(buf_in);
        // Calculate the next set of data to go
        fill_buffer_with_data(buf_out);

        spi.begin_transaction(0, 100, SPI_MODE_0);
        spi.init_transfer_array_8(move(buf_in), move(buf_out));
        break;
    }
  }
}
```

The SPI asynchronous task is combinable so can be run on a logical core with other tasks (including the application task it is connected to).

2.3 Slave usage

SPI slave components are instantiated as parallel tasks that run in a par statement. The application can connect via an interface connection.

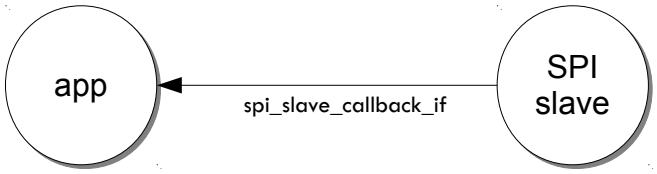

Figure 2: SPI slave task diagram

For example, the following code instantiates an SPI slave component and connect to it:

```
port p_miso    = XS1_PORT_1A;
port p_ss      = XS1_PORT_1B;
port p_sclk    = XS1_PORT_1C;
port p_mosi    = XS1_PORT_1D;
clock clk_spi  = XS1_CLKBLK_1B;

int main(void) {
  spi_slave_if spi;
  par {
    spi_slave(i_spi, p_sclk, p_mosi, p_miso, p_ss, clk, SPI_MODE_0,
              SPI_TRANSFER_SIZE_8);
    my_application(i2c);
  }
  return 0;
}
```

When a slave component is instantiated the mode and transfer size needs to be specified.

The slave component acts as the client of the interface connection. This means it can "callback" to the application to respond to requests from the bus master. For example, the following code snippet shows part of an application that responds to SPI transactions where the first word is a command to read or write command and subsequent transfers either provide or consume data:

```
while (1) {
  uint32_t command = 0;
  size_t index = 0;
  select {
    case spi.master_starts_transaction():
       // The master has asserted slave select to start a transaction.
       command = 0;
       index = 0;
       break;
    case spi.master_supplied_data32(uint32_t data):
       if (command == 0) {
         command = data;
       } else if (command == WRITE_COMMAND) {
         handle_write_data_item(data, index);
         index++;
       }
       break;
    case spi.master_requires_data32() -> uint32_t data:
       if (command == 0) {
         // Not got the command yet.
         data = 0;
       } else if (command == READ_COMMAND) {
         data = get_read_data_item(index);
         index++;
       } else {
         data = 0;
       }
       break;
    case spi.master_requires_data() -> uint8_t data:
       // We are working in 32-bit mode so nothing to do here.
       break;
    case spi.master_supplied_data(uint8_t data):
       // We are working in 32-bit mode so nothing to do here.
       break;
    case spi.master_ends_transaction():
       // The master has de-asserted slave select.
       break;
   }
}
```

Note that the time taken to handle the callbacks will determine the timing requirements of the SPI slave. See application note AN????? for more details on different ways of working with the SPI slave component.

3 Master API

All SPI master functions can be accessed via the `spi.h` header:

```
#include <spi.h>
```

You will also have to add `lib_spi` to the `USED_MODULES` field of your application Makefile.

3.1 Supporting types

The following type is used to configure the SPI components.

Type	**spi_mode_t**
Description	This type indicates what mode an SPI component should use.

Values	`SPI_MODE_0`
	SPI Mode 0 - Polarity = 0, Clock Edge = 1.
	
	`SPI_MODE_1`
	SPI Mode 1 - Polarity = 0, Clock Edge = 0.
	
	`SPI_MODE_2`
	SPI Mode 2 - Polarity = 1, Clock Edge = 0.
	
	`SPI_MODE_3`
	SPI Mode 3 - Polarity = 1, Clock Edge = 1.

3.2 Creating an SPI master instance

Function	spi_master
Description	Task that implements the SPI proctocol in master mode that is connected to a multiple slaves on the bus. Each slave must be connected to using the same SPI mode. You can access different slave devices over the interface connection using the device_index parameter of the interface functions. The task will allocate the device indices in the order of the supplied array of slave select ports.

Type	`[[distributable]]`
	`void`
	`spi_master(server interface spi_master_if i[num_clients],`
	`  size_t num_clients,`
	`  out port sclk,`
	`  out port mosi,`
	`  in port miso,`
	`  out port p_ss[num_slaves],`
	`  size_t num_slaves,`
	`  clock clk)`

Parameters	`i` an array of interface connection to the clients of the task. `num_clients` the number of clients connected to the task. `clk` a clock block used by the task. `sclk` the SPI clock port. `mosi` the SPI MOSI (master out, slave in) port. `miso` the SPI MISO (master in, slave out) port. `p_ss` an array of ports connected to the slave select signals of the slave. `num_slaves` The number of slave devices on the bus. `clk` a clock for the component to use.

Function	spi_master_async
Description	SPI master component for asynchronous API. This component implements SPI and allows a client to connect using the asynchronous SPI master interface.

Type	`[[combinable]]`
	`void`
	`spi_master_async(`
	`  server interface spi_master_async_if i[num_clients],`
	`  size_t num_clients,`
	`  out port sclk,`
	`  out port mosi,`
	`  in port miso,`
	`  out port p_ss[num_slaves],`
	`  size_t num_slaves,`
	`  clock clk)`

Parameters	`i` an array of interface connection to the clients of the task. `num_clients` the number of clients connected to the task. `clk` a clock block used by the task. `sclk` the SPI clock port. `mosi` the SPI MOSI (master out, slave in) port. `miso` the SPI MISO (master in, slave out) port. `p_ss` an array of ports connected to the slave select signals of the slave. `num_slaves` The number of slave devices on the bus. `clk` a clock for the component to use.

3.3 SPI master interface

Type	spi_master_if
Description	This interface allows clients to interact with SPI master task.
Functions	

Function	begin_transaction
Description	Begin a transaction. This will start a transaction on the bus. During a transaction, no other client to the SPI component can send or receive data. If another client is currently using the component then this call will block until the bus is released.

Type	`[[guarded]]`
	`void`
	`begin_transaction(unsigned device_index,`
	`unsigned speed_in_khz,`
	`spi_mode_t mode)`

Parameters	`device_index` the index of the slave device to interact with. `speed_in_khz` The speed that the SPI bus should run at during the transaction (in kHZ). `mode` The mode of spi transfers during this transaction.

Function	end_transaction
Description	End a transaction. This ends a transaction on the bus and releases the component to other clients.
Type	`void end_transaction(void)`

Continued on next page

Type	spi_master_if (continued)

Function	transfer8
Description	Transfer a byte over the spi bus. This function will transmit and receive 8 bits of data over the SPI bus. The data will be transmitted least-significant bit first.
Type	`uint8_t transfer8(uint8_t data)`
Parameters	`data` the data to transmit the MOSI port.
Returns	the data read in from the MISO port.

Function	transfer32
Description	Transfer a 32-bit word over the spi bus. This function will transmit and receive 32 bits of data over the SPI bus. The data will be transmitted least-significant bit first.
Type	`uint32_t transfer32(uint32_t data)`
Parameters	`data` the data to transmit the MOSI port.
Returns	the data read in from the MISO port.

3.4 SPI master asynchronous interface

Type	spi_master_async_if
Description	Asynchronous interface to an SPI component. This interface allows programs to offload SPI bus transfers to another task. An asynchronous notification occurs when the transfer is complete.
Functions	

Function	begin_transaction
Description	Begin a transaction. This will start a transaction on the bus. During a transaction, no other client to the SPI component can send or receive data. If another client is currently using the component then this call will block until the bus is released.

Type	`[[guarded]]`
	`void`
	`begin_transaction(unsigned device_index,`
	`unsigned speed_in_khz,`
	`spi_mode_t mode)`

Parameters	`device_index`
	the index of the slave device to interact with.
	
	`speed_in_khz`
	The speed that the SPI bus should run at during the transaction (in kHZ)
	
	`mode` The mode of spi transfers during this transaction

Function	end_transaction
Description	End a transaction. This ends a transaction on the bus and releases the component to other clients.
Type	`void end_transaction(void)`

Continued on next page

Type	spi_master_async_if (continued)

Function	init_transfer_array_8	
Description	Initialize Transfer an array of bytes over the spi bus. This function will initialize a transmit of 8 bit data over the SPI bus.	

Type	`void`
	`init_transfer_array_8(uint8_t *movable inbuf,`
	`uint8_t *movable outbuf,`
	`size_t nbytes)`	

Parameters	`inbuf`	A *movable* pointer that is moved to the other task pointing to the buffer area to fill with data. If this parameter is NULL then the incoming data of the transfer will be discarded.
	`outbuf`	A *movable* pointer that is moved to the other task pointing to the buffer area to with data to transmit. If this parameter is NULL then the outgoing data of the transfer will consist of undefined values.
	`nbytes`	The number of bytes to transfer over the bus.

Function	init_transfer_array_32	
Description	Initialize Transfer an array of bytes over the spi bus. This function will initialize a transmit of 32 bit data over the SPI bus.	

Type	`void`
	`init_transfer_array_32(uint8_t *movable inbuf,`
	`uint8_t *movable outbuf,`
	`size_t nwords)`	

Parameters	`inbuf`	A *movable* pointer that is moved to the other task pointing to the buffer area to fill with data. If this parameter is NULL then the incoming data of the transfer will be discarded.
	`outbuf`	A *movable* pointer that is moved to the other task pointing to the buffer area to with data to transmit. If this parameter is NULL then the outgoing data of the transfer will consist of undefined values.
	`nwords`	The number of words to transfer over the bus.

Continued on next page

Type	spi_master_async_if (continued)

Function	operation_complete
Description	Transfer completed notification. This notification occurs when a transfer is completed.
Type	`[[notification]]` `slave void operation_complete(void)`

Function	retrieve_transfer_buffers_8	
Description	Retrieve transfer buffers. This function should be called after the `transfer_complete()` notification and will return the buffers given to the other task by `init_transfer_array_8()`.	

Type	`[[clears_notification]]`
	`void`
	`retrieve_transfer_buffers_8(uint8_t *movable &inbuf,`
	`  uint8_t *movable &outbuf)`	

Parameters	`inbuf`	A movable pointer that will be set to the buffer pointer that was filled during the transfer.
	`outbuf`	A movable pointer that will be set to the buffer pointer that was transmitted during the transfer.

Function	retrieve_transfer_buffers_32	
Description	Retrieve transfer buffers. This function should be called after the `transfer_complete()` notification and will return the buffers given to the other task by `init_transfer_array_32()`.	

Type	`[[clears_notification]]`
	`void`
	`retrieve_transfer_buffers_32(uint8_t *movable &inbuf,`
	`  uint8_t *movable &outbuf)`	

Parameters	`inbuf`	A movable pointer that will be set to the buffer pointer that was filled during the transfer.
	`outbuf`	A movable pointer that will be set to the buffer pointer that was transmitted during the transfer.

4 Slave API

All SPI slave functions can be accessed via the `spi.h` header:

```
#include <spi.h>
```

You will also have to add `lib_spi` to the `USED_MODULES` field of your application Makefile.

4.1 Creating an SPI slave instance

Function	**spi_slave**
Description	SPI slave component. This function implements an SPI slave bus.

Type	`[[combinable]]`
	`void`
	`spi_slave(client spi_slave_callback_if i,`
	`in port p_sclk,`
	`in port p_mosi,`
	`out port p_miso,`
	`in port p_ss,`
	`clock clk,`
	`spi_mode_t mode,`
	`spi_transfer_type_t transfer_type)`

Parameters	`i` The interface to connect to the user of the component. The component acts as the client and will make callbacks to the application. `p_sclk` the SPI clock port. `p_mosi` the SPI MOSI (master out, slave in) port. `p_miso` the SPI MISO (master in, slave out) port. `p_ss` the SPI SS (slave select) port. `clk` clock to be used by the component. `mode` the SPI mode of the bus. `transfer_type` the type of transfer the slave will perform: either SPI_TRANSFER_SIZE_8 or SPI_TRANSFER_SIZE_32.

Type	spi_transfer_type_t
Description	This type specifies the transfer size from the SPI slave component to the application.
Values	SPI_TRANSFER_SIZE_8 Transfers should by 8-bit. SPI_TRANSFER_SIZE_32 Transfers should be 32-bit.

4.2 The SPI slave interface API

Type	spi_slave_callback_if
Description	This interface allows clients to interact with SPI slave tasks by completing callbacks that show how to handle data.
Functions	

Function	master_starts_transaction
Description	This callback will get called when the master asserts on the slave select line to start a transaction.
Type	`void master_starts_transaction(void)`

Function	master_ends_transaction
Description	This callback will get called when the master de-asserts on teh slave select line to end a transaction.
Type	`vois master_ends_transaction(void)`

Function	master_requires_data
Description	This callback will get called when the master initiates a bus transfer or when more data is required during a transaction. The application must supply the data to transmit to the master. If the spi slave component is set to SPI_TRANSFER_SIZE_32 mode then this callback will not be called and `master_requires_data32()` will be called instead. Data is transmitted for the least significant bit first. If the master completes the transaction before 8 bits are transferred the remaining bits are discarded.
Type	`uint8_t master_requires_data(void)`
Returns	the 8-bit value to transmit.

Continued on next page

Type	spi_slave_callback_if (continued)

Function	master_requires_data32
Description	This callback will get called when the master initiates a bus transfer. The application must supply the data to transmit to the master. Data is transmitted for the least significant bit first. If the master completes the transaction before 32 bits are transferred the remaining bits are discarded.
Type	`uint32_t master_requires_data32(size_t &num_bytes)`
Returns	the 32-bit value to transmit.

Function	master_supplied_data
Description	This callback will get called after a transfer. It will occur after every 8 bits transferred if the slave component is set to SPI_TRANSFER_SIZE_8. If the component is set to SPI_TRANSFER_SIZE_32 then it will occur if the master ends the transaction before 32 bits are transferred.
Type	`void master_supplied_data(uint8_t datum)`
Parameters	`datum` the data received from the master.

Function	master_supplied_data32
Description	This callback will get called after a transfer. It will only occur if the slave component is set to SPI_TRANSFER_SIZE_32 mode and will occur after every 32 bits received.
Type	`void master_supplied_data32(uint32_t datum)`
Parameters	`datum` the data received from the master.