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.

The SPI bus 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 up to 100 Mbit.
- Multiple slave device support
- All clock polarity and phase configurations supported.

Typical Resource Usage

This following table shows typical resource usage in some different configurations. Exact resource usage will depend on the particular use of the library by the application.

Configuration	Pins	Ports	Clocks	Ram	Logical cores
Master (synchronous, zero clock blocks)	4	4 (1-bit)	0	~1.3K	0
Master (synchronous, one clock block)	4	4 (1-bit)	1	~2.7K	0
Master (asynchronous)	4	4 (1-bit)	2	~3.3K	≤ 1
Slave (32 bit transfer mode)	4	4 (1-bit)	1	~0.8K	≤ 1
Slave (8 bit transfer mode)	4	4 (1-bit)	1	~0.8K	≤ 1

The number of pins is reduced if either of the data lines are not required.

Software version and dependencies

This document pertains to version 3.0.2 of this library. It is known to work on version 14.1.1 of the xTIMEcomposer tools suite, it may work on other versions.

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:

- AN00160 - How to communicate as SPI master
- AN00161 - How to communicate as SPI slave

1 External signal description

The SPI protocol requires a clock, one or more slave selects and either one or two data wires.

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

Table 1: SPI data wires

During any transfer of data, the master will assert the *SS* line and then output a series of transitions on the *SCLK* wire. During this time, the slave will drive data to be sampled by the master and the master will drive data to be sampled by the slave. At the end of the transfer, the *SS* is de-asserted.

If the slave select line is not driven high then the slave should ignore any transitions on the other lines.

1.1 SPI modes

The data sample points for SPI are defined by the clock polarity (CPOL) and clock phase (CPHA) parameters. SPI clock polarity may be inverted or non-inverted by the CPOL and the CPHA parameter is used to shift the sampling phase. The following for sections illustrate the MISO and MOSI data lines relative to the clock. The timings are given by:

t1	The minimum time from the start of the transaction to data being valid on the data pins.
t2	The inter-transmission gap. This is the minimum amount of time that the slave select must be de-asserted.
MAX CLOCK RATE	This is the maximum clock rate supported by the configuration.

Table 2: SPI timings

The setup and hold timings are inherited from the underlying xCORE device. For details on these timing please refer to the device datasheet.

1.1.1 Mode 0 - CPOL: 0 CPHA 1

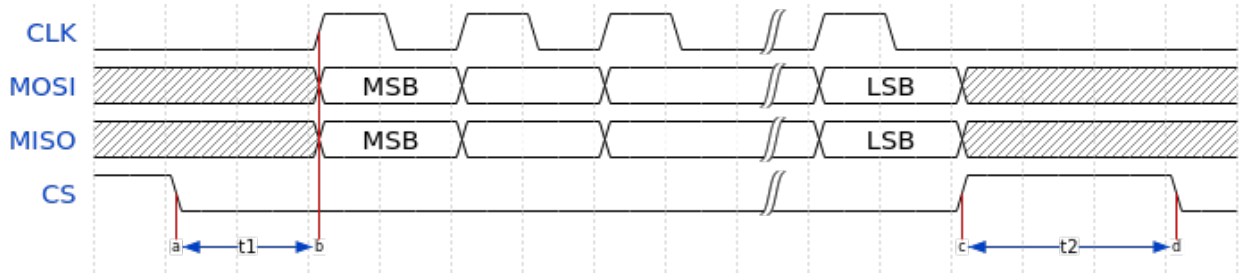

Figure 1: Mode 0

The master and slave will drive out their first data bit on the first rising edge of the clock and sample on the subsequent falling edge.

1.1.2 Mode 1 - CPOL: 0 CPHA 0

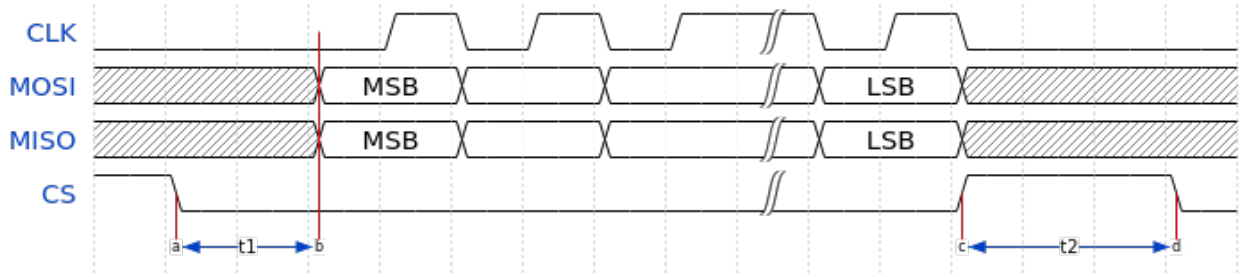

Figure 2: Mode 1

The master and slave will drive out their first data bit before the first rising edge of the clock then drive on subsequent falling edges. They will sample on rising edges.

1.1.3 Mode 2 - CPOL: 1 CPHA 0

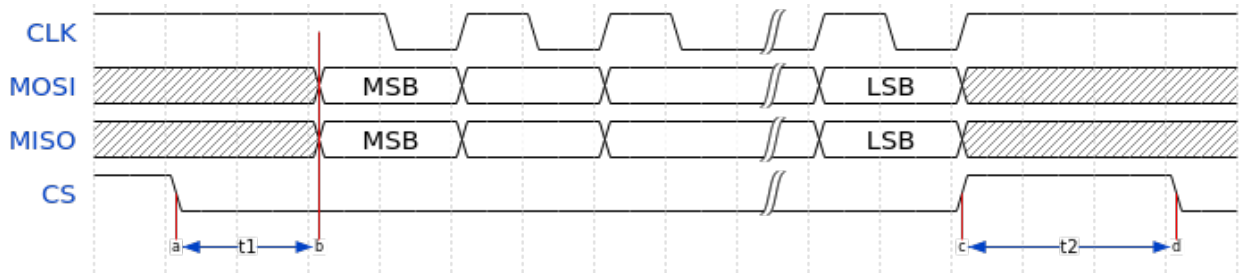

Figure 3: Mode 2

The master and slave will drive out their first data bit before the first falling edge of the clock then drive on subsequent rising edges. They will sample on falling edges.

1.1.4 Mode 3 - CPOL: 1 CPHA 1

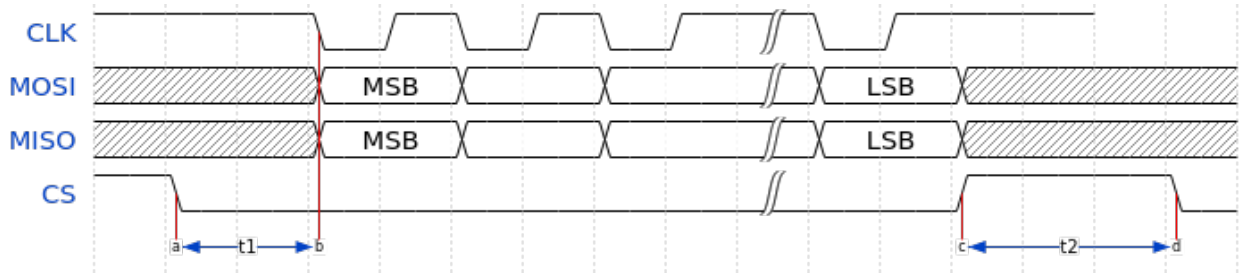

Figure 4: Mode 3

The master and slave will drive out their first data bit on the first falling edge of the clock and sample on the subsequent rising edge.

1.2 SPI master timing characteristics

The application calls functions to begin a transaction (which asserts the slave select line) and to transfer data. So the minimum time between these (*t1*) can be controlled by the application.

The inter-transmission gap (*t2*) is also controlled by the user application since the function to specify the end of the transaction (i.e. the de-assertion of the slave select line) has an argument which is the minimum amount of time before which another transaction can start.

1.2.1 Synchronous SPI master clock speeds

The maximum speed that the SPI bus can be driven depends on whether a clock block is used, the speed of the logical core that the SPI code is running on and where both the *MISO* and *MOSI* lines are used. The timings can be seen in Table 3.

Clock blocks	MISO enabled	MOSI enabled	Max kbps (62.5 MHz core)	Max kbps (125 MHz core)
0	1	0	2497	3366
0	1	1	1765	3366
1	1	0	2149	2149
1	1	1	2149	2149

Table 3: SPI master timings (synchronous)

1.2.2 Asynchronous SPI master clock speeds

The asynchronous SPI master is limited only by the clock divider on the clock block. This means that for the 100MHz reference clock, the asynchronous master can output a clock at up to 100MHz

Clock blocks	MISO enabled	MOSI enabled	Max kbps (62.5 MHz core)	Max kbps (125 MHz core)
1	x	x	100000	100000

Table 4: SPI master timings (asynchronous)

1.3 Connecting to the xCORE SPI master

The SPI wires need to be connected to the xCORE device as shown in Figure 5. The signals can be connected to any one bit ports on the device provide they do not overlap any other used ports and are all on the same tile.

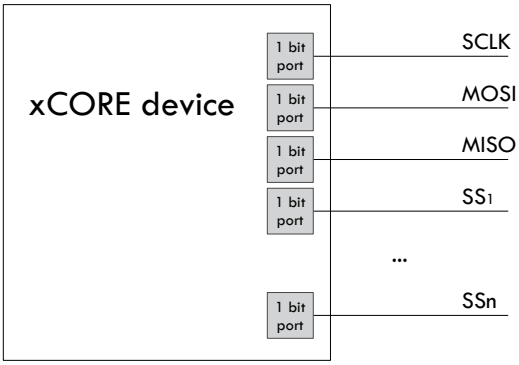

Figure 5: SPI master connection to the xCORE device

If only one data direction is required then the *MOSI* or *MISO* line need not be connected. However, **asynchronous mode is only supported if the MISO line is connected**.

The master component of this library supports multiple slaves on unique slave select wires. This means that a single slave select assertion cannot be used to communicate with multiple slaves at the same time.

1.4 SPI slave timings

The logical core running the SPI slave task will wait for the slave select line to assert and then begin processing the transaction. At this point it will call the `master_requires_data` callback to application code. The time taken for the application to perform this call will affect how long the logical core has to resume processing SPI data. This will affect the minimum allowable time between slave select changing and data transfer from the master (*t1*). The user of the library will need to determine this time based on their application.

After slave select is de-asserted the SPI slave task will call the `master_ends_transaction` callback. The time the application takes to process this will affect the minimum allowable inter-transmission gap between transactions (*t2*). The user of the library will also need to determine this time based on their application.

If the SPI slave task is combined will other tasks running on the same logical core then the other task may process an event delaying the time it takes for the SPI slave task to react to events. This will add these delays to the minimum times for both *t1* and *t2*. The library user will need to take these into account in determining the timing restrictions on the master.

1.5 Connecting to the xCORE SPI slave

The SPI wires need to be connected to the xCORE device as shown in Figure 6. The signals can be connected to any one bit ports on the device provide they do not overlap any other used ports and are all on the same tile.

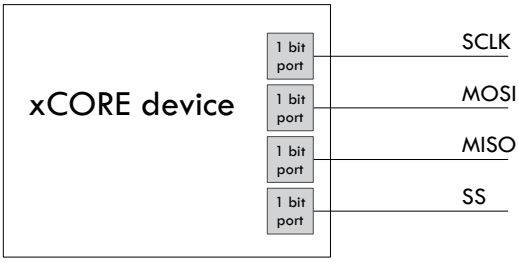

Figure 6: SPI slave connection to the xCORE device

The slave will only send and receive data when the slave select is driven high.

If the *MISO* line is not required then it need not be connected. The *MOSI* line must always be connected.

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:

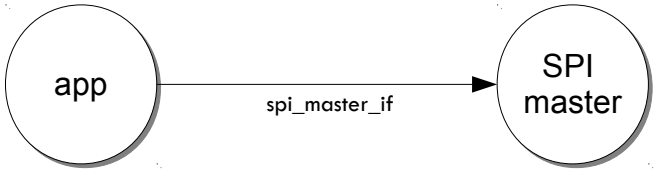

Figure 7: SPI master task diagram

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

```
out buffered port:32 p_miso    = XS1_PORT_1A;
out port p_ss[1]               = {XS1_PORT_1B};
out buffered port:22 p_sclk    = XS1_PORT_1C;
out buffered port:32 p_mosi    = XS1_PORT_1D;
clock clk_spi                  = XS1_CLKBLK_1;

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 final parameter of the `spi_master` task is an optional clock block. If the clock block is supplied then the maximum transfer rate of the SPI bus is increased (see Table 3). If `null` is supplied instead then the performance is less but no clock block is used.

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(1000);
  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).

2.1.1 Synchronous master usage state machine

The function calls made on the SPI master interface must follow the sequence shown by the state machine in Figure 8. If this sequence is not followed then the behavior is undefined.

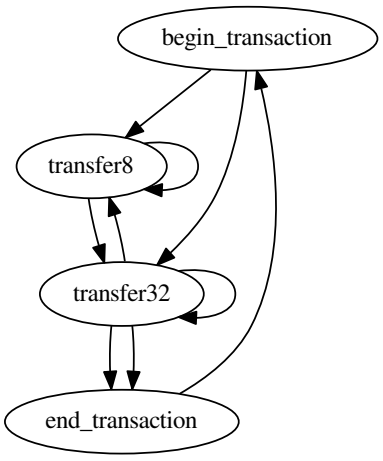

Figure 8: SPI master use state machine

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:

```
out buffered port:32 p_miso    = XS1_PORT_1A;
out port p_ss[1]               = {XS1_PORT_1B};
out buffered port:22 p_sclk    = XS1_PORT_1C;
out buffered port:32 p_mosi    = XS1_PORT_1D;

clock cb0      = XS1_CLKBLK_1;
clock cb1      = XS1_CLKBLK_2;

int main(void) {
  spi_master_async_if i_spi[1];
  par {
    spi_master_async(i_spi, 1, p_sclk, p_mosi, p_miso, p_ss, 1, cb0, cb1);
    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():
        retrieve_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).

2.2.1 Asynchronous master command buffering

In order to provide asynchronous behaviour for multiple clients the asynchronous master will store up to one `begin_transaction` and one `init_transfer_array_8` or `init_transfer_array_32` from each client. This means that if the master is busy doing a transfer for client *X*, then client *Y* will still be able to begin a transaction and send data fully asynchronously. Consequently, after client *Y* has issued `init_transfer_array_8` or `init_transfer_array_32` the it will be able to continue operation whilst waiting for the notification.

2.2.2 Asynchronous master usage state machine

The function calls made on the SPI master asynchronous interface must follow the sequence shown by the state machine in Figure 9. If this sequence is not followed then the behavior is undefined.

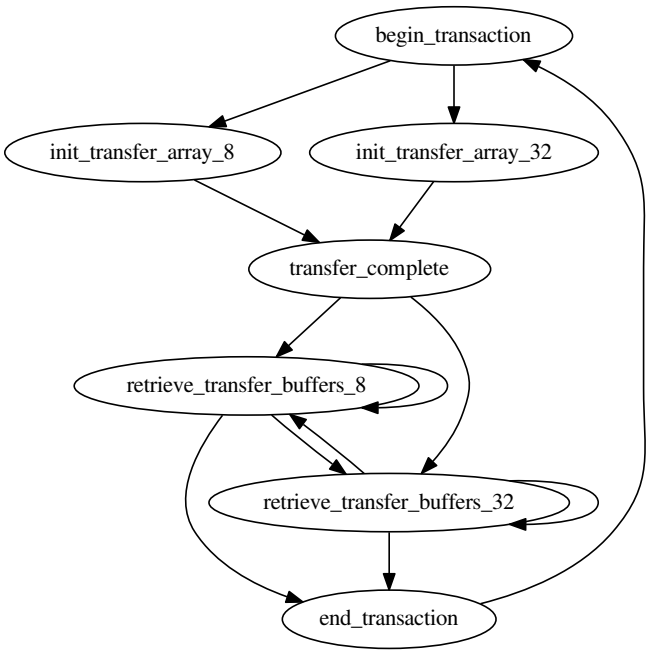

Figure 9: SPI master use state machine (asynchronous)

2.3 Master inter-transaction gap

For both synchronous and asynchronous modes the `end_transaction` requires a slave select deassert time. This parameter will provide a minimum deassert time between two transaction on the same slave select. In the case where a `begin_transaction` asserting the slave select would violate the previous `end_transaction` then the `begin_transaction` will block until the slave select deassert time has been satisfied.

2.4 Slave usage

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

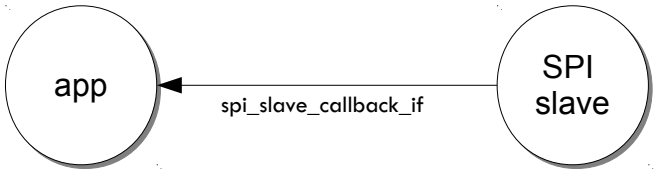

Figure 10: SPI slave task diagram

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

```
out buffered port:32    p_miso = XS1_PORT_1E;
in port                 p_ss = XS1_PORT_1F;
in port                 p_sclk = XS1_PORT_1G;
in buffered port:32     p_mosi = XS1_PORT_1H;
clock                   cb   = XS1_CLKBLK_1;

int main(void) {
  interface spi_slave_callback_if i_spi;
  par {
    spi_slave(i_spi, p_sclk, p_mosi, p_miso, p_ss, cb, SPI_MODE_0,
              SPI_TRANSFER_SIZE_8);
    my_application(i_spi);
  }
  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_requires_data() -> uint32_t data:
       if (command == 0) {
         // Not got the command yet. This will be the
         // first word of the transaction.
         data = 0;
       } else if (command == READ_COMMAND) {
         data = get_read_data_item(index);
         index++;
       } else {
         data = 0;
       }
       break;
    case spi.master_supplied_data(uint32_t data, uint32_t valid_bits):
       if (command == 0) {
         command = data;
       } else if (command == WRITE_COMMAND) {
         handle_write_data_item(data, index);
         index++;
       }
       break;
    case spi.master_ends_transaction():
       // The master has de-asserted slave select.
       command = 0;
       index = 0;
       break;
   }
}
```

Note that the time taken to handle the callbacks will determine the timing requirements of the SPI slave. See application note AN00161 for more details on different ways of working with the SPI slave component.

2.5 Disabling data lines

The *MOSI* and *MISO* parameters of the `spi_master` task are optional. So in the top-level `par` statement the function can be called with `null` instead of a port e.g.:

```
spi_master(i_spi, 1, p_sclk, null, p_miso , p_ss, 1, clk_spi);
```

Similarly, the *MOSI* parameter of the `spi_master_async` task is optional (but the *MISO* port must be provided).

The `spi_slave` task has an optional *MISO* parameter (but the *MOSI* port must be supplied).

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],`
	`  static const size_t num_clients,`
	`  out buffered port:32 sclk,`
	`  out buffered port:32 ?mosi,`
	`  in buffered port:32 ?miso,`
	`  out port p_ss[num_slaves],`
	`  static const 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],`
	`  static const size_t num_clients,`
	`  out buffered port:32 sclk,`
	`  out buffered port:32 ?mosi,`
	`  in buffered port:32 miso,`
	`  out port p_ss[num_slaves],`
	`  static const size_t num_slaves,`
	`  clock clk0,`
	`  clock clk1)`

Parameters	`i` an array of interface connection to the clients of the task. `num_clients` the number of clients connected to 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. `clk0` a clock for the component to use. `clk1` 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(unsigned ss_deassert_time)`

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	`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(unsigned ss_deassert_time)`
Parameters	`ss_deassert_time` The minimum time in reference clock ticks between assertions of the selected slave select. This time will be ignored if the next transaction is to a different slave select.

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(uint32_t *movable inbuf,`
	`uint32_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	transfer_complete
Description	Transfer completed notification. This notification occurs when a transfer is completed.
Type	`[[notification]]` `slave void transfer_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(uint32_t *movable &inbuf,`
	`  uint32_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 spi_i,`
	`  in port p_sclk,`
	`  in buffered port:32 p_mosi,`
	`  out buffered port:32 ?p_miso,`
	`  in port p_ss,`
	`  clock clk,`
	`  static const spi_mode_t mode,`
	`  static const spi_transfer_type_t transfer_type)`

Parameters	`spi_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_ends_transaction
Description	This callback will get called when the master de-asserts on the slave select line to end a transaction.
Type	`void 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	`uint32_t master_requires_data(void)`
Returns	the 8-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(uint32_t datum, uint32_t valid_bits)`
Parameters	`datum` the data received from the master. `valid_bits` the number of valid bits of data received from the master.

APPENDIX A - Known Issues

There are no known issues with this library.

APPENDIX B - SPI library change log

B.1 3.0.2

- Update to source code license and copyright

B.2 3.0.1

- Minor user guide and documentation fixes

B.3 3.0.0

- Consolidated version, major rework from previous SPI components