SpartanMC

Serial Peripheral Interface Bus (SPI)

Table of Contents

List of Figures

List of Tables

Serial Peripheral Interface Bus (SPI)

The SPI is a SpartanMC peripheral device for serial communication using the SPI bus. The SPI enables bit frames to be shifted in and out of the component at programmable speed. The frame width can be changed during runtime so that one single SPI master is able to control multiple different slaves. An SPI master can be connected with up to 15 SPI-slaves which share 3 wires:

- SCLK (*serial clock*)
- MOSI (*master out slave in*)
- MISO (*master in slave out*)

Besides the three shared signals above, there is also one dedicated low-active slave-select-signal for each slave. A slave may use the shared wires, only if it has been selected by the master using this select-signal. The block diagram below shows the brief structure of the SPI master and its interfaces to the slave and SpartanMC side.

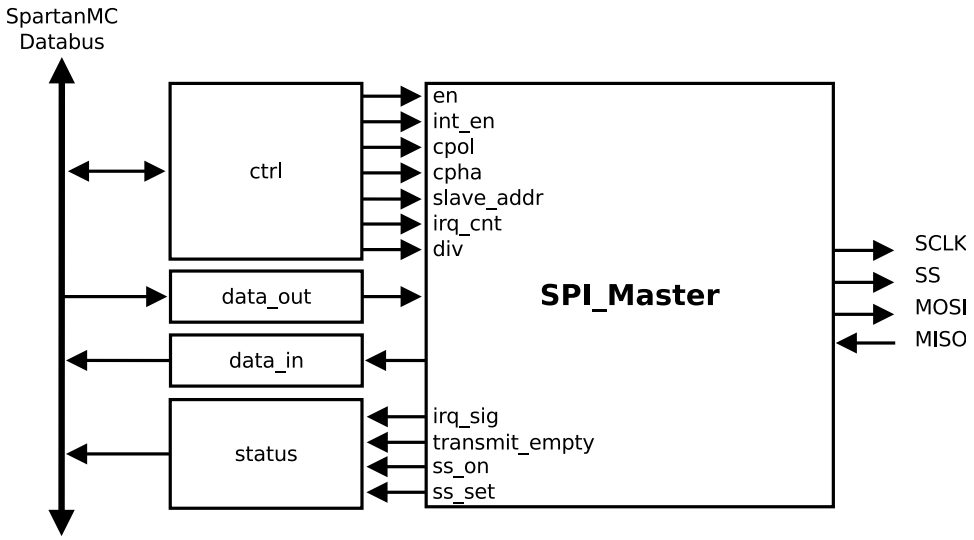

Figure 1: SPI block diagram

As shown in this diagram, the SPI master uses four registers on the SpartanMC side, namely `ctrl` , `data_out` , `data_in` and `status` . The former two are writable and the latter two are read-only.The SPI master can be configured by setting the different fields of the `ctrl` register, like the clock divider, the frame width or the slave address etc. After configuration, the data can be sent to the target slave by writing the `data_out` register and the received data can be read from the `data_in` register. Since an external SPI slave works asynchronously with the SpartanMC, the status of the current transmission should be checked to ensure if it has been finished. This can be done by either polling the `TRANS_EMPTY` flag in the `status` register or using the interrupt controller.

1. Communication

To start a transfer to a slave, the master has to clear the select signal for this slave. After this, the SPI master can generate the clock signal and shift data to the slave. During each SPI clock cycle, one bit is sent to the slave and one bit is received from the slave. The polarity and phase of the clock can be configured by two bits of the control register, namely CPOL and CPHA, in the following way:

If CPOL = 0, the base value of the clock is zero

- if CPHA = 0, the data are read on the rising edge and refreshed on the falling edge
- if CPHA = 1, the data are read on the falling edge and refreshed on the rising edge

If CPOL = 1, the base value of the clock is one

- if CPHA = 0, the data are read on the falling edge and refreshed on the rising edge
- if CPHA = 1, the data are read on the rising edge and refreshed on the falling edge

In other words, the data are always sampled on the first edge of one clock cycle if CPHA = 0, and on the second one if CPHA = 1, regardless of whether the edge is rising or falling. The timing diagram below shows the clock polarity and clock phase according to an example SPI frame of 8 bits.

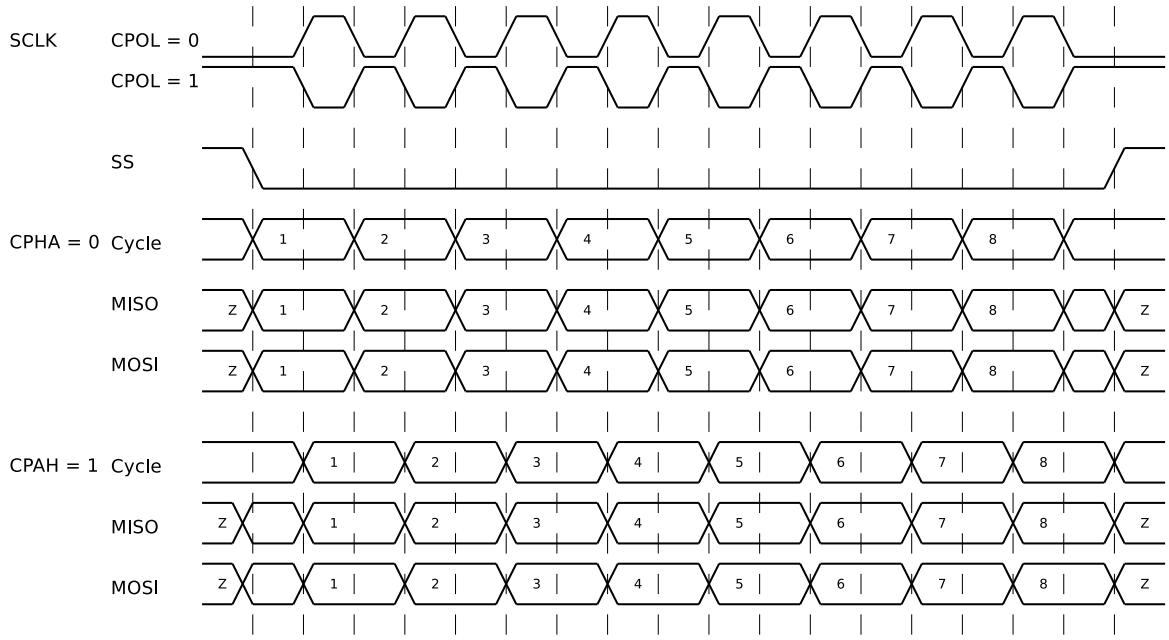

Figure 2: SPI frame

Note: By CPHA = 0, the data must be set up one half clock cycle before the first clock edge.

2. Module parameter

The SPI master uses only one parameter named `SPI_SS` which gives the number of connected slaves. This parameter can be set in JConfig with respect to the SoC system being built.

Table 1: SPI module parameters

Parameter	Default	Descripton
SPI_SS	1	Number of connected slaves

3. Peripheral Registers

3.1. SPI Register Description

As shown in the table below, the SPI peripheral provides four 18-bit registers which are mapped to the SpartanMC address space.

Table 2: SPI registers

Offset	Name	Access	Description
0	spi_control	r/w	Contains the current SPI settings e.g. CPOL, CPHA, clock divider etc.
1	spi_data_out	r/w	Register for outgoing data.
2	spi_data_in	r	Register for incoming data.
3	spi_status	r	Status register.

In the following, the control and status register are described in more details.

3.2. SPI Control Register

The table below gives an overview of the layout of the control register:

Table 3: SPI control register layout

Bit	Name	Access	Default	Description
0	EN	r/w	0	Enable the master.
1	IRQ_EN	r/w	0	Enable the interrupt.

Bit	Name	Access	Default	Description
2	CPOL	r/w	0	Clock polarity.
3	CPHA	r/w	0	Clock phase.
4-7	SLAVE	r/w	0	Address of the selected slave. 0 deactivates all slave-select-signals 1 activates the first slave-select-signal .... 15 activates the 15th slave-select-signal
8-12	BITCNT	r/w	00000	Number of bits contained in one frame. Only the range from 1 to 18 can be used. Other values will be ignored. 1 : 1-bit SPI frame .... 18 : 18-bit SPI frame
13-15	CLK_DIV	r/w	111	Clock divider. SCLK = (clk_peri) / (4 * divider) 0 : 2^0= 1 1 : 2^1= 2 .... 7 : 2^7= 128

Table 3: SPI control register layout

3.3. SPI Status Register

The following table shows the layout of the status register:

Table 4: SPI control register layout

Bit	Name	Access	Default	Description
0	TRANS_EMPTY	r	0	Is set to 1, if the transmission has been finished.
1	IRQ_Sig	r	0	Is set to 1 together with `TRANS_EMPTY` , but will be cleared on a read access to the `data_in` register.
2	SS_ON	r	0	Is set to 1, if the slave address is not equal 0.
3	SS_SET	r	1	Is set to 1, if the slave address has been assigned a new value.

Table 4: SPI control register layout

3.4. SPI C-Header spi.h for Register Description

```
#ifndef __SPI_H
#define __SPI_H

#ifdef __cplusplus
extern "C" {
#endif

#include <peripherals/spi_master.h>
#include <peripherals/spi_slave.h>
#include <bitmagic.h>

//master only functions
void spi_master_activate(spi_master_regs_t *spi,unsigned int
device);
void spi_master_deactivate(spi_master_regs_t *spi);
void spi_master_set_div(spi_master_regs_t *spi, unsigned int
div);

//master duplicate functions
unsigned int spi_master_readwrite(spi_master_regs_t *spi,
unsigned int data);
void spi_master_write(spi_master_regs_t *spi, unsigned int
data);
void spi_master_enable(spi_master_regs_t *spi);
void spi_master_disable(spi_master_regs_t *spi);
void spi_master_enable_irq(spi_master_regs_t *spi);
void spi_master_disable_irq(spi_master_regs_t *spi);
void spi_master_set_cpol(spi_master_regs_t *spi, unsigned int
cpol);
void spi_master_set_cpah(spi_master_regs_t *spi, unsigned int
cpah);
void spi_master_set_bitcnt(spi_master_regs_t *spi, unsigned
int bitcnt);

//slave duplicate functions
unsigned int spi_slave_readwrite(spi_slave_regs_t *spi,
unsigned int data);
void spi_slave_write(spi_slave_regs_t *spi, unsigned int
data);
void spi_slave_enable(spi_slave_regs_t *spi);
void spi_slave_disable(spi_slave_regs_t *spi);
void spi_slave_enable_irq(spi_slave_regs_t *spi);
void spi_slave_disable_irq(spi_slave_regs_t *spi);
```

```
void spi_slave_set_cpol(spi_slave_regs_t *spi, unsigned int
cpol);
void spi_slave_set_cpah(spi_slave_regs_t *spi, unsigned int
cpah);
void spi_slave_set_bitcnt(spi_slave_regs_t *spi, unsigned int
bitcnt);

#ifdef __cplusplus
}
#endif

#endif
```

3.5. SPI C-Header spi_master.h for Register Description

```
#ifndef __SPI_MASTER_H
#define __SPI_MASTER_H

#ifdef __cplusplus
extern "C" {
#endif

#include <peripherals/spi_common.h>

// CONTROL
#define SPI_MASTER_CTRL_EN      SPI_CTRL_EN
#define SPI_MASTER_CTRL_INT_EN   SPI_CTRL_INT_EN
#define SPI_MASTER_CTRL_CPOL      SPI_CTRL_CPOL
#define SPI_MASTER_CTRL_CPHA      SPI_CTRL_CPHA
#define SPI_MASTER_CTRL_SLAVE   0x000F0 // 00 0000 0000 1111
0000
#define SPI_MASTER_CTRL_BITCNT   0x01F00 // 00 0001 1111 0000
0000
#define SPI_MASTER_CTRL_DIV      0x0E000 // 00 1110 0000 0000
0000

//STATUS COMPATIBILITY SECTION FOR OLD PROJECTS
#define SPI_MASTER_STAT_TRANS_EMPTY SPI_STAT_TRANS_EMPTY
#define SPI_MASTER_STAT_INT      SPI_STAT_INT
#define SPI_MASTER_STAT_SS_ON   SPI_STAT_SS_ON
#define SPI_MASTER_STAT_SS_SET   SPI_STAT_SS_SET

typedef struct {
   spi_t spi;
} spi_master_regs_t;
```

```
#ifdef __cplusplus
}
#endif

#endif
```

3.6. SPI C-Header spi_slave.h for Register Description

```
#ifndef __SPI_SLAVE_H
#define __SPI_SLAVE_H

#ifdef __cplusplus
extern "C" {
#endif

#include <peripherals/spi_common.h>

#define SPI_SLAVE_CTRL_EN      SPI_CTRL_EN
#define SPI_SLAVE_CTRL_INT_EN   SPI_CTRL_INT_EN
#define SPI_SLAVE_CTRL_CPOL      SPI_CTRL_CPOL
#define SPI_SLAVE_CTRL_CPHA      SPI_CTRL_CPHA
#define SPI_SLAVE_CTRL_DONE      0x00100 // 00 0000 0001 0000
0000
#define SPI_SLAVE_CTRL_INT      0x00200 // 00 0000 0010 0000
0000
#define SPI_SLAVE_CTRL_BITCNT   0x01C00 // 00 0001 1100 0000
0000

typedef struct {
   spi_t spi;
} spi_slave_regs_t;

#ifdef __cplusplus
}
#endif

#endif
```

3.7. Basic Usage of the SPI Registers

The structures shown above can be used in a program directly, if `<spi.h>` has been included. They serve as the interface between software and hardware. A programmer can configure and control the SPI master simply using these structures without having to care about any low-level details (e.g. timing) at all. According to several trivial examples, this section illustrates how to use this interface to communicate with the SPI

master. First of all, assume that `SPI_MASTER_0` is a pointer which has been assigned the physical address of a SPI master. The registers of the SPI master can be accessed via this pointer.

- **Example 1 : Enable the SPI master**

```
SPI_MASTER_0->spi_control |= SPI_MASTER_CTRL_EN;
```

- **Example 2 : Set the frame width to 16**

```
/* don't forget to clear the default value */
SPI_MASTER_0->spi_control &= ~SPI_MASTER_CTRL_BITCNT;
SPI_MASTER_0->spi_control |= (16<<8);
```

- **Example 3 : Send the constant value 256 to the slave 1**

```
/* activate the slave 1 */
SPI_MASTER_0->spi_control |= (1 << 4);
/* data written into spi_data_out will be sent */
SPI_MASTER_0->spi_data_out = 256;
/* deactivate the slave 1 */
SPI_MASTER_0->spi_control &= ~SPI_MASTER_CTRL_SLAVE;
```

- **Example 4 : Read the received value**

```
int v = SPI_MASTER_0->spi_data_in;
```

- **Example 5 : Check IRQ_Sig of the status register**

```
/* wait until the bit has been set */
while(!(SPI_MASTER_0->spi_status&SPI_MASTER_STAT_INT));

/* handle the interrupt here */
```

4. SPI Sample Application

This sample application reads the Circuit-ID from an M25P32 Flash EPROM via SPI. The application was implemented on an Xilinx ML507 evaluation board.

```
#include <system/peripherals.h>
#include <uart.h>
#include <stdio.h>
#include <spi.h>
#include "m25p32.h"

FILE * stdout = &UART_LIGHT_0_FILE;

void main() {
   unsigned int i;
   printf("\r\nHello SPI_Sample:");
```

```
   printf(\r\nEnable the SPI-Core:");
   SET(SPI_MASTER_0->spi_control, SPI_MASTER_CTRL_EN);

   printf("\r\nPower-Up the connected SPI-Flash:");
   spi_activate(SPI_MASTER_0,1);
   spi_readwrite(SPI_MASTER_0,0xAB);
   spi_deactivate(SPI_MASTER_0);

   printf("\r\nRead ID of the SPI-Flash:\r\n");
   unsigned int id[4];
   m25p32_read_id(SPI_MASTER_0, &id[0]);

   for(i=0;i<3;i++) {
       printf("ID %u : 0x%x\r\n",i,id[i]);
   }
   UNSET(SPI_MASTER_0->spi_control, SPI_MASTER_CTRL_EN);
   while(1);
}

void m25p32_read_id(spi_t* spi, unsigned int* data) {
   unsigned int i;
   spi_activate(spi,1);
   spi_readwrite(spi,M25P32_RDID);
   for (i = 0; i< 3; i++){
       data[i] = spi_readwrite(spi,0);
   }
   spi_deactivate(spi);
}
```

The output is sent to a host PC via serial connection. Therefore a UART peripheral is required in the SoC. ID 0 (0x20) specifies the manufacturer type (ST), ID 1 (0x20) specifies device type and ID 2 (0x16) indicates the memory capacity.

```
SpMC loader v20120927

Hello SPI_Sample:
Enable the SPI-Core:
Power-Up the connected SPI-Flash:
Read ID of the SPI-Flash:
ID 0 : 0x20
ID 1 : 0x20
ID 2 : 0x16
```