SpartanMC

I2C Master

Table of Contents

List of Figures

List of Tables

I2C Master

I2C (also referred to as *two-wire interface*) is a serial bus which allows for connection of multiple master devices to multiple slave devices, only using two single bidirectional lines:

- SCL (*serial clock line*)
- SDA (*serial data line*)

Both lines need to be pulled up with resistors. Because of this, both of them remain simply high, if there is no communication between any master and slave. The clock line needs to be driven by a master. Using this clock, the data will be transmitted bit by bit between the master and the corresponding slave over the data line.

The SpartanMC I2C master controller is a quite simple peripheral device which supports basic I2C functions. The following block diagram gives an overview of its structure and interfaces to both the slave and SpartanMC side.

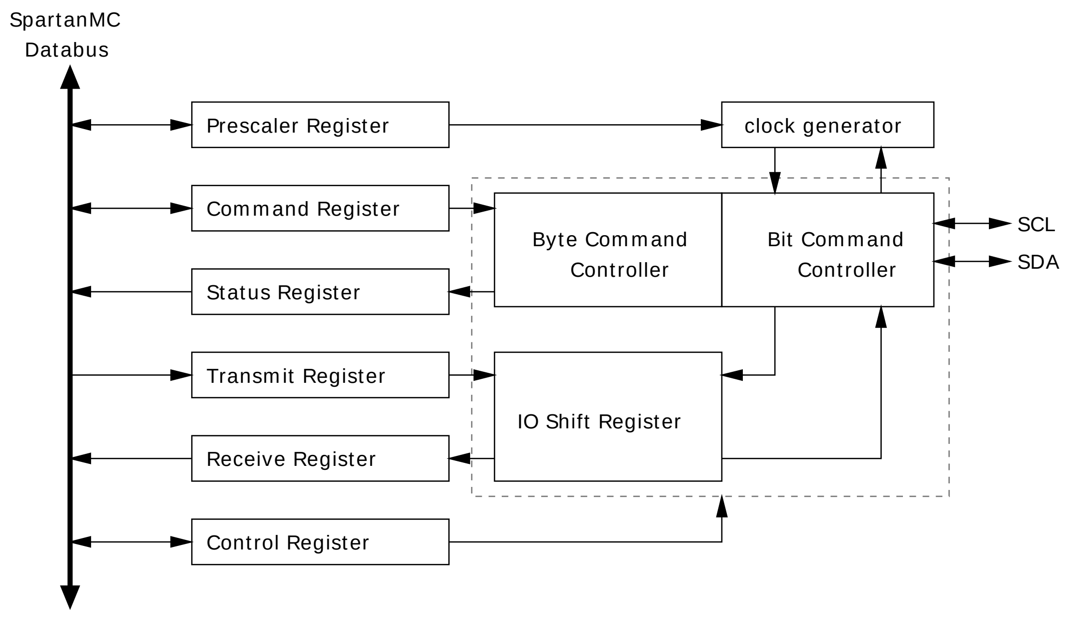

Figure 1: I2C block diagram

The I2C master controller can be configured and controlled by setting the writable registers such as the `Control` , `Command` and `Transmit` register on the SpartanMC side with flags, commands, slave addresses or data to be sent. With respect to the current settings, it will operate autonomously, i.e. send/receive data to/from slave. After current operation, the corresponding bits in the `Status` register will be set and the received data will be written in the `Receive` register. Both of the `Status` and `Receive` register are read-only.

1. Communication

As mentioned above, both SDA and SCL remain high, if there is no transmission between any master and slave. In this case, the I2C bus is considered as idle and can be used by any master. To start a transmission, SDA is pulled low while SCL remains high. After the start signal, 8-bit data packets will be transferred, one bit on each rising edge of SCL. Since multiple slaves can be attached to the I2C bus, each of them should have a unique 7-bit address so that it can be distinguished from the other slaves. As the first packet, the master should always put the 7-bit address of the target slave and one direction bit on the bus. If the direction bit is 1, the master wants to read data from the slave, otherwise write data to it. After the corresponding slave has received the start packet, it needs to send 1-bit acknowledge back to the master as response. After this handshake, the master can begin reading or writing data. If the current transmission is over, SDA must be released to float high again which is used as stop signal and idle marker. Except for the start and stop signal, the SDA line only changes while SCL is low. The timing diagram below shows an example transmission of two data packets.

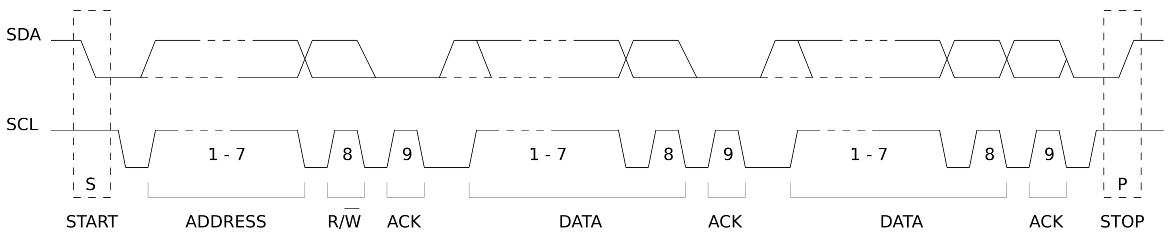

Figure 2: SCL, SDA Timing for Data Transmission

Each time after a data packet has been transmitted in one direction, an acknowledge bit needs to be transmitted in the other direction, as shown in the following diagram.

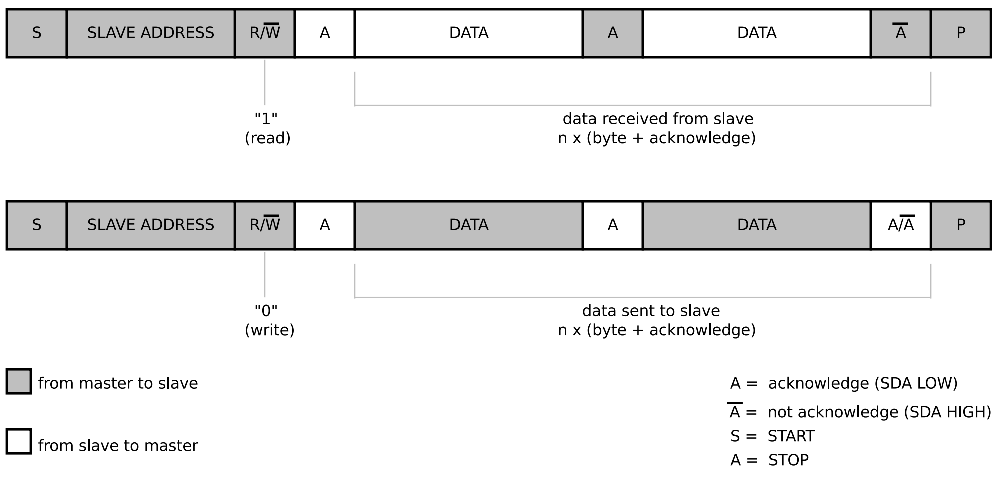

Figure 3: I2C Acknowledge

If the transmitter gets a "0" (ACK) as acknowledge, the transmission has succeeded. Otherwise, if it gets a "1", meaning that:

- If the transmitter is master
    - Unknown slave
    - Busy slave
    - Unknown command
- If the transmitter is slave
    - Stop request from the master

2. Bus Arbitration

Since multiple masters can be connected to an I2C bus, several of them may start the transmission simultaneously. To overcome this situation, all masters monitor SDA and SCL continuously. If one of them detects that SDA is low while it should actually be high on the next rising edge of SCL, it will stop the current transmission immediately. This process is called *arbitration* and illustrated in the following diagram.

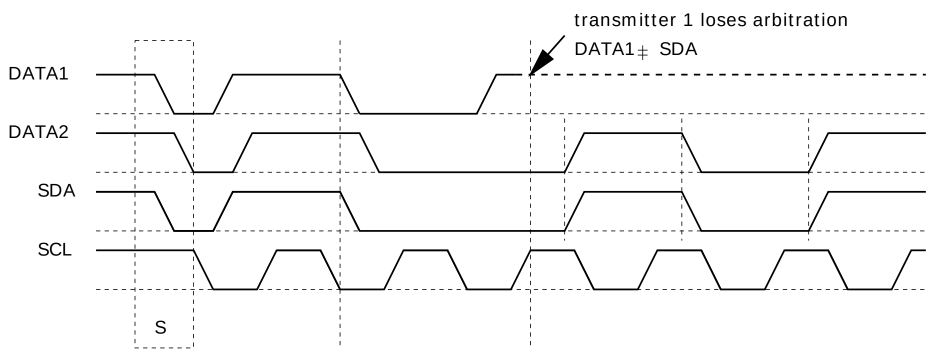

Figure 4: I2C Arbitration

3. Peripheral Registers

3.1. I2C Register Description

The I2C peripheral provides five 18-bit registers which are mapped to the SpartanMC address space. In the following, the layout of each register is described in more detail.

Table 1: I2C registers

Offset	Name	Access	Description
0	CONTROL	r/w	Contains a 16-bit clock divider and two enable bits for the I2C master itself and the interrupt controller respectively.
1	TX	w	Contains the current byte to be sent.
2	RX	r	Contains the current recieved byte.
3	COMMAND	w	Used to set I2C commands.
4	STATUS	r	Contains the controller status flags.

3.2. CONTROL Register

Table 2: I2C control register layout

Bit	Name	Access	Default	Description
0-15	PRESCALER	r/w	65535	This field is used to set the clock frequency of the SCL line. To change its value the CORE_EN bit must be set to zero. The prescaler factor can be dermined through the following equation: prescaler = (peripheral_clock / (5 * desired_SCL)) -1.
16	CORE_EN	r/w	0	Enable I2C core. If set to 1 the I2C core is enabled. (The prescaler value remains constant.)
17	IEN	r/w	0	Enable interrupt. If set to 1 the interrupt is enabled.

3.3. TX Register

Table 3: I2C transmit data register layout

Bit	Name	Access	Default	Description
0-7	TX	w	0	Register for data to be sent.
8-17	-	w	0	Not used.

3.4. RX Register

Table 4: I2C receive data register layout

Bit	Name	Access	Default	Description
0-7	RX	r	0	Register for received data.
8-17	-	r	0	Not used. (Read as zero)

3.5. COMMAND Register

Table 5: I2C command register layout

Bit	Name	Access	Default	Description
0	IACK	r/w	0	Interrupt acknowledge. If set to 1 the pending interrupt will be cleared.
1-2	-	r/w	0	Not used.
3	ACK	r/w	0	If set to 0, acknowledge (0) will be sent. Otherwise, not acknowledge (1) will be sent.
4	WR	r/w	0	If WR = 1, the data in the TX register will be written to slave.
5	RD	r/w	0	If RD = 1, the RX register will be filled with data from slave.
6	STO	r/w	0	Send stop signal.
7	STA	r/w	0	Send (re-)start signal.
8-17	-	r/w	0	Not used.

Note: If both WR and RD are set to 1 at the same time, the read operation will be carried out.

3.6. STATUS Register

Table 6: I2C status register layout

Bit	Name	Access	Default	Description
0	IF	r	0	This bit is set to 1 when an interrupt is pending and IEN in Control register has been set. An interrupt occurs, if: • A byte transfer has been completed. • The arbitration has been lost.
1	TIP	r	0	Is set to 1 when a transfer is in progress.
2-4	-	r	0	Not used.
5	AL	r	0	Is set to 1 if the arbitration has been lost.
6	I2C_BUSY	r	0	Is set to 1 after a start signal has been detected and set to 0 after a stop signal has occurred.
7	RX_ACK	r	0	Is set to 1 if a not acknowledge (NAK) has been received.
8-17	-	r	0	Not used.

3.7. I2C C-Header i2c_master.h for Register Description

```
#ifndef __I2C_MASTER_
#define __I2C_MASTER_

#ifdef __cplusplus
extern "C" {
#endif

/*
* Definitions for the Opencores i2c master core
*/
// Rückgabewerte für non blocking read
#define   I2C_OK      0
#define   I2C_NO_ACK   1

/* --- Definitions for i2c master's registers --- */

/* ----- Read-write access                                 */

//#define I2C_PRER   0x00   /* Low byte clock prescaler
register */
#define   I2C_CTR      0x00   /* Control
register            */

/* ----- Write-only registers                              */

#define   I2C_TXR      0x01   /* Transmit byte
register         */
#define   I2C_CR      0x03   /* Command
register            */

/* ----- Read-only registers                              */

#define   I2C_RXR      0x02   /* Receive byte
register         */
#define   I2C_SR      0x04   /* Status
register               */

/* ----- Bits definition                                 */

/* ----- Control register                                */

#define   I2C_EN      (1<<16)   /* Core enable
bit:           */
            /*   1 - core is enabled      */
            /*   0 - core is disabled      */
```

```
#define   I2C_IEN      (1<<17)   /* Interrupt enable
bit           */
            /*   1 - Interrupt enabled      */
            /*   0 - Interrupt disabled      */
            /* Other bits in CR are reserved   */

/* ----- Command register bits                              */
#define I2C_STA      (1<<7)   /* Generate (repeated) start
condition*/
#define I2C_STO      (1<<6)   /* Generate stop
condition         */
#define I2C_RD      (1<<5)   /* Read from
slave              */
#define I2C_WR      (1<<4)   /* Write to slave              */
#define I2C_NAK      (1<<3)   /* Acknowledge send to
slave      */
            /*   0 - ACK               */
            /*   1 - NACK               */
#define   I2C_ACK      0
#define I2C_IACK   (1<<0)   /* Interrupt acknowledge         */

/* ----- Status register bits                              */

#define I2C_RXACK   (1<<7)   /* ACK received from
slave         */
            /*   0 - ACK               */
            /*   1 - NACK               */
#define I2C_BUSY   (1<<6)   /* Busy bit               */
#define I2C_AL      (1<<5)   /* Arbitration lost            */
#define I2C_R_W      (1<<2)   /* last byte read or
write         */
            /*   0 - READ               */
            /*   1 - WRITE               */
#define I2C_TIP      (1<<1)   /* Transfer in
progress         */
#define I2C_IF      (1<<0)   /* Interrupt flag              */

/* bit testing and setting macros                           */

#define ISSET(reg,bitmask)   ((reg)&(bitmask))
#define ISCLEAR(reg,bitmask)   (!(ISSET(reg,bitmask)))
#define BITSET(reg,bitmask)   ((reg)|(bitmask))
#define BITCLEAR(reg,bitmask)   ((reg)|(~(bitmask)))
#define BITTOGGLE(reg,bitmask)   ((reg)^(bitmask))
#define REGMOVE(reg,value)   ((reg)=(value))

typedef   volatile struct {
   volatile unsigned int   ctrl;   // (r/w)
   volatile unsigned int   txr;   // (r/w)
   volatile unsigned int   rxr;   // (r)
   volatile unsigned int   cmd;   // (r/w)
   volatile unsigned int   stat;   // (r)
} i2c_master_regs_t;

#ifdef __cplusplus
}
#endif

#endif //define __I2C_MASTER
```

3.8. Basic Usage of the I2C Registers

The structure shown above serves as interface between hardware and software. It can be used directly in a C program by including the header file `<i2c_master.h>` . This section presents several quite simple examples to illustrate the usage of this register.

First, assume that `I2C_MASTER_0` is a pointer which contains the physical address of an I2C master.

- **Example 1 : Enable the I2C master and set the prescaler to 134**

```
I2C_MASTER_0->ctrl = I2C_EN | 134;
```

- **Example 2 : Send write request to the slave at the address 0x70**

```
I2C_MASTER_0->txr = 0x70 << 1; // or 0xE0
I2C_MASTER_0->cmd = I2C_WR | I2C_STA;
```

- **Example 3 : Check if the current 8-bit packet has been transfered**

```
/* wait as long as TIP is set */
while(I2C_MASTER_0->stat & I2C_TIP);

/* do something here */
```

- **Example 4 : Check if a not acknowledge has been received**

```
if(I2C_MASTER_0->stat & I2C_RXACK)
    return I2C_NO_ACK;
```

- **Example 5 : Write a constant value 0xFF to the slave**

```
I2C_MASTER_0->txr = 0xFF;
I2C_MASTER_0->cmd = I2C_WR;
```

- **Example 6 : Send read request to the slave at the address 0x70**

```
I2C_MASTER_0->txr = (0x70 << 1) + 1; // or 0xE1
```

```
I2C_MASTER_0->cmd = I2C_WR | I2C_STA;
```

- **Example 7 : Read one last packet from the slave**

```
int v;
I2C_MASTER_0->cmd = I2C_RD | I2C_NAK | I2C_STO;
while(I2C_MASTER_0->stat & I2C_TIP);
v = I2C_MASTER_0->rxr;
```

Note: Sometimes, a hardware manufacturer may give an 8-bit slave ID instead of a 7-bit address. This ID is actually equal `address << 1` and implies that the direction bit is 0. Therefore, it can be sent to the slave as write request directly and `ID + 1` can be used as read request.