17. Serial communication - I²C

Two typical serial busses will be discussed and their use will be demonstrated in this and next chapter, these are I²C (Inter Integrated Circuit, IIC or I2C) and SPI (Serial Peripheral Bus). Only a simple variant of busses without the error checking will be implemented. The experiment requires a sensor that can communicate using the bus, and suitable accelerometers / gyroscopes will be used in our laboratory.

17.1. The I2C protocol - hardware

Sometimes sensors are distant to the microcontroller. In such case it might be unreasonable to send analog signal from sensor to a distant microcontroller containing an ADC due to the possible degradation of analog signal along the line. In would be better to include an ADC into the sensor, and pass the digital result of the conversion to the microcontroller. Digital signals are far less prone to degradation, and ADCs are simple and cheap to include into sensors. Such sensors are quite frequent at the present state of technology.

Digital signals coming from an ADC are 8 or 16 bits wide, and ADC chip alone needs additional control and status signals. When one considers the number of required wires and pins at the microcontroller to accommodate all these signals, one recognizes the need for a more efficient transfer of data into a microcontroller. Serial busses were invented to avoid these problems. They use considerably less wires to transfer the same information, and several devices can be connected to the same bus. The information is conveyed serially bit by bit in accurately defined time slots. To avoid problems with time slots and synchronization, a common clock signal is used by all devices connected to the same serial bus. The price one has to pay to use fewer wires is the prolonged time to transfer the same information and a rather complex set of rules to be obeyed when transferring information or devices connected to the common bus might rebel and refuse to transfer.

The I²C is the name of the bus that requires three wires between devices: a common ground and two wires for clock signal (SCL, Serial CLock) and data signal (SDA, Serial DAta). Several units may be connected to the same bus as shown in Fig. 17.1, but only one of the devices controls the dataflow; this device is called a master, others are called slaves. All devices have a so called “open collector” (“open drain”) outputs meaning that they can only pull a signal low, but never force it to go high. This is the task of two pull-up resistors, one for each wire to pull the line to logic high. Therefore, if no device requests the line to be low, the line remains high due to the pull-up resistor. If any or many devices turn on their outputs pulling the line low, the line becomes logic low. The value of a pull-up resistor depends on the

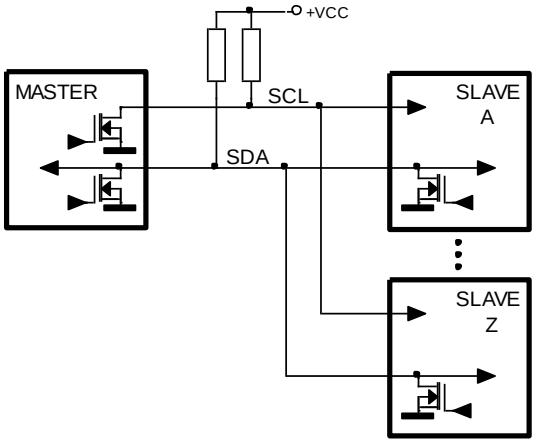


Figure 17.1: The I2C bus can be used to transfer data between the MASTER and one of the SLAVE devices

required speed, and is typically few kΩ. The logic levels are defined by the power supply +VCC of the devices connected to the bus and their technology; in our case the power supply is close to 3.3V.

The procedure to transfer the data over I^2C bus is described in I^2C protocol. There are four typical combinations of signals SCL and SDA:

a) Start combination. Both signals SDA and SCL are initially high. First signal SDA goes low, next signal SCL goes low, as in Fig. 17.2a, left. The important part is that signal SDA changes from high to low when signal SCL is high. The start combination defines the beginning of the transmission over the bus.

b) Stop combination. Both signals SDA and SCL are initially low. First SCL goes high, and then signal SDA goes high, as in Fig. 17.2b, right. The important part is that the signal SDA changes from low to high when signal SCL is high. The stop combination defines the end of transmission over the bus.

c) Bit transfer combination. Signal SDA (serial data) can be either low or high, and the signal SCL changes from low to high and then to low again forming one clock pulse on the SCL line. This combination clocks one data bit into the destination device, Fig. 17.2c. A string of eight (ten in special cases) bits is used to send the complete byte. The signal SDA must not change during the time signal SCL in high.

d) Acknowledge combination. The destination device is expected to confirm safe receipt of a byte. A string of eight data bits is followed by an acknowledge combination, where the originating device releases the SDA line, and the master device issues one clock pulse at the SCL line. The receiving device is supposed to pull the SDA line low during the clock pulse if it managed to receive eight bits successfully, Fig. 17.2d.

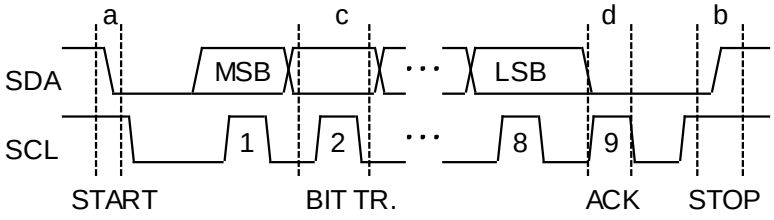


Figure 17.2: The signals on the I2C bus

There can be more than one slave device on the bus, and the master device can communicate with one slave device at a time. The protocol defines the addressing procedure to activate one of the slave devices. The I^2C standard allows the use of either 7-bit addressing or 10-bit addressing; we will demonstrate the use of 7-bit version.

Consider the situation where the master is writing into the slave, Fig. 17.3, top. The master first sends a start combination, followed by eight bits; here master defines consecutive values at the SDA line and issues eight clock pulses on the SCL line. Out of these eight bits first seven represent the address of the slave, and the 8^{th} bit is low signaling the writing into the slave. If a slave with the address specified is connected to the bus, then the slave confirms its presence by the acknowledge combination; the master sends the 9^{th} clock pulse and the slave pulls the SDA line low during the time of this pulse. From now on the master can send as many bytes to the selected slave as desired. Each byte is followed by the acknowledge combination, where the slave pulls the SDA line low telling the master that it is still able to receive bytes. When the transmission is complete, the master issues the "stop" combination releasing the slave. The address of the slave in this example is 0x37, and the data written is 0x57.

Similarly the reading from the slave into the master is started by a “start” combination and the addressing byte, Fig. 17.3, bottom; out of this first seven bits represent the address, and the 8th bit is high signaling the reading from the slave. If a slave with the address specified is connected to the bus, then the slave confirms its presence by the acknowledge combination; the master sends the 9th clock pulse and the slave pulls the SDA line low during the time of the pulse. From now on the master can read as many bytes from the slave as desired by issuing a pack of eight clock pulses for every byte read. Each byte is followed by the acknowledge combination, where the master pulls the SDA line low telling the slave that it has received the byte and that the slave is expected to continue sending bytes. The transmission is terminated by the master not acknowledging the receipt of the last byte and issuing the “stop” combination. The address of the slave is again 0x37, and it returns 0x39.

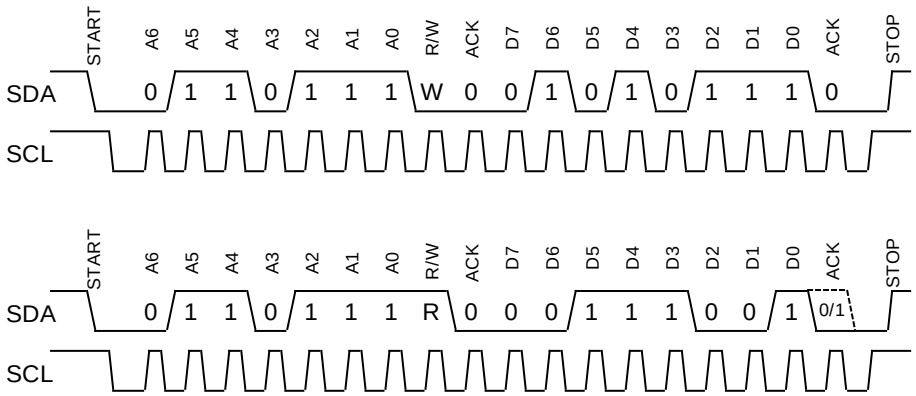


Figure 17.3: I2C bus signals during the writing into the slave (top), and reading from the slave (bottom)

Three blocks named I2C1 to I2C3 intended for I²C communication are built into the microcontroller. These blocks are identical and a simplified diagram for one of them is given in Fig. 17.4. Two pins at the microcontroller are required for lines SDA and SCL, and certain pins can be configured for alternate functions SDA and SCL. The I²C block includes two data related registers:

- the ‘Data Register DR’; the processor writes data to or reads data from this register, the register serves as an buffer between the processor and the actual hardware of the I²C block, and
- the ‘Data Shift Register DSR’; the data is transmitted serially, and this register is used to shift the data byte bit-by-bit out-of or in-to the block.

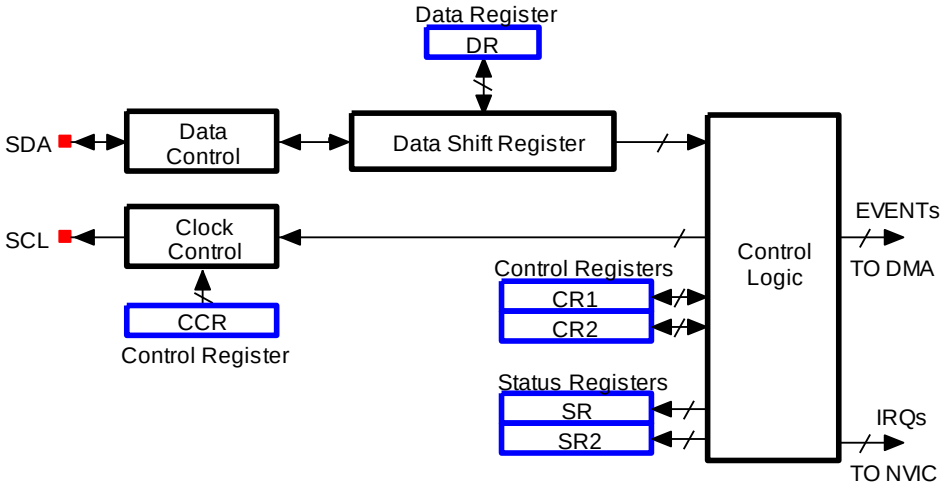


Figure 17.4: A simplified block diagram of an I²C block

The content of the register DR can be accessed by reading from or writing to register ‘I2Cx->DR’. When sending data the processor writes data into the register DR, and the content of this register is copied into the register DSR once the register DSR is free to accept data.

The speed of transmission is defined by control bits in register CCR (Clock Control Register), connected to the Clock Control block. The information of clock pulses, status of SDA and SCL lines, the status of the data register and alike is available to the Control Logic, and can be read-out through status registers SR and SR2. The operation of the control logic is defined by bits in control registers CR1 and CR2. The control logic can issue events and interrupt requests to call assistance to the block. In order to configure these registers efficiently a CMSIS library includes several functions, and they are available in the source file "stm32f4xx_I2c.c". Brief instructions on the use of functions are included in the file. These CMSIS functions require the use of data structures and definitions that are declared in the header file "stm32f4xx_i2c.h".

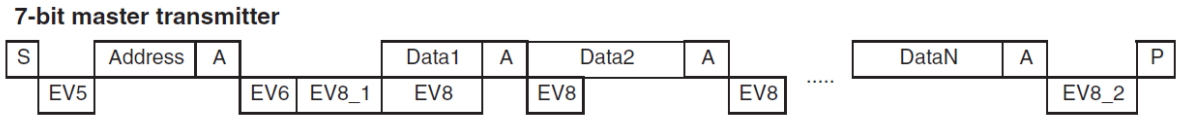
17.2. The I2C protocol – sequence of events, writing

The protocol I^2C requires interaction of the I^2C block with the processor. After every major step in communication the processor must instruct the block on successive action, and all major states of the block are signaled by events. The procedure is as follows: the processor first configures the I^2C block, and then requests a certain action. It takes some time for the block to complete the action, and the processor must wait for the competition which is signaled by an event. Put into hardware: an event toggles a bit in status register SR; when set, the event has happened. The event can be used to trigger an interrupt, or the processor can simply waste time executing an empty loop waiting for the event (the bit to be set). This second option is not very effective, but will be used here for simplicity.

A slave unit commonly houses more than one byte for data, so the master should tell where inside the slave the data should be written to. This is resolved by master sending three bytes; the first byte represents the address of the slave, the second byte represents the address of the register within the slave, and the third byte is the actual data to be written into the selected register within the slave. The procedure for sending a byte from processor into the peripheral using I^2C bus can be dissected into:

1. start the transmission (S),
2. wait for the start condition on the I^2C bus (this is called EV5 in CMSIS for STM32F4xx processors),
3. send the address of the slave (Address) to register DR,
4. wait for the acknowledge signal from the slave (EV6),
5. send the address within the slave (Data1) to register DR,
6. wait for the register DR to become empty (EV8),
7. send the actual data (Data2) to register DR,
8. wait for the acknowledge signal from the slave(EV8_2, here the transmission has finished),
9. stop the transmission (P).

These steps are depicted in Fig. 17.5, a copy from the RM0090, page 826, figure 243 for actions and events during the writing a byte from master to slave, 7 bit addressing.



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge,
EVx= Event (with interrupt if ITEVFEN = 1)

EV5: SB=1, cleared by reading SR1 register followed by writing DR register with Address.
EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2.
EV8_1: TxE=1, shift register empty, data register empty, write Data1 in DR.
EV8: TxE=1, shift register not empty,.data register empty, cleared by writing DR register
EV8_2: TxE=1, BTF = 1, Program Stop request. TxE and BTF are cleared by hardware by the Stop condition

Figure 17.5: Commands and events during the transmission of a byte from master to slave

As stated above: events are connected with bits in status register SR. Waiting for the event EV5, for instance, means waiting for a certain combination of bits in register SR, or, put into the CMSIS terms, waiting for 'I2C_EVENT_MASTER_MODE_SELECT'. Similarly, waiting for event EV6 becomes waiting for 'I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED', and waiting for event EV8 equals waiting for 'I2C_EVENT_MASTER_BYTE_TRANSMITTING'. Finally, waiting for event EV8_2 is the same as waiting for'I2C_EVENT_MASTER_BYTE_TRANSMITTED', see the include file "stm32f4xx_i2c.h", lines 320 to 407, for the justification of terms.

All passing of data and checking of events is also translated into CMSIS terms, and the function "I2C_Write1Byte" to send one byte of data into the slave to a given address is presented next.

```
char I2C_Write1Byte(char SlaveAddr, char addr, char byte1)   {
  while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY));                    // wait if I2C1 is busy   // 2
  I2C_GenerateSTART(I2C1, ENABLE);                                  // generate START         // 3
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));       // wait for EV5           // 4
  I2C_Send7bitAddress(I2C1, (SlaveAddr << 1), I2C_Direction_Transmitter); // send slave addr   // 5
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED)); // wait for EV6    // 6
  I2C_SendData(I2C1, addr);                                         // addr within slave      // 7
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTING));  // wait for EV8          // 8
  I2C_SendData(I2C1, byte1);                                        // send data to slave     // 9
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));   // wait for EV8_2        // 10
  I2C_GenerateSTOP(I2C1, ENABLE);                                   // generate STOP          // 11
  return (0);
}
```

The function receives three arguments for defining the address of the slave ('SlaveAddr'), selecting the register within the slave ('addr') and the data to be sent to this register ('byte1'); all arguments are 8 bits wide and declared as characters. The rest of the function strictly follows what was written in the paragraph above, but in CMSIS terms.

The availability of the I^2C block is tested first in line 2. If the block is busy then the processor waits. The actual procedure starts in line 3 where a processor requests a "start" condition on the I^2C bus, a CMSIS function "I2C_GenerateSTART()" is used. The processor must then wait for the actual "start" condition to be established, and this is done in line 4, where the processor periodically checks the status register against the state 'I2C_EVENT_MASTER_MODE_SELECT' (EV5). The CMSIS function "I2C_CheckEvent()" is used to do the checking.

Once the condition is confirmed the processor proceeds with writing the address of the slave into the register DR, and this is performed by a call to next CMSIS function "I2C_Send7bitAddress()" in line 5. The arguments of the function are the slave address as it should show in the timing diagram from Fig. 17.3 top, followed by a bit to define the read/write option ('I2C_Direction_Transmitter' in this case). The processor must wait for event EV6 now, and this is done in line 6, where the status register is periodically checked against status 'I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED'. Once this status is confirmed it is also true that the slave device responded, and that the processor can continue by sending the address within the slave.

The address within the slave is sent in line 7, and a CMSIS function "I2C_SendData()" is used. The argument is the address within the slave, and the processor must now wait until the register DR is ready to receive next byte; the processor must wait for event EV8. This is performed in line 8 of the function, where the content of the status register is periodically checked against status 'I2C_EVENT_MASTER_BYTE_TRANSMITTING'. Once this status is detected, the processor proceeds to line 9 to send the actual data to the slave.

The data is sent to the slave by the same CMSIS function call, this time the argument is the byte to be sent. This terminates the transmission, and the "stop" condition should be re-established to release the I^2C bus, but not immediately. The processor must wait for all bits to be safely transmitted to the

slave and for slave to confirm the receipt; the processor must wait for event EV8_2, in CMSIS terms 'I2C_EVENT_MASTER_BYTE_TRANSMITTED'. This is performed in line 10, then the "stop" condition is requested in line 11, and the transmission is terminated.

If more than one byte is to be sent to the slave, then commands from lines 8 and 9 should be repeated for each additional byte to be sent, and these bytes are written to consecutive locations within the slave.

17.3. The I2C protocol – sequence of events, reading

A typical reading from slave is performed in two major steps due to properties of most of the slaves. The first major step A is used to send the address within the slave from where the master is supposed to read. The second major step B is used to actually read data from the slave. The procedure for reading a byte from slave to the master using I^2C bus can be dissected into:

A1. start the transmission (S),

A2. wait for the start condition on the I^2C bus (EV5),

A3. send the address of the slave (Address) to register DR, select writing

A4. wait for the acknowledge signal from the slave (EV6),

A5. send the address within the slave (Data1) to register DR,

A6. wait for the acknowledge signal (EV8_2) from the slave telling that the transmission has ended,

B1. start the transmission once again (S),

B2. wait for the start condition on the I^2C bus (EV5),

B3. send the address of the slave (Address) to register DR once more, but select reading,

B4. wait for the acknowledge signal from the slave (EV6),

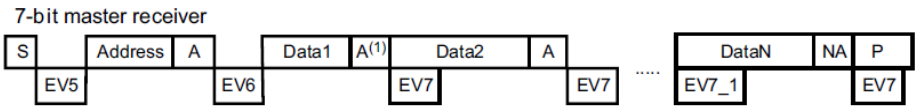
B5. configure the I2C block not to acknowledge the byte received, the reading automatically starts,

B6. wait for one byte to accumulate in the data register (EV7),

B8. retrieve the byte read from the slave by reading it from data register DR,

B9. stop the transmission (P).

Steps under A are the same as for writing and are taken from the previous figure. Steps for B are depicted in Fig. 17.5, a copy from the RM0090, page 828, figure 244 for actions and events during reading a byte from slave to master, 7 bit addressing. Note that reading of a byte in step B5 starts automatically after the configuration of the acknowledge action.



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge,
EVx= Event (with interrupt if ITEVFEN=1)
EV5: SB=1, cleared by reading SR1 register followed by writing DR register.
EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2. In 10-bit master receiver mode, this sequence should be followed by writing CR2 with START = 1.
In case of the reception of 1 byte, the Acknowledge disable must be performed during EV6 event, i.e. before clearing ADDR flag.
EV7: RxNE=1 cleared by reading DR register.
EV7_1: RxNE=1 cleared by reading DR register, program ACK=0 and STOP request
EV9: ADD10=1, cleared by reading SR1 register followed by writing DR register.

Figure 17.5: Commands and events during the reading of a byte from slave to master

Events are again associated with some predefined names in CMSIS library: waiting for the event EV7 means waiting for 'I2C_EVENT_MASTER_BYTE_RECEIVED'.

All passing of data and checking of events is translated into CMSIS terms, and the function “I2C_Read1Byte” to read one byte of data from slave is presented next.

```
char I2C_Read1Byte(char SlaveAddr, char addr)     {
  while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY));                    // wait if I2C1 is busy   // 2
  I2C_GenerateSTART(I2C1, ENABLE);                                  // generate START         // 3
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));       // wait EV5               // 4
  I2C_Send7bitAddress(I2C1, (SlaveAddr << 1), I2C_Direction_Transmitter);  // send device addr // 5
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED)); // wait EV6        // 6
  I2C_SendData(I2C1, addr);                                         // addr within device     // 7
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));  // wait for EV8_2         // 8

  I2C_GenerateSTART(I2C1, ENABLE);                                  // generate START         // 10
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));       // wait EV5               // 11
  I2C_Send7bitAddress(I2C1, (SlaveAddr << 1), I2C_Direction_Receiver);  // send device addr    // 12
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED)); // wait EV6           // 13
  I2C_AcknowledgeConfig(I2C1, DISABLE);                             // do not acknowledge     // 14
  while( !I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED) );   // wait EV7               // 15
  char byte1 = I2C_ReceiveData(I2C1);                               // retrieve byte          // 16
  I2C_GenerateSTOP(I2C1, ENABLE);                                   // generate STOP          // 17
  return (byte1);                                                                            // 18
}
```

The function receives two arguments for defining the address of the slave (‘SlaveAddr’), and selecting the register within the slave (‘addr’); both arguments are 8 bits wide and declared as characters. The rest of the function strictly follows what was written in the paragraph above, but in CMSIS terms. The function returns a byte retrieved from the slave.

Lines 2 to 7 are the same as used for writing of a byte to the slave, and will not be discussed here again. Line 8 differs, since the processor must wait until a complete address is safely transferred into the slave (EV8_2) before proceeding with steps for reading.

Reading starts in line 10, where the processor requests the “start” condition once again. It must wait for this condition to be implemented, this is done in line 11. Line 12 re-sends the address of the slave, this time the direction is reversed from before. The processor must wait for this address to reach the slave and the slave to acknowledge the receipt of the address, in CMSIS terms it must wait for condition ‘I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED’.

The processor then proceeds to initiate the actual reading. This is done by configuring the action of the master whether to acknowledge the receipt of the byte read from the slave or not in line 14. In out example we are reading one byte only, and we should not confirm the receipt in order for the slave to stop sending after one byte. Once the byte is safely stored in the register DR, the condition ‘I2C_EVENT_MASTER_BYTE_RECEIVED’ can be detected in line 15, and the processor can read the content of the register DR in line 16. Since this is the last byte to be read, the operation is terminated by requesting a “stop” condition in line 17 and returning the byte read in line 18.

If more than one byte is to be read from the slave, then the last part of the function following line 13 should be replaced by the listing below.

```
  I2C_AcknowledgeConfig(I2C1, ENABLE);                              // acknowledge 1st byte   // 14
  while( !I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED) );   // wait EV7               // 15
  char byte1 = I2C_ReceiveData(I2C1);                               // retrieve byte          // 16
  I2C_AcknowledgeConfig(I2C1, DISABLE);                             // do not acknowledge 2nd byte
  while( !I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_RECEIVED) );   // wait EV7               // 18
  char byte2 = I2C_ReceiveData(I2C1);                               // retrieve byte          // 19
  I2C_GenerateSTOP(I2C1, ENABLE);                                   // generate STOP          // 20
  return ((byte2 << 8) + byte1);                                                             // 21
```

The master should acknowledge the receipt of the first byte from the slave, and not acknowledge the receipt of the second byte. Both bytes are combined into a 16 bit short integer and returned; this requires a redefinition of the function as well from ‘char’ to ‘short’.

17.4. The program to read data from sensor LIS3LV02DQ using I²C bus

The program to initialize and read the value from the accelerator chip is listed next.

```
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_i2c.c"
#include "LCD2x16.c"
#include "dd.h"

#define LIS_chip_I2C_address 0x1d

int main ()  {
  I2C1init();                                  // I2C1 init
  LCD_init();                                  // Init LCD
  LCD_string("I2C demo Z", 0);                 // write title
  LCD_string("X        Y", 0x40);              // write title

  I2C_Write1Byte (LIS_chip_I2C_address, 0x20,0xc7);                                    // 15

  while (1) {
    LCD_sInt3DG(I2C_Read2Byte (LIS_chip_I2C_address, 0x28), 0x41, 0x01);      // show acc x
    LCD_sInt4DG(I2C_Read2Byte (LIS_chip_I2C_address, 0x2a), 0x4a, 0x01);      // show acc y
    LCD_sInt4DG(I2C_Read2Byte (LIS_chip_I2C_address, 0x2c), 0x0a, 0x01);      // show acc z
    for (int i = 0; i < 10000000; i++)   {};    // waste time
  };
}
```

The listing starts with inclusion of necessary files, and defines the address of the slave chip as 0x1d, as specified in the datasheet of the chip LIS3LV02DQ, the accelerometer. The processor starts the execution at “main”, where it first calls the configuration function “I2C1init()”, to be discussed still. The execution continues with a call to configuration function for LCD screen and two introductory writes to the screen.

The call in line 15 configures (turns on for all three axes) the accelerometer chip, as described in its datasheets, by writing a value 0xc7 into location 0x20.

The execution of the program then enters the infinite loop where three 16 bit numbers (the acceleration in all three axes) are retrieved from the accelerometer chip and displayed at the LCD.

The last thing to discuss is the configuration of the microcontroller and the I²C block. We are going to use I²C block I2C1, its connections are available at port B, pins 6 and 7. The configuration includes mapping of port pins to the I²C block, and the configuration of I²C block itself. The function is listed below.

```
void I2C1init(void) {
GPIO_InitTypeDef       GPIO_InitStructure;
I2C_InitTypeDef        I2C_InitStructure;

  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);                                // 5
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1,  ENABLE);                                // 6

  GPIOB->BSRRL = BIT_6 | BIT_7;                          // SDA, SCL -> hi            // 8
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_6 | GPIO_Pin_7; // SDA, SCL def            // 9
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF;          // alternate function        // 10
```

```
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;                                    // 11
  GPIO_InitStructure.GPIO_OType = GPIO_OType_OD;          // use open drain !          // 12
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;                                    // 13
  GPIO_Init(GPIOB, &GPIO_InitStructure);                                               // 14

  GPIO_PinAFConfig(GPIOB, GPIO_PinSource6, GPIO_AF_I2C1);  // PB6:I2C1_SCL            // 16
  GPIO_PinAFConfig(GPIOB, GPIO_PinSource7, GPIO_AF_I2C1);  // PB7:I2C1_SDA            // 17

  I2C_InitStructure.I2C_ClockSpeed      = 100000;                                      // 19
  I2C_InitStructure.I2C_Mode            = I2C_Mode_I2C;                                // 20
  I2C_InitStructure.I2C_DutyCycle       = I2C_DutyCycle_16_9;                          // 21
  I2C_InitStructure.I2C_OwnAddress1     = 0;                                           // 22
  I2C_InitStructure.I2C_Ack             = I2C_Ack_Disable;                             // 23
  I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;            // 24
  I2C_Init(I2C1, &I2C_InitStructure);                                                  // 25
  I2C_Cmd(I2C1, ENABLE);                                                               // 26
}
```

Two data structures are required by CMSIS functions for configuration, those are declared first. Next clock signals for port B and I2C1 blocks are enabled in lines 5 and 6.

The configuration of port B is given in lines 8 to 17. Pins 6 and 7 are to be dealt with, those are indicated in line 9. Both pins should implement alternate function, have no terminating resistors, and use open drain configuration of the output, as stated in lines 10 to 12. The speed of operation is not critical and selected as 25MHz here. The configuration function for port “GPIO_Init()” is called in line 14 utilizing the pointer to the initialized data structure ‘GPIO_InitStructure’.

The alternate function is selected next for those two pins in lines 16 and 17.

Lines 19 to 26 are used to initialize and configure the I2C1 block. A brief description of the configuration procedure is given in the source file “stm32F4xx_I2C.c”. The data structure ‘I2C_InitStructure’ is described in the header file “stm32f4xx_i2c.h”, lines 54 to 73, and is initialized as follows:

- The member ‘.I2C_ClockSpeed’ specifies the clock frequency for the data transfer, it must be less than 400000 [Hz].
- The member ‘.I2C_Mode’ specifies the mode of operation for the I^2C block; the block can implement either I^2C protocol or SM bus protocol. In our example the I^2C protocol is required, and the member is initialized accordingly.
- The member ‘.I2C_DutyCycle’ specifies the ratio of clock pulse width versus the pause between pulses; the possibilities are listed in the header file in lines 114 and 115, and are of no importance here; an arbitrary one of the available is selected.
- The member ‘.I2COwnAddress1’ specifies the first device own address, in our case 0.
- The member ‘.I2C_Ack’ enables or disables the acknowledgement step in general; we will disable it here, and enable it when needed by appropriate CMSIS function call.
- The last member ‘.I2C_AckknowledgeAddress’ specifies the expected length of the address; it can be either 7 or 10 bit, and is initialized to 7 bit in our example.

The function call in line 25 uses this data structure to configure the I2C1 block, and the last function call in line 26 enables the I2C1 block.