17. I²C communication channel

Sometimes sensors are distant to the microcontroller. In such case it might be impractical to send analog signal from the sensor to the ADC included in the microcontroller due to the possible degradation of analog signal quality along the line. In would be better to include the ADC into the sensor, and pass the digital result of the ADC conversion to the microcontroller. Digital signals are far less prone to degradation, and ADCs are easy and cheap to include into integrated circuits. Sensors with ADCs included are quite frequent at the present state of technology.

The digital signals at the output of the ADC are 8 or 16 bits wide, and ADC chip alone uses 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 ADC results into the microcontroller. Serial busses were invented to avoid these problems. They use considerably less wires to transfer the same signals as mentioned before. The data is conveyed serially bit by bit in precisely defined time slots. To avoid problems with time slots, a common clock signal is used by all devices connected on a 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 programming the transfer of data or the devices connected to the common bus might rebel and refuse to transfer data.

The two typical serial busses will be explained and implemented in this and next chapter, these are I^{2}C (Inter Integrated Circuit, IIC or I^{2}C) and SPI (Serial Peripheral Bus). Only a simple variant of busses without the error checking will be implemented. The experiment requires a sensor conforming to a bus used, and accelerometers / gyroscopes will be used here.

The I^{2}C is the name of the bus that requires three wires between units: 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. 1, but only one of the devices controls the dataflow; this device is called a master, others are called slaves. All devices have so called "open collector" or "open drain" outputs meaning that they can only pull a signal low, but never force it to go high. There are 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

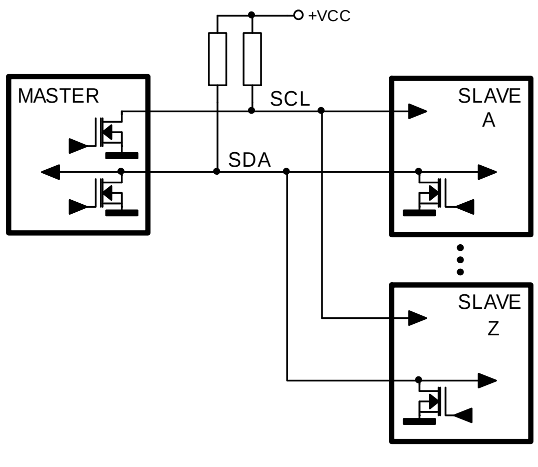


Figure 1: The I2C bus can be used to transfer data between the MASTER and one of the SLAVE devices

pull-up resistor. If any or many of the devices turn on their outputs pulling the line low, the line becomes logic low. The value of a pull-up resistor depends on the required speed, and is typically around 10 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 equals 3.5V.

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

- Start combination. Both SDA and SCL are initially high. First signal SDA goes low, next signal SCL goes low, as in Fig. 2a, left. The important part is that signal SDA changes from high to low during the time signal SCL is high. The start combination defines the beginning of the transmission over the bus.
- Stop combination. Both signals SDA and SCL are initially low. First SCL goes high, and then signal SDA goes high, as in Fig. 2d, right. The important part is that the signal SDA changes from low to high during the time signal SCL is high. The stop combination defines the end of transmission over the bus.
- 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. 2c. A string of eight (ten in special cases) bits is used to send the complete byte. The line SDA must not change during the time signal SCL in high.
- 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 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 the eight bits successfully, Fig. 2d.

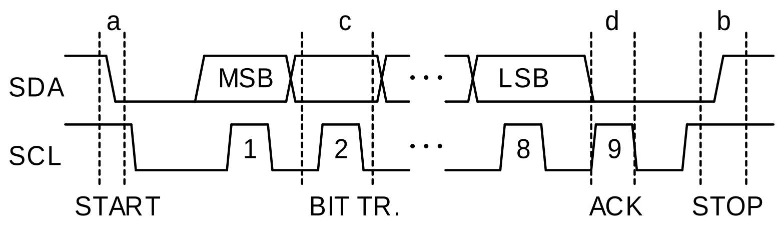


Figure 1: The signals on the I2C bus

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

Consider the situation where the master is writing into the slave, Fig. 3, top. The master first sends a start combination, followed by eight bits; here master defines 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 8th 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 9th 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 37hex, and the data written is 57hex.

Similarly the reading from the slave into the master is started by s "start" combination and the addressing byte, Fig. 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. 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 can continue sending bytes. The transmission is terminated by the master issuing the "stop" combination. The address of the slave is again 37hex, and it returns 39hex.

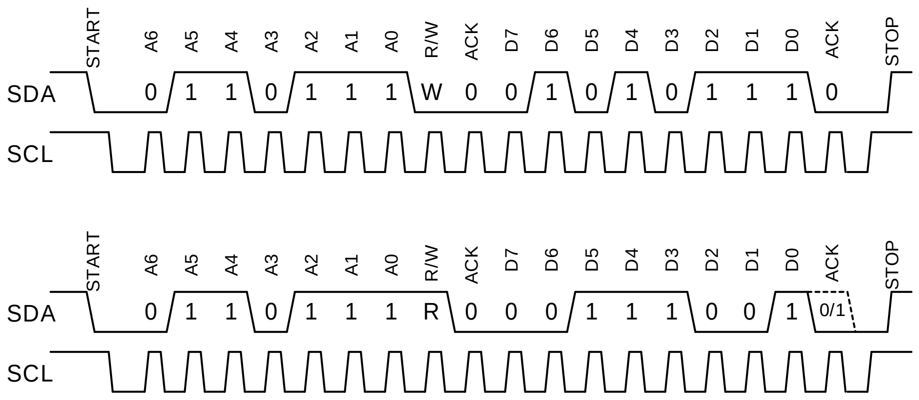

START
A6
A5
A4
A3
A2
A1
A0
R/W
ACK
D7
D6
D5
D4
D3
D2
D1
D0
ACK
STOP
SDA
0 1 1 0 1 1 1 W 0 0 1 0 1 0 1 1 1 0
SCL
SDA
0 1 1 0 1 1 1 R 0 0 0 1 1 1 0 0 1 0/1
SCL

Figure 3: I2C bus signals during the writing into the slave (top), and reading from the slave (bottom)

All digital logic elements are integrated into a block I^{2}C, and three such blocks are built into the microcontroller, these are named I2C1 to I2C3. They are identical and a simplified block diagram for one of them is shown in Fig. 4. Two pins at the microcontroller chip are required (for signals SDA and SCL), and they are mapped from regular ports of the microcontroller. The block includes a Data Shift Register to shift the data byte bit-by-bit out-of or in-to the block. The content of the shift register can be accessed by reading from or writing to register DR. The speed of transmission is defined by writing control bits into the register CCR (Clock Control Register), which defines the behavior of 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

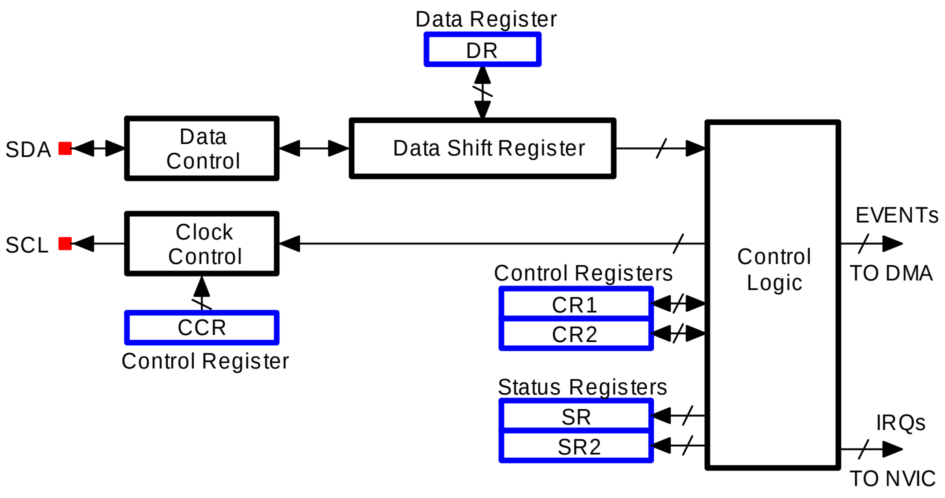
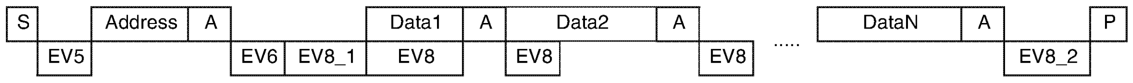


Figure 4: A simplified block diagram of an I^{2}C block

and SR2. The operation of the control logic is be defined by bits in control registers CR1 and CR2. The control logic can issue events and interrupt requests to call assistance to the block.

The protocol I^2C requires interaction of the processor. After every major step in communication the processor must supply instructions to the hardware of what to do next, and all major steps are signaled by events. The procedure is as follows: the processor initializes the I^2C block, and then requests a certain action. It takes some time for the block to complete the requested action, and the competition is signaled by issuing an event. The event can be used as an interrupt, or the processor can simply waste time executing an empty loop waiting for the event. This second option is not very effective, but will be used here for the simplicity of the example.

The diagram on Fig. 5 gives a copy from the RM0090, page 712, figure 242 for actions and events during writing a byte from master to slave.



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge,
EVx= Event

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 5: Commands and events during the transmission of a byte to slave

The procedure starts by processor requesting the I^2C block to issue a "start" condition (S) on the bus. When the "start" condition is established the I^2C block responds with an event (EV5) and sets bit SB in the status register SR1. Once this happens the processor can continue by writing the address of the slave device into the data register DR. The I^2C block proceeds by sending address out bit-by-bit and waiting for acknowledge A from the slave during the ninth bit of the clock SCL . When the ninth bit is over the block issues event (EV6) and sets the bit ADDR in the status register SR1. Once this bit is set the processor can continue. First the bit ADDR must be cleared by reading from the status register SR2, and then the byte to be sent to the slave is written into the data register DR. The fresh content of the register DR is immediately transferred into the shift register, and sending starts. This now empties the data register DR and the block issues new event (EV8, bit TxE in SR1 gets set, Transmitter Empty) telling the processor that a new byte of data can be written into the data register DR in case there are more bytes to be sent. The new byte will not be used by the shift register until the transmission of the current byte is complete, and then the new byte will be automatically transferred into the shift register and sent out, and repeated event (EV8) will be issued signaling the emptiness of the data register DR. If we ignore this event since we have no data to transfer any more, the last event takes place once the complete byte is shifted out of the shift register and both the data register DR and the shift register are empty. This is the event (EV8_2), it signals that the transmission has finished, and bit BTF (Byte Transfer Finished) in status register SR1 is set. The processor can now request a "stop" condition on the I^2C bus to terminate the writing into the slave.

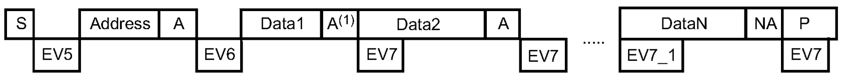
A slave unit commonly houses more than one byte for data, so the master should tell where inside the slave the data should be written to. This is resolved by master sending three bytes; the first byte represents the address of the slave, the second byte represents the address of the register within the slave, and the third byte is the actual data to be written into the selected register within the slave. This is the case in our example.

The function to implement the above is shown in Fig. 6. The function receives two arguments for selecting the register within the slave (Adr) and the data to be sent to this register (Dat); both arguments are 8 bits wide and declared as characters. The rest of the function strictly follows what was written in the paragraphs above. First the “start” condition is requested by setting bit 8 (START) of the register CR1. The processor then waits executing an empty loop for confirmation from the I2C block on the “start” condition, this confirmation comes in a form of a zero-th bit (SB) being set to one in register SR. The address of the slave (0xd0 in this example) is then written into the data register DR, the least significant bit of the address is low requesting the write operation from the I^{2}C block. The processor then again enters an empty loop where it waits for setting of the bit 1 (ADDR) in the status register SR1. Once this bit is set the transmission of the slave address is completed and the processor reads the status register SR2 to clear the bit ADDR, and then writes the address of the register within the slave into the data register DR. After writing it enters an empty loop waiting for the data register DR to become empty. The I^{2}C block transfers the content of the data register into the shift register to transmit it, and sets the bit TxE signaling the processor can write next byte into the data register DR. This allows the processor to exit the empty loop and write the last byte Dat into the data register DR. After this the processor enters into another empty loop where it first waits for the data register DR to become empty, and then for the transmission to be finished. The last is signaled by setting of the bit BTF in the status register SR1. Once this is done the processor terminates the transmission by setting the bit STOP (bit 9) in control register CR1, and the I^{2}C block returns to idle state by sending the “stop” condition.

```
void I2C2_WriteChar (char Adr, char Dat)    {
  I2C2->CR1         |= 0x0100;       // send START bit
  while (!(I2C2->SR1 & 0x0001)) {};  // wait for START condition (SB=1)
  I2C2->DR           = 0xd0;         // slave address   -> DR & write
  while (!(I2C2->SR1 & 0x0002)) {};  // wait for ADDRESS sent (ADDR=1)
  int Status2        = I2C2->SR2;    // read status to clear flag
  I2C2->DR           = Adr;          // Address in chip -> DR & write
  while (!(I2C2->SR1 & 0x0080)) {};  // wait for DR empty (TxE)
  I2C2->DR           = Dat;          // Dat -> DR & write
  while (!(I2C2->SR1 & 0x0080)) {};  // wait for DR empty (TxE)
  while (!(I2C2->SR1 & 0x0004)) {};  // wait for Byte sent (BTF)
  I2C2->CR1         |= 0x0200;       // send STOP bit
}
```

Figure 6: A listing of the function to send a byte over I2C bus to a slave

The diagram on Fig. 7 gives a copy from the manual RM0090, page 714, figure 243 for actions and events during reading from the slave.



Legend: S= Start, S_r = repeated Start, P = Stop, A= Ackowledge, NA = Non-acknowledge, EVx= Event

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 SART = 1.

In case of the reception of 1 byte, the Acknowledge disable must be performed suring 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, programming ACK = 0 and STOP request.

Figure 7: Commands and events during the reception of a byte from the slave

The reading from the slave again starts with the processor requesting the “start” condition. The I^2C block executes the request and issues an event (EV5), where it confirms the “start” condition on the I^2C lines by setting the bit SB (bit 0, status resister SB1). The processor has been waiting for this event, and can now proceed with writing the address of the slave into the Data Register DR. The address is composed of seven bits to select a device on the I^2C bus, and the eighth bit (logic high) to request reading. After this, the processor must wait for the address to be sent and the receipt of the address confirmed by the slave. Once this happens, the event (EV6) is issued by the I^2C block, and the bit ADDR (bit 1, status register SR1) is set. The processor responds by reading the status register SR2 (this clears the bit ADDR). This read, in combination with the eights bit sent as address, signals the I^2C block to continue by issuing nine clock pulses and receive a data byte from the slave. Once the data byte is composed in the shift register, it gets automatically transferred into the Data Register, and an event (EV7) is issued. This sets bit RxNE (bit6, status register SR1), and the processor can now read the received byte from the data register. If this is all to receive, then the processor can request the “stop” combination from the I^2C block and terminate the reading.

The example function in Fig. 8 is more complex. There are many registers within one device connected to the I^2C bus, and one must select a location within the device to read from. The reading becomes more complex and is composed out of two steps. The first step is the writing of the address within the device, and the second step is the reading from the device. Both steps are visually separated by blank lines in Fig. 8.

```
char I2C2_ReadChar (char Adr)    {   // procedure: RM0090, pg. 584!

  I2C2->CR1          |= 0x0100;      // send START bit
  while (!(I2C2->SR1 & 0x0001)) {};  // wait for START condition (SB=1)
  I2C2->DR            = 0xd0;        // slave address -> DR      (LSB=1)
  while (!(I2C2->SR1 & 0x0002)) {};  // wait for ADDRESS sent    (ADDR=1)
  int Status2         = I2C2->SR2;   // read SR2 to clear flag
  I2C2->DR            = Adr;         // register in chip -> DR
  while (!(I2C2->SR1 & 0x0080)) {};  // wait for DR empty        (TxE=1)
  while (!(I2C2->SR1 & 0x0004)) {};  // wait for ByteTransferred (BTF=1)

  I2C2->CR1          |= 0x0100;      // send START bit
  while (!(I2C2->SR1 & 0x0001)) {};  // wait for START condition (SB=1)
  I2C2->DR            = 0xd1;        // slave address -> DR      (LSB=0)
  while (!(I2C2->SR1 & 0x0002)) {};  // wait for ADDRESS sent    (ADDR=1)
  int Status4         = I2C2->SR2;   // read status to clear flag
  while (!(I2C2->SR1 & 0x0040)) {};  // wait for ByteReceived    (RxNE=1)
  I2C2->CR1          |= 0x0200;      // send STOP bit

  return ((char)I2C2->DR);           // return byte
}
```

Figure 8: A listing of the function to retrieve a byte over I2C bus from the slave

The sending of the address within the device is the same as in Fig. 6, but only one byte (the address within the device) gets transferred. Once this is sent, the bit BTF (Byte transferred, bit 2, status register SR1) is set, and the processor requests another “start” condition at the beginning of the second step. Once this condition is re-established, the processor again addresses the slave device, but this time with the least significant bit of the address set to read from the device, and waits for the flag ADDR (ADDR sent, bit 1, SR1). Once this flag is set, the processor clears it by reading the status register SR2, and waits for a byte to become ready in the data register. This is signaled by setting of the bit RxNE (bit 6, Receiver Not Empty, SR1). The processor then reads the byte received and terminates the transmission by requesting the “stop” condition. The function ends by a return statement, where the properly formatted byte is sent back to the main program.

Similar function is prepared for reading two bytes in a row from the same device on the I^2C bus. It would be a waste of time to repeat the function from Fig. 8 two times, and devices on the I^2C bus normally allow sequential reading, where two consecutive addresses within the device I^2C get read. The function is given in Fig. 9. It returns the combined bytes as a 16-bit “short” variable.

```
short I2C2_ReadShort (char Adr)    {

  I2C2->CR1          |= 0x0100;       // send START bit
  while (!(I2C2->SR1 & 0x0001)) {};  // wait for START condition (SB=1)
  I2C2->DR            = 0xd0;         // slave address -> DR      (LSB=1)
  while (!(I2C2->SR1 & 0x0002)) {};  // wait for ADDRESS sent    (ADDR=1)
  int Status2         = I2C2->SR2;    // read SR2 to clear flag
  I2C2->DR            = Adr;          // register in chip -> DR
  while (!(I2C2->SR1 & 0x0080)) {};  // wait for DR empty        (TxE=1)
  while (!(I2C2->SR1 & 0x0004)) {};  // wait for ByteTransferred (BTF=1)

  I2C2->CR1          |= 0x0100;       // send START bit
  while (!(I2C2->SR1 & 0x0001)) {};  // wait for START condition (SB=1)
  I2C2->DR            = 0xd1;         // slave address -> DR      (LSB=0)
  while (!(I2C2->SR1 & 0x0002)) {};  // wait for ADDRESS sent    (ADDR=1)
  I2C2->CR1          |= 0x0800;       // POS enable
  int Status4         = I2C2->SR2;    // read status to clear flag
  while (!(I2C2->SR1 & 0x0004)) {};  // wait for ByteReceived    (BTF=1)
  I2C2->CR1          |= 0x0200;       // send STOP bit
  short x1 = I2C2->DR << 8;           // safe place
  short x2 = I2C2->DR;                // safe place

  return ((short)(x1 + x2));          // return combined bytes
}
```

Figure 9: A listing of the function to retrieve two bytes over I2C bus from the slave

The microcontroller needs initialization prior to the use of I2C block. The initialization includes mapping of the port pins to reach the I2C block, and the initialization of the control registers within the I2C block. The function to initialize is shown in Fig. 10.

```
void I2C2_Init (void)   {

  // declare and initialize pins to be used for I2C
  RCC->AHB1ENR  |=  0x00000002;      // Enable clock for GPIOB
  GPIOB->AFR[1] |=  0x00004400;      // select AF4 (I2C) for PB10,11 -> I2C2
  GPIOB->MODER  |=  0x00a00000;      // PB10,11 => alternate functions
  GPIOB->OTYPER |=  0x0c00;          // use open-drain output on these pins!

  // initialize I2C block
  RCC->APB1ENR  |= 0x00400000;       // Enable clock for I2C2
  I2C2->CR2     |= 0x0008;           // clock == 8MHz!
  I2C2->CCR     |= 0x0040;           // clock control register (270kHz)
  I2C2->TRISE   |= 0x0009;           // rise time register
  I2C2->CR1     |= 0x0001;           // I2C2 enable
}
```

Figure 10: A listing of the function to initialize the I2C block

By inspection of mapping table (Table 8, DM00037051, page 59) one can see that the I2C block number two can be accessed through port B, bits 10 (SCL) and 11 (SDA) if one selects alternate function 4 for these bits. Pins that correspond to these bits are conveniently connected to connector K470 on the test board. In order to map these pins to the block I2C, the port B block must first receive the clock (line 1, body of the function, Fig. 10), and then alternate functions 4 must be selected for bits 10 and 11. The use of alternate functions must be activated in the MODE register (line 3), and the outputs must be defined as “open-drain”, since the I^{2}C standard requires this type of driving for all lines I^{2}C.

The I2C block used also requires a clock, and the bit to enable it is located in register APB1ENR. The appropriate speed of transmission must be selected by selecting the basic clock frequency (8MHz in our case, as selected in the 6th line of the function body), and then selecting the clock division ratio (7th line, define the content of the Clock Control Register CCR). The clock is adjusted to the slowest device connected on the I^{2}C bus; in our case the slowest device runs at 400 kHz, but about 270 Khz clock speed is used. The allowed rise time of the signal at the bus is selected (register TRISE), and lastly the I^{2}C reception and transmission is enabled by setting the least significant bit in control register CR1.

A device for measuring the acceleration and the speed of rotation is used in this example. The device houses a local microcontroller and three independent accelerometers (xyz axis) combined with three independent gyro (the speed of rotation, XYZ axis) sensors. The result from each individual sensor is available as a “short” in two consecutive registers within the device. The program to read the sensor data is presented in Fig. 11

```
#include "stm32f4xx.h"
#include "LCD2x16.c"

void main ()  {

  // Init accelerometer chip
  I2C2_Init();                  // initialize I2C2 block
  I2C2_WriteChar(0x6b,0x00);    // wake-up!
  I2C2_WriteChar(0x1b,0x10);    // Gyro full scale = +/-1000 deg/s

  // Init LCD & prepare display
  LCD_init();                           // Init LCD
  LCD_string("A(z)=       mg", 0x00);   // prepare 1st row
  LCD_string("R(z)=       d/s", 0x40);  // prepare 2nd row

  // endless loop
  while (1) {
    short AccZ = I2C2_ReadShort(0x3f);  // read acc in z axis
    LCD_sInt16(AccZ / 16, 0x05, 1);     // write to LCD
    short RotZ = I2C2_ReadShort(0x47);  // read gyro in z axis
    LCD_sInt16(RotZ / 33, 0x45, 1);     // write to LCD
    for (int i=0; i<3000000; i++) {};   // waste time, ~100ms
  };
}
```

Figure 11: A listing of the program to read data from sensor using the I2C bus

The program starts with initialization of the I2C block within the microcontroller (a call to the function I2C2_Init) and the initialization of the device connected to the I2C bus (the two consecutive writes to the I2C device, see data on the gyro/accel device). The program then proceeds with the initialization of the LCD screen where the results will be written to, and enters the endless loop to periodically read results of measurement from the device and writes the results on the LCD screen.