I2C communication with the LM75 sensor

In this tutorial, we assume that the device is connected and returns already a meaningful temperature, as introduced in the previous section. We will in particular analyse in detail the sample code providing temperature measurements: Getting your first temperature measurements

We will provide more details about how the device is configured, linking the code with the relevant sections of the datasheet.

Most of the information provided here is based on the section 7 of the datasheet. Please glance through it, and make sure to have it visible while reading this page.

Device registers

Section 7.4 of the datasheet lists the four data registers present in the sensor (see table 5 in datasheet). The configuration register controls the different modes of operation of the device; you can read or write on it, although you would most write on it to set the desired behaviour of the sensor. The temp register contains the last temperature reading; it is read only, as expected. The other two registers contain the information needed to control the threshold temperatures, and will be further discussed in the next section of the tutorial.

All interactions with the device involve writing and reading the content of these registers, so this is what we will look at next.

Reading and writing on the registers

Each register has an address. There is a special register in the device called pointer register that sets which data register will be involved in the following reading or writing operation. We do not need to worry to much about the details, as communications will be handled by special functions in the mbed I2C library. But it is important to understand the sequence of typical I2C communications to be able to use properly these high level functions.

A typical sequence to write in the data registers consists in sending through I2C the device address (7 bits and R/W bit set to W), followed by the value of the pointer register, to indicate which data register we want to write on, and the data to store on this register (see figs 7 and 11 in the datasheet).

Note the the value of the data line (SDA) changes when the clock is low, and must not change when the clock (SCL) is high, as this is when it would be read. However there are two exceptions, which are particular signals to indicate the start and end of communications. To indicate a start of I2C communication, the master would take SDA from high to low while the SCL is high. To indicate the end of the communication, the master would take SDA from low to high while the SCL is high. You will spot these as START and STOP in figs 7 to 12.

Note also that data is only sent one byte at a time, followed by the acknowledgement bit.

To read, a similar precess is followed, but two steps are needed (See figs 8 and 11 in the datasheet). First, we send to the I2C bus the device address (7 bits and R/W bit set to W), followed by the value of the pointer register to indicate what we would want to read. The microcontroller would then send another start signal. The next part involves sending again to the I2C bus the device address, but this time with the R/W bit set to R. The master (microcontroller) would continue to control the clock, but this time the slave (sensor) would control the data line, and send, one byte at a time, the data requested.

I2C library functions

We will use here the write and read functions of the mbed library to performs a complete write/read transactions. Both functions handle the start, stop and acknowledgement signals for us.

Write to an I2C slave

```
int     write (int address, const char *data, int length, bool repeated=false)
```

Read from an I2C slave

```
int     read (int address, char *data, int length, bool repeated=false)
```

Let's look at what these parameters are, and shed some light on the sample code provided by the arm-mbed environment.

The address:

The mbed documentation says: "address: 8-bit I2C slave address [addr | 0]". The address should therefore be passed as 8 bits, including the 7 bits for the address, followed by the read/write bit (as the least significant bit). The value of this bit actually doesn't matter as it is overridden by the library. So how to determine this value from the datasheet?

From section 7.3 of the datasheet, we know that the seven bit address should be 1001000 (the last three bits depends on how you soldered the address pads on the chip). This corresponds to 72. To turn it into the 8 bits information needed for the mbed library, we need to add an extra bit at the end, that we can set to 0 or 1. Let's set it to 0: 10010000 = 144 in base 10, or 90 in hexadecimal. Hexadecimal numbers are typed with the prefix 0x, for instance 0x90 would be equivalent to 144. Hexadecimal numbers are commonly used to represent integer data covering 1 byte (2 hexadecimal digits) or 2 bytes (4 hexadecimal digits).

This is why in the code the address is defined as:

```
#define LM75_ADDR     (0x90) // LM75 address
```

The data buffer:

Whether we need to write or read, we need a bit of memory to handle this information. A byte array of the right size is therefore needed.

- To control the config register, we need two bytes, one to store the register pointer, and one of the register value.
- To write on any of the three temperature registers, we need three bytes, one for the register pointer, and two for the temperature value.
- To read any of the three temperature registers, we need to write one byte for the register pointer, and then read two for the temperature value.

The following lines in the sample code define the relevant buffers:

```
char data_write[2];
char data_read[2];
```

The buffers then need to be manipulated to contain the relevant information. This would set the value of the configuration buffer:

```
#define LM75_REG_CONF (0x01) // Configuration Register

data_write[0] = LM75_REG_CONF;
data_write[1] = 0x02;
```

Repeated start:

By default, the read and write commands would complete the transaction with the STOP signal (repeated=false). See for instance: .. code-block:: c

int status = i2c.write(LM75_ADDR, data_write, 2, 0);

However, to read data, we need two steps, one to indicate, with a write command which register we want to read, followed by a read. The write call should in this case be sent with repeated=true.

```
data_write[0] = LM75_REG_TEMP;
i2c.write(LM75_ADDR, data_write, 1, 1); // no stop
i2c.read(LM75_ADDR, data_read, 2, 0);
```

Returned values:

0 on success (ACK), non-0 on failure (NACK). The sample code uses this returned value to signal any error in the communication:

```
if (status != 0) { // Error
    while (1) {
        myled = !myled;
        wait(0.2);
    }
}
```

Converting the raw data into a temperature

Transforming data buffers into floating point temperature, and vice-versa, is a tricky task. You may not need to create such code, and could reuse the relevant sections of the examples code provided, but it helps to understand how they work.

The way temperatures are stored on the registers is defined in section 7.4.3 and 7.4.4. Have a look at it first. This is the content of the buffer data_read at the start:

data_read[0]								data_read[1]							
D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	0	0	0	0	0

These 11 bits represents the whole temperature range, with a 0.125 degree Celsius precision, i.e. 1/8 of a degree. The binary value of each bit, including sign, is detailed in the table below:

D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

Sign	64	32	16	8	4	2	1	1/2	1/4	1/8

The sign convention follows an approach called two's complement. Table 10 of the datasheet shows examples of temperature values and their equivalent representation in bits.

The gist of what follows consists in manipulating the bit array to extract the exact value of the temperature. You may want to learn a bit about bitwise operations in C++ if you never encountered this before.

The mbed example code for the LM75 sensor does something really complicated and long winded to build the temperature as a text. The appendix below explains what it does, but we are going to explain here a much simpler method.

The representation we get from the sensor, stored in data_read, is not too far from the representation of a 16-bit signed integer:

	Most significant byte								Least significant byte							
bit value for 16 bit int	Sign	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
bit value for sensor data	Sign	64	32	16	8	4	2	1	1/2	1/4	1/8	0	0	0	0	0

If we were to paste the 16 bits of data_read in a 16-bit integer, we would get a number that is the temperature scaled by a factor 256, since the bit corresponding to 1 celsius in the sensor data corresponds to 256 in the 16-bit int. This may be a good strategy to follow.

To use precisely defined integer types, we will use the header file stdint.h:

```
#include "stdint.h" //This allow the use of integers of a known width
```

To declare a 16-bit signed int called i16, we would type:

```
int16_t i16;
```

How to fill i16 with the relevant bits stored in data_read? This is where bitwise operations are handy!

```
i16 = data_read[0];
```

would create this :

								data_read[0]							
0	0	0	0	0	0	0	0	D10	D9	D8	D7	D6	D5	D4	D3

To place the bits D10 - D3 at the right place, we need to shift them bitwise using the left-shift operator “<<”:

```
int16_t i16 = data_read[0] << 8
```

data_read[0]															
D10	D9	D8	D7	D6	D5	D4	D3	0	0	0	0	0	0	0	0

To complete the number, we need to add the bits D2-D0 contained in data_read[1]. This is done using the bitwise OR operator, “|”, between data_read[0] << 8 and data_read[1].

```
int16_t i16 = (data_read[0] << 8) | data_read[1];
```

data_read[0]								data_read[1]							
D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	0	0	0	0	0

To get the temperature in degree Celsius, we need to divide this number by 256, making sure the output is a floating point number. To indicate to the compiler that we want the floating point division, we write 256 with a decimal point, 256.0. The conversion code therefore becomes:

```
int16_t i16 = (data_read[0] << 8) | data_read[1];
float temp = i16 / 256.0;
```

Overall, the code with the new conversion function would be:

```
#include "mbed.h"
#include "stdint.h" //This allow the use of integers of a known width

#define LM75_REG_TEMP (0x00) // Temperature Register
#define LM75_REG_CONF (0x01) // Configuration Register
#define LM75_ADDR     (0x90) // LM75 address

I2C i2c(I2C_SDA, I2C_SCL);

DigitalOut myled(LED1);

Serial pc(SERIAL_TX, SERIAL_RX);

int main()
{

        char data_write[2];
        char data_read[2];

        /* Configure the Temperature sensor device STLM75:
        - Thermostat mode Interrupt
        - Fault tolerance: 0
        */
        data_write[0] = LM75_REG_CONF;
        data_write[1] = 0x02;
        int status = i2c.write(LM75_ADDR, data_write, 2, 0);
        if (status != 0) { // Error
                while (1)
                {
                        myled = !myled;
                        wait(0.2);
                }
        }

        while (1)
        {
                // Read temperature register
                data_write[0] = LM75_REG_TEMP;
                i2c.write(LM75_ADDR, data_write, 1, 1); // no stop
                i2c.read(LM75_ADDR, data_read, 2, 0);

                // Calculate temperature value in Celcius
                int16_t i16 = (data_read[0] << 8) | data_read[1];
                // Read data as twos complement integer so sign is correct
                float temp = i16 / 256.0;

                // Display result
                pc.printf("Temperature = %.3f\r\n",temp);
                myled = !myled;
                wait(1.0);
        }

}
```

Comments regarding the sample code provided through the mbed compiler

Feel free at this stage to look again at the sample code provided with the mbed compiler: Getting your first temperature measurements

You will recognise similar operations to transform the buffer into a number. However, because the code uses int (32 bits by default) instead of int16_t, the sign bit is not at the right position, and the conversion has to be done carefully as a result.

Moreover, the mbed code only uses 9 bits on the data, as the shift “tempval >>= 7” destroys the values of D1 and D0, hence the 0.5 degree precision, most likely to ensure compatibility with older sensors operating with 9-bit precision.

Note that the mbed code creates the string array digit by digit rather than using the printf function. A string is an array of bytes representing text characters according to what is called the ascii table. The characters “0” to “9” corresponds to values 30 to 39 in hexadecimal representation. So “k + 0x30” represents the ascii value of the character corresponding to the digit value k, with 0<=k<=9.

We encourage you to use the method explained above (using the 16-bit integer) to record and display temperature data.

In the next (and final) section, you will be given a code to test the interrupt mode of the sensor.