**STM Lesson 121. Temperature, pressure and humidity sensor BME280. Part 1**

Posted on [June 7, 2018](http://narodstream.ru/stm-urok-121-datchik-temperatury-davleniya-i-vlazhnosti-bme280-chast-1/)by [http://1.gravatar.com/avatar/4824b24065500834db4b9f331b608833?s=32&d=mm&r=gNarod Stream](http://narodstream.ru/author/admin/) Posted in [I2C](http://narodstream.ru/i2c/) , [Programming STM32](http://narodstream.ru/rub_stm32/)- [2 comments ↓](http://narodstream.ru/stm-urok-121-datchik-temperatury-davleniya-i-vlazhnosti-bme280-chast-1/#comments)

[Programming LessonsLearn all the in-house cooking **programming** training ! Get the book on the e-mail!To learn moresheremetev.info](https://an.yandex.ru/count/LG5hM06khhm50Cq1CMV0ari00000ECgs7402I09Wl0Xe172WmvR10O01sE3CgmU80QE8tP0fa06cakID99W1eiM-tYEW0SAapOqag07iWR3U8xW1tAUPz1N00GBO0PQmhH3W0UQKqmxe0HRu0RQLthu1Y083e0BUkAe2kGAIvi4xTQnxAl02uT2oq0Ju0eA0W820a1EO0yy8g0CIi0C4k0J_0UW4tGFu1CDqY0N3T905qxa1e0Nx8gW5q1sm1T0Tk0MPBy05y9a7o0MmwW7G1QeDu0K_y0K1c0Q0qApp3g06xWAe1ku2oGR6AvDQIO1r4z46nh_8qKc0THFP1W00071n0000gGVuKIJDAuV5IB07W82G3D070k07XWhu1m60207G2BgAW870a802u0Yld9K1W0e1mGe00000003mFzWA0k0AW8bw-0h0_1M82mQg2n0oVHDd-5400BIGcK9RTGK0m0k0emN82u3Kam7P2_XH9CqhXyL8w0l3TF0B1eWCjfNUlW7e30AO3VYSPV8D0FeD088E08aE00000000y3-G3i24FPWEnjVCr9M9uBeJe0x0X3sm3W6X3m0000000F0_g0_ueu7bhENXuaW0?stat-id=3&test-tag=84112668909569&format-type=24&banner-test-tags=eyI0ODQ2MjkwMDY3IjoiODQxMTI2Mzk1NTc2MzIifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

[Evaluation BoardsUnique collection: selection by parameters, prices, detailed descriptions.To learn morerlocman.ru](https://an.yandex.ru/count/G08_eeofsa050Bq1CMV0ari00000ECgs7402I09Wl0Xe172uqyEm0O01XgYg68W1iCodh12G0SommVO6c06swhFd2RW1XjFLdWV00GBO0OZEW0RW0UxRsGde0HJu0G680WIW0k03kGAIvi4xTQnxAl02lOgvdmBu0eA0W820a1E00-3amQiEY0EAkz7h2x031BW4_m7e1847-0ION8W5c5oG1UjSe0NK6wW5k1Um1RWNk0ND5y05iBW1o0NmBj05nWhW1MBm1G6O1iwlg6gW1ku2g0Rk0ia6nYkJMac0THFH1iQ_oD59W7KJsGO0001mSG000907Aga7-54a5qk7nKYm1u20a0pG1mBW1uOA-0S1W0W1q0YQYe21m9200k08y0602W712W0000000F0_s0e2u0g0YNhu2i3y5OWB2AeB45OrD7BuKG00_MAXGbjr1G302u2Z1SWBWDIJ0TaB-54a5qk7nKZe2vXSy0i6Y0oQvDw-0UWC0fWD_BrcyWq0-Wq0WWu0YGu00000003mFv0Em8Gzc0x6rypKbOdWkXEW3i24FR0E0Q4F00000000y3-e3_YZWUAYwU7YI000?stat-id=4&test-tag=84112668909569&format-type=24&banner-test-tags=eyI5Mjk1NzM5MjYiOiI4NDExMjYzOTUyNDg2NCJ9&)[Yandex.Direct](https://direct.yandex.ru/?partner)

In this lesson we will, according to numerous requests, consider the sensor from **Bosch Sensortec** - **BME280** .

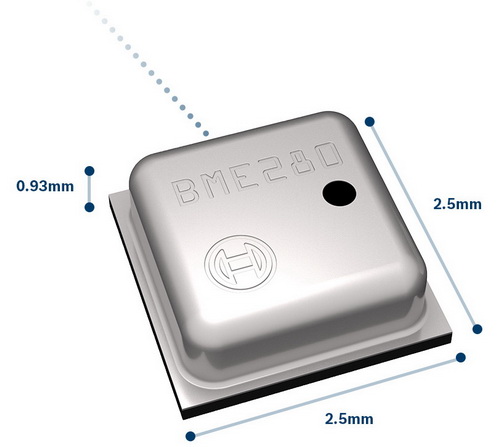
This sensor measures several very important environmental values ​​- temperature, air humidity, and atmospheric pressure.

There is also a similar sensor - BMP280, which also measures these values, but only in addition to humidity. But it is much cheaper.

But since the requests were just to consider the sensor **BME280** , then we will do it.

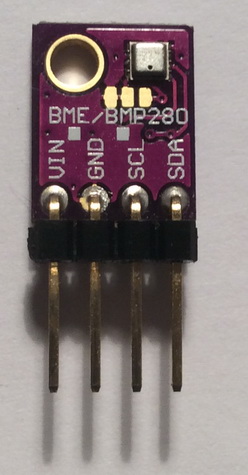
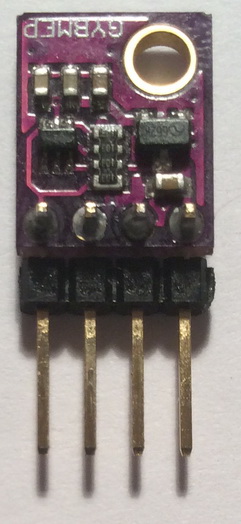
In addition to the fact that the sensor measures all these above mentioned values, it also manages to filter the noise, it has a very fine calibration of the readings, at the same time, it does all this work brilliantly and very quickly. Also, unlike previous similar sensors we have examined, this sensor performs all these measurements not only quickly, but also in wider ranges. For example, many sensors do not know how to measure humidity less than 20 percent.

The sensor looks like this (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2018/05/stm121img00.jpg)

And, before starting to program and connect this sensor, we briefly get acquainted with its main characteristics. If someone is interested to know in detail, then all the information is on the official website of [**Bosch Sensortec**](http://www.bosch-sensortec.com/) .

Until we start with how to connect this sensor to the controller. It can be connected either via the SPI bus or via the I2C bus. We will connect it via the I2C bus, since I got a module with this sensor in my hands, which is connected only through I2C and the SPI interface in it is not output from the sensor to the external contacts. This module looks like this

On the SPI bus, the sensor can be clocked at a frequency of up to 10 megahertz, and on the I2C bus it can be up to 3.4 megahertz

This module can be powered from 1.71 volts to 3.6 volts.

The sensor I / O port is powered in the range 1.2 - 3.6 V (there is a separate power supply for this purpose).

Sensor consumption current:

when measuring humidity and temperature at a sampling frequency of 1 hertz - 1.8 μA,

when measuring pressure and temperature at a sampling frequency of 1 hertz - 2.8 μA,

when measuring pressure, humidity and temperature at a sampling frequency of 1 hertz - 3.6 μA,

in sleep mode, 0.1 μA.

We see that such currents will occur only when sampling 1 hertz, without oversampling (oversampling). And if we apply any filters, then the output will have a completely different frequency, because of the filter coefficients. Therefore, in this case, these consumption currents will be observed even at a lower measurement frequency.

Measuring ranges: temperatures - -40 ... +85 ° C, humidity - 0 ... 100%, pressure - 300 ... 1100 hPa.

Of course, in different ranges, the measurement accuracy will be different.

For example, the humidity measurement accuracy of ± 3% RH is only provided in the range of 20 ... 80% RH, and the accuracy of the pressure measurement ± 1.0 hPa is in the range 300 ... 1100 hPa (that is, in the whole range). Temperature measurement accuracy ± 0.5 ° C (at 25 ° C),

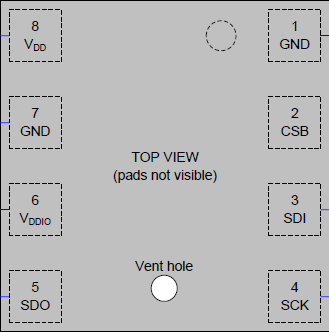
The sensor sounds as follows:

Humidity: 0.07% - without oversampling, with maximum oversampling the noise is reduced to 0.02%.

Pressure: 1.3 Pa - without the use of oversampling and filters, with the maximum oversampling and filtering reduced to 0.2 Pa.

Temperature: 0.005 ° C - without the use of oversampling and filters, with the maximum oversampling and filtering reduced to 0.002 ° C.

Now let's look at the purpose of the sensor contacts



**Vdd** - analog power;

**GND** - common wire;

**Vddio** - powering the digital path;

**SDO** - digital signal output when using 4-wire SPI mode, when using I2C mode, the address bus (if connected to a common wire, then the logical zero in the address zero bit, if the power supply is logical 1);

**GND** - common wire;

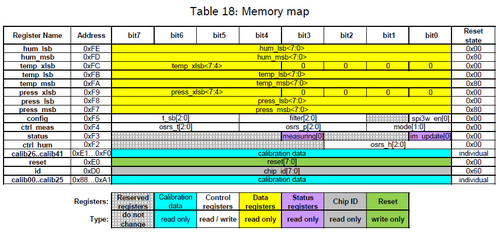
**CSB** - CS contact (chip select) when using SPI mode;

**SDI** - digital signal input when using 4-wire SPI mode, digital input and output when using 3-wire SPI mode and I2C mode;

**SCK** is a digital signal clock bus.

Also, the sensor has a lot of different registers, with which we will get acquainted in detail while writing the code.

That's how the memory is organized in the sensor (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2018/05/stm121img04.png)

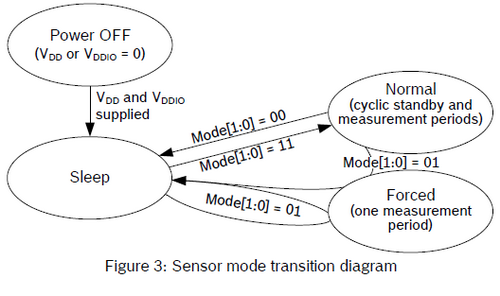
It should also be noted that the sensor can be in three modes: in normal mode ( **NORMAL MODE** ), in sleep mode ( **SLEEP MODE** ) and in forced mode ( **FORCE MODE** ).

Well, with the sleeping regime everything is clear. And what is the difference between normal and forced?

But this is what.

In normal mode, after reading the values ​​from the sensor, the latter remains in this mode and further, but in the compulsory mode, when we consider the values, the sensor automatically enters sleep mode. And, to us again to read the indicators, we need to re-initialize either normal or forced modes.

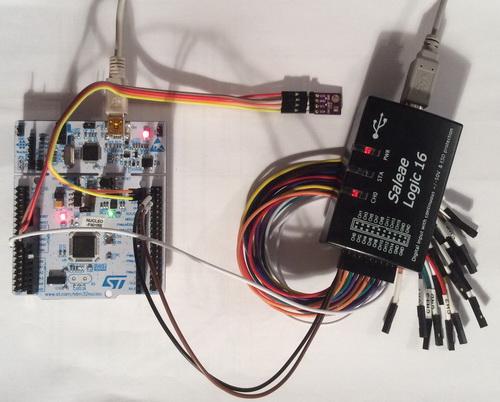
This chart shows us the best way



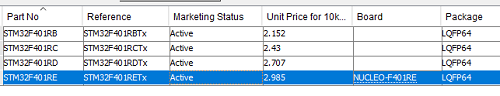
Now the connection diagram.

We will connect the sensor module to the **Nucleo STM32F401RE board** . Also to the information contacts I2C we connect a logic analyzer to see how our sensor will respond to commands. Then, once the reception is adjusted, we also connect the LCD2004 character display to the I2C board as well.

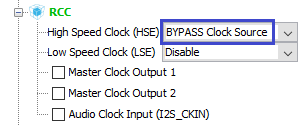
This is how our scheme looks without a display (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2018/05/stm121img05.jpg)

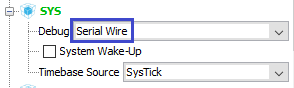
Create a new project in the Cube MX and select the controller



Turn on the controller timing



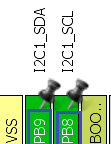
Then turn on the debugger



Turn on the I2C bus to connect the sensor

http://narodstream.ru/wp-content/uploads/2018/05/stm121img09.png

Override the legs for easy connection of the sensor

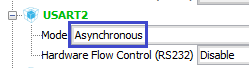


Let's at the same time immediately turn on the bus and connect the display

http://narodstream.ru/wp-content/uploads/2018/05/stm121img11.png

For this bus, we do not override anything, the feet remain the default.

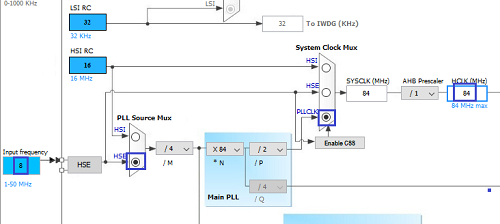
We also include the USART bus to view the debugging information, but we still do not need any wires or adapters, USART works through the debug port



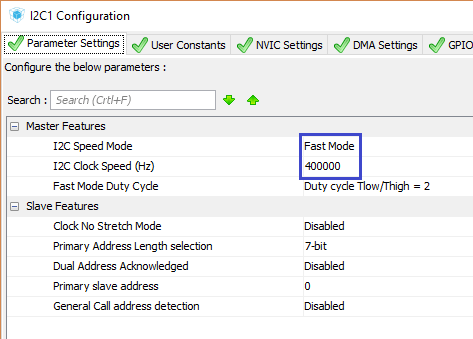
We also turn on the output of the LED leg



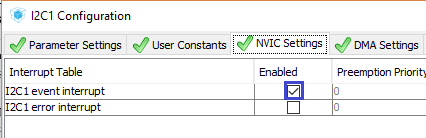
Go to **System Configuration** and configure the following frequencies (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2018/05/stm121img14.png)

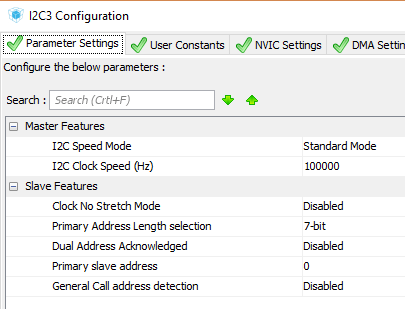
Go to **Configuration** and configure **I2C** for the sensor



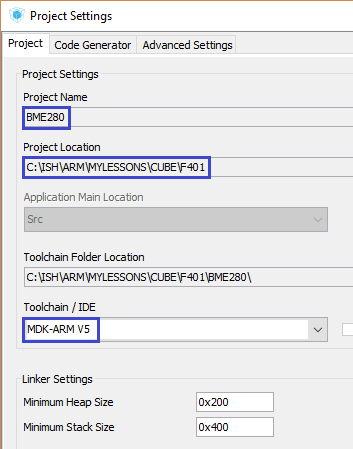
Just in case we turn on the interrupts, suddenly come in handy



Let's go into the settings of another I2C bus and make sure everything is right there



Let's go into the project settings, select the programming environment, the project folder and call our project



Save the project settings, generate the project for Keil, open it in it, then configure the programmer for auto-reload, set the optimization level to 1 and try to assemble the project.

If everything is going well, then we will continue.

Open the file **main.c** and connect the library to work with the lines

/\* USER CODE BEGIN Includes \*/

**#include <string.h>**

/\* USER CODE END Includes \*/

Let's add a global string array

/\* Private variables ---------------------------------------------------------\*/

**char str1[100];**

/\* USER CODE END PV \*/

Create a module for our new library working with the sensor in the form of two files **BME280.h** and **BME280.c the** following contents

**BME280.h** :

**#ifndef BME280\_H\_**

**#define BME280\_H\_**

**//------------------------------------------------**

**#include "stm32f4xx\_hal.h"**

**#include <string.h>**

**#include <math.h>**

**//------------------------------------------------**

**//------------------------------------------------**

**#endif /\* BME280\_H\_ \*/**

**BME280.c** :

**#include "BME280.h"**

**//------------------------------------------------**

**extern I2C\_HandleTypeDef hi2c1;**

**extern UART\_HandleTypeDef huart2;**

**extern char str1[100];**

**//------------------------------------------------**

We connect the module in the file **main.c**

#include <string.h>

**#include "BME280.h"**

In the [**next part of the**](http://narodstream.ru/stm-urok-121-datchik-temperatury-davleniya-i-vlazhnosti-bme280-chast-2/) lesson, we'll start writing the sensor initialization, in which we count the identifier, the calibration data from the sensor, and also configure the timing for the Standby mode.

**STM Lesson 121. Temperature, pressure and humidity sensor BME280. Part 2**

Posted on [June 8, 2018](http://narodstream.ru/stm-urok-121-datchik-temperatury-davleniya-i-vlazhnosti-bme280-chast-2/)by [http://1.gravatar.com/avatar/4824b24065500834db4b9f331b608833?s=32&d=mm&r=gNarod Stream](http://narodstream.ru/author/admin/) Posted in [I2C](http://narodstream.ru/i2c/) , [Programming STM32](http://narodstream.ru/rub_stm32/)- [No Comments ↓](http://narodstream.ru/stm-urok-121-datchik-temperatury-davleniya-i-vlazhnosti-bme280-chast-2/#respond)

[Programming LessonsLearn all the in-house cooking **programming** training ! Get the book on the e-mail!To learn moresheremetev.info](https://an.yandex.ru/count/AB_nG4sFlsq50D01COx0ari00000ECgs7402I09Wl0Xe172WmvR10O01sE3CgmU80QE8tP0fa06cakID99W1eiM-tYEW0SAapOqag07iWR3U8xW1tAUPz1N00GBO0PQmhH3W0UQKqmxe0HRu0RQLthu1Y083e0BUkAe2kG9KDpJM3C-LVF02qC7Wp0pu0eA0W820a1AO0wKVg0CIi0C4k0J_0UW4X0Fu1D_70OW5tyS1a0Mhu06W1VOxg0NaCh05v3Au1OGvm0NmcGV81R3g0T05xHRW1J_m1G6O1e3GhFCEe0Rk0gW6xWB91iOharf9W7KJqGR6lyZHIO1r4za60000S740002f1_XHV2gHaCL8i0U0W90Cq0S2u0U62lW70O080T08keg0WS2GW0BW2A-SbG602W712W0000000F0_s0e2u0g0YNhu2i3y5OWB1geB48KpnMRuKG003UZoH5jr1G302u2Z1SWBWDIJ0TaB-55yAf6GnKZe2z_70V0B1eWCjfNUlW7e30AO3VYSPV8D0FeD088E08aE00000000y3-G3i24FPWEnjVCr9M9uBeJe0x0X3sm3W6X3m0000000F0_g0_ueu7eWvBauaW0?stat-id=3&test-tag=79714622398465&format-type=24&banner-test-tags=eyI0ODQ2MjkwMDY3IjoiNzk3MTQ1OTMwNDY1MjgifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

[Buy robot constructorTeach your child to program. 6900 for a set of shares. OrderTo learn moredorado-uno.ru](https://an.yandex.ru/count/9VF5BddzqKm50DO1CO_0ari00000ECgs7402I09Wl0Xe172AffoG1e01xCtW6OW1rBMagJMG0VZAnCelc07gZ-3LBg01zih4oY-e0UYEuDKkk07st_Rr6S010jW1ieMB5U01qBto4UW1OFW10OW21A02eBxP5ha2L3SqrWpFbNp80f-mqOsbyToZLF02xBsE9_W2We20W82G4e03-8tKHuW3qwcLfmUm0mJe1943-0Imsm681R3R0P05pCm2e0NeGQW5upUm1UCtk0N7LC05iFiCo0NmfGFG1OmPu0K9y0K1c0RUgOuPe0Rk0gW6xWB91iOharf9W7KJqGR6lyZHIO1r4za6000jS740002f1_XHyEwnaCL8i0U0W90Cq0S2u0U62lW70O080T08keg0WS2GW0BW293Vh0k02W712W0000000F0_s0e2u0g0YNhu2i3y5OWB2AeB4AvtI0duKG00IHdxH5jr1G302u2Z1SWBWDIJ0TaB-57mxh6GnKZe2x3R0V0B1eWCelhUlW7e30AO3ScoP_8D0FeD088E08aE00000000y3-G3i24FPWEnjVCr9M9uBeJe0x0X3sm3W6X3m0000000F0_g0_ue-7_aPRauaW0?stat-id=4&test-tag=79714622398465&format-type=24&banner-test-tags=eyI2Mzg2Mzg5NjkyIjoiNzk3MTQ1OTMwNDY1MjgifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

In the [**previous part of the**](http://narodstream.ru/stm-urok-121-datchik-temperatury-davleniya-i-vlazhnosti-bme280-chast-1/) lesson we got acquainted with this sensor, studied its characteristics, connection methods, its properties, and also created and configured a project for its programming.

In the **BME280.h** file, **we** add macros for the LED

#include <math.h>

**//------------------------------------------------**

**#define LED\_GPIO\_PORT GPIOA**

**#define LED\_PIN GPIO\_PIN\_5**

**#define LED\_ON HAL\_GPIO\_WritePin(LED\_GPIO\_PORT, LED\_PIN, GPIO\_PIN\_SET)**

**#define LED\_OFF HAL\_GPIO\_WritePin(LED\_GPIO\_PORT, LED\_PIN, GPIO\_PIN\_RESET)**

**#define LED\_TGL HAL\_GPIO\_TogglePin(LED\_GPIO\_PORT, LED\_PIN)**

**//------------------------------------------------**

Go to the file **BME280.c** and add the error function

extern char str1[100];

**//------------------------------------------------**

**void Error(void)**

**{**

**LED\_OFF;**

**}**

**//------------------------------------------------**

Since there is only one control LED on the board, we will extinguish it on error.

After this function, add the initialization function of the sensor, in which we first light our LED

**//------------------------------------------------**

**void BME280\_Init(void)**

**{**

**uint8\_t value=0;**

**uint32\_t value32=0;**

**LED\_ON;**

**}**

**//------------------------------------------------**

Also, after the error function, we will add write and read functions to a specific byte address for the I2C bus

**//------------------------------------------------**

**static void I2Cx\_WriteData(uint16\_t Addr, uint8\_t Reg, uint8\_t Value)**

**{**

**HAL\_StatusTypeDef status = *HAL\_OK*;**

**status = HAL\_I2C\_Mem\_Write(&hi2c1, Addr, (uint16\_t)Reg, I2C\_MEMADD\_SIZE\_8BIT, &Value, 1,0x10000);**

**if(status != *HAL\_OK*) Error();**

**}**

**//------------------------------------------------**

**static uint8\_t I2Cx\_ReadData(uint16\_t Addr, uint8\_t Reg)**

**{**

**HAL\_StatusTypeDef status = *HAL\_OK*;**

**uint8\_t value = 0;**

**status = HAL\_I2C\_Mem\_Read(&hi2c1, Addr, Reg, I2C\_MEMADD\_SIZE\_8BIT, &value, 1, 0x10000);**

**if(status != *HAL\_OK*) Error();**

**return value;**

**}**

**//------------------------------------------------**

We use these functions constantly, so they do not need an explanation.

Below we also add two more similar functions for writing two-byte and three-byte values

**//------------------------------------------------**

**static void I2Cx\_ReadData16(uint16\_t Addr, uint8\_t Reg, uint16\_t \*Value)**

**{**

**HAL\_StatusTypeDef status = *HAL\_OK*;**

**status = HAL\_I2C\_Mem\_Read(&hi2c1, Addr, Reg, I2C\_MEMADD\_SIZE\_8BIT, (uint8\_t\*)Value, 2,0x10000);**

**if(status != *HAL\_OK*) Error();**

**}**

**//------------------------------------------------**

**static void I2Cx\_ReadData24(uint16\_t Addr, uint8\_t Reg, uint32\_t \*Value)**

**{**

**HAL\_StatusTypeDef status = *HAL\_OK*;**

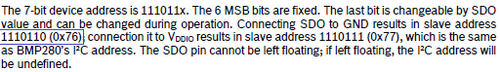
**status = HAL\_I2C\_Mem\_Read(&hi2c1, Addr, Reg, I2C\_MEMADD\_SIZE\_8BIT, (uint8\_t\*)Value, 3,0x10000);**

**if(status != *HAL\_OK*) Error();**

**}**

**//------------------------------------------------**

Now go to the header file BME280.h and write a macro for the address of our sensor, by which it will respond to the write and read commands, after learning it in the technical documentation



As we already, I hope, we all know, we write the address shifted to the left by 1 position

#define LED\_TGL HAL\_GPIO\_TogglePin(LED\_GPIO\_PORT, LED\_PIN)

//------------------------------------------------

**#define BME280\_ADDRESS 0xEC //BME280 I2C ADDRES (0x76<<1)**

//------------------------------------------------

Back in the file BME.h and after the function **I2Cx\_ReadData24** we add the function of writing the byte to the register

**//------------------------------------------------**

**void BME280\_WriteReg(uint8\_t Reg, uint8\_t Value)**

**{**

**I2Cx\_WriteData(BME280\_ADDRESS, Reg, Value);**

**}**

**//------------------------------------------------**

We add similar functions in order to not enter the address of the device every time you call.

Add a similar function to read a byte from the register

**//------------------------------------------------**

**uint8\_t BME280\_ReadReg(uint8\_t Reg)**

**{**

**uint8\_t res = I2Cx\_ReadData(BME280\_ADDRESS,Reg);**

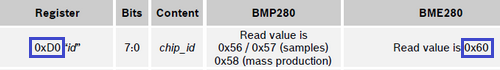
**return res;**

**}**

**//------------------------------------------------**

Well, by tradition, we will read the device identifier to make sure that we have such a device, and that it normally exchanges data on the **I2C** bus .

Find the register address with the ID in the documentation, at the same time there and see the identifier itself



Zaydom file **BME280.h** and write macros to register and ID values

#define BME280\_ADDRESS 0xEC //BME280 I2C ADDRES (0x76<<1)

**#define BME280\_REG\_ID 0xD0 //BME280 ID REGISTER**

**#define BME280\_ID 0x60 //BME280 I2C ID**

Back in the file **BME280.c** and in the initialization function, we try to read the identifier from the sensor

LED\_ON;

**value = BME280\_ReadReg(BME280\_REG\_ID);**

Create a prototype of the sensor initialization function and call it in the function **main ()** the file **main.c**

/\* USER CODE BEGIN 2 \*/

**BME280\_Init();**

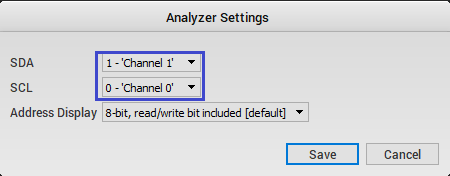
/\* USER CODE END 2 \*/

We will collect the project and we will sew the controller.

We see that the LED is still on, so no error has occurred, it's already good



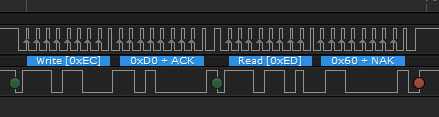
Run the logical analysis program and configure the I2C bus



Also, configure channel 0 trigger on the positive front

http://narodstream.ru/wp-content/uploads/2018/05/stm121img23.png

Click on "Start" and reboot the controller. We obtain the following



Excellent! We have returned the correct ID.

Let's return to the BME280.c file in the initialization function and write the validation check, not always the logical analysis to do. At the same time we will display our ID in the terminal program

value = BME280\_ReadReg(BME280\_REG\_ID);

**sprintf(str1, "rnrnID: 0x%02Xrn", value);**

**HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);**

**if(value !=BME280\_ID)**

**{**

**Error();**

**return;**

**}**

Now after assembling the code and firmware controller we have to watch this in the terminal program

http://narodstream.ru/wp-content/uploads/2018/05/stm121img25.png

Next, we must perform a software reset of the sensor, for this purpose there is such a register

http://narodstream.ru/wp-content/uploads/2018/05/stm121img26.png

To reboot the sensor, we must put a strictly defined number - 0xB6 - at this address.

Therefore, go to the header file **BME280.h** and add the corresponding macros there

#define BME280\_ID 0x60 //BME280 I2C ID

**#define BME280\_REG\_SOFTRESET 0xE0 //BME280 SOFT RESET REGISTER**

**#define BME280\_SOFTRESET\_VALUE 0xB6 //BME280 SOFT RESET VALUE**

We return to the **BME280.c** file in the initialization function and reboot our sensor, and also wait for it to reset accordingly

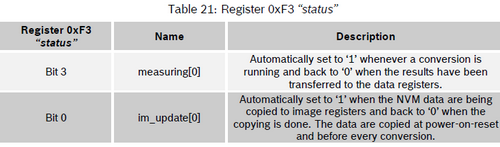
  return;

}

**BME280\_WriteReg(BME280\_REG\_SOFTRESET,BME280\_SOFTRESET\_VALUE);**

Next, we have two ways to wait for the sensor to reboot. Either wait for a while with a margin, or something to track. Let's go on the second way. He is more professional. We consider the status register and find out if we are ready to continue working with our sensor.

Let us turn to the documentation for this



We see that in this register there are only 2 working bits. We are now interested in the second.

Let's **go** to the header file **BME280.h** and add the corresponding macros there

#define BME280\_SOFTRESET\_VALUE 0xB6 //BME280 SOFT RESET VALUE

**#define BME280\_REGISTER\_STATUS 0XF3 //BME280 STATUS REGISTER**

**#define BME280\_STATUS\_MEASURING 0X08 //Running conversion**

**#define BME280\_STATUS\_IM\_UPDATE 0X01 //NVM data copying**

Let's return to the file **BME280.c** and add the function of reading the status above the initialization function

**//------------------------------------------------**

**uint8\_t BME280\_ReadStatus(void)**

**{**

**//clear unuset bits**

**uint8\_t res = BME280\_ReadReg(BME280\_REGISTER\_STATUS)&0x09;**

**return res;**

**}**

**//------------------------------------------------**

Together with the reading of the value from the register, we use the mask to zero the nonexistent bits

Now, in the initialization function, we wait for the desired state of the zero bit

BME280\_WriteReg(BME280\_REG\_SOFTRESET,BME280\_SOFTRESET\_VALUE);

**while (BME280\_ReadStatus() & BME280\_STATUS\_IM\_UPDATE) ;**

Next, we need to count the calibration cells and distribute them in terms of variables, in order to later use it to correct the sensor readings. How it is done, is written in the code example in the technical documentation for the sensor. Therefore, we will not go into details of the cells.

Let's **go** to the header file **BME280.h** and add there the structure for the calibration values

#define BME280\_STATUS\_IM\_UPDATE 0X01 //NVM data copying

//------------------------------------------------

**typedef struct**

**{**

**uint16\_t dig\_T1;**

**int16\_t dig\_T2;**

**int16\_t dig\_T3;**

**uint16\_t dig\_P1;**

**int16\_t dig\_P2;**

**int16\_t dig\_P3;**

**int16\_t dig\_P4;**

**int16\_t dig\_P5;**

**int16\_t dig\_P6;**

**int16\_t dig\_P7;**

**int16\_t dig\_P8;**

**int16\_t dig\_P9;**

**uint8\_t dig\_H1;**

**int16\_t dig\_H2;**

**uint8\_t dig\_H3;**

**int16\_t dig\_H4;**

**int16\_t dig\_H5;**

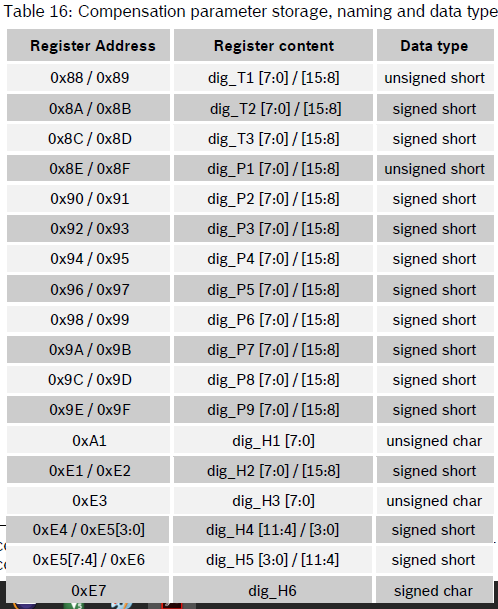
**int8\_t dig\_H6;**

**} BME280\_CalibData;**

**//------------------------------------------------**

All the calibration values ​​of different types, so we need the functions to read different types of data from the corresponding registers, we will necessarily add such functions later.

Let's see these registers and how exactly the data is stored there.



The data in these registers is stored in the correct form (the lowest byte forward), with a rare exception, so we do not have to change the bytes.

Also in the header file, we'll add register data macros

#define BME280\_STATUS\_IM\_UPDATE 0X01 //NVM data copying

**//------------------------------------------------**

**#define BME280\_REGISTER\_DIG\_T1 0x88**

**#define BME280\_REGISTER\_DIG\_T2 0x8A**

**#define BME280\_REGISTER\_DIG\_T3 0x8C**

**#define BME280\_REGISTER\_DIG\_P1 0x8E**

**#define BME280\_REGISTER\_DIG\_P2 0x90**

**#define BME280\_REGISTER\_DIG\_P3 0x92**

**#define BME280\_REGISTER\_DIG\_P4 0x94**

**#define BME280\_REGISTER\_DIG\_P5 0x96**

**#define BME280\_REGISTER\_DIG\_P6 0x98**

**#define BME280\_REGISTER\_DIG\_P7 0x9A**

**#define BME280\_REGISTER\_DIG\_P8 0x9C**

**#define BME280\_REGISTER\_DIG\_P9 0x9E**

**#define BME280\_REGISTER\_DIG\_H1 0xA1**

**#define BME280\_REGISTER\_DIG\_H2 0xE1**

**#define BME280\_REGISTER\_DIG\_H3 0xE3**

**#define BME280\_REGISTER\_DIG\_H4 0xE4**

**#define BME280\_REGISTER\_DIG\_H5 0xE5**

**#define BME280\_REGISTER\_DIG\_H6 0xE7**

**//------------------------------------------------**

Let's return to the file **BME280.c** and add a global variable of the type of our structure

extern char str1[100];

**BME280\_CalibData CalibData;**

After the function  **BME280\_ReadReg** we add two functions for reading double-byte values ​​from sensor registers - one for unsigned, one for signed

**//------------------------------------------------**

**void BME280\_ReadReg\_U16(uint8\_t Reg, uint16\_t \*Value)**

**{**

**I2Cx\_ReadData16(BME280\_ADDRESS,Reg,Value);**

**}**

**//------------------------------------------------**

**void BME280\_ReadReg\_S16(uint8\_t Reg, int16\_t \*Value)**

**{**

**I2Cx\_ReadData16(BME280\_ADDRESS,Reg, (uint16\_t\*) Value);**

**}**

**//------------------------------------------------**

After the function of reading the status  **BME280\_ReadStatus** we add the function of reading the calibration cells and distributing them along the structure fields

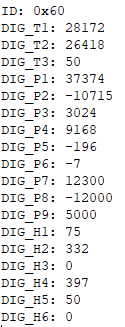
**BME280\_ReadCoefficients**

Call this function in the initialization function

while (BME280\_ReadStatus() & BME280\_STATUS\_IM\_UPDATE) ;

**BME280\_ReadCoefficients();**

We will collect the code, we will sew the controller and in the terminal program we will look at the calibration values ​​read from the sensor

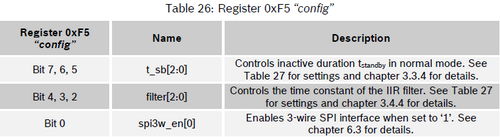


You, accordingly, will have different meanings.

Set the time of the STANDBY mode.

The sensor goes into this mode after making the measurements and there is a certain time in it, which we now will adjust by including certain bits in the CONFIG register.

Here is a description of the groups of its bits

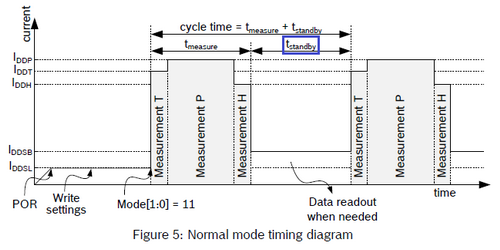


The first group (bits 7: 5) is the settings for the duration of the inactive mode (STANDBY) in normal operation. This is the setting of our STANDBY.

Bits 4: 2 - settings for IIR filter timings.

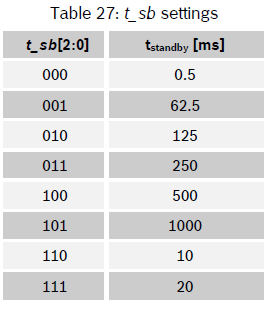
Bit 0 - Enables the 3-wire SPI mode.

Here is a diagram of the sensor's operation in normal mode, which perfectly illustrates how the sensor goes to and returns from STANDBY. While in this state, if necessary, we read the data. Also here you can see at what time the current consumes the sensor.



Therefore, if we do not need to read data often, then it makes sense for a smaller consumption to set this timing to the maximum value.

Let's look at the assignment of the bits of the configuration register responsible for this timing



Let's **go** to the header file **BME280.h** and add there the macro for the configuration register

#define BME280\_STATUS\_IM\_UPDATE 0X01 //NVM data copying

**#define BME280\_REG\_CONFIG 0xF5 // Configuration register**

Also add a mask and settings for the timing bits of the STANDBY mode

#define BME280\_REGISTER\_DIG\_H6 0xE7

//------------------------------------------------

**#define BME280\_STBY\_MSK 0xE0**

**#define BME280\_STBY\_0\_5 0x00**

**#define BME280\_STBY\_62\_5 0x20**

**#define BME280\_STBY\_125 0x40**

**#define BME280\_STBY\_250 0x60**

**#define BME280\_STBY\_500 0x80**

**#define BME280\_STBY\_1000 0xA0**

**#define BME280\_STBY\_10 0xC0**

**#define BME280\_STBY\_20 0xE0**

Back in the file **BME280.c** and add the function of setting the timing of the STANDBY mode after the function **BME280\_ReadCoefficients**

**//------------------------------------------------**

**void BME280\_SetStandby(uint8\_t tsb) {**

**uint8\_t reg;**

**reg = BME280\_ReadReg(BME280\_REG\_CONFIG) & ~BME280\_STBY\_MSK;**

**reg |= tsb & BME280\_STBY\_MSK;**

**BME280\_WriteReg(BME280\_REG\_CONFIG,reg);**

**}**

**//------------------------------------------------**

Call this function in the sensor initialization function by setting the maximum time

BME280\_ReadCoefficients();

**BME280\_SetStandby(BME280\_STBY\_1000);**

In the [**next part of the**](http://narodstream.ru/stm-urok-121-datchik-temperatury-davleniya-i-vlazhnosti-bme280-chast-3/) lesson we will configure the filter and oversampling, and also consider the value of the air temperature from the sensor.

**STM Lesson 121. Temperature, pressure and humidity sensor BME280. Part 3**

Posted on [June 14, 2018](http://narodstream.ru/stm-urok-121-datchik-temperatury-davleniya-i-vlazhnosti-bme280-chast-3/)by [http://1.gravatar.com/avatar/4824b24065500834db4b9f331b608833?s=32&d=mm&r=gNarod Stream](http://narodstream.ru/author/admin/) Posted in [I2C](http://narodstream.ru/i2c/) , [Programming STM32](http://narodstream.ru/rub_stm32/)- [No Comments ↓](http://narodstream.ru/stm-urok-121-datchik-temperatury-davleniya-i-vlazhnosti-bme280-chast-3/#respond)

[Programming for beginnersDownload the free book and become a real programmer. Get it now:To learn moresheremetev.infoYandex.Direct](https://an.yandex.ru/count/K2blVBXIpkC50EK1CQ30ari00000ECgs7402I09Wl0Xe172WmvR10O01vE3CgmU80VY6tP0fa06YakID99W1Zhs-tYEW0VwIp8qag06M_w_U8xW1XgMPz1N00GBO0PQmhH3W0UQKqmxe0HRu0G680WEW0jwugWAv0YLfkJJTcLbRo0ASiTFLhyYrw2xm0gtetky3-0A2W820WF0IW0FuZTH7Y0FJgPMd1vW3oeu4g0CIi0C4k0J_0UW4x0Fu1BM_0OW5jRy1a0NCamAW1ROvg0NLCB05rJ2u1OX6m0NmcGV81R3g0T05ynNW1Gdm1G6O1e3GhFCEe0Rk0gW6xWB91iOharf9W7KJqGR6lyZHIO1r4za6000jS740002f1_XHwC_BbCL8i0U0W90Cq0S2u0U62lW70O080T08keg0WS2GW0BW2A-SbG602W712W0000000F0_s0e2u0g0YNhu2i3y5OWB1geB43dMiXBuKG00Qv41Hbjr1G302u2Z1SWBWDIJ0TaB-57epykKnKZe2xM_0V0B1eWCjfNUlW7e30AO3VYSPV8D0FeD088E08aE00000000y3-G3i24FPWEnjVCr9M9uBeJe0x0X3sm3W6X3m0000000F0_g0_uex78gPdbuaW0?stat-id=3&test-tag=83014231023617&format-type=24&banner-test-tags=eyI0ODQ2MjkwMDY1IjoiODMwMTMxMjc5Mjk4NTYifQ%3D%3D&)

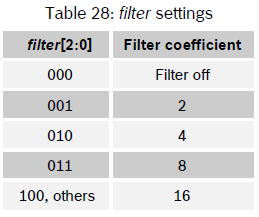
[Switches contactlessSwitches **contactless capacitive** type VBEYandex.Direct](https://an.yandex.ru/count/EweEkjoWCNi50Bq1CQ70ari00000ECgs7402I09Wl0Xe172Osh300u01zl2S28W1oeMH6901nCMv19W1pfVW1BW1fFFV1C010jW1-9uxu06QWEG2w04s-04EY084e0BanGcv0YLfkJJTcLbRy0A8uBAo1FW2We20W83m4e03vPIEy0g80ylfez4Ec0EqZmom0mJe1F01-0IygG681Rof0P05chi3e0NiCgW5Yokm1Oihk0NNSC05wEOAo0N8tmBG1R4Ju0LQy0K1c0RCrlP_e0Rk0gW6xWB91iOharf9W7KJqGR6lyZHIO1r4za60000S740002f1_XHK5dgbCL8i0U0W90Cq0S2u0U62lW70O080T08ceg0WS2GW0BW2FBY180A0S4A00000000y3_O2WBW2e29UlWAmFmLY0i8gWiGWK9bfVXH002qCmb6MtK50C0BWAC5o0k0r9C1sGluKL1PwfJ5IEWBlAa1y0i6Y0oE_Tw-0UWC0fWDoR9dyWq0-Wq0WWu0YGu00000003mFv0Em8Gzc0x6rypKbOdWkXEW3i24FR0E0Q4F00000000y3-e3_YZmTQMdUNYI000?stat-id=4&test-tag=83014231023617&format-type=24&banner-test-tags=eyI0NjY1Njk4IjoiODMwMTMxMjc4OTcwODgifQ%3D%3D&)

In the [**previous part of the**](http://narodstream.ru/stm-urok-121-datchik-temperatury-davleniya-i-vlazhnosti-bme280-chast-2/) lesson we began to write the sensor initialization, in which the identifier was counted, the calibration data from the sensor, also set the timing for the Standby mode.

Now oversampling and filters.

I saw a few examples, and also much envied from the Arduino libraries, and decided to use these settings.

Let's start with the filter. The filter coefficient is also set using the corresponding bits in the configuration register



Let's **go** to the header file **BME280.h** and add a mask and settings for bits of the filter coefficient

#define BME280\_STBY\_20 0xE0

//------------------------------------------------

**#define BME280\_FILTER\_MSK 0x1C**

**#define BME280\_FILTER\_OFF 0x00**

**#define BME280\_FILTER\_2 0x04**

**#define BME280\_FILTER\_4 0x08**

**#define BME280\_FILTER\_8 0x0C**

**#define BME280\_FILTER\_16 0x10**

Let's return to the **BME280.c** file and add the function of filter coefficient adjustment after the function **BME280\_SetStandby**

**//------------------------------------------------**

**void BME280\_SetFilter(uint8\_t filter) {**

**uint8\_t reg;**

**reg = BME280\_ReadReg(BME280\_REG\_CONFIG) & ~BME280\_FILTER\_MSK;**

**reg |= filter & BME280\_FILTER\_MSK;**

**BME280\_WriteReg(BME280\_REG\_CONFIG,reg);**

**}**

**//------------------------------------------------**

Call this function in the initialization function of the sensor by setting the average filter factor of the readings

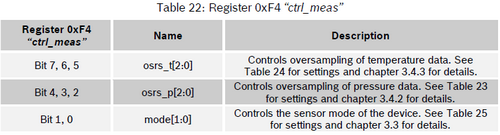
BME280\_SetStandby(BME280\_STBY\_1000);

**BME280\_SetFilter(BME280\_FILTER\_4);**

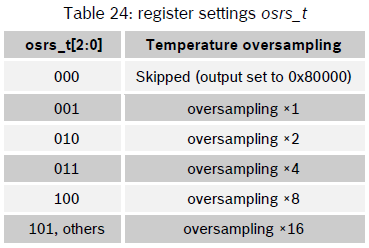
Next, oversamplings, which are configured separately for each type of measurement.

Over-sampling (oversampling) settings are stored in two registers.

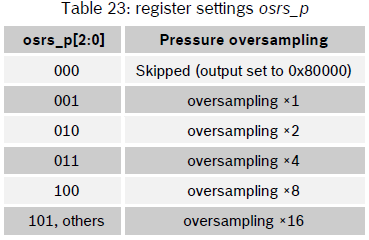
In one register - **ctrl\_meas** - the settings of temperature and pressure **oversampling** are stored



Here is the over-sampling settings for temperature readings

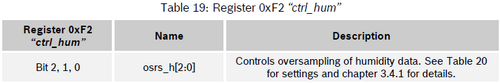


But the settings for oversampling of pressure readings

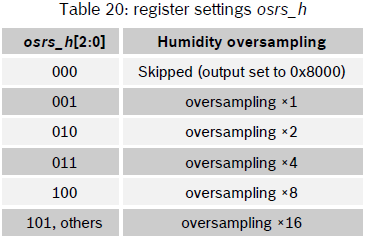


Also in this register there are settings for the operating mode of the sensor. We will look at them a little later.

And there is a separate register for setting up humidity overuse



Here are the settings



Let's **go** to the header file **BME280.h** and add there macros for these registers

#define BME280\_SOFTRESET\_VALUE 0xB6 //BME280 SOFT RESET VALUE

**#define BME280\_REG\_CTRL\_HUM 0xF2 // Humidity measure control register**

#define BME280\_REGISTER\_STATUS 0XF3 //BME280 STATUS REGISTER

**#define BME280\_REG\_CTRL\_MEAS 0xF4 // Control register pressure and temperature measure**

Also we will add macros for masks and settings of oversampling bits

#define BME280\_FILTER\_16 0x10

//------------------------------------------------

**#define BME280\_OSRS\_T\_MSK 0xE0**

**#define BME280\_OSRS\_T\_SKIP 0x00**

**#define BME280\_OSRS\_T\_x1 0x20**

**#define BME280\_OSRS\_T\_x2 0x40**

**#define BME280\_OSRS\_T\_x4 0x60**

**#define BME280\_OSRS\_T\_x8 0x80**

**#define BME280\_OSRS\_T\_x16 0xA0**

**#define BME280\_OSRS\_P\_MSK 0x1C**

**#define BME280\_OSRS\_P\_SKIP 0x00**

**#define BME280\_OSRS\_P\_x1 0x04**

**#define BME280\_OSRS\_P\_x2 0x08**

**#define BME280\_OSRS\_P\_x4 0x0C**

**#define BME280\_OSRS\_P\_x8 0x10**

**#define BME280\_OSRS\_P\_x16 0x14**

**#define BME280\_OSRS\_H\_MSK 0x07**

**#define BME280\_OSRS\_H\_SKIP 0x00**

**#define BME280\_OSRS\_H\_x1 0x01**

**#define BME280\_OSRS\_H\_x2 0x02**

**#define BME280\_OSRS\_H\_x4 0x03**

**#define BME280\_OSRS\_H\_x8 0x04**

**#define BME280\_OSRS\_H\_x16 0x05**

We return to the file **BME280.c** and add the over-sampling settings functions for different readings after the function **BME280\_SetFilter**

**//------------------------------------------------**

**void BME280\_SetOversamplingTemper(uint8\_t osrs)**

**{**

**uint8\_t reg;**

**reg = BME280\_ReadReg(BME280\_REG\_CTRL\_MEAS) & ~BME280\_OSRS\_T\_MSK;**

**reg |= osrs & BME280\_OSRS\_T\_MSK;**

**BME280\_WriteReg(BME280\_REG\_CTRL\_MEAS,reg);**

**}**

**//------------------------------------------------**

**void BME280\_SetOversamplingPressure(uint8\_t osrs)**

**{**

**uint8\_t reg;**

**reg = BME280\_ReadReg(BME280\_REG\_CTRL\_MEAS) & ~BME280\_OSRS\_P\_MSK;**

**reg |= osrs & BME280\_OSRS\_P\_MSK;**

**BME280\_WriteReg(BME280\_REG\_CTRL\_MEAS,reg);**

**}**

**//------------------------------------------------**

**void BME280\_SetOversamplingHum(uint8\_t osrs)**

**{**

**uint8\_t reg;**

**reg = BME280\_ReadReg(BME280\_REG\_CTRL\_HUM) & ~BME280\_OSRS\_H\_MSK;**

**reg |= osrs & BME280\_OSRS\_H\_MSK;**

**BME280\_WriteReg(BME280\_REG\_CTRL\_HUM,reg);**

**//The 'ctrl\_hum' register needs to be written**

**//after changing 'ctrl\_hum' for the changes to become effwctive.**

**reg = BME280\_ReadReg(BME280\_REG\_CTRL\_MEAS);**

**BME280\_WriteReg(BME280\_REG\_CTRL\_MEAS,reg);**

**}**

**//------------------------------------------------**

I think you noticed that after entering the values ​​of the moisture oversampling in the corresponding register, we counted and re-recorded the data of register F4. We did this because there is such a requirement that the changes in the humidity oversampling register will become relevant only after recording the F4 register, which we reported in the commentary

http://narodstream.ru/wp-content/uploads/2018/05/stm121img40.png

We will configure the oversampling by calling these functions with the desired settings in the initialization function

BME280\_SetFilter(BME280\_FILTER\_4);

**BME280\_SetOversamplingTemper(BME280\_OSRS\_T\_x4);**

**BME280\_SetOversamplingPressure(BME280\_OSRS\_P\_x2);**

**BME280\_SetOversamplingHum(BME280\_OSRS\_H\_x1);**

We consider the settings and see them in the terminal program

BME280\_SetOversamplingHum(BME280\_OSRS\_H\_x1);

**value32 = BME280\_ReadReg(BME280\_REG\_CTRL\_MEAS);**

**value32 |= BME280\_ReadReg(BME280\_REG\_CTRL\_HUM) << 8;**

**sprintf(str1, "Measurements status: %04Xrn", value32);**

**HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);**

**sprintf(str1, "Temperature: %srnPressure: %srnHumidity: %srn",**

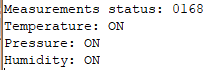
**(value32 & BME280\_OSRS\_T\_MSK) ? "ON" : "OFF",**

**(value32 & BME280\_OSRS\_P\_MSK) ? "ON" : "OFF",**

**((value32 >> 8) & BME280\_OSRS\_H\_MSK) ? "ON" : "OFF");**

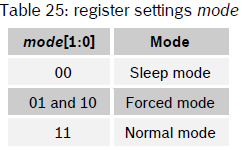
**HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);**

We will collect the code, we will sew the controller and see what settings we will display in the terminal program



It remains to fully initialize the sensor, we only enable the NORMAL mode.

As we recall, modes are configured in certain bits of the **ctrl\_meas** register



Go to the header file **BME280.h** and add there macros for the mask and settings for the bits of the mode setting

#define BME280\_OSRS\_H\_x16 0x05

//------------------------------------------------

**#define BME280\_MODE\_MSK 0x03**

**#define BME280\_MODE\_SLEEP 0x00**

**#define BME280\_MODE\_FORCED 0x01**

**#define BME280\_MODE\_NORMAL 0x03**

//------------------------------------------------

Let's return to the **BME280.c** file and add the function of setting the current mode after the function **BME280\_SetOversamplingHum**

**//------------------------------------------------**

**void BME280\_SetMode(uint8\_t mode) {**

**uint8\_t reg;**

**reg = BME280\_ReadReg(BME280\_REG\_CTRL\_MEAS) & ~BME280\_MODE\_MSK;**

**reg |= mode & BME280\_MODE\_MSK;**

**BME280\_WriteReg(BME280\_REG\_CTRL\_MEAS,reg);**

**}**

**//------------------------------------------------**

Enable NORMAL mode in the sensor initialization function

  HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);

**BME280\_SetMode(BME280\_MODE\_NORMAL);**

}

And by the closing parenthesis of the body of the function, we see that we have finally completed the initialization of the sensor.

The smallest thing left is to count the readings of the quantities. Well nothing. Thanks to the examples of the code in the technical documentation, we still, I hope, will cope with this.

Above the initialization function, we add for the time being the readings of the reading functions of the various values ​​measured by our sensor

**//------------------------------------------------**

**float BME280\_ReadTemperature(void)**

**{**

**float temper\_float = 0.0f;**

**return temper\_float;**

**}**

**//------------------------------------------------**

**float BME280\_ReadPressure(void)**

**{**

**float press\_float = 0.0f;**

**return press\_float;**

**}**

**//------------------------------------------------**

**float BME280\_ReadHumidity(void)**

**{**

**float hum\_float = 0.0f;**

**return hum\_float;**

**}**

**//------------------------------------------------**

**float BME280\_ReadAltitude(float seaLevel)**

**{**

**float att = 0.0f;**

**return att;**

**}**

**//------------------------------------------------**

Add for these functions the prototypes in the header file.

Also in the header file **BME280.h** and add macros for the registers of different sensor readings

#define BME280\_REG\_CONFIG 0xF5 // Configuration register

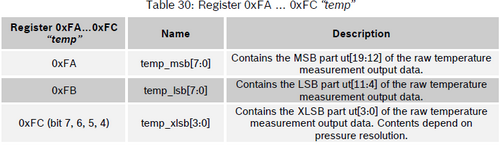
**#define BME280\_REGISTER\_PRESSUREDATA 0xF7**

**#define BME280\_REGISTER\_TEMPDATA 0xFA**

**#define BME280\_REGISTER\_HUMIDDATA 0xFD**

Let's start with the temperature.

The initial "raw" temperature value directly from the sensor after processing by the ADC block is written to the next register



It turns out that the temperature readings are recorded in 20 bits.

Also, we see that the bytes in the registers are in the reverse order - the highest byte ahead, so you'll have to make a coup.

Let's just consider this register without a coup.

To do this, return to the file **BME280.c** and add the function of reading the 24-bit register after the function **BME280\_ReadReg\_S16**

**//------------------------------------------------**

**void BME280\_ReadReg\_U24(uint8\_t Reg, uint32\_t \*Value)**

**{**

**I2Cx\_ReadData24(BME280\_ADDRESS,Reg,Value);**

**\*(uint32\_t \*) Value &= 0x00FFFFFF;**

**}**

**//------------------------------------------------**

Since we do not have 24-bit types, we use 32-bit ones, respectively, so we remove the extra high byte.

Let's read the register of temperature values ​​in the body of the function **BME280\_ReadTemperature**

float temper\_float = 0.0f;

**uint32\_t temper\_raw;**

**BME280\_ReadReg\_U24(BME280\_REGISTER\_TEMPDATA,&temper\_raw);**

In the **main ()** function of the **main.c** file, we add the local variables for storing the data of the various parameters of the sensor

/\* USER CODE BEGIN 1 \*/

**float tf = 0.0f, pf = 0.0f, af = 0.0f, hf = 0.0f;**

/\* USER CODE END 1 \*/

Let's go to the endless loop and call the temperature reading function, after which we add a delay of 1000 milliseconds, which is stipulated by the STANDBY mode set by us after reading the sensor readings

  /\* USER CODE BEGIN 3 \*/

**tf = BME280\_ReadTemperature();**

**sprintf(str1, "Temperature: %.3f \*Crn", tf);**

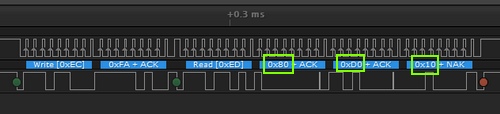
**HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);**

**HAL\_Delay(1000);**

}

Of course, in the terminal we are not going to see any evidence yet, since they are not yet calculated, but in the logical analysis program we will see what bytes we read from the register of temperature values. To do this, we'll compile the code and run the controller.

Here are our accepted bytes



The bytes are read, this is already good.

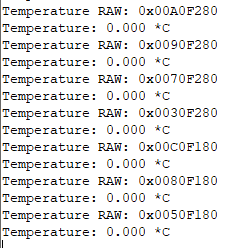
Now let's see how they are written into a variable. To do this, go back to the **BME280.c** file in the **BME280\_ReadTemperature** function and display the raw temperature value in the terminal program there

BME280\_ReadReg\_U24(BME280\_REGISTER\_TEMPDATA,&temper\_raw);

**sprintf(str1, "Temperature RAW: 0x%08Xrn", temper\_raw);**

**HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);**

Let's collect the code, we'll tell the controller and see our data



According to the data changes we see that they are turned upside down, that is, the high byte changes, the temperature can not go on an order of magnitude. Well, in principle, we expected it, because the bytes in the register are reversed, judging by the datasheet, the high byte ahead (big endian format). So we need to somehow turn the whole thing around. We have already met with this phenomenon when we programmed the chip ENC28J60.

Let's **go** to the header file **BME280.h** and add the macros of the byte permutation, at the same time also for 16-bit values, they too will be needed

#define LED\_TGL HAL\_GPIO\_TogglePin(LED\_GPIO\_PORT, LED\_PIN)

//------------------------------------------------

**#define be16toword(a) ((((a)>>8)&0xff)|(((a)<<8)&0xff00))**

**#define be24toword(a) ((((a)>>16)&0x000000ff)|((a)&0x0000ff00)|(((a)<<16)&0x00ff0000))**

We return to the file **BME280.c** and add the function of reading the 24-bit register in Big Endian format after the function **BME280\_ReadReg\_U24**

**//------------------------------------------------**

**void BME280\_ReadReg\_BE\_U24(uint8\_t Reg, uint32\_t \*Value)**

**{**

**I2Cx\_ReadData24(BME280\_ADDRESS,Reg,Value);**

**\*(uint32\_t \*) Value = be24toword(\*(uint32\_t \*) Value) & 0x00FFFFFF;**

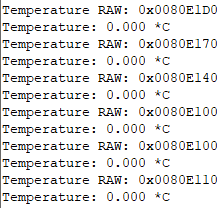
**}**

**//------------------------------------------------**

Let's return to the function of reading and processing the temperature **BME280\_ReadTemperature**  and change the function there

BME280\_ReadReg**\_BE**\_U24(BME280\_REGISTER\_TEMPDATA,&temper\_raw);

We will collect the code, we will sew the controller and see the result now



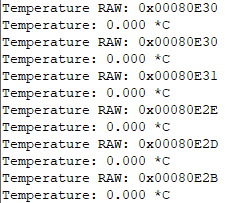
It's quite another matter!

Now, move it all 4 bits to the right, because that's how we store the raw temperature value and the four least significant bits do not participate in the readings

BME280\_ReadReg\_BE\_U24(BME280\_REGISTER\_TEMPDATA,&temper\_raw);

**temper\_raw >>= 4;**

We will collect the code, we will sew the controller and we will look now result



All perfectly moved.

Now add a global variable for temporary storage of temperature readings

BME280\_CalibData CalibData;

**int32\_t temper\_int;**

Let's return to the function **BME280\_ReadTemperature**  and add some more local variables for intermediate calculations

uint32\_t temper\_raw;

**int32\_t val1, val2;**

Then, following the example from the datasheet, we will calculate the intermediate temperature value, while without the use of a floating point

HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);

**val1 = ((((temper\_raw>>3) - ((int32\_t)CalibData.dig\_T1 <<1))) \***

**((int32\_t)CalibData.dig\_T2)) >> 11;**

**val2 = (((((temper\_raw>>4) - ((int32\_t)CalibData.dig\_T1)) \***

**((temper\_raw>>4) - ((int32\_t)CalibData.dig\_T1))) >> 12) \***

**((int32\_t)CalibData.dig\_T3)) >> 14;**

**temper\_int = val1 + val2;**

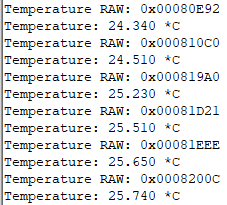
And now we will calculate the reading with the floating point, again following the recommendations of the datasheet

temper\_int = val1 + val2;

**temper\_float = ((temper\_int \* 5 + 128) >> 8);**

**temper\_float /= 100.0f;**

We will collect the code, we will tell the controller and see the result of the function in the terminal program



Well, since we have read everything so well, we can output the raw readings to the terminal

~~sprintf(str1, "Temperature RAW: 0x%08Xrn", temper\_raw);~~

~~HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);~~

In the [**next part of the**](http://narodstream.ru/stm-urok-121-datchik-temperatury-davleniya-i-vlazhnosti-bme280-chast-4/) lesson we will learn how to read atmospheric pressure and air humidity from the sensor and then connect the LCD2004 display for easy monitoring of the readings.

**STM Lesson 121. Temperature, pressure and humidity sensor BME280. Part 4**

Posted on [June 15, 2018](http://narodstream.ru/stm-urok-121-datchik-temperatury-davleniya-i-vlazhnosti-bme280-chast-4/)by [http://1.gravatar.com/avatar/4824b24065500834db4b9f331b608833?s=32&d=mm&r=gNarod Stream](http://narodstream.ru/author/admin/) Published in [I2C](http://narodstream.ru/i2c/) , [Programming STM32](http://narodstream.ru/rub_stm32/)- [1 comment ↓](http://narodstream.ru/stm-urok-121-datchik-temperatury-davleniya-i-vlazhnosti-bme280-chast-4/#comments)

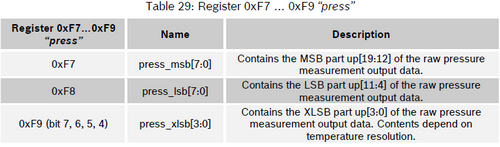
[Switches contactlessSwitches **contactless capacitive** type VBETo learn moreferrol.ru](https://an.yandex.ru/count/RFiuS4PRgsW50C01CQt0ari00000ECgs7402I09Wl0Xe172Osh300u01zl2S28W1oeMH6901nCMv19W1pfVW1BW1fFFV1C010jW1-9uxu06QWEG2w04s-04EY083e0BanGcv0j5VvJvDEO64y0BuWP_43FW2We20W8304u03vPIEy0g80ylfez4Ec0EqZmoe0nAm0mJe1Da1-0IOg0E81PYe0v05aBm3e0MXVwW5x6km1Unhk0NqSC05wEOAo0N8tmBG1SCmu0LQy0K1c0RCrlP_e0Rk0gW6xWB91iOharf9W7KJqGR6lyZHIO1r4za60000S740002f1_XHBDxvbyL8i0U0W90Cq0S2u0U62lW70O080T08ceg0WS2GW0BW2FBY180A0S4A00000000y3_O2WBW2e29UlWAmFmLY0i6gWiGKCI5-FXH00315Cr6MtK50C0BWAC5o0k0r9C1sGluKIpU-PV5IEWBcAW3y0i6Y0oE_Tw-0UWC0fWD-9nbyWq0-Wq0WWu0YGu00000003mFv0Em8Gzc0x6rypKbOdWkXEW3i24FR0E0Q4F00000000y3-e3_YZmRYL_-NYI000?stat-id=3&test-tag=85761936351233&format-type=24&banner-test-tags=eyI0NjY1Njk4IjoiODU3NjE5MDY5NjY1MjgifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

[Story about the business programmerI tell about my business on site templates. Read more on the blogTo learn moregoodlifer.ru](https://an.yandex.ru/count/Fp9eT_NQpBO50Di1CQx0ari00000ECgs7402I09Wl0Xe173g_OZ11801uA_0sfR0qA2n0OW1kDgojpEG0SZneyekc06CXFksBA01ll6ZoYwe0OQ4-xOik060kigo6S010jW1niAV5E01iE_e0-W1HlW10OW21A02l9M25xa2qL_bFaqvWOJm0gJfpxy6-0A2W820WC0JW0El_AMm3eW3_yNHrGUm0mIu1Fy1w0Ik0VW4h_W2Y0Ml-0AG1Rgg0w05zN2e1TXVi0NONxW5jMp01TkF2SW5yAW2q0M7A-05Ol050PW6jlMxlG6W1ku2g0Rk0ia6nYkJMac0THFH1iQ_oD59W7KJsGO0001mSG000907Aga7-540j3UOnKYm1u20a0pG1mBW1uOA-0S1W0W1q0YwYe21m9200k08teh12u0A0S4A00000000y3_O2WBW2e29UlWAmFmLY0i8gWiGdg_wQFXH000jYjn6MtK50C0BWAC5o0k0r9C1sGluKG2qDvZ5IEWBh_W2y0i6Y0pupkA-0UWC0fWDoR9dyWq0-Wq0WWu0YGu00000003mFv0Em8Gzc0x6rypKbOdWkXEW3i24FR0E0Q4F00000000y3-e3_YZiShjXkRYI000?stat-id=4&test-tag=85761936351233&format-type=24&banner-test-tags=eyI2MjUxOTA0MTAwIjoiODU3NjE5MDY5OTkyOTYifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

In the [**previous part of the**](http://narodstream.ru/stm-urok-121-datchik-temperatury-davleniya-i-vlazhnosti-bme280-chast-3/) lesson, we set up a filter and oversampling, and also read the temperature value from the sensor.

Now the pressure.

Let's see the register



Since the pressure is stored in the same way as the temperature, we pass to the corresponding BME280\_ReadPressure function and we will read the raw pressure indications there in a similar way, but again, without fail following the requirements of the technical documentation, we first read the temperature, because the temperature readings are involved in calculating the atmospheric pressure

float press\_float = 0.0f;

**uint32\_t press\_raw, pres\_int;**

**BME280\_ReadTemperature(); // must be done first to get t\_fine**

**BME280\_ReadReg\_BE\_U24(BME280\_REGISTER\_PRESSUREDATA,&press\_raw);**

**press\_raw >>= 4;**

I think you can in the terminal program, the raw indicators do not look and do not spend for this time, everything will be fine, I've checked it many times.

Therefore, first we will add local variables for intermediate calculations

uint32\_t press\_raw, pres\_int;

**int64\_t val1, val2, p;**

And then we calculate the pressure, following the example from the datasheet

press\_raw >>= 4;

**val1 = ((int64\_t) temper\_int) - 128000;**

**val2 = val1 \* val1 \* (int64\_t)CalibData.dig\_P6;**

**val2 = val2 + ((val1 \* (int64\_t)CalibData.dig\_P5) << 17);**

**val2 = val2 + ((int64\_t)CalibData.dig\_P4 << 35);**

**val1 = ((val1 \* val1 \* (int64\_t)CalibData.dig\_P3) >> 8) + ((val1 \* (int64\_t)CalibData.dig\_P2) << 12);**

**val1 = (((((int64\_t)1) << 47) + val1)) \* ((int64\_t)CalibData.dig\_P1) >> 33;**

**if (val1 == 0) {**

**return 0; // avoid exception caused by division by zero**

**}**

**p = 1048576 - press\_raw;**

**p = (((p << 31) - val2) \* 3125) / val1;**

**val1 = (((int64\_t)CalibData.dig\_P9) \* (p >> 13) \* (p >> 13)) >> 25;**

**val2 = (((int64\_t)CalibData.dig\_P8) \* p) >> 19;**

**p = ((p + val1 + val2) >> 8) + ((int64\_t)CalibData.dig\_P7 << 4);**

**pres\_int = ((p >> 8) \* 1000) + (((p & 0xff) \* 390625) / 100000);**

**press\_float = pres\_int / 100.0f;**

Let's go to the **main ()** function of the **main.c** file and in the endless loop also read and display the pressure in different values

HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);

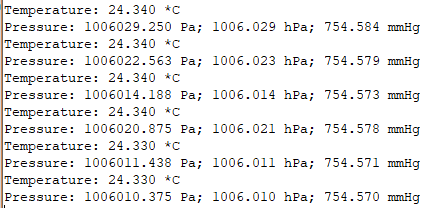
**pf = BME280\_ReadPressure();**

**sprintf(str1, "Pressure: %.3f Pa; %.3f hPa; %.3f mmHg\r\n", pf, pf/1000.0f, pf \*0.000750061683f);**

**HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);**

HAL\_Delay(1000);

We will collect the code, we will sew the controller and see the result of the code work of the reading function and the calculation of atmospheric pressure



Now let's return to the **BME280.c** file and write the body of the function of calculating the altitude above sea level (this value depends on the atmospheric pressure, although, I confess, this dependence, as I noted, is unstable, or rather, the output is the altitude)

float att = 0.0f;

**float atm = BME280\_ReadPressure();**

**att = 44330.0 \* (1.0 - pow(atm / seaLevel, 0.1903));**

Now we need some more coefficients.

Let's **go** to the header file **BME280.h** and add macros

#define LED\_TGL HAL\_GPIO\_TogglePin(LED\_GPIO\_PORT, LED\_PIN)

//------------------------------------------------

**#define SEALEVELPRESSURE\_HPA (1013.25)**

**#define SEALEVELPRESSURE\_PA (1013250)**

//------------------------------------------------

Let's go to the **main ()** function of the **main.c** file and in the infinite loop also compute and display the height above sea level

HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);

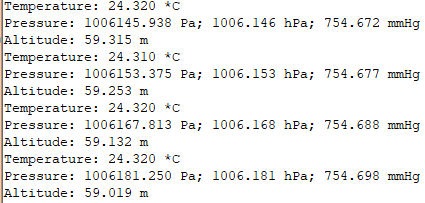
**af = BME280\_ReadAltitude(SEALEVELPRESSURE\_PA);**

**sprintf(str1, "Altitude: %.3f m\r\n", af);**

**HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);**

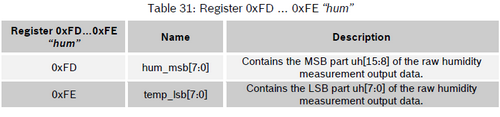
HAL\_Delay(1000);

We will collect the code, we will sew the controller and see the result in the terminal program



It remains for us to read only the humidity.

To do this, go back to the **BME280.c** file in the **BME280\_ReadHumidity** function and try to read the raw value from the register that we have for the two-byte



float hum\_float = 0.0f;

**int16\_t hum\_raw;**

**BME280\_ReadTemperature(); // must be done first to get t\_fine**

**BME280\_ReadReg\_S16(BME280\_REGISTER\_HUMIDDATA,&hum\_raw);**

**sprintf(str1, "Humidity RAW: 0x%08X\r\n", hum\_raw);**

**HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);**

Also here we see that we need to read the temperature necessarily, since its value also affects the calculation of humidity.

Let's go to the **main ()** function of the **main.c** file and in an infinite loop call the function of reading and calculating the air humidity

HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);

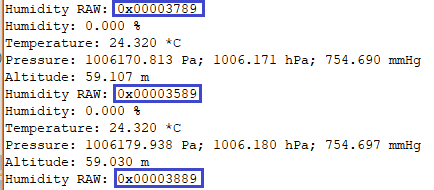
**hf = BME280\_ReadHumidity();**

**sprintf(str1, "Humidity: %.3f %%\r\n", hf);**

**HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);**

HAL\_Delay(1000);

We will collect the code, we will sew the controller and see the result of reading the two-byte value in the terminal program



As we can see, we are not the same byte again.

So go into the **BME280.c** file and add the corresponding read function there after the function **BME280\_ReadReg\_S16**

**//------------------------------------------------**

**void BME280\_ReadReg\_BE\_S16(uint8\_t Reg, int16\_t \*Value)**

**{**

**I2Cx\_ReadData16(BME280\_ADDRESS,Reg,(uint16\_t\*)Value);**

**\*(uint16\_t \*) Value = be16toword(\*(uint16\_t \*) Value);**

**}**

**//------------------------------------------------**

Now we return to the function **BME280\_ReadHumidity**  and change the called function

BME280\_ReadReg**\_BE**\_S16(BME280\_REGISTER\_HUMIDDATA,&hum\_raw);

Let's add some local variables for intermediate calculations

int16\_t hum\_raw;

**int32\_t hum\_raw\_sign, v\_x1\_u32r;**

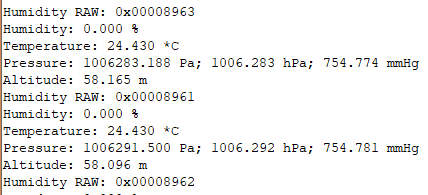
Let's assign the read value to a 32-bit sign variable and look in the terminal program for it already

BME280\_ReadReg\_BE\_S16(BME280\_REGISTER\_HUMIDDATA,&hum\_raw);

**hum\_raw\_sign = ((int32\_t)hum\_raw)&0x0000FFFF;**

sprintf(str1, "Humidity RAW: 0x%08X\r\n",hum\_raw**\_sign**);

Let's collect the code, we'll tell the controller and see the value



Now at us the younger byte dances - so should be.

Also, following the recommendations of the technical documentation, calculate the value of humidity in the value of a floating point

HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);

**v\_x1\_u32r = (temper\_int - ((int32\_t)76800));**

**v\_x1\_u32r = (((((hum\_raw\_sign << 14) - (((int32\_t)CalibData.dig\_H4) << 20) -**

**(((int32\_t)CalibData.dig\_H5) \* v\_x1\_u32r)) + ((int32\_t)16384)) >> 15) \***

**(((((((v\_x1\_u32r \* ((int32\_t)CalibData.dig\_H6)) >> 10) \***

**(((v\_x1\_u32r \* ((int32\_t)CalibData.dig\_H3)) >> 11) + ((int32\_t)32768))) >> 10) +**

**((int32\_t)2097152)) \* ((int32\_t)CalibData.dig\_H2) + 8192) >> 14));**

**v\_x1\_u32r = (v\_x1\_u32r - (((((v\_x1\_u32r >> 15) \* (v\_x1\_u32r >> 15)) >> 7) \***

**((int32\_t)CalibData.dig\_H1)) >> 4));**

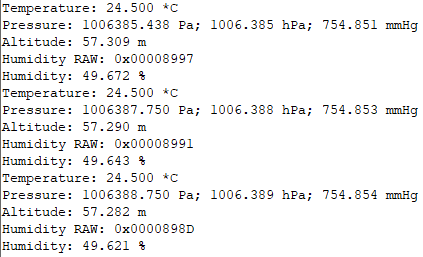
**v\_x1\_u32r = (v\_x1\_u32r < 0) ? 0 : v\_x1\_u32r;**

**v\_x1\_u32r = (v\_x1\_u32r > 419430400) ? 419430400 : v\_x1\_u32r;**

**hum\_float = (v\_x1\_u32r>>12);**

**hum\_float /= 1024.0f;**

We will collect the code, we will tell the controller and we will see the result



All indications are perfectly read and updated!

You can now delete the display of the raw value

~~sprintf(str1, "Humidity RAW: 0x%08X\r\n", hum\_raw\_sign);~~

~~HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);~~

It remains for us to connect the display and display our readings on it, since it is more beautiful with the display than in the terminal program, and more autonomously, since it will not be necessary to connect the circuit to a PC.

For this we copy the files **lcd.h** and **lcd.c** from the [**lesson**](http://narodstream.ru/stm-urok-105-nrf24l01-peredayom-dannye-chast-1/)  project [**105**](http://narodstream.ru/stm-urok-105-nrf24l01-peredayom-dannye-chast-1/)**NRF24\_RX\_LCD** to our project.

In the file main.c we connect this library

#include "BME280.h"

**#include "lcd.h"**

Also connect the file  **lcd.c** to the project tree.

Let's move to the file and remove the declaration of the string variable, since it is not needed there

~~char str1[100];~~

Back in the **main.c** file and in the **main ()** function, we call the display initialization

/\* USER CODE BEGIN 2 \*/

**LCD\_ini();**

BME280\_Init();

In an endless cycle, we display the various indicators on our display by inserting in the appropriate places the appropriate code

tf = BME280\_ReadTemperature();

sprintf(str1, "Temperature: %.3f \*C\r\n", tf);

HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);

**LCD\_SetPos(0, 0);**

**sprintf(str1, "%11.3f \*C", tf);**

**LCD\_String(str1);**

pf = BME280\_ReadPressure();

sprintf(str1, "Pressure: %.3f Pa; %.3f hPa; %.3f mmHg\r\n", pf, pf/1000.0f, pf \*0.000750061683f);

HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);

**LCD\_SetPos(0, 1);**

**sprintf(str1, "%11.3f hPa", pf/1000.0f);**

**LCD\_String(str1);**

**LCD\_SetPos(0, 2);**

**sprintf(str1, "%11.3f mmHg", pf \* 0.000750061683f);**

**LCD\_String(str1);**

af = BME280\_ReadAltitude(SEALEVELPRESSURE\_PA);

sprintf(str1, "Altitude: %.3f m\r\n", af);

HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);

hf = BME280\_ReadHumidity();

sprintf(str1, "Humidity: %.3f %%\r\n", hf);

HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);

**LCD\_SetPos(0, 3);**

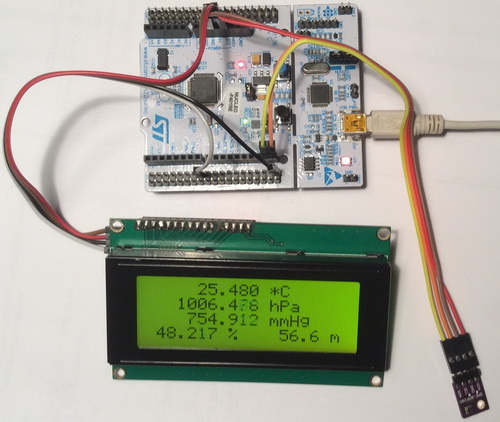
**sprintf(str1, "%7.3f %% %4.1f m", hf, af);**

**LCD\_String(str1);**

HAL\_Delay(1000);

It remains to collect the code, flash the controller, disable the logic analyzer and connect the display.

We admire the result of the work of our code (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2018/05/stm121img56.jpg)

Thus, today we have mastered yet another excellent weather sensor, which with high accuracy and speed can measure both temperature and pressure, and humidity of air.

Thank you all for your attention!