**Lesson 52**

**Part 1**

**Pressure sensor LPS25HB**

Today we will get acquainted with a sensor that measures pressure. We will review and study the sensor installed on the expansion board **X-NUCLEO-IKS01A1**, designed to work with the Nucleo debug card. We will connect this evaluation board to the Nucleo STM32F401RE board. This sensor is also implemented using MEMS technology.

This pressure sensor can also be connected to the I2C interface via the SPI interface. But we will use the connection on the usual I2C.

The sensor has the following specifications:

The range of pressure readings is 260 - 1260 hPa (hectopascals);

Range of temperature readings -30 - + 105 ° C

24 bit resolution for pressure readings and 16 bits for temperature readings;

Sensitivity 4096 LSB / hPa (bit per hectopascal) for pressure;

480 LSB / ° C for temperature.

The rms noise level (RMS noise) is 0.03 hPa without the built-in filter and 0.01 with the built-in filter;

The deviation from zero at a temperature of + 25 ° C is ± 0.1 hPa.

The deviation of the temperature reading is ± 2 ° C.

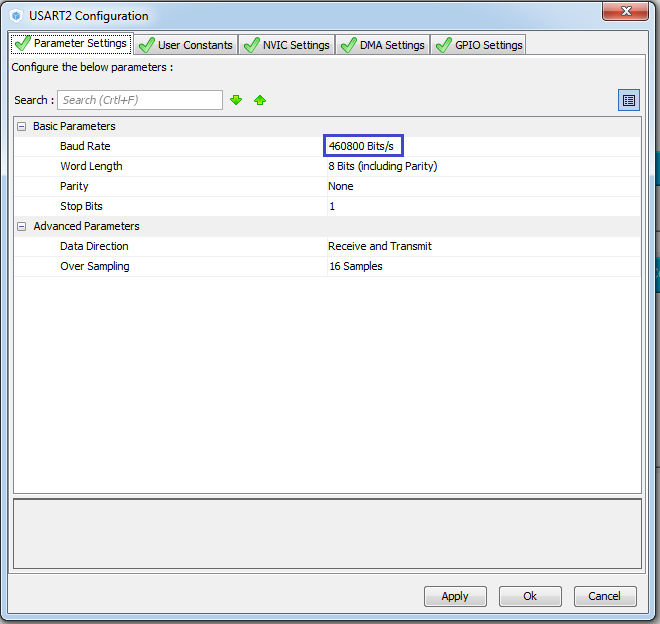
The measurement frequency is 1 - 25 Hz.

With some other indicators, registers, values ​​and other details of the gyroscope, we will get acquainted in the course of its programming.

We will create the project from the project of the last lesson, in which we worked with a magnetometer installed on the same expansion board - from the project Mag\_LIS3MDL, only we will name this project now, respectively, Press\_  LPS25HB.

The files lis3mdl.c and lis3mdl.h, respectively, will be renamed to lps25hb.c and lps25hb.hm

Run the Cube MX project. Speed ​​USART while we install 115200 bps



We do not touch anything else. Generate the project, open it. Let's set up the programmer for auto-cutting. Add the file lps25hb.c. We will compile the project.

Due to the renaming of the files, we will have errors. To eliminate these errors, we fix the connection of header files in main.c and in lps25hb.s

#include "stm32f4xx\_hal.h"  
**#include "lps25hb.h"**  
**#include "lps25hb.h"**  
**//--------------------------------------------**

In the timer interrupt handler, we comment out the call to the data reading function and send it to USART.

In the endless cycle, he has already commented on us

                // Mag\_Read ();

                HAL\_UART\_Receive\_IT (& huart2, (uint8\_t \*) str, 8);

In the lps25hb.h file, delete all the code and copy it there to save valuable time in advance the prepared code with all the macros and variables from the file macro.txt

**#ifndef LPS25HB\_H\_**

**#define LPS25HB\_H\_**

**#include "stm32f4xx\_hal.h"**

**#include <string.h>**

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

**#define ABS (x) (x <0)? (-x): x**

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

**#define LD2\_Pin GPIO\_PIN\_5**

**#define LD2\_GPIO\_Port GPIOA**

**#define LD2\_ON HAL\_GPIO\_WritePin (GPIOA, GPIO\_PIN\_5, GPIO\_PIN\_SET) // GREEN**

**#define LD2\_OFF HAL\_GPIO\_WritePin (GPIOA, GPIO\_PIN\_5, GPIO\_PIN\_RESET)**

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

**#define LPS25HB\_ADDRESS 0xBA**

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

**#define LPS25HB\_WHO\_AM\_I\_REG 0x0F**

**#define LPS25HB\_RES\_CONF\_REG 0x10**

**#define LPS25HB\_CTRL\_REG1 0x20**

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

**#define LPS25HB\_WHO\_AM\_I 0xBD**

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

**#define LPS25HB\_PD\_ACTIVE\_MODE 0x80**

**#define LPS25HB\_PD\_POWERDOWN\_MODE 0x00**

**#define LPS25HB\_PD\_MASK 0x80**

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

**#define LPS25HB\_ODR\_ONE\_SHOT 0x00**

**#define LPS25HB\_ODR\_1HZ 0x10**

**#define LPS25HB\_ODR\_7HZ 0x20**

**#define LPS25HB\_ODR\_12\_5HZ 0x30**

**#define LPS25HB\_ODR\_25HZ 0x40**

**#define LPS25HB\_ODR\_MASK 0x70**

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

**#define LPS25HB\_DIFF\_EN\_ENABLE 0x08**

**#define LPS25HB\_DIFF\_EN\_DISABLE 0x00**

**#define LPS25HB\_DIFF\_EN\_MASK 0x08**

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

**#define LPS25HB\_BDU\_DISABLE 0x00**

**#define LPS25HB\_BDU\_ENABLE 0x40**

**#define LPS25HB\_BDU\_MASK 0x04**

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

**#define LPS25HB\_SIM\_3\_WIRE 0x01**

**#define LPS25HB\_SIM\_4\_WIRE 0x00**

**#define LPS25HB\_SIM\_MASK 0x01**

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

**#define LPS25HB\_AVGP\_8 0x00**

**#define LPS25HB\_AVGP\_32 0x01**

**#define LPS25HB\_AVGP\_128 0x02**

**#define LPS25HB\_AVGP\_512 0x03**

**#define LPS25HB\_AVGT\_8 0x00**

**#define LPS25HB\_AVGT\_16 0x04**

**#define LPS25HB\_AVGT\_32 0x08**

**#define LPS25HB\_AVGT\_64 0x0C**

**#define LPS25HB\_AVGT\_MASK 0x0C**

**#define LPS25HB\_AVGP\_MASK 0x03**

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

**#define LPS25HB\_TEMP\_OUT\_L\_REG 0x2B**

**#define LPS25HB\_TEMP\_OUT\_H\_REG 0x2C**

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

**#define LPS25HB\_PRESS\_OUT\_XL\_REG 0x28**

**#define LPS25HB\_PRESS\_OUT\_L\_REG 0x29**

**#define LPS25HB\_PRESS\_OUT\_H\_REG 0x2A**

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

**void Press\_Ini (void);**

**void Press\_Read (void);**

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

**#endif / \* LPS25HB\_H\_ \* /**

The function Mag\_Ini is renamed to Press\_Ini in main.c and in lps25hb.s

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

void **Press**\_Ini (void)

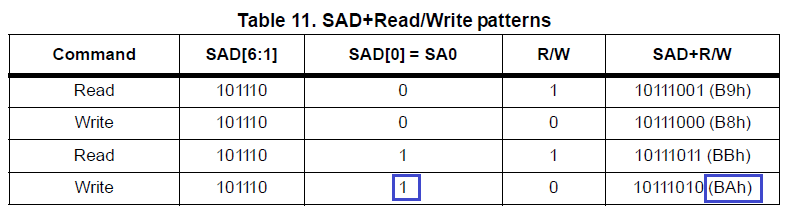
{

**Press**\_Ini ();

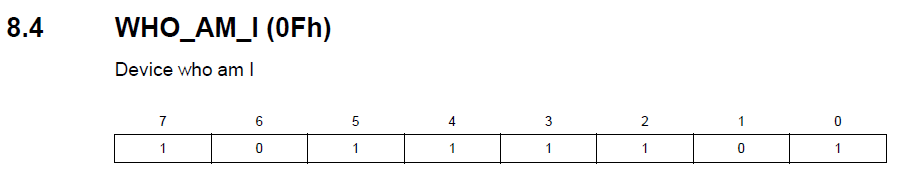
  / \* USER CODE END 2 \* /

The next task for us is to write the sensor initialization code. And, as always, according to the established tradition, since we have the connection on the I2C bus, we start it, of course, from the chip ID reading.

Based on the connection of the chip, the address we select is **0xBA**



The register for reading the identifier is used by **WHO\_AM\_I (0Fh)**



In this table, we also see by translating the binary code into hexadecimal, that the identifier should be exactly 0xBD.

The function Mag\_ReadID is renamed to Press\_ReadID both in the implementation and in the call

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

uint8\_t **Press**\_ReadID (void)

{

        HAL\_Delay (1000);

        if ( **Press**\_ReadID () == LIS3MDL\_MAG\_WHO\_AM\_I) LD2\_ON;

Similarly, we will do with the functions Mag\_IO\_Read and Mag\_IO\_Write

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

uint8\_t **Press**\_IO\_Read (uint16\_t DeviceAddr, uint8\_t RegisterAddr)

{

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

void **Press**\_IO\_Write (uint16\_t DeviceAddr, uint8\_t RegisterAddr, uint8\_t Value)

{

Also we will correct the code in the function of reading the identifier

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

uint8\_t **Press**\_ReadID (void)

{

        uint8\_t ctrl = 0x00;

        ctrl = **Press**\_IO\_Read ( **LPS25HB\_ADDRESS,** **LPS25HB\_WHO\_AM\_I\_REG**);

        return ctrl;

}

Let's fix the code in the main initialization function

        HAL\_Delay (1000);

        if ( **Press**\_ReadID () == **LPS25HB\_WHO\_AM\_I**) LD2\_ON;

In functions that we do not use yet, we will comment out all the code, wrapping it here in such tags / \* \* /.

Also comment in the initialization is this

        else Error ();

**// LD2\_OFF;**

**// MagInit (ctrl);**

**// LD2\_ON;**

We compile the code, we will write the controller and check the result of our work. The green LED should light up.

In the [**next part of**](http://narodstream.ru/stm-urok-52-datchik-davlenija-lps25hb-chast-2/) this lesson we will write the initialization of the pressure sensor, and also write the function of reading and preprocessing the temperature value.

**Lesson 52**

**Part 2**

# ****Pressure sensor LPS25HB****

In the [**previous part of the**](http://narodstream.ru/stm-urok-52-datchik-davlenija-lps25hb-chast-1/) lesson, we added all the macros that make the reading and writing of the code more convenient, and also counted the identifier of the sensor.

Let's continue writing the initialization.

Let's rename the MagInit function to PressInit

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

void **Press**Init (uint16\_t InitStruct)

{

Also, we will correct and uncomment the call of this function in the main sensor initialization function

        LD2\_OFF;

**Press**Init (ctrl);

        LD2\_ON;

We continue to write the initialization, gradually uncommenting and correcting the code in the function

                uint8\_t value = 0;

        // while we turn off the sensor ( **PD = 0**)

        value = **Press\_IO\_Read**( **LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1**);

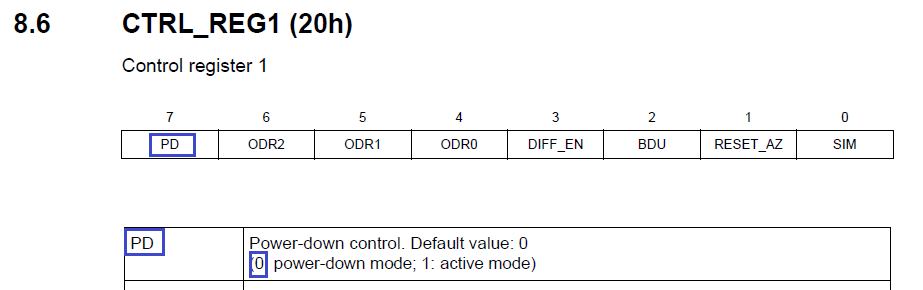
        value & = ~ **LPS25HB\_PD\_MASK**;

        value | = **LPS25HB\_PD\_POWERDOWN\_MODE**;

**Press\_IO\_Write**( **LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1**, value);

        / \*

Here we use the **CTRL\_REG1**register **(20h)** and reset the PD bit there to stop the sensor



Write the code further

        Press\_IO\_Write (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1, value);

        // Enable Data Rate 25 Hz

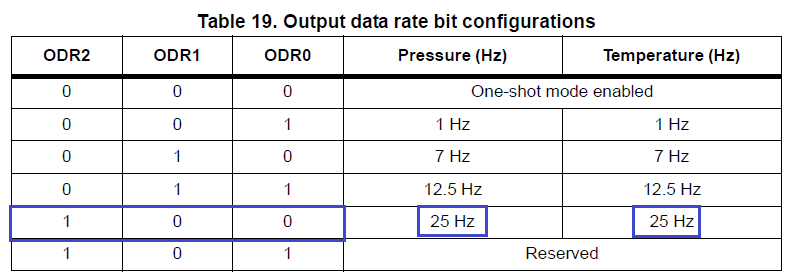
**value = Press\_IO\_Read (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1);**

**value & = ~ LPS25HB\_ODR\_MASK;**

**value | = LPS25HB\_ODR\_25HZ;**

**Press\_IO\_Write (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1, value);**

Here we use the same register **CTRL\_REG1 (20h)** and set the bits responsible for the reading frequency



Continue writing the initialization

        Press\_IO\_Write (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1, value);

**// Enable Interrupt Circuit**

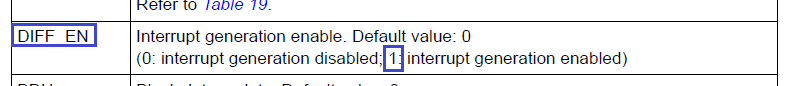
**value = Press\_IO\_Read (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1);**

**value & = ~ LPS25HB\_DIFF\_EN\_MASK;**

**value | = LPS25HB\_DIFF\_EN\_ENABLE;**

**Press\_IO\_Write (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1, value);**

Here we again use the register **CTRL\_REG1 (20h)**and set there a bit that is responsible for enabling the interrupt scheme



Continue on

        Press\_IO\_Write (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1, value);

**// Enable BDU**

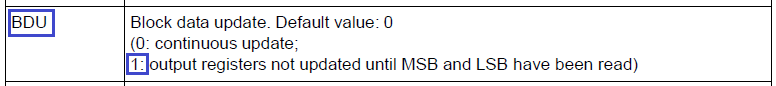
**value = Press\_IO\_Read (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1);**

**value & = ~ LPS25HB\_BDU\_MASK;**

**value | = LPS25HB\_BDU\_ENABLE;**

**Press\_IO\_Write (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1, value);**

And again we have the same register **CTRL\_REG1 (20h).**We now set the bit in it, which is responsible for enabling BDU (Block data update). With this regime, we are already familiar with you



We write further the code

        Press\_IO\_Write (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1, value);

**// Set SPI mode 3 WIRE**

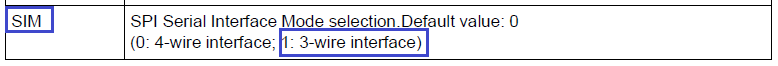
**value = Press\_IO\_Read (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1);**

**value & = ~ LPS25HB\_SIM\_MASK;**

**value | = LPS25HB\_SIM\_3\_WIRE;**

**Press\_IO\_Write (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1, value);**

We have the same register register **CTRL\_REG1 (20h)**. Turn on 3-wire mode, because by default it is set to 4-wire



Let's continue to write further

        Press\_IO\_Write (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1, value);

**// Set internal averaging sample counts for pressure and temperature**

**value = Press\_IO\_Read (LPS25HB\_ADDRESS, LPS25HB\_RES\_CONF\_REG);**

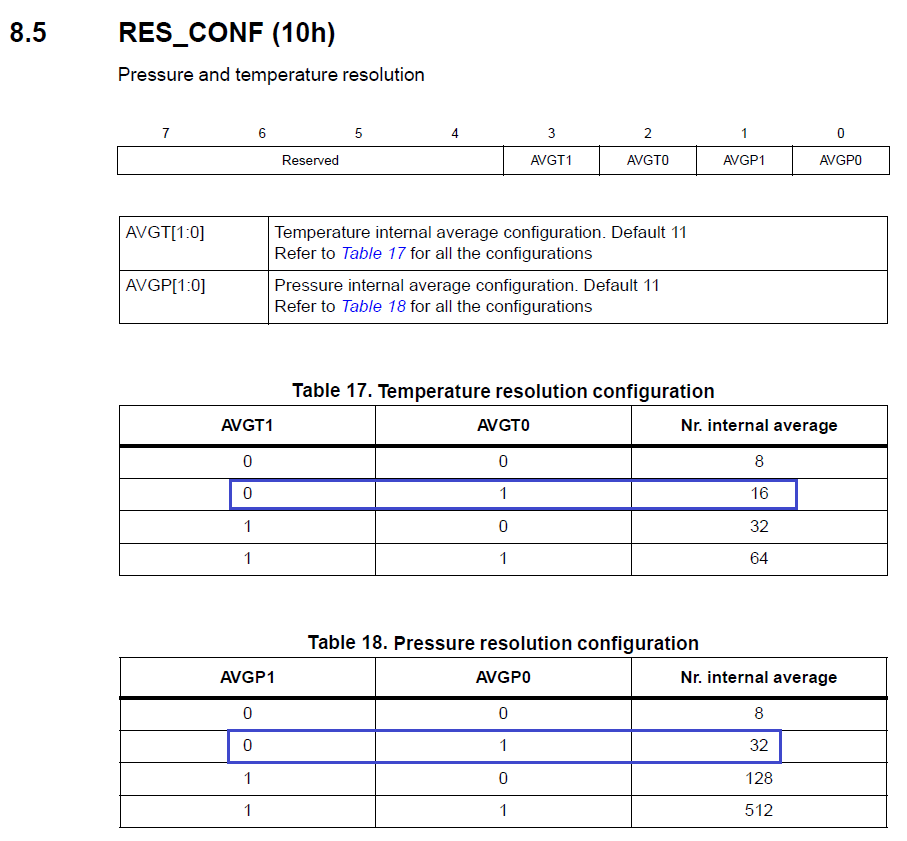
**value & = ~ (LPS25HB\_AVGT\_MASK | LPS25HB\_AVGP\_MASK);**

**value | = LPS25HB\_AVGP\_32;**

**value | = LPS25HB\_AVGT\_16;**

**Press\_IO\_Write (LPS25HB\_ADDRESS, LPS25HB\_RES\_CONF\_REG, value);**

Here the register is already different - **RES\_CONF (10h)**. This register is responsible for averaging several indications (low-frequency filtering) of pressure and temperature readings. Set 32 ​​readings for pressure and 16 for temperature



And the last:

        Press\_IO\_Write (LPS25HB\_ADDRESS, LPS25HB\_RES\_CONF\_REG, value);

**// Turn on the sensor**

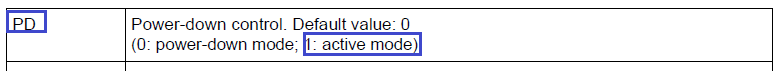
**value = Press\_IO\_Read (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1);**

**value & = ~ LPS25HB\_PD\_MASK;**

**value | = LPS25HB\_PD\_ACTIVE\_MODE;**

**Press\_IO\_Write (LPS25HB\_ADDRESS, LPS25HB\_CTRL\_REG1, value);**

Here, we transfer the sensor to the active mode by bit 1 of the register, that is, turn it on



This completes the initialization.

We will collect the code, we will sew the controller and check if the LED is off.

If it's okay, then we'll start writing the code for reading the readings from the sensor . First, we write the function of reading the temperature readings

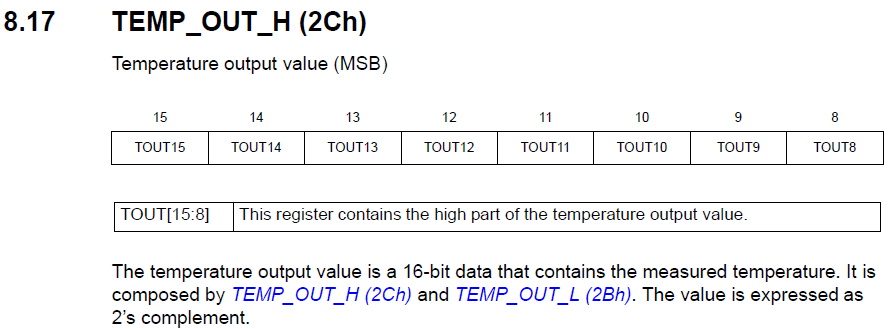
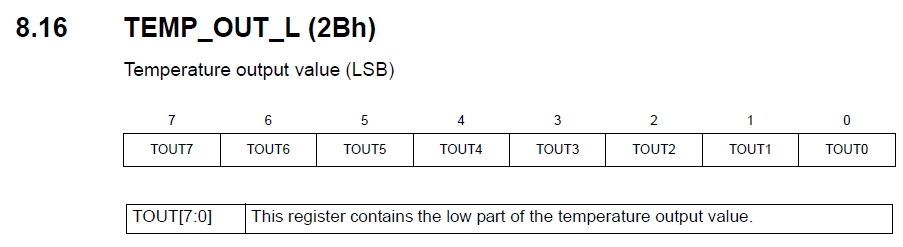
The function Mag\_GetXYZ is renamed to Press\_Get\_Temp, and also the type of the input parameter

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

void **Press\_Get\_Temp** (float \* pData)

{

To read the temperature readings, we use two registers



Uncomment the code and make changes to the reading function

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

        void Press\_Get\_Temp (float \* pData)

{

**int16\_t raw\_data = 0, tmp\_data = 0;**

        buffer [0] = **Press**\_IO\_Read ( **LPS25HB**\_ADDRESS, **LPS25HB\_TEMP\_OUT\_L\_REG**);

        buffer [1] = **Press**\_IO\_Read ( **LPS25HB**\_ADDRESS, **LPS25HB\_TEMP\_OUT\_H\_REG**);

**raw\_data = (((uint16\_t) buffer [1]) << 8) + (uint16\_t) buffer [0];**

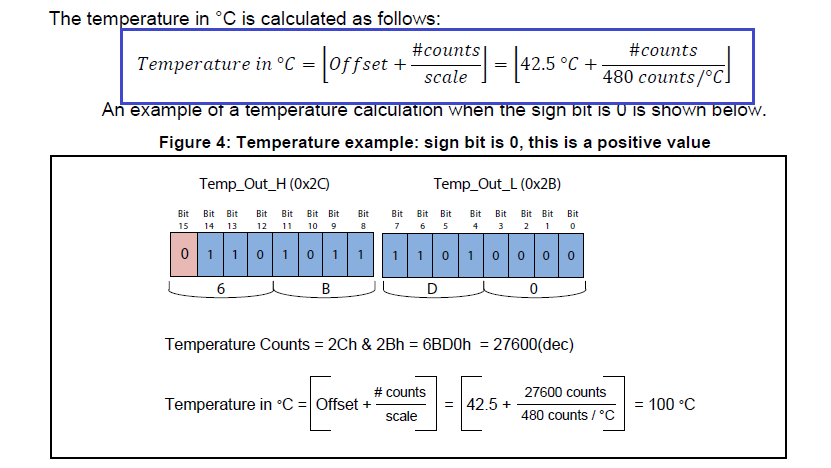
**tmp\_data = raw\_data / 48 + 425;**

**\* pData = (float) tmp\_data / 10.0f;**

}

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

Why this formula for calculating the temperature in degrees is described in detail in the documentation, only we use the multiplied 10 left side of the formula to exclude too many floating-point operations, and then divide the result by 10



In the [next part of](http://narodstream.ru/stm-urok-52-datchik-davlenija-lps25hb-chast-3/) this lesson we will write the function of reading and preprocessing the pressure value, also change the function code of the moving average filter.   
Also we will display the value of temperature and pressure (and immediately in three units in the terminal program).

**Lesson 52**

**Part 3**

# ****Pressure sensor LPS25HB****

In the [**previous part of the**](http://narodstream.ru/stm-urok-52-datchik-davlenija-lps25hb-chast-2/) lesson we wrote the initialization of the pressure sensor, and also wrote the function of reading and pretreatment of the temperature value.

The Mag\_Read function will also be renamed to Press\_Read

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

void **Press**\_Read (void)

{

Let's fix this in calls to main.c: both in the handler and in the infinite loop, and in the infinite loop at the same time and uncomment

  / \* USER CODE BEGIN 3 \* /

                Press\_Read ();

**// Press\_Read ();**

                HAL\_UART\_Receive\_IT (& huart2, (uint8\_t \*) str, 8);

We will now begin to make changes to this function. We will add a new variable, remove unnecessary variables and code, comment out the sending code in USART for the visualization program, and uncomment and make certain changes to the sending function in USART for the terminal program, and also add a delay

void Press\_Read (void)

{

**float temper;**

**Press\_Get\_Temp (& temper);**

        sprintf (str1, " **TEMP:% 06f**rn", **temper**);

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

        / \*

        buf2 [0] = 0x11;

        buf2 [1] = 0x55;

        buf2 [2] = (uint8\_t) (val [0] >> 8);

        buf2 [3] = (uint8\_t) val [0];

        buf2 [4] = (uint8\_t) (val [1] >> 8);

        buf2 [5] = (uint8\_t) val [1];

        buf2 [6] = (uint8\_t) (val [2] >> 8);

        buf2 [7] = (uint8\_t) val [2];

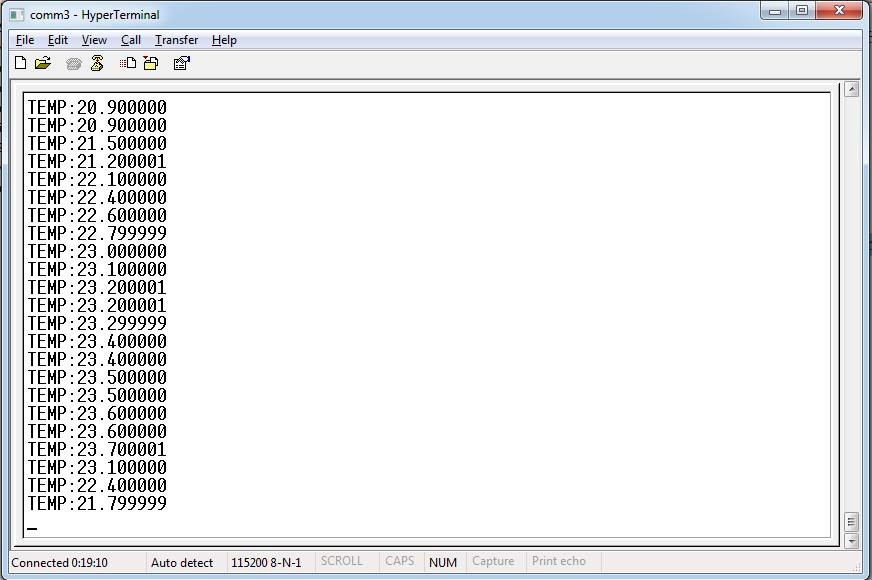
        HAL\_UART\_Transmit (& huart2, buf2.8.0 × 1000);

        \* /

        HAL\_Delay ( **200**);

}

We'll compile the code, we'll run the controller and see the result in the terminal program. In the process of viewing several times we will touch the sensor with your finger for a long time and see the temperature changes



We will also need a function to read the pressure readings from the sensor.

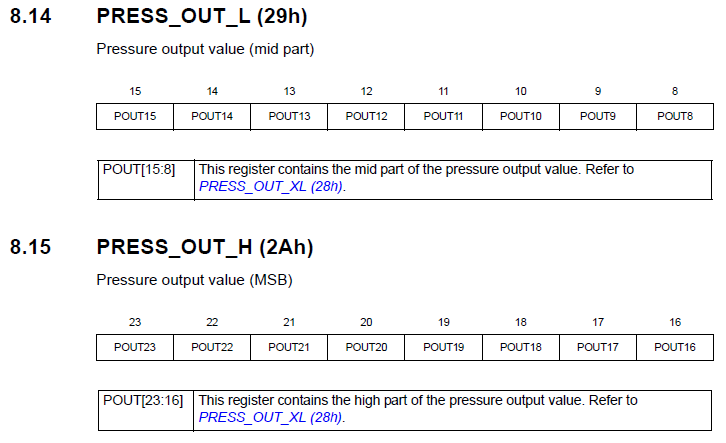
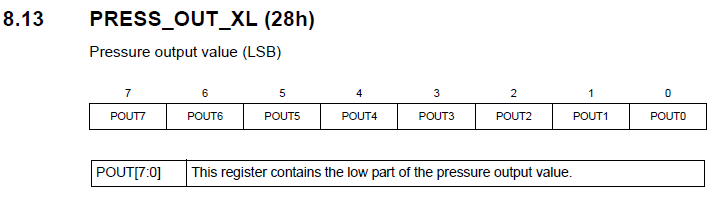
Copy the function Press\_Get\_Temp, paste it again into the code and fix it with the name

void Press\_Get\_ **Press** (float \* pData)

{

Let's start making changes to it.

Since we have a 24-bit pressure sensor, we use 3 registers



Therefore, we will use 3 elements of the buffer and 32-bit values ​​for calculation, one variable will be deleted, because calculations will be smaller due to the absence of a shift

void Press\_Get\_Press (float \* pData)

{

**uint8\_t i;**

**int32\_t** raw\_data = 0;

        buffer [0] = Press\_IO\_Read (LPS25HB\_ADDRESS, **LPS25HB\_PRESS\_OUT\_XL\_REG**);

        buffer [1] = Press\_IO\_Read (LPS25HB\_ADDRESS, **LPS25HB\_PRESS\_OUT\_L\_REG**);

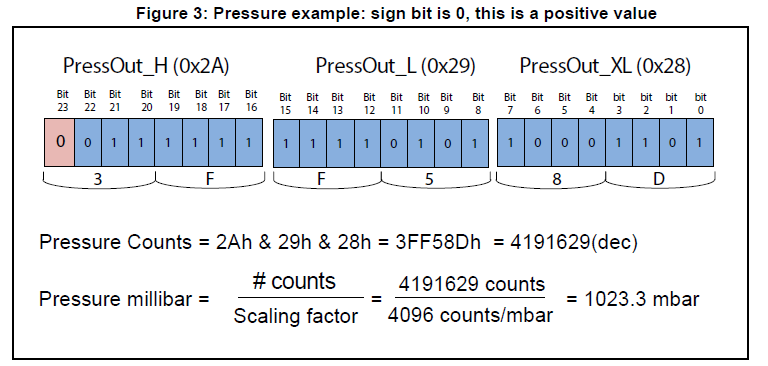
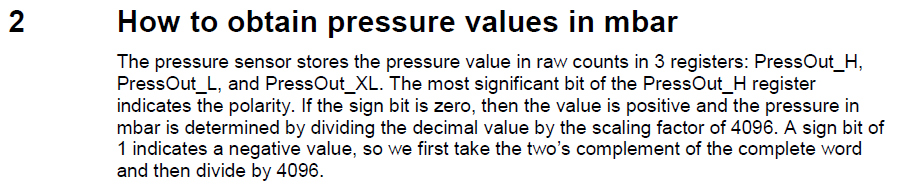
**buffer [2] = Press\_IO\_Read (LPS25HB\_ADDRESS, LPS25HB\_PRESS\_OUT\_H\_REG);**

**for (i = 0; i <3; i ++) raw\_data | = (((intint32\_t) buffer [i]) << (8 \* i));**

        \* pData = (float) **raw\_data** / **4096.0**f;

}

For calculation we also use technical documentation



Add some code to the high-level sensor processing function

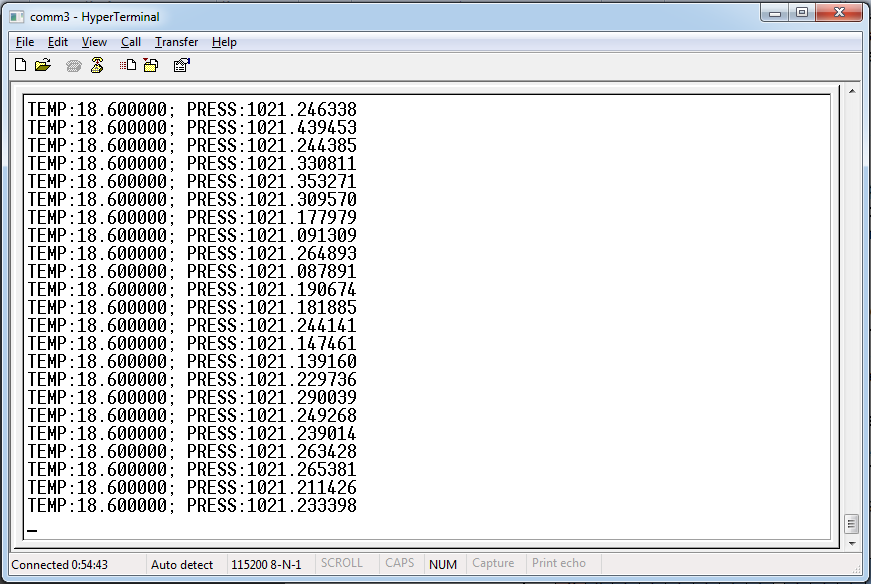
        float temper, **press**;

        Press\_Get\_Temp (& temper);

**Press\_Get\_Press (& press);**

        sprintf (str1, "TEMP:% 06f **; PRESS:% 06f**rn", temper **, press**);

We will collect the code, we will sew the MK and see the result



Now let's try to convert pressure readings to other units of measurement. We have it in hectopascals or in milibars, and we will transform it into atmospheres and millimeters of mercury. To do this, you will also need to add some code, but before we forget to add the cells to our string variable, otherwise I forgot and got a lot of bugs

uint8\_t buf2 [14] = {0};

**char str1 [100] = {0};**

And now the code

        float temper, press **, patm, pmmhg**;

        Press\_Get\_Temp (& temper);

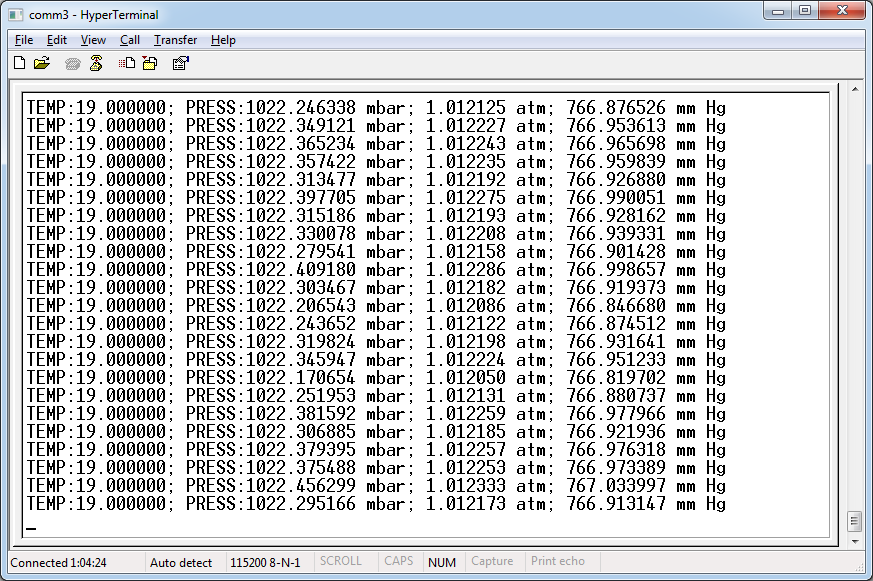
        Press\_Get\_Press (& press);

        patm = press / 1010;

        pmmhg = press / 1.333f;

        sprintf (str1, "TEMP:% 06f; PRESS:% 06f **mbar;% 06f atm;% 06f mm Hg**rn", temper, press **, patm, pmmhg**);

We'll compile the code, we'll write the controller and see the result of our work



Now remove the noise, which is almost not at the expense of the fact that it has already been removed.There is no temperature at all, the internal filter of the sensor works very well, but it is insignificant by the pressure. Well, let's still apply the moving average. Only use it will be more appropriate in low-level functions, since the already converted value in a floating-point value is heavier to filter and the filter uses more hardware resources, which I would not like at all. The temperature will be simpler, since the value is 16-bit, but with pressure, there will be another function due to the fact that the value is used 24-bit. But we'll go the other way and just correct our function of averaging values ​​to universal, just making it 32-bit.

To do this, first, correct the input argument, make it not an address but an ordinary variable, but then the return value will also appear

int32\_t MovingAverage (int **32**\_t \* dt)

Also, we will correct global variables, we will remove the axes, since we will only work with one value

// buffer for moving average

int volatile **32\_**t **buf\_avg**[8] = {0};

We also make this variable normal, not an array

// sum for the arithmetic mean

volatile int64\_t tmp64;

Add a variable to the function to return a value

int32\_t MovingAverage (int32\_t dt)

{

**int32\_t dt\_ret;**

In from the whole function with fixes

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

**int32\_t MovingAverage (int32\_t dt)**

**{**

**int32\_t dt\_ret = 0;**

**if (avg\_cnt <8)**

**{**

**buf\_avg [avg\_cnt] = dt;**

**if (avg\_cnt == 7)**

**{**

**tmp64 = buf\_avg [7] + buf\_avg [6] + buf\_avg [5] + buf\_avg [4] +**

**buf\_avg [3] + buf\_avg [2] + buf\_avg [1] + buf\_avg [0];**

**dt\_ret = tmp64 / 8;**

**}**

**avg\_cnt ++;**

**}**

**else**

**{**

**// subtract the first (oldest) element from the total sum**

**tmp64- = buf\_avg [0];**

**// shift the buffer to 1 element**

**memcpy ((void \*) buf\_avg, (void \*) (buf\_avg + 1), sizeof (int32\_t) \* 7);**

**// replace the 7 element with a new one**

**buf\_avg [7] = dt;**

**// add a new element**

**tmp64 + = dt;**

**// update the average value**

**dt\_ret = tmp64 / 8;**

**}**

**return dt\_ret;**

**}**

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

In the [**next part of the**](http://narodstream.ru/stm-urok-52-datchik-davlenija-lps25hb-chast-4/) lesson, we will already pass both readings, with the floating point through the USART bus on the PC, apply the moving average filter on them and see the change in the readings in the graphs.

**Lesson 52**

**Part 4**

# ****Pressure sensor LPS25HB****

In the [**previous part of the**](http://narodstream.ru/stm-urok-52-datchik-davlenija-lps25hb-chast-3/) lesson we wrote the function of reading and preprocessing the pressure values, and also changed the function code of the moving average filter.   
We also displayed the temperature and pressure (and immediately in three units in the terminal program).

Let's also add the ability to disable the moving average filter by clicking. Let's add a variable

void Press\_Get\_Press (float \* pData)

{

**uint8\_t btnstat;**

And add a poll button and apply a filter

        for (i = 0; i <3; i ++) raw\_data | = (((intint32\_t) buffer [i]) << (8 \* i));

**btnstat = HAL\_GPIO\_ReadPin (GPIOC, GPIO\_PIN\_13);**

**// If the button is not pressed,**

**// then call the moving average filter**

**if (btnstat! = 0) raw\_data = MovingAverage (raw\_data);**

        \* pData = (float) raw\_data / 4096.0f;

The same will be done for temperature. Only if we use the same function of the moving average filter, we can not avoid artifacts, since it will be called for temperature and for pressure in turn, but we work with global variables that store the previous readings. These indications will certainly get lost and nothing will come of it, even if we use local variables. Therefore, after long dances with a tambourine (structures, input values, etc.), it was decided to add global variables, except the counter to the filter function, then copy this function and add it again. Then add another counter specifically for the temperature, change the name of the added function, can not we have two functions with the same name,

And now in order.

Delete the global variables, add the temperature counter

// the counter of buffers of moving average

volatile int8\_t avg\_cnt;

**volatile int8\_t avg\_cnt\_temp;**

Add local variables to the function

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

int32\_t MovingAverage (int32\_t dt)

{

**// buffer for moving average**

**static int32\_t buf\_avg [8] = {0};**

**// sum for the arithmetic mean**

**static int64\_t tmp64; int32\_t dt\_ret = 0;**

        if (avg\_cnt <8)

Copy and correct the filter function

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

int32\_t MovingAverage **Temp**(int32\_t dt)

{

// buffer for moving average

static int32\_t buf\_avg [8] = {0};

// sum for the arithmetic mean

static int64\_t tmp64; int32\_t dt\_ret = 0;

        if ( **avg\_cnt\_temp**<8)

        {

                buf\_avg [ **avg\_cnt\_temp**] = dt;

                if ( **avg\_cnt\_temp**== 7)

                {

                        tmp64 = buf\_avg [7] + buf\_avg [6] + buf\_avg [5] + buf\_avg [4] +

                                                         buf\_avg [3] + buf\_avg [2] + buf\_avg [1] + buf\_avg [0];

                        dt\_ret = tmp64 / 8;

                }

**avg\_cnt\_temp**++;

        }

        else

        {

                // subtract the first (oldest) element from the total sum

                tmp64- = buf\_avg [0];

                // shift the buffer to 1 element

                memcpy ((void \*) buf\_avg, (void \*) (buf\_avg + 1), sizeof (int32\_t) \* 7);

                // replace the 7 element with a new one

                buf\_avg [7] = dt;

                // add a new element

                tmp64 + = dt;

                // update the average value

                dt\_ret = tmp64 / 8;

        }

        return dt\_ret;

}

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

Call the filter function in the temperature reading function

void Press\_Get\_Temp (float \* pData)

{

**uint8\_t btnstat;**

        int16\_t raw\_data = 0, tmp\_data = 0;

        buffer [0] = Press\_IO\_Read (LPS25HB\_ADDRESS, LPS25HB\_TEMP\_OUT\_L\_REG);

        buffer [1] = Press\_IO\_Read (LPS25HB\_ADDRESS, LPS25HB\_TEMP\_OUT\_H\_REG);

        raw\_data = (((uint16\_t) buffer [1]) << 8) + (uint16\_t) buffer [0];

**btnstat = HAL\_GPIO\_ReadPin (GPIOC, GPIO\_PIN\_13);**

**// If the button is not pressed,**

**// then call the moving average filter**

**if (btnstat! = 0) raw\_data = MovingAverageTemp (raw\_data);**

        tmp\_data = raw\_data / 48 + 425;

        \* pData = (float) tmp\_data / 10.0f;

}

Well and initialization

        // initialize the moving average fill counters

        avg\_cnt = 0;

**avg\_cnt\_temp = 0;**

        HAL\_Delay (1000);

Now we will try to see the result without the button and with the button, but already in the visualization program, which I again, as always, finished (well, I can without modifications). The completion is that we now directly obtain a floating-point value, modified the ranges for a floating point, and also included an edit field for setting the value shift. Well, the most important thing is that now we do not stupidly look at the cells, but there is a scale that already specifically talks about the testimony of y. The truth is for the time being only for two values, and at the same time with a large run-up of their values, we will not see them, but in the future I plan to finalize it and add a new scale to the right of the graph for the second value, and add a separate divisor and a separate shift for the second value. But so far it's already kind of cool.

But before we look at the result, we'll have to modify the code a bit to somehow send floating-point values ​​to the visualization program. To do this, first, comment out the call to the processing and sending function in the main.c file in an infinite loop and uncomment it in the timer handler

  / \* USER CODE BEGIN 3 \* /

**// Press\_Read ();**

  }

                if (huart2.RxXferCount == 0)

                {

**Press\_Read ();**

                        HAL\_UART\_Receive\_IT (& huart2, (uint8\_t \*) str, 8);

Now comment in the function of processing and sending data code for sending in text form, uncomment the code to send in binary form, comment out the delay, convert our sensor readings from the floating-point value into a normal binary memory, and also correct the sending to USART. We will now send 10 eight-digit values

        // sprintf (str1, "TEMP:% 06f; PRESS:% 06f mbar;% 06f atm;% 06f mm Hgrn", temper, press, patm, pmmhg);

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

**uint8\_t \* t = (uint8\_t \*) & temper;**

**uint8\_t \* p = (uint8\_t \*) & press;**

        buf2 [0] = 0x11;

        buf2 [1] = 0x55;

        buf2 [2] = **(uint8\_t) (\* (uint32\_t \*) t)**;

        buf2 [3] = **(uint8\_t) ((\* (uint32\_t \*) t) >> 8)**;

        buf2 [4] = **(uint8\_t) ((\* (uint32\_t \*) t) >> 16)**;

        buf2 [5] = **(uint8\_t) ((\* (uint32\_t \*) t) >> 24)**;

        buf2 [6] = **(uint8\_t) (\* (uint32\_t \*) p)**;

        buf2 [7] = **(uint8\_t) ((\* (uint32\_t \*) p) >> 8)**;

**buf2 [8] = (uint8\_t) ((\* (uint32\_t \*) p) >> 16);**

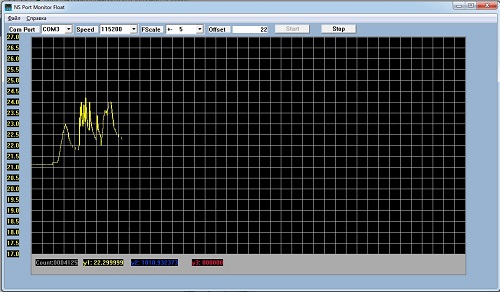
**buf2 [9] = (uint8\_t) ((\* (uint32\_t \*) p) >> 24);**

        HAL\_UART\_Transmit (& huart2, buf2, **10**, 0x1000);

**//**HAL\_Delay (200);

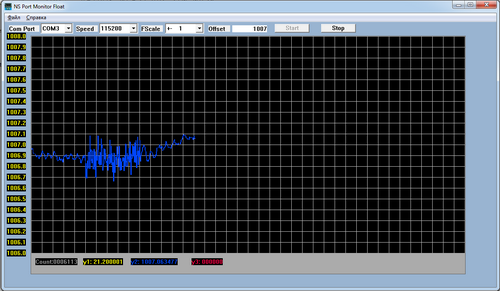
}

We compile the code, let's go through the controller and look at the result with the button and without the button in the visualization program, also applying different values ​​of limits and shifting the readings (click on the image to increase the size)

[](http://narodstream.ru/wp-content/uploads/2016/11/image08_1152.jpg)

By measuring the temperature, we bring the finger of the hand to the sensor gradually and also gradually remove it, first without the button then with the button. A small difference, but still there.

Now the pressure. It's unchanged, since it's hard to affect the atmospheric pressure, unless you put the debug card into the car tire and pump it up (joke). Also, try without the button and with the button, applying other values ​​of the divisor and shift in the visualization program (click on the image to increase the size)

[](http://narodstream.ru/wp-content/uploads/2016/11/image11_1153.png)

Here, of course, the difference is obvious. The first and last five cells are with a moving average filter, and the middle five cells are without a filter. So the application of the filter gives its results.