**Lesson 92**

**Part 1**

**Temperature sensor DS18B20**

Today we will consider the work with the **DS18B20** temperature sensor , which, despite its apparent simplicity at first glance, gained a very wide spread due to its characteristics. The positive characteristics of this sensor include its low energy consumption, simplicity of design due to the presence of its TO-92 body, only three legs, and also an inexpensive cost, which is also an important condition for its wide distribution.

With this sensor we already worked, connecting it to the microcontroller AVR, in [**lesson 20**](http://narodstream.ru/avr-urok-20-podklyuchaem-datchik-temperatury-ds18b20-chast-1/) .

Then came a flurry of comments and requests in groups and in a personal with the desire to see a lesson with the subtleties of connecting and programming this sensor in conjunction with the STM32 controller. For a long time I did not dare to take this step. But on the threshold of work with technologies of wireless data transmission, I did not find a more interesting solution for what data by means of these technologies to transmit, that is, what kind of payload. This load is useful in the literal sense. Since it is often the task to collect some kind of device that can measure the temperature (as well as humidity, pressure, etc.) and transfer it to some kind of shared server. Of course, there are many such devices. But there are a lot of people who want to understand the essence of this process, and also organize this process at their own discretion.

For this purpose, this sensor and controller comes to us.

This sensor is connected via a 1-WIRE bus, that is, data transfer is carried out in both directions with only one wire, which of course introduces certain difficulties in the process of writing the code. The fewer wires, the more difficult it is to organize data transfer.

Moreover, one can hang several such devices, and not necessarily the same, and on this occasion there were even more requests for writing a lesson for me.

And I hasten to boast that I have also mastered this process and I will certainly share with you my experience in one of the following sessions.

Now a little about the sensor in more detail.

Some characteristics of the sensor:

The measuring range is from -55 0C to +125 0C, but the highest accuracy of the readings is achieved in the range from -10 0C to +85 0C.

The error of this sensor is 0.5 degrees.

The measurement resolution is from 9 to 12 bits of data.

Dependent on the established resolution (resolution), the duration of the temperature measurement is from 93.75 milliseconds to 750 milliseconds.

A unique 64-bit serial code, thanks to which several such sensors can be used on one bus.

We will connect this sensor to the microcontroller **STM32F103C8T6** , located on an inexpensive debugger board, with which we have worked quite a bit recently.

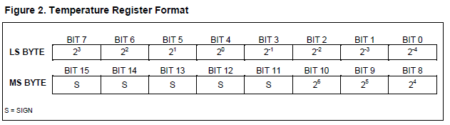
This controller does not have hardware support for the **1-wire** bus , which makes our occupation even more interesting, as there were questions about how this process can be organized if there is no hardware support and the port feet using functions of the HAL library do not react as quickly to commands as for the AVR. We also have to solve this problem, and at the same time learn to use the operative switching of the port legs (bareheaded).

This sensor exists in two types of housing - DIP and TO-92. We will use the second type.

Since we will only use one sensor, we will address it in a different way, without using the ROM.

We feed the sensor command 44h with a sequential code, thereby causing the sensor to convert the temperature.

Let's look at the registers in which the temperature value is stored after conversion



The four least significant bits of the low register store fractions of degrees, in the four least significant bits, and in the three low-order bits of the upper register - integer degrees.

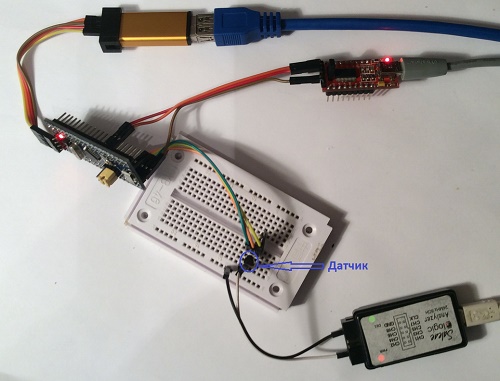
In the remaining bits - a sign. If 0, then plus, if 1 - then negative.

Also in the technical documentation it is written that it is necessary to tighten the resistor to the information leg of the sensor to the power supply. The recommended value is 4.7 kilo.

Also written are a number of time and other characteristics that we will already consider as we write the code.

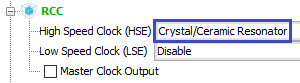
In general, we will have the following scheme. Place the sensor in the breadboard. Between the information output of the sensor and the power terminal we connect a resistor to 4.7 kilo-ohms. Power supply to the power supply 3.3 volts from the debug board, and connect the information output of the sensor to the foot PB11 of the microcontroller. We also connect the **USART-USB** module without supplying a positive power supply to it and also connect an inexpensive ST-Link V2 programmer, and supply power to the controller from it. We will not use any separate power supplies, since the sensor consumes almost nothing, unlike, for example, the LAN module ENC28J60. We also connect a logic analyzer to the information output of the sensor to investigate the behavior of this output.

Get this assembly (click on the image to enlarge the image)

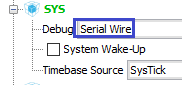
[](http://narodstream.ru/wp-content/uploads/2017/09/Image00.jpg)

Create a new project in the **Cube MX** by selecting the **STM32F103C8Tx** controller .

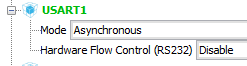
Let us set the clock from the quartz resonator



Turn on the debugger



Set up USART



We plug in the leg of the PC13 port, which is responsible for the LED, to the output

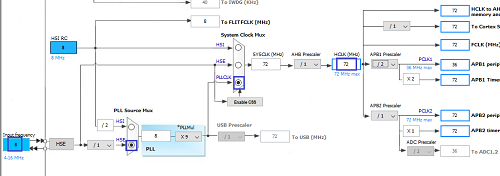
Image04

We also include the foot PB11 on the input, to which we will connect the information contact of the sensor

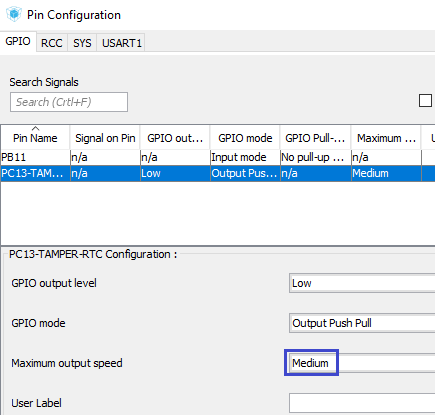


In principle, this leg could not be turned on, anyway we will initialize it in our own way. But still include, so that you can not forget about it later and do not want to use it for anything else.

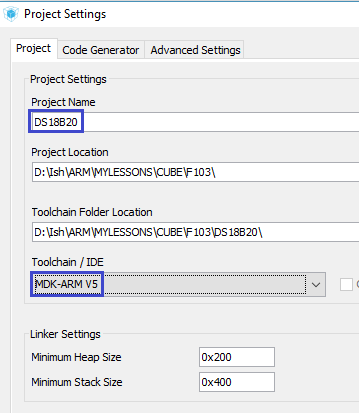
Let's go to the **Clock Configuration** section and adjust the frequencies and switches there as follows (click on the picture to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/09/Image06.png)

Now go to the Configuration section, USART will not be touched, we will leave the default settings (speed 115200 kbps), but the PC13 port foot will be adjusted slightly faster



Now go into the project settings, select there as the programming environment Keil 5, and also name the project by the name of the sensor



Apply the settings, collect the project, open it in Keil, configure the programmer for auto-cutting, and reduce the optimization level to level 1.

Let's try to assemble the project.

Let's create two files **ds18b20.h** and **ds18b20.c as** follows

**ds18b20.h** :

**#ifndef DS18B20\_H\_**

**#define DS18B20\_H\_**

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

**#include "stm32f1xx\_hal.h"**

**#include <string.h>**

**#include <stdlib.h>**

**#include <stdint.h>**

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

**#endif /\* DS18B20\_H\_ \*/**

**ds18b20.c** :

**#include "ds18b20.h"  
//--------------------------------------------------**

We connect our future library for the sensor in the **main.h** file

/\* USER CODE BEGIN Includes \*/

**#include "ds18b20.h"**

/\* USER CODE END Includes \*/

In the file  **ds18b20.c** add a delay function in microseconds

#include "ds18b20.h"

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

**\_\_STATIC\_INLINE void DelayMicro(\_\_IO uint32\_t micros)**

**{**

**micros \*= (SystemCoreClock / 1000000) / 9;**

**/\* Wait till done \*/**

**while (micros--) ;**

**}**

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

The divisor **9** was selected experimentally using a logic analyzer.

Now we need to somehow learn how to control the leg of PB11, creating high and low states on it if necessary.

But before doing this, we need to adjust this leg and initialize it.

Add the function for this

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

**void port\_init(void)**

**{**

**HAL\_GPIO\_DeInit(GPIOB, GPIO\_PIN\_11);**

**GPIOB->CRH |= GPIO\_CRH\_MODE11;**

**GPIOB->CRH |= GPIO\_CRH\_CNF11\_0;**

**GPIOB->CRH &= ~GPIO\_CRH\_CNF11\_1;**

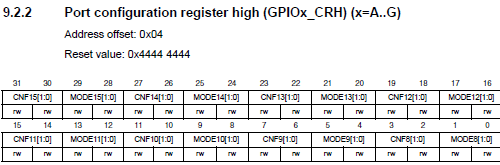
**}**

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

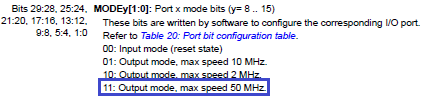
For a long time we did not work with the CMSIS library, but it does not frighten us. It is somewhat similar to the standard library for working with ports of AVR controllers.

First we cancel the existing initialization, then configure the configuration register of the **CRH** port . Rather, it is not a register, but only the older half of one large **CR** register . The lower half of the register serves to adjust the port feet from 0 to 7, and the older one from 8 to 15. Since we have leg 11, we, accordingly, require CRH.

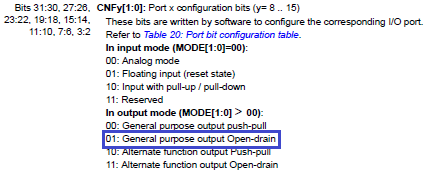
This is how the register looks in the documentation for the controller (Reference Manual)



First, initialize the bit pair MODE11 with a mask, including a high level in both bits, which will correspond to the mode to output at a speed of 50 megahertz



Then we put [01] into the CNF bit pair, which corresponds to the open collector in the output mode



We add a prototype to this function and call it in **main ()**

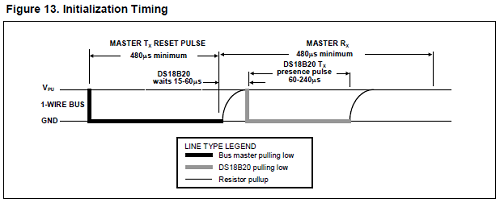
/\* USER CODE BEGIN 2 \*/

**port\_init();**

/\* USER CODE END 2 \*/

The port was initialized, now we will write the initialization of the sensor.

This process is also described in the documentation



Moreover, this procedure will have to occur not only when the sensor is initially turned on, but almost constantly, at the beginning of certain actions, for example starting a temperature measurement. Therefore, the code for it will be written in the next separate function, which we now add, having returned to the file **ds18b20.c**

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

**uint8\_t ds18b20\_Reset(void)**

**{**

**uint16\_t status;**

**}**

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

Judging by the documentation, it is obvious that we are attracting the bus here, we store it in this state for at least 480 microseconds, then release it, wait for a maximum of 60 μs (we will wait more than 65 μs) of the sensor response, the sensor should respond with the same pull of the tire to the ground for the given time. If a zero is detected, then there is a device on the bus, and if the bus remains suspended in the air, then there is no sensor on the bus.

Well. and on the skeleton of the above-read, we continue our function of determining the presence of a sensor on the bus.

We expose a certain level on the leg with the help of the corresponding bit in the **ODR** register , and we check - using the bit register **IDR** . And it should be noted that it is not necessary to reconfigure the foot to the input, which significantly reduces the execution time of the code

uint8\_t ds18b20\_Reset(void)

{

  uint16\_t status;

**GPIOB->ODR &= ~GPIO\_ODR\_ODR11;//низкий уровень**

**DelayMicro(485);//задержка как минимум на 480 микросекунд**

**GPIOB->ODR |= GPIO\_ODR\_ODR11;//высокий уровень**

**DelayMicro(65);//задержка как минимум на 60 микросекунд**

**status = GPIOB->IDR & GPIO\_IDR\_IDR11;//проверяем уровень**

**DelayMicro(500);//задержка как минимум на 480 микросекунд**

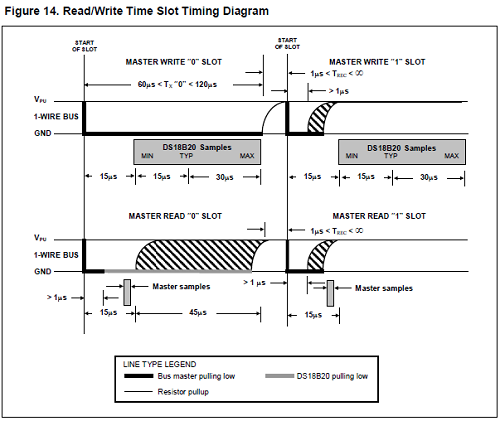
**//(на всякий случай подождём побольше, так как могут быть неточности в задержке)**

**return (status ? 1 : 0);//вернём результат**

}

Before proceeding to the initialization procedure, you need to learn how to send data to the sensor, well, at the same time, although this is not required in the initialization, and read the data from the sensor.

Let's see how this process should occur



And we will write the functions of reading and writing bits, and also bytes, in the [**next part of**](http://narodstream.ru/stm-urok-92-datchik-temperatury-ds18b20-chast-2/) our lesson, in which we will also get acquainted with the organization of sensor memory.

**Lesson 92**

**Part 2**

# Temperature sensor DS18B20

In the [**previous part of the**](http://narodstream.ru/stm-urok-92-datchik-temperatury-ds18b20-chast-1/) lesson we got acquainted with the sensor, created and configured the project and started writing the function of initializing the temperature sensor.

Let's start with the bit reading function. We will write it immediately after the body of the delay function.

Judging by the picture from the documentation, after pulling the bus to the ground, we wait at least one microsecond, well, wait two microseconds just in case, before pulling the bus to the ground

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

**uint8\_t ds18b20\_ReadBit(void)**

**{**

**uint8\_t bit = 0;**

**GPIOB->ODR &= ~GPIO\_ODR\_ODR11;//низкий уровень**

**DelayMicro(2);**

**}**

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

We look further. We need to release the tire from the ground, and then delay to read the result. Here there is a total delay of 15 microseconds, since we have already waited two, we have 13

DelayMicro(2);

**GPIOB->ODR |= GPIO\_ODR\_ODR11;//высокий уровень**

**DelayMicro(13);**

After this delay, the result should already be on the bus. We consider it similarly, as in the previous function we recognized the presence of an agent on the bus

DelayMicro(13);

**bit = (GPIOB->IDR & GPIO\_IDR\_IDR11 ? 1 : 0);//проверяем уровень**

Before reading the next byte, judging by the diagram, we are waiting for 45 microseconds, then we return the result of the function of reading the bit on the bus

  bit = (GPIOB->IDR & GPIO\_IDR\_IDR11 ? 1 : 0);//проверяем уровень

**DelayMicro(45);**

**return bit;**

}

Now, using this function, we'll write the function of reading from the sensor of the whole byte, sequentially bytes to the variable c, applying a bit shift, and then returning it.

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

**uint8\_t ds18b20\_ReadByte(void)**

**{**

**uint8\_t data = 0;**

**for (uint8\_t i = 0; i <= 7; i++)**

**data += ds18b20\_ReadBit() << i;**

**return data;**

**}**

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

Only these functions are not enough for us. Since the sensor will never give anything to read until it receives a specific command. And to pass a command, you need to write the function of writing a byte. And for this you must first write the function of writing (sending) a bit to the bus.

In the diagram of recording a bit in the device that we looked at in the sensor datasheet, it is clear that if we transmit a unit, then we draw the bus to the ground, wait for 1 microsecond (wait just in case 3), then release the bus, wait about 60 microseconds (wait 65), and can transmit the next bit. If we transmit 0, then we attract the bus, wait for 60 to 120 microseconds, wait 65, then release the bus. If you simplify the code a little, you will get something like this

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

**void ds18b20\_WriteBit(uint8\_t bit)**

**{**

**GPIOB->ODR &= ~GPIO\_ODR\_ODR11;**

**DelayMicro(bit ? 3 : 65);**

**GPIOB->ODR |= GPIO\_ODR\_ODR11;**

**DelayMicro(bit ? 65 : 3);**

**}**

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

Using this function, we will write the function of sending a whole byte to the bus

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

**void ds18b20\_WriteByte(uint8\_t dt)**

**{**

**for (uint8\_t i = 0; i < 8; i++)**

**{**

**ds18b20\_WriteBit(dt >> i & 1);**

**//Delay Protection**

**DelayMicro(5);**

**}**

**}**

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

Here, also using the bit shift, we gradually send a bit by bit, starting from the youngest one, to the 1-WIRE bus.

Well now, when we wrote the main routine functions, it's time to take up the initialization of the sensor itself. To do this, add the following function at the very bottom of the file

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

**uint8\_t ds18b20\_init(uint8\_t mode)**

**{**

**return 0;**

**}**

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

Let's create a prototype for this function.

The function, besides the outgoing argument for the return status, has an incoming argument. This is the mode. We will have two modes. One mode is to skip reading a unique identifier when we are working with a single sensor on the bus, and another mode is when to read and then use the identifiers when we work with several sensors (two or more).

For this purpose, create macros in the file **ds18b20.h**

**#include <stdint.h>**

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

**#define SKIP\_ROM 0**

**#define NO\_SKIP\_ROM 1**

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

We will use the SKIP ROM mode. Let's return to the file **ds18b20.c** and start writing the body of the initialization function

uint8\_t ds18b20\_init(uint8\_t mode)

{

**if(ds18b20\_Reset()) return 1;**

**if(mode==SKIP\_ROM)**

**{**

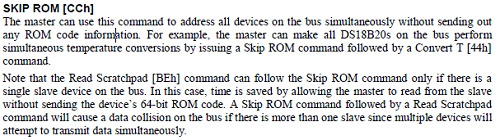
**}**

  return 0;

}

We rebooted the sensor, making sure it was present on the bus and created a condition for the mode without reading the unique code.

In order to skip the code and so that the sensor understands that we are not going to read the code and access it, we must send the appropriate command to it



Call this command in the function

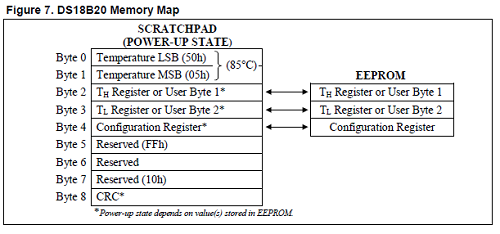
if(mode==SKIP\_ROM)

{

**//SKIP ROM**

**ds18b20\_WriteByte(0xCC);**

Also let's see how the memory is organized in our sensor



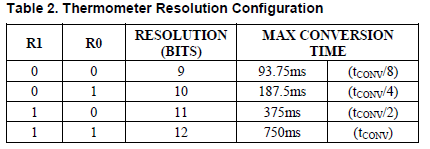
The first 2 bits (the lowest) are the value of the last read temperature. Format we have already considered.

The following 2 bits of **Th** and **Tl** are the upper and lower threshold registers of the measured temperature. Rather, it will be measured outside of these thresholds, but alarm signals will already come from the sensor. The next is the configuration register

Image16

It stores practically only the useful information of the resolution of the readable temperature in bits - from 9 to 12.

This is how this bit is adjusted. At the same time, the table gives the time of reading the temperature (conversion)



Using the register and the table for the convenience of writing the code, go back to the file **ds18b20.h**and add there macro substitutions for variants of discreteness

#define NO\_SKIP\_ROM 1

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

**#define RESOLUTION\_9BIT 0x1F**

**#define RESOLUTION\_10BIT 0x3F**

**#define RESOLUTION\_11BIT 0x5F**

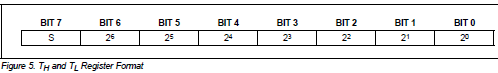
**#define RESOLUTION\_12BIT 0x7F**

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

In the next three bytes of memory there is nothing (they are reserved), and the latter contains a checksum.

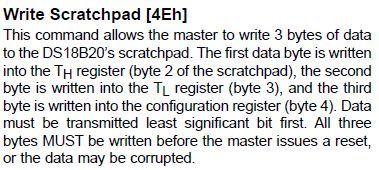
When you turn on 3 tuning bytes, information comes from non-volatile memory. But then we can make our variants of values ​​there, and moreover, there is even a special command with which we can save the user settings to non-volatile memory.

Let's look at the boundary temperature format for threshold registers



In the most significant bit, the sign is stored (if the unit is minus, if zero is positive), and in the remaining value the temperature is without fractional parts.

We will also need another command to write values ​​to memory (in StratchPad)



This command writes the values ​​that we send to the bus, sequentially in the size of three bytes in the registers **Th** , **Tl** and in the configuration register.

Knowing now all this information, we can calmly add the sensor initialization function

  ds18b20\_WriteByte(0xCC);

**//WRITE SCRATCHPAD**

**ds18b20\_WriteByte(0x4E);**

**//TH REGISTER 100 градусов**

**ds18b20\_WriteByte(0x64);**

**//TL REGISTER - 30 градусов**

**ds18b20\_WriteByte(0x9E);**

**//Resolution 12 bit**

**ds18b20\_WriteByte(RESOLUTION\_12BIT);**

}

return 0;

All that we have written here is reflected in the comments.

Let's write a prototype for this function and go to the main.c file and create there first a global array

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

**char str1[60];**

/\* USER CODE END PV \*/

Then the local variable in **main ()**

/\* USER CODE BEGIN 1 \*/

**uint8\_t status;**

/\* USER CODE END 1 \*/

Also, in this function, we call our sensor initialization function and output the result to the terminal program window

port\_init();

**status = ds18b20\_init(SKIP\_ROM);**

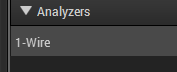
**sprintf(str1,"Init Status: %drn",status);**

**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)str1,strlen(str1),0x1000);**

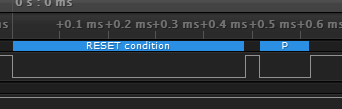
We will collect the code, we will sew the controller and see the result in the terminal program

Image20

A zero in the terminal indicates that the initialization was normal and the sensor responded. We will also see this more clearly in the program of logical analysis. We will configure this program to recognize our data exchange protocol

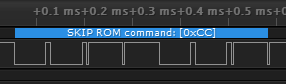


We see the response of the sensor here on the diagram of the behavior of the legs

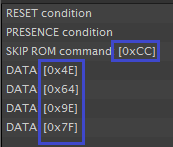


The first depression is our impulse with a length of 480 microseconds - and the second is the impulse sent to us by the sensor.

Now the recorded bytes. Consider the transmission of each bit in more detail



We see on the chart our recognized SKIP ROM command sent by us in the form of a hexadecimal number 0xCC, which in binary form will look like 11001100. But we remember that everything is passed backwards, beginning with the low bit. Large cavities are transmitted zeros, and small ones are units, and we see from left to right the sequence 00110011. That's the way the bits are transmitted. Also we see that all our five sent bytes were also recognized by the analyzer



It is very good!

But I would like to take something from the sensor.

Let's do this gradually.

We consider not only the temperature, but in general the entire Strathpad.

We return to the file  **ds18b20.c** and add there the function of passing the command to measure the temperature

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

**void ds18b20\_MeasureTemperCmd(uint8\_t mode, uint8\_t DevNum)**

**{**

**ds18b20\_Reset();**

**if(mode==SKIP\_ROM)**

**{**

**//SKIP ROM**

**ds18b20\_WriteByte(0xCC);**

**}**

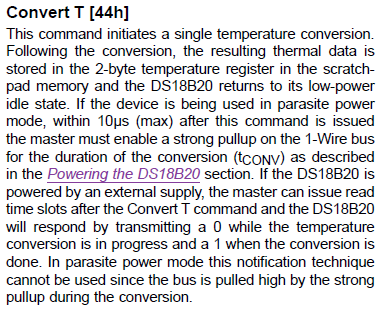
**//CONVERT T**

**ds18b20\_WriteByte(0x44);**

**}**

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

This function also starts with a reboot. I think, here we will not check the presence of the device on the bus, anywhere it will not go away. Then we also check the condition of calling our mode, we again give a command to skip reading the code, and then give the command to read (convert or convert, as in ADC) the temperature



The second input argument to the function is the device number. This is done for the future, because we are going to use several sensors on the bus.

In the [**next part of the**](http://narodstream.ru/stm-urok-92-datchik-temperatury-ds18b20-chast-3/) lesson we will write a few more functions and finish our code for reading and displaying the temperature from the sensor.

**Lesson 92**

**Part 3**

# Temperature sensor DS18B20

In the [**previous part of the**](http://narodstream.ru/stm-urok-92-datchik-temperatury-ds18b20-chast-2/) lesson we got acquainted with the organization of sensor memory, and also wrote several other necessary functions.

The next function that we need is a function of reading the memory, we need to know the temperature.

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

**void ds18b20\_ReadStratcpad(uint8\_t mode, uint8\_t \*Data, uint8\_t DevNum)**

**{**

**uint8\_t i;**

**ds18b20\_Reset();**

**if(mode==SKIP\_ROM)**

**{**

**//SKIP ROM**

**ds18b20\_WriteByte(0xCC);**

**}**

**//READ SCRATCHPAD**

**ds18b20\_WriteByte(0xBE);**

**for(i=0;i<8;i++)**

**{**

**Data[i] = ds18b20\_ReadByte();**

**}**

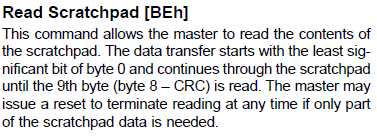
**}**

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

This function also does not need a special explanation. The input arguments are the address of the array into which we will read 8 bytes of memory, the next is the mode, and the last is the device number.

We also reboot our first sensor on the condition mode give to skip reading of ROM, and then click We serve **0xBE** , which causes the device to give us bytes of memory, which we then read in the cycle in our array.

Here is an explanation of the memory read command from the technical documentation



Add to these two functions the prototypes in the header file.

Let's go to the main () function and add some more local variables

uint8\_t status;

**uint8\_t dt [8];**

**uint16\_t raw\_temper;**

**float temper;**

**char c;**

Then we'll add the following code in an infinite loop

/\* USER CODE BEGIN 3 \*/

**ds18b20\_MeasureTemperCmd(SKIP\_ROM, 0);**

**HAL\_Delay(800);**

**ds18b20\_ReadStratcpad(SKIP\_ROM, dt, 0);**

**sprintf(str1,"STRATHPAD: %02X %02X %02X %02X %02X %02X %02X %02X; ",**

**dt[0], dt[1], dt[2], dt[3], dt[4], dt[5], dt[6], dt[7]);**

**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)str1,strlen(str1),0x1000);**

**sprintf(str1,"rn");**

**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)str1,strlen(str1),0x1000);**

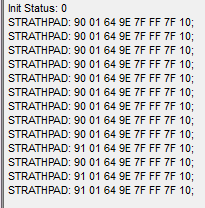
**HAL\_Delay(150);**

}

/\* USER CODE END 3 \*/

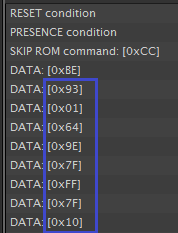
We first give the command to measure and convert the temperature, wait for 800 milliseconds (just in case, put at least 750 for our discreteness), Then we give a command to read the memory and display it in hexadecimal form in the terminal program. Then, for the time being, we separately move the line and translate the carriage, and also wait a little before the next reading. All together will be about a second. For such an accurate reading of the temperature is very good.

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

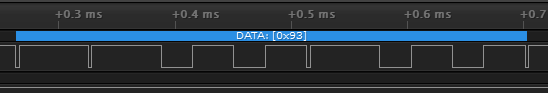


That's what we read. The first two bytes are the temperature value. In our case, 0x9101. you can even freely calculate the temperature. In binary terms, these 2 bytes will look like 0000000110010001. It turns out that the temperature is positive, since the first five most significant bits are in our zeros. In the next seven bits, we have the result 0011001, which in decimal will be 25, and in the lowest four bits - 0001, this value is 1/16 degree fraction. it turns out that the temperature is 25 and 1/16 degrees. I have a thermometer in the clock, which shows about so much and shows a little more. So the testimony is like the truth.

See also reading the bytes in the logical analysis program. Here is our Strathpad



Also, let's see how exactly the bits appear on the graph



And they look similarly written down.

Here we read 0x93, which in binary rotation will be 10010011. We read from right to left. The way it is. The units are again in our form in the form of short impulses directed downwards, and the zeros are long.

But we will not calculate the temperature every time, so go back to  **ds18b20.c** and add the function of determining the result sign

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

**uint8\_t ds18b20\_GetSign(uint16\_t dt)**

**{**

**//Проверим 11-й бит**

**if (dt&(1<<11)) return 1;**

**else return 0;**

**}**

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

Here everything is simple. It is enough for us to test any of the 5 highest bits. check the 11th. Especially since it's something we can check for a long time, we also have the 11th leg of the port (magic, you will not say anything). And if he is in the unit, then we will return it, otherwise we will return zero.

Also we will write another function of converting the temperature value itself into a floating-point result

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

**float ds18b20\_Convert(uint16\_t dt)**

**{**

**float t;**

**t = (float) ((dt&0x07FF)>>4); //отборосим знаковые и дробные биты**

**//Прибавим дробную часть**

**t += (float)(dt&0x000F) / 16.0f;**

**return t;**

**}**

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

Here, too, everything is simple. Disable all the signed bits with a mask, move the whole sequence 4 places to the right, separating the fractional part, then divide the value of the 4 lower bits by 16 and add to the result. That's all. Very easy.

Add our prototypes to our functions in the header file and go to the infinite loop of the **main ()** function .

Delete the carriage return and linefeed code there and add the following code instead

HAL\_UART\_Transmit(&huart1,(uint8\_t\*)str1,strlen(str1),0x1000);

**raw\_temper = ((uint16\_t)dt[1]<<8)|dt[0];**

**if(ds18b20\_GetSign(raw\_temper)) c='-';**

**else c='+';**

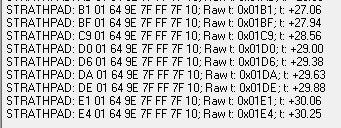
**temper = ds18b20\_Convert(raw\_temper);**

**sprintf(str1,"Raw t: 0x%04X; t: %c%.2frn", raw\_temper, c, temper);**

**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)str1,strlen(str1),0x1000);**

HAL\_Delay(150);

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



Now everything is clear!

I touched the sensor with my finger and held it. The temperature began to rise.

So. Today we learned how to work with the temperature sensor DS18B20, connected via the 1-WIRE bus to the STM32 controller without any hardware support by jerking the port leg. Also we studied a number of commands of this sensor, we were able to count the temperature from it. While this is the only sensor on the bus, but I hope that we will cope with several. Especially since I have already prepared such a sensor.

Wait for the next lessons.

Thank you all for attention!