**STM Lesson 94. DS18B20. We connect several sensors to the wire. Part 1**

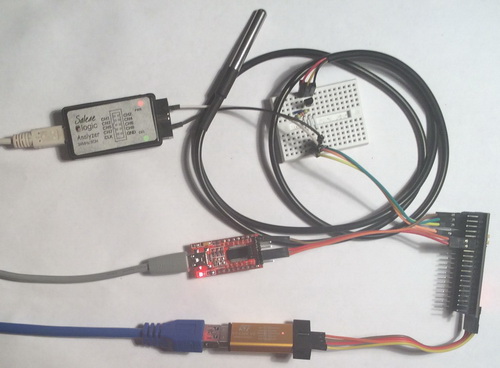
Posted on [October 20, 2017](http://narodstream.ru/stm-urok-94-ds18b20-podklyuchaem-neskolko-datchikov-na-provod-chast-1/)by [http://1.gravatar.com/avatar/4824b24065500834db4b9f331b608833?s=32&d=mm&r=gNarod Stream](http://narodstream.ru/author/admin/) Published in [Programming STM32](http://narodstream.ru/rub_stm32/)

We continue to work with a temperature sensor **DS18B20** , which is connected via a **1-** wire **1-WIRE** bus . Rather not with the sensor, but with several sensors connected to one wire. Or rather not with a few, but with two, because I do not have any more.

This lesson was very much in demand, and, as I promised in the lesson on this sensor, it did not take long.

The first sensor we already had on the breadboard, though I replaced it with a smaller one, and the second sensor is somewhat different in my screen and on the wire. Such variants also exist in a wide sale and are used in more severe conditions (unprotected from atmospheric and other influences).

Now our diagram looks like this (click on the image to enlarge the image):

[](http://narodstream.ru/wp-content/uploads/2017/10/image00.jpg)

The second sensor is connected completely parallel to the first (all three terminals).

Let's start now to sort out, as it is possible to give commands and receive data from two sensors simultaneously on one wire. As you know, there is a unique serial 64-bit code for this, which we have to count from both sensors today in order to use these codes separately.

The project will be created entirely on the basis of project  **DS18B20** **lesson 92** and assign it the name **DS18B20\_MORE\_SENSORS** .

Open our project in Cube MX and, without touching it, we will generate a project for Keil and also open it there. Set up the programmer for auto- **reload, enable the** optimization level 1, connect the file **ds18b20.c** and think how we consider this serial identifier. It turns out that this is not quite easy. There is a command **Read ROM** with code **0x33**, which reads the serial code, it is not complicated. We send it to the sensor and the sensor gives us 8 treasured bytes. But there is one small problem: this command works only if we have only one sensor connected. You can connect the sensors in turn, read these codes from them and then connect them together and use these codes, but, you see, for an average user, for whom we design the device, it will be, to put it mildly, difficult. Therefore, there is another command that reads identifiers from all the sensors connected at the same time. This is the **Search Rom** command with code **0xF0**. But here, too, it's not so simple, I'll have to write a very difficult algorithm, since the first time this command does not work. That's what we'll do today. Therefore, there was a separate lesson for this.

Also, to find out in detail how the identifier search command works, there is already another documentation developed not for our sensor alone, but for the **1-WIRE** interface using **iButton** technology . **Application Note 937** . If you want to know even more than what I give you here, you can easily find this book.

We will gradually deal with this search. In the meantime, let's create a function in the file **ds18b20.c**above the sensor initialization function

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

**uint8\_t ds18b20\_SearhRom(uint8\_t \*Addr)**

**{**

**uint8\_t id\_bit\_number;**

**uint8\_t last\_zero, rom\_byte\_number, search\_result;**

**uint8\_t id\_bit, cmp\_id\_bit;**

**uint8\_t rom\_byte\_mask, search\_direction;**

**//проинициализируем переменные**

**id\_bit\_number = 1;**

**last\_zero = 0;**

**rom\_byte\_number = 0;**

**rom\_byte\_mask = 1;**

**search\_result = 0;**

**return search\_result;**

**}**

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

In the function, we created and initialized a series of variables, which we will need later in this function. As an input argument, we will transfer the address of an eight-byte array into which the function will write the next serial code of a certain sensor.

Add two global variables to the **main.c** file

char str1[60];

**uint8\_t Dev\_ID[8][8]={0};**

**uint8\_t Dev\_Cnt;**

One variable will be for storing several serial numbers (I think we will not connect more than eight sensors, and if we do, we will change the first dimension), and the second for the number of sensors on the bus, which we then define programmatically.

Also we will add some global variables to the file **ds18b20.c** , and also connect the two variables added above

#include "ds18b20.h"

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

**uint8\_t LastDeviceFlag;**

**uint8\_t LastDiscrepancy;**

**uint8\_t LastFamilyDiscrepancy;**

**uint8\_t ROM\_NO[8];**

**extern uint8\_t Dev\_ID[8][8];**

**extern uint8\_t Dev\_Cnt;**

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

Let's return to our function **ds18b20\_SearhRom**  and write the following condition

search\_result = 0;

**if (!LastDeviceFlag)**

**{**

**ds18b20\_Reset();**

**ds18b20\_WriteByte(0xF0);**

**}**

In the body of this condition, we get if the flag of the last device is not installed. We'll reload our sensors here and send them to all **Search Rom** teams .

Well, now let's take a look at the process of searching for serial numbers. All this is described in the above-mentioned book, but I'll try to bring it out in a nutshell and gradually, also taking out the knowledge in the code.

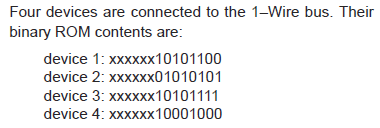
After sending the **Search Rom** command **(0xF0),** all devices will consistently form on the bus state **0** and **1** , corresponding to the value of their actual bit in the ROM. The reading process begins with the least significant bit of the code and takes two time steps. If all the devices on the bus in the next read bit contain **0**, it will be sequentially present on the bus **01** . If both devices contain **1** , then the result will be **10** . All this takes place if all devices have the same bits in this position. And if at least one device has a different result, it will result in the result **00**, indicating a conflict. Next, in the next (in the third) clock cycle, the controller sends 0 or 1 to select devices that match this bit with the bit contained in its identifier. Devices that do not match the bit will go into the standby state and will remain in it until they receive a reset pulse. Then there is the next similar cycle of three measures for the next bit. The total number of such cycles will be 64, and the clock will be 192, during which the controller will determine the code of one of the devices, the rest will necessarily be in the standby mode, then it is unique - this code. For this code, the controller can then quietly then access this device. That is, the remaining cycles will already pass without conflicts and will already have a certain result.

Next, we still have devices that we have not yet considered the code for. Therefore, we repeat this procedure first and must go to the cycle when the controller last time caught the conflict, but we will not send 0 but 1 in the next cycle, which will lead to a waiting mode for another device, it will be that device whose code we considered in the first procedure.

Next, we repeat the procedure again until the last conflict (well, or the penultimate one, if we consider the last one the last) and behave in the same way as when reading the code of the second device, and this action will put both devices in the standby mode, the codes of which we considered in the second stage .

And so we act by repeating the procedures until there are no conflicts at all.

Of course, without practice, I understand that this process can not be understood. Therefore, the documentation gave a concrete example, in which 4 devices on the 1-WIRE bus were identified



For simplicity, all bits of the identifiers are not shown and the high-order bits are indicated by the symbols **x** .

The reading process, as I already said, takes place from the smallest byte. Let's try it on this example.

1. The controller generates a reset pulse, to which all devices respond with a presence pulse.

2. The controller sends a 0xF0 (Search ROM) command.

3. The controller reads one bit from the bus, to which the devices respond **00** , which allows us to talk about a bit conflict, meaning that the devices in a given bit have different states.

4. The controller sends bit **0** to the bus, which will cause devices 2 and 3 to be idle and also to the fact that in all subsequent operations before resetting, device data will no longer respond.

5. The controller reads the next bit, receiving **01** , since the result for the 1 and 4 devices that remained able to respond to the commands are the same bits and equal to 0.

6. The controller sends bit **0** to the bus in order for both devices to remain in the active mode, not to disconnect them all.

7. The controller reads the next bit again using two read cycles and receives **00** , which corresponds to the bit conflict, again allowing to judge that the bits of the devices are different.

8. The controller again sends bit **0** to the bus, which will cause device 1 to go into standby mode and as a result, only one device will be able to respond. But the fact that it is one, we just know knowingly. But in practice we do not yet know this. Therefore, we continue reading to the oldest 63 bits.

9. The rest of all read cycles will not lead to conflicts, since we will not have any different bits on the bus, since we have only one device, thus we consider one identifier to the end.

10. Now we have to read the identifier of the next device, for which we must repeat steps 1 to 7, respectively, activating all the devices again.

11. At the most senior position, in which the controller wrote 0 after the last conflict was written, the controller is already writing 1. This will already lead to the transition to the standby mode already device 4, which will allow us to read the ROM code of the device 1.

12. Read the device code 1 to the end.

13. Next, we need to read the ROM of the next device (one of devices 2 or 3). For this, we already repeat steps from 1 to 3.

14. At the most senior position, in which the controller wrote 0 after detecting the last conflict in the second pass, the controller is already writing 1. This will lead to the transition to the standby mode of the device 1 and 4.

15. Next, read the next bit, getting **00** , which means that the bits for devices 2 and 3 are different.

16. The controller sends to bus **0** , which will cause device 3 to be turned off, leaving facility 2 as the only active one.

17. Read the device code 2 to the end.

18. Next, we must read the identification code of the last device 3, performing steps 13 - 15.

19. In the oldest position in which the controller wrote 0 after detecting the last conflict in the third pass, the controller is already writing 1. This will cause the device 2 to go into standby mode, leaving device 3 active.

20. Read the ROM code of the device 3 to the end.

Now we have the codes of all 4 devices. If there are more devices on the bus, then there will be more passes and steps, therefore, too.

I hope that now you have a better understanding of the process of reading codes.

Also we will have the opportunity to fix it once again when we invent an algorithm when writing the code search code, also fixing it visually, looking at the results of logical analysis on the bus.

In the [**next part of the**](http://narodstream.ru/stm-urok-94-ds18b20-podklyuchaem-neskolko-datchikov-na-provod-chast-2/) lesson, we will write completely the function of finding the identifiers of the sensors and consider the unique codes from all the temperature sensors.

**STM Lesson 94. DS18B20. We connect several sensors to the wire. Part 2**

Posted on [October 25, 2017](http://narodstream.ru/stm-urok-94-ds18b20-podklyuchaem-neskolko-datchikov-na-provod-chast-2/)by [http://1.gravatar.com/avatar/4824b24065500834db4b9f331b608833?s=32&d=mm&r=gNarod Stream](http://narodstream.ru/author/admin/) Published in [Programming STM32](http://narodstream.ru/rub_stm32/)

[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/KC7hy-eNRCS50D01CPcuari00000ECgs7402I09Wl0Xe172WmvR10O01sE3CgmU80QE8tP0fa06cakID99W1eiM-tYEW0SAapOqag07iWR3U8xW1tAUPz1N00GBO0PQmhH3W0UQKqmxe0HRu0RQLthu1Y083e0BUkAe2kG97LQzfFxPypF02zjh0lGtu0eA0W820u1EO0wKVg0CIi0C4k0J_0UW4omJu18-50OW5ZuK1a0Maym6W1VWdg0Ni8R05x26u1Umzm0NmcGV81R3g0T05dG_W1J_m1G6O1e3GhFCEe0Rk0gW6xWB91iOharf9W7KJqGR6lyZHIO1r4za60000S740002f1_XHJ8ZmgiD8i0U0W90Cq0S2u0U62lW70O080T08keg0WS2GW0BW2A-SbG602W712W0000000F0_s0e2u0g0YNhu2i3y5OWB1geB4BMnWh3uKG00dB-Aorfr1G302u2Z1SWBWDIJ0TaB-55CYF2gmqZe2u-50V0B1eWCjfNUlW7e30AO3PMCQF8D0FeD088E08aE00000000y3-G3i24FPWEnjVCr9M9uBeJe0x0X3sm3W6X3m0000000F0_g0_ue-7q-jtguKW0?stat-id=3&test-tag=86861447979009&format-type=24&banner-test-tags=eyI0ODQ2MjkwMDY3IjoiODY4NjE0MTg2MjcwNzIifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

[Switches contactlessSwitches **contactless inductive** type VBITo learn moreferrol.ru](https://an.yandex.ru/count/47n3XFUkOvK50Bu1CPguari00000ECgs7402I09Wl0Xe172Osh300u01mFES28W1ix6H6901riMv19W1a9pW1BW1vlVV1C010jW1-9uxu06QWEG2w04s-04EY084e0BanGcv0aTLhsa_jdpCy0AAp9210lW2We20W83W4u03bhEWY0w80xUChhm9c0Eco0om0mJe1Bi2-0Ixa0681RkG0P05tQ02e0MhAwW5tIIm1Tqak0MpIS05oSO6o0MPrG7G1SCGu0K-y0K1c0QirvQ00Q06xWAe1ku2oGR6AvDQIO1r4z46nh_8qKc0THFP1W00071n0000gGVuKVXn5wl3IB07W82G3D070k07XWhu1m60207G29gAW870a802u0ZouWI02W712W0000000F0_s0e2u0g0YNhu2i3y5OWB2AeB45b5CiJuKG00wdcKorfr1G302u2Z1SWBWDIJ0TaB-57uSHUhmqZe2xkG0V0B1eWCZltUlW7e30AO3VozPl8D0FeD088E08aE00000000y3-G3i24FPWEnjVCr9M9uBeJe0x0X3sm3W6X3m0000000F0_g0_uez6jv-BguKW0?stat-id=4&test-tag=86861447979009&format-type=24&banner-test-tags=eyI0NjY1NzA3IjoiODY4NjE0MTg1OTQzMDQifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

In the [**previous part of the**](http://narodstream.ru/stm-urok-94-ds18b20-podklyuchaem-neskolko-datchikov-na-provod-chast-1/) lesson we studied how to use the Search ROM command of 1-WIRE devices to read the identification codes of several devices on the same bus, created and configured the project.

Let's **go** into the initialization function **ds18b20\_init** , changing the code a bit there.

While we transfer the reset of the devices in the condition

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

*{*

*if(mode==SKIP\_ROM)*

*{*

***if(ds18b20\_Reset()) return 1;***

*//SKIP ROM*

Add to this function several local variables

uint8\_t ds18b20\_init(uint8\_t mode)

{

**int i = 0, j=0;**

**uint8\_t dt[8];**

  if(mode==SKIP\_ROM)

In the same function, we will now start processing one more condition, when we do not skip reading the ROM code

  ds18b20\_WriteByte(RESOLUTION\_12BIT);

}

**else**

**{**

**for(i=1;i<=8;i++)**

**{**

**if(ds18b20\_SearhRom(dt))**

**{**

**Dev\_Cnt++;**

**memcpy(Dev\_ID[Dev\_Cnt-1],dt,sizeof(dt));**

**}**

**else break;**

**}**

**}**

return 0;

In the body of this condition, we will read the code until it is the last (until the function returns the normal result), and copy the last read code into our two-dimensional array for later use. But since the search function for the identifier is still only begun, it will always return 0, which will lead to the fact that it will be executed only once, which we actually need to start studying the response of devices on the bus.

Go to the file main.c, remove all the code in an infinite loop and change the input argument in the call to the initialization function of the sensor

status = ds18b20\_init(**NO\_**SKIP\_ROM);

We return to the file **ds18d20.c** in the search function of the identifier **ds18b20\_SearhRom** and in the body of our condition we add a cycle of reading 8 bytes of code

  ds18b20\_WriteByte(0xF0);

**do**

**{**

**} while(rom\_byte\_number < 8); // считываем байты с 0 до 7 в цикле**

}

And in the body of the loop, we count 2 bits, as we agreed in our read instruction, and place the reading results in different variables, and then go out of the loop forcibly. This is a temporary measure, so that we can see the process in the analysis program and that at that time the code is not obsessed with

do

{

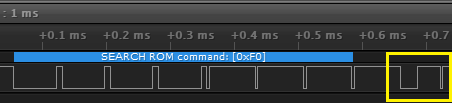
**id\_bit = ds18b20\_ReadBit();**

**cmp\_id\_bit = ds18b20\_ReadBit();**

**break;**

} while(rom\_byte\_number < 8); // считываем байты с 0 до 7 в цикле

We will collect the code, we will tell the controller and see in the analyzer what bits we received



And we got **01** , which corresponds to the fact that both our devices (and we know that we have only 2 of them) have the same low bit. And moreover, we can judge by the result that this bit is precisely **0** .

Go ahead. Now we must send bit 0 to the bus.

But before you send, let's filter a little more. Suddenly, 11 will come to us, which in general can not be. Then we will leave the cycle

  ds18b20\_WriteByte(0xF0);

  cmp\_id\_bit = ds18b20\_ReadBit();

**if ((id\_bit == 1) && (cmp\_id\_bit == 1))**

**break;**

  break;

}

Then we write the opposite case (which for us will be the opposite good

  break;

**else**

**{**

**}**

break;

Then we begin to filter further in the body of the else statement. This is the case, if something normal has come (not **11** ).

else

{

**if (id\_bit != cmp\_id\_bit)**

**search\_direction = id\_bit; // bit write value for search**

}

By this code, we processed the condition if the bits are not equal and put the value of the first bit in a certain variable. That is, this is the value of the bit that is recorded in both sensors.

Then the opposite case of this condition. That is, the case when **00** came to us from the bus , that is, when the bits in the devices are not equal. We will add this case at once with the code, with which then we will understand

    search\_direction = id\_bit; // bit write value for search

**else**

**{**

**if (id\_bit\_number < LastDiscrepancy)**

**search\_direction = ((ROM\_NO[rom\_byte\_number] & rom\_byte\_mask) > 0);**

**else**

**search\_direction = (id\_bit\_number == LastDiscrepancy);**

**if (search\_direction == 0)**

**{**

**last\_zero = id\_bit\_number;**

**if (last\_zero < 9)**

**LastFamilyDiscrepancy = last\_zero;**

**}**

**}**

}

First, we process the condition here, that this is not the bit where the last mismatch occurred in the last pass. In this case, we write to the result from the previous pass. But at us the variable responsible for number of a bit of the last mismatch is equal to zero, if we entered the function for the first time, accordingly to us while to read all this from nowhere and we while here do not get.

We'll get into the opposite case, in the body of which we put into the result the value of the variable **id\_bit\_number** , which we initialized as **1** .

Next we will examine the result by zero. If true, we write the value of the above variable in the variable **last\_zero** (last zero).

Then we compare this variable with 9 and if the value is less, then we also assign this value to the **LastFamilyDiscrepancy** variable , which will carry the last bit of the vendor's device mismatch (in the least significant byte a unique identifier of the device type is stored). We will then see that this byte will be the same for both devices. We do not need this variable, but let it be. This can be useful for if we use our library for different devices and further code will depend on the type of device.

We go further on the function. Let us exit the two conditions and add the following code

      LastFamilyDiscrepancy = last\_zero;

    }

  }

**if (search\_direction == 1)**

**ROM\_NO[rom\_byte\_number] |= rom\_byte\_mask;**

}

This code, provided the result with a value of 1, will insert it into the desired byte bit according to the mask in which we have 1 so far, that is, we are working with 0 bit so far.

Otherwise, we discard this bit

    ROM\_NO[rom\_byte\_number] |= rom\_byte\_mask;

**else**

**ROM\_NO[rom\_byte\_number] &= ~rom\_byte\_mask;**

}

And then we send this bit to the bus

    ROM\_NO[rom\_byte\_number] &= ~rom\_byte\_mask;

**ds18b20\_WriteBit(search\_direction);**

}

Let's see what we can do. To do this, we will compile the code, we will sew the controller and see the result in the program of logical analysis

image02

We see that 0 of us has left the bus, which we are doing all the time. This is also the first pass, and secondly - we do not need to disconnect the device with a single.

Further everything is simple. We increment the bit number and shift the mask, since in the next repetition of the loop we will already read another bit

  ds18b20\_WriteBit(search\_direction);

**id\_bit\_number++;**

**rom\_byte\_mask <<= 1;**

}

Next we explore the exhaustion of the mask, that is, the case when we ran through all the bits of the next byte

  rom\_byte\_mask <<= 1;

**if (rom\_byte\_mask == 0)**

**{**

**rom\_byte\_number++;**

**rom\_byte\_mask = 1;**

**}**

}

In this case, we increase the byte number and again in the current mask set the lowest bit (bit 0).

Remove our temporary break, since we have already completed the loop to the end

  }

~~break;~~

} while(rom\_byte\_number < 8); // считываем байты с 0 до 7 в цикле

Leaving the loop body, filtering out that we can reach the number of bits greater than 64, we do not need this, since our identifier is exactly 64 bits

  } while(rom\_byte\_number < 8); // считываем байты с 0 до 7 в цикле

**if (!(id\_bit\_number < 65))**

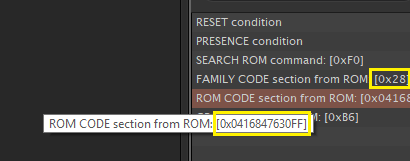
**{**

**}**

}

return search\_result;

Let's collect the code and let's say the controller. Let's see the result of our work in the program of logical analysis



We see that the device code, as well as the device type code, and therefore the checksum we have read. True, the code is read only one device, but it's already good. Simply, we have not yet determined the number of devices. That is, we need in the event that there was at least one discrepancy between the bits in the pass, to increment the number of devices on the bus, and also to do something with some variables and flags.

While we do not increment anything.

Let's write in the function **ds18b20\_SearhRom**  the body code of the read-back condition to the end of all 64 bits

if (!(id\_bit\_number < 65))

{

**// search successful so set LastDiscrepancy,LastDeviceFlag,search\_result**

**LastDiscrepancy = last\_zero;**

**// check for last device**

**if (LastDiscrepancy == 0)**

**LastDeviceFlag = 1;**

**search\_result = 1;**

}

In this code, we assign the value of the bit variable of the last mismatch, and then in case there were no mismatches in this pass, then set the last device flag and assign the value to the result of the search 1. This value does not depend on whether the last device is or not last thing. This means only that we have successfully reached 63 bits.

Further we come out of all conditions and cycles and write one more condition together with the body

    search\_result = 1;

  }

}

**if (!search\_result || !ROM\_NO[0])**

**{**

**LastDiscrepancy = 0;**

**LastDeviceFlag = 0;**

**LastFamilyDiscrepancy = 0;**

**search\_result = 0;**

**}**

return search\_result;

In this condition, we get in case we do not get to 63 bits or when in the lowest byte (code of the device type) we will have all zeros. In this case, we assign zeroes to all of our flags, including the return result variable. But, I hope that we will never get here.

Therefore, we leave the body of this condition and rewrite our bytes into our two-dimensional array

    search\_result = 0;

  }

**else**

**{**

**for (int i = 0; i < 8; i++) Addr[i] = ROM\_NO[i];**

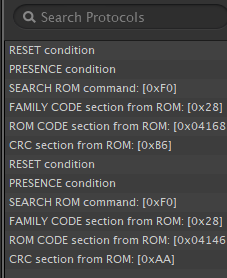
**}**

  return search\_result;

}

On this our long-suffering function of code search is finished.

Once again we will compile the code, we will sew the controller and see the result in the program of logical analysis



We see that we have the codes of both sensors read. Unfortunately, this program does not respond to the expansion of areas, so I can not show them completely, although by bringing the mouse cursor to each of them, we can still see them.

But despite this shortcoming, we will still see them. Do not forget that in addition to the logical analysis program, we still have a terminal program, in which, of course, not everything can be visually seen as in a program for a logical analyzer, but still we will see all the codes.

In the [**next part of the**](http://narodstream.ru/stm-urok-94-ds18b20-podklyuchaem-neskolko-datchikov-na-provod-chast-3/) lesson we will finish the code of all our functions and check all our code in practice by measuring the temperature with two sensors and displaying the data in the terminal program.

**STM Lesson 94. DS18B20. We connect several sensors to the wire. Part 3**

Posted on [October 26, 2017](http://narodstream.ru/stm-urok-94-ds18b20-podklyuchaem-neskolko-datchikov-na-provod-chast-3/)by [http://1.gravatar.com/avatar/4824b24065500834db4b9f331b608833?s=32&d=mm&r=gNarod Stream](http://narodstream.ru/author/admin/) Published in [Programming STM32](http://narodstream.ru/rub_stm32/)

In the [**previous part of the**](http://narodstream.ru/stm-urok-94-ds18b20-podklyuchaem-neskolko-datchikov-na-provod-chast-2/) lesson we wrote completely the function of finding the identifiers of the sensors and counted the unique codes from all the temperature sensors.

Let's move on to the function **main ()** the file **main.c** and add a local variable to the counter

char c;

**uint8\_t i;**

Then, after calling the initialization function, we show our results in the terminal program

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

**sprintf(str1,"Dev count: %d\r\n", Dev\_Cnt);**

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

**for(i=1;i<=Dev\_Cnt;i++)**

**{**

**sprintf(str1,"Device %d\r\n", i);**

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

**sprintf(str1,"ROM RAW: %02X %02X %02X %02X %02X %02X %02X %02X\r\n",**

**Dev\_ID[i-1][0], Dev\_ID[i-1][1], Dev\_ID[i-1][2], Dev\_ID[i-1][3],**

**Dev\_ID[i-1][4], Dev\_ID[i-1][5], Dev\_ID[i-1][6], Dev\_ID[i-1][7]);**

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

**sprintf(str1,"Family CODE: 0x%02X\r\n", Dev\_ID[i-1][0]);**

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

**sprintf(str1,"ROM CODE: 0x%02X%02X%02X%02X%02X%02X\r\n", Dev\_ID[i-1][6], Dev\_ID[i-1][5],**

**Dev\_ID[i-1][4], Dev\_ID[i-1][3], Dev\_ID[i-1][2], Dev\_ID[i-1][1]);**

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

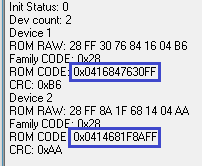
**sprintf(str1,"CRC: 0x%02X\r\n", Dev\_ID[i-1][7]);**

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

**}**

**/\* USER CODE END 2 \*/**

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



It remains for us to learn how to separately send commands to each device in order to read the current temperature readings from them.

Let's move on to the function **ds18b20\_MeasureTemperCmd** file **ds18b20.c** and add it to start in a local variable

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

{

**int i = 0;**

  ds18b20\_Reset();

In this function, we have a condition for the case when we skip reading the ROM code. Add a nasty case also immediately with the body

ds18b20\_WriteByte(0xCC);

}

**else**

**{**

**//Match Rom**

**ds18b20\_WriteByte(0x55);**

**for(i=0;i<=7;i++)**

**{**

**ds18b20\_WriteByte(Dev\_ID[DevNum-1][i]);**

**}**

**}**

//CONVERT T

Here we use the **Match Rom** command with the code **0x55** , after which we transfer the entire ROM-code of the device together with the vendor and the checksum. Therefore, we do not need to generate it anew. And after that all the commands and all the readings before the reset of the devices will belong to the device whose code is transmitted to the bus.

We proceed as follows. We will give the command to read the temperature to all devices, then wait for the right time and only then will we read them from all temperatures. This is done in order not to waste time. That is, the conversion of the read temperature will occur simultaneously in all sensors.

Back in the **main.c** file and in the endless loop of the **main ()** function, let's all our sensors start the command to convert the temperature value

/\* USER CODE BEGIN 3 \*/

**for(i=1;i<=Dev\_Cnt;i++)**

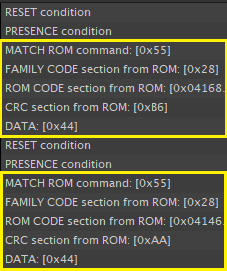
**{**

**ds18b20\_MeasureTemperCmd(NO\_SKIP\_ROM, i);**

**}**

**HAL\_Delay(800);**

We will collect the code, we will tell the controller, and we will see how there is a reference to different devices in the program of logical analysis



Let's return to the **ds18b20.c** file in the **ds18b20\_ReadStratcpad** function and the **code** already known to us from the previous function in it. The variable for the counter we already have

  ds18b20\_WriteByte(0xCC);

}

**else**

**{**

**//Match Rom**

**ds18b20\_WriteByte(0x55);**

**for(i=0;i<=7;i++)**

**{**

**ds18b20\_WriteByte(Dev\_ID[DevNum-1][i]);**

**}**

**}**

//READ SCRATCHPAD

Also we did not finish something in the initialization function. We did not initialize some configuration registers in the sensor, because we did not know how to handle them and with which code.

Let's go into this function from the cycle of reading ROM codes from sensors and write another cycle that will write down the necessary values ​​in the registers of all our sensors

    else break;

  }

**for(i=1;i<=Dev\_Cnt;i++)**

**{**

**if(ds18b20\_Reset()) return 1;**

**//Match Rom**

**ds18b20\_WriteByte(0x55);**

**for(j=0;j<=7;j++)**

**{**

**ds18b20\_WriteByte(Dev\_ID[i-1][j]);**

**}**

**//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;

We return to the infinite loop of the main () function of the main.c file and add the following code there for reading and processing the readings

  HAL\_Delay(800);

**for(i=1;i<=Dev\_Cnt;i++)**

**{**

**ds18b20\_ReadStratcpad(NO\_SKIP\_ROM, dt, i);**

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

**i, 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);**

**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%.2f\r\n", raw\_temper, c, temper);**

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

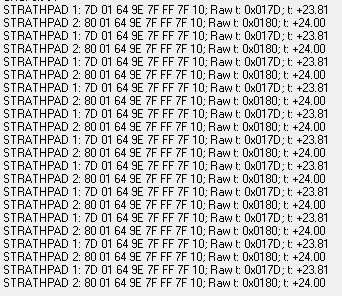
**}**

**HAL\_Delay(150);**

}

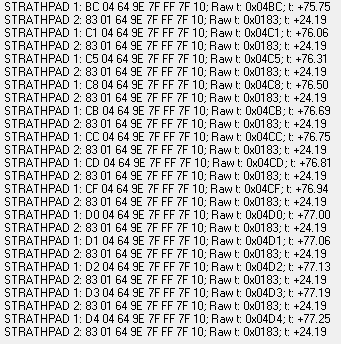
/\* USER CODE END 3 \*/

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



We see how the temperature from both sensors is read alternately.

Let's put one of the sensors in a glass of boiling water and see the reaction of the program



We see that the temperature readings of the first sensor are steadily increasing, and the readings of the second sensor are not changing.

This confidently allows us to verify that. that we wrote everything correctly and by the same to make sure that we learned how to work with several devices on the 1-WIRE bus, giving them separate commands, reading out unique codes from them regardless of the number of such devices, and also contacting each of them separately .

Thank you all for your attention!