**HAL I2C Connect real time clock DS3231**

Today we will try using the **HAL** library to work with the **I2C** bus .

In advance, we determine that we will connect the **DS3231** real-time clock chip to this bus .

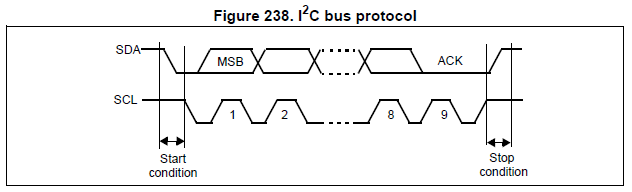
First, let's talk a little about the I2C bus itself.

On this bus, I talked a lot in a series of parts of [**lesson 16**](http://narodstream.ru/avr-urok-16-interfejs-twi-i2c-chast-1/) on **AVR** , so here we will get acquainted more briefly so as not to repeat.

The I2C bus is a bus, controlled by two wires and by a certain protocol.

The first wire is **SDA (Serial DAta))** , and the second is **SCL (Serial CLock)** . The data is mainly transmitted over the SDA wire. The second wire is mainly for clocking,

The transmission always begins with the formation of the **START** condition on the given bus by the master . It is formed by forming a negative front on the SDA bus, and then a negative front on the SCL bus



At the end of any complete package, the **STOP** condition is generated , which, on the contrary, first requires a positive front on the SCL, and then on the SDA.

The I2C bus can theoretically hang up to 127 devices due to the fact that each device has its own 7-bit address, which, after the STOP condition, the **master** (master) transmits to the bus. And only the **SLAVE**(slave) device responds to this address , the address of which was transmitted. After the address, a bit is transmitted. which determines exactly how we are going to communicate with the slave device, that is, we will read it from it or write to it. Next, the Master waits for confirmation from the slave, the slave should lower the SDA bus to a low level, and if it is transmission, then it starts transmitting data accordingly, and if receiving, then take it.

How exactly and in which order byte-by-bit and bit by bit reception and transmission are performed, we, as a rule, look already in the technical documentation for the device that we are connecting. There can also be addresses of memory cells and registers, they can be 16 and 8 bit and a lot more.

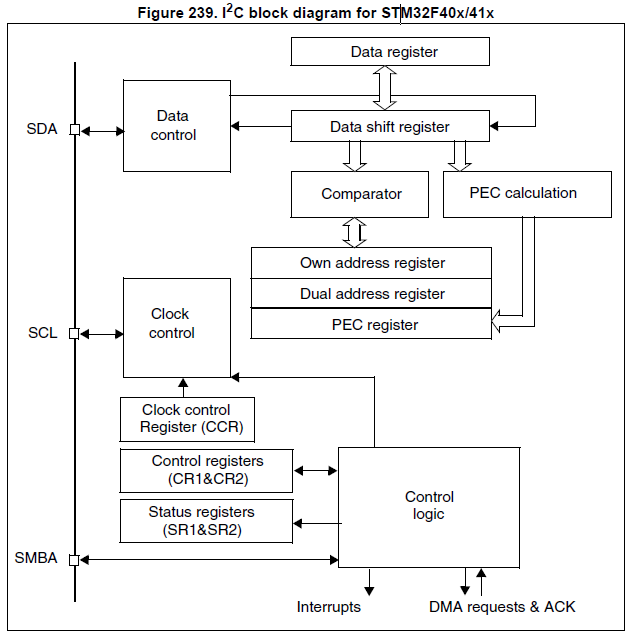
It is also worth mentioning that the wires of this bus can be said to hang in the air, since an open collector is used, and therefore, it is necessary to tighten them to the power bus through a resistor with a resistance of 4.7 - 10 kilograms. You can also do this programmatically. And this denomination should not be adhered to on every device, if there are several of them, but on the whole bus. That is, if we connected 10 devices with different addresses, then pull-up resistors we leave only on one of them, and on the rest we have to evaporate them so that they do not parallel.

Before we proceed to the analysis of the 3231 chip, we still need to know exactly how the I2C bus is organized on our controller.

Firstly, the frequency of data transmission or bitrate is supported by a controller of 100 kHz and 400 kHz. We already decide which one to choose, based on the capabilities of the connected device and our requirements for data transmission.

And, of course, it's worth emphasizing that the I2C bus is organized by itself on our STM32F4 controller hardware, and there are several of them.

Here is the flow chart



Here we see our SDA and SVL wires, there are also registers that we set up for operation, a data register, an address register, a double address register that already supports a 16-bit address, two control registers - CR1 and CR2, status registers - SR1 and SR2, as well as a register specifying the frequency of data transmission or speed.

But since we are going to use the HAL library, we do not have to worry about programming bits of register data and when exactly to enter the same data into the address and data register, and the HAL library promises to take this burden.

Therefore, we only need to know what functions we use and how to use them from this library.

Therefore, we will open the user's guide for the HAL library and find the functions we need, all of which we do not need. They are very, very many. We will analyze only those that we will use in our work. The rest will be disassembled in other lessons, or maybe we will not disassemble at all, but will only use it.

Moreover, it makes no sense to consider the functions of initialization and deinitialization, since this is the job of the Cube MX code generator and, as practice shows, is quite manageable.

And the functions that we need in today's lesson, we'll analyze as they are written in the code.

Now, in fact, the chip DS3231. This chip is a real-time clock developed by **Dallas** , which I believe is the most common among users of such chips.

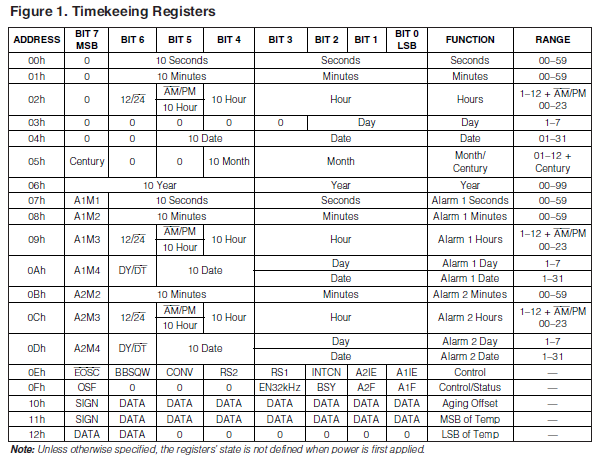
These real-time clock work similar to the predecessor DS1307, only very much improved. The quartz resonator is already inside and therefore we do not solder it, we also have very good thermal compensation, so the accuracy of this watch is just amazing. Although there is still a register for the correction of the stroke, but I never used it, because without it the chip provides an excellent accuracy of the stroke.

This chip is powered with success both from 3 volts and from 5 volts, so there's nothing to be afraid of.

Well, as well as all modern RTCs, this chip has a very well-organized energy independence from external sources. For this, there are two contacts for connecting a 3.7-volt backup lithium battery, which ensures the continuation of the clock's progress when the power is turned off. Low power consumption in power mode from the battery is provided by the fact that almost the entire function except the clock does not work.

The data rate is supported by this chip and 100 kHz and 400 kHz.

Now about the registers of the chip. This is about what exactly this chip can contain in itself



Here are how many, these registers.

**00h** - seconds. The seconds are stored in binary-decimal form. That is, in the lower 4 bits, a few seconds are stored, and in the older three bits, dozens.

**01h** - minutes. They are stored the same way.

**02h** - more universal register. The clock is stored here. The four least significant bits are the units of the chas, the next two are the tens, the next 6 bits are the flag of the one, after noon now is the time or until noon, in 7 bit - the storage mode is 12 hours or 24 hours.

**03h** is the day of the week. It is stored in the lower 3 bits, the remaining bits are not used.

**04h** - the day of the month is stored here, also in binary-decimal format. In the four smallest bits - a unit, in the next two are tens, the next bit is not used.

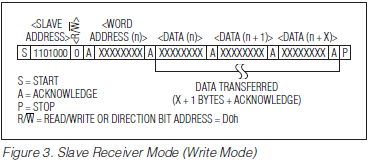
**05h** - the month number in the year - is stored in binary-decimal format exactly as the clock. The last bit will automatically be set when the last day of the century ends and the next century begins. Laughter and only. I think it can store a unit, it will say that we have 21st century in our yard.

**06h** - the number of the year, and not full four-digit, but only two-digit. In the lower four bits - a unit, in the senior - dozens. And what century to mean, you can store in the 7th bit of the month.

We will use these seven registers. Next are the alarm registers, and the last two registers are the temperature measured on the thermal sensor, which is installed in the microcircuit. On idea, it is possible to use the given register to deduce temperature indications though it and temperature of a microcircuit. But thermocompensation is so arranged that if the microcircuit starts to get warm, then the technologies that prevent it will start working and it will cool down. I tried to read this register, and. in principle it was the ambient air temperature.

There are also two registers of management and status. In the framework of this lesson we will not use them, but I'll attach the technical documentation to the microcircuit on the page below and you can download and read the registers bits by downloading it.

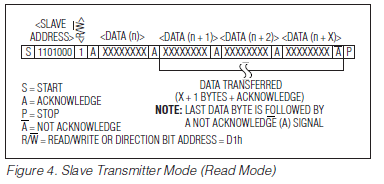
Now, in fact, data transmission:



This is the transfer of data.

First START, then a 7-bit address, then 0, which means that we will write to this chip, then bit the confirmation, then the register address, which we just reviewed above, again the confirmation, then the data itself, . Confirmations are waiting after each transmitted byte, and in the end the condition is STOP. So we can safely set the time, transfer the bytes of all seven registers at once, passing the byte of the very first of them, that is, 0x00, before that.

Now reading



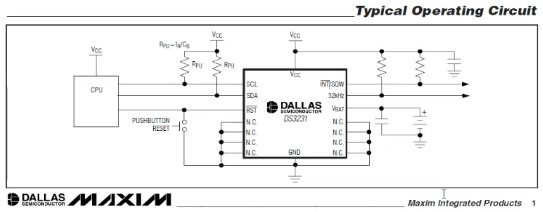
Here we can also read not one register in the package, but also all 7 registers.

In general, this is the documentation for the most leading device, the slave has a little bit wrong.

Before using this diagram, we first do the START, then the device address, then the write bit, then the record, then the register address, then the restart, then the address of the device, then the read bit or 1, then wait for the data with the confirmation, and A line above the letter A after the last received byte means that we do not wait for confirmation or wait for the "No confirmation" condition, that is, the SDA bus will be set to high and at the end of the STOP.

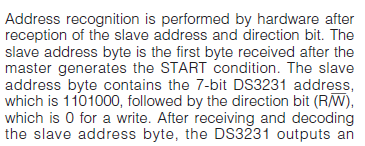
Like this. Of course, the HAL library will deprive us of the pleasure of playing with these algorithms of reception and transmission, as it will do it itself. But, I think, we already played enough with this in the lessons on AVR and with the EEPROM chip and the DS1307 chip.

Here is a typical scheme for connecting this chip to the controller



Here we see that eight knives are connected to the body, there are also SDA and SCL legs, VCC and GND power legs, RST for rebooting the chip, we do not use this foot. There is also a leg for connecting a VBAT battery, an SQW output so that we can take pulses of a certain frequency from this chip for some synchronization, we use this foot as a rule for blinking the colon. Also the frequency on this leg can be controlled by certain bits in a certain register. And also an output of 32 kHz.

Almost forgot about the address, which would have to refer to the device. Without knowing it, we will not be able to communicate with the microchip, since it simply will not answer us

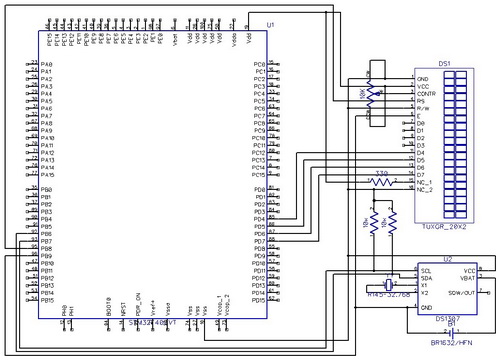


This is an extract from the datasheet, then there is no sense to read.

The address is **0b1101000** . Accordingly, in functions, we will use it in a shifted by 1 bit to the left state, that is, immediately with the trailing zero. in the functions. intended for reading, we will also use bit 0, and the function itself will shift everything as it should, so they are arranged, these functions are HAL. That is, we will have the address **0xD0** .

The clock from this chip will be shown on the display. which we connected in the [**last lesson**](http://narodstream.ru/stm-urok-7-biblioteka-hal-lcd-20x4-4-bitnyj-rezhim-vivodim-informaciju/) .

Here is the connection diagram (click on the image to enlarge the image)

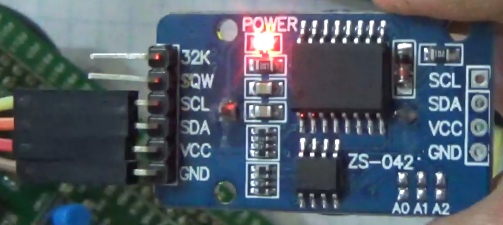
[](http://narodstream.ru/wp-content/uploads/2016/12/Image07.jpg)

Here is a slightly different chip, but the connection is no different. I just did not find such a microcircuit in the program-editor of electronic circuits.

Also I will say that I do not just have a chip, but a ready-made module, and immediately with a battery and all the contacts that are output, a link to the seller is under the video version of the lesson in the description.

The module looks like this



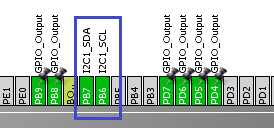


Just as before, the project is created from the previous one. A new project called MYDS3231

Run it in the Cube. We turn on the I2C1 bus in this way

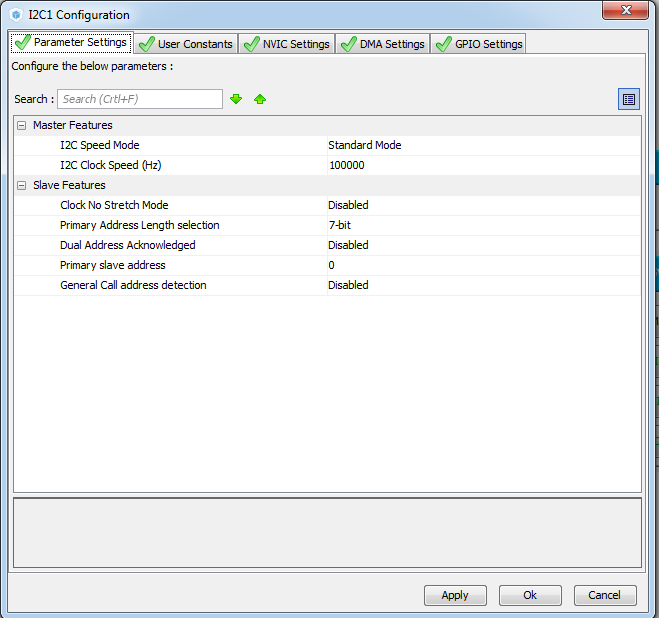
image12

We will also see the virtual controller on the right. that we had the involved legs of the ports

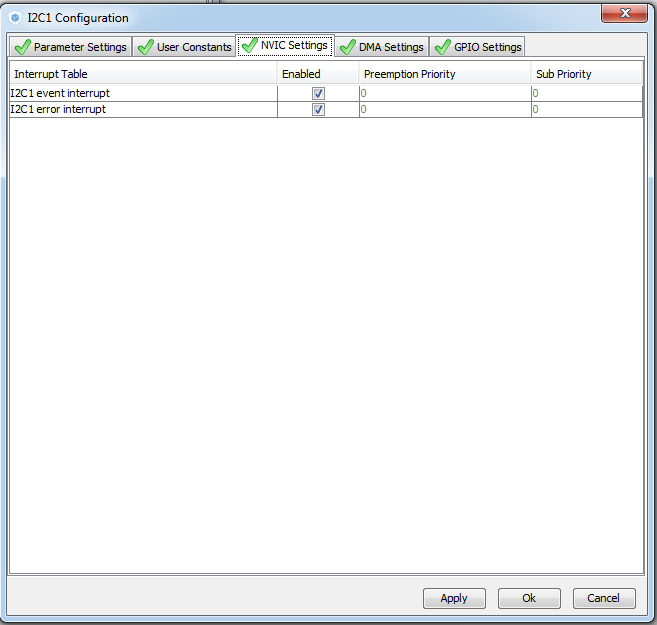


Here we will connect the corresponding legs of the module to these legs PB6 and PB7.

Settings in Configuration will be

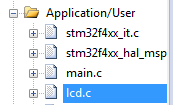


For the future include and interruptions



Generate the project and go to the mailbox.

Connect the lcd.c file to the project tree



Create the files i2c.c and i2c.h

Let's try to assemble the project.

In i2c.h we connect libraries

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

**#include "lcd.h"**

We connect the file in both i2c.c and main.h

**#include "i2c.h"**

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

Also we create and connect files RTC.h and RTC.c

In RTC.h we connect the library

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

After that, the contents of the main.h file

**#ifndef MAIN\_H\_**

**#define MAIN\_H\_**

**#include "lcd.h"**

**#include "i2c.h"**

**#include "RTC.h"**

**#endif / \* MAIN\_H\_ \* /**