**Lesson 88**

**Part 1**

**SD. SPI. FATFS**

We continue to work with the folk debug board **STM32F103RCT6** , with which we have been dealing for a long time and which. judging by the results of our studies, showed herself with a very good side. This board and the controller installed on it, though not all the delights of progress in themselves are, but very many, that at such a price is very good.

Also I think that now we have come to the point that no memory on board this controller may soon be insufficient to store our constantly growing volumes of data, so it's time to connect something that will allow us to store files of considerable volume. I initially leaned towards USB, which is also present on board this board, but, having tested it tightly, I realized that it was not quite working confidently.

Then the final my choice fell on the **SD card**. Only one problem is that the SDIO interface, which is so fond of these cards, does not exist on our controller. But it still did not stop me and I remembered that SD cards also work well using the SPI interface, although some functionality with this bus becomes unavailable, and the data transfer rate is much lower in this case, but it also did not stop me. And in many forums, I saw that many experienced problems connecting SPI data cards to STM controllers, especially with the use of library functions HAL, began to invent some software jerking with their feet, which naturally introduces incorrect work of the whole program (the hardware is always better) . And all these rumors gave me even more push to dispel all these myths and so I still decided on this lesson.

We will also face one more challenge. Connect the FatFS library so that we can work with the file system, not with the blocks and sectors.

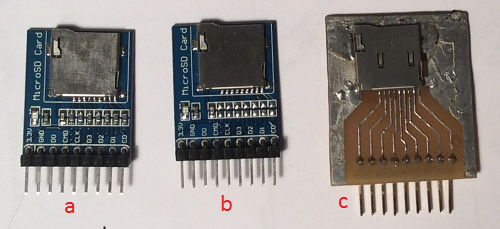
Well, let's begin!

In [**lesson 44,**](http://narodstream.ru/stm-urok-44-sdio-fatfs/) we already connected such a card via the SDIO interface to another controller, but nevertheless it will be useful to look at it, since we also used the FatFS library there. But it's later. While the library is far away. Now the main thing is to learn how to work with the card on the SPI interface, and for this it will be equally useful to look [**at the AVR lesson 33**](http://narodstream.ru/avr-urok-33-spi-karta-sd-fat-chast-1/) , in which we also use the SPI as the bus for connecting the SD card. It's just that we coordinated all our actions with the datasheet and would not like to repeat the same thing over and over again. Nevertheless, this does not relieve us from reading the documentation, so that later on we will not face the same rakes that the authors of questions had to face on the forums. All documentation is available on the official website<https://www.sdcard.org/downloads/pls/> .

Let's start with the scheme. Just like in the previous lessons on connecting the LAN module, we will feed our controller from a stabilized 3.3-volt DC-DC converter, we will program the cheapest with the help of the ST-Link programmer, and we will not supply our board with it We will. Module with a Micro-SD card we will also feed from the converter.

Modules with a card reader Micro-SD I have the whole 3. One of them is self-made, and two I ordered in the online store.

Here are my modules (click on the image to enlarge the image):

[](http://narodstream.ru/wp-content/uploads/2017/08/index00.jpg)

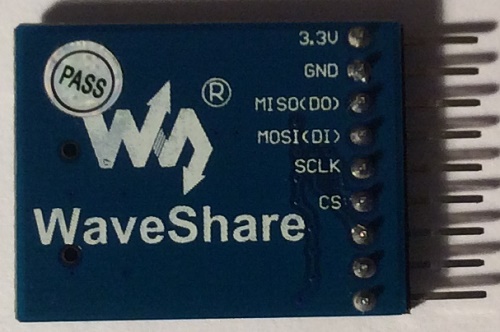
a) This is a serial module with aliexpress as it came;

b) it's the same module, only here I undressed (blew off) all the pull-up resistors.

c) homemade module, which was used in lesson 44.

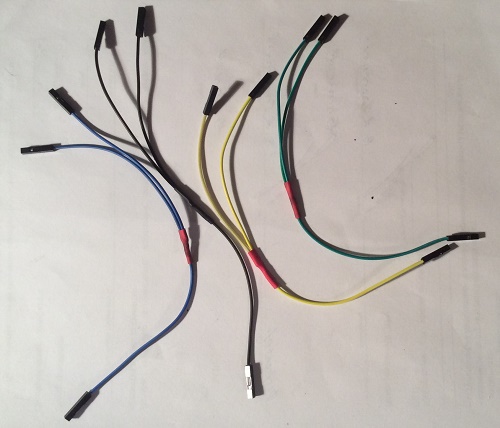
I think that we will stop on option b, in which there are no pull-up resistors, without them everything works fine at a speed of 10 megabits per second, I tried.

Also the sent module is not just beautiful, but it has the signatures of each leg, which it performs a role. Although in the figure we see all this for SDIO mode, but let's try to flip the module



And we will see that there are already inscriptions for the SPI mode, which is undoubtedly very convenient.

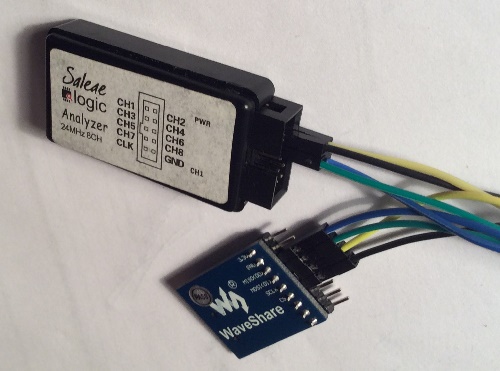
I must say that before I give this lesson, I tinkered with the response of the map, I studied everything with a logical analyzer and I want to share this process with you. Therefore, we will connect this module to the controller with non-standard wires like Mama-Mama, but three-ended ones, which I soldered myself, and the spot of adhesion was designed for heat shrinkage



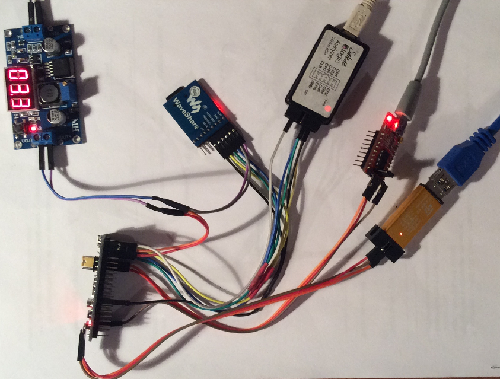
But the logical analyzer that I have in stock. The simplest, probably the cheapest, but it helps me a lot



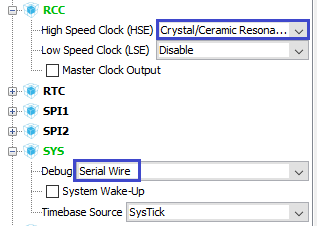
Connect the wires first to the MicroSD card module and to the logic analyzer



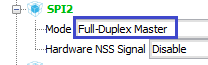
Then we bring the common wire to the analyzer from the controller, and also bring power to the card module, connect the power, connect the logic analyzer to the PC with the Mini-USB wire, and connect all the wires of the SPI bus to the debug board. We will connect to the SPI2 bus, because SPI1 will be used by the LAN module later. Perhaps later we will connect everything from one bus and I think all this will work, and we have the same speed, and we have three-pointed wires already, not all the time we have to work with the connected logic analyzer. But for now it will be SPI2. At the same time, we will also check it for operability. And also connect a USART-USB adapter so that we can track some information. Once we did this using a display, but USART, as practice confirmed, was more convenient. And more information, and connect easier. As a result, you will get here such a scheme (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/08/index05.jpg)

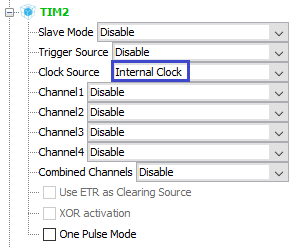
Now, I think, it's time to move on to the project. It is most expedient to create a new project. Let's run the Cube MX, select our controller, connect the ceramic resonator first, and turn on the Serial Programming programming bus



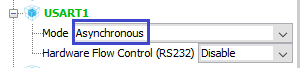
Enable SPI2



Turn on the 2nd timer



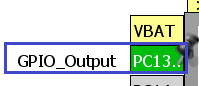
Also include USART1



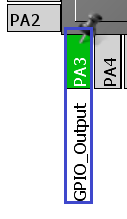
Enable FatFS

index19

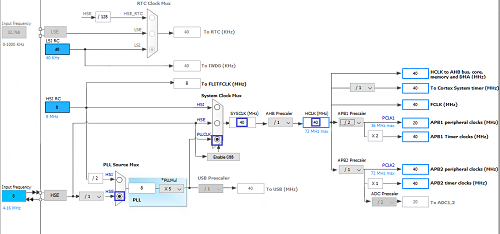
**Turn on the** leg **PC13** , responsible for the LED to the output



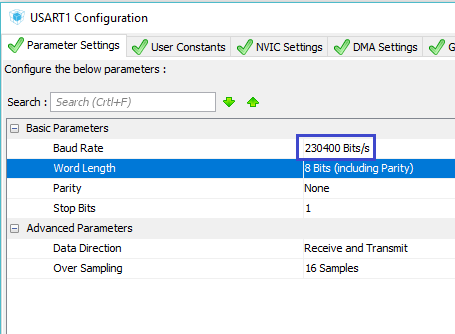
Still on the output, **turn** on the leg **PA3** , which will act as the **Chip Select** leg for the **SPI** bus



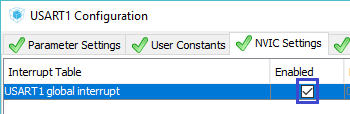
Go to the **Clock Configuration** section of the corresponding tab and configure the following parameters (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/08/index13.png)

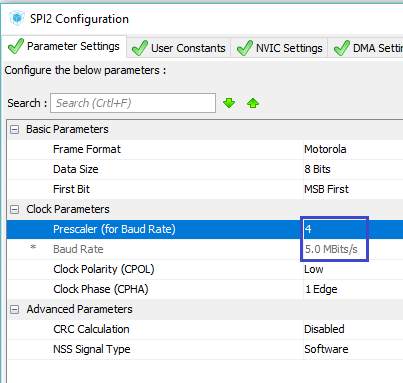
Go to the next tab in the **Configuration** section . First set up **USART**



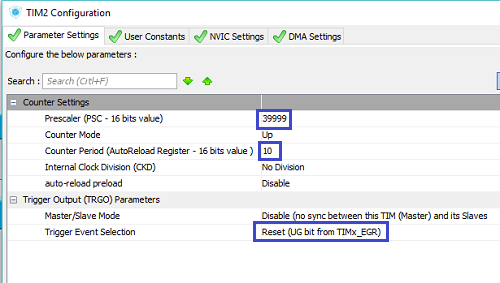
Enable USART interrupts



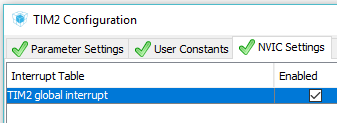
Save the USART settings, now configure the SPI



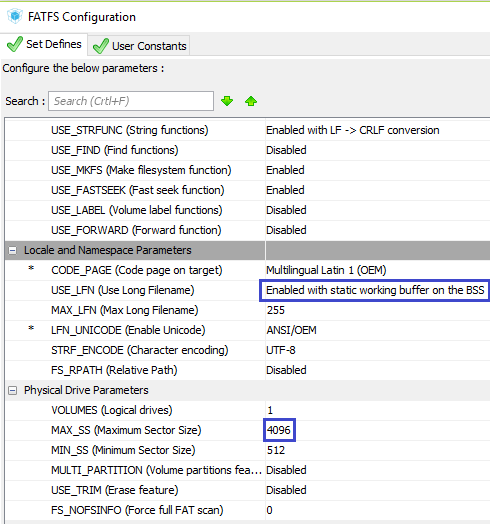
Also save the settings. Now the timer, which we set to the frequency of 100 kilohertz, to interrupt occur every 10 milliseconds



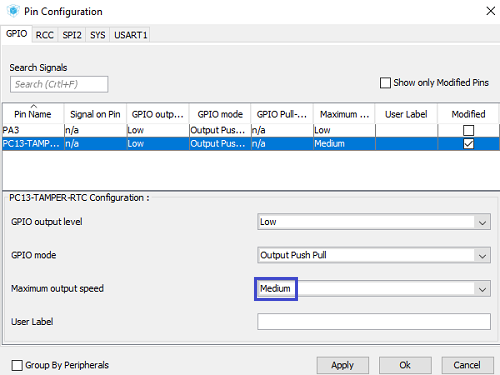
Interrupts also include



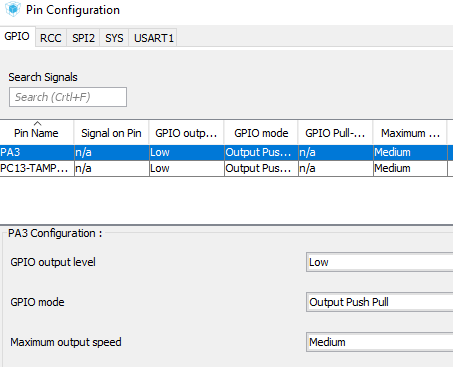
Also configure FATFS, including support for a large sector and long file names. The code page will be chosen by each person. Just remember that not all code pages contain files with arrays in the supplied FATFS library. Then either use existing pages, or search for files, or write the arrays themselves. We'll leave the default code page



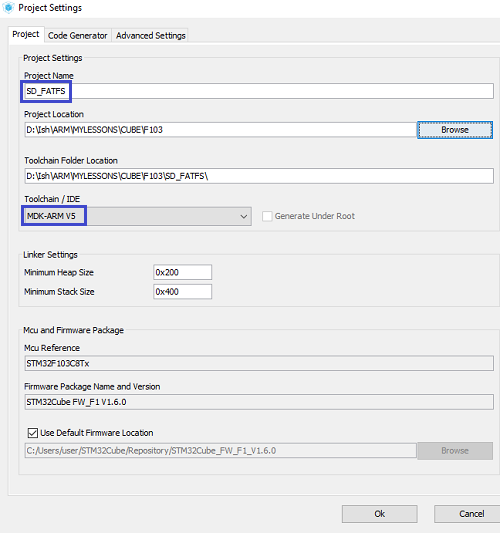
Now adjust the feet of the **GPIO** . Let's start with the leg of **PC13** . Let's increase the speed a bit



The same will be done with the PA3 foot



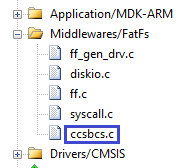
Now we'll go into the settings, come up with a name for the project, and also choose the desired programming environment



Save the settings, generate the project, open it in Keil, configure the programmer for autoregistration and try to assemble the project.

We will have two mistakes, see for yourself what. Errors will be because we included support for long names and the Cube project generator was supposed to connect a file with the functions that the linker curses on, but for some reason it did not. We will help him! I think he will "not forget" our kindness.

We connect the file **ccsbcs.c** from the folder " **Our project / Middlewares \ Third\_Party \ FatFs \ src \ option** " to the section of the project tree " **Middlewares / FatFs** "



Run the timer in **main ()**

/\* USER CODE BEGIN 2 \*/

**HAL\_TIM\_Base\_Start\_IT(&htim2);**

In the main.c file, create a global variable for the counting timer

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

**volatile uint16\_t Timer1=0;**

Also we will create an interrupt handler, in which we will constantly increment this variable

/\* USER CODE BEGIN 4 \*/

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

**void HAL\_TIM\_PeriodElapsedCallback(TIM\_HandleTypeDef \*htim)**

**{**

**if(htim==&htim2)**

**{**

**Timer1++;**

**}**

**}**

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

/\* USER CODE END 4 \*/

Create two files - **sd.c** and **sd.h** to work with the SD card next

**sd.h**

**#ifndef SD\_H\_**

**#define SD\_H\_**

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

**#include "stm32f1xx\_hal.h"**

**#include <string.h>**

**#include <stdlib.h>**

**#include <stdint.h>**

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

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

**#endif /\* SD\_H\_ \*/**

**sd.c**

**#include "sd.h"**

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

Connect our library in **main.c**

/\* USER CODE BEGIN Includes \*/

**#include "sd.h"**

/\* USER CODE END Includes \*/

Go to the file sd.c and connect in it our variable timer

#include "sd.h"

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

**extern volatile uint16\_t Timer1;**

We'll create an inclusion function, in which we just wait 20 milliseconds to set the voltage

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

**void SD\_PowerOn(void)**

**{**

**Timer1 = 0;**

**while(Timer1<2) //ждём 20 милисекунд, для того, чтобы напряжение стабилизировалось**

**;**

**}**

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

Also below, we will create in it an SD card initialization function

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

**uint8\_t sd\_ini(void)**

**{**

**return 0;**

**}**

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

Let's create prototypes for these functions in the header file and call them in **main.c** in the main function

HAL\_TIM\_Base\_Start\_IT(&htim2);

**SD\_PowerOn();**

**sd\_ini();**

Add macros for the LED, as well as for the SPI bus in the file **sd.h**

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

**#define CS\_SD\_GPIO\_PORT GPIOA**

**#define CS\_SD\_PIN GPIO\_PIN\_3**

**#define SS\_SD\_SELECT() HAL\_GPIO\_WritePin(CS\_SD\_GPIO\_PORT, CS\_SD\_PIN, GPIO\_PIN\_RESET)**

**#define SS\_SD\_DESELECT() HAL\_GPIO\_WritePin(CS\_SD\_GPIO\_PORT, CS\_SD\_PIN, GPIO\_PIN\_SET)**

**#define LD\_ON HAL\_GPIO\_WritePin(GPIOC, GPIO\_PIN\_13, GPIO\_PIN\_RESET); //RED**

**#define LD\_OFF HAL\_GPIO\_WritePin(GPIOC, GPIO\_PIN\_13, GPIO\_PIN\_SET); //RED**

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

Let's go to the sd.c file and connect the USART and SPI

#include "sd.h"

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

**extern SPI\_HandleTypeDef hspi2;**

**extern UART\_HandleTypeDef huart1;**

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

 Add a function to turn on the red LED in case of any error at the top

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

**static void Error (void)**

**{**

**LD\_ON;**

**}**

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

Below this function, add a function for writing and reading the **SPI** bus

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

**uint8\_t SPIx\_WriteRead(uint8\_t Byte)**

**{**

**uint8\_t receivedbyte = 0;**

**if(HAL\_SPI\_TransmitReceive(&hspi2,(uint8\_t\*) &Byte,(uint8\_t\*) &receivedbyte,1,0x1000)!=*HAL\_OK*)**

**{**

**Error();**

**}**

**return receivedbyte;**

**}**

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

Also add below this function three functions for SPI - read, write and normal byte byte

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

**void SPI\_SendByte(uint8\_t bt)**

**{**

**SPIx\_WriteRead(bt);**

**}**

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

**uint8\_t SPI\_ReceiveByte(void)**

**{**

**uint8\_t bt = SPIx\_WriteRead(0xFF);**

**return bt;**

**}**

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

**void SPI\_Release(void)**

**{**

**SPIx\_WriteRead(0xFF);**

**}**

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

The last **SPI\_Release** function **will** need a prototype, so add it in the **sd.h** file . Also in this file we add another structure for storing the properties of our map. From the properties will be so far only the type of card. At the same time, above this structure, we will add map type macros

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

**/\* Card type flags (CardType) \*/**

**#define CT\_MMC 0x01 /\* MMC ver 3 \*/**

**#define CT\_SD1 0x02 /\* SD ver 1 \*/**

**#define CT\_SD2 0x04 /\* SD ver 2 \*/**

**#define CT\_SDC (CT\_SD1|CT\_SD2) /\* SD \*/**

**#define CT\_BLOCK 0x08 /\* Block addressing \*/**

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

**typedef struct sd\_info {**

**volatile uint8\_t type;//тип карты**

**} sd\_info\_ptr;**

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

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

extern volatile uint16\_t Timer1;

**sd\_info\_ptr sdinfo;**

Let's go to the initialization function. The initialization task is to determine the type of our card, which, as we know, happens to be several, and also to switch the card to normal mode from the **IDLE** mode .

Add here the variables for the counter, for the virtual timer, as well as another variable for temporarily storing the values, turn off the LED and initialize the type of card

uint8\_t sd\_ini(void)

{

**uint8\_t i;**

**int16\_t tmr;**

**uint32\_t temp;**

**LD\_OFF;**

**sdinfo.type = 0;**

Let's save the value of the SPI bus divider, set another divider and re-initialize the bus to reduce the speed, because the map can not understand too quickly

sdinfo.type = 0;

**temp = hspi2.Init.BaudRatePrescaler;**

**hspi2.Init.BaudRatePrescaler = SPI\_BAUDRATEPRESCALER\_128; //156.25 kbbs**

**HAL\_SPI\_Init(&hspi2);**

Then we will transmit several pulses to the SPI synchronization leg. We need at least 74 data-dots, we'll give 80 for just in case. This can be done by the usual byte-by-pass on the SPI bus. And there is still a requirement - the choice foot should be raised

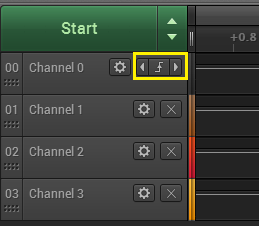
HAL\_SPI\_Init(&hspi2);

**SS\_SD\_DESELECT();**

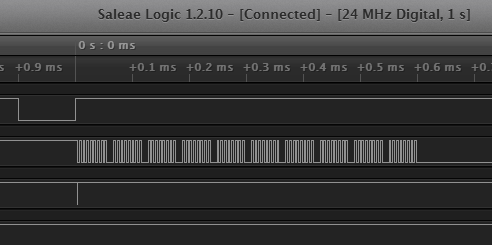
**for(i=0;i<10;i++) //80 импульсов (не менее 74) Даташит стр 91**

**SPI\_Release();**

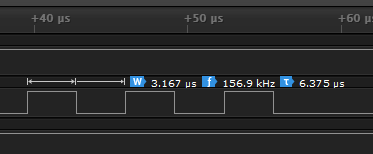
And now we will collect the code, we will sew the controller and see the result in the program of the logical analyzer. To do this, we will set up Saleae Logic in channel 0, which is responsible for the CS leg, a trigger triggered on the rising edge, so that the recording begins with the first appearance of such a pulse on this leg.



Press the Start button and reboot the controller. We obtain the following result



We see 10 packets of 8 pulses. CS in our high state, pulses only on the bus synchronization. A pulse on the MOSI leg indicates the inclusion of a high level for the transmission of 0xFF. But he worries us least of all. We will stretch a little the plot of the mouse wheel



We will see the frequency and period of pulses. The frequency at us is really very close to the declared and makes 157 kilohertz.

Excellent! In theory, our flash card should already have realized that they want it to work in SPI mode.

In the [**next part of**](http://narodstream.ru/stm-urok-88-sd-spi-fatfs-chast-2/) this lesson, we will write the full function of initializing the SD card, constantly checking each step in practice.

**Lesson 88**

**Part 2**

# SD. SPI. FATFS

In the [**previous part of**](http://narodstream.ru/stm-urok-88-sd-spi-fatfs-chast-1/) our lesson we connected an SD card, created a number of service functions for its work and started writing the initialization function.

We return the frequency to the one declared in the Cube. Also, we lower the leg of CS and try to pass on the SPI bus a pair of some quantities

  SPI\_Release();

**hspi2.Init.BaudRatePrescaler = temp;**

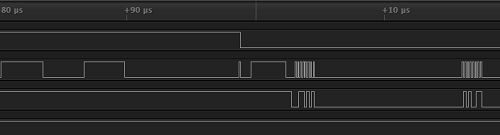
**HAL\_SPI\_Init(&hspi2);**

**SS\_SD\_SELECT();**

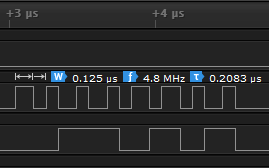
**SPI\_SendByte(0x35);**

**SPI\_SendByte(0x53);**

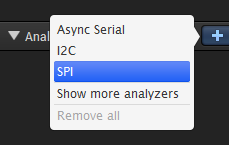
We will collect the code, we will sew the controller and see the result in the analyzer window (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/08/image27.png)

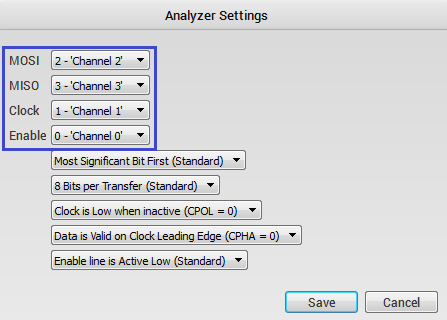
We see that the pulses on the synchronization leg have significantly increased their frequency. stretch the chart and see



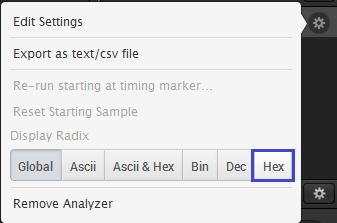
We see that the frequency became 4.8 megahertz, also close to the claimed (5 megahertz). At such large frequencies, this is the normal accuracy for this class of analyzer. Now let's enable analysis in SPI mode in the analysis program



Set up the channels, the rest is not touched, because in theory we have SPI mode 0. This is the only mode in which the SD card works



Let's adjust the display of the results in hexadecimal form



Only when the SPI frequency is changed, the analyzer is confused in the measurements, so we comment on the frequency switching in the code. The card will work without it, that is, at a high frequency.

sdinfo.type = 0;

//temp = hspi2.Init.BaudRatePrescaler;

//hspi2.Init.BaudRatePrescaler = SPI\_BAUDRATEPRESCALER\_128; //156.25 kbbs

//HAL\_SPI\_Init(&hspi2);

SS\_SD\_DESELECT();

for(i=0;i<10;i++) //80 импульсов (не менее 74) Даташит стр 91

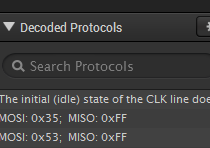
SPI\_Release();

//hspi2.Init.BaudRatePrescaler = SPI\_BAUDRATEPRESCALER\_16;

//HAL\_SPI\_Init(&hspi2);

SS\_SD\_SELECT();

Again, we'll compile the code and run the controller. As a result, the analyzer will now display everything correctly



Also look at the chart, which now also shows everything (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/08/image34.png)

Delete the test code

~~SPI\_SendByte(0x35);~~

~~SPI\_SendByte(0x53);~~

Now try to send a command to the map.

For this we need a special function, which we add above the inclusion and initialization functions, so as not to make a prototype

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

**static uint8\_t SD\_cmd (uint8\_t cmd, uint32\_t arg)**

**{**

**uint8\_t n, res;**

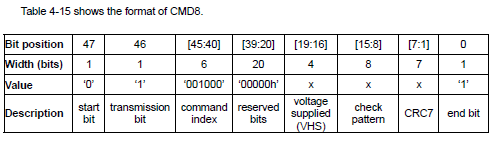
**return res;**

**}**

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

Briefly describe the process of transferring the team to the card.

Let's look at the example in datash



Each team has an index. This command has an index of 8, since it is called **CMD8** . There are also differences in the types of commands. But about this later, we are still interested in this type, and this command will later have to be transferred to us.

We see that the command transmission consists of 48 bits, that is, from 6 bytes. The first line shows the position of the bit in the command, the second - the value of the parameter in bits, the third - the value, and the fourth - the explanation of the parameter.

The start bit is always 0.

The transmit bit is 1.

The command index - in the case of this command is 8.

then go reserved bits - as many as 20 pieces, all equal to zero.

Then there are the parameters, the 7 bits of the checksum and the stop bit - always 1.

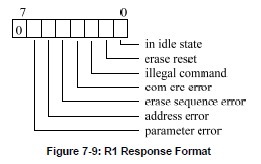
Also, there are regular (CMD) and extended (ACMD) commands that consist of two commands at once, the first of which is always CMD55.

After the transfer of the team, the result usually follows.

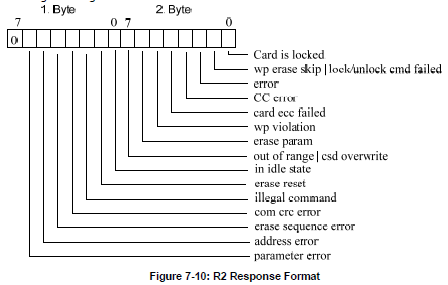
And the commands that we transmit to the card can not only manage something with them, they can also return something.

Moreover, the returned result, moreover, is of different types (I mentioned above about command types, it's not the same) - it starts with type R1, etc.

Here, for example, type R1



But R2



First we'll add the commands macros at the top of the file

extern UART\_HandleTypeDef huart1;

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

**// Definitions for MMC/SDC command**

**#define CMD0 (0x40+0) // GO\_IDLE\_STATE**

**#define CMD1 (0x40+1) // SEND\_OP\_COND (MMC)**

**#define ACMD41 (0xC0+41) // SEND\_OP\_COND (SDC)**

**#define CMD8 (0x40+8) // SEND\_IF\_COND**

**#define CMD9 (0x40+9) // SEND\_CSD**

**#define CMD16 (0x40+16) // SET\_BLOCKLEN**

**#define CMD17 (0x40+17) // READ\_SINGLE\_BLOCK**

**#define CMD24 (0x40+24) // WRITE\_BLOCK**

**#define CMD55 (0x40+55) // APP\_CMD**

**#define CMD58 (0x40+58) // READ\_OCR**

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

We continue our function of passing the command, the second argument of which is our arguments, combined into one 32-bit value, and write the code that will determine what kind of team we have. The ACMD command is distinguished from the CMD by the included seventh bit

uint8\_t n, res;

**// ACMD<n> is the command sequense of CMD55-CMD<n>**

**if (cmd & 0x80)**

**{**

**cmd &= 0x7F;**

**res = SD\_cmd(CMD55, 0);**

**if (res > 1) return res;**

**}**

Further we make a choice of a card here such here in an interesting way

  if (res > 1) return res;

}

**// Select the card**

**SS\_SD\_DESELECT();**

**SPI\_ReceiveByte();**

**SS\_SD\_SELECT();**

**SPI\_ReceiveByte();**

First we lift the foot of the CS, then drive the byte 0xFF on the bus, then lower the leg and run another one of the same bytes.

Next we already send the command directly

SPI\_ReceiveByte();

**// Send a command packet**

**SPI\_SendByte(cmd); // Start + Command index**

**SPI\_SendByte((uint8\_t)(arg >> 24)); // Argument[31..24]**

**SPI\_SendByte((uint8\_t)(arg >> 16)); // Argument[23..16]**

**SPI\_SendByte((uint8\_t)(arg >> 8)); // Argument[15..8]**

**SPI\_SendByte((uint8\_t)arg); // Argument[7..0]**

That is, here we consistently transmit to the bus the arguments of the command, first passing its index.

Next, we enter the checksum together with the stop bit into the variable, depending on the command. That is, for these two teams the amount has already been calculated, for others, zero, it's understandable with the stop bit (0x01)

SPI\_SendByte((uint8\_t)arg); // Argument[7..0]

**n = 0x01; // Dummy CRC + Stop**

**if (cmd == CMD0) {n = 0x95;} // Valid CRC for CMD0(0)**

**if (cmd == CMD8) {n = 0x87;} // Valid CRC for CMD8(0x1AA)**

**SPI\_SendByte(n);**

While we are only interested in the first type. If other types are needed, then the first byte will still be waiting for us, and with the rest we'll figure it out in the code after calling the command

Add a conditional loop to our function

  SPI\_SendByte(n);

**// Receive a command response**

**n = 10; // Wait for a valid response in timeout of 10 attempts**

**do {**

**res = SPI\_ReceiveByte();**

**} while ((res & 0x80) && --n);**

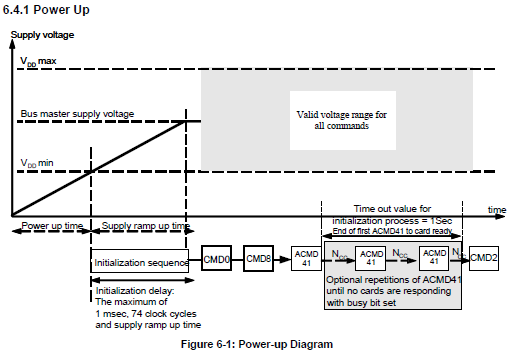
  return res;

}

That is, here we are waiting for the response from the map.

Now back to the initialization function.

Let's look in the technical documentation for the initialization process, while the simplified



The inclusion timeout we added, we have already done the 74 cycles, now we need to pass the CMD0 command, which does a software reset, by inserting the card into IDLE mode, in which we can read certain parameters before deciding how to behave further with the map. As arguments, this command requires all zeros

  SS\_SD\_SELECT();

**if (SD\_cmd(CMD0, 0) == 1) // Enter Idle state**

**{**

**}**

**else**

**{**

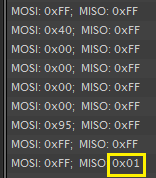
**return 1;**

**}**

  return 0;

}

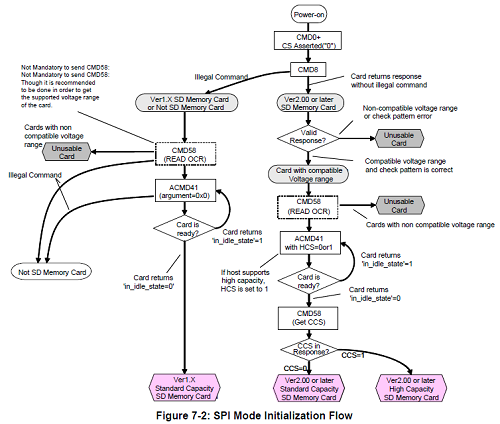
We will collect the code, we will sew the controller and check the analyzer that we have received the desired answer from the card



And we waited for the answer pretty quickly, skipping only one cycle.

Well, great! The map responds, we will initialize it further by examining its type.

Detailed initialization with a description of what and for which commands we are calling is also illustrated in the technical documentation (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/08/index42.png)

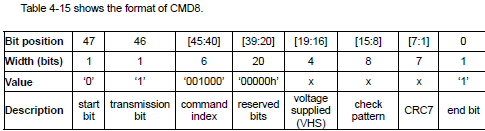
Let's add a global string array in the file sd.c

sd\_info\_ptr sdinfo;

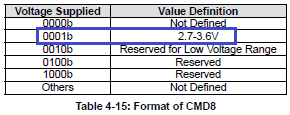
**char str1[60]={0};**

Let's return to our initialization function and send the command **CMD8** to our map in order to find out the version of our map. We call the command in the body of a positive return from the function of calling the CMD0 command. The arguments are the **OCR** register , which we enter into data. We put 0,0,1,0xAA, such requirements are for version 2 of the SD card.

Here is the description of the **CMD8 command**



That is, a byte with a value of 1 is the value of the VHS register



And there is a footnote of the following content on the same page

index51

A '10101010' is nothing else than 0xAA. That's where the output data for the team comes from.

Before calling the command, run the byte through the bus

if (SD\_cmd(CMD0, 0) == 1) // Enter Idle state

{

**SPI\_Release();**

**if (SD\_cmd(CMD8, 0x1AA) == 1) // SDv2**

**{**

**}**

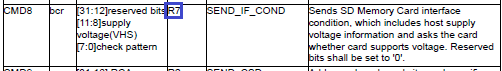
**else //SDv1 or MMCv3**

**{**

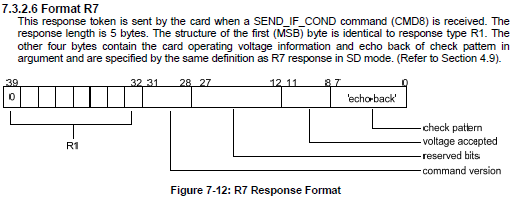
**}**

}

It should be noted that the **CMD8** command returns the result of the **R7** format



I will also describe the format of **R7**



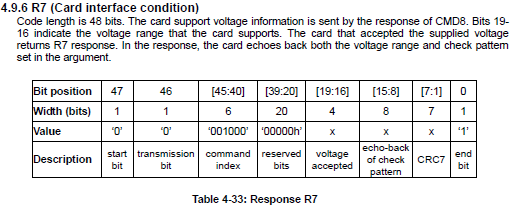
As you can see, the returned result consists of 5 bytes, the oldest one (the one that comes first) is the result of the R1 format. We will receive it immediately in the return from the function. And the remaining 4 bytes we get separately. Therefore, prepare for them a local array in our initialization function

uint8\_t i;

**uint8\_t ocr[4];**

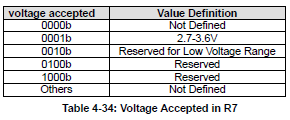
We get the data bytes and display them in the terminal program, if, of course, the returned by the card result is the bytes required by us, that is, the map will be exactly version 2.

There are more descriptions of the result of the R7 format



That is, if everything is normal, then 8 bytes [15: 8] will be the same as they came in the command parameter, so they are listed in the description as "echo-back of check pattern".

And the bytes [19: 6] are the supported voltages



The table exactly in the point is the same as in the input to the command, so it is returned when there is a map of exactly this type, which we want the same thing as we sent it

if (SD\_cmd(CMD8, 0x1AA) == 1) // SDv2

{

**for (i = 0; i < 4; i++) ocr[i] = SPI\_ReceiveByte();**

**sprintf(str1,"OCR: 0x%02X 0x%02X 0x%02X 0x%02Xrn",ocr[0],ocr[1],ocr[2],ocr[3]);**

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

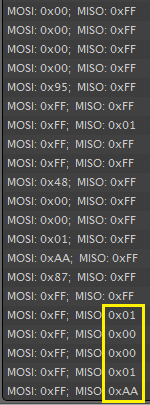
}

We called the variable ocr, and we also use OCR in the string, although this is not any OCR. We will get acquainted with the OCR register when we consider the result of another format later.

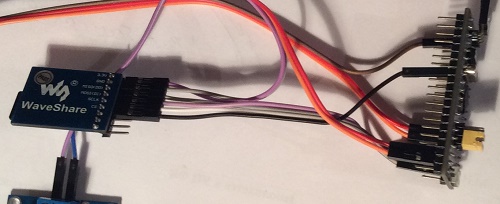
We will collect the code, we will sew the controller and see the output of the result in the terminal program

index45

Also, we can see the whole result of R7 format in the program of the logical analyzer



In principle, now we can disconnect the logical analyzer and attach the microSD card module with ordinary wires Mom-Mom



Also uncomment the code that we commented on for correct analysis of the bus

**temp = hspi2.Init.BaudRatePrescaler;**

**hspi2.Init.BaudRatePrescaler = SPI\_BAUDRATEPRESCALER\_128; //156.25 kbbs**

**HAL\_SPI\_Init(&hspi2);**

SS\_SD\_DESELECT();

for(i=0;i<10;i++) //80 импульсов (не менее 74) Даташит стр 91

SPI\_Release();

**hspi2.Init.BaudRatePrescaler = temp;**

**HAL\_SPI\_Init(&hspi2);**

Add one more variable for the command in our function

uint8\_t sd\_ini(void)

{

  uint8\_t i**, cmd**;

Now consider the opposite case. That case, if the card turned out to be not of the same type, that is, the team did not even return a result of type R1 to the unit.

This means that it has either type 1 or even MMC. Check this out

if (SD\_cmd(CMD8, 0x1AA) == 1) // SDv2

{

  for (i = 0; i < 4; i++) ocr[i] = SPI\_ReceiveByte();

  sprintf(str1,"OCR: 0x%02X 0x%02X 0x%02X 0x%02Xrn",ocr[0],ocr[1],ocr[2],ocr[3]);

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

}

**else //SDv1 or MMCv3**

**{**

**if (SD\_cmd(ACMD41, 0) <= 1)**

**{**

**sdinfo.type = CT\_SD1; cmd = ACMD41; // SDv1**

**}**

**else**

**{**

**sdinfo.type = CT\_MMC; cmd = CMD1; // MMCv3**

**}**

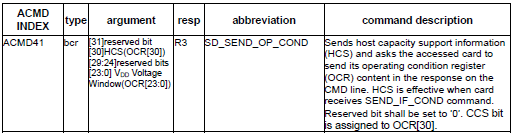
**for (tmr = 25000; tmr && SD\_cmd(cmd, 0); tmr--) ; // Wait for leaving idle state**

**if (!tmr || SD\_cmd(CMD16, 512) != 0) // Set R/W block length to 512**

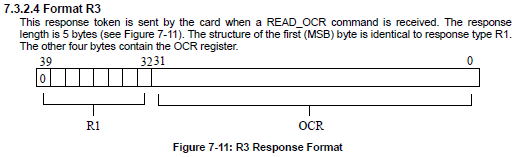
**sdinfo.type = 0;**

**}**

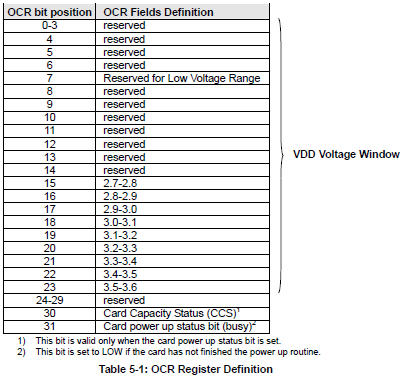
Here we already use the ACMD41 command. Here is its brief description



In the description, we see that the command returns a result of type R3



This is where the **OCR** register is used . Here its contents



We do not need all the bits, we will deal only with some. And in this body of the cycle, we only need a result of type R1, in which we define the type of the map. Next, there will be a loop and a condition that works on the type of MMC card. There we are already using the CMD16 command. Frankly, this piece of code was taken from the example of working with the FatFS library and I did not even go into the meaning of this command because it is unlikely that now cards of this type are generally used. If you want, you can study this command yourself.

Now we continue the body of the condition with a positive result. We displayed the data in the terminal program, now we learn that 0 and 0xAA returned to us as a result of the command CMD8

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

**// Get trailing return value of R7 resp**

**if (ocr[2] == 0x01 && ocr[3] == 0xAA) // The card can work at vdd range of 2.7-3.6V**

**{**

**}**

Further in the body of this condition, we call the ACMD41 command with the HCS bit enabled

if (ocr[2] == 0x01 && ocr[3] == 0xAA) // The card can work at vdd range of 2.7-3.6V

{

**for (tmr = 12000; tmr && SD\_cmd(ACMD41, 1UL << 30); tmr--)**

**; // Wait for leaving idle state (ACMD41 with HCS bit)**

}

Then we call the CMD58 command to read the OCR register, in which we know the state of the **CCS** bit , and if it is set, it allows us to assume that the card is of the SDHC type, that is, the increased capacity, and if the state of the bit is zero, then SDSC capacity). Input arguments are zeros

  ; // Wait for leaving idle state (ACMD41 with HCS bit)

**if (tmr && SD\_cmd(CMD58, 0) == 0) { // Check CCS bit in the OCR**

**for (i = 0; i < 4; i++) ocr[i] = SPI\_ReceiveByte();**

We display our OCR (now, indeed, OCR) in the terminal program and put the map type in the corresponding data field of the structure

    for (i = 0; i < 4; i++) ocr[i] = SPI\_ReceiveByte();

**sprintf(str1,"OCR: 0x%02X 0x%02X 0x%02X 0x%02Xrn",ocr[0],ocr[1],ocr[2],ocr[3]);**

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

**sdinfo.type = (ocr[0] & 0x40) ? CT\_SD2 | CT\_BLOCK : CT\_SD2; // SDv2 (HC or SC)**

**}**

  }

}

else //SDv1 or MMCv3

Now at the end of the function, we will display the type of our card in hexadecimal form

    return 1;

  }

**sprintf(str1,"Type SD: 0x%02Xrn",sdinfo.type);**

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

return 0;

}

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

index57

At this point, the initialization of the map will be considered finished.

In the [**next part of**](http://narodstream.ru/stm-urok-88-sd-spi-fatfs-chast-3/) this lesson we will add the necessary code to the user functions of the FATFS library and try to read the text file from the SD card.

**Lesson 88**

**Part 3**

# SD. SPI. FATFS

In the [**previous part of**](http://narodstream.ru/stm-urok-88-sd-spi-fatfs-chast-2/) this lesson, we have written completely the function of initializing the SD card, constantly checking each step in practice.

Now our task is to read some data from the map. Until then, we'll try to read the block, since we can not simply read the files without the ability to read the elementary block.

We add a function in the **sd.c** file just above the function **sd\_ini** , in the input parameters of which there will be a pointer to the buffer into which the block data and block address will be read

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

**uint8\_t SD\_Read\_Block (uint8\_t \*buff, uint32\_t lba)**

**{**

**uint8\_t result;**

**uint16\_t cnt;**

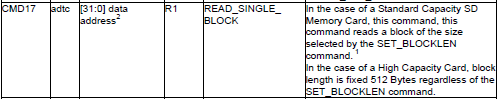
**return 0;**

**}**

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

In order to read one block, there is a command **CMD17** , the argument of which will be the 32-bit address of the block. Moreover, if the card has an increased capacity (SDHC or SDXC), then the address of the block is considered in blocks of 512 or more bytes, depending on the block size. In other cases, the address is the byte number from the beginning of the address space of the card's memory. Addresses are considered with 0.

Here is the command description



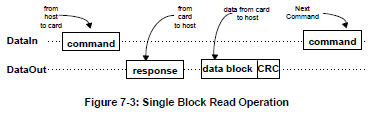
Call this command in our block readout function

uint16\_t cnt;

**result=SD\_cmd (CMD17, lba); //CMD17 даташит стр 50 и 96**

**if (result!=0x00) return 5; //Выйти, если результат не 0x00**

Let's see how the data is read from the card



Therefore, skip the byte and read the data into the buffer

  if (result!=0x00) return 5; //Выйти, если результат не 0x00

**SPI\_Release();**

**cnt=0;**

**do{ //Ждем начала блока**

**result=SPI\_ReceiveByte();**

**cnt++;**

**} while ( (result!=0xFE)&&(cnt<0xFFFF) );**

**if (cnt>=0xFFFF) return 5;**

**for (cnt=0;cnt<512;cnt++) buff[cnt]=SPI\_ReceiveByte(); //получаем байты блока из шины в буфер**

**SPI\_Release(); //Пропускаем контрольную сумму**

**SPI\_Release();**

  return 0;

}

Add a prototype to this function in the header file and go to the main.c file and add there an empty global array for 512 bytes so that we can store the bytes from the block

volatile uint16\_t Timer1=0;

**uint8\_t sect[512];**

Until we call the function, but return to the **sd.c** file and write the block write function. It will look like a read function, only the command will be used already CMD24. The format of the command is the same, so to give its description there is no sense

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

**uint8\_t SD\_Write\_Block (uint8\_t \*buff, uint32\_t lba)**

**{**

**uint8\_t result;**

**uint16\_t cnt;**

**result=SD\_cmd(CMD24,lba); //CMD24 даташит стр 51 и 97-98**

**if (result!=0x00) return 6; //Выйти, если результат не 0x00**

**SPI\_Release();**

**SPI\_SendByte (0xFE); //Начало буфера**

**for (cnt=0;cnt<512;cnt++) SPI\_SendByte(buff[cnt]); //Данные**

**SPI\_Release(); //Пропустим котрольную сумму**

**SPI\_Release();**

**result=SPI\_ReceiveByte();**

**if ((result&0x05)!=0x05) return 6; //Выйти, если результат не 0x05 (Даташит стр 111)**

**cnt=0;**

**do { //Ждем окончания состояния BUSY**

**result=SPI\_ReceiveByte();**

**cnt++;**

**} while ( (result!=0xFF)&&(cnt<0xFFFF) );**

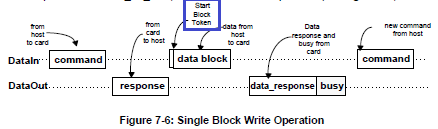
**if (cnt>=0xFFFF) return 6;**

**return 0;**

**}**

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

I think the code is clear. The truth is there is a question about the FE byte that we transmit, it's a sort of block start marker so that the card can determine from which byte to start recording. To understand the process, I will bring the process of writing a block from the technical documentation



I allocated a token. Here is a separate description (page 112)

index61

Create a prototype on the write function and return to the main.c. file again. Add here a global array with any string of 512 bytes. I took an excerpt from the datasheet of the map, I will not bring it completely here. You can take any line

**char buffer1[512] ="Selection ... The..."; //Буфер данных для записи/чтения**

Now call our function in main ()

sd\_ini();

**SD\_Write\_Block((uint8\_t\*)buffer1,0x0400); //Запишем блок в буфер**

We will collect the code and edit the controller, thus the data should be registered in the sector with the given address. Since the map we have the right type, the sector will be far from the beginning and the data, which is not so much on the map, should not suffer.

Now comment this line and add a line with a call to the read function of the block, thus we consider the same bytes, since we use the same address in the input arguments as when writing

//SD\_Write\_Block((uint8\_t\*)buffer1,0x0400); //Запишем блок в буфер

**SD\_Read\_Block(sect,0x0400); //Считаем блок из буфера**

Add a local variable to the main () for the loop counter

/\* USER CODE BEGIN 1 \*/

**uint16\_t i;**

/\* USER CODE END 1 \*/

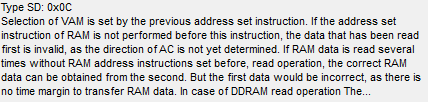
We display the read sector data in the terminal program

SD\_Read\_Block(sect,0x0400); //Считаем блок из буфера

**for(i=0;i<512;i++) HAL\_UART\_Transmit(&huart1,sect+i,1,0x1000);**

**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"rn",2,0x1000);**

We will collect the code, we will tell the controller and see the result (for some reason I have to pause and then the controller also with the button, apparently the costs of the separate power supply)



We see our line. This means that the unit is normally written and read. Excellent. Comment out all this sector read code

**//SD\_Read\_Block(sect,0x0400); //Считаем блок из буфера**

**//for(i=0;i<512;i++) HAL\_UART\_Transmit(&huart1,sect+i,1,0x1000);**

**//HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"rn",2,0x1000);**

Also comment out the array with a string, leave an empty array, it will still come in handy

// char buffer1 [512] = "Sel ...

uint8\_t sect [512];

Now we'll start to slowly cling to the FatFS library so that we can work not with blocks, but with files.

To do this, there is a file user\_discio.c, which is located in the project tree in the **Application / User**section .

If we open this file, then we'll see a few half-empty functions, which we should fill with the code, so that the library can work with our map.

We need the following functions:

USER\_initialize, USER\_status, USER\_read, USER\_write, USER\_ioctl.

That is, all the functions that are in this file.

Let's connect our library to this file first

/\* Includes ------------------------------------------------------------------\*/

#include <string.h>

#include "ff\_gen\_drv.h"

**#include "sd.h"**

/\* Private typedef -----------------------------------------------------------\*/

Then we connect some variables and a string array

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

**extern UART\_HandleTypeDef huart1;**

**extern char str1[60];**

**extern sd\_info\_ptr sdinfo;**

/\* Disk status \*/

Let's see what input arguments are passed in the functions of the file

1) Function **USER\_initialize** :

**BYTE pdrv** is the physical carrier number. We have only one supported and will be zero.

2) **USER\_status** function :

Same.

3) **USER\_read** function :

**BYTE pdrv** - physical media number

**BYTE \* buff** - pointer to the beginning of the buffer for reading data

**DWORD sector** - logical address of the first block

**UINT count** is the number of blocks read.

4) **USER\_write** function :

**BYTE pdrv** - physical media number

**BYTE \* buff** - pointer to the beginning of the buffer for writing data

**DWORD sector** - logical address of the first block

**UINT count** - number of blocks to be written

5) Function **USER\_ioctl** (function of input-output operations):

**BYTE pdrv** - physical media number

**BYTE cmd** - the command code (these are not the commands that are described in the technical documentation for the card, and which ones - we will see later).

**void \* buff** is the address of the buffer to read or write control data.

Now let's first examine the conditions under which we get into some functions of this file. To do this, we place the traps in the form of lines in the terminal.

You can do this of course with the help of breakpoints, but there are certain rumors that it's better not to slow down the work with the card, because the initialization is very quickly overdue and you need to re-run it, although I worked in debugging and these rumors were not confirmed. But nevertheless, after all, the terminal program.

Headings of functions do not result, since from comments it is visible which function, that there was not too much text

/\* USER CODE BEGIN INIT \*/

Stat = STA\_NOINIT;

**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"USER\_initializern",17,0x1000);**

return Stat;

/\* USER CODE END INIT \*/

/\* USER CODE BEGIN STATUS \*/

Stat = STA\_NOINIT;

**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"USER\_statusrn",13,0x1000);**

return Stat;

/\* USER CODE END STATUS \*/

/\* USER CODE BEGIN READ \*/

**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"USER\_readrn",11,0x1000);**

**sprintf(str1,"sector: %lu; count: %drn",sector, count);**

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

return RES\_OK;

/\* USER CODE END READ \*/

/\* USER CODE BEGIN WRITE \*/

/\* USER CODE HERE \*/

**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"USER\_writern",12,0x1000);**

**sprintf(str1,"sector: %lurn",sector);**

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

return RES\_OK;

/\* USER CODE END WRITE \*/

/\* USER CODE BEGIN IOCTL \*/

DRESULT res = *RES\_ERROR*;

**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"USER\_ioctlrn",12,0x1000);**

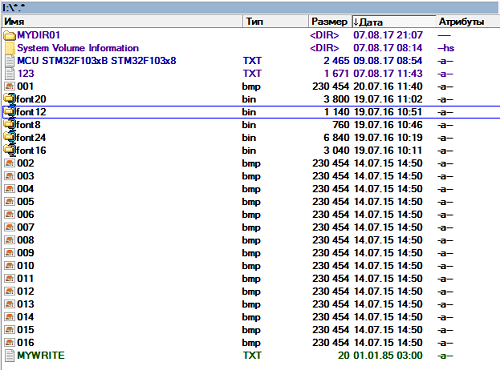
**sprintf(str1,"cmd: %drn",cmd);**

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

return res;

/\* USER CODE END IOCTL \*/

On the flash card I have the following structure of files and directories



Let's go to the **main.c** file and add some number of global variables and structures

uint8\_t sect[512];

**extern char str1[60];**

**uint32\_t byteswritten,bytesread;**

**uint8\_t result;**

**extern char USER\_Path[4]; /\* logical drive path \*/**

**FATFS SDFatFs;**

**FATFS \*fs;**

**FIL MyFile;**

/\* USER CODE END PV \*/

In the main () function, we first comment out the embedding and initialization code for our SD card

**//SD\_PowerOn();**

**//sd\_ini();**

We now call the function to initialize the disk from the FatFS library, although this is not necessary, if necessary, it will be called later, but still let's call this process in advance separately from all other functions of working with the library

//HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"rn",2,0x1000);

**disk\_initialize(SDFatFs.drv);**

/\* USER CODE END 2 \*/

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

index64

We see that the initialization function was called.

So let's go back to the **user\_discio.c** file and write the code into this function, removing something unnecessary

~~Stat = STA\_NOINIT;~~

HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"USER\_initializern",17,0x1000);

**SD\_PowerOn();**

**if(sd\_ini()==0) {Stat &= ~STA\_NOINIT;} //Сбросим статус STA\_NOINIT**

return Stat;

We will collect the code, we will tell the controller and see that our card is normally initialized

index65

We continue our code in main () and start writing the procedure for reading the contents of the file from the SD card. We used to do this several times using the SDIO interface, so I think that there will not be any difficulties with this.

disk\_initialize(SDFatFs.drv);

**//read**

**if(f\_mount(&SDFatFs,(TCHAR const\*)USER\_Path,0)!=FR\_OK)**

**{**

**Error\_Handler();**

**}**

**else**

**{**

**if(f\_open(&MyFile,"123.txt",FA\_READ)!=FR\_OK)**

**{**

**Error\_Handler();**

**}**

**else**

**{**

**f\_close(&MyFile);**

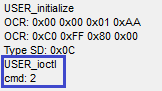
**}**

**}**

/\* USER CODE END 2 \*/

While we do not actually read the file, there is only opening in read mode.

Again, we'll compile the code and run the controller. Let's see the result in the terminal



We see that by calling the function of opening the file, we had a call to the USER\_ioctl function with the command code 2.

Let's see what kind of team it is. And in general we look at the codes of all teams

/**\* Generic command (Used by FatFs) \*/**

**#define CTRL\_SYNC 0 /\* Complete pending write process (needed at \_FS\_READONLY == 0) \*/**

**#define GET\_SECTOR\_COUNT 1 /\* Get media size (needed at \_USE\_MKFS == 1) \*/**

**#define GET\_SECTOR\_SIZE 2 /\* Get sector size (needed at \_MAX\_SS != \_MIN\_SS) \*/**

**#define GET\_BLOCK\_SIZE 3 /\* Get erase block size (needed at \_USE\_MKFS == 1) \*/**

**#define CTRL\_TRIM 4 /\* Inform device that the data on the block of sectors is no longer used (needed at \_USE\_TRIM == 1) \*/**

**/\* Generic command (Not used by FatFs) \*/**

**#define CTRL\_POWER 5 /\* Get/Set power status \*/**

**#define CTRL\_LOCK 6 /\* Lock/Unlock media removal \*/**

**#define CTRL\_EJECT 7 /\* Eject media \*/**

**#define CTRL\_FORMAT 8 /\* Create physical format on the media \*/**

**/\* MMC/SDC specific ioctl command \*/**

**#define MMC\_GET\_TYPE 10 /\* Get card type \*/**

**#define MMC\_GET\_CSD 11 /\* Get CSD \*/**

**#define MMC\_GET\_CID 12 /\* Get CID \*/**

**#define MMC\_GET\_OCR 13 /\* Get OCR \*/**

**#define MMC\_GET\_SDSTAT 14 /\* Get SD status \*/**

**/\* ATA/CF specific ioctl command \*/**

**#define ATA\_GET\_REV 20 /\* Get F/W revision \*/**

**#define ATA\_GET\_MODEL 21 /\* Get model name \*/**

**#define ATA\_GET\_SN 22 /\* Get serial number \*/**

That's how many of them!

But we will need a maximum of 3 kinds of teams, not more.

**Let's see what kind of** command this is with index 2. This is the command **GET\_SECTOR\_SIZE** , whose requirement is to return the number of bytes in the sector in the buffer.

That's what we'll do. We go into the function  **USER\_ioctl** in the file **user\_diskio.c** and write the following code there, slightly correcting the existing one

DRESULT res~~=~~*~~RES\_ERROR~~*;

HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"USER\_ioctlrn",12,0x1000);

sprintf(str1,"cmd: %drn",cmd);

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

**if (pdrv) return RES\_PARERR;**

**if (Stat & STA\_NOINIT) return RES\_NOTRDY;**

**res = *RES\_ERROR*;**

**switch (cmd)**

**{**

**case GET\_SECTOR\_SIZE : /\* Get sectors on the disk (WORD) \*/**

**\*(WORD\*)buff = 512;**

**res = RES\_OK;**

**break;**

**default:**

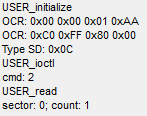
**res = RES\_PARERR;**

**}**

**SPI\_Release();**

return res;

Once again we will collect the code and we will sew the controller. The result in the terminal program will now be the following



We see that we have called the zero-block read function to read the information about the file table (MBR sector).

So let's move on to this function and write the following code

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

**if (pdrv || !count) return RES\_PARERR;**

**if (Stat & STA\_NOINIT) return RES\_NOTRDY;**

**if (!(sdinfo.type & 4)) sector \*= 512; /\* Convert to byte address if needed \*/**

**if (count == 1) /\* Single block read \*/**

**{**

**SD\_Read\_Block(buff,sector); //Считаем блок в буфер**

**count = 0;**

**}**

**else /\* Multiple block read \*/**

**{**

**}**

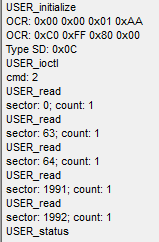
**SPI\_Release();**

**return count ? RES\_ERROR : RES\_OK;**

/\* USER CODE END READ \*/

The code was not so complicated. We do not process the reading of several sectors, since only one sector is requested from us, and I will say ahead of time that most likely we have several and will not be asked. Also note that if we do not have a SDHC card, then we multiply the block number by 512, thereby determining the byte-by-byte address and not the block one.

Once again, try to collect the code and flash the controller, and then see the result in the terminal



We see that we had requested a few more sectors, and then the status was requested by calling the corresponding function.

Few tweaks the code in the status call function

~~Stat = STA\_NOINIT;~~

HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"USER\_statusrn",13,0x1000);

**if (pdrv) return STA\_NOINIT;**

return Stat;

If we check the code, then we will see the same with the only difference that the call to the status determination function will be executed twice.

Now go to the **main.c** file and take care of actually reading the contents of our file. But since from our considerable practice of working with files we remember that it is not easy to read large blocks, it is better to operate with small portions, and we will spend less time on memory, then we will write a separate function for this

/\* USER CODE BEGIN 0 \*/

**FRESULT ReadLongFile(void)**

**{**

**uint16\_t i=0, i1=0;**

**uint32\_t ind=0;**

**uint32\_t f\_size = MyFile.fsize;**

**sprintf(str1,"fsize: %lurn",(unsigned long)f\_size);**

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

**ind=0;**

**do**

**{**

**if(f\_size<512)**

**{**

**i1=f\_size;**

**}**

**else**

**{**

**i1=512;**

**}**

**f\_size-=i1;**

**f\_lseek(&MyFile,ind);**

**f\_read(&MyFile,sect,i1,(UINT \*)&bytesread);**

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

**{**

**HAL\_UART\_Transmit(&huart1,sect+i,1,0x1000);**

**}**

**ind+=i1;**

**}**

**while(f\_size>0);**

**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"rn",2,0x1000);**

**return *FR\_OK*;**

**}**

/\* USER CODE END 0 \*/

The code of the function was almost taken from lessons on the output of picture files to the display screen, so there is no need for an explanation. It only needs to be recalled that we read the contents of the file in portions of 512 bytes. We do not pass the name of the file, since it is already in the structure.

We now call this function in the appropriate place in the code of **main ()**

else

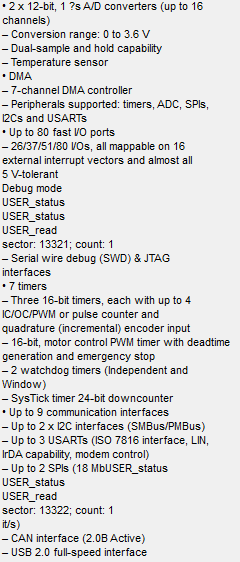
{

**ReadLongFile();**

  f\_close(&MyFile);

}

We'll collect the code, we'll tell the controller and see the contents of our file in the terminal program. I will give only the last part of the text from the file, since it is too large. This is an excerpt from the technical documentation for our controller and I specifically took from there more to track the work with several blocks



We see that sometimes in the text there is information about calling functions and about reading certain sectors. We will then remove the output of this information, but for now we need it.

In the [**next part of**](http://narodstream.ru/stm-urok-88-sd-spi-fatfs-chast-4/) this lesson, we will learn how to write files to an SD card, and also get information from it in the form of a list of files and directories, as well as the overall size and free size for the data.

**Lesson 88**

**Part 4**

# SD. SPI. FATFS

In the [**previous part of**](http://narodstream.ru/stm-urok-88-sd-spi-fatfs-chast-3/) this lesson, we added the necessary code to the user functions of the FATFS library and read the text file from the SD card.

Well, let's assume that we coped with reading the contents of the files. Now write to the file.

To start with the main () function, add several local variables

uint16\_t i;

**FRESULT res; //результат выполнения**

**uint8\_t wtext[]="Hello from STM32!!!";**

/\* USER CODE END 1 \*/

Let's comment the code for reading the file in the main () function and write the write code. We do not plan to write large files yet, so we will not write a separate function for writing, but we will write the following code familiar to us about past studies

**//write**

**if(f\_mount(&SDFatFs,(TCHAR const\*)USER\_Path,0)!=FR\_OK)**

**{**

**Error\_Handler();**

**}**

**else**

**{**

**if(f\_open(&MyFile,"mywrite.txt",FA\_CREATE\_ALWAYS|FA\_WRITE)!=FR\_OK)**

**{**

**Error\_Handler();**

**}**

**else**

**{**

**res=f\_write(&MyFile,wtext,sizeof(wtext),(void\*)&byteswritten);**

**if((byteswritten==0)||(res!=FR\_OK))**

**{**

**Error\_Handler();**

**}**

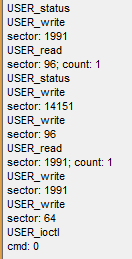
**f\_close(&MyFile);**

**}**

**}**

/\* USER CODE END 2 \*/

Although we have not written the recording function yet, we will still compile the code and edit the controller, only before this, in order to accurately verify that everything is written down, delete this file from the card, because from there is present



Looking at the output in the terminal program, we can assume that everything is already written down, but if you remove the card, insert it into the card reader and look at the contents on the PC, we will see that the file is not even created.

So let's write the code in the **USER\_write** function in the file **user\_diskio.c**

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

**if (pdrv || !count) return RES\_PARERR;**

**if (Stat & STA\_NOINIT) return RES\_NOTRDY;**

**if (Stat & STA\_PROTECT) return RES\_WRPRT;**

**if (!(sdinfo.type & 4)) sector \*= 512; /\* Convert to byte address if needed \*/**

**if (count == 1) /\* Single block read \*/**

**{**

**SD\_Write\_Block((BYTE\*)buff,sector); //Считаем блок в буфер**

**count = 0;**

**}**

**else /\* Multiple block read \*/**

**{**

**}**

**SPI\_Release();**

**return count ? RES\_ERROR : RES\_OK;**

/\* USER CODE END WRITE \*/

The code was no more complex than the code in the sector reading function.

We also saw in the terminal output that at the end we are calling the USER\_Ioctl function with the 0 command, which we have not yet processed.

Correct this defect by first going into the sd.c file and writing there some waiting function after the function **SPI\_Release**

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

**uint8\_t SPI\_wait\_ready(void)**

**{**

**uint8\_t res;**

**uint16\_t cnt;**

**cnt=0;**

**do { //Ждем окончания состояния BUSY**

**res=SPI\_ReceiveByte();**

**cnt++;**

**} while ( (res!=0xFF)&&(cnt<0xFFFF) );**

**if (cnt>=0xFFFF) return 1;**

**return res;**

**}**

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

Add a prototype to this function and return to the **user\_diskio.c** file  in the **USER\_write** function .

We will add there the necessary case

switch (cmd)

{

**case CTRL\_SYNC : /\* Flush dirty buffer if present \*/**

**SS\_SD\_SELECT();**

**if (SPI\_wait\_ready() == 0xFF)**

**res = RES\_OK;**

**break;**

  case GET\_SECTOR\_SIZE : /\* Get sectors on the disk (WORD) \*/

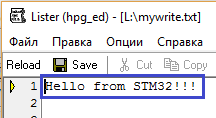
Once again, we'll collect the code and patch the controller. Here is the output to terminal

index71

The status was requested, perhaps everything went well. To find out, extract the card and see its contents on the PC. And there is. The file appeared in size 20 bytes

index73

Let's see also its contents



The content corresponds to the contents of the buffer that we wrote down. So everything is fine. With the record, too, figured out.

Let's try some more functionality of the map and the FATFS library.

First, we comment out the write code for the map in the **main ()** function , and add the call to the deinitialization function

\*/

**FATFS\_UnLinkDriver(USER\_Path);**

/\* USER CODE END 2 \*/

We will try to read the file contents of our map, that is, somehow get a list of files and directories.

Let's add some local variables and structures to the **main ()** function

uint8\_t wtext[]="Hello from STM32!!!";

**FILINFO fileInfo;**

**char \*fn;**

**DIR dir;**

**DWORD fre\_clust, fre\_sect, tot\_sect;**

**/\* USER CODE END 1 \*/**

Now add some code also to main ()

**//read dir**

**if(f\_mount(&SDFatFs,(TCHAR const\*)USER\_Path,0)!=FR\_OK)**

**{**

**Error\_Handler();**

**}**

**else**

**{**

**fileInfo.lfname = (char\*)sect;**

**fileInfo.lfsize = sizeof(sect);**

**result = f\_opendir(&dir, "/");**

**if (result == FR\_OK)**

**{**

**f\_closedir(&dir);**

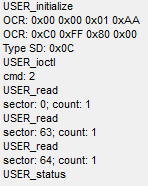
**}**

**}**

FATFS\_UnLinkDriver(USER\_Path);

We mount the file system, then bind the structure fields to certain variables that determine the memory addresses, then, using the desired function of the FATFS library, read the root directory in the DIR structure and, if the result is successful, close the directory.

There will be a lot of code here, but let's just collect what we have and we'll sew the controller. Let's see the result of our actions



So far we see only attempts to read some sectors. Apparently, our structure was initialized. We still do not see the list of files and directories, since we have not written the code for this yet.

Let's continue the code

if (result == FR\_OK)

{

**while(1)**

**{**

**result = f\_readdir(&dir, &fileInfo);**

**if (result==FR\_OK && fileInfo.fname[0])**

**{**

**}**

**else break;**

**}**

  f\_closedir(&dir);

Here we will have an infinite loop, with which we will track that the list of files and directories in the root folder is over. We define this by a null byte in the array **fname of** the **fileinfo** structure .

We initialize this structure with the function f\_readdir, which uses the DIR structure and brings all the data of the next record to the file system. and after the call of this function the pointer automatically passes to the next file-catalog entry in the root directory.

Now we continue the code in the body of the condition

if (result==FR\_OK && fileInfo.fname[0])

{

**fn = fileInfo.lfname;**

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

**else HAL\_UART\_Transmit(&huart1,(uint8\_t\*)fileInfo.fname,strlen((char\*)fileInfo.fname),0x1000);**

**if(fileInfo.fattrib&AM\_DIR)**

**{**

**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)" [DIR]",7,0x1000);**

**}**

}

In this body, we first assume that the file or directory name is long (longer than 8 characters) and will bind this field to a string array. Then, if we define that this array will be non-zero length, then we will output this long name to the terminal program. If the array is of zero length, then the name is short and we will deduce it already in the terminal program using the **fname** field .

Next, by the value of the attribute, we define that this is not a file, but a directory, in this case we will output a specific line **[DIR]** to the terminal .

Now, after the condition, including after the nasty component, we pass to a new line

  else break;

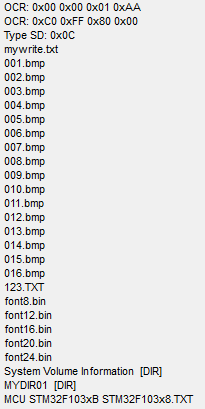
**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"rn",2,0x1000);**

}

f\_closedir(&dir);

In the file  **user\_diskio.c** comment out all the output to the terminal program, otherwise there will be too many output of the file-catalog list.

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



Excellent! We see the entire list of directories. There are many more interesting functions in the FATFS library that can create directories, delete files and directories, add content to files, but with that, I think you'll figure it out for yourself, we do not need it yet. Also, we can list any directory by correcting the name right here.

result = f\_opendir(&dir, "**/MYDIR01**");

We'll collect the code, we'll tell the controller and see the contents of this directory

index77

After displaying information about files and directories, we will try to display some more important information that we may well need when working with the map

  HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"rn",2,0x1000);

}

**f\_getfree("/", &fre\_clust, &fs);**

**sprintf(str1,"fre\_clust: %lurn",fre\_clust);**

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

**sprintf(str1,"n\_fatent: %lurn",fs->n\_fatent);**

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

**sprintf(str1,"fs\_csize: %drn",fs->csize);**

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

**tot\_sect = (fs->n\_fatent - 2) \* fs->csize;**

**sprintf(str1,"tot\_sect: %lurn",tot\_sect);**

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

**fre\_sect = fre\_clust \* fs->csize;**

**sprintf(str1,"fre\_sect: %lurn",fre\_sect);**

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

**sprintf(str1, "%lu KB total drive space.rn%lu KB available.rn",**

**fre\_sect/2, tot\_sect/2);**

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

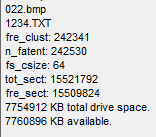
f\_closedir(&dir);

We are trying to learn some sizes.

To do this, we call the library FATFS **f\_getfree** , which will determine the available space on the map, as a parameter passing it the name of the root folder. This function will write the value into a variable that will carry the number of clusters. But as we know, the cluster is a very abstract value and it will not tell us much about the free space, because we want to see it in bytes. Nevertheless, we will also display this value in the terminal.

Also, the above function will not only determine the size in clusters, it also initializes the **fs** file structure , from which we take some more information. First we take from it the size in the clusters complete and not free (the field **fatent** ) and display it in the terminal. Then we **'ll** do the same with the **csize** field , which carries the information about the cluster size in kilobytes, and then it will be easy for us to calculate the size in bytes of the total and the remaining one.

We will collect the code, we will sew the controller. And here is the result



Correctness of this information, I checked, connecting the card to the computer, it all came down to kilobyte.

So thank you all for attention!

I hope that my lesson was useful to many. Wait for a new lesson!