**Unit 91**

**Part 1**

**LAN. W5500. HTTP Server**

Today we will continue the theme of working with LAN. Only for these purposes we will take another chip - **W5500** . This chip is developed by **Wiznet** and is interesting in that it includes a whole hardware stack of TCP / IP protocols besides the physical and data link layer. Also, compared to the **ENC28J60** chip , with which we worked for a long time, consistently and carefully studying the operation of protocols of different levels, the **W5500** has much more computing power. The shared memory for read and write buffers is 32 kilobytes, and buffers are also divided into sockets, which allow simultaneous work with several connections.

It should be noted that this manufacturer, in addition to this chip, in the line of LAN-chips with a ready stack there are several other similar chips that differ in power, interface connection and many other parameters.

The W5500 chip differs from its counterparts in that it has for communication with the controller and only **SPI** interface for data transfer between it and the controller . Some other chips in this line also have a parallel interface for communication, which allows faster exchange between this chip and the MC. But the W5500 unlike other chips of this line is more convenient for wide use because the case has a smaller number of legs, which makes it easier to install it on the board.

Also I want to note that the chip works already in more advanced modes at the physical level, rather than ENC28J60. It supports the following modes:

10BT Half-duplex,

10BT Full-duplex,

100BT Half-duplex,

100BT Full-duplex.

Also supported is the **Auto-negotiation** mode , which allows you to automatically switch the speed depending on the speed of the device with which it is exchanged over the LAN.

The **auto-MDIX** mode in the chip is unfortunately not implemented. This, if you know, a mode that allows you to connect devices with a straight cable without the use of routers, switches and switches. It is by connecting them with a cross-over cable. And if the device, to which we connect a given microcircuit via LAN, supports this mode, then we can safely use a straight cable.

Well, now the most important thing.

Which protocols from the TCP / IP stack support hardware?

Here is their list: **TCP, UDP, ICMP, IPv4, ARP, IGMP, PPPoE** .

And with all the other delights and pitfalls, we will get acquainted already with the programming of this chip in our project.

As you can see from the topic of the lesson, our project will be aimed at organizing a small **HTTP** server , which will support only one connection so far. I think it's good for a start. Since to a similar server on the chip, we considered earlier we went for a good ten lessons.

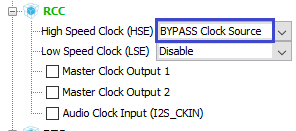
This chip for the project we will use in the form of inexpensive (as they say, people's) module, which can be fully purchased at a price of about 5 dollars. I think this is a ridiculous price for such a module. Manufacturer - **Waveshare** , whose products we very often use. He looks like this



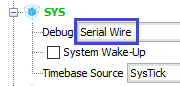
Controller for our project, we will use **STM32F401RET6** , which is located on the same motherboard **NUCLEO-F401RE** .

Start the **Cube-MX** project generator and create a new project there, selecting the STM32F401RETx controller.

Let's configure the clock from ST-Link



Turn on the programmer for debugging



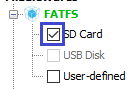
Also include the SPI interface for connecting our module

image03

Documents for the transfer of the client we will use as files located on the Micro-SD card. Since our controller supports SDIO hardware, we will include it

image04

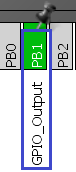
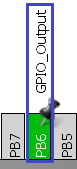
Accordingly, include and FATFS



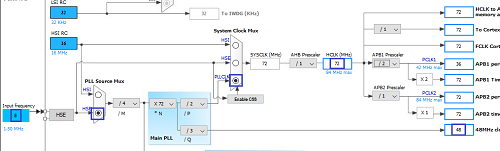
Include USART for easy project debugging in real time

image06

Also include the legs **PB1** and **PB6** on the output for the **RESET** and **CS** module respectively . It was these legs that were chosen for easy connection according to the pinout of the Nucleo board

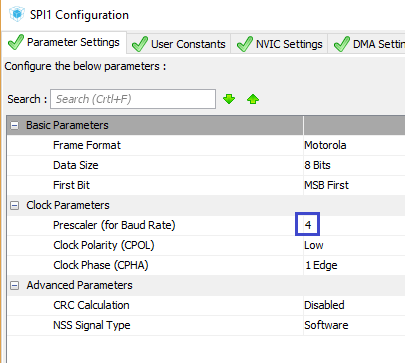
  

Now adjust the parameters in the **Clock Configuration** section  (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/09/image09.png)

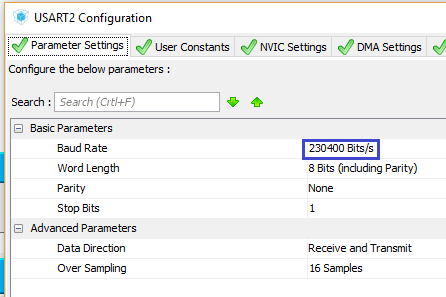
Go to the **Configuration** section .

First, configure **SPI1.**We turn on the divider 4, which will allow us to work with the SPI bus at a speed of 18 megahertz (megabit per second)

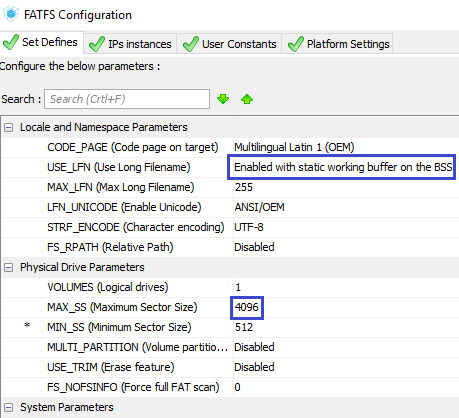


The chip supports the frequency of work on the SPI bus up to 80 megahertz, but this is only theoretically, practically checked up to 33.3. So it is written in the technical documentation. I really put divisor 2, which corresponded to 36 MHz, and the module normally transmitted everything and accepted it. But we will not take any chances. You can always try. And then suddenly there will be some glitches because of this, and we'll start thinking badly about our code.

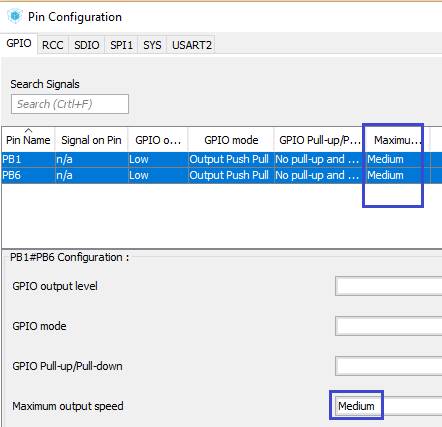
Now **USART**



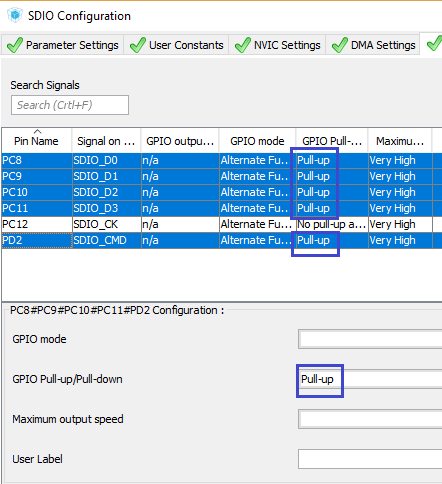
Further **FATFS** . As usual, enable long names and long sectors



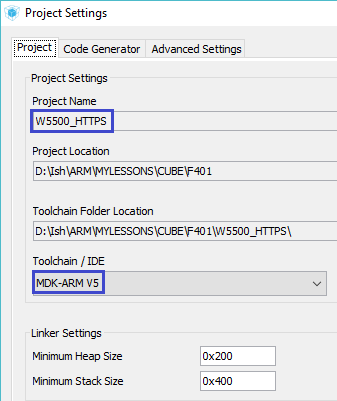
Also, we'll add a bit of speed to the GPIO service feet



Well, that's probably the most important thing for correct operation, since we do not have pull-up resistors in the SD module, we will tighten them to certain **SDIO** bus legs



Let's now go to the project settings window and select Keil as the programming environment, and also give the project the name **W5500\_HTTPS**



Save the settings, generate the project, open it in Keil, configure the programmer for auto-cutting, and also configure the code optimization to level 1.

Let's try to assemble the project.

If everything is fine, then we'll start writing code.

Our project, of course, may not be assembled because of the lack of the **ccsbcs.c** file . Take it and connect from the project  [lesson](http://narodstream.ru/stm-urok-89-lan-enc28j60-tcp-web-server-podklyuchaem-kartu-sd/)**79 ENC28J60\_HTTPS\_SD** .

Now the project is likely to be assembled.

Next, create two files **net.h** and **net.c** to manage the LAN interface, as follows:

**net.h**

**#ifndef \_\_NET\_H**

**#define \_\_NET\_H**

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

**#include "stm32f4xx\_hal.h"**

**#include <string.h>**

**#include <stdlib.h>**

**#include <stdint.h>**

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

**#define IP\_ADDR {192,168,1,197}**

**#define IP\_GATE {192,168,1,1}**

**#define IP\_MASK {255,255,255,0}**

**#define LOCAL\_PORT 80**

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

**#define be16toword(a) ((((a)>>8)&0xff)|(((a)<<8)&0xff00))**

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

**#endif /\* \_\_NET\_H \*/**

**net.c**

**#include "net.h"**

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

**extern UART\_HandleTypeDef huart2;**

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

**uint8\_t ipaddr[4]=IP\_ADDR;**

**uint8\_t ipgate[4]=IP\_GATE;**

**uint8\_t ipmask[4]=IP\_MASK;**

**uint16\_t local\_port = LOCAL\_PORT;**

**char str1[60]={0};**

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

In the **main.c** file we also connect this library

/\* USER CODE BEGIN Includes \*/

**#include "net.h"**

/\* USER CODE END Includes \*/

In the file **net.c** we will create a hollow function for receiving network packets

char str1[60]={0};

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

**void packet\_receive(void)**

**{**

**}**

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

Below we will write one more function for network exchange, in which we call the previous one

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

**void net\_poll(void)**

**{**

**packet\_receive();**

**}**

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

Let's create a prototype for this function and call it in an infinite loop of the main module of **main ()**

/\* USER CODE BEGIN 3 \*/

**net\_poll();**

}

/\* USER CODE END 3 \*/

We also create in the file **net.c** the initialization function

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

**void net\_ini(void)**

**{**

**}**

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

Let's create a prototype for it and call it in the main module in **main ()**

/\* USER CODE BEGIN 2 \*/

**net\_ini();**

/\* USER CODE END 2 \*/

We will also create 2 more files **w5500.h** and **w5500.c** for a lower level of communication with the microcircuit with the following contents

**w5500.h**

**#ifndef W5500\_H\_**

**#define W5500\_H\_**

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

**#include "stm32f4xx\_hal.h"**

**#include <string.h>**

**#include <stdlib.h>**

**#include <stdint.h>**

**#include "fatfs.h"**

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

**#define CS\_GPIO\_PORT GPIOB**

**#define CS\_PIN GPIO\_PIN\_6**

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

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

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

**#define MAC\_ADDR {0x00,0x15,0x42,0xBF,0xF0,0x51}**

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

**#define be16toword(a) ((((a)>>8)&0xff)|(((a)<<8)&0xff00))**

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

**#endif /\* W5500\_H\_ \*/**

**w5500.c**

**#include "w5500.h"**

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

**extern SPI\_HandleTypeDef hspi1;**

**extern UART\_HandleTypeDef huart2;**

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

**extern char str1[60];**

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

**uint8\_t macaddr[6]=MAC\_ADDR;**

**extern uint8\_t ipaddr[4];**

**extern uint8\_t ipgate[4];**

**extern uint8\_t ipmask[4];**

**extern uint16\_t local\_port;**

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

**static void Error (void)**

**{**

**HAL\_UART\_Transmit(&huart2,(uint8\_t\*)"Error!rn",8,0x1000);**

**}**

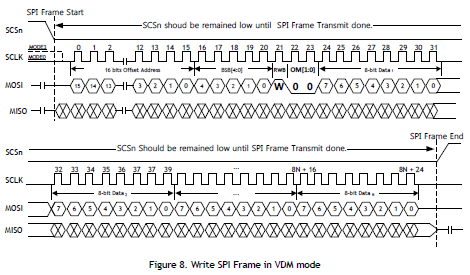
**//-----------------------------------------------**

Before we start writing the initialization function, we'll get a little acquainted with the memory organization. Firstly, there are two types of registers. One type is general purpose registers. There are stored settings and properties that are common to all sockets. And the second type is registers for sockets, in which the settings and properties for a particular socket are stored.

And the memory is organized as follows. In total for the data memory 32 kilobytes, half of which is reserved for the received data, and the other half for the sent data. Each half is broken by default into 8 parts - 2 kilobytes each. Each part is for a specific socket. At our request, the boundaries of these buffers can move. For example, we can allocate 9 kilobytes to a null socket, and the rest 7 - one by one or as we please. While we leave everything by default and do not touch it. And in general, as I wrote above, we will work with only one socket, although we will organize some variables, arrays and structures for working with several sockets, so to speak (future).

It's about memory. Now for how the data is transmitted via the SPI interface of the chip. There are several ways to transfer and receive data. But they are divided into two types. This is sending and receiving a fixed number of data bytes (1, 2 or 4 bytes) and sending and receiving variable-sized data. We need both ways, so let's see how this is generally done.

Write a variable-length data packet to the SPI bus



First, we transfer a 16-bit memory address (register), respectively, the first byte - the senior, the second - the younger one. Then we pass the code of the operation, which we will examine in more detail below, and then - the data one by one. The end of the data transfer chip recognizes the raised leg of the ChipSelect.

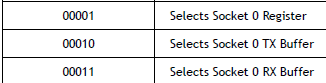
Now about the byte of the operation code, which looks like this

image17

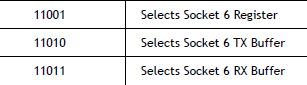
The **BSB (Block Select Bits)** bits  are a pointer to the block with which we are working. The first 3 bits - BSB4-BSB2 - contain the number of the socket, and the last two - the memory block.

Accordingly, the block of registers common for all sockets has the number 00000

For example, here such numbers have socket blocks 0

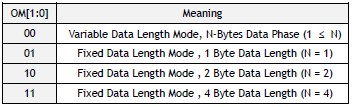


And here such, for example - a socket 6

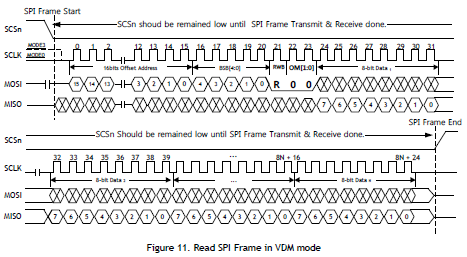


The **RWB (Read / Write Access Mode Bit)**  of the operation code is responsible for the type of operation. If it is 1 - then this is a record, and if 0 is a read.

And the bit pair  **OM (Operation Mode)**  is just responsible for the type of transmission / reception, that is, the fixed length of the data will be exchange or variable. Here are their options here



The next mode is reading variable length data from the SPI bus



Here we also first transmit 2 bytes of the register address (memory), then we pass the opcode with the read / write bit that is cleared, through which the chip recognizes that we want it to transfer data, and will start from this address and then transfer by byte information from memory, until we stop this process by raising the foot of the CS.

Well, the modes of reading and writing fixed length data work in a similar way, only the transmission / reception is carried out by a strict number of bytes.

Well, with the registers themselves we will get acquainted as we write the code of our project.

In the [**next part of the**](http://narodstream.ru/stm-urok-91-lan-w5500-http-server-chast-2/) lesson we will write the initialization function of the chip, and also start writing the function of receiving and processing network packets.

**Unit 91**

**Part 2**

# LAN. W5500. HTTP Server

In the [**previous part of the**](http://narodstream.ru/stm-urok-91-lan-w5500-http-server-chast-1/) lesson we got acquainted with the W5500 module, studied the organization of memory and data exchange of the W5500 chip, created and configured the project.

Initialization begins with a reboot of the module. I took it from the example written for the evaluation fee and posted on the official website, in the technical documentation, I did not find it.

So start writing the initialization function in **w5500.c**

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

**void w5500\_ini(void)**

**{**

**uint8\_t dtt=0;**

**uint8\_t opcode=0;**

**//Hard Reset**

**HAL\_GPIO\_WritePin(GPIOB, GPIO\_PIN\_1, *GPIO\_PIN\_RESET*);**

**HAL\_Delay(70);**

**HAL\_GPIO\_WritePin(GPIOB, GPIO\_PIN\_1, *GPIO\_PIN\_SET*);**

**HAL\_Delay(70);**

**}**

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

Let's write a prototype for this function and connect our module in the file **net.h**

#include <stdint.h>

**#include "w5500.h"**

In the file net.c in the corresponding function, we call our initialization function

void net\_ini(void)

{

**w5500\_ini();**

}

After a hardware reset using the RESET foot, we will do a software reset.

To do this, first write some macros in the file **w5500.h**

#define MAC\_ADDR {0x00,0x15,0x42,0xBF,0xF0,0x51}

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

**#define BSB\_COMMON 0x00**

**#define BSB\_S0 0x01**

**#define BSB\_S0\_TX 0x02**

**#define BSB\_S0\_RX 0x03**

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

**#define RWB\_WRITE 1**

**#define RWB\_READ 0**

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

**#define OM\_FDM0 0x00//режим передачи данных переменной длины**

**#define OM\_FDM1 0x01//режим передачи данных по одному байту**

**#define OM\_FDM2 0x02//режим передачи данных по два байта**

**#define OM\_FDM3 0x03//режим передачи данных по четыре байта**

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

**#define MR 0x0000//Mode Register**

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

Above the initialization function in the file w5500.c we write the function of writing the byte to the register

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

**void w5500\_writeReg(uint8\_t op, uint16\_t addres, uint8\_t data)**

**{**

**uint8\_t buf[] = {addres >> 8, addres, op|(RWB\_WRITE<<2), data};**

**SS\_SELECT();**

**HAL\_SPI\_Transmit(&hspi1, buf, 4, 0xFFFFFFFF);**

**SS\_DESELECT();**

**}**

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

We consecutively write the register address (memory location) into the 4-byte buffer, then the operation code with the **OR** bit associated with it and the data byte proper. Then we omit the selection leg, pass the 4 bytes using the corresponding HAL library function, and then lift the device selection leg.

And now we'll write the code for the software reset of the module into the initialization function

HAL\_Delay(70);

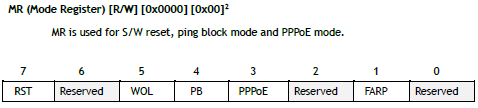
**//Soft Reset**

**opcode = (BSB\_COMMON<<3)|OM\_FDM1;**

**w5500\_writeReg(opcode, MR, 0x80);**

**HAL\_Delay(100);**

Register we use the general **MR,** and the fixed transfer mode of one byte of data



We only cock the 7th bit in it. After that, you need to wait until the bit is reset. So we find out that the reboot has occurred. But it's quite enough just to wait a few milliseconds, not often we initialize the microcircuit.

Next, we will configure all the addresses for our microcircuit by putting them in certain registers as well. Before this, we add macros to the header file **w5500.h** for this **.**

#define MR 0x0000//Mode Register

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

**#define SHAR0 0x0009//Source Hardware Address Register MSB**

**#define SHAR1 0x000A**

**#define SHAR2 0x000B**

**#define SHAR3 0x000C**

**#define SHAR4 0x000D**

**#define SHAR5 0x000E// LSB**

**#define GWR0 0x0001//Gateway IP Address Register MSB**

**#define GWR1 0x0002**

**#define GWR2 0x0003**

**#define GWR3 0x0004// LSB**

**#define SUBR0 0x0005//Subnet Mask Register MSB**

**#define SUBR1 0x0006**

**#define SUBR2 0x0007**

**#define SUBR3 0x0008// LSB**

**#define SIPR0 0x000F//Source IP Address Register MSB**

**#define SIPR1 0x0010**

**#define SIPR2 0x0011**

**#define SIPR3 0x0012// LSB**

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

Which register is responsible for that, can be seen from the comments to it.

Let's pass to initialization function in file **w5500.c** and configure these registers

HAL\_Delay(100);

**//Configute Net**

**w5500\_writeReg(opcode, SHAR0,macaddr[0]);**

**w5500\_writeReg(opcode, SHAR1,macaddr[1]);**

**w5500\_writeReg(opcode, SHAR2,macaddr[2]);**

**w5500\_writeReg(opcode, SHAR3,macaddr[3]);**

**w5500\_writeReg(opcode, SHAR4,macaddr[4]);**

**w5500\_writeReg(opcode, SHAR5,macaddr[5]);**

**w5500\_writeReg(opcode, GWR0,ipgate[0]);**

**w5500\_writeReg(opcode, GWR1,ipgate[1]);**

**w5500\_writeReg(opcode, GWR2,ipgate[2]);**

**w5500\_writeReg(opcode, GWR3,ipgate[3]);**

**w5500\_writeReg(opcode, SUBR0,ipmask[0]);**

**w5500\_writeReg(opcode, SUBR1,ipmask[1]);**

**w5500\_writeReg(opcode, SUBR2,ipmask[2]);**

**w5500\_writeReg(opcode, SUBR3,ipmask[3]);**

**w5500\_writeReg(opcode, SIPR0,ipaddr[0]);**

**w5500\_writeReg(opcode, SIPR1,ipaddr[1]);**

**w5500\_writeReg(opcode, SIPR2,ipaddr[2]);**

**w5500\_writeReg(opcode, SIPR3,ipaddr[3]);**

Since the data registers are also common, the opcode does not change with us.

Now configure the port for socket 0.

To do this, add registers for the port number to file **w5500.h**

#define SIPR3 0x0012// LSB

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

**#define Sn\_PORT0 0x0004 // Socket 0 Source Port Register MSB**

**#define Sn\_PORT1 0x0005 // Socket 0 Source Port Register LSB**

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

Now, in the initialization function in the **w5500.h** file, **let's** put our port data in the registers

w5500\_writeReg(opcode, SIPR3,ipaddr[3]);

**//Настраиваем сокет 0**

**opcode = (BSB\_S0<<3)|OM\_FDM1;**

**w5500\_writeReg(opcode, Sn\_PORT0,local\_port>>8);**

**w5500\_writeReg(opcode, Sn\_PORT1,local\_port);**

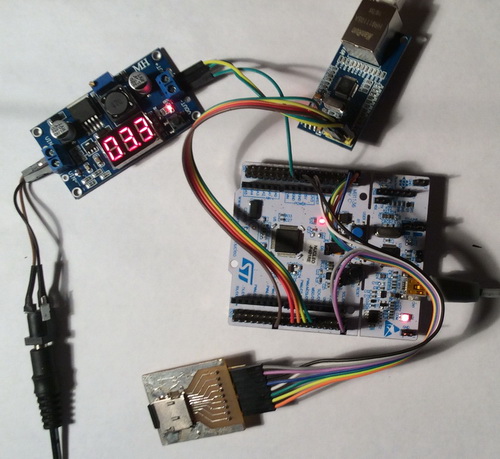
Here we have already changed the type of the register, so we initialize the opcode again.

In principle, at this step we can already collect the code and flash the controller.

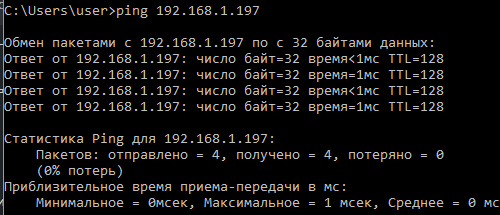
And we already have our module will be ping. That is, ICMP and ARP will work.

Let's do it.

Before flashing the controller, let's look at our schematic (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/09/image03.jpg)

Let's run the controller and check the availability of our module



Everything is working! You can go on.

In the **w5500.h** file, **we** add a structure for the connection properties

#define be16toword(a) ((((a)>>8)&0xff)|(((a)<<8)&0xff00))

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

**typedef struct tcp\_prop {**

**volatile uint8\_t cur\_sock;//активный сокет**

**} tcp\_prop\_ptr;**

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

Add a global variable of the type of our structure in the file **w5500.c**

extern char str1[60];

**tcp\_prop\_ptr tcpprop;**

In the initialization function, we initialize the active (current) socket

w5500\_writeReg(opcode, Sn\_PORT1,local\_port);

**//инициализируем активный сокет**

**tcpprop.cur\_sock = 0;**

After the register write function, we add the function of reading data from the register

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

**uint8\_t w5500\_readReg(uint8\_t op, uint16\_t addres)**

**{**

**uint8\_t data;**

**uint8\_t wbuf[] = {addres >> 8, addres, op, 0x0};**

**uint8\_t rbuf[4];**

**SS\_SELECT();**

**HAL\_SPI\_TransmitReceive(&hspi1, wbuf, rbuf, 4, 0xFFFFFFFF);**

**SS\_DESELECT();**

**data = rbuf[3];**

**return data;**

**}**

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

Here we work somewhat differently. We type the same into the buffer, but we do not enable the write bit, but we can use any data byte, write it to buffer 0. Then we organize another buffer and call the function of circular data exchange on the SPI bus, which will return to us the last element of the array to us byte from the register.

In the w5500.h file, we'll add a few more macros with register addresses and some parameters and states that we'll get to know later

#define Sn\_PORT1 0x0005 // Socket 1 Source Port Register 1 LSB

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

**#define Sn\_MR 0x0000 // Socket 0 Mode Register**

**#define Sn\_CR 0x0001 // Socket 0 Command Register**

**#define Sn\_SR 0x0003 // Socket 0 Status Register**

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

**//Socket mode**

**#define Mode\_CLOSED 0x00**

**#define Mode\_TCP 0x01**

**#define Mode\_UDP 0x02**

**#define Mode\_MACRAV 0x04**

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

**//Socket states**

**#define SOCK\_CLOSED 0x00**

**#define SOCK\_INIT 0x13**

**#define SOCK\_LISTEN 0x14**

**#define SOCK\_ESTABLISHED 0x17**

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

**#define Sn\_MSSR0 0x0012**

**#define Sn\_MSSR1 0x0013**

**#define Sn\_TX\_FSR0 0x0020**

**#define Sn\_TX\_FSR1 0x0021**

**#define Sn\_TX\_RD0 0x0022**

**#define Sn\_TX\_RD1 0x0023**

**#define Sn\_TX\_WR0 0x0024**

**#define Sn\_TX\_WR1 0x0025**

**#define Sn\_RX\_RSR0 0x0026**

**#define Sn\_RX\_RSR1 0x0027**

**#define Sn\_RX\_RD0 0x0028**

**#define Sn\_RX\_RD1 0x0029**

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

Let's **go to w5500.c** and add two more functions to initialize and wait for the socket to be initialized

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

**void OpenSocket(uint8\_t sock\_num, uint16\_t mode)**

**{**

**uint8\_t opcode=0;**

**opcode = (((sock\_num<<2)|BSB\_S0)<<3)|OM\_FDM1;**

**w5500\_writeReg(opcode, Sn\_MR, mode);**

**w5500\_writeReg(opcode, Sn\_CR, 0x01);**

**}**

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

**void SocketInitWait(uint8\_t sock\_num)**

**{**

**uint8\_t opcode=0;**

**opcode = (((sock\_num<<2)|BSB\_S0)<<3)|OM\_FDM1;**

**while(1)**

**{**

**if(w5500\_readReg(opcode, Sn\_SR)==SOCK\_INIT)**

**{**

**break;**

**}**

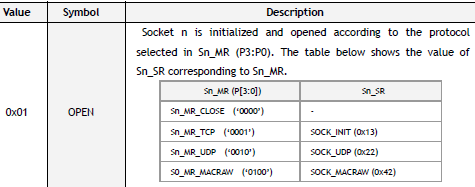
**}**

**}**

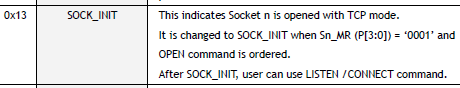
**//-----------------------------------------------**

We will add prototypes for these functions.

In the first function, we enter the socket mode register corresponding to the socket with the number that came in the incoming parameter, we enter the type of the mode of operation that will also come in the incoming parameter. Then we set the bit in the socket control register that corresponds to the socket opening command. Here this tablet at once to us about all this and will tell



And in the second function, we query the socket status register, and wait for the value 0x13 corresponding to the state of "INIT"



Call these functions in the initialization function

tcpprop.cur\_sock = 0;

**//Открываем сокет 0**

**OpenSocket(0,Mode\_TCP);**

**SocketInitWait(0);**

Now we will have to serve this socket, that is, watch out for when a TCP packet comes to us from the client. For this we also write two functions

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

**void ListenSocket(uint8\_t sock\_num)**

**{**

**uint8\_t opcode=0;**

**opcode = (((sock\_num<<2)|BSB\_S0)<<3)|OM\_FDM1;**

**w5500\_writeReg(opcode, Sn\_CR, 0x02); //LISTEN SOCKET**

**}**

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

**void SocketListenWait(uint8\_t sock\_num)**

**{**

**uint8\_t opcode=0;**

**opcode = (((sock\_num<<2)|BSB\_S0)<<3)|OM\_FDM1;**

**while(1)**

**{**

**if(w5500\_readReg(opcode, Sn\_SR)==SOCK\_LISTEN)**

**{**

**break;**

**}**

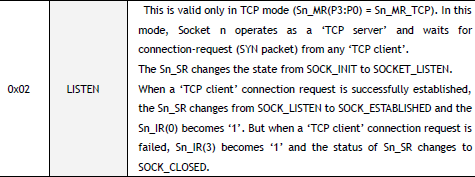
**}**

**}**

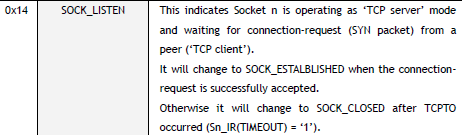
**//-----------------------------------------------**

We also add prototypes for these functions in the header file.

Here we have a similar situation. In the first function, we pass the command 0x02, which translates the socket to the state " **LISTEN** "



And the second function waits for this state to be established by reading the byte of the status register



Call these functions in the initialization function

SocketInitWait(0);

**//Начинаем слушать сокет**

**ListenSocket(0);**

**SocketListenWait(0);**

Just in case, a little wait and see the state of the chip (although this is not necessary). I heard about the cases that the program still waited for the desired state in the function, and then this state was reset by itself. This is what we are controlling

  SocketListenWait(0);

**HAL\_Delay(500);**

**//Посмотрим статусы**

**opcode = (BSB\_S0<<3)|OM\_FDM1;**

**dtt = w5500\_readReg(opcode, Sn\_SR);**

**sprintf(str1,"First Status Sn0: 0x%02X\r\n",dtt);**

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

}

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

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

image28

This initialization can be considered complete.

Now we will need to try to accept a packet from the network.

Below the initialization function, we will create a function for receiving the packet

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

**void w5500\_packetReceive(void)**

**{**

**uint16\_t point;**

**uint16\_t len;**

**}**

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

Create a prototype for this function in the header file and call it in the corresponding function in the file **net.c**

void packet\_receive(void)

{

**w5500\_packetReceive();**

}

In the file **w5500.c** above the initialization function, we add the function of determining the current state of the socket

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

**uint8\_t GetSocketStatus(uint8\_t sock\_num)**

**{**

**uint8\_t dt;**

**uint8\_t opcode=0;**

**opcode = (((sock\_num<<2)|BSB\_S0)<<3)|OM\_FDM1;**

**dt = w5500\_readReg(opcode, Sn\_SR);**

**return dt;**

**}**

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

To find out the state of the socket, we query the socket status register, the number of which we pass in the incoming argument.

Now we will create a condition condition of the current socket in the function of receiving the packet. It must be in the status "Connected"

uint16\_t len;

**if(GetSocketStatus(tcpprop.cur\_sock)==SOCK\_ESTABLISHED)**

**{**

**}**

In the [**next part of the**](http://narodstream.ru/stm-urok-91-lan-w5500-http-server-chast-3/) lesson, we will examine the HTTP request that comes from the client and begin to respond to it.

**Unit 91**

**Part 3**

# LAN. W5500. HTTP Server

In the [**previous part of the**](http://narodstream.ru/stm-urok-91-lan-w5500-http-server-chast-2/) lesson, we wrote the initialization function of the microchip, and also started to write the function of receiving and processing network packets.

Since we are writing an HTTP server, let's put all the work with this protocol into a separate module for which we will create **httpd.h** and **httpd.c files of the** following content

**httpd.h**

**#ifndef HTTPD\_H\_**

**#define HTTPD\_H\_**

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

**#include "stm32f4xx\_hal.h"**

**#include <string.h>**

**#include <stdlib.h>**

**#include <stdint.h>**

**#include "fatfs.h"**

**#include "w5500.h"**

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

**#endif /\* HTTPD\_H\_ \*/**

**httpd.c**

**#include "httpd.h"**

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

**extern UART\_HandleTypeDef huart2;**

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

**extern char str1[60];**

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

Also we will connect this library in file **w5500.h** at the **very bottom of the** file

**void w5500\_packetReceive(void);**

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

**#include "httpd.h"**

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

In the **httpd.h** file, **we** add a structure for the properties of the HTTP package, as well as macros for document types

#include "w5500.h"

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

**typedef struct http\_sock\_prop {**

**volatile uint8\_t data\_stat;//статус передачи данных**

**volatile uint32\_t data\_size;//размер данных для передачи**

**volatile uint16\_t last\_data\_part\_size;//размер последней части данных для передачи**

**volatile uint16\_t cnt\_data\_part;//общее количество частей данных для передачи**

**volatile uint16\_t cnt\_rem\_data\_part;//количество оставшихся частей данных для передачи**

**volatile uint32\_t total\_count\_bytes;//количество переданных байтов документа**

**volatile uint8\_t http\_doc;//вариант документа для передачи**

**volatile uint8\_t prt\_tp;//вариант документа для передачи**

**char fname[20];//имя файла (документа)**

**} http\_sock\_prop\_ptr;**

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

**//Варианты документов HTTP**

**#define EXISTING\_HTML 0**

**#define E404\_HTML 1**

**#define EXISTING\_JPG 2**

**#define EXISTING\_ICO 3**

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

In the **httpd.c** file **,** add the variable of our structure, and also connect the TCP property structure

extern char str1[60];

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

**http\_sock\_prop\_ptr httpsockprop[2];**

**extern tcp\_prop\_ptr tcpprop;**

In the w5500.c file, we also connect the HTTP property structure variable

tcp\_prop\_ptr tcpprop;

**extern http\_sock\_prop\_ptr httpsockprop[2];**

In the file **w5500.h,** add **macros** for data transfer states

} tcp\_prop\_ptr;

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

**//Статусы передачи данных**

**#define DATA\_COMPLETED 0 //передача данных закончена**

**#define DATA\_ONE 1 //передаём единственный пакет**

**#define DATA\_FIRST 2 //передаём первый пакет**

**#define DATA\_MIDDLE 3 //передаём средний пакет**

**#define DATA\_LAST 4 //передаём последний пакет**

**#define DATA\_END 5 //закрываем соединение после передачи данных**

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

In the file  **w5500.c** in the function of receiving a packet in the body of our condition of the socket state, we will add one more condition of the data transfer state

if(GetSocketStatus(tcpprop.cur\_sock)==SOCK\_ESTABLISHED)

{

**if(httpsockprop[tcpprop.cur\_sock].data\_stat == DATA\_COMPLETED)**

**{**

**}**

}

In the body of this condition, we will accept the HTTP request, provided that the data from us is all passed. And, since the macro of this state has the value **0** , we do not need to initialize the state when starting the program and we will immediately enter the body of this condition.

Above the initialization function, we add a new function for determining the size of the received data in the TCP packet, using a special socket register

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

**uint16\_t GetSizeRX(uint8\_t sock\_num)**

**{**

**uint16\_t len;**

**uint8\_t opcode=0;**

**opcode = (((sock\_num<<2)|BSB\_S0)<<3)|OM\_FDM1;**

**len = (w5500\_readReg(opcode,Sn\_RX\_RSR0)<<8|w5500\_readReg(opcode,Sn\_RX\_RSR1));**

**return len;**

**}**

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

Further in the body of the new condition we will know the size of the received data, display it in the terminal program, and if it is zero, that is, there is no data in the TCP packet, then we exit the function

if(httpsockprop[tcpprop.cur\_sock].data\_stat == DATA\_COMPLETED)

{

**//Отобразим размер принятых данных**

**len = GetSizeRX(0);**

**sprintf(str1,"len\_rx\_buf:0x%04Xrn",len);**

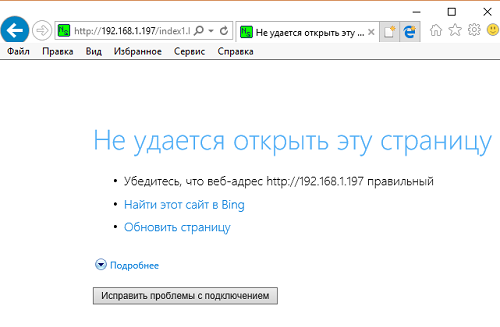
**HAL\_UART\_Transmit(&huart2,(uint8\_t\*)str1,strlen(str1),0x1000);**

**//Если пришел пустой пакет, то уходим из функции**

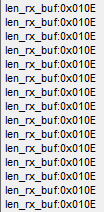
**if(!len) return;**

}

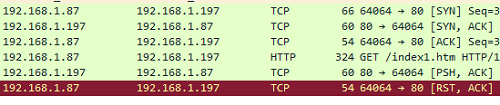
We will collect the code, we will sew the controller and try to request any page. Although, of course, our server has not yet responded, but at least we will see that a package of a certain length has come



In the terminal program, we see endlessly incoming data with the size of the data length, that is, until we process the data, we will fall into this condition. and it is independent, whether or not our connection is closed



We also see in WareShark that the request from the client was only one



And to stop this "infinite loop" is possible only by rebooting the controller.

But at least we see that the request came to us.

Also above the initialization function, we'll write a function that will return the address of the data in the receiving buffer, also using the corresponding socket register

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

**uint16\_t GetReadPointer(uint8\_t sock\_num)**

**{**

**uint16\_t point;**

**uint8\_t opcode;**

**opcode = (((sock\_num<<2)|BSB\_S0)<<3)|OM\_FDM1;**

**point = (w5500\_readReg(opcode,Sn\_RX\_RD0)<<8|w5500\_readReg(opcode,Sn\_RX\_RD1));**

**return point;**

**}**

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

Let's add a prototype for this function. It will be useful to us in another module.

Let's also learn the address of data in the buffer of their reception in the body of the same condition in the function of reading and processing network packets, also displaying it in the terminal program

if(!len) return;

**//здесь обмениваемся информацией: на запрос документа от клиента отправляем ему запрошенный документ**

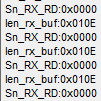
**//указатель на начало чтения приёмного буфера**

**point = GetReadPointer(tcpprop.cur\_sock);**

**sprintf(str1,"Sn\_RX\_RD:0x%04Xrn",point);**

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

We will collect the code, we will write the controller and once again we will request the document in the browser. But the result



We have no address yet. Well, it's understandable. This is the very first request and it is not processed. Therefore, the address will not change yet.

Add a global array for the temporary buffer

extern char str1[60];

**char tmpbuf[30];**

In the **w5500.h** file **,** add the structure for the transfer buffer

} tcp\_prop\_ptr;

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

**typedef struct data\_sect {**

**volatile uint16\_t addr;**

**volatile uint8\_t opcode;**

**uint8\_t data[];**

**} data\_sect\_ptr;**

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

Also, we'll add macros for the TCP protocol options. While there will be only two of them - unknown and HTTP

#define DATA\_END 5 //закрываем соединение после передачи данных

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

**//Варианты протоколов TCP**

**#define PRT\_TCP\_UNCNOWN 0**

**#define PRT\_TCP\_HTTP 1**

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

Back in the file **w5500.c** and after the function of reading the register, add a few more reading functions

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

**void w5500\_readBuf(data\_sect\_ptr \*datasect, uint16\_t len)**

**{**

**SS\_SELECT();**

**HAL\_SPI\_Transmit(&hspi1, (uint8\_t\*) datasect, 3, 0xFFFFFFFF);**

**HAL\_SPI\_Receive(&hspi1, (uint8\_t\*) datasect, len, 0xFFFFFFFF);**

**SS\_DESELECT();**

**}**

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

**uint8\_t w5500\_readSockBufByte(uint8\_t sock\_num, uint16\_t point)**

**{**

**uint8\_t opcode, bt;**

**opcode = (((sock\_num<<2)|BSB\_S0\_RX)<<3)|OM\_FDM1;**

**bt = w5500\_readReg(opcode, point);**

**return bt;**

**}**

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

**void w5500\_readSockBuf(uint8\_t sock\_num, uint16\_t point, uint8\_t \*buf, uint16\_t len)**

**{**

**data\_sect\_ptr \*datasect = (void\*)buf;**

**datasect->opcode = (((sock\_num<<2)|BSB\_S0\_RX)<<3)|OM\_FDM0;**

**datasect->addr = be16toword(point);**

**w5500\_readBuf(datasect,len);**

**}**

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

The first of these functions reads from the buffer using a method of reading variable-length data, the second reads one byte from the buffer of a particular socket, and the third - several bytes from a particular socket.

The second function is  **w5500\_readSockBufByte** - we will add a prototype in the header file.

Now, in the function of reading and processing network packets, we add another code to our condition, which will determine that it is the HTTP request

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

**w5500\_readSockBuf(tcpprop.cur\_sock, point, (uint8\_t\*)tmpbuf, 5);**

**if (strncmp(tmpbuf,"GET /", 5) == 0)**

**{**

**HAL\_UART\_Transmit(&huart2,(uint8\_t\*)"HTTPrn",6,0x1000);**

**httpsockprop[tcpprop.cur\_sock].prt\_tp = PRT\_TCP\_HTTP;**

**}**

When the lines match, we will display the abbreviation "HTTP" in the terminal program, and also initialize the field of the structure with a certain protocol.

In the file **httpd.c** we add the response function to the HTTP request, in which we find the address of the occurrence of the symbol " **/** "

extern tcp\_prop\_ptr tcpprop;

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

**void http\_request(void)**

**{**

**uint16\_t point;**

**uint8\_t RXbyte;**

**uint16\_t i=0;**

**char \*ss1;**

**int ch1='.';**

**// ищем первый "/" в HTTP заголовке**

**point = GetReadPointer(tcpprop.cur\_sock);**

**i = 0;**

**while (RXbyte != (uint8\_t)'/')**

**{**

**RXbyte = w5500\_readSockBufByte(tcpprop.cur\_sock,point+i);**

**i++;**

**}**

**}**

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

Let's create a prototype for this function and call it in the function of reading the package in our condition in the file **w5500.c**

httpsockprop[tcpprop.cur\_sock].prt\_tp = PRT\_TCP\_HTTP;

**http\_request();**

So that in the future we can not get confused in the places where conditions are added, let's get out of the bodies of two conditions right away and add two more conditions in advance

        http\_request();

      }

    }

**else if(httpsockprop[tcpprop.cur\_sock].data\_stat==DATA\_MIDDLE)**

**{**

**if(httpsockprop[tcpprop.cur\_sock].prt\_tp == PRT\_TCP\_HTTP)**

**{**

**}**

**}**

**else if(httpsockprop[tcpprop.cur\_sock].data\_stat==DATA\_LAST)**

**{**

**if(httpsockprop[tcpprop.cur\_sock].prt\_tp == PRT\_TCP\_HTTP)**

**{**

**}**

**}**

  }

}

These conditions will be triggered if one of the transmission statuses is set-the transfer of the middle packet and or the transmission of the last data packet. Similarly, we worked with packages in the lessons via HTTP with the enc28j60 module.

In the file **httpd.c we** connect a temporary buffer

extern char str1[60];

**extern char tmpbuf[30];**

We continue to write the response function to the HTTP request.

We consider the next character after the slash and, if there is a space, then this is the request for the main page. Set the appropriate document type in the structure field

  i++;

}

**point+=i;**

**RXbyte = w5500\_readSockBufByte(tcpprop.cur\_sock,point);**

**if(RXbyte==(uint8\_t)' ')**

**{**

**strcpy(httpsockprop[tcpprop.cur\_sock].fname,"index.htm");**

**httpsockprop[tcpprop.cur\_sock].http\_doc = EXISTING\_HTML;**

**}**

**else**

**{**

**}**

We write the body of the condition that otherwise operates

else

{

**// ищем следующий пробел (" ") в HTTP заголовке, таким образом считывая имя документа из запроса**

**i=0;**

**while (1)**

**{**

**tmpbuf[i] = w5500\_readSockBufByte(tcpprop.cur\_sock, point+i);**

**if(tmpbuf[i] == (uint8\_t)' ') break;**

**i++;**

**}**

**tmpbuf[i] = 0; //закончим строку**

**strcpy(httpsockprop[tcpprop.cur\_sock].fname,tmpbuf);**

}

In this body, we get all the bytes to the next blank. This will be the name of the document.

We continue the function further. Display the name of the file that the client requested in the terminal program

  strcpy(httpsockprop[tcpprop.cur\_sock].fname,tmpbuf);

}

**HAL\_UART\_Transmit(&huart2,(uint8\_t\*)httpsockprop[tcpprop.cur\_sock].fname,strlen(httpsockprop[tcpprop.cur\_sock].fname),0x1000);**

**HAL\_UART\_Transmit(&huart2,(uint8\_t\*)"rn",2,0x1000);**

We will collect the code, we will collate the controller and try to request the main page first, and then some other, then we'll see the result in the terminal

image34  image35

All is normally read.

Now, in the **main.c** file, we add some global variables to work with the file system, and also connect the string array

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

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

**FATFS SDFatFs;//указатель на объект**

**extern char str1[60];**

/\* USER CODE END PV \*/

In the **main ()** function, we will also add a local variable to the result

/\* USER CODE BEGIN 1 \*/

**FRESULT result; //результат выполнения**

/\* USER CODE END 1 \*/

In the same function, we mount our media on the SD card and display the result in the terminal program

/\* USER CODE BEGIN 2 \*/

**result=f\_mount(&SDFatFs,(TCHAR const\*)SDPath,0);**

**sprintf(str1,"f\_mount: %drn",result);**

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

net\_ini();

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

image36

Everything is normally mounted.

Let's return to the **httpd.c** file and add the necessary global variables to work with the file system

extern tcp\_prop\_ptr tcpprop;

**FIL MyFile;**

**FRESULT result; //результат выполнения**

**uint32\_t bytesread;**

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

Let's continue our function of answering the HTTP request.

Open the desired file from the map and find out its size. Also show it in the terminal program

HAL\_UART\_Transmit(&huart2,(uint8\_t\*)"rn",2,0x1000);

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

**result=f\_open(&MyFile,httpsockprop[tcpprop.cur\_sock].fname,FA\_READ); //Попытка открыть файл**

**sprintf(str1,"f\_open: %drn",result);**

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

**sprintf(str1,"f\_size: %lurn",MyFile.fsize);**

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

Before opening the file, we close the previous one. You can do this in other places, but it works best for me.

Again, we will collect the project, we'll run through the controller and try to ask for any existing document

image37

The size is determined normally.

I created several html pages of different sizes to test our project with requesting documents consisting of different number of packages.

We will add several types of HTTP headers for the answer in the form of holographic arrays, as well as the array with the page transmitted in case of the absence of the requested document

uint32\_t bytesread;

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

**const char http\_header[] = { "HTTP/1.1 200 OKrnContent-Type: text/htmlrnrn"};**

**const char jpg\_header[] = {"HTTP/1.0 200 OKrnServer: nginxrnContent-Type: image/jpegrnConnection: closernrn"};**

**const char icon\_header[] = { "HTTP/1.1 200 OKrnContent-Type: image/x-iconrnrn"};**

**const char error\_header[] = {"HTTP/1.0 404 File not foundrnServer: nginxrnContent-Type: text/htmlrnConnection: closernrn"};**

**char \*header;**

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

**const uint8\_t e404\_htm[] = {**

**0x3c,0x68,0x74,0x6d,0x6c,0x3e,0x0a,0x20,0x20,0x3c,0x68,0x65,0x61,0x64,0x3e,0x0a,**

**0x20,0x20,0x20,0x20,0x3c,0x74,0x69,0x74,0x6c,0x65,0x3e,0x34,0x30,0x34,0x20,0x4e,**

**0x6f,0x74,0x20,0x46,0x6f,0x75,0x6e,0x64,0x3c,0x2f,0x74,0x69,0x74,0x6c,0x65,0x3e,**

**0x0a,0x20,0x20,0x3c,0x2f,0x68,0x65,0x61,0x64,0x3e,0x0a,0x3c,0x62,0x6f,0x64,0x79,**

**0x3e,0x0a,0x3c,0x68,0x31,0x20,0x73,0x74,0x79,0x6c,0x65,0x3d,0x22,0x74,0x65,0x78,**

**0x74,0x2d,0x61,0x6c,0x69,0x67,0x6e,0x3a,0x20,0x63,0x65,0x6e,0x74,0x65,0x72,0x3b,**

**0x22,0x3e,0x34,0x30,0x34,0x20,0x45,0x72,0x72,0x6f,0x72,0x20,0x46,0x69,0x6c,0x65,**

**0x20,0x4e,0x6f,0x74,0x20,0x46,0x6f,0x75,0x6e,0x64,0x3c,0x2f,0x68,0x31,0x3e,0x0a,**

**0x3c,0x68,0x32,0x20,0x73,0x74,0x79,0x6c,0x65,0x3d,0x22,0x74,0x65,0x78,0x74,0x2d,**

**0x61,0x6c,0x69,0x67,0x6e,0x3a,0x20,0x63,0x65,0x6e,0x74,0x65,0x72,0x3b,0x22,0x3e,**

**0x20,0x54,0x68,0x65,0x20,0x70,0x61,0x67,0x65,0x20,0x79,0x6f,0x75,0x20,0x61,0x72,**

**0x65,0x20,0x6c,0x6f,0x6f,0x6b,0x69,0x6e,0x67,0x20,0x66,0x6f,0x72,0x20,0x6d,0x69,**

**0x67,0x68,0x74,0x20,0x68,0x61,0x76,0x65,0x20,0x62,0x65,0x65,0x6e,0x20,0x72,0x65,**

**0x6d,0x6f,0x76,0x65,0x64,0x2c,0x20,0x3c,0x62,0x72,0x20,0x2f,0x3e,0x68,0x61,0x64,**

**0x20,0x69,0x74,0x73,0x20,0x6e,0x61,0x6d,0x65,0x20,0x63,0x68,0x61,0x6e,0x67,0x65,**

**0x64,0x2c,0x20,0x6f,0x72,0x20,0x69,0x73,0x20,0x74,0x65,0x6d,0x70,0x6f,0x72,0x61,**

**0x72,0x69,0x6c,0x79,0x20,0x75,0x6e,0x61,0x76,0x61,0x69,0x6c,0x61,0x62,0x6c,0x65,**

**0x2e,0x3c,0x2f,0x68,0x32,0x3e,0x0a,0x3c,0x2f,0x62,0x6f,0x64,0x79,0x3e,0x3c,0x2f,**

**0x68,0x74,0x6d,0x6c,0x3e};**

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

We continue to write our response function to the HTTP request. If the document is opened, then we will study its extension and expose the required status, and if not, we will establish the document's existence error status on the server. Also, we will calculate the length of the HTTP response in advance in order to determine later how many packets and windows it will fit in

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

**if (result==FR\_OK)**

**{**

**//изучим расширение файла**

**ss1 = strchr(httpsockprop[tcpprop.cur\_sock].fname,ch1);**

**ss1++;**

**if (strncmp(ss1,"jpg", 3) == 0)**

**{**

**httpsockprop[tcpprop.cur\_sock].http\_doc = EXISTING\_JPG;**

**//сначала включаем в размер размер заголовка**

**httpsockprop[tcpprop.cur\_sock].data\_size = strlen(jpg\_header);**

**}**

**if (strncmp(ss1,"ico", 3) == 0)**

**{**

**httpsockprop[tcpprop.cur\_sock].http\_doc = EXISTING\_ICO;**

**//сначала включаем в размер размер заголовка**

**httpsockprop[tcpprop.cur\_sock].data\_size = strlen(icon\_header);**

**}**

**else**

**{**

**httpsockprop[tcpprop.cur\_sock].http\_doc = EXISTING\_HTML;**

**//сначала включаем в размер размер заголовка**

**httpsockprop[tcpprop.cur\_sock].data\_size = strlen(http\_header);**

**}**

**//затем размер самого документа**

**httpsockprop[tcpprop.cur\_sock].data\_size += MyFile.fsize;**

**}**

**else**

**{**

**httpsockprop[tcpprop.cur\_sock].http\_doc = E404\_HTML;**

**//сначала включаем в размер размер заголовка**

**httpsockprop[tcpprop.cur\_sock].data\_size = strlen(error\_header);**

**//затем размер самого документа**

**httpsockprop[tcpprop.cur\_sock].data\_size += sizeof(e404\_htm);**

**}**

In the [**next part of the**](http://narodstream.ru/stm-urok-91-lan-w5500-http-server-chast-4/) lesson, we will continue the function of answering the HTTP request, and also write the function of passing the document to the client, which fits together with the header into one window.

**Unit 91**

**Part 4**

# LAN. W5500. HTTP Server

In the [**previous part of the**](http://narodstream.ru/stm-urok-91-lan-w5500-http-server-chast-3/) lesson, we examined the HTTP request that came from the client and began to form an answer to it.

Add a global variable for the window size

uint32\_t bytesread;

**volatile uint16\_t tcp\_size\_wnd = 2048;**

The size of the window for the socket is the size of its buffer. But as long as we do not touch the sizes of the socket buffers and they all by default make up 2 kilobytes, then it's enough for the window size of one variable.

And the maximum size of the segment by default, we also have a certain, in my opinion 1460, it can also be changed at any time. For this, there is a special register in the chip.

We continue the function of the response to the HTTP request. Count the number of windows (the number of segments we do not need to count, the segments of the chip divides the buffer itself without our participation).

  httpsockprop[tcpprop.cur\_sock].data\_size += sizeof(e404\_htm);

}

**httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part = httpsockprop[tcpprop.cur\_sock].data\_size / tcp\_size\_wnd + 1;**

**httpsockprop[tcpprop.cur\_sock].last\_data\_part\_size = httpsockprop[tcpprop.cur\_sock].data\_size % tcp\_size\_wnd;**

**//борьба с неправильным расчётом, когда общий размер делится на минимальный размер окна без остатка**

**if(httpsockprop[tcpprop.cur\_sock].last\_data\_part\_size==0)**

**{**

**httpsockprop[tcpprop.cur\_sock].last\_data\_part\_size=tcp\_size\_wnd;**

**httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part--;**

**}**

**httpsockprop[tcpprop.cur\_sock].cnt\_data\_part = httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part;**

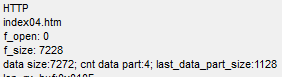
**sprintf(str1,"data size:%lu; cnt data part:%u; last\_data\_part\_size:%urn",**

**(unsigned long)httpsockprop[tcpprop.cur\_sock].data\_size, httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part, httpsockprop[tcpprop.cur\_sock].last\_data\_part\_size);**

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

The entire calculation is similar to the calculation that we used when working with the previous module, so in the explanation he, I think does not need.

We will collect the code, we will tell the controller and ask for more documentation



Everything is considered. Excellent! To check, I think, the calculation is not necessary. Since I have done this many times before giving a lesson. If something goes wrong, then we will see it in the form of a distorted document in the browser.

Depending on the result, we set the status of data transfer

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

**if (httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part==1)**

**{**

**httpsockprop[tcpprop.cur\_sock].data\_stat = DATA\_ONE;**

**}**

**else if (httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part>1)**

**{**

**httpsockprop[tcpprop.cur\_sock].data\_stat = DATA\_FIRST;**

**}**

Here everything is simple. If we have an HTTP response that includes a document along with the header, it climbs into one window (buffer), so this is one status. And if not - then another.

And now, based on these statuses, we will call certain functions. While we only add conditions

  httpsockprop[tcpprop.cur\_sock].data\_stat = DATA\_FIRST;

}

**if(httpsockprop[tcpprop.cur\_sock].data\_stat==DATA\_ONE)**

**{**

**}**

**else if(httpsockprop[tcpprop.cur\_sock].data\_stat==DATA\_FIRST)**

**{**

**}**

We add above the function of sending a document that fits into a single package along with the title

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

**void tcp\_send\_http\_one(void)**

**{**

**uint16\_t i=0;**

**uint16\_t data\_len=0;**

**uint16\_t header\_len=0;**

**uint16\_t end\_point;**

**uint8\_t num\_sect=0;**

**uint16\_t len\_sect;**

**}**

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

In the **w5500.c** file **,** after the SocketListenWait function, add a couple more functions

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

**void SocketClosedWait(uint8\_t sock\_num)**

**{**

**uint8\_t opcode=0;**

**opcode = (((sock\_num<<2)|BSB\_S0)<<3)|OM\_FDM1;**

**while(1)**

**{**

**if(w5500\_readReg(opcode, Sn\_SR)==SOCK\_CLOSED)**

**{**

**break;**

**}**

**}**

**}**

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

**void DisconnectSocket(uint8\_t sock\_num)**

**{**

**uint8\_t opcode=0;**

**opcode = (((sock\_num<<2)|BSB\_S0)<<3)|OM\_FDM1;**

**w5500\_writeReg(opcode, Sn\_CR, 0x08); //DISCON**

**}**

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

The first function waits for the socket to close, and the second one - gives the command to disconnect.

We will add prototypes to these functions in the header file, go back to the **httpd.c** file and process the transfer condition of the document along with the header consisting of a single window

if(httpsockprop[tcpprop.cur\_sock].data\_stat==DATA\_ONE)

{

**tcp\_send\_http\_one();**

**DisconnectSocket(tcpprop.cur\_sock); //Разъединяемся**

**SocketClosedWait(tcpprop.cur\_sock);**

**OpenSocket(tcpprop.cur\_sock,Mode\_TCP);**

**//Ждём инициализации сокета (статус SOCK\_INIT)**

**SocketInitWait(tcpprop.cur\_sock);**

**//Продолжаем слушать сокет**

**ListenSocket(tcpprop.cur\_sock);**

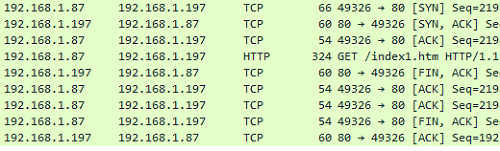
**SocketListenWait(tcpprop.cur\_sock);**

}

Although the function of sending the document is still empty, but, nevertheless, we will collect the code, we'll tell the controller and see what will change. We will of course request a document that will fit into one package

The result in our terminal program is similar to the previous one, only the difference is that it is output only once, that is, the code does not fixate us anymore.

Also see the result in the traffic analyzer



At the request of the document, we simply close the connection correctly. This is already good.

Now we continue to write the function **tcp\_send\_http\_one** .

Depending on the document type, set the HTTP header to the desired array. And all this we will do when fulfilling the condition of the existence of the document

uint16\_t len\_sect;  
**if ((httpsockprop[tcpprop.cur\_sock].http\_doc==EXISTING\_HTML)||**

**(httpsockprop[tcpprop.cur\_sock].http\_doc==EXISTING\_JPG)||**

**(httpsockprop[tcpprop.cur\_sock].http\_doc==EXISTING\_ICO))**

**{**

**switch(httpsockprop[tcpprop.cur\_sock].http\_doc)**

**{**

**case EXISTING\_HTML:**

**header = (void\*)http\_header;**

**break;**

**case EXISTING\_ICO:**

**header = (void\*)icon\_header;**

**break;**

**case EXISTING\_JPG:**

**header = (void\*)jpg\_header;**

**break;**

**}**

**}**

**else**

**{**

**}**

In the file **w5500.c** over the initialization function, we add two more functions

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

**uint16\_t GetWritePointer(uint8\_t sock\_num)**

**{**

**uint16\_t point;**

**uint8\_t opcode;**

**opcode = (((sock\_num<<2)|BSB\_S0)<<3)|OM\_FDM1;**

**point = (w5500\_readReg(opcode,Sn\_TX\_WR0)<<8|w5500\_readReg(opcode,Sn\_TX\_WR1));**

**return point;**

**}**

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

**void SetWritePointer(uint8\_t sock\_num, uint16\_t point)**

**{**

**uint8\_t opcode;**

**opcode = (((sock\_num<<2)|BSB\_S0)<<3)|OM\_FDM1;**

**w5500\_writeReg(opcode, Sn\_TX\_WR0, point>>8);**

**w5500\_writeReg(opcode, Sn\_TX\_WR1, (uint8\_t)point);**

**}**

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

The first function will return the data start address for writing to the send buffer, and the second will set it.

We add prototypes to these functions and continue writing the function  **tcp\_send\_http\_one** in the file **httpd.c** , continuing the body of our condition

  break;

}

**header\_len = strlen(header);**

**data\_len = (uint16\_t)MyFile.fsize;**

**end\_point = GetWritePointer(tcpprop.cur\_sock);**

**end\_point+=header\_len+data\_len;**

We in this code know the size of the header, data, and the address of the beginning of the record. Then we set the address with the added data size and header and read it again.

Although I checked that the previous address is being read, without adding the data size and header, but nevertheless, it does not work differently.

I also want to note that all buffers are ring-type and if the size ends suddenly at the gray address, the data is written and read from zero. No intervention on our part in this process is required, the chip will do everything on its own.

In the file **w5500.c** add another global array for the sector, which we will then bind to a specific structure

char tmpbuf[30];

**uint8\_t sect[515];**

Also in this file, after the **w5500\_writeReg** function **,** add two more functions

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

**void w5500\_writeBuf(data\_sect\_ptr \*datasect, uint16\_t len)**

**{**

**SS\_SELECT();**

**HAL\_SPI\_Transmit(&hspi1, (uint8\_t\*) datasect, len, 0xFFFFFFFF);**

**SS\_DESELECT();**

**}**

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

**void w5500\_writeSockBuf(uint8\_t sock\_num, uint16\_t point, uint8\_t \*buf, uint16\_t len)**

**{**

**data\_sect\_ptr \*datasect = (void\*)buf;**

**datasect->opcode = (((sock\_num<<2)|BSB\_S0\_TX)<<3)|(RWB\_WRITE<<2)|OM\_FDM0;**

**datasect->addr = be16toword(point);**

**w5500\_writeBuf(datasect,len+3);//3 служебных байта**

**}**

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

One of these functions will write variable length data into the buffer, and the other will do the same only with binding to a specific socket.

Moreover, we bind the buffer to the **datasect** structure , the first 2 bytes of which contain the address of the memory (register), with the high byte first, and then the younger one, and the third byte is the opcode. Therefore, we must prepare all this in advance.

In the second function, we do this. Bytes in the address we interchange with the macro **be16toword** . And we prepare the data in the buffer before calling this function, and for convenience immediately from the address of the fourth byte.

Let's create a prototype for the second function in the header file, go back to the **httpd.c** file and connect our buffer and there

extern char tmpbuf[30];

**extern uint8\_t sect[515];**

Let's return to our function, fill the buffer with header data, write it to the send buffer and move the pointer

end\_point+=header\_len+data\_len;

**//Заполним данными буфер для отправки пакета**

**SetWritePointer(tcpprop.cur\_sock, end\_point);**

**end\_point = GetWritePointer(tcpprop.cur\_sock);**

**memcpy(sect+3,header,header\_len);**

**w5500\_writeSockBuf(tcpprop.cur\_sock, end\_point, (uint8\_t\*)sect, header\_len);**

**end\_point+=header\_len;**

Here we just fill the data with the buffer from the address of the fourth byte.

Then we calculate the number of sectors of 512 bytes, add a loop in which we read them from the SD card and write it to the buffer for transmission

  end\_point+=header\_len;

**num\_sect = data\_len / 512;**

**for(i=0;i<=num\_sect;i++)**

**{**

**//не последний сектор**

**if(i<(num\_sect-1)) len\_sect=512;**

**else len\_sect=data\_len;**

**result=f\_lseek(&MyFile,i\*512); //Установим курсор чтения в файле**

**sprintf(str1,"f\_lseek: %drn",result);**

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

**result=f\_read(&MyFile,sect+3,len\_sect,(UINT \*)&bytesread);**

**w5500\_writeSockBuf(tcpprop.cur\_sock, end\_point, (uint8\_t\*)sect, len\_sect);**

**end\_point+=len\_sect;**

**data\_len -= len\_sect;**

**}**

}

else

Let's **go** to file **w5500.c** and add there two more functions for sending the package after the function **GetSocketStatus**

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

**void RecvSocket(uint8\_t sock\_num)**

**{**

**uint8\_t opcode;**

**opcode = (((sock\_num<<2)|BSB\_S0)<<3)|OM\_FDM1;**

**w5500\_writeReg(opcode, Sn\_CR, 0x40); //RECV SOCKET**

**}**

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

**void SendSocket(uint8\_t sock\_num)**

**{**

**uint8\_t opcode;**

**opcode = (((sock\_num<<2)|BSB\_S0)<<3)|OM\_FDM1;**

**w5500\_writeReg(opcode, Sn\_CR, 0x20); //SEND SOCKET**

**}**

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

The first function prepares the sending by moving all the buffer pointers, and the second directly sends the buffer to the network device with which the connection was created.

We will create prototypes for these functions and return to our unfinished function in the file **httpd.c** .

While the opposite part of our condition, in which we will send a response with an error, we skip and at the end of the function we send our buffer

  else

  {

  }

**//отправим данные**

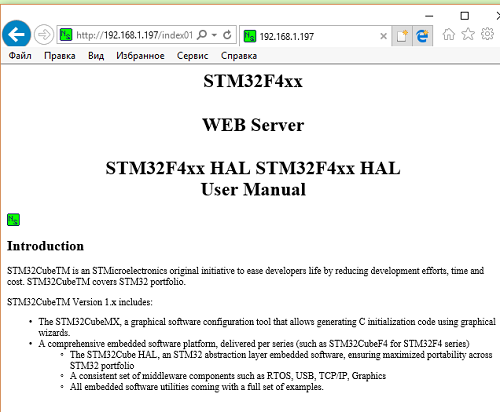
**RecvSocket(tcpprop.cur\_sock);**

**SendSocket(tcpprop.cur\_sock);**

**httpsockprop[tcpprop.cur\_sock].data\_stat = DATA\_COMPLETED;**

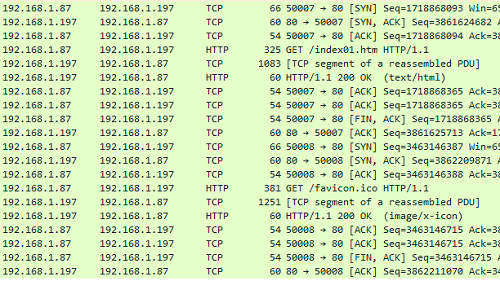
}

We will collect the code, we will collate the controller, then we will try to request first a document that fits not only in one buffer (window), but even in one segment



We see that the document was normally transmitted. An icon was also transferred, which I linked to the document.

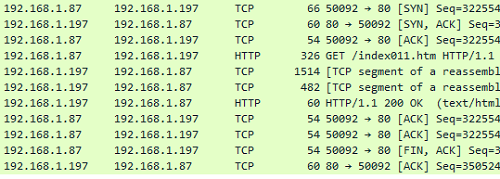
Let's see the traffic analyzer



Here, too, everything is fine.

Now we will request a page that will fit into one window (buffer), but not into the segment.

It was also transmitted, I think there's no point in showing it. Let's see just the output in WireShark



We see that here, too, the process is right. Everything passed and broke into segments. All confirmations have been received. The connection is created and terminated on time.

Let's pass now in a nasty case and we will do the same with an array of a heading for an error and an array of the transferred document of page with an error

data\_len -= len\_sect;

}

}

else

{

**header\_len = strlen(error\_header);**

**data\_len = sizeof(e404\_htm);**

**end\_point = GetWritePointer(tcpprop.cur\_sock);**

**end\_point+=header\_len+data\_len;**

**SetWritePointer(tcpprop.cur\_sock, end\_point);**

**end\_point = GetWritePointer(tcpprop.cur\_sock);**

**//Заполним данными буфер для отправки пакета**

**memcpy(sect+3,error\_header,header\_len);**

**w5500\_writeSockBuf(tcpprop.cur\_sock, end\_point, (uint8\_t\*)sect, header\_len);**

**end\_point+=header\_len;**

**memcpy(sect+3,e404\_htm,data\_len);**

**w5500\_writeSockBuf(tcpprop.cur\_sock, end\_point, (uint8\_t\*)sect, data\_len);**

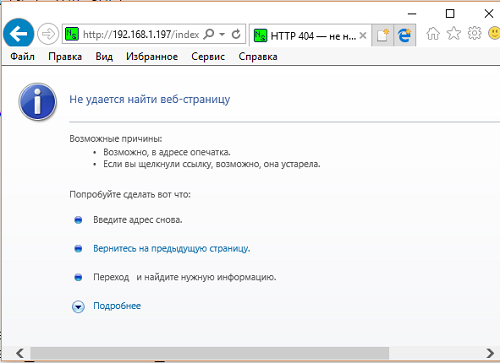
**end\_point+=data\_len;**

}

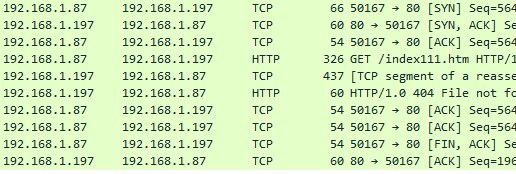
//отправим данные

The code here is similar to the code of a true body and I do not think there is anything to explain here.

We will collect the code, we will tell the controller and try to request a nonexistent document



The page with the error and its title are transferred. We will also verify this with WireShark



In the [**next part of the**](http://narodstream.ru/stm-urok-91-lan-w5500-http-server-chast-5/) lesson, we will write three more functions for the transfer of a large document and thus complete the writing of our server code.

**Unit 91**

**Part 5**

# LAN. W5500. HTTP Server

In the [**previous part of the**](http://narodstream.ru/stm-urok-91-lan-w5500-http-server-chast-4/) lesson, we continued the function of answering the HTTP request, and also wrote the function of passing a document to the client, which fit together with the header into one window.

Following this function, we will create a function to transfer the first buffer of a multi-windowed (multi-buffer) document

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

**void tcp\_send\_http\_first(void)**

**{**

**uint8\_t prt;**

**uint16\_t i=0;**

**uint16\_t data\_len=0;**

**uint16\_t header\_len=0;**

**uint16\_t end\_point;**

**uint8\_t num\_sect=0;**

**uint16\_t len\_sect;**

**uint16\_t last\_part, last\_part\_size;**

**}**

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

Call it in the proper place in the function **http\_request**

  else if(httpsockprop[tcpprop.cur\_sock].data\_stat==DATA\_FIRST)

  {

**tcp\_send\_http\_first();**

  }

}

Moreover, unlike the first function call, we do not close the connection here, since this is only the first part of our **HTTP** response .

We continue to write the function of sending the first part of our answer.

As in the previous function, we will bind the header buffer to a certain array by the state of the document type, determine its length, and also enter into the variable data length, which will be equal to the window size

uint16\_t last\_part, last\_part\_size;

**//На EXISTING проверять не будем, так как будем считать, что error404\_htm у нас всегда будет умещаться в один буфер**

**switch(httpsockprop[tcpprop.cur\_sock].http\_doc)**

**{**

**case EXISTING\_HTML:**

**header = (void\*)http\_header;**

**break;**

**case EXISTING\_ICO:**

**header = (void\*)icon\_header;**

**break;**

**case EXISTING\_JPG:**

**header = (void\*)jpg\_header;**

**break;**

**}**

**header\_len = strlen(header);**

**data\_len = tcp\_size\_wnd-header\_len;**

As indicated in the commentary, we will assume that if the document does not exist, then the page with the error and the corresponding header was already sent to the previous function, since we easily fit this answer into one buffer.

Define and move the pointer in the send buffer

data\_len = tcp\_size\_wnd-header\_len;

**end\_point = GetWritePointer(tcpprop.cur\_sock);**

**end\_point+=header\_len+data\_len;**

**SetWritePointer(tcpprop.cur\_sock, end\_point);**

**end\_point = GetWritePointer(tcpprop.cur\_sock);**

Next, we save some parameters of the structure, since after the further code they are lost due to unknown reasons. That is, they acquire random values

end\_point = GetWritePointer(tcpprop.cur\_sock);

**//сохраним некоторые параметры, а то почему-то они теряются**

**last\_part = httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part;**

**last\_part\_size = httpsockprop[tcpprop.cur\_sock].last\_data\_part\_size;**

**prt = httpsockprop[tcpprop.cur\_sock].prt\_tp;**

Of course, instead of the structure in the future, we will pass these parameters somewhere in the functions, and there is already entered into the static variables, say they are less so lost. But while we temporarily solve this problem in this way. Such a decision also has the right to life.

Just like in the previous function, fill the buffer with header data, write it to the send buffer and move the pointer

prt = httpsockprop[tcpprop.cur\_sock].prt\_tp;

**//Заполним данными буфер для отправки пакета**

**memcpy(sect+3,header,header\_len);**

**w5500\_writeSockBuf(tcpprop.cur\_sock, end\_point, (uint8\_t\*)sect, header\_len);**

**end\_point+=header\_len;**

**num\_sect = data\_len / 512;**

Similarly to the previous function, we, we count the number of sectors of 512 bytes, add a loop in which to read them from the SD card, and is written in the buffer for transmission

num\_sect = data\_len / 512;

**for(i=0;i<=num\_sect;i++)**

**{**

**//не последний сектор**

**if(i<(num\_sect-1)) len\_sect=512;**

**else len\_sect=data\_len;**

**result=f\_lseek(&MyFile,i\*512); //Установим курсор чтения в файле**

**sprintf(str1,"f\_lseek: %drn",result);**

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

**result=f\_read(&MyFile,sect+3,len\_sect,(UINT \*)&bytesread);**

**w5500\_writeSockBuf(tcpprop.cur\_sock, end\_point, (uint8\_t\*)sect, len\_sect);**

**end\_point+=len\_sect;**

**data\_len -= len\_sect;**

**}**

Now we return the values ​​of the fields of the structure that we saved above

  data\_len -= len\_sect;

}

**//вернем параметры**

**httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part = last\_part;**

**httpsockprop[tcpprop.cur\_sock].last\_data\_part\_size = last\_part\_size;**

**httpsockprop[tcpprop.cur\_sock].prt\_tp = prt;**

Send the data to the client

httpsockprop[tcpprop.cur\_sock].prt\_tp = prt;

**//отправим данные**

**RecvSocket(tcpprop.cur\_sock);**

**SendSocket(tcpprop.cur\_sock);**

Since we have transferred one part of the data, we will reduce the number of remaining parts for transmission

SendSocket(tcpprop.cur\_sock);

**//будем считать, что одну часть отправили, поэтому количество оставшихся частей декрементируем**

**httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part--;**

Next, we, depending on the number of remaining parts, determine what will be the next part - the middle or the last

httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part--;

**if(httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part>1)**

**{**

**httpsockprop[tcpprop.cur\_sock].data\_stat=DATA\_MIDDLE;**

**}**

**else**

**{**

**httpsockprop[tcpprop.cur\_sock].data\_stat=DATA\_LAST;**

**}**

And at the end of the function, we calculate the total number of bytes transferred. It will be necessary for the following functions to set the pointer in the readable file to the correct place

    httpsockprop[tcpprop.cur\_sock].data\_stat=DATA\_LAST;

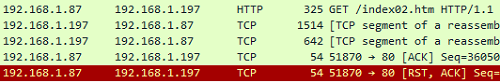
  }

**//Количество переданных байтов**

**httpsockprop[tcpprop.cur\_sock].total\_count\_bytes = tcp\_size\_wnd - header\_len;**

}

We'll collect the code, we'll sew the controller and ask for a larger page in the browser, though, most likely he will not show it to us, but we have WireShark, in which all the moves are written, as they say. And there we'll see if the packets left us



The data has been transmitted, the confirmation has been received. Excellent!

Add immediately the workpiece for the transfer functions of the middle and the last part after the function we just wrote

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

**void tcp\_send\_http\_middle(void)**

**{**

**uint16\_t i=0;**

**uint16\_t data\_len=0;**

**uint16\_t end\_point;**

**uint8\_t num\_sect=0;**

**uint8\_t prt;**

**uint16\_t len\_sect;**

**uint16\_t last\_part, last\_part\_size;**

**uint32\_t count\_bytes;**

**}**

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

**void tcp\_send\_http\_last(void)**

**{**

**uint16\_t i=0;**

**uint16\_t data\_len=0;**

**uint16\_t end\_point;**

**uint8\_t num\_sect=0;**

**uint16\_t len\_sect;**

**}**

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

We will not only call the data of the function from the function of the response to the request, but from the function of receiving and processing the TCP packet **w5500\_packetReceive** . If you remember, we specially prepared there for this condition. Only to call our functions there, it is necessary that they can be seen there. To do this, add the prototypes to them and then go to the function **w5500\_packetReceive** , finding it in the file **w5500.c** , and call our functions

    else if(httpsockprop[tcpprop.cur\_sock].data\_stat==DATA\_MIDDLE)

    {

      if(httpsockprop[tcpprop.cur\_sock].prt\_tp == PRT\_TCP\_HTTP)

      {

**tcp\_send\_http\_middle();**

      }

    }

    else if(httpsockprop[tcpprop.cur\_sock].data\_stat==DATA\_LAST)

    {

      if(httpsockprop[tcpprop.cur\_sock].prt\_tp == PRT\_TCP\_HTTP)

      {

**tcp\_send\_http\_last();**

**DisconnectSocket(tcpprop.cur\_sock); //Разъединяемся**

**SocketClosedWait(tcpprop.cur\_sock);**

**OpenSocket(tcpprop.cur\_sock,Mode\_TCP);**

**//Ждём инициализации сокета (статус SOCK\_INIT)**

**SocketInitWait(tcpprop.cur\_sock);**

**//Продолжаем слушать сокет**

**ListenSocket(tcpprop.cur\_sock);**

**SocketListenWait(tcpprop.cur\_sock);**

      }

    }

  }

}

Well, of course, after the last part of the document is transferred, we disconnect from the client and continue listening to the socket.

We return to the file **httpd.c** and first start writing the body of the transfer function of the last part of the document. Since we will first try to transfer a document that fits into two parts, and then is more voluminous.

We enter into the variable length of the data, define the data transfer address and move it

uint16\_t len\_sect;

**data\_len = httpsockprop[tcpprop.cur\_sock].last\_data\_part\_size;**

**end\_point = GetWritePointer(tcpprop.cur\_sock);**

**end\_point+=data\_len;**

**SetWritePointer(tcpprop.cur\_sock, end\_point);**

**end\_point = GetWritePointer(tcpprop.cur\_sock);**

Count the number of sectors of 512 bytes, add a loop in which we read them from the SD card and write it to the buffer for transmission

end\_point = GetWritePointer(tcpprop.cur\_sock);

**//Заполним данными буфер для отправки пакета**

**num\_sect = data\_len / 512;**

**//борьба с неправильным расчётом, когда размер данных делится на размер сектора без остатка**

**if(data\_len%512==0) num\_sect--;**

**for(i=0;i<=num\_sect;i++)**

**{**

**//не последний сектор**

**if(i<(num\_sect-1)) len\_sect=512;**

**else len\_sect=data\_len;**

**result=f\_lseek(&MyFile, (DWORD)(i\*512) + httpsockprop[tcpprop.cur\_sock].total\_count\_bytes); //Установим курсор чтения в файле**

**sprintf(str1,"f\_lseek: %drn",result);**

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

**result=f\_read(&MyFile,sect+3,len\_sect,(UINT \*)&bytesread);**

**w5500\_writeSockBuf(tcpprop.cur\_sock, end\_point, (uint8\_t\*)sect, len\_sect);**

**end\_point+=len\_sect;**

**data\_len -= len\_sect;**

**}**

As can be seen from the comment text, there can be a situation where we have a size divided by the number of sectors without a remainder. It can turn out to be a wrong calculation. And we corrected this here.

Send our data and establish an appropriate status

  data\_len -= len\_sect;

}

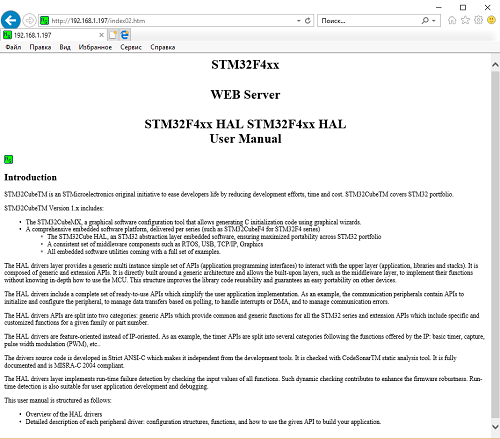
**//отправим данные**

**RecvSocket(tcpprop.cur\_sock);**

**SendSocket(tcpprop.cur\_sock);**

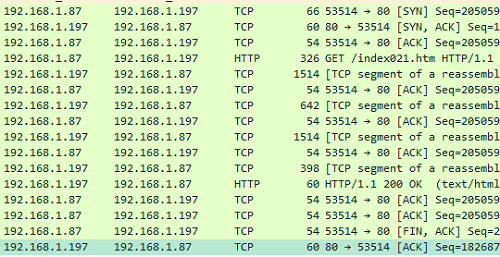
**httpsockprop[tcpprop.cur\_sock].data\_stat = DATA\_COMPLETED;**

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



The page was completely transferred. In this case, we have the second buffer fit into one package.

Also try to transfer a page in which the second part no longer fit into one buffer. Screenshot browser I will not show, because the data is getting bigger and bigger, so you can trust me on the word, I'll show a screenshot of the traffic analyzer



Everything is normally transmitted and confirmed. The connection is opened and closed correctly.

Now we will start writing the body of the function of passing the middle part of the document.

It's a bit simpler here. We no longer pass the title, since we have already transmitted it in the last part.

While we determine the data transfer address and move it

uint32\_t count\_bytes;

**data\_len = tcp\_size\_wnd;**

**end\_point = GetWritePointer(tcpprop.cur\_sock);**

**end\_point+=data\_len;**

**SetWritePointer(tcpprop.cur\_sock, end\_point);**

**end\_point = GetWritePointer(tcpprop.cur\_sock);**

We again retain some parameters of the structure

end\_point = GetWritePointer(tcpprop.cur\_sock);

**//сохраним некоторые параметры, а то почему-то они теряются**

**last\_part = httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part;**

**last\_part\_size = httpsockprop[tcpprop.cur\_sock].last\_data\_part\_size;**

**count\_bytes = httpsockprop[tcpprop.cur\_sock].total\_count\_bytes;**

**prt = httpsockprop[tcpprop.cur\_sock].prt\_tp;**

We calculate the number of sectors and write them into the transmission buffer, having previously read from the carrier

prt = httpsockprop[tcpprop.cur\_sock].prt\_tp;

**//Заполним данными буфер для отправки пакета**

**num\_sect = data\_len / 512;**

**//борьба с неправильным расчётом, когда размер данных делится на размер сектора без остатка**

**if(data\_len%512==0) num\_sect--;**

**for(i=0;i<=num\_sect;i++)**

**{**

**//не последний сектор**

**if(i<(num\_sect-1)) len\_sect=512;**

**else len\_sect=data\_len;**

**result=f\_lseek(&MyFile,(DWORD)(i\*512) + count\_bytes); //Установим курсор чтения в файле**

**result=f\_read(&MyFile,sect+3,len\_sect,(UINT \*)&bytesread);**

**w5500\_writeSockBuf(tcpprop.cur\_sock, end\_point, (uint8\_t\*)sect, len\_sect);**

**end\_point+=len\_sect;**

**data\_len -= len\_sect;**

**}**

Return the previously saved parameters to the fields of the structure

  data\_len -= len\_sect;

}

**//вернем параметры**

**httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part = last\_part;**

**httpsockprop[tcpprop.cur\_sock].last\_data\_part\_size = last\_part\_size;**

**httpsockprop[tcpprop.cur\_sock].total\_count\_bytes = count\_bytes;**

**httpsockprop[tcpprop.cur\_sock].prt\_tp = prt;**

Send the buffer to the client

httpsockprop[tcpprop.cur\_sock].prt\_tp = prt;

**RecvSocket(tcpprop.cur\_sock);**

**SendSocket(tcpprop.cur\_sock);**

We will reduce the number of remaining parts for the transfer and further, depending on the number of remaining parts, we will determine what will be the next part - the middle or the last

  SendSocket(tcpprop.cur\_sock);

**//будем считать, что одну часть отправили, поэтому количество оставшихся частей декрементируем**

**httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part--;**

**if(httpsockprop[tcpprop.cur\_sock].cnt\_rem\_data\_part>1)**

**{**

**httpsockprop[tcpprop.cur\_sock].data\_stat=DATA\_MIDDLE;**

**}**

**else**

**{**

**httpsockprop[tcpprop.cur\_sock].data\_stat=DATA\_LAST;**

**}**

And at the end of the function, we will add the number of bytes transferred from the document

    httpsockprop[tcpprop.cur\_sock].data\_stat=DATA\_LAST;

  }

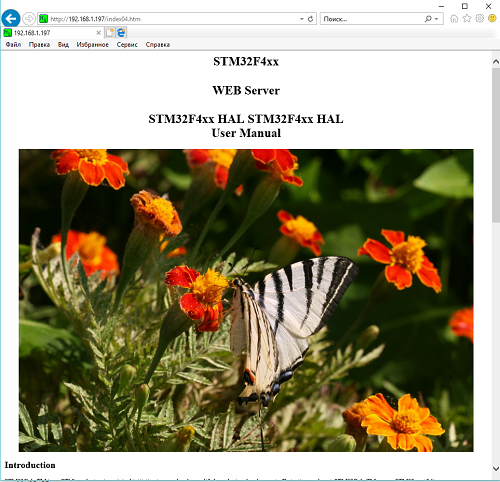
**//Количество переданных байтов**

**httpsockprop[tcpprop.cur\_sock].total\_count\_bytes += (uint32\_t) tcp\_size\_wnd;**

}

Well, like everything.

We will collect the code, we will sew the controller and try to transfer various documents of different sizes, including those including pictures



So, today we have mastered one more interesting microcircuit that provides data transfer over the LAN - **W5500** , which allows you to work with the TCP / IP stack at the hardware level, which will save very much our hardware resources, which, as we all know, very often lacks.

Thank you all for attention!