**Lesson 86**

**Part 1**

**LAN. ENC28J60. TCP WEB Server. Sending a small page**

Today we will continue our topic of data transfer using the TCP transport layer protocol and try to transfer a WEB page using this protocol from our controller. And to send or request such a page, but we already need the **HTTP (HyperText Transfer Protocol) protocol** , which is already an application layer protocol, although in some cases it can act as a transport layer protocol.

The HTTP protocol is already an unstructured protocol, that is, its header does not use the strict order of the fields, nor their size. In general, this protocol is textual, but this does not make it easier, or it can be said, on the contrary - it becomes more difficult.

The order of transmission on this protocol is usually the following.

The client tries to request a connection from the server, the connection is established.

Then the client requests a file from the server (web page, image, archive or any other file), the server first confirms the request, and then sends the requested file to the client. Transmission can occur in several packets or segments.

Then the server waits to confirm the last sent packet from the client and initiates the connection termination.

At the present time, progress is in such a state that for the exchange of the client with the server via HTTP occurs with the opening of several connections, but since we have limited resources, we will not use these technologies and the server will be simplified and we will have a connection for data transfer only one.

And denser we with messages HTTP in request and the answer we will get acquainted in process of data transfer. All the options we will not study, but only those messages that we will use. Full information about the HTTP protocols of all versions is now easy to find in the global network. The latest version at the moment 2.0, but it is not used yet, so to speak, is in the experimental state, but 1.1 is widely used.

Therefore, I will not torture you with a theory and move on to the project.

We will create the project  **ENC28J60\_HTTPS** , reworking it from the project of the [**previous lesson**](http://narodstream.ru/stm-urok-85-lan-enc28j60-tcp-server-peredayom-dannye/) with the name **ENC28J60\_TCPS\_DATA** .

Run the project in the **Cube MX** generator and, without touching anything, generate a project for the Keil programming environment, open it, make the programmer settings for auto-reload, and connect our libraries to the project tree.

Let's change the function of sending Ethernet packets **eth\_send a little** . To do this, open the file **net.c** and add to this function another incoming argument - the type of Ethernet protocol

void eth\_send(enc28j60\_frame\_ptr \*frame,**uint16\_t type,** uint16\_t len)

Also in the body of this function, we will enter the type of protocol in the corresponding header field

memcpy(frame->addr\_src,macaddr,6);

**frame->type=type;**

At the same time, in the same file, we correct the incoming argument of the function of calculating the checksum for the signed

uint16\_t checksum(uint8\_t \*ptr, **int16\_t** len, uint8\_t type)

Also, we correct in the body of this function an error that leads to an incorrect calculation of the checksum in the case of an odd number of bytes. There will be not **0** , but **1**

while(len>**1**)

{

  sum += (uint16\_t) (((uint32\_t)\*ptr<<8)|\*(ptr+1));

Vnesom the same changes in the prototypes of these functions in the header file **net.h** .

Then we will correct the code in the places of the function call **eth\_send**

1. In the same file **net.h**  in function **ip\_send**

eth\_send(frame,**ETH\_IP**,len);

2. In the **arp.c** file in the **arp\_send** function

eth\_send(frame,**ETH\_ARP**,sizeof(arp\_msg\_ptr));

We'll compile our project, we'll patch the controller and check the previous performance of our stack (we'll ping our module, and also send the lines in **Putty** to 80 port.

If everything works, then now we will try to send a request to the server from the PC.

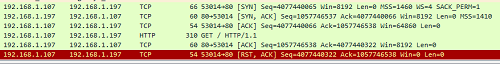
You can even do this with **Putty** , but we'll go in a more mundane way and do it by entering the IP address of our module into the address bar of the browser. As a browser, we will use the good old **Internet Explorer** , since it does not try to open several connections, does not try to request a page icon, that is, in this case behaves quite adequately.

But before we request this page, we will launch the WireShark network traffic analyzer, filtering this traffic, as usual, with the MAC address of our module, and launch the terminal program by connecting to our USART port.

Now enter the IP address of the module in the address bar of the browser and press the "Enter" key or the arrow to the right of the address bar. Our browser does not display any page for obvious reasons, so we will not wait for the page and click on the X, which is also to the right of the address bar, thereby forcing the client to disconnect

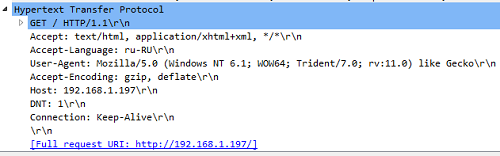
image00

Let's move on to Wireshark and see that the connection to the server was installed on the customer's initiative, then the client sent an HTTP request, which the server confirmed, since this procedure has already been implemented in one way or another. Then the client severed the connection (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/07/image01-8.png)

Therefore, our further task is to determine the correctness of the confirmation from the server, and also after this confirmation to send the main web page - **index.html** . Despite the fact that this does not sound very clever, it will not be very easy to carry out this task. Therefore, let us not hurry and do some preparatory steps.

For starters, let's examine the package that the client sent us as a request for an html-page for general erudition. Since the package was defined as an HTTP package, we'll open the HTTP section in the lower window - Hypertext Transfer Protocol



We see here the following lines of the message

1) **GET / HTTP / 1.1**

Using this message, the client requests ( **GET** method ) the main page (slash with a space) and reports the version of the protocol - 1.1.

That is, first the GET method, then the path to the main file, if there is no name, then the main page - **index.html**or **index.php is** requested , followed by the type and version of the protocol.

2) **Accept: text / html, application / xhtml + xml, \* / \***

The client uses the **Accept** method to tell the server about supported types of data (documents)

3) **Accept-Language: en-US**

Using the **Accept-Language** method, the  client communicates the user's language.

4) **User-Agent: Mozilla / 5.0 (Windows NT 6.1; WOW64; Trident / 7.0; rv: 11.0) like Gecko**

Using this method, the client reports the browser version

5) **Accept-Encoding: gzip, deflate**

This method is used by the client to send to the server a list of supported methods for encoding the content in the transmission /

6) **Host: 192.168.1.193**

The client uses this method to send the host address to the server, since there may be several hosts on one domain and this version is already supported, and this message is mandatory.

7) **DNT: 1**

This is an HTTP method that does not allow the web application to track your actions. If as argument **1** , then it means that the user is not being watched.

8) **Connection: Keep-Alive**

The client tells the server that there is no need to close the connection after each request, that is, the connection can be left open. If the server does not mind, then it must send the same message to the client in return.

Also, in order for the server to understand that the request is terminated, after the last message the client sends the code of the line feed and carriage return twice.

Well, sort of sorted out with the request.

Let's move the code to a file **tcp.c** . We will not create the file pair separately for the HTTP protocol, since the latter is inextricably linked to TCP.

For now, let's take a little optimization of the code in this module, since the function of sending the TCP packet is constantly growing, and this is a mess and will inevitably lead to errors.

Right after all global declarations over this function, we will create a function to prepare the TCP header

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

**//Подготовка заголовка TCP-пакета**

**void tcp\_header\_prepare(tcp\_pkt\_ptr \*tcp\_pkt, uint16\_t port, uint8\_t fl, uint16\_t len)**

**{**

**}**

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

As input arguments, a TCP packet pointer, a destination port, a variable with flags, and a packet length are used here.

Before dealing with the body code of this function, look at the tcp.h header file and add a structure for storing some TCP data (properties)

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

**typedef struct tcp\_prop {**

**volatile uint16\_t port\_dst;//порт получателя**

**volatile uint32\_t seq\_num;//порядковый номер байта**

**volatile uint32\_t ack\_num;//номер подтверждения**

**volatile uint32\_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;//количество оставшихся частей данных для передачи**

**} tcp\_prop\_ptr;**

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

Basically, I wrote comments after the end of variables for each field of texture. And if something is not clear now, then in the process of using the fields of the structure everything will necessarily fall into place.

Now back to the file **tcp.c** and create a global variable of the type of our structure

extern uint8\_t ipaddr[4];

**tcp\_prop\_ptr tcpprop;**

And then fill the body of our new function

void tcp\_header\_prepare(tcp\_pkt\_ptr \*tcp\_pkt, uint16\_t port, uint8\_t fl, uint16\_t len)

{

**tcp\_pkt->port\_dst = be16toword(port);**

**tcp\_pkt->port\_src = be16toword(LOCAL\_PORT\_TCP);**

**tcp\_pkt->bt\_num\_seg = tcpprop.seq\_num;**

**tcp\_pkt->num\_ask = tcpprop.ack\_num;**

**tcp\_pkt->fl = fl;**

**tcp\_pkt->size\_wnd = be16toword(8192);**

**tcp\_pkt->urg\_ptr = 0;**

**tcp\_pkt->len\_hdr = len << 2;**

**tcp\_pkt->cs = 0;**

**tcp\_pkt->cs=checksum((uint8\_t\*)tcp\_pkt-8, len+8, 2);**

}

Here, I think, everything is clear and simple. We fill out all fields of the TCP header, and we have already done this.

After the code of this function, we add a function that will similarly prepare the header of the IP packet, and immediately with the body, because it's still easier there. This is what we have done with you several times.

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

**//Подготовка заголовка IP-пакета**

**void ip\_header\_prepare(ip\_pkt\_ptr \*ip\_pkt, uint8\_t \*ip\_addr, uint8\_t prt, uint16\_t len)**

**{**

**ip\_pkt->len=be16toword(len);**

**ip\_pkt->id = 0;**

**ip\_pkt->ts = 0;**

**ip\_pkt->verlen = 0x45;**

**ip\_pkt->fl\_frg\_of=0;**

**ip\_pkt->ttl=128;**

**ip\_pkt->cs = 0;**

**ip\_pkt->prt=prt;**

**memcpy(ip\_pkt->ipaddr\_dst,ip\_addr,4);**

**memcpy(ip\_pkt->ipaddr\_src,ipaddr,4);**

**ip\_pkt->cs = checksum((void\*)ip\_pkt,sizeof(ip\_pkt\_ptr),0);**

**}**

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

The incoming arguments are a pointer to the IP packet, the IP address value, the protocol type, and the data length.

We go again to the header file **tcp.h** and add there several TCP connection statuses

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

**//Статусы TCP**

**#define TCP\_CONNECTED 1**

**#define TCP\_DISCONNECTED 2**

**#define TCP\_DISCONNECTING 3 //закрываем соединение после подтверждения получателя**

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

Let's return to **tcp.c** and add two global variables

tcp\_prop\_ptr tcpprop;

**volatile uint16\_t tcp\_mss = 458;**

**volatile uint8\_t tcp\_stat = TCP\_DISCONNECTED;**

After the function of preparing the header of the IP packet, we add a function that will send a response to the connection request

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

**/\*Отправка ответа на запрос соединения\*/**

**uint8\_t tcp\_send\_synack(enc28j60\_frame\_ptr \*frame, uint8\_t \*ip\_addr, uint16\_t port)**

**{**

**uint8\_t res=0;**

**uint16\_t len=0;**

**ip\_pkt\_ptr \*ip\_pkt = (void\*)(frame->data);**

**tcp\_pkt\_ptr \*tcp\_pkt = (void\*)(ip\_pkt->data);**

**return res;**

**}**

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

In the body of this function, we will add the necessary variables as well as connect to the packages.

Then fill the fields with byte numbers in the stream

tcp\_pkt\_ptr \*tcp\_pkt = (void\*)(ip\_pkt->data);

**tcpprop.seq\_num = rand();**

**tcpprop.ack\_num = be32todword(be32todword(tcp\_pkt->bt\_num\_seg) + 1);**

Then fill in the options, which indicate the maximum size of the segment

tcpprop.ack\_num = be32todword(be32todword(tcp\_pkt->bt\_num\_seg) + 1);

**tcp\_pkt->data[0]=2;//Maximum Segment Size (2)**

**tcp\_pkt->data[1]=4;//Length**

**tcp\_pkt->data[2]=(uint8\_t) (tcp\_mss>>8);//MSS = 458**

**tcp\_pkt->data[3]=(uint8\_t) tcp\_mss;**

Calculate the length of the TCP packet and call the TCP header preparation function

tcp\_pkt->data[3]=(uint8\_t) tcp\_mss;

**len = sizeof(tcp\_pkt\_ptr)+4;**

**tcp\_header\_prepare(tcp\_pkt, port, TCP\_SYN|TCP\_ACK, len);**

The same is done for the IP header

tcp\_header\_prepare(tcp\_pkt, port, TCP\_SYN|TCP\_ACK, len);

**len+=sizeof(ip\_pkt\_ptr);**

**ip\_header\_prepare(ip\_pkt, ip\_addr, IP\_TCP, len);**

Then prepare and send our package

ip\_header\_prepare(ip\_pkt, ip\_addr, IP\_TCP, len);

**//Заполним заголовок Ethernet**

**memcpy(frame->addr\_dest,frame->addr\_src,6);**

**eth\_send(frame,ETH\_IP,len);**

And at the end of the function, we will write the status of the established connection to a variable

eth\_send(frame,ETH\_IP,len);

**tcp\_stat = TCP\_CONNECTED;**

return res;

Now, in the **tcp\_read** function **,** we first save the receiver port in the appropriate field of the structure with the properties after connecting to the packages

tcp\_pkt\_ptr \*tcp\_pkt = (void\*)(ip\_pkt->data);

**tcpprop.port\_dst = be16toword(tcp\_pkt->port\_src);**

Where we called the **tcp\_send**function with the specified option to establish a connection , we call our new function

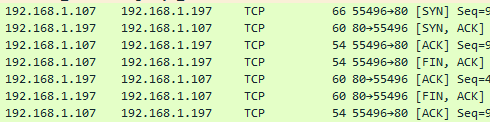
if (tcp\_pkt->fl == TCP\_SYN)

{

**tcp\_send\_synack(frame, ip\_pkt->ipaddr\_src, tcpprop.port\_dst);**

}

We will collect the code, we will tell the controller and try to install and break the connection, thereby making sure that our server is responding to this adequately, as before. You can use the Putty utility without passing any rows



Excellent! We continue further.

Now we need to do the same steps to break the connection.

To do this, above the **tcp\_send** function, add a function that will send a response to the disconnect request and then send the same request under a certain condition. The code for sending the confirmation packet will be filled immediately, and the rest of the code a little later

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

**/\*Отправка ответа на запрос разъединения и затем отправка такого же запроса при условии\*/**

**uint8\_t tcp\_send\_finack(enc28j60\_frame\_ptr \*frame, uint8\_t \*ip\_addr, uint16\_t port)**

**{**

**uint8\_t res=0;**

**uint16\_t len=0;**

**ip\_pkt\_ptr \*ip\_pkt = (void\*)(frame->data);**

**tcp\_pkt\_ptr \*tcp\_pkt = (void\*)(ip\_pkt->data);**

**tcpprop.seq\_num = tcp\_pkt->num\_ask;**

**tcpprop.ack\_num = be32todword(be32todword(tcp\_pkt->bt\_num\_seg) + 1);**

**len = sizeof(tcp\_pkt\_ptr);**

**tcp\_header\_prepare(tcp\_pkt, port, TCP\_ACK, len);**

**len+=sizeof(ip\_pkt\_ptr);**

**ip\_header\_prepare(ip\_pkt, ip\_addr, IP\_TCP, len);**

**//Заполним заголовок Ethernet**

**memcpy(frame->addr\_dest,frame->addr\_src,6);**

**eth\_send(frame,ETH\_IP,len);**

**return res;**

**}**

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

We sent the confirmation packet, and before we send the next packet, we will check the status of our connection, and if the status indicates that our connection is already broken, we will exit the function. That is, we have already sent a request for separation. This is done in the event that we need to complete the connection on the initiative of the server

eth\_send(frame,ETH\_IP,len);

**if(tcp\_stat == TCP\_DISCONNECTED) return 0;**

Then, after refilling only some fields of our headers, we will send a request for disconnection and after posting we set the disconnect status

if(tcp\_stat == TCP\_DISCONNECTED) return 0;

**tcp\_pkt->fl = TCP\_FIN|TCP\_ACK;**

**len = sizeof(tcp\_pkt\_ptr);**

**tcp\_pkt->cs = 0;**

**tcp\_pkt->cs=checksum((uint8\_t\*)tcp\_pkt-8, len+8, 2);**

**len+=sizeof(ip\_pkt\_ptr);**

**eth\_send(frame,ETH\_IP,len);**

**tcp\_stat = TCP\_DISCONNECTED;**

return res;

Here we only change the flags, the byte numbers do not change, we do not recalculate the IP checksum, since it is calculated only for the header bytes, but they did not change.

It is also very important to note that in no case is it possible to use the code for copying the MAC address from the source to the receiver before sending the packet, otherwise the packet will not reach the recipient, since we have already changed them in places in the same function.

Call this function in the **tcp\_read** function in the corresponding two places instead of calling the **tcp\_send**function

else if (tcp\_pkt->fl == (TCP\_FIN|TCP\_ACK))

{

**tcp\_send\_finack(frame, ip\_pkt->ipaddr\_src, tcpprop.port\_dst);**

}

Once again, we will compile the code, we will write the controller and check the operation of our function in practice, repeating the same actions in the Putty program as above.

If everything works fine, now we'll deal with the data transfer.

To do this, we also add a function over the function tcp\_send, which will send a confirmation to the data packet and the answer if the line matches the declared one. Since the code we have practically not changed in comparison with what it was in the condition in the function **tcp\_send** , then we will write it in the body of the function at once

**tcp\_send\_data**

Call this function in the right place in the **tcp\_read**

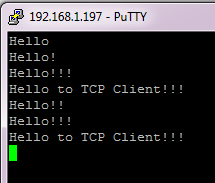
if (tcp\_pkt->fl&TCP\_ACK)

{

**tcp\_send\_data(frame, ip\_pkt->ipaddr\_src, tcpprop.port\_dst);**

}

We will collect the code, we will sew the controller and check the result



If everything is exactly the same in the WireShark utility and everything is green, then you can delete the **tcp\_send** function together with all the contents. Also, you can remove all the operations of the TCP header file **tcp.h** .

In the [**next part of the**](http://narodstream.ru/stm-urok-86-lan-enc28j60-tcp-web-server-peredayom-maluyu-stranicu-chast-2/) lesson we will write the entire set of functions for transferring a short WEB page to the client's browser and see the result of our work in the form of displaying this page.

**Lesson 86**

**Part 2**

# LAN. ENC28J60. TCP WEB Server. Sending a small page

In the [**previous part of the**](http://narodstream.ru/stm-urok-86-lan-enc28j60-tcp-web-server-peredayom-maluyu-stranicu-chast-1/) lesson we briefly introduced the HTTP protocol, analyzed the HTTP request from the client, and also wrote a number of functions for the convenience of working with the code.

Now finally go to work with the HTTP protocol and, after returning to the **tcp.c** file, **we** 'll think about how to add an array for the header of the HTTP response packet, while not "eating" an impressive amount of RAM.

And we will use for this purpose a place in flash memory, in which we have more available storage space.

To do this, we apply the declaration of **const** , declaring the following global array

volatile uint8\_t tcp\_stat = TCP\_DISCONNECTED;

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

**const char http\_header[] = {"HTTP/1.1 200 OKrnServer: nginxrnContent-Type: text/htmlrnConnection: keep-alivernrn"};**

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

After that, this line will be stored in flash memory and from there and will be called if necessary.

Also, I find it necessary to explain what exactly is in the header of the HTTP package that we send to the request.

We only have three lines, I think in order for the client browser to understand that in the header, that's enough.

1)   **HTTP / 1.1 200 OK**

Here the server transmits the version of the supported protocol, the message code and the message itself. 200 means a successful request, that is, the document that the client requests, on the server is present and will be transferred to the client, OK means "good".

The codes for all messages can be found in the official HTTP protocol description documents. I will only say that the codes starting with the number 2 are a group of codes that inform about the cases of successful acceptance and processing of the client's request. If the code starts with 1, it means that the server informs about the transfer process, if with 3 - then this is a redirection and the client will have to make another request, and if 4, then the server thus tells the client about the error. We often see the error message 404 when the address is incorrectly typed - the page was not found. This is one example of a mistake. As a rule, the server stores a page in which there is readable information about the absence of the page, and when the client requests a nonexistent document on the server, then the server sends this page after the transmission of such code.

2)  **Server: nginx**

Server type. In our case, this is not so important and I passed the most frequently used type.

3)  **Connection: keep-alive**

With this message we are already familiar with the header of the client's request and here we give our consent to the fact that if we do not close the connection after each request of the client.

Now, in the tcp\_read function, where we call the data transfer function, we will change the code slightly, in which we will determine which data we have - ordinary or HTTP request

if (tcp\_pkt->fl&TCP\_ACK)

{

**//Если строка "GET / ", то значит это запрос HTTP главной страницы, пока будем работать только с одной**

**if (strncmp((char\*)tcp\_pkt->data,"GET / ", 6) == 0)**

**{**

**}**

**//Иначе обычные данные**

**else**

**{**

tcp\_send\_data(frame, ip\_pkt->ipaddr\_src, tcpprop.port\_dst);

**}**

}

Until the condition is left with the empty body, since further in this body we need to write a code that will determine the size of our HTTP package, which we will send with the purpose to find out whether this package will fit into one TCP packet or there will be several of them, and if several, then how many and exactly how many bytes in the last packet. And, as is known in the HTTP package, in addition to the header, there are also data, which in our case will be the main page, which we have not yet prepared. We will also place its contents in FLASH memory, but not in text form, since the text can have different characters, including those that the compiler does not really like, but in the form of an array of 8-bit integers that we will create using **makefsdata.exe**, which is easy to find on the Internet and which comes in the form of a single executable file, this utility also requires installed **Microsoft Visual C ++ Redistributable libraries** . If the package is not installed, the utility will request the missing library. Well, I think it's not a problem for us.

We will place the file makefsdata.exe in a separate folder, we will also create another folder with the name " **fs** " in this folder and put the desired page **index.html** in the created folder , for example with such content

**<html><body><h1 style="text-align: center;">STM32F103x8<br><br>WEB Server</h1>**

**<p></p>**

**<h2>Features</h2>**

**<p>ARM® 32-bit Cortex®-M3 CPU Core</p>**

**</body></html>**

Run the utility and we will need to create a file called **fsdata.c** .

Before opening it, we will prepare a place where we will put the data of the created array in our project immediately after the declaration of the HTTP header array

const char http\_header[] = {"HTTP/1.1 200 OKrnServer: nginxrnContent-Type: text/htmlrnConnection: keep-alivernrn"};

**const uint8\_t index\_htm[] = {};**

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

Here we also use const in order not to occupy the memory.

Now open the file **fsdata.c** , find there is such a place

**/\* raw file data (160 bytes) \*/**

And we copy everything that is after this comment to the clipboard before closing the brace. The last comma, in principle, we also do not need.

Then copy the copied data into our prepared array inside the curly braces. It turns out that's what

const uint8\_t index\_htm[] = {

**0x3c,0x68,0x74,0x6d,0x6c,0x3e,0x3c,0x62,0x6f,0x64,0x79,0x3e,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,0x53,0x54,0x4d,0x33,**

**0x32,0x46,0x31,0x30,0x33,0x78,0x38,0x3c,0x62,0x72,0x3e,0x3c,0x62,0x72,0x3e,0x57,**

**0x45,0x42,0x20,0x53,0x65,0x72,0x76,0x65,0x72,0x3c,0x2f,0x68,0x31,0x3e,0x0a,0x3c,**

**0x70,0x3e,0x3c,0x2f,0x70,0x3e,0x0a,0x3c,0x68,0x32,0x3e,0x46,0x65,0x61,0x74,0x75,**

**0x72,0x65,0x73,0x3c,0x2f,0x68,0x32,0x3e,0x0a,0x3c,0x70,0x3e,0x41,0x52,0x4d,0xc2,**

**0xae,0x20,0x33,0x32,0x2d,0x62,0x69,0x74,0x20,0x43,0x6f,0x72,0x74,0x65,0x78,0xc2,**

**0xae,0x2d,0x4d,0x33,0x20,0x43,0x50,0x55,0x20,0x43,0x6f,0x72,0x65,0x3c,0x2f,0x70,**

**0x3e,0x0a,0x3c,0x2f,0x62,0x6f,0x64,0x79,0x3e,0x3c,0x2f,0x68,0x74,0x6d,0x6c,0x3e**

};

This is the text of our page, converted into an array.

Now, return to the tcp\_read function where we left the empty body condition for the HTTP packet in the TCP packet, measure the packet sent there and determine how many segments it will fit in, and the length of the data in the last segment, and display it all, including the sender's port, in the terminal program

if (strncmp((char\*)tcp\_pkt->data,"GET / ", 6) == 0)

{

**tcpprop.data\_size = strlen(http\_header) + sizeof(index\_htm);**

**tcpprop.cnt\_data\_part = tcpprop.data\_size / tcp\_mss + 1;**

**tcpprop.last\_data\_part\_size = tcpprop.data\_size % tcp\_mss;**

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

**(unsigned long)tcpprop.data\_size, tcpprop.cnt\_data\_part, tcpprop.last\_data\_part\_size,tcpprop.port\_dst);**

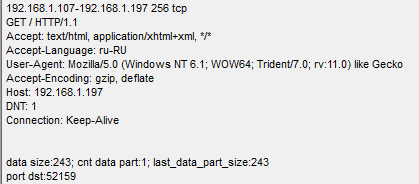
**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)str1,strlen(str1),0x1000);**

}

We created the page so that the package with it in the composition fit into one segment, since we still have at least one segmental response sent to the client.

We will collect our code, we will sew the controller and try again to request a page in the address bar. The consequences will be the same again, but we can already observe our dimensions in the terminal program, thereby making sure that our code is working and we determined that we have an HTTP request, and we should see the entire text of the request in the terminal program thanks to the code written above our condition.

We see in the terminal program the following information



Now, in the file tcp.h after the TCP statuses, we add some more data transfer statuses

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

**//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**

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

We will return to **tcp.c** to the same place from where we left and insert the code there to set the transmission status

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

**if (tcpprop.cnt\_data\_part==1)**

**{**

**tcpprop.data\_stat = DATA\_ONE;**

**}**

**else if (tcpprop.cnt\_data\_part>1)**

**{**

**tcpprop.data\_stat = DATA\_FIRST;**

**}**

}

Above the function **tcp\_read** we add the function of sending a response to the HTTP request in case it fits in one package, filling its body with standard content

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

**/\*Отправка однопакетного ответа HTTP\*/**

**uint8\_t tcp\_send\_http\_one(enc28j60\_frame\_ptr \*frame, uint8\_t \*ip\_addr, uint16\_t port)**

**{**

**uint8\_t res=0;**

**uint16\_t len=0;**

**uint16\_t sz\_data=0;**

**ip\_pkt\_ptr \*ip\_pkt = (void\*)(frame->data);**

**tcp\_pkt\_ptr \*tcp\_pkt = (void\*)(ip\_pkt->data);**

**return res;**

**}**

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

Again, go back to where we defined the size and set the statuses in the **tcp\_read** function , and call our function there if we have the data for one package

    tcpprop.data\_stat = DATA\_FIRST;

  }

**if(tcpprop.data\_stat==DATA\_ONE)**

**{**

**tcp\_send\_http\_one(frame, ip\_pkt->ipaddr\_src, tcpprop.port\_dst);**

**}**

}

Continue to write the body of our function **tcp\_send\_http\_one** .

Just in case, once again, we have our data

tcp\_pkt\_ptr \*tcp\_pkt = (void\*)(ip\_pkt->data);

**//Отправим сначала подтверждение на пакет запроса**

**sz\_data = be16toword(ip\_pkt->len)-20-(tcp\_pkt->len\_hdr>>2);**

Fill in some fields and call the TCP header preparation function

sz\_data = be16toword(ip\_pkt->len)-20-(tcp\_pkt->len\_hdr>>2);

**tcpprop.seq\_num = tcp\_pkt->num\_ask;**

**tcpprop.ack\_num = be32todword(be32todword(tcp\_pkt->bt\_num\_seg) + sz\_data);**

**len = sizeof(tcp\_pkt\_ptr);**

**tcp\_header\_prepare(tcp\_pkt, port, TCP\_ACK, len);**

Then, after preparing the IP header, we will send our packet

tcp\_header\_prepare(tcp\_pkt, port, TCP\_ACK, len);

**len+=sizeof(ip\_pkt\_ptr);**

**ip\_header\_prepare(ip\_pkt, ip\_addr, IP\_TCP, len);**

**//Заполним заголовок Ethernet**

**memcpy(frame->addr\_dest,frame->addr\_src,6);**

**eth\_send(frame,ETH\_IP,len);**

We can assume that the confirmation we sent. Now, in the same function, we send the response to the client in the form of a header and the page itself.

To do this, we first create a data field in the TCP header. We will not create a separate structure for the header and HTTP data because of the unstructured nature of the latter

eth\_send(frame,ETH\_IP,len);

**//Отправляем страницу**

**strcpy((char\*)tcp\_pkt->data,http\_header);**

**memcpy((void\*)(tcp\_pkt->data+strlen(http\_header)),(void\*)index\_htm,sizeof(index\_htm));**

First, we copy the header, and then at the address enlarged by the size of the title, the page itself.

Install the necessary flags, calculate the length of the TCP packet, including the data, but since the data is not included in the field intended for them in the TCP header, we do not fill it, because it is already full. Then, we recalculate the TCP checksum, since it involves the data

memcpy((void\*)(tcp\_pkt->data+strlen(http\_header)),(void\*)index\_htm,sizeof(index\_htm));

**len = sizeof(tcp\_pkt\_ptr);**

**len+=tcpprop.data\_size;**

**tcp\_pkt->fl = TCP\_PSH|TCP\_ACK;**

**tcp\_pkt->cs = 0;**

**tcp\_pkt->cs=checksum((uint8\_t\*)tcp\_pkt-8, len+8, 2);**

Next, we calculate the length of the IP packet, put it in the appropriate header field, recalculate the checksum of the IP header and send our packet, again without forgetting that the MAC addresses we do not change locally, as it is done above. And at the end we set the desired status, which will help us send a request to end the connection after receiving the packet with confirmation from the client

tcp\_pkt->cs=checksum((uint8\_t\*)tcp\_pkt-8, len+8, 2);

**len+=sizeof(ip\_pkt\_ptr);**

**ip\_pkt->len=be16toword(len);**

**ip\_pkt->cs = 0;**

**ip\_pkt->cs = checksum((void\*)ip\_pkt,sizeof(ip\_pkt\_ptr),0);**

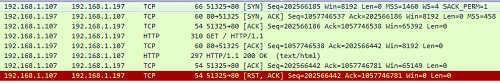
**//Заполним заголовок Ethernet**

**eth\_send(frame,ETH\_IP,len);**

**tcpprop.data\_stat=DATA\_END;**

return res;

You can now, in principle, collect the code and flash the controller. Although we do not see the page in the browser, since the browser will display it only when the connection is complete, but at least we'll see our packages in Wireshark. Therefore, try again to request in the browser our page (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/07/image06-5.png)

We see that our confirmation and our response to the HTTP client received our confirmation.

Let's go into the code and add another function over the **tcp\_read** function, which will send a request for disconnection after receiving confirmation after sending the last packet with HTTP response data, in our case so far only. Closer to the end of the function body, we set all the necessary statuses

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

**/\*Разъединяемся после получения подтверждения на последний пакет данных\*/**

**uint8\_t tcp\_send\_http\_dataend(enc28j60\_frame\_ptr \*frame, uint8\_t \*ip\_addr, uint16\_t port)**

**{**

**uint8\_t res=0;**

**uint16\_t len=0;**

**ip\_pkt\_ptr \*ip\_pkt = (void\*)(frame->data);**

**tcp\_pkt\_ptr \*tcp\_pkt = (void\*)(ip\_pkt->data);**

**tcpprop.seq\_num = tcp\_pkt->num\_ask;**

**tcpprop.ack\_num = tcp\_pkt->bt\_num\_seg;**

**len = sizeof(tcp\_pkt\_ptr);**

**tcp\_header\_prepare(tcp\_pkt, port, TCP\_FIN|TCP\_ACK, len);**

**len+=sizeof(ip\_pkt\_ptr);**

**ip\_header\_prepare(ip\_pkt, ip\_addr, IP\_TCP, len);**

**//Заполним заголовок Ethernet**

**memcpy(frame->addr\_dest,frame->addr\_src,6);**

**eth\_send(frame,ETH\_IP,len);**

**tcpprop.data\_stat=DATA\_COMPLETED;**

**tcp\_stat = TCP\_DISCONNECTED;**

**return res;**

**}**

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

It remains for us now only to call this function in the required place. And this necessary place is in the function **tcp\_read** at the very bottom of the file

else if (tcp\_pkt->fl == TCP\_ACK)

{

**if (tcpprop.data\_stat==DATA\_END)**

**{**

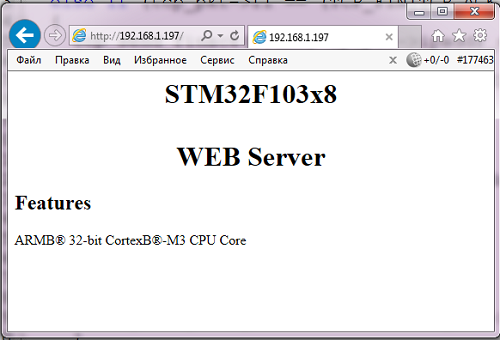
**tcp\_send\_http\_dataend(frame, ip\_pkt->ipaddr\_src, tcpprop.port\_dst);**

**}**

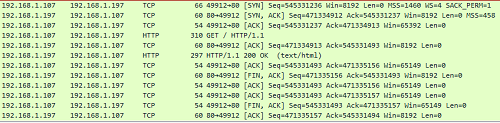
  HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"ACKrn",5,0x1000);

}

We will collect the code, let's say the controller and see the result of our hard work in the browser



Also we see that in WireShark we also have everything fine (click on the picture to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/07/image09-4.png)

Thus, we created a very simple web server, which can give the client small pages as a response to the HTTP request.

s