**STM Lesson 96. LAN8720. LWIP. TCP Client. Part 1**

Posted on [November 15, 2017](http://narodstream.ru/stm-urok-96-lan8720-lwip-tcp-client-chast-1/)by [http://1.gravatar.com/avatar/4824b24065500834db4b9f331b608833?s=32&d=mm&r=gNarod Stream](http://narodstream.ru/author/admin/) Published in [Programming STM32](http://narodstream.ru/rub_stm32/)- [No Comments ↓](http://narodstream.ru/stm-urok-96-lan8720-lwip-tcp-client-chast-1/#respond)

[Programming LessonsLearn all the in-house cooking **programming** training ! Get the book on the e-mail!To learn moresheremetev.info](https://an.yandex.ru/count/BzrDPvM_mte50Dy1CQQxari00000ECgs7402I09Wl0Xe172WmvR10O01sE3CgmU80QE8tP0fa06cakID99W1eiM-tYEW0SAapOqag07iWR3U8xW1tAUPz1N00GBO0PQmhH3W0UQKqmxe0HRu0RQLthu1Y083e0BUkAe2kG9x5NBVoQDxrV02mD7Dq0Vu0eA0W820e1I00-RYgP81Y0EIhhwt29W3ky06g0CIi0C4k0J_0UW4lmNu1A_90OW5hya1a0NCpmAW1RKyg0MPCx05cJEu1QbLm0NmcGV81R3g0T05X1VW1GRm1G6O1e3GhFCEe0Rk0gW6xWB91iOharf9W7KJqGR6lyZHIO1r4za60000S740002f1_XH5EO8PSH8i0U0W90Cq0S2u0U62lW70O080T08keg0WS2GW0BW2A-SbG602W712W0000000F0_s0e2u0g0YNhu2i3y5OWB1geB4CoWH-NuKG007DSG-bfr1G302u2Z1SWBWDIJ0TaB-54KvWXbn4Ze2w_90V0B1eWCjfNUlW7e30AO3PMCQF8D0FeD088E08aE00000000y3-G3i24FPWEnjVCr9M9uBeJe0x0X3sm3W6X3m0000000F0_g0_ue_6fWQ6PuaW0?stat-id=3&test-tag=89060471234561&format-type=24&banner-test-tags=eyI0ODQ2MjkwMDY3IjoiODkwNjA0NDE4ODI2MjQifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

[10 favorite techniques of artistsParticipate in a series of webinars and get your favorite painting lessons immediately!To learn morenewdirections.ru](https://an.yandex.ru/count/AZ6kNIFkW4e50CW1CQQxari00000ECgs7402I09Wl0Xe173mgPjnW07Cl-VExEgVyII80OF0sv4ra06WjhIyBvW1ygUrjowW0PYsjBmlg07SfhMtBhW1qE_sxXd00GBO0UQClmtW0UgOlWte0Go80WIW0fBxjnMv0diLSj_9etlLy0Axvh-J2_W2We20W82W5803xQhkeGw80zpicVi2i0C4k0J_0UW4nWlu1F9SY0NoN905qRu2e0Nl6wW5pnUm1SyNk0M1KS05y9a7o0MmwW7G1T0Au0LQy0K1c0QSuUGve0Rk0gW6xWB91iOharf9W7KJqGR6lyZHIO1r4za60000S740002f1_XHv7WePSH8i0U0W90Cq0S2u0U62lW70O080T08keg0WS2GW0BW2Cdzsmg02W712W0000000F0_s0e2u0g0YNhu2i3y5OWB2AeB40a0MiNuKG00thiO-bfr1G302u2Z1SWBWDIJ0TaB-57aU2Xbn4Ze2_9Sy0i6w0m2c0tylMRo3G3w3G223W293W0000000F0_a0x0X3sO3iRNpDILYU2w4w0EmzeSi0u1eGy00000003mFwWF-AFHrFYacU98?stat-id=4&test-tag=89060471234561&format-type=24&banner-test-tags=eyI2MzcxNTc3MjMyIjoiODkwNjA0NDE4ODI2MjQifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

We have already studied a couple of chips, designed for data transfer over the LAN interface: these are ENC28J60 and W5500. These are microchips from two different manufacturers. The first includes two levels - channel and physical, and the second - to all this, also network and transport, that is, it is a kind of incarnation in the hardware of the finished TCP / IP stack. This is very convenient for the developer, since you do not have to write the entire stack, as we did in the case of ENC28J60, but imposes a number of restrictions on the freedom of action, since something can not be changed in the stack, although there is a RAW mode that allows the stack to be disabled . But I did not use this mode, because the code would most likely be similar to the code of all lessons on the first chip, and why we need a repetition - it would be a huge waste of time,

Therefore, today we will try to work with one more chip, which deals with data transfer on this bus - it's **LAN8720** . This chip, like the first mentioned representative, is the brainchild of **Microchip** , only it on its board contains only the physical level. It would seem that this puts a huge responsibility on the developer, requiring him to organize so many levels competently, but this is not so. Because the **LWIP** library comes to our **aid**. This library was originally developed by Adam Dankels at the Swedish Institute of Computer and Networks Architectures (CNA). At the moment, this library is occupied by a group of developers (lwIP developers group) distributed by a modified BSD license, that is, it is free for use. **LWIP (lightweight IP)** is a library that implements a protocol stack with source code that supports the following protocols:

* IPv4 and IPv6 (Internet Protocol v4 and v6)
* ICMP (Internet Control Message Protocol) for network maintenance and debugging
* IGMP (Internet Group Management Protocol) for multicast traffic management
* UDP (User Datagram Protocol) • TCP (Transmission Control Protocol)
* DNS (Domain Name Server)
* SNMP (Simple Network Management Protocol)
* DHCP (Dynamic Host Configuration Protocol)
* PPP (Point to Point Protocol)
* ARP (Address Resolution Protocol)

And this is not all the protocols that are supported by this library, there are several more.

Also, there are three APIs that you can use to develop software for data transfer

**RAW / native API** is the interface used without the operating system. There are a number of shortcomings, but also a number of advantages, for example, provides freedom of action, which is often not enough for more advanced programmers.

**Netconn API** - is a high-level serial API, which requires a real-time operating system (RTOS). The Netconn API allows you to perform multi-threaded operations.

**BSD Socket API** is an API similar to Berkeley-Socket (developed over the Netconn API).

We are still working with the first RAW interface. Firstly, because we have not properly studied the operating system of FreeRTOS, and secondly, we do not get used to the network events themselves, and I think it will not be too difficult to do without using the OS.

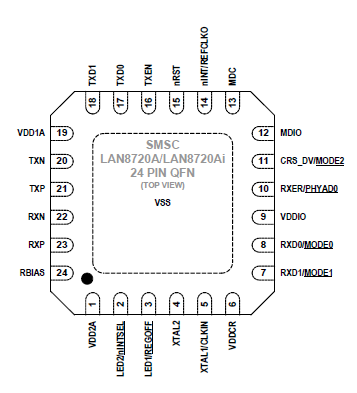
About the chip **LAN8720**  much to tell is not necessary, because there is no such abundance of registers as the W5500.

It's worth mentioning that unlike the W5500, it fully supports **Auto-MDIX** technology , which is often not enough to connect nodes without using hubs and routers, that is, directly, with a conventional non-crossover cable.

Also, this chip supports the exchange at speeds of 10/100 megabits per second in both half-duplex and full-duplex modes, it supports the Auto-negotiation mode, which allows the nodes to negotiate between themselves about the data exchange rates.

The rest of the charms of these chips, I think, we will feel when we start to use it. Also you can always download the latest version of the technical documentation for this chip on the official website of the manufacturer and read all the interesting details of this chip.

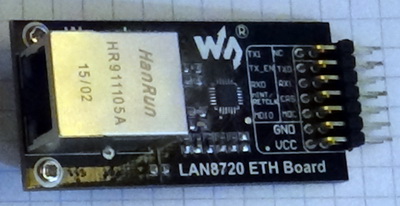
Let's see the location of the legs on it



These feet represent all the necessary contacts for the RMII bus, which we will talk about a little later and also all the necessary contacts for providing the physical level, also there are legs for clocking with a quartz resonator, as well as legs for power supply.

This chip I have on the module from **WaveShare** . We often use modules from this company.

This is how our module looks



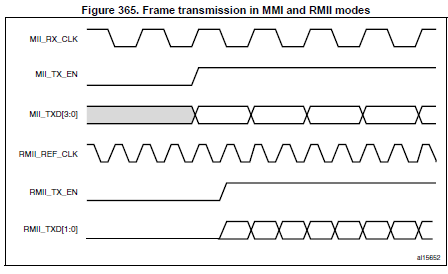
All contacts are marked, that is, you can safely use.

Only we can connect this chip, as well as the module, to each controller. Since this chip does not have a channel level, it should be supported and preferably by the hardware by the microcontroller itself. As such a controller, we'll take STM32F407VG, which is located on our favorite STM32F4-Discovery card, with which we passed many lessons before and know about it far from first-hand.

At the moment, many, but not many, but almost all of their new generation STM motherboards also install a similar chip, though a newer generation, but I have not noticed any difference in performance yet. But since our favorite Discovery 4 does not have such a microcircuit, and I'm sure many people want to organize a network data exchange with this card, we will connect to it just such a module, since the two previously considered chips with the channel level present in them to this there is no point in connecting a debug card because the MK located on it already has a hardware-implemented channel level, and I think that there is no reason for it to stand idle.

To the microcontroller STM32F407 similar chips having only a physical layer are connected via one of the inter-channel interfaces **MII** or **RMII**. These are similar tires, differing slightly. The data is transmitted in different directions on separate wires. For transmitting and receiving data in the MII, 4 wires are used for each of the directions. The interface is synchronous, that is, there is a synchronization leg, which is fed into the case of using MII pulses with a frequency of 25 megahertz. Therefore, we transmit 4 bits of information at once for one tick of synchronization. That's how our speed is 100 megabits per second. And we can transmit at this speed simultaneously in both directions, since we have 4 wires per direction. In the case of using RMII we already have more wires in the presence of wires, that is, only 2 wires per direction, but due to the fact that the clock is already produced at a frequency of 50 megahertz, then we also fart and receive data at a speed of 100 megabits per second. That's the difference between these interfaces.

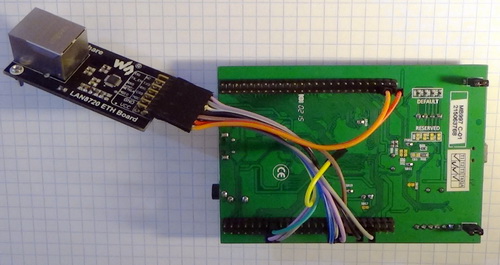
Let me also give some examples of data transfer via interfaces from the documentation to the controller



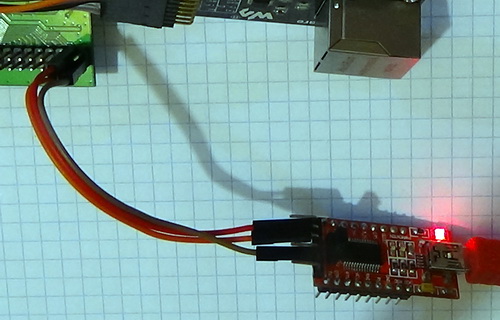
Now, I think, all became much more obvious. We see that the timing on the RMII interface is twice as fast, and the legs are half as large.

We will use RMII, since we do not have a choice, because our LAN8720 chip works exactly on this interface. In this there is a positive side. We win in the number of wires, but this imposes on us the responsibility for the quality of these wires. We can not use them too long, because 50 megahertz for data transfer - speed is not small, although not very large. I do not think we need to shield the wires. At least, everything is fine with me.

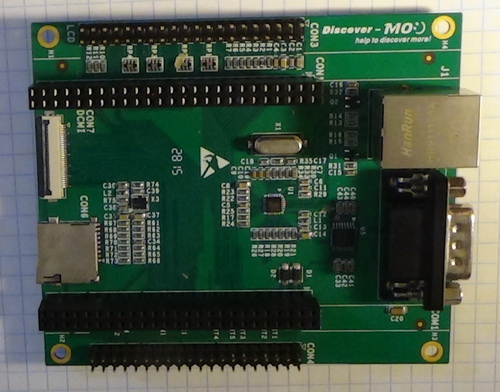
So here I have a chip connected to the board (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/11/image03.jpg)

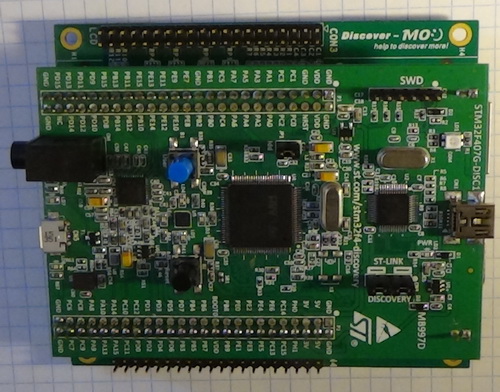
Also, let's connect a USB-TTL adapter for management and monitoring via the USART interface. We connect it to the contacts of the USART bus6



It is to the USART6 bus that I connected the adapter not only for reasons of convenience of its location on the edge, but also because of compatibility with the **STM32F4DIS-BB** card , which I also have available, which is very convenient for inserting the STM32F4-DISCOVERY debug card and which is very convenient , since it has many ready-made solutions, for example, the LAN8720 chip is already installed on board. In addition, you can connect the display and for which the individual legs, a video camera, and a Micro-SD card reader are also available, in general, there are many interesting things. Here is its top view



And this is how this board looks with the STM32F4-DISCOVERY debug card inserted into it



As we can see, USART is made on this board as a COM-port connector and schematically it is connected to USART6. True, I have not yet been able to use it. He did not get along with my virtual COM-port, but nevertheless, that's why I connected my USB-TTL adapter without using this expansion card to the USART6 bus contacts.

First, we will work without an expansion card, and then I will go to it with your permission, so it's more convenient for me, and you can continue to use the module from WaweShare, which is also convenient, and it is very inexpensive. Well, of course, if you do not have an expansion board **STM32F4DIS-BB** , if there is, then I recommend using it. And then I'll talk about the main difference between setting up a project using an expansion card and without using it. I did not immediately notice that I was not allowed to do without this board at all.

In the [**next part of the**](http://narodstream.ru/stm-urok-96-lan8720-lwip-tcp-client-chast-2/) lesson we will create and configure the project, write several functions and try to connect to the TCP server in practice.

# STM Lesson 96. LAN8720. LWIP. TCP Client. Part 2

Posted on [November 22, 2017](http://narodstream.ru/stm-urok-96-lan8720-lwip-tcp-client-chast-2/)by [http://1.gravatar.com/avatar/4824b24065500834db4b9f331b608833?s=32&d=mm&r=gNarod Stream](http://narodstream.ru/author/admin/) Published in [Programming STM32](http://narodstream.ru/rub_stm32/)- [12 comments ↓](http://narodstream.ru/stm-urok-96-lan8720-lwip-tcp-client-chast-2/#comments)

[Programming for beginnersDownload the free book and become a real programmer. Get it now:To learn moresheremetev.info](https://an.yandex.ru/count/CCNFznYjnvq50Di1CR-xari00000ECgs7402I09Wl0Xe172WmvR10O01vE3CgmU80VY6tP0fa06YakID99W1Zhs-tYEW0VwIp8qag06M_w_U8xW1XgMPz1N00GBO0PQmhH3W0UQKqmxe0HRu0RQLthu1Y083e0BUkAe2kGAJsOB2Kqa24l02uBoKtGFu0eA0W820q1E00y-UfQuDY0Ezp-qMc0E9nGIe0nAm0mIu1Fy1w0Iu1lW4XdE81OPpa0M0cWAW1S8Yg0MW7R05e1su1Tb7m0NmcGV81R3g0T05b0tW1Lhm1G6O1e3GhFCEe0Rk0gW6xWB91iOharf9W7KJqGR6lyZHIO1r4za60000S740002f1_XHX9xjQiH8i0U0W90Cq0S2u0U62lW70O080T08keg0WS2GW0BW2A-SbG602W712W0000000F0_s0e2u0g0YNhu2i3y5OWB1geB45MLhYZuKG00D0MA-rfr1G302u2Z1SWBWDIJ0TaB-564dkrgn4Ze2uPpy0i6Y0osbTw-0UWC0fWDbOneyWq0-Wq0WWu0YGu00000003mFv0Em8Gzc0x6rypKbOdWkXEW3i24FR0E0Q4F00000000y3-e3_YZWSxAtPhYI000?stat-id=3&test-tag=86311692165121&format-type=24&banner-test-tags=eyI0ODQ2MjkwMDY1IjoiODYzMTE2NjI4MTMxODQifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

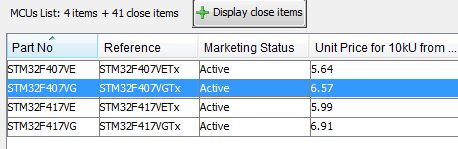
[Buy robot constructorTeach your child to program. 6900 for a set of shares. OrderTo learn moredorado-uno.ru](https://an.yandex.ru/count/KGw6LjmTQ6K50D41CR-xari00000ECgs7402I09Wl0Xe172AffoG1e01xCtW6OW1rBMagJMG0VZAnCelc07gZ-3LBg01zih4oY-e0UYEuDKkk07st_Rr6S010jW1ieMB5U01qBto4UW1OFW1elhUlW680WIW0g2-sHQv0fFPWi9JIG8Iy0Bmnf7t2VW2We20W83G4u03uU7RcWc80wRI-B43i0C4w0Jx0VW4vT82Y0NbqWAG1PJC0w05qcMe1P5Mi0MHLhW5_dJ01R3x3CW5yAK3q0NX9k05Ml050PW6_f3f6Q06xWAe1ku2oGR6AvDQIO1r4z46nh_8qKc0THFP1W00071n0000gGVuKU3X3sl4IB07W82G3D070k07XWhu1m60207G2BgAW870a802u0YGtwmBW0e1mGe00000003mFzWA0k0AW8bw-0h0_1M82mYg2n1VyvtC-5400FkLaljQTGK0m0k0emN82u3Kam7P2_XHuE4FQyH8w0lbqWBm2mQ83ABwthu1w0m2c0tylMRo3G3w3G223W293W0000000F0_a0x0X3sO3iRNpDILYU2w4w0Em8Gzi0u1eGy00000003mFwWF-AFHzV7Xck98?stat-id=4&test-tag=86311692165121&format-type=24&banner-test-tags=eyI2Mzg2Mzg5NjkyIjoiODYzMTE2NjI4MTMxODQifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

In the [**previous part of the**](http://narodstream.ru/stm-urok-96-lan8720-lwip-tcp-client-chast-1/) lesson we got acquainted with the LAN8720 chip, with inter-channel interfaces and with the LWIP protocol stack.

Well. Let's finally get to our project.

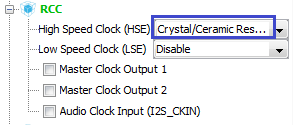
Our task is to create a TCP client that will be able to connect and disconnect to the TCP server on its own initiative, and our program should be able to receive and see the incoming data as text, and also send the same data to the TCP server. That is, there will be a kind of chat using the TCP protocol.

We will create a new project. As a controller, we choose, respectively, our controller **STM32F407VGTx**

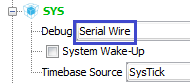


Then we'll set up our project.

The first thing RCC



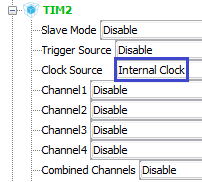
Then the programmer



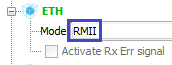
Turn on USART

image10

Turn on the timer for the future, perhaps in this project, it does not come in handy.



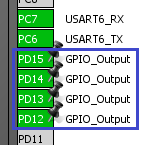
Now Ethernet



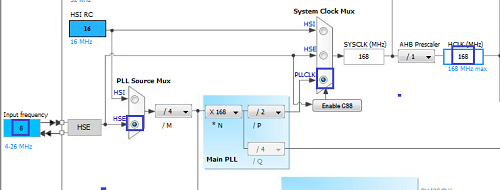
Then we connect the LWIP protocol stack library

image13

Include the LED feet on the output, which will bring some convenience and visibility into the debugging of the project

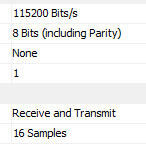


Let's go to **Clock Configuration** and adjust the dividers and multipliers to set the required frequencies (click on the image to enlarge the image)

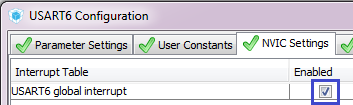
[](http://narodstream.ru/wp-content/uploads/2017/11/image15.png)

Go to Configuration.

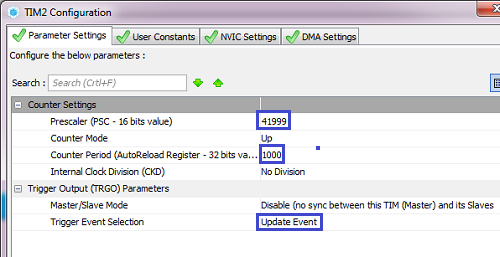
If you have such parameters in **USART6** , then we do not touch anything here



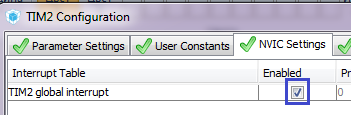
Let's go to the USART6 tab with interrupts and turn them on



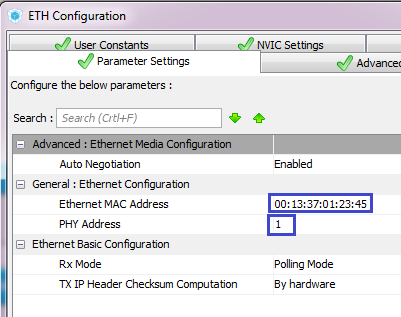
Then let's set up our timer



Also enable the interrupt timer

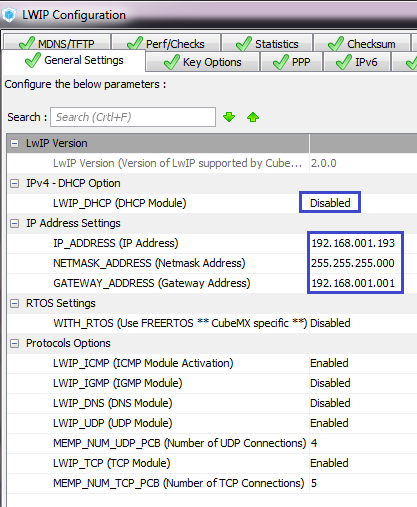


Now ETH. Bookmark with parameters

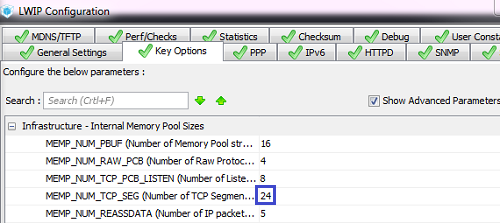


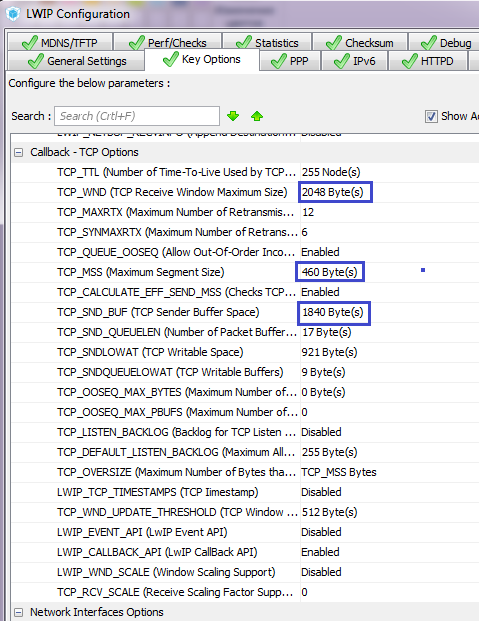
We set there the desired MAC-address, as well as the register address for access to the chip by RMII. We leave **1** . And here it is and the difference! **1** - this is when using the module from WaveShare, and **0** - if you use the expansion card STM32F4DIS-BB.

Now configure LWIP. First, the **General Settings** tab . We disable DHCP, we will use static addressing, so that we do not constantly recognize the address of our node on another node. Well and also in the future purposes. We once will directly connect each other to the two controllers via the LAN interface and it is unlikely that the other controller will correctly distribute the IP to our controller



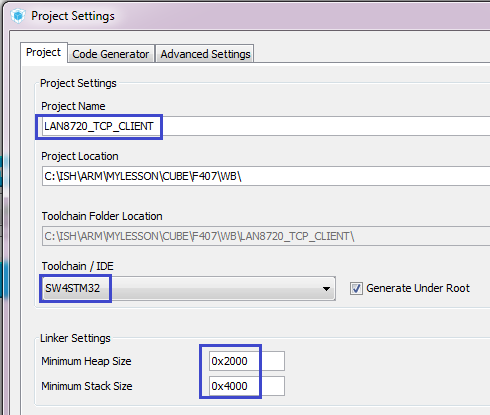
Then go to the **Key Options** tab , make sure that the dummy with advanced settings was installed and change some parameters here to make the window size habitual for us - 2048





No more protocols are touched or connected.

Let's go into the project settings, increase the stack and the heap and configure the generation of the project for System Workbench, well, give our project a name



Apply the settings, generate the project, go to **System Workbench** , connect our generated project there, go into its settings, remove the entire debug configuration, if any, and set the optimization level as usual to **1**.

Let's try to collect our project.

Let's go to the **main.c** file  and connect the structure for our network interface

/\* USER CODE BEGIN PV \*/

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

**extern struct netif gnetif;**

/\* USER CODE END PV \*/

Let's enter here such standard code in an infinite loop for maintenance of a constant work of our stack

/\* USER CODE BEGIN 3 \*/

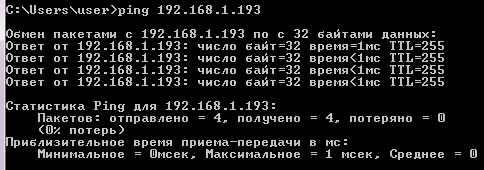
**ethernetif\_input(&gnetif);**

**sys\_check\_timeouts();**

}

/\* USER CODE END 3 \*/

We will connect our scheme, we will collect the code, we will sew the controller and we will try to ping our module



Create two files to work with the network - **net.h** and **net.c** with the following initial contents

**net.h:**

**#ifndef NET\_H\_**

**#define NET\_H\_**

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

**#include "stm32f4xx\_hal.h"**

**#include <string.h>**

**#include <stdlib.h>**

**#include <stdint.h>**

**#include "lwip.h"**

**#include "lwip/tcp.h"**

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

**#endif /\* NET\_H\_ \*/**

**net.c**

**#include "net.h"**

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

**extern UART\_HandleTypeDef huart6;**

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

We also connect our library in **main.c**

/\* USER CODE BEGIN Includes \*/

**#include "net.h"**

/\* USER CODE END Includes \*/

Let us now turn to the header file **net.h** .

Since we are writing a TCP client, we somehow need to be given a command to connect to the TCP server. To do this, we will need to know its IP address, as well as the port address. It is not difficult to learn them, of course, but they can differ each time, that's why we can not rigidly put them into our project. We have to somehow send them to the project each time as a command to the connection. To do this, our USART bus will serve us and we will do it from the terminal program, later on in the project, parsing the line that came from there. So first let's write a structure with the properties of our USART interface

#include "lwip/tcp.h"

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

**typedef struct USART\_prop{**

**uint8\_t usart\_buf[26];**

**uint8\_t usart\_cnt;**

**uint8\_t is\_tcp\_connect;//статус попытки создать соединение TCP с сервером**

**uint8\_t is\_text;//статус попытки передать текст серверу**

**} USART\_prop\_ptr;**

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

Now **go** to the file **net.c** and create a variable of the type of our structure

extern UART\_HandleTypeDef huart6;

**USART\_prop\_ptr usartprop;**

Also, we will create the initialization function of our network, zeroing out the field values ​​of the structure in it

USART\_prop\_ptr usartprop;

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

**void net\_ini(void)**

**{**

**usartprop.usart\_buf[0]=0;**

**usartprop.usart\_cnt=0;**

**usartprop.is\_tcp\_connect=0;**

**usartprop.is\_text=0;**

**}**

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

We add a prototype for our function in the header file and call it in **main ()** in the **main.c** file , at the same time we will include our timer

/\* USER CODE BEGIN 2 \*/

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

**net\_ini();**

/\* USER CODE END 2 \*/

Let's return to **net.c** and add an interrupt routine from USART to receive data

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

**void UART6\_RxCpltCallback(void)**

**{**

**}**

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

Let's create a prototype for this function, go back to main.c and add the actual handler for this interrupt, in which we call our function

/\* USER CODE BEGIN 4 \*/

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

**void HAL\_UART\_RxCpltCallback(UART\_HandleTypeDef \*huart)**

**{**

**if(huart==&huart6)**

**{**

**UART6\_RxCpltCallback();**

**}**

**}**

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

/\* USER CODE END 4 \*/

Back in the file net.c and add the global variable as a string array

USART\_prop\_ptr usartprop;

**char str[30];**

Again, go back to the **main.c** file and connect our line array

extern struct netif gnetif;

**extern char str[30];**

In the **main ()** function, we call the command to receive a character, otherwise our USART will never start receiving

net\_ini();

**HAL\_UART\_Receive\_IT(&huart6,(uint8\_t\*)str,1);**

Then we return to the file **net.c** , go to our handler, create a variable there and process the case of exceeding the size of the line buffer, and also at the end we increment our byte counter in the buffer and then call the data receiving command again

void UART6\_RxCpltCallback(void)

{

**uint8\_t b;**

**b = str[0];**

**//если вдруг случайно превысим длину буфера**

**if (usartprop.usart\_cnt>25)**

**{**

**usartprop.usart\_cnt=0;**

**HAL\_UART\_Receive\_IT(&huart6,(uint8\_t\*)str,1);**

**return;**

**}**

**usartprop.usart\_cnt++;**

**HAL\_UART\_Receive\_IT(&huart6,(uint8\_t\*)str,1);**

}

After checking the buffer overrun, we will take the received character to the buffer

  return;

}

**usartprop.usart\_buf[usartprop.usart\_cnt] = b;**

usartprop.usart\_cnt++;

If we encounter a newline, we'll put the line end character in the next element of the line array

usartprop.usart\_buf[usartprop.usart\_cnt] = b;

**if(b==0x0A)**

**{**

**usartprop.usart\_buf[usartprop.usart\_cnt+1]=0;**

**}**

Above we add the parsing function of our received string

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

**void string\_parse(char\* buf\_str)**

**{**

**}**

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

Let's return to our handler and call this function in the condition

if(b==0x0A)

{

  usartprop.usart\_buf[usartprop.usart\_cnt+1]=0;

**string\_parse((char\*)usartprop.usart\_buf);**

Then, in the same condition, after we parse the line, we zero out our counter, send the command to receive the character on the USART bus, which will wait for the next character from the bus, and exit our function

  string\_parse((char\*)usartprop.usart\_buf);

**usartprop.usart\_cnt=0;**

**HAL\_UART\_Receive\_IT(&huart6,(uint8\_t\*)str,1);**

**return;**

}

In the parsing function of the line, we will display the received string in the terminal program to see that we have correctly accepted

void string\_parse(char\* buf\_str)

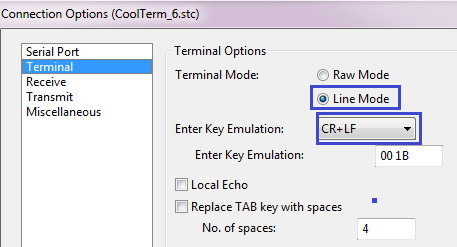
{

**HAL\_UART\_Transmit(&huart6, (uint8\_t\*)buf\_str,strlen(buf\_str),0x1000);**

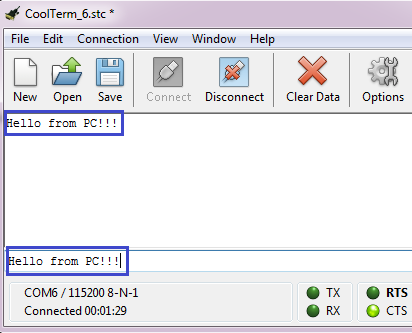
}

We will collect the code, we will sew the controller, start the terminal program. I used Cool Term this time, because there is a possibility to flexibly configure the data transfer.

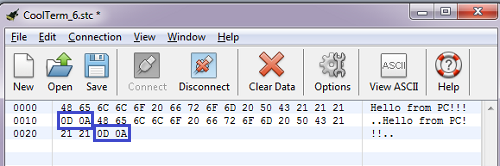
Let's go into the settings of this program, configure our port and speed (this, I think, is not worth showing - you can do it yourself), and also configure the transmission at the end of the line of carriage return and line feed characters



Let's connect to the port and try to transfer something



As we can see, the echo at us works, so we caught the line correctly on the controller. To clarify that we are also coming, and also that we do not lose them - the carriage return and linefeed characters can be transmitted by the next line, it should be echoed from a new line, and also by clicking the View Hex button in the terminal program toolbar, and then we see the codes of the characters



We see that the symbols are coming.

Let's return to our function of parsing the line and continue to write her body.

And before writing code for the function body, let's agree that we will send the following command to connect to the server

**t:** Server IP address: Server port address

And to disconnect with the help of a similar command

**c:** Server IP address: Server port address

In theory, for the separation, you do not need to specify the addresses, but suddenly we will have several connections and we will want to break just some of them. Until we have only one, but in the future everything is possible.

And now the body

  HAL\_UART\_Transmit(&huart6, (uint8\_t\*)buf\_str,strlen(buf\_str),0x1000);

**// если команда попытки соединения ("t:")**

**if (strncmp(buf\_str,"t:", 2) == 0)**

**{**

**}**

**//статус попытки разорвать соединение ("c:")**

**else if (strncmp(buf\_str,"c:", 2) == 0)**

**{**

**}**

**else**

**{**

**}**

}

Until we have only three variants of the lines - the command for the connection, the disconnect command, and the usual text for the chat.

Above we add the command processing function

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

**void net\_cmd(char\* buf\_str)**

**{**

**}**

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

We process the command to connect to the server by placing a certain code in the body of the corresponding condition in the parse function

if (strncmp(buf\_str,"t:", 2) == 0)

{

**usartprop.usart\_cnt-=1;**

**usartprop.is\_tcp\_connect=1;//статус попытки создать соединение TCP с сервером**

**net\_cmd(buf\_str+2);**

**HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_12, *GPIO\_PIN\_SET*);**

}

We'll do the same in the condition for disconnecting from the server

else if (strncmp(buf\_str,"c:", 2) == 0)

{

**usartprop.usart\_cnt-=1;**

**usartprop.is\_tcp\_connect=2;//статус попытки разорвать соединение TCP с сервером**

**net\_cmd(buf\_str+2);**

**HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_13, *GPIO\_PIN\_SET*);**

}

With the line, we'll figure it out later. In the meantime, go into the command processing function, add there an array of four elements for the IP address, a variable and two conditions

void net\_cmd(char\* buf\_str)

{

**uint8\_t ip[4];**

**uint16\_t port;**

**if(usartprop.is\_tcp\_connect==1)//статус попытки создать соединение TCP с сервером**

**{**

**}**

**if(usartprop.is\_tcp\_connect==2)//статус попытки разорвать соединение TCP с сервером**

**{**

**}**

}

Above we add parsing functions with IP address and port values, We already used such functions in early projects, so it does not make any sense to condemn them. So I'll just give their code

**port\_extract**

**ip\_extract**

We will also create global variables for storing the addresses of our server, which we hope to connect to soon

#include "net.h"

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

**uint8\_t ipaddr\_dest[4];**

**uint16\_t port\_dest;**

Also add one more line array for the preparation and output in USART of various service data

char str[30];

**char str1[100];**

At the very top after the declaration of various variables, we add the function of connection with the server via the TCP protocol

char str1[100];

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

**void tcp\_client\_connect(void)**

**{**

**}**

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

Let while hollow remains, simply it is necessary for us something to cause in our function of processing of commands **net\_cmd** in which we and we will return to write a body of a condition of connection with the server

if(usartprop.is\_tcp\_connect==1)//статус попытки создать соединение TCP с сервером

{

**ip\_extract(buf\_str,usartprop.usart\_cnt-1,ipaddr\_dest);**

**port\_dest=port\_extract(buf\_str,usartprop.usart\_cnt-1);**

**usartprop.usart\_cnt=0;**

**usartprop.is\_tcp\_connect=0;**

**tcp\_client\_connect();**

**sprintf(str1,"%d.%d.%d.%d:%u\r\n", ipaddr\_dest[0],ipaddr\_dest[1],ipaddr\_dest[2],ipaddr\_dest[3],port\_dest);**

**HAL\_UART\_Transmit(&huart6,(uint8\_t\*)str1,strlen(str1),0x1000);**

**HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_13, *GPIO\_PIN\_SET*);**

}

In the body of this condition, we get the server addresses from the string, writing them into global variables, reset the status variables, call the connection function with the server, display the addresses in the terminal program, and light the LED.

Above the initialization function, we add one more function that will be called when creating a connection to the server, the so-called callback function)

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

**static err\_t tcp\_client\_connected(void \*arg, struct tcp\_pcb \*tpcb, err\_t err)**

**{**

**return err;**

**}**

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

As arguments to this function, a pointer to the arguments will come, as well as a pointer to the structure of the TCP connection and the status of a possible error, which we also return back to the output from this function.

Add for this function a prototype in the same file (not in the header) higher

char str1[100];

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

**static err\_t tcp\_client\_connected(void \*arg, struct tcp\_pcb \*tpcb, err\_t err);**

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

We will also add a global variable of the TCP connection structure type and a message counter

char str1[100];

**struct tcp\_pcb \*client\_pcb;**

**\_\_IO uint32\_t message\_count=0;**

And also we will add an array for the data

char str1[100];

**u8\_t data[100];**

Let's return now to write the body of the connection function with the server **tcp\_client\_connect**

void tcp\_client\_connect(void)

{

**ip\_addr\_t DestIPaddr;**

**client\_pcb = tcp\_new();**

**if (client\_pcb != NULL)**

**{**

**IP4\_ADDR( &DestIPaddr, ipaddr\_dest[0], ipaddr\_dest[1], ipaddr\_dest[2], ipaddr\_dest[3]);**

**tcp\_connect(client\_pcb,&DestIPaddr,port\_dest,tcp\_client\_connected);**

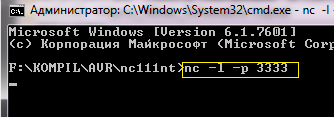
**HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_14, *GPIO\_PIN\_SET*);**

**}**

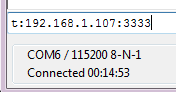
}

We created an instance of the connection structure, then entered the IP address of the server into the variable and called the connection function with the server, as parameters, we passed a structure instance, the server address, the server port, as well as the callback function, which will be called in case of connection with the server. Although we still need to write a lot of functionality that ensures the reliable work of the client, but if we now compile the code and try to connect to the server, then most likely we will succeed. Let's try.

We collect the code, flush the controller, then run the netcat program on the PC to create a TCP server that will listen to a specific port



We will also run the Wireshark traffic analyzer and give a command in the terminal program to connect to server port 3333



Let's see in WireShark

image31

We successfully connected to the server. So we are on the right track.

In the [**next part of the**](http://narodstream.ru/stm-urok-96-lan8720-lwip-tcp-client-chast-3/) lesson, we will write a series of functions and also learn how to properly disconnect from the client, as well as transfer and receive data from it.

# STM Lesson 96. LAN8720. LWIP. TCP Client. Part 3

Posted on [November 23, 2017](http://narodstream.ru/stm-urok-96-lan8720-lwip-tcp-client-chast-3/)by [http://1.gravatar.com/avatar/4824b24065500834db4b9f331b608833?s=32&d=mm&r=gNarod Stream](http://narodstream.ru/author/admin/) Published in [Programming STM32](http://narodstream.ru/rub_stm32/)- [No Comments ↓](http://narodstream.ru/stm-urok-96-lan8720-lwip-tcp-client-chast-3/#respond)

[Programming for beginnersDownload the free book and become a real programmer. Get it now:To learn moresheremetev.info](https://an.yandex.ru/count/69_whi7wSgS50Dy1CT6xari00000ECgs7402I09Wl0Xe172WmvR10O01vE3CgmU80VY6tP0fa06YakID99W1Zhs-tYEW0VwIp8qag06M_w_U8xW1XgMPz1N00GBO0PQmhH3W0UQKqmxe0HRu0RQLthu1Y083e0BUkAe2kG9sRRCu9Fej_l02ZjUSeGVu0eA0W820e1600__Fl_qBY0ElXgkW2fW3YSK4g0CIi0C4k0J_0UW4imRu18-H0eW5Zv42a0Nye0AW1VXHg0MuHR05k4Mu1Rj9m0NmcGV81R3g0T05dn_W1Lhm1G6O1e3GhFCEe0Rk0gW6xWB91iOharf9W7KJqGR6lyZHIO1r4za60000S740002f1_XHr9P4RyH8i0U0W90Cq0S2u0U62lW70O080T08keg0WS2GW0BW2A-SbG602W712W0000000F0_s0e2u0g0YNhu2i3y5OWB1geB4DFrceduKG00uCAV_5fr1G302u2Z1SWBWDIJ0TaB-57KbaHln4Ze2u-H0l0B1eWCjfNUlW7e30AO3PMCQF8D0FeD088E08aE00000000y3-G3i24FPWEnjVCr9M9uBeJe0x0X3sm3W6X3m0000000F0_g0_ue_6Ql-YRuaW0?stat-id=3&test-tag=75866331701249&format-type=24&banner-test-tags=eyI0ODQ2MjkwMDY1IjoiNzU4NjYzMDIzNDkzMTIifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

[Earn remotely, working 2-3 hours.Free course "Specialist in Internet Advertising" for 10 days! Sign up before 10.09.18To learn moreprofiinternet.artur-grant.ru](https://an.yandex.ru/count/A8mfJBcLSWK50Dy1CTAxari00000ECgs7402I09Wl0Xe173qmTYm0e01u_CDY06viP23CP01-ewzZosO0UBngS8ge07yZhsFBQW1pEwfmYgu0PYHfjeOm042s068r8uIu06wgvW6w04a-041Y084e0AixfmHkG9sRRCu9Fej_iW2YehjmVgOsTX_y0BLpE7x2_W2We20W82W4O03-8tKHuW3qwcLfmUm0mIu1Fy1w0JM0lW4hEG4Y0Miv0IG1RVZ1g05shS1g0NNcm6m1TUR0RW5nDm1m0NrW1Z81Q2D1j05XqRW1Gdm1G6O1e3GhFCEe0Rk0gW6xWB91iOharf9W7KJqGR6lyZHIO1r4za6000jS740002f1_XHD7o3RyH8i0U0W90Cq0S2u0U62lW70O080T08keg0WS2GW0BW29QtpWY02W712W0000000F0_s0e2u0g0YNhu2i3y5OWB2AeB45fHdSFuKG00atol_5fr1G302u2Z1SWBWDIJ0TaB-54qV8Dln4Ze2wpa1F0B1eWClhtUlW7e30AO3VozPl8D0FeD088E08aE00000000y3-G3i24FPWEnjVCr9M9uBeJe0x0X3sm3W6X3m0000000F0_g0_uev7Lh_2RuaW0?stat-id=4&test-tag=75866331701249&format-type=24&banner-test-tags=eyI2MDU2MDI3MDY5IjoiNzU4NjYzMDIzNDkzMTIifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

In the [**previous part of the**](http://narodstream.ru/stm-urok-96-lan8720-lwip-tcp-client-chast-2/) lesson we created and configured the project, wrote several functions and tried in practice to connect to the TCP server.

Now we must somehow disconnect, and we have not written the code yet. On the command line with netcat, you can use the **Ctrl + C** key combination and the server will disconnect from the client

image32

Let's try to disconnect from our server.

To do this, in net\_cmd we write the body code of the second condition.

if(usartprop.is\_tcp\_connect==2)//статус попытки разорвать соединение TCP с сервером

{

**ip\_extract(buf\_str,usartprop.usart\_cnt-1,ip);**

**port=port\_extract(buf\_str,usartprop.usart\_cnt-1);**

**usartprop.usart\_cnt=0;**

**usartprop.is\_tcp\_connect=0;**

**//проверим что IP правильный**

**if(!memcmp(ip,ipaddr\_dest,4))**

**{**

**//также проверим, что порт тоже правильный**

**if(port==port\_dest)**

**{**

**/\* close tcp connection \*/**

**tcp\_recv(client\_pcb, NULL);**

**tcp\_sent(client\_pcb, NULL);**

**tcp\_poll(client\_pcb, NULL,0);**

**tcp\_close(client\_pcb);**

**HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_15, *GPIO\_PIN\_SET*);**

**}**

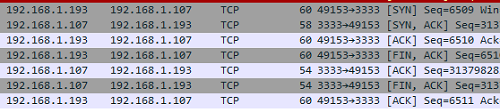
**}**

}

Here we also extract the IP address and port number of the server from the string, put them into local variables, compare them with global variables, and if it converges, then disconnect from the server, destroying the addresses of the callback-functions of the connection first.

Let's collect the code, we'll tell the controller and check if the connection to the server is correctly broken by the appropriate command.

Again, using the same command, connect to the server, and then use the **"c:"** command to request the client to disconnect



We see that the disconnection occurred correctly.

Excellent! It remains for us to teach our client to receive and transmit data.

First let's add our all service functions.

To do this, create four more callback functions, adding them over the initialization function

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

**static err\_t tcp\_client\_recv(void \*arg, struct tcp\_pcb \*tpcb, struct pbuf \*p, err\_t err)**

**{**

**err\_t ret\_err;**

**return ret\_err;**

**}**

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

**static void tcp\_client\_send(struct tcp\_pcb \*tpcb, struct client\_struct \* es)**

**{**

**}**

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

**static err\_t tcp\_client\_sent(void \*arg, struct tcp\_pcb \*tpcb, u16\_t len)**

**{**

**return *ERR\_OK*;**

**}**

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

**static err\_t tcp\_client\_poll(void \*arg, struct tcp\_pcb \*tpcb)**

**{**

**err\_t ret\_err;**

**return ret\_err;**

**}**

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

We will get acquainted with the purpose of these functions later.

Add for these functions also prototypes in the current file

static err\_t tcp\_client\_connected(void \*arg, struct tcp\_pcb \*tpcb, err\_t err);

**static err\_t tcp\_client\_recv(void \*arg, struct tcp\_pcb \*tpcb, struct pbuf \*p, err\_t err);**

**static void tcp\_client\_send(struct tcp\_pcb \*tpcb, struct client\_struct \* es);**

**static err\_t tcp\_client\_sent(void \*arg, struct tcp\_pcb \*tpcb, u16\_t len);**

**static err\_t tcp\_client\_poll(void \*arg, struct tcp\_pcb \*tpcb);**

Create an enumeration for connection states

struct tcp\_pcb \*client\_pcb;

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

**enum client\_states**

**{**

***ES\_NOT\_CONNECTED* = 0,**

***ES\_CONNECTED*,**

***ES\_RECEIVED*,**

***ES\_CLOSING*,**

**};**

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

Then, the structure for the properties of the client connection

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

**struct client\_struct**

**{**

**enum client\_states state; /\* connection status \*/**

**struct tcp\_pcb \*pcb; /\* pointer on the current tcp\_pcb \*/**

**struct pbuf \*p\_tx; /\* pointer on pbuf to be transmitted \*/**

**};**

**struct client\_struct \*cs;**

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

The field assignments are deciphered in the comments.

Now we add the function for the correct disconnection of the connection with the server after the function **tcp\_client\_connected**

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

**static void tcp\_client\_connection\_close(struct tcp\_pcb \*tpcb, struct client\_struct \* es)**

**{**

**/\* remove callbacks \*/**

**tcp\_recv(tpcb, NULL);**

**tcp\_sent(tpcb, NULL);**

**tcp\_poll(tpcb, NULL,0);**

**if (es != NULL)**

**{**

**mem\_free(es);**

**}**

**/\* close tcp connection \*/**

**tcp\_close(tpcb);**

**}**

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

In the body of this function, we destroy the addresses of callback functions, also free the memory allocated to the fields of our structure, and disconnect from the server.

Add for this function also a prototype in the current file

static err\_t tcp\_client\_connected(void \*arg, struct tcp\_pcb \*tpcb, err\_t err);

**static void tcp\_client\_connection\_close(struct tcp\_pcb \*tpcb, struct client\_struct \* es);**

static err\_t tcp\_client\_recv(void \*arg, struct tcp\_pcb \*tpcb, struct pbuf \*p, err\_t err);

Now start writing the body of the function **tcp\_client\_connected** , which has long been bored without a body

static err\_t tcp\_client\_connected(void \*arg, struct tcp\_pcb \*tpcb, err\_t err)

{

**struct client\_struct \*es = NULL;**

**if (err == *ERR\_OK*)**

**{**

**}**

**else**

**{**

**tcp\_client\_connection\_close(tpcb, es);**

**}**

  return err;

}

We create a connection structure, then write a condition that we got into this function without error, and if with an error, then we split up with the server.

We begin to write the body of the condition in the case of a positive result

if (err == *ERR\_OK*)

{

**es = (struct client\_struct \*)mem\_malloc(sizeof(struct client\_struct));**

**if (es != NULL)**

**{**

**}**

**else**

**{**

**tcp\_client\_connection\_close(tpcb, es);**

**return *ERR\_MEM*;**

**}**

}

Here we allocate memory under our structure and in case of impossibility to allocate memory we break connection with the server and we leave from function with a memory error.

Now we write the body code of the memory allocation condition in the case of a true result

if (es != NULL)

{

**es->state = *ES\_CONNECTED*;**

**es->pcb = tpcb;**

**es->p\_tx = pbuf\_alloc(*PBUF\_TRANSPORT*, strlen((char\*)data) , *PBUF\_POOL*);**

}

We assign the corresponding values ​​to the fields of our structure - the status of the successful connection, also the address of the connection structure, and also assign the address of our array for the data as a buffer.

Without leaving the condition, we write the code further

es->p\_tx = pbuf\_alloc(*PBUF\_TRANSPORT*, strlen((char\*)data) , *PBUF\_POOL*);

**if (es->p\_tx)**

**{**

**/\* copy data to pbuf \*/**

**pbuf\_take(es->p\_tx, (char\*)data, strlen((char\*)data));**

**/\* pass newly allocated es structure as argument to tpcb \*/**

**tcp\_arg(tpcb, es);**

**/\* initialize LwIP tcp\_recv callback function \*/**

**tcp\_recv(tpcb, tcp\_client\_recv);**

**/\* initialize LwIP tcp\_sent callback function \*/**

**tcp\_sent(tpcb, tcp\_client\_sent);**

**/\* initialize LwIP tcp\_poll callback function \*/**

**tcp\_poll(tpcb, tcp\_client\_poll, 1);**

**/\* send data \*/**

**tcp\_client\_send(tpcb,es);**

**return *ERR\_OK*;**

**}**

If the allocation of memory for the buffer is successful, we copy the data from it, then take the arguments to our structure, and then assign the callback functions we added as functions for various conditions of our TCP connection and send the buffer if it is not empty. For the function **tcp\_client\_poll,** we also pass the parameter - one. This is the value in seconds, meaning - how often this function will be called.

Now start writing the body of the callback-function of receiving TCP packets.

Also add the variable of our structure

static err\_t tcp\_client\_recv(void \*arg, struct tcp\_pcb \*tpcb, struct pbuf \*p, err\_t err)

{

**struct client\_struct \*es;**

  err\_t ret\_err;

Send her the address of the arguments, which are similar to the layout

err\_t ret\_err;

**es = (struct client\_struct \*)arg;**

Add a code with several conditions for different occasions

es = (struct client\_struct \*)arg;

**if (p == NULL)**

**{**

**}**

**else if(err != *ERR\_OK*)**

**{**

**}**

**else if(es->state == *ES\_CONNECTED*)**

**{**

**}**

**else if (es->state == *ES\_RECEIVED*)**

**{**

**}**

**else**

**{**

**}**

return ret\_err;

We process the first condition - obtaining a packet without data with a disconnect status. This is usually a package with the FIN flag from the server

if (p == NULL)

{

**es->state = *ES\_CLOSING*;**

**if(es->p\_tx == NULL)**

**{**

**tcp\_client\_connection\_close(tpcb, es);**

**}**

**ret\_err = *ERR\_OK* ;**

}

In this case, we are disconnected from the server.

Write the body code of the next condition - a non-empty packet with an error status

else if(err != *ERR\_OK*)

{

**if (p != NULL)**

**{**

**pbuf\_free(p);**

**}**

**ret\_err = err;**

}

In this case, we release the buffer memory and leave the loop body by entering the error status into the return status variable.

We process the following condition - the packet came in the normal connection

else if(es->state == *ES\_CONNECTED*)

{

**message\_count++;**

**tcp\_recved(tpcb, p->tot\_len);**

**HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_15, *GPIO\_PIN\_SET*);**

**es->p\_tx = p;**

**strncpy(str1,es->p\_tx->payload,es->p\_tx->len);**

**str1[es->p\_tx->len] = '';**

**HAL\_UART\_Transmit(&huart6, (uint8\_t\*)str1,strlen(str1),0x1000);**

**ret\_err = *ERR\_OK*;**

}

We increment the message counter, take our packet to the buffer, assign the pointer to the buffer to our structure, copy the contents of the buffer to a string array, end the string with zero, pass it to the terminal program for viewing, and leave the condition condition by entering the success status variable into the return status variable.

We write the body code of the next condition - successful reception of data

else if (es->state == *ES\_RECEIVED*)

{

**HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_13, *GPIO\_PIN\_SET*);**

**ret\_err = *ERR\_OK*;**

}

In this case, we just light the LED and go off with a successful status.

And the last body is a case that does not fit all the others - as a rule it is a confirmation of data reception

else

{

**/\* Acknowledge data reception \*/**

**tcp\_recved(tpcb, p->tot\_len);**

**/\* free pbuf and do nothing \*/**

**pbuf\_free(p);**

**ret\_err = *ERR\_OK*;**

}

return ret\_err;

We write the code of the next callback function, called when sending data to the server

static void tcp\_client\_send(struct tcp\_pcb \*tpcb, struct client\_struct \* es)

{

**struct pbuf \*ptr;**

**err\_t wr\_err = *ERR\_OK*;**

**while ((wr\_err == *ERR\_OK*) &&**

**(es->p\_tx != NULL) &&**

**(es->p\_tx->len <= tcp\_sndbuf(tpcb)))**

**{**

**}**

}

We create a pointer to the buffer, create a loop with the status, and create a loop in which we will be in the case of three conditions at once - we will not have an error, there will be a nonzero pointer to the buffer, and the buffer length will be less or equal to the amount of data transferred.

Now we write the body of our cycle

while ((wr\_err == *ERR\_OK*) &&

(es->p\_tx != NULL) &&

(es->p\_tx->len <= tcp\_sndbuf(tpcb)))

{

**ptr = es->p\_tx;**

**wr\_err = tcp\_write(tpcb, ptr->payload, ptr->len, 1);**

We get a pointer to the buffer and call the function to send the contents to the buffer for sending. The last unit is the flag. If one - then the packet will be sent without the PSH flag, and if the two are with him.

We write the body of the cycle further. Let's create several conditions

wr\_err = tcp\_write(tpcb, ptr->payload, ptr->len, 1);

**if (wr\_err == *ERR\_OK*)**

**{**

**}**

**else if(wr\_err == *ERR\_MEM*)**

**{**

**}**

**else**

**{**

**/\* other problem ?? \*/**

**}**

Let's write the body code of the first condition - if the function of writing to the buffer returns normal result to us

if (wr\_err == *ERR\_OK*)

{

**es->p\_tx = ptr->next;**

**if(es->p\_tx != NULL)**

**{**

**pbuf\_ref(es->p\_tx);**

**}**

**pbuf\_free(ptr);**

}

Here we go to the next data chain, if it is, then increase the reference count, and then release the buffer.

We process the following condition - if there is a memory error.

else if(wr\_err == *ERR\_MEM*)

{

**es->p\_tx = ptr;**

}

We assign a buffer pointer to the corresponding field of the structure, that is, postpone the transfer. That is, our buffer is still busy.

Well, let's leave the nasty case empty.

We work with the body of the next callback function - **tcp\_client\_sent** , which is called after the buffer is **sent** to the server.

static err\_t tcp\_client\_sent(void \*arg, struct tcp\_pcb \*tpcb, u16\_t len)

{

**struct client\_struct \*es;**

**LWIP\_UNUSED\_ARG(len);**

**es = (struct client\_struct \*)arg;**

**if(es->p\_tx != NULL)**

**{**

**tcp\_client\_send(tpcb, es);**

**}**

  return *ERR\_OK*;

We also create a pointer to the structure, pass arguments to it, and if something is left in the buffer, then we pass it to the server.

And the last callback function whose body we start to write is the function **tcp\_client\_poll** , which is called constantly and monitors the network, even if there is no exchange

err\_t ret\_err;

**struct client\_struct \*es;**

**es = (struct client\_struct\*)arg;**

**HAL\_GPIO\_TogglePin(GPIOD, GPIO\_PIN\_14);**

We will also create a pointer to the structure, give it arguments and change the state of the LED.

Write the function body code on

HAL\_GPIO\_TogglePin(GPIOD, GPIO\_PIN\_14);

**if (es != NULL)**

**{**

**}**

**else**

**{**

**}**

return ret\_err;

We have created a condition that will verify that the pointer to the connection structure exists.

First, we write the body of the positive result of condition

if (es != NULL)

{

**if (es->p\_tx != NULL)**

**{**

**}**

**else**

**{**

**if(es->state == *ES\_CLOSING*)**

**{**

**tcp\_client\_connection\_close(tpcb, es);**

**}**

**}**

**ret\_err = *ERR\_OK*;**

}

If we have a non-zero buffer, then we do nothing, for this we have another function, and if the connection is closed, then we close it out of the condition with good status.

Now a nasty case

else

{

**tcp\_abort(tpcb);**

**ret\_err = ERR\_ABRT;**

}

return ret\_err;

In this case, we do nothing, with what there is a special function **tcp\_abort** for this , and we exit with a special return status.

With the event handler functions, we are done.

It remains to us to process a string from the terminal, which is not a command.

To do this, we **'ll** write a function for parsing the string  **string\_parse to** pass the string to the server

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

**void sendstring(char\* buf\_str)**

**{**

**tcp\_sent(client\_pcb, tcp\_client\_sent);**

**tcp\_write(client\_pcb, (void\*)buf\_str, strlen(buf\_str), 1);**

**tcp\_output(client\_pcb);**

**usartprop.usart\_cnt=0;**

**usartprop.is\_text=0;**

**}**

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

The string (and in general any data) is transmitted via TCP as follows: the handler function is initialized, then the data is written to the buffer using a special library function, and then the transfer function of the prepared buffer is called. After the transfer, we reset the counter and the status of the normal line.

Well, it's clear, the function will never work unless it is called.

Let's **go** to the **string\_parse** function  and write the body of the parse condition

else

{

**usartprop.is\_text=1;**

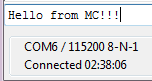
**sendstring(buf\_str);**

**HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_12|GPIO\_PIN\_13, *GPIO\_PIN\_RESET*);**

}

Here like and all with the code. It remains to look at the result of our labors.

We'll collect the code, we'll run the controller, run netcat to listen to the port, connect it with a special command and try something and send it to the computer of their terminal program



Let's see if the line came to the console netcat

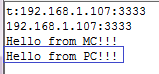
image35

Everything has come! Excellent!

Now, on the contrary, we pass the string from the console to our MK

image36

Let's see if the message came to the controller, in the terminal program window

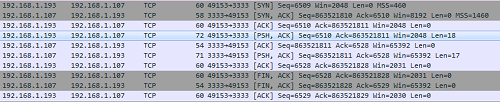


Everything has come.

Let's send the command to disconnect.

The client with the server was successfully disconnected.

Let's see an exchange in WireShark



The exchange was successful. You can also look. all the bytes have arrived, but we will not do this, as I have repeatedly checked this, and if there were errors, I already eliminated them as the code was written.

Thus, we learned how to organize a simple TCP-client using LWIP protocol stack using the chip **LAN8720** , which includes only the physical layer.

Thank you for attention!