**STM Lesson 125. LAN8742A. LWIP. NETCONN. TCP Client**

Posted on [July 19, 2018](http://narodstream.ru/stm-urok-125-lan8742a-lwip-netconn-tcp-client/)by [http://1.gravatar.com/avatar/4824b24065500834db4b9f331b608833?s=32&d=mm&r=gNarod Stream](http://narodstream.ru/author/admin/) Posted in [FreeRTOS](http://narodstream.ru/freertos/) , [LAN](http://narodstream.ru/lan/) , [Programming STM32](http://narodstream.ru/rub_stm32/)- [No Comments ↓](http://narodstream.ru/stm-urok-125-lan8742a-lwip-netconn-tcp-client/#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/JfKmXAZmIZW50Du1CPN1ari00000ECgs7402I09Wl0Xe172WmvR10O01sE3CgmU80QE8tP0fa06cakID99W1eiM-tYEW0SAapOqag07iWR3U8xW1tAUPz1N00GBO0PQmhH3W0UQKqmxe0HRu0RQLthu1Y083e0BUkAe2kG8V1ah76i1KxV02tOUl0lW2We20W82W5803vkAfaW680vAklhS8c0Exm0Qe0nAm0mIu1Fy1w0J70_W4wRS1Y0Nfjm6G1O740g05apUe1Tuki0NUBhW5i5B01V2P1yW5iEe1q0M35U051l050PW6WD2iymwW1ku2g0Rk0ia6nYkJMac0THFH1iQ_oD59W7KJsGO0001mSG000Aa7-57e43tFnKYm1u20a0pG1mBW1uOA-0S1W0W1q0YwYe21m9200k08hvoL0O0A0S4A00000000y3_O2WBW2e29UlWAmFmLY0i6gWiGuMv4qFXH002buPrKMtK50C0BWAC5o0k0r9C1sGluKUWGFS_5IEWBwRS1y0i6Y0osbTw-0UWC0fWD-9nbyWq0-Wq0WWu0YGu00000003mFv0Em8Gzc0x6rypKbOdWkXEW3i24FR0E0Q4F00000000y3-e3_YZyTt8v_FYI000?stat-id=3&test-tag=89061544976385&format-type=24&banner-test-tags=eyI0ODQ2MjkwMDY3IjoiODkwNjA0NDE4ODI2MjQifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

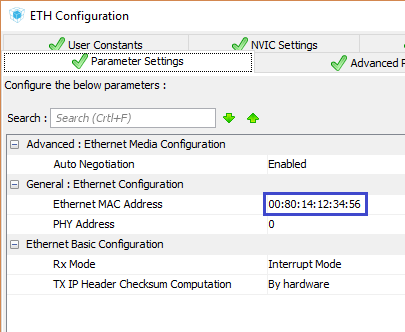
[PHP Programming Courses2 lectures a week, personal tutor, final project site, certificate](https://an.yandex.ru/count/2-6jaTZt4SS50Cy1CPR1ari00000ECgs7402I09Wl0Xe173MZO-53e01q8gZjGE80UMTcjGqa06KnkkWBvW1wBlk-IsW0Oh6ww0lg06Gk-xvBRW1w9tytXd00GBO0OgrwnJW0TIzXWVe0QO2-07czzw-0OW21A02hCMi3xa27mPAnnh0LEtm0kgisSm5-0A2W820WA0Kc0EHAB031BW4_m7e19a2-0J1zmA81S7t0f05jjW6e0NKSAW5l5-m1RnVk0NUsG701Pkq6iW5iDi6q0NwAk05F_050PW6mA_S_GsW1ku2g0Rk0ia6nYkJMac0THFH1iQ_oD59W7KJsGO0001mSG000Aa7-5681L_FnKYm1u20a0pG1mBW1uOA-0S1W0W1q0YwYe21m9200k08rgBM1u0A0S4A00000000y3_O2WBW2e29UlWAmFmLY0i8gWiG6fTl5lXH000DNwPKMtK50C0BWAC5o0k0r9C1sGluKOW5Ny_5IEWBmVS2y0i6Y0pczzw-0UWC0fWDoR9dyWq0-Wq0WWu0YGu00000003mFv0Em8Gzc0x6rypKbOdWkXEW3gAa3R0E0Q4F00000000y3-e3_YZaSJew_FYI000?stat-id=4&test-tag=89061544976385&format-type=24&banner-test-tags=eyI2MzQyNjY4NjgyIjoiODkwNjA0NDE4ODI2MjQifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)18+

We continue to work with the TCP protocol and today we will try to create a client. If someone thinks that the client is easier to write than the server, then this is in vain, and we have seen this many times.

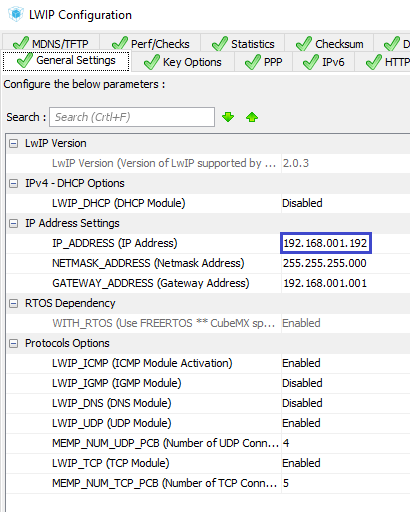
As a server, we now have, on the contrary, a PC.

As a client, by tradition, we will use the **STM32F746-Discovery** card , and we will create a project from the project of the [**previous lesson**](http://narodstream.ru/stm-urok-124-lan8742a-lwip-netconn-tcp-server/) **LAN8742\_TCP\_SERVER\_NETCONN** , only we will call it now. respectively, **LAN8742\_TCP\_CLIENT\_NETCONN** .

Open our project in the Cube MX, in **Configuration we** 'll go first to the **ETH** settings and change the MAC address there, because I think you already guessed that we will then connect our client to the server on the debug card



In the LWIP settings, we also change the network address of our client



Generate the project for  **System Workbench**  and open it there. Set the optimization level to  **1** , remove it with debugging settings and comment out the lines unknown to the compiler in the **main.c** file  .

Let's try to compile the project and start working with the file  **main.c.**

The first step is to fix the header in  **main ()**

TFT\_DisplayString(0, 10, (uint8\_t \*)"TCP **Client**", *CENTER\_MODE*);

We will remove the global structure for sockets, as well as the variables of this structure

~~typedef struct struct\_sock\_t {~~

~~uint16\_t y\_pos;~~

~~struct netconn \*conn;~~

~~} struct\_sock;~~

~~struct\_sock sock01, sock02;~~

And add us instead a structure for the connection, as well as one variable, since we on one client will create only one connection

} struct\_out;

**typedef struct struct\_conn\_t {**

**uint32\_t conn;**

**uint32\_t buf;**

**} struct\_conn;**

**struct\_conn conn01;**

In the **tcp\_thread** function **,** change the name and clear the body

static void **send\_thread**(void \*arg)

{

}

In the function of the task, by default **StartDefaultTask will**  remove almost all user code, leaving only this

/\* USER CODE BEGIN 5 \*/

struct netconn \*conn;

err\_t err;

/\* Infinite loop \*/

Enter the IP address of the computer in the structure (you will have, respectively, another)

err\_t err;

**ip\_addr\_t ServerIPaddr;**

**IP4\_ADDR(&ServerIPaddr, 192, 168, 1, 87);**

Set another text color

IP4\_ADDR(&ServerIPaddr, 192, 168, 1, 87);

**TFT\_SetTextColor(LCD\_COLOR\_BLUE);**

Initialize the connection structure by getting the pointer to it

TFT\_SetTextColor(LCD\_COLOR\_BLUE);

**conn = netconn\_new(*NETCONN\_TCP*);**

If the structure is successfully created, assign a port to our connection and associate it with the connection

conn = netconn\_new(*NETCONN\_TCP*);

**if(conn!=NULL)**

**{**

**err = netconn\_bind(conn, NULL, 4555);**

**}**

In case of successful creation of connection with the port, we try to connect to the server, namely with a certain port of it, and otherwise we destroy the structure, freeing the memory occupied by it

err = netconn\_bind(conn, NULL, 4555);

**if (err == *ERR\_OK*)**

**{**

**err = netconn\_connect(conn, &ServerIPaddr, 5444);**

**}**

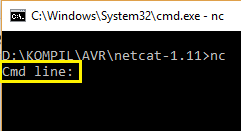
**else**

**{**

**netconn\_delete(conn);**

**}**

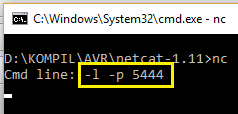
Run the netcat program by entering only the program name on the command line. As a result, we get this prompt in the command line



Also launch the WireShark traffic analyzer, filtering it according to the IP address of our client

http://narodstream.ru/wp-content/uploads/2018/07/stm125img03.png

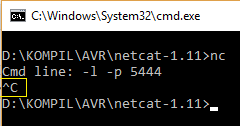
In netcat, at the command line, enter the following command so that the server listens on the port the client is trying to connect to



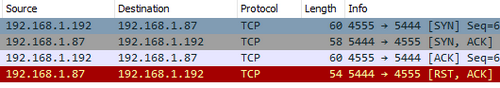
We will collect the project, we will sew the controller and see the result in the program WireShark (click on the picture to enlarge the image)

[http://narodstream.ru/wp-content/uploads/2018/07/stm125img05_0500.png](http://narodstream.ru/wp-content/uploads/2018/07/stm125img05.png)

To disconnect from the client, in the program netcat we enter the key combination " **Ctrl + C** "



The connection will break even if it is not correct, but the port will be freed from wiretapping



With the correctness we will understand later. We will sever the connection on the initiative of the client, then everything will be normalized.

After the task function for sending packets **send\_thread,**  we will create another function of the task of receiving packets, because the client must take something and be able to do something

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

**static void recv\_thread(void \*arg)**

**{**

**for(;;)**

**{**

**osDelay(1);**

**}**

**}**

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

In the default task of **StartDefaultTask,**  we create our tasks and transfer the structures to them as parameters

err = netconn\_connect(conn, &ServerIPaddr, 5444);

**if (err == *ERR\_OK*)**

**{**

**conn01.conn = conn;**

**sys\_thread\_new("send\_thread", send\_thread, (void\*)&conn01, DEFAULT\_THREAD\_STACKSIZE,*osPriorityNormal* );**

**sys\_thread\_new("recv\_thread", recv\_thread, (void\*)&conn01, DEFAULT\_THREAD\_STACKSIZE,*osPriorityNormal* );**

**}**

Let's start with the transfer, so let's move on to the task function to send **send\_thread** packages  and declare there several structures and a variable for tracking errors

static void send\_thread(void \*arg)

{

**struct\_conn \*arg\_conn;**

**struct\_out \*qstruct;**

**struct netconn \*conn;**

**err\_t sent\_err;**

We take the pointer to the connection structure from the parameters

err\_t sent\_err;

**arg\_conn = (struct\_conn\*) arg;**

**conn = (void\*)arg\_conn->conn;**

Create a variable to store the number of system quanta, as well as a small string buffer

conn = (void\*)arg\_conn->conn;

**uint32\_t syscnt = 0;**

**char buf[15] = {};**

Create an infinite loop in which we will cyclically transfer packets to the server

char buf[15] = {};

**for(;;)**

**{**

**osDelay(1000);**

**}**

We learn in the cycle the number of system quanta that have passed since the controller was turned on

for(;;)

{

**syscnt = osKernelSysTick();**

To check the process of disconnection, we will transfer our quanta to the server not infinitely, but for example, until their number exceeds 50,000. In this case, we will disconnect the connection to the server and report on it in the display at the bottom, after leaving cycle

syscnt = osKernelSysTick();

**if(syscnt>50000)**

**{**

**netconn\_close(conn);**

**netconn\_delete(conn);**

**qstruct = osMailAlloc(strout\_Queue, osWaitForever);**

**qstruct->y\_pos = 160;**

**strcpy(qstruct->str,"Connection was closed!");**

**osMailPut(strout\_Queue, qstruct);**

**osMailFree(strout\_Queue, qstruct);**

**osDelay(2);**

**break;**

**}**

If we do not insert a delay, the queue will not work, and if we put 1, it will not work until the end.

And if we have not yet reached this value, then we will transfer the number of passed system quanta to the server in the form of a line with carriage return and translation to a new line

  break;

  }

**sprintf(buf,"%lu\r\n",syscnt);**

**sent\_err = netconn\_write(conn, (void \*) buf, strlen(buf), NETCONN\_COPY);**

If everything was okay, we will display our transmitted buffer on the display, replacing the carriage return with the end of line character - zero

sent\_err = netconn\_write(conn, (void \*) buf, strlen(buf), NETCONN\_COPY);

**if(sent\_err == *ERR\_OK*)**

**{**

**qstruct = osMailAlloc(strout\_Queue, osWaitForever);**

**qstruct->y\_pos = 60;**

**buf[strlen(buf)-2]=0;**

**strcpy(qstruct->str,buf);**

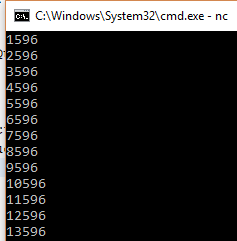
**osMailPut(strout\_Queue, qstruct);**

**osMailFree(strout\_Queue, qstruct);**

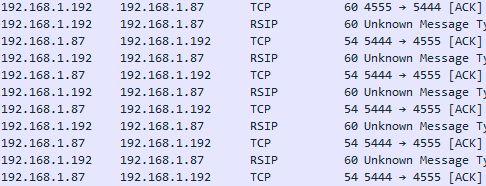
**}**

Let's run our netcat again in the same way, forcing it to listen again to the port, then we'll compile the code and run the controller.

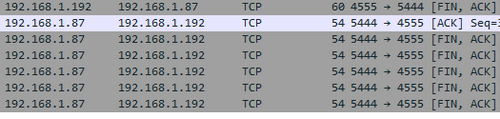
The result in netcat will be



In WireShark we should see such a picture

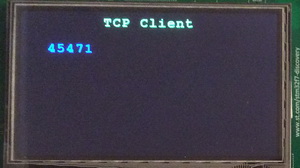


Upon reaching 50,000 we see an attempt at disengagement

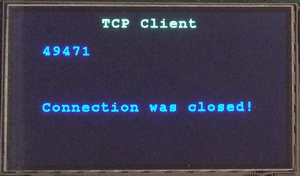


The client requested a connection, the server responded with a confirmation, then the server also requested a disconnect, but the client did not respond, the server repeated four more attempts, but the client did not respond. And he did not answer because he does not know how to do it, but for now.

On the display we see this picture during the transfer of packets



At the end of the transfer of packages



And netcat will return to the standard line at the end of the reception



With the transfer understood, go into the function of the task of receiving packages  **recv\_thread** and create there also some similar structures and variables

static void recv\_thread(void \*arg)

{

**struct\_conn \*arg\_conn;**

**err\_t recv\_err;**

**struct netconn \*conn;**

**struct netbuf \*inbuf;**

**struct\_out \*qstruct;**

**uint8\_t\* buf;**

**u16\_t buflen;**

Take a pointer to the connection structure from the task parameters

u16\_t buflen;

**arg\_conn = (struct\_conn\*) arg;**

**conn = (void\*)arg\_conn->conn;**

We remove the delay from the infinite loop, and instead we try to accept the packet

for(;;)

{

~~osDelay(1);~~

**recv\_err = netconn\_recv(conn, &inbuf);**

}

If the packet has arrived, then we will take it to the buffer

recv\_err = netconn\_recv(conn, &inbuf);

**if (recv\_err == *ERR\_OK*)**

**{**

**if (netconn\_err(conn) == *ERR\_OK*)**

**{**

**netbuf\_data(inbuf, (void\*\*)&buf, &buflen);**

**}**

**}**

In case the buffer length is greater than 1, then display the contents of the buffer on the display, and then free the memory under the buffer structure

netbuf\_data(inbuf, (void\*\*)&buf, &buflen);

**if(buflen>1)**

**{**

**qstruct = osMailAlloc(strout\_Queue, osWaitForever);**

**qstruct->y\_pos = 100;**

**strncpy(str\_buf,(char\*)buf,buflen);**

**str\_buf[buflen-1]=0;**

**sprintf(qstruct->str,"%-20s", str\_buf);**

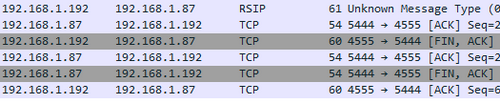
**osMailPut(strout\_Queue, qstruct);**

**osMailFree(strout\_Queue, qstruct);**

**}**

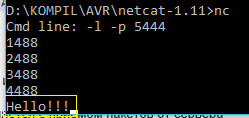
**netbuf\_delete(inbuf);**

We will collect the code, we will tell the controller, and we will see that at the end of the transfer of all data, the connection is broken correctly by us,

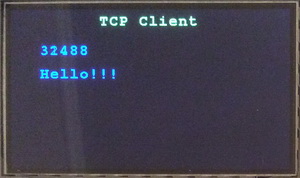


Simply, most likely, we did not accept all server confirmations for our packages, since the receiving buffer was not released, so we could not accept the packet with the desire to break the connection.

While receiving packets from the client on the server in the netcat program, we'll try to pass something to the client and see how the client handles receiving packets from the server



Let's look at the display



Everything comes perfectly.

So, in this lesson, we were able to create a workable TCP client from our board, which is able to request a connection, to break it as well, and, most importantly, is able to exchange information with the server in the form of TCP packets.

Thank you all for attention!

bv