**STM Lesson 124. LAN8742A. LWIP. NETCONN. TCP Server**

Posted on [July 10, 2018](http://narodstream.ru/stm-urok-124-lan8742a-lwip-netconn-tcp-server/)by [http://1.gravatar.com/avatar/4824b24065500834db4b9f331b608833?s=32&d=mm&r=gNarod Stream](http://narodstream.ru/author/admin/) Posted in [LAN](http://narodstream.ru/lan/) , [Programming STM32](http://narodstream.ru/rub_stm32/)- [No Comments ↓](http://narodstream.ru/stm-urok-124-lan8742a-lwip-netconn-tcp-server/#respond)

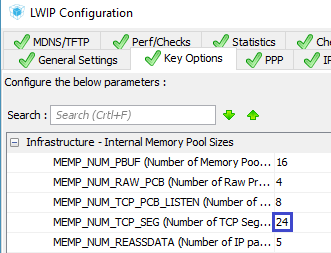
[Help Desk from 79 000 rubles a year!No restrictions on the number of users. Test for free!To learn morevsdesk.ru](https://an.yandex.ru/count/H2178WwHFSW50D41CN31ari00000ECgs7402I09Wl0Xe173asxZP3O01tnY80OwZtgKqa07ky_A3BvW1pDYClYsu0S22jz4Pm042s076ZjeKu06KhP8Dw06G0lW1tg3UlW680WEW0k3SaH6v0iuAFhJeUkn7y0BneRtz1lW2We20W82W4u03wjN3FOW3iUcUsmkO0_M-1QW34h031BW4_m7e17Ru1FRH1uW5zj47a0NxgWgW1R2b0gW5qFW1i0NG-06u1QpG0i05i84ao0NmamdG1V1lu0K-y0K1c0Q0qApp3g06xWAe1ku2oGR6AvDQIO1r4z46nh_8qKc0THFP1W00071n0000gGVuKLXPGyR5IB07W82G3D070k07XWhu1m60207G29gAW870a802u0YmxiW8W0e1mGe00000003mFzWA0k0AW8bw-0h0_1M82mQg2n2Cmvd6-5400EjpNr9RTGK0m0k0emN82u3Kam7P2_XHM5b3niL8w0lsqGVm2mQ83DwWthu1w0m2c0sLZ6Zo3G3w3G223W293W0000000F0_a0x0X3sO3iRNpDILYU2w4w0Em8Gzi0u1eGy00000003mFwWF-AEncQd8yU98?stat-id=3&test-tag=84662424723457&format-type=24&banner-test-tags=eyI2MzEyMzIwMjQ3IjoiODQ2NjIzOTUzMzg3NTIifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)18+

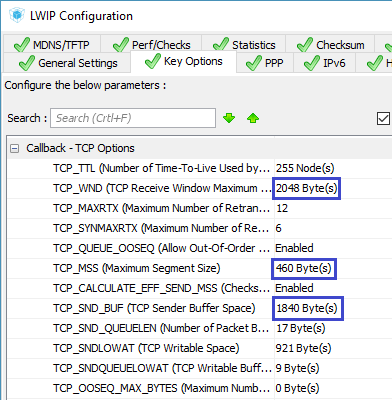
[Programming kitRobot, mobile application and study guide. Learn now is interesting!To learn moredorado-uno.ru](https://an.yandex.ru/count/21nMtRAdFQy50Di1CN31ari00000ECgs7402I09Wl0Xe172AffoG1e01d_tFcvogwEbHY06ZjgIfDP01Yih4oY-O0SIGuDKke06CoiJABwW1me_WrIwu0T3WzlKPm042s06oXOiLu07GlV8Hw05W-06Y-jw-0OW21A02eBxP5ha2pWe-jEXwx4Vm0lJpqfm3-0A2W820WA0JW0EqyEs838W3n_FGpG-O0vlD0h031EW4oWFu19ce0eW5cQW2a0MHsGEW1UvOg0MdIx05fqku1QHum0Mm-mp81V2b0z05yI7W1Lhm1G6O1jwfZXcW1ku2g0Rk0ia6nYkJMac0THFH1iQ_oD59W7KJsGO0001mSG000Aa7-54W7cJ6nKYm1u20a0pG1mBW1uOA-0S1W0W1q0YwYe21m9200k08aD-i2u0A0S4A00000000y3_O2WBW2e29UlWAmFmLY0i8gWiGVUKEdFXH003pf6TIMtK50C0BWAC5o0k0r9C1sGluKI0UPCR5IEWBcQW2y0i6Y0oY-jw-0UWC0fWDoR9dyWq0-Wq0WWu0YGu00000003mFv0Em8Gzc0x6rypKbOdWkXEW3i24FR0E0Q4F00000000y3-e3_YZqToqpF7YI000?stat-id=4&test-tag=84662424723457&format-type=24&banner-test-tags=eyI2Mzg2Mzg5NjM3IjoiODQ2NjIzOTUzNzE1MjAifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

Let's move on to the next, more serious type of transport layer connection - **TCP** ( **Transmission Control Protocol**, transmission control protocol). We have met repeatedly with this protocol. Therefore, although it is not an easy one, we learned it very well. We know how the connection is created and broken, we know how TCP packets are transmitted, how they are divided into segments. Therefore, we can already work with him and analyze certain emerging non-standard situations. Studying the stack of LWIP protocols, we also encountered the TCP protocol, even connected with it two controllers, organized data transfer between them. But all this was done using the RAW interface. Now it's time to work with this protocol using the NETCOON interface, which, as you know, works with the real-time FREERTOS system.

As a test card, we'll take **STM32F746-Discovery** , and we'll take the project for the prototype from [**lesson 120**](http://narodstream.ru/stm-urok-120-lan8742f-lwip-netconn-udp-server/) with the name **LAN8742\_UDP\_SERVER\_NETCONN** . Now assign it the appropriate name **LAN8742\_TCP\_SERVER\_NETCONN** and open it in the Cube MX.

Few adjust the **LWIP** settings   in the **Key Options** section  to match the client settings in future lessons





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 build the project and start working with the **main.c** file .

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

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

Our project will support two connections (or, as it is called, a socket). The difference between TCP and UDP is, among other things, that we can work with two connections on one server port. Therefore, in the structure for the socket, we will remove the port property, but add the TCP connection property

typedef struct struct\_sock\_t {

  uint16\_t y\_pos;

~~uint16\_t port;~~

**struct netconn \*conn;**

} struct\_sock;

Correct the name of the connection task function, removing almost all the code from it, except for just a few ads

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

{

  struct\_out \*qstruct;

  err\_t err, recv\_err;

  struct netconn \*conn;

}

Now go to the default task **StartDefaultTask** and also remove almost all user code from it, leaving only the initialization of the structures in terms of the vertical coordinate of the output of information on the display screen

/\* USER CODE BEGIN 5 \*/

**sock01.y\_pos = 60;**

**sock02.y\_pos = 180;**

/\* Infinite loop \*/

We declare here the pointer to the variable of the connection structure and the variable for the error

/\* USER CODE BEGIN 5 \*/

**struct netconn \*conn;**

**err\_t err;**

Create a connection

sock02.y\_pos = 180;

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

If everything is fine, then we will pass a pointer to this structure to both connections

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

**if(conn!=NULL)**

**{**

**sock01.conn = conn;**

**sock02.conn = conn;**

**}**

We link the structure, for example, with port 80, which will be useful to us for the future to create an HTTP server

sock02.conn = conn;

**err = netconn\_bind(conn, NULL, 80);**

If there are no errors, then start listening to the port, and if not - delete the structure

err = netconn\_bind(conn, NULL, 80);

**if (err == *ERR\_OK*)**

**{**

**netconn\_listen(conn);**

**}**

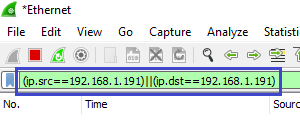
**else**

**{**

**netconn\_delete(conn);**

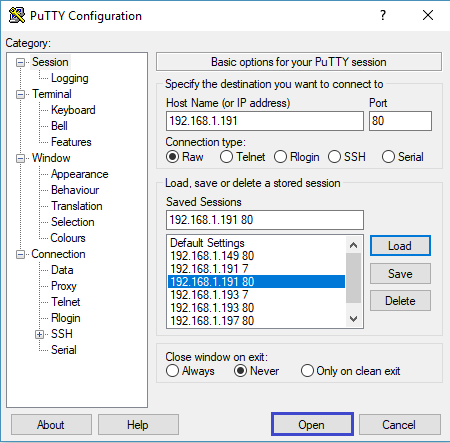
**}**

Launch the Wireshark traffic analyzer and filter the thread there by the IP address of our server



Now we can check the creation of the connection.

We'll collect the code, we'll run the controller and run the Putty network exchange program, in which we select the connection we created earlier. If anyone does not, then create something like that, in the screenshot, everything is there. Start the connection using the **Open** button



We see that the connection has been successfully established, a three-time handshake happened (click on the picture to enlarge the image)

[http://narodstream.ru/wp-content/uploads/2018/06/stm124img04_0500.png](http://narodstream.ru/wp-content/uploads/2018/06/stm124img04.png)

We close the connection window in Putty, the connection is broken, but only unilaterally. The server did not want to break, but we remember that everything here should be mutual (click on the image to enlarge the image)

[http://narodstream.ru/wp-content/uploads/2018/06/stm124img05_0500.png](http://narodstream.ru/wp-content/uploads/2018/06/stm124img05.png)

The desire of the server to break the connection, we will initiate later.

Also, in our connection, we can not send packets yet, we have not yet configured it, all this will happen in connection problems, which we will now create

netconn\_listen(conn);

**sys\_thread\_new("tcp\_thread1", tcp\_thread, (void\*)&sock01, DEFAULT\_THREAD\_STACKSIZE,*osPriorityNormal* );**

**sys\_thread\_new("tcp\_thread2", tcp\_thread, (void\*)&sock02, DEFAULT\_THREAD\_STACKSIZE,*osPriorityNormal* );**

Also here we passed a pointer to the connection structure.

Let's pass now a function for the given tasks **tcp\_thread** and create there a structure for the receiving buffer and one more connection structure intended purely for data exchange. Such a structure will be for each compound of two

struct netconn \*conn;

**struct netbuf \*inbuf;**

**struct netconn \*newconn;**

We catch the pointer passed to the task on the connection structure and send it to the local pointer

struct netconn \*newconn;

**struct\_sock \*arg\_sock;**

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

**conn = arg\_sock->conn;**

Create a variable to store the length of the buffer, as well as a pointer to a normal string buffer

conn = arg\_sock->conn;

**u16\_t buflen;**

**char\* buf;**

Change the color of the output of information on the display

char\* buf;

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

We organize an infinite loop, in which we associate the structure of the socket for data transfer with a common TCP structure

TFT\_SetTextColor(LCD\_COLOR\_BLUE);

**for(;;)**

**{**

**err = netconn\_accept(conn, &newconn);**

**}**

Then in this cycle we will check the success of our operation. In case of success, we end up in another infinite loop, in which we try to receive the packet and write it to the buffer, and in case of failure we will pass to another infinite loop with a delay, so as not to let it hang over the other processes of our program

err = netconn\_accept(conn, &newconn);

**if (err == *ERR\_OK*)**

**{**

**for(;;)**

**{**

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

**}**

**}**

**else**

**{**

**osDelay(1);**

**}**

And, when we wait for the package, we do not turn in this endless cycle, we "hang" waiting for the package and fail in the bottom only if it is received. We, of course, do not hang, at this time, there is a certain process of the library that allows other processes to work, because when we wait for the packet, other connections must also be freely transferred and received by us at this time.

Now that we have accepted the package and fell down, we will check the success of the packet and, if successful, take the data from the packet to our line buffer using a special function, and in case of failure we will free the memory of the receiving buffer structure and break the connection with the client, and also Let's get out of the loop.

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

**if (recv\_err == *ERR\_OK*)**

**{**

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

**}**

**else**

**{**

**netbuf\_delete(inbuf);**

**netconn\_close(newconn);**

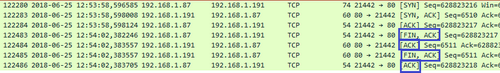
**break;**

**}**

Now we will have a separation with the client.

Let's check this by collecting the code, flashing the controller and starting the connection using the Putty program, and then breaking it by closing the connection window of this program.

We must now get this result (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2018/06/stm124img06.png)

Let's check the ending of the buffer for the line feed and carriage return. If there is one, then free the memory of the buffer structure and skip one iteration in the loop using the **continue** statement

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

**if((buf[0]==0x0D)||(buf[0]==0x0A))**

**{**

**netbuf\_delete(inbuf);**

**continue;**

**}**

And if this is not the end of the line, then we continue our cycle.

We will send the contents of the buffer to the display using a queue, zeroing a certain byte in the line buffer to create the end of the line

  continue;

}

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

**qstruct->y\_pos = arg\_sock->y\_pos;**

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

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

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

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

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

Now, in our line buffer, we will enter the line feed and carriage return codes and send this buffer back to the client, then free the buffer structure memory

osMailFree(strout\_Queue, qstruct);

**str\_buf[buflen] = '\r';**

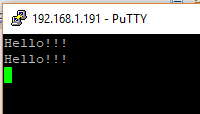
**str\_buf[buflen+1] = '\n';**

**netconn\_write(newconn, str\_buf, buflen+2, NETCONN\_COPY);**

**netbuf\_delete(inbuf);**

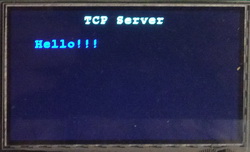
Actually, that's all and the code.

We'll collect it, we'll sew the controller, run the Putty program, open the port and try to transfer something to our server

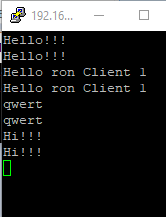


We see that the line from the server has returned to us successfully.

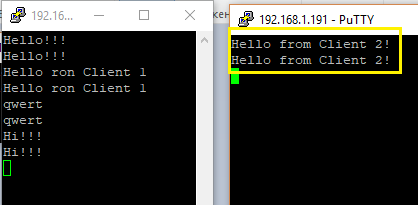
Also we see that the line also came to the display



Let's try to pass some more lines

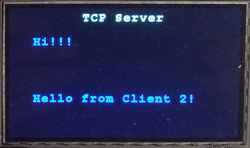


Create another connection from the client by running another Putty session, and check the lines from there as well

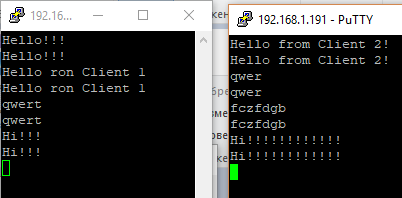


The data was also successfully transmitted and returned.

They also came in the form of a line on the display in another, lower position

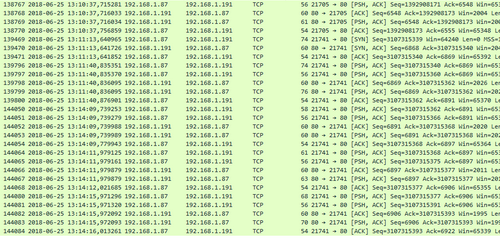


Let's try and pass a few more lines here

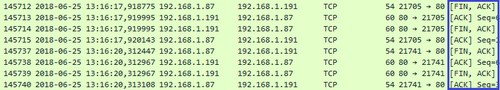


The data is also successfully transmitted and returned to where it is necessary, without being confused with each other.

Also we see that in the traffic analysis program the packets also pass normally (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2018/06/stm124img11.png)

We will close both sessions and see if the connection is correct (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2018/06/stm124img12.png)

So, in this lesson we created a simple TCP server that works correctly and correctly connects and disconnects clients, and also exchanges packets between nodes.

Thank you all for attention!