**STM Lesson 98. LAN8720. LWIP. TCP Server. Part 1**

Posted on [December 1, 2017](http://narodstream.ru/stm-urok-98-lan8720-lwip-tcp-server-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/)- [3 comments ↓](http://narodstream.ru/stm-urok-98-lan8720-lwip-tcp-server-chast-1/#comments)

[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/OB0T3Wm9Ecu50Du1CHoyari00000ECgs7402I09Wl0Xe172WmvR10O01sE3CgmU80QE8tP0fa06cakID99W1eiM-tYEW0SAapOqag07iWR3U8xW1tAUPz1N00GBO0PQmhH3W0UQKqmxe0HRu0RQLthu1Y083e0BUkAe2kGB_mxP8aC1TS_02ePlv8lW2We20W82m4e03vkAfaW680vAklhS8c0Exm0Qe0nAm0mIu1Fy1w0IE1FW4Wk01Y0M2u06G1Tpd0g05caEe1Veui0NwEBW5kbl01V2P1yW5iEe1q0NG6U051l050PW6WD2iymwW1ku2g0Rk0ia6nYkJMac0THFH1iQ_oD59W7KJsGO0001mSG000Aa7-54eK1I1n4Ym1u20a0pG1mBW1uOA-0S1W0W1q0YwYe21m9200k08hvoL0O0A0S4A00000000y3_O2WBW2e29UlWAmFmLY0i6gWiG_W-JRlXH001miHC1MtK50C0BWAC5o0k0r9C1sGluKIXG5874IEWBWk01y0i6Y0osbTw-0UWC0fWDbOneyWq0-Wq0WWu0YGu00000003mFv0Em8Gzc0x6rypKbOdWkXEW3i24FR0E0Q4F00000000y3-e3_YZqS-reg3YI000?stat-id=3&test-tag=80814134026241&format-type=24&banner-test-tags=eyI0ODQ2MjkwMDY3IjoiODA4MTQxMDQ2NzQzMDQifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

[Dating and chatting on web camerasRegistration and communication on the site are free. There is a guest entrance.To learn morehitlove.ru](https://an.yandex.ru/count/8gApx3-YAE850DW1CHoyari00000ECgs7402I09Wl0Xe1724kxc20O01nfYFzRESyguWY06-YR7-Av01j92OyYAO0UxNkFyYe06CvvxN0QW1gh7jhWAu0OYYjfC2m042s07gYUa2w04E-06s0OW21A02ykA82Ra2_yEsI930NNFm0kIt_Q43-0A2W820WB0IW0EOdTdG0eW3WxVyxmgO0_7i0h031BW4_m7e1Em3-0J-nm681Vx70P05yFW1e0N_EwW5x3Am1Umok0MZFy05ee03o0NPOT05y1RW1MBm1G6O1eQty7QW1ku2g0Rk0ia6nYkJMac0THFH1iQ_oD59W7KJsGO0001mSG000907Aga7-56ev3I1n4Ym1u20a0pG1mBW1uOA-0S1W0W1q0YwYe21m9200k08kR74180A0S4A00000000y3_O2WBW2e29UlWAmFmLY0i8gWiGdL9Qz_XH001wrXi1MtK50C0BWAC5o0k0r9C1sGluKQZaD874IEWB_iS1y0i6Y0oiljw-0UWC0fWD_BrcyWq0-Wq0WWu0YGu00000003mFv0Em8Gzc0x6rypKbOdWkXEW3i24FR0E0Q4F00000000y3-e3_YZiVIyfg3YI000?stat-id=4&test-tag=80814134026241&format-type=24&banner-test-tags=eyI0NjgzMTM4MDc0IjoiODA4MTQxMDQ2NzQzMDQifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

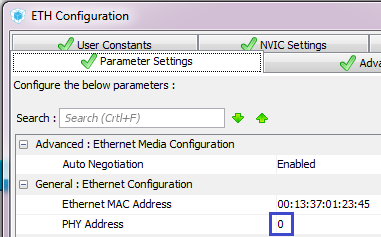
Today we will continue to work on programming the **LAN 8720** chip using the **LWIP** protocol stack library and now we will try to create with it a simple **TCP** server. To admit, I thought it would be much easier than writing a client, but it was not there. The thing is that the TCP structure that we save at the port listening stage is then lost after the client connects to us and I had to search for a long time how to catch it. And all because when we listen we do not know the addresses of the client who will connect to us. But then I guessed gradually how to catch all this. Maybe it's not right, but everything works correctly.

Well, let's begin!

The project will be done entirely from the [lesson](http://narodstream.ru/stm-urok-96-lan8720-lwip-tcp-client-chast-1/) project [96](http://narodstream.ru/stm-urok-96-lan8720-lwip-tcp-client-chast-1/)**LAN8720\_TCP\_CLIENT** and is named **LAN8720\_TCP\_SERVER** .

Run it in the **Cube MX** .

Since I decided to use the **STM32F4DIS-BB** board **today** , which is much more convenient than the module on the wires, I just change the address in the **ETH** interface



Generate the project and open it in the **System Workbench** .

We will configure it as usual, removing the debugging point and setting the optimization level 1.

Next we begin to edit the code.

Open the file **net.c** and fix the name of the function **tcp\_client\_connect** on**tcp\_server\_init**

**void** **tcp\_server\_init**(**void**)

Create a prototype for this function and call it in **main.c**

net\_ini();

**tcp\_server\_init();**

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

Remove the functions **port\_extract** , **ip\_extract** and **net\_cmd** .

Also, remove the call to **net\_cmd** from the **string\_parse** function . In this function, only this will remain

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

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

**{**

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

**sendstring(buf\_str);**

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

**}**

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

Let's fix the global structure

**static**struct tcp\_pcb \***server\_pcb;**

Also, further, we fix the global enumeration, the structure and the variable of its type

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

**enum server\_states**

**{**

***ES\_NONE* = 0,**

***ES\_ACCEPTED*,**

***ES\_RECEIVED*,**

***ES\_CLOSING***

**};**

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

**struct server\_struct**

**{**

**u8\_t state; /\* current connection state \*/**

**u8\_t retries;**

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

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

**};**

**struct server\_struct \*ss;**

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

In this code, little has changed. Simply names were named taking into account that at us now the server, and also statuses some began to be called in a little bit differently.

The functions **tcp\_client\_connected** ,**tcp\_client\_connection\_close** , **tcp\_client\_recv** , **tcp\_client\_send** , **tcp\_client\_sent,** and **tcp\_client\_poll,** we will delete. It's easier to remove them and write them again than to edit the code. Also remove their prototypes.

After the **tcp\_server\_init** function, we add a function that will be called when we instruct our server to listen to the connection

**static err\_t tcp\_server\_accept(void \*arg, struct tcp\_pcb \*newpcb, err\_t err)**

**{**

**err\_t ret\_err;**

**return ret\_err;**

**}**

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

Create a prototype for this function in the current file

struct server\_struct \*ss;

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

**static err\_t tcp\_server\_accept(void \*arg, struct tcp\_pcb \*newpcb, err\_t err);**

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

Next, replace the code in the function **tcp\_server\_init**

void tcp\_server\_init(void)

{

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

**if (server\_pcb != NULL)**

**{**

**err\_t err;**

**err = tcp\_bind(server\_pcb, IP\_ADDR\_ANY, 7);**

**if (err == *ERR\_OK*)**

**{**

**server\_pcb = tcp\_listen(server\_pcb);**

**tcp\_accept(server\_pcb, tcp\_server\_accept);**

**}**

**else**

**{**

**memp\_free(MEMP\_TCP\_PCB, server\_pcb);**

**}**

**}**

}

Here, too, everything is not difficult. The algorithm differs little from the algorithm that was in the case of client connection. Who looked at the lesson on the client, I think he will understand. Anyway, let's explain a bit what's going on here.

First, we create an instance of the TCP structure, in case of successful creation, we record using our special function in this structure our IP-address and our port address, let it be 7. Then, if everything is properly assigned, then we give a command to listen to our port, and then specify the structure of our handler function (or rather its address), which will be called if the client attempts to connect to our server. And before writing the body code of this function, let's add after its body the frames of the handler functions, which we will then need

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

**static void tcp\_server\_error(void \*arg, err\_t err)**

**{**

**}**

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

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

**{**

**err\_t ret\_err;**

**return ret\_err;**

**}**

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

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

**{**

**return *ERR\_OK*;**

**}**

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

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

**{**

**err\_t ret\_err;**

**return ret\_err;**

**}**

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

Also, create prototypes in the current file on these handlers

static err\_t tcp\_server\_accept(void \*arg, struct tcp\_pcb \*newpcb, err\_t err);

**static void tcp\_server\_error(void \*arg, err\_t err);**

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

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

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

From the **sendstring** function, **we** will remove all the code so that the compiler does not swear during the assembly, because it uses many old calls.

In principle, we can already collect the code, flash the controller and try to connect to the server. But let's first insert the code that will light the LED in the handler function **tcp\_server\_accept** to make sure that it lights up exactly when the client connection attempts

err\_t ret\_err;

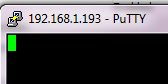
**HAL\_GPIO\_WritePin(GPIOD, GPIO\_PIN\_12, *SET*);**

return ret\_err;

Now try to collect the code, flash the controller, run the WIreShark program, filtering it at the IP address of our server, and also the **Putty** program , in which we will try to connect to our port. How to add the settings for connecting to the TCP server in Putty - I told you in one of my previous lessons. I think that you have already looked at all of them. Well, if someone thinks that he has already achieved more, then the Putty program is all the more familiar to him.

We try to connect to the server.

Connection was successful



We can also see this in the WireShark output window

http://narodstream.ru/wp-content/uploads/2017/11/image03-2.png

Excellent! All is correct and the LED is on.

Disconnect from the server by closing the Putty program. Let's see how we are going through the process of breaking the connection in WireShark

http://narodstream.ru/wp-content/uploads/2017/11/image04-2.png

Instead of four steps, the disconnection takes place in three steps. No, I did not forget to grab another line in the screenshot, it's just that our server does not know how to confirm the process of disconnection. Well, at least it responds with disconnection - and that's good! The rest will be cured a little later.

After the **tcp\_server\_accept** function . add the end of connection function

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

**static void tcp\_server\_connection\_close(struct tcp\_pcb \*tpcb, struct server\_struct \*es)**

**{**

**// remove all callbacks**

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

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

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

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

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

**if (es != NULL)**

**{**

**mem\_free(es);**

**}**

**tcp\_close(tpcb);**

**}**

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

In this function, we destroy pointers to handler functions and free the memory of the TCP structure.

For this function, we will also create a prototype

static err\_t tcp\_server\_accept(void \*arg, struct tcp\_pcb \*newpcb, err\_t err);

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

Now write the body of our function **tcp\_server\_accept** . Remove the code for turning on the LED from it and insert this code

err\_t ret\_err;

**struct server\_struct \*es;**

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

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

**tcp\_setprio(newpcb, TCP\_PRIO\_MIN);**

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

**ss = (struct server\_struct \*)mem\_malloc(sizeof(struct server\_struct));**

**if (es != NULL)**

**{**

**es->state = *ES\_ACCEPTED*;**

**es->pcb = newpcb;**

**ss->pcb = newpcb;**

**es->retries = 0;**

**es->p = NULL;**

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

**tcp\_recv(newpcb, tcp\_server\_recv);**

**tcp\_err(newpcb, tcp\_server\_error);**

**tcp\_sent(newpcb, tcp\_server\_sent);**

**tcp\_poll(newpcb, tcp\_server\_poll, 0);**

**ret\_err = *ERR\_OK*;**

**}**

**else**

**{**

**tcp\_server\_connection\_close(newpcb, es);**

**ret\_err = *ERR\_MEM*;**

**}**

return ret\_err;

In this function code, we first create an instance of the connection structure, assign priority to the connection. We allocate memory under structure, and also also under a similar structure, only global. This is the time for us to initialize it, since here the TCP structure that came in the input parameter is exactly the structure that we will need later to transfer the data at any time, rather than in the handlers. In the handlers, we already have the address of this structure. Then, if the memory is normally allocated (we will only check the local structure, in theory, we would need a global one, well, all right), then we assign the address of the incoming structure to our local and global address. And then we initialize the local structure. Then we assign to it the arguments of the incoming structure, and at the end of the condition body we specify the names of the handler functions.

We will not collect the code and flashing the controller yet. While we have not done anything for this yet.

After the function **tcp\_server\_recv,** we add the functions of sending the packet to the client

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

**static void tcp\_server\_send(struct tcp\_pcb \*tpcb, struct server\_struct \*es)**

**{**

**struct pbuf \*ptr;**

**err\_t wr\_err = *ERR\_OK*;**

**while ((wr\_err == *ERR\_OK*) &&**

**(es->p != NULL) &&**

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

**{**

**ptr = es->p;**

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

**if (wr\_err == *ERR\_OK*)**

**{**

**u16\_t plen;**

**u8\_t freed;**

**plen = ptr->len;**

**es->p = ptr->next;**

**if(es->p != NULL)**

**{**

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

**}**

**do**

**{**

**freed = pbuf\_free(ptr);**

**}**

**while(freed == 0);**

**tcp\_recved(tpcb, plen);**

**}**

**else if(wr\_err == *ERR\_MEM*)**

**{**

**es->p = ptr;**

**}**

**else**

**{**

**//other problem**

**}**

**}**

**}**

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

D This function has not changed much since the lesson of the client, so the code does not need explanation.

In the [**next part of**](http://narodstream.ru/stm-urok-98-lan8720-lwip-tcp-server-chast-2/) our lesson, we will write the bodies of all the handler functions, also write some more code and then test the operation of our server in practice.

**STM Lesson 98. LAN8720. LWIP. TCP Server. Part 2**

Posted on [December 1, 2017](http://narodstream.ru/stm-urok-98-lan8720-lwip-tcp-server-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/)- [4 comments ↓](http://narodstream.ru/stm-urok-98-lan8720-lwip-tcp-server-chast-2/#comments)

[10 eternal themes for paintingFree to participate in a series of webinars and get your favorite painting lessons!To learn morenewdirections.ru](https://an.yandex.ru/count/GrAGRMjZO8C50Cq1CIcyari00000ECgs7402I09Wl0Xe173mgPjnW07Cl-VExEgVyII80OF0sv4ra07CjhIyBvW1zgUrjowW0QAtjBmlg07WfhMtBhW1qE_sxXd00GBO0UQClmtW0UgOlWte0Go80WEW0fBxjnMv0jwY8JfN-bfiy0AxbVqg-0A2W820WA0IW0FjgkwX3eW3tEoP-mAe0nAm0mIu1Fy1w0JT2lW4gTS1Y0Mfrm6G1RRN0g05p42e1Tysi0NVDhW5frV01V2P1yW5iEe1q0NH6E05Ml050PW6skIsEQ06xWAe1ku2oGR6AvDQIO1r4z46nh_8qKc0THFP1W00071n0000gGVuKVGhCeJ4IB07W82G3D070k07XWhu1m60207G2BgAW870a802u0Z9_TiAW0e1mGe00000003mFzWA0k0AW8bw-0h0_1M82mQg2n2kJy-_-54004Sesm5RTGK0m0k0emN82u3Kam7P2_XHz2ioXCH8w0kfrm7m2mRe30AO3PMCQF8D0FeD088E08aE00000000y3-G3i24FPWEnjVCr9M9uBeJe0x3sXom3W6X3m0000000F0_g0_ue-7ac8QXuaW0?stat-id=3&test-tag=80264378212353&format-type=24&banner-test-tags=eyI2MzcxNTc3MjU0IjoiODAyNjQzNDg4NjA0MTYifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

[Programming for beginnersDownload the free book and become a real programmer. Get it now:To learn moresheremetev.info](https://an.yandex.ru/count/8VCdiR1RUza50Di1CIgyari00000ECgs7402I09Wl0Xe172WmvR10O01vE3CgmU80VY6tP0fa06YakID99W1Zhs-tYEW0VwIp8qag06M_w_U8xW1XgMPz1N00GBO0PQmhH3W0UQKqmxe0HRu0RQLthu1Y084e0BUkAe2kGBUeY4wL_fQRF02gfoUcmdu0eA0W820e1A00y-UfQuDY0Ezp-qMc0E9nGIm0mIu1Fy1w0Id2FW4su41Y0NRWG6G1Psi0g05zIQe1VqWi0Nz8BW5gap01V2P1yW5iEe1q0Nh3k05Ml050PW6WD2iymwW1ku2g0Rk0ia6nYkJMac0THFH1iQ_oD59W7KJsGO0001mSG000Aa7-54WGM-4n4Ym1u20a0pG1mBW1uOA-0S1W0W1q0YwYe21m9200k08hvoL0O0A0S4A00000000y3_O2WBW2e29UlWAmFmLY0i8gWiGORHu_VXH0038RUe1MtK50C0BWAC5o0k0r9C1sGluKI11RuJ4IEWBsu41y0i6Y0osbTw-0UWC0fWD_BrcyWq0-Wq0WWu0YGu00000003mFv0Em8Gzc0x6rypKbOdWkXEW3i24FR0E0Q4F00000000y3-e3_YZWP_dZQ7YI000?stat-id=4&test-tag=80264378212353&format-type=24&banner-test-tags=eyI0ODQ2MjkwMDY1IjoiODAyNjQzNDg4NjA0MTYifQ%3D%3D&)[Yandex.Direct](https://direct.yandex.ru/?partner)

In the [**previous part of**](http://narodstream.ru/stm-urok-98-lan8720-lwip-tcp-server-chast-1/) our lesson we added, and also replaced a number of service functions and tried to respond to the client's desire to create a connection with our server.

Let's start writing the body of the handler, which is called when the server receives the packet. Also create a pointer to the structure first

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

{

**struct server\_struct \*es;**

We will give her arguments

err\_t ret\_err;

**LWIP\_ASSERT("arg != NULL",arg != NULL);**

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

Well, we will now add different conditions for different cases of life

es = (struct server\_struct \*)arg;

**if (p == NULL)**

**{**

**}**

**else if(err != *ERR\_OK*)**

**{**

**}**

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

**{**

**}**

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

**{**

**}**

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

**{**

**}**

**else**

**{**

**}**

return ret\_err;

And now we begin to process these conditions.

First the body of the first condition

if (p == NULL)

{

**es->state = *ES\_CLOSING*;**

**if(es->p == NULL)**

**{**

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

**}**

**else**

**{**

**//acknowledge received packet**

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

**//send remaining data**

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

**}**

**ret\_err = *ERR\_OK*;**

}

If an empty packet arrives (usually a disconnect confirmation), we set the status to disconnection, if the buffer is empty, we give the command to send a disconnect confirmation, if not empty, then send the buffer to the server.

If we now collect the code and patch the controller, then when we try to disconnect, our server will already send the client a confirmation, but the flag for the disconnection will now not be sent, that is, everything is exactly the opposite

http://narodstream.ru/wp-content/uploads/2017/11/image05-2.png

It seems that one has been repaired, the other has been broken. But this may seem only at first glance. Now disengagement will occur more consciously, so we will fix also that "have broken".

Now the following condition

else if(err != *ERR\_OK*)

{

**if (p != NULL)**

**{**

**es->p = NULL;**

**pbuf\_free(p);**

**}**

**ret\_err = err;**

}

This is the case if the error status came, we pass it to the function return and release the buffers.

The following condition

else if(es->state == *ES\_ACCEPTED*)

{

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

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

**str1[p->len] = '\0';**

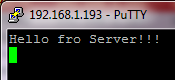
**HAL\_UART\_Transmit(&huart6, (uint8\_t\*)str1,strlen(str1),0x1000);**

**ret\_err = *ERR\_OK*;**

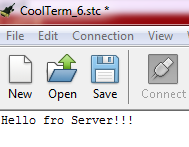
}

This is the case if a normal package comes up. We check the status, and after that get into the body of the condition, we take the data from the controller buffer into the structure buffer, then copy it into a string array, terminate with zero and display it in the terminal program.

You can collect the code again and flash the controller. Also, run the terminal program with the same settings as in the lesson on the TCP client, connect to the server and try to transfer the string from the command line Putty



Let's see if the line came to the terminal program



The line we received. Excellent. We disconnect from the server and write our code further.

We write the body of the following condition

else if (es->state == *ES\_RECEIVED*)

{

**if(es->p == NULL)**

**{**

**ret\_err = *ERR\_OK*;**

**}**

**else**

**{**

**struct pbuf \*ptr;**

**ptr = es->p;**

**pbuf\_chain(ptr,p);**

**}**

**ret\_err = *ERR\_OK*;**

}

This status is usually set when data is processed for the purpose of declaring a larger window. If the buffer is empty, then we do nothing if non-empty then we create a pointer to the buffer, pass it the buffer address of our local structure, and call a function that will bind our buffer to the buffer that came at the input to the function.

The following condition

else if(es->state == *ES\_CLOSING*)

{

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

**es->p = NULL;**

**pbuf\_free(p);**

**ret\_err = *ERR\_OK*;**

}

This is the condition when you want to handle the disconnection status. We send a confirmation, and zero all relevant pointers, and also free the buffer memory.

Well and last condition is when does not converge with the others, any special case

else

{

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

**es->p = NULL;**

**pbuf\_free(p);**

**ret\_err = *ERR\_OK*;**

}

In this case, we do the same.

Now we will write the body of the next handler, called in case of packet transfer - **tcp\_server\_sent**

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

{

**struct server\_struct \*es;**

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

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

**es->retries = 0;**

**if(es->p != NULL)**

**{**

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

**}**

**else**

**{**

**if(es->state == *ES\_CLOSING*)**

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

**}**

  return *ERR\_OK*;

}

The same is a pointer to the structure, we assign arguments, if the buffer is not empty, then we pass, if empty and plus if the status is on disconnection, then we are disconnected.

We write the body of the following function - tcp\_server\_poll. This is a handler, which is called after a certain time for monitoring the network, it does not matter whether there are packets that are currently transmitted to one side or another or not

err\_t ret\_err;

**struct server\_struct \*es;**

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

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

We will also create a structure, assign it a pointer to the arguments, and switch the state of the LED to ensure the timeliness of the calls of this handler. If we collect the code, we need to clean up the controller and connect to our server, then we will see that the red LED will flash in a period of half a second, because in the parameters, when we declared this function, we passed 0.

At the moment, even if we disconnect from the server, this LED will not stop flashing, because the server thinks that it has not yet disconnected from the client.

No, it will of course cease, but not soon, when the client gets tired of waiting for the disconnection commands and he will give the server the RST flag.

Write the rest of the code of our function

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

**if (es != NULL)**

**{**

**if (es->p != NULL)**

**{**

**}**

**else**

**{**

**if(es->state == *ES\_CLOSING*)**

**{**

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

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

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

**}**

**if(usartprop.is\_text==1)**

**{**

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

**}**

**}**

**ret\_err = *ERR\_OK*;**

**}**

**else**

**{**

**tcp\_abort(tpcb);**

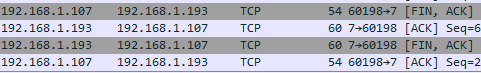
**ret\_err = *ERR\_ABRT*;**

**}**

return ret\_err;

If the structure is normally created, then we check that the buffer is not empty. If it is empty, then we do nothing, if non-empty, then we check the status, if we are in the disconnection stage, then we turn off the red LED, ignite the green and disconnect from the client, reset the text transmission status to the client and assign the result a successful value. Otherwise (if the structure was not created or the pointer is not received), we pass the RST flag to the client and leave with an error.

If we now collect the code, we'll run through the controller, connect to the server, and then disconnect, then we'll see that the disconnection process is correct



It remains for us only to teach the server to send the lines to the client at any time.

To do this, we write the body of the function of passing a string

void sendstring(char\* buf\_str)

{

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

**tcp\_sent(ss->pcb, tcp\_server\_sent);**

**tcp\_write(ss->pcb, (void\*)buf\_str, strlen(buf\_str), 1);**

**tcp\_recved(ss->pcb, strlen(buf\_str));**

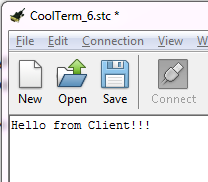
**usartprop.usart\_cnt=0;**

}

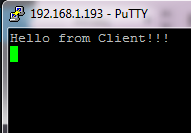
We first set the text transfer status, then we show our structure "expensive" to the data-handling function, then we write our string into the buffer of the TCP structure, then we perform the transfer. Well, at the end, reset the USART counter.

We will collect the code, let's solve the controller, connect to our server and try to alternate the lines from the client to the server and back. From the client to the server we have already transferred the data, so I will show the process of transferring from the server to the client.

To do this, we will enter a string in the terminal program and send it to the USART. The controller will process this line and first it will echo it back to USART



And then we'll send it to the server, which we see in the Putty window



In theory, for complete order, we must release and nullify our global structure when disconnecting from the client, but since we are still working with one connection, it is not necessary, I think, nothing will happen with the memory, because by and large - our structure only a pointer, it does not take memory of itself. Therefore, I consider the goal of the lesson to be achieved, with which I congratulate you! We created a simple server that is able to respond to the client's desire to create a TCP connection with it, also to disconnect, and also, most importantly, the server is able to receive and transmit data packets over the TCP protocol.

Thank you all for attention!