**Lesson 81**

**LAN. ENC28J60. UDP Client**

In the previous lesson we learned about the new **UDP** transport layer **protocol**and wrote a primitive echo server. Today we will set the reverse task - to write a client. As we already know from our practice, it is often more difficult for a client to write than a server, since we are accessing a network address without knowing the physical address and this imposes certain inconveniences, since we need to know this address first and then send it there are some packages on it. And, moreover, the package has to be created completely, which is usually not required in the server, it is enough to replace only some fields in the headers in the package that came. But the client is required more often and we can not get away from this. And UDP-client is also very popular. Later we will try to request from the Internet the current time with the help of such a client. But there will already be another problem. We are still working only on the local network, with the global will be a little different. But nevertheless we will manage all this.

In the meantime, let's create the project in the same way, as always, from the project of the [**last lesson**](http://narodstream.ru/stm-urok-80-lan-enc28j60-udp-server/)**ENC28J60\_UDPS** . The project will be called **ENC28J60\_UDPC** . Only one letter has changed, since we now have a client.

Run the project in Cube MX and, without touching anything, generate the project and run it in Keil. Then, as usual, we connect our libraries to the project and configure the programmer for autorestart.

Let's go first to **net.h** and increase the line for the input line from the terminal program a little, because in this line we should now write another port, and it sometimes happens even five-character

typedef struct USART\_prop{

**uint8\_t usart\_buf[25];**

Accordingly, in the file  **net.c, we**  also make an adjustment in the interrupt service function from USART

//если вдруг случайно превысим длину буфера

**if(usartprop.usart\_cnt>25)**

We will send **UDP-** packages, as you understand, also from the terminal line, only after the IP-address we will write the port number through the colon ( **:)** and the line will end not by the letter **a** , but by the letter **u** for obvious reasons.

In the same function we add one more condition

  net\_cmd ();

}

**else if (b == 'u' )**

**{**

**usartprop.is\_ip = 2; // status of trying to send a UDP packet**

**net\_cmd ();**

**}**

In the file **net.c** (since we are still here) we will remove all the commented code from all functions, so as not to get in the way, since it is hardly necessary for us.

Let's go to the function **net\_cmd** and while we delete all the output of information in USART

~~HAL\_UART\_Transmit(&huart1,usartprop.usart\_buf,usartprop.usart\_cnt,0x1000);~~

~~HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"\r\n",2,0x1000);~~

ip\_extract((char\*)usartprop.usart\_buf,usartprop.usart\_cnt,ip);

We add here one more condition

  usartprop.usart\_cnt=0;

}

**else if(usartprop.is\_ip==2)//статус попытки отправить UDP-пакет**

**{**

**ip\_extract((char\*)usartprop.usart\_buf,usartprop.usart\_cnt,ip);**

**usartprop.is\_ip=3;//статус отправки UDP-пакета**

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

**arp\_request(ip);//узнаем mac-адрес**

**}**

After processing the line with the IP address, we set a different status, also reset the USART byte counter, well, we will also send the ARP request.

Now go to the function **ip\_extract** and there also we **'ll** rework the code a bit so that it also feels the colon. Add another variable with this symbol, and also change the name of another variable a little

int ch**1** = '.';

**int ch2 = ':';**

We will correct the name in the code as well

ss1 = strchr(ip\_str,ch**1**);

After the body of the **for** loop, we recognize the occurrence of our symbol

  len-=offset;

}

**ss1=strchr(ip\_str,ch2);**

Next, using the condition, we know if we have a colon, extract a string with the last byte of the IP address in its body, convert it to a number, and exit the function, and we will not execute any further code

ss1=strchr(ip\_str,ch2);

**if (ss1!=NULL)**

**{**

**offset=ss1-ip\_str+1;**

**strncpy(ss2,ip\_str,offset);**

**ss2[offset]=0;**

**ipextp[3] = atoi(ss2);**

**return;**

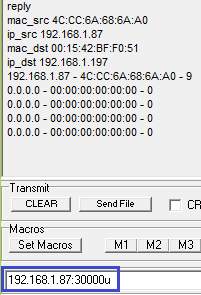
**}**

strncpy(ss2,ip\_str,len);

Let's check the work of this code. We will collect the code, we will sew the controller, start the terminal program. and also the **WireShark** utility , just let's filter the output now not by IP, but by **MAC** . This is done as follows

Image00

In the terminal program, we try to enter the following line



As we can see, everything worked out and the ARP package also went off and the computer responded to it. You can also check that we have not stopped working on the ARP request in the old way, without the port and with the letter **a** .

Also we see that in the WireShark utility we also see sent and received ARP packets

Image02

Now we need to extract a port from the string.

Create for this purpose a function that will do this. Since the procedures in the body of the function are similar to the previous one, I immediately give it with all the code

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

**uint16\_t port\_extract(char\* ip\_str, uint8\_t len)**

**{**

**uint16\_t port=0;**

**int ch1=':';**

**char \*ss1;**

**uint8\_t offset = 0;**

**ss1=strchr(ip\_str,ch1);**

**offset=ss1-ip\_str+1;**

**ip\_str+=offset;**

**port = atoi(ip\_str);**

**return port;**

**}**

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

But we will not call this function yet, because it's too early. And early because there will not be any sense anyway, since we are currently in the USART reception handler and by definition we have not yet accepted the **ARP** response from the node and therefore we do not know the MAC address, and moreover, maybe we have it already in the ARP table. Here such an ambush is formed. We will need to promptly leave the handler and wait for an answer, and then return to the **net\_cmd** function in time . Well, nothing, we all razrulim. Where did ours not disappear?

Well, firstly, to return once to our function, we found there our data uninstalled, so to speak did not return to the broken trough, for this we have two ways. Make a variable with this data global, or assign static. Let's go along the second path, since we already have enough global variables. While we only have ip, so at the same time we add the second variable

void net\_cmd(void)

{

**static** uint8\_t ip[4]={0};

**static uint16\_t port=0;**

We will also prepare a condition for the next status, so that we could go back

    arp\_request(ip);//узнаем mac-адрес

  }

**else if(usartprop.is\_ip==3)//статус отправки UDP-пакета**

**{**

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

**}**

}

Now we will think about where we should come from here, since there is a condition before us - at this moment we must necessarily know the mac-address.

But before we do this, we must find another ambush (and I did warn that the client is more difficult, since I did not promise to have an easy life, well, for that, I'm here).

The second ambush. In the **eth\_send** function **,** we simply change the MAC addresses of the receiver and source before destroying the packet to the node. But we do not always have the source address to become the address of the receiver. Just sending a UDP packet from the client is just this case. We did not receive exemplary UDP packets and we have no source. Therefore, we remove the first row from this function

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

{

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

But now we have the responsibility for the address of the receiver in all other cases that we used in previous lessons, we did not specify it in advance, so we'll have to do it now. Well it will not be difficult. We have the string that is now deleted, we will add it in certain places before calling our function of sending the Ethernet packet. Well, or a little change it. Well, we'll start looking for these places and, of course, after each insertion of the line into the code, we will check the former performance that we have now destroyed. But all this of course for good purposes.

We will remember all over again how we created our project and how it developed in our country, which ones we sent first packages, which later.

First we answered the ARP request. Therefore, go into the function **arp\_send** in the file **arp.c** and add our line before sending the package

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

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

Well, and how to check the efficiency of the ARP response and the rest of our functions, I will not show it, I hope everyone remembers it.

Then we answered the **ICMP** request. In the function **icmp\_read we** also **insert** our line in the proper place, because the function ip\_send to us with such things can not be littered, since it is necessary to come to it with the right address, because it will be formed differently everywhere, and we do not want to produce different cases, so we'll come there already prepared for sending the package

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

ip\_send(frame,len+sizeof(ip\_pkt\_ptr));

Next, we sent an ARP request. And it happens in the function **arp\_request** file **arp.c** . Therefore, go to the appropriate file and first also clear it of the commented code, and then go to the function **arp\_request**and ... we will not touch anything there, because there we directly call the function **enc28j60\_packetSend** , as they say, without intermediaries.

Well, also, in the last lesson, we sent a **UDP** packet in response to an echo reply. This occurred in the function **udp\_reply** file **udp.c** . Let's add the code in this function

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

ip\_send(frame,len+sizeof(ip\_pkt\_ptr));

Wow! three times they wrote the same thing. Well, okay. But it's easy to do. I hope that everyone has checked up the working capacity, which now, as we were convinced, was not injured.

Well, now we'll start slowly to fulfill our main task: "Return to the net\_cmd function from the right places and send out the UDP packet, armed with this value of the IP address and the port number of the receiver.

For the **net\_cmd** function,  **we** will create a prototype in **net.h** and look for a place to return to this function.

It turns out that there is such a place and it does not hide very far, but very near. This function **eth\_read** file **net.c** . After calling the function filling the ARP-table, we will return to our function **net\_cmd** , but only if we have the appropriate status

else if(res==2)

{

  arp\_table\_fill(frame);

**if(usartprop.is\_ip==3)//статус отправки UDP-пакета**

**{**

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

**net\_cmd();**

**}**

}

Also we see that we will not return empty-handed, but with a MAC address.

But this is not enough. This is one case. This is when we filled the table. And if we already had a record? Then there is another function. And the function is **arp\_request** in the file **arp.c** , which we will now pass on to, and before we call our beloved **net\_cmd** , we'll make some adjustments here, after working on the errors. We use the same variable for counting the cycles in two cycles at once, although one cycle is nested in the other. I still do not understand how the code worked. We will correct this situation by adding one more variable

uint8\_t arp\_request(uint8\_t \*ip\_addr)

{

  uint8\_t i**,j**;

Next, replace the variable in all the places in the main loop (allocated, as usual, in bold)

for(**j**=0;**j**<5;**j**++)

{

  //Если записи уже более 12 часов, то удалим её

  if((clock\_cnt-arp\_rec[**j**].sec)>43200)

  {

    memset(arp\_rec+(sizeof(arp\_record\_ptr)\***j**),0,sizeof(arp\_record\_ptr));

  }

  if(!memcmp(arp\_rec[**j**].ipaddr,ip\_addr,4))

  {

  //смотрим ARP-таблицу

We will transfer the code for setting the pointer to the packet at the beginning of the function

uint8\_t i,j;

**enc28j60\_frame\_ptr \*frame=(void\*)net\_buf;**

Connect the global structure variable to the file **arp.c**

uint8\_t current\_arp\_index=0;

**extern USART\_prop\_ptr usartprop;**

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

Well and now in a necessary place in function **arp\_request we will**  call our **net\_cmd**

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

  }

**memcpy(frame->addr\_dest,arp\_rec[j].macaddr,6);**

**if(usartprop.is\_ip==3)//статус отправки UDP-пакета**

**{**

**net\_cmd();**

**}**

  return 0;

}

**Go to the udp.c** file and create a function to send the **UDP** package at the top after the global ads

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

**uint8\_t udp\_send(uint8\_t \*ip\_addr, uint16\_t port)**

**{**

**uint8\_t res=0;**

**return res;**

**}**

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

We will write a prototype in the header file for this function and in the **net.c** file in **net\_cmd** function we call this function, having learned before the port number

else if(usartprop.is\_ip==3)//статус отправки UDP-пакета

{

**port=port\_extract((char\*)usartprop.usart\_buf,usartprop.usart\_cnt);**

**udp\_send(ip,port);**

  usartprop.is\_ip=0;

}

Well, now we still have the most important job - to write the code of the **udp\_send** function , which is what we will do, going into it in the file **udp.c**

And before we move, add a global buffer array

extern char str1[60];

**extern uint8\_t net\_buf[ENC28J60\_MAXFRAME];**

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

And now let's move on

Create a variable for the length and get pointers to all the required packages

uint8\_t res=0;

**uint16\_t len;**

**enc28j60\_frame\_ptr \*frame=(void\*) net\_buf;**

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

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

In the **udp.h** file, **we** set the variable for the permanent local port of our application in the controller

#include "net.h"

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

**#define LOCAL\_PORT 333**

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

You can have any value of your own.

Let's return to our function of sending the UDP package and fill in the header fields of the source and receiver ports, remembering that these values ​​in the fields are stored in the format **big endian**

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

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

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

We will enter some string in the data field and after we learn the length of the UDP packet

udp\_pkt->port\_src = be16toword(LOCAL\_PORT);

**strcpy((char\*)udp\_pkt->data,"UDP Reply:\r\nHello to UDP Client!!!\r\n");**

**len = strlen((char\*)udp\_pkt->data) + sizeof(udp\_pkt\_ptr);**

Enter the length in the corresponding header field

len = strlen((char\*)udp\_pkt->data) + sizeof(udp\_pkt\_ptr);

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

Then zero the field with the checksum and, calling the necessary function, we enter there a new checksum

udp\_pkt->len = be16toword(len);

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

**udp\_pkt->cs=checksum((uint8\_t\*)udp\_pkt-8, len+8, 1);**

With UDP-package like everything. Now wrap it in the IP header, this is the order that you can do. To do this, fill in the necessary IP header fields

udp\_pkt->cs=checksum((uint8\_t\*)udp\_pkt-8, len+8, 1);

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

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

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

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

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

Next, the IP packet is wrapped in the Ethernet header, for which we need to fill in its fields. We already have the MAC-address of the source, and the address of the receiver is entered in the function of sending the Ethernet packet. Therefore, we can only fill the package type

ip\_pkt->verlen = 0x45;

**frame->type=ETH\_IP;**

Well, of course, do not forget to call the function of sending the package

  frame->type=ETH\_IP;

**ip\_send(frame,len+sizeof(ip\_pkt\_ptr));**

  return res;

}

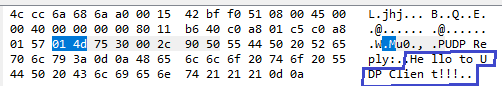
We'll collect the code, we'll tell the controller, and try to send the packet from the terminal program, it's better 2 times, since the first time the function is called in a separate place, and the second time in another

Image03

Make sure that our packages came safely

Image05

Great, the packages all came down. Make sure, also that they have reached completely



You can also make sure of the netcat utility by sending the UDP packet to our controller first (I tell the controller, because it is not already a module.) We already send packets to the application, because we use the port number and the transport protocol). This is necessary to find out the port to which packets are sent first.

Image07

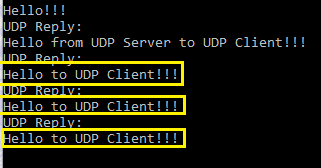
The package went off, the controller replied. We learn in the terminal program the port to which we will send the packet

Image08

And now we will send the package to this port several times

Image10

And the packages came, though for some reason not all, but it's probably some expense of the utility



So, today we learned how to send UDP packets to a specific node using a certain port, thus creating a UDP client, although this was not as easy as it seems at first glance. All this speaks for the growing qualification of us, which confirms the fruits of previous studies, that they were not in vain. In the next lesson, we will try to send some packages not to the local network, but to the global ones. I think, and we will definitely succeed!