**Lesson 80**

**LAN. ENC28J60. UDP Server**

Well, the time has come for us to transmit through the local network, through our module **ENC28J60** , any conscious data. For this, we need to wrap our data in some transport protocol. The transport layer is the level of the **OSI model** that is responsible for transferring data to a specific application of the node, rather than just a node. For this, there are ports and the transport layer is already transferring data to the network address, addressing them to a specific port.

There are two main transport protocols: **TCP** and **UDP** . The first one is more complicated, requires connection, verification, but it supports the transfer of large amounts of data, combining packets into segments, as well as into windows. Also **TCP** supports guaranteed delivery of data packets, since after transmission of the latter we are waiting for confirmation from the data receiver. **UDP** does not have such a "power", here we already transmit data in short packets (datagrams), which is why it is called the User Datagram Protocol, and does not require confirmation.

But in connection with this, this protocol has a simplicity of implementation, and also because of the uselessness of the confirmations and the greater speed and continuity. And which of these protocols you will use according to your tasks - it's up to you.

We will follow the path of consistent development and consider for the time being a simpler protocol, **UDP** .

The **UDP** header is very simple (only 4 fields) and has the following appearance:



**The port of the poisoner** is the value of the data source port,

**The destination port** is the value of the data receiver port,

**The length of UDP**  is the number of bytes in the entire datagram, including the header,

**The cash amount** is a calculated checksum in the same way, as in the case of the protocols we have already considered, only the bytes of not only the header, but also the data are considered. Moreover, the heading takes into account not only the UDP protocol, but also something else attached to it from above, and all this is called a **pseudo** - **header** . There are also:

The source IP-address, the IP-address of the receiver, zero, you can not count zeros, the protocol identifier (in our case 17 - the UDP ID), as well as the length of the UDP packet along with the data, that is, the same value as in the third field itself the UDP header.

If the header is structured, it will look something like this



Slightly everything is confusing, but it's so accepted and you can not get away anywhere, otherwise our package will not be identified and runs the risk of not being allowed through any firewall. So the checksum must converge.

Well, nothing, we'll write the code - we'll figure it out.

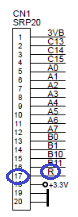
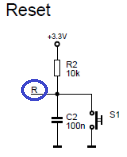
That's exactly what we're going to do right now, enough theory, it's time to rest from it.

The project is based on the [**lesson**](http://narodstream.ru/stm_urok-77_lan_enc28j60_podkljuchaem_vneshnie_preryvanija_exti/) project [**77**](http://narodstream.ru/stm_urok-77_lan_enc28j60_podkljuchaem_vneshnie_preryvanija_exti/) called **ENC28J60\_INT** and is named **ENC28J60\_UDPS** .

Open this project in **Cube MX**and, without touching it, we generate a project, open it in Keil, connect all our libraries, and also configure the programmer for auto-cutting. We will try to assemble the project, flash it and test the efficiency of pings and our ARP-requests in the terminal program.

If everything works, then we will continue to write the code.

But first of all it's worth mentioning that the **RESET** contact from the module was connected to the board's **R** leg , which, judging by the circuitry of the board, is connected to the **RESET** button and also pulled up to the power bus through a 10 kilo-ohm resistor, which frees us from installing an additional resistor and. most importantly, when the controller is rebooted, the LAN module can synchronously restart. Also, this leg is connected, of course, with the foot of the **NRST**controller

  image04-1

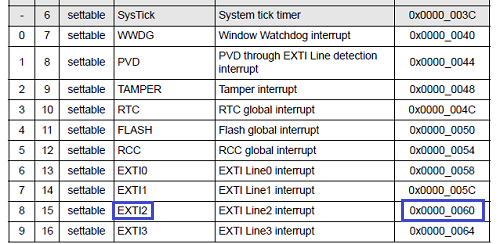
Well, now let's move on to the code.

First, let's adjust our project a little.

We do not quite properly organize the processing of the network.

We all do interrupts from the INT knob in one handler, and we also send arp requests from the terminal program right there. This is a mess. And suddenly at this time we generally do not get the package and we will not get into this handler then. We risk not sending our request for a very long time. So I decided to send it directly from the USART interrupt handler on the receive. But here's the question. And suddenly at this time we will process the request from INT? Nothing wrong. USART is patiently waiting for its turn, that is, interruptions are very tactful and do not get into other people's business.

Well, if that is even steeper, these two types of interrupts will occur simultaneously. What then? But nothing. In our controller this is thought out and for this there are interrupt priorities. At whom the priority is higher, then it will be processed. And who is he higher? Also a question. There's an answer. And for whom the address of the vector is smaller, it is steeper, and the priority is higher. Here is the table of interrupt vectors **STM32F103x8,**which are presented in the **Reference manual .**We only need **EXTI** and **USART** , so I will not show the whole table because it is much larger than the AVR controllers





In general, as we can see, the external interrupt priority is higher. Therefore, if you need to receive the packet and process the reception of USART, we will first receive the packet, so everything is fine.

True, we still have a **sixth timer** , but its priority is even lower than that of **USART1** (vector **0x0000\_0118** ). But, since the mission timer is not the most important, it only considers pseudo-seconds for ARP-table recording time, then everything is fine too.

And if suddenly we do not like the priorities, then we can easily change them in the Cube MX and assign there others for our interrupts. But since we do not need to do this, and we already faced changing priorities, in order to save space on the page, we will not consider it, and it is not entirely on the topic.

So, a little bit so we'll rework the code.

Let's comment in net.poll net.poll everything related to sending the ARP package from the USART

void net\_poll(void)

{

  uint16\_t len;

**// uint8\_t ip[4]={0};**

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

  while((len=enc28j60\_packetReceive(net\_buf,sizeof(net\_buf)))>0)

  {

    eth\_read(frame,len);

  }

  /**/ if(usartprop.is\_ip==1)//статус отправки ARP-запроса**

**// {**

**//   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);**

**//   arp\_request(ip);**

**//   usartprop.is\_ip = 0;**

**//   usartprop.usart\_cnt=0;**

**// }**

}

And below add a new function and add to it all this code, which we commented above

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

**void net\_cmd(void)**

**{**

**uint8\_t ip[4]={0};**

**if(usartprop.is\_ip==1)//статус отправки ARP-запроса**

**{**

**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);**

**arp\_request(ip);**

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

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

**}**

**}**

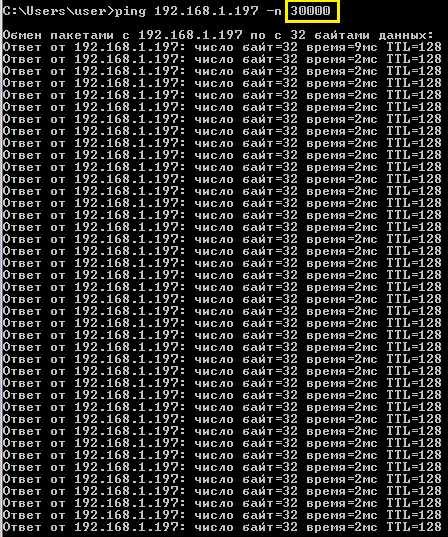
**//-----------------------------------------------**

Call this function in the USART handler below

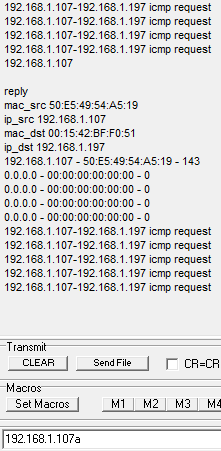
else if(b == 'a')  
{  
  usartprop.is\_ip=1;//статус отправки ARP-запроса  
**net\_cmd();**  
}

That's all, it was possible in general to remove the structure of the flag and the type of requests to pass the argument, but for now, let's leave it for the future, whether there will be few troubles.

We'll collect the code and check it all. For completeness, let's start an infinite ping. Well, it is not more true than infinite, but we will set a large number for pings to be sent constantly, not 4 pieces



And at this time we will send an ARP request from the terminal program



Everything works fine. So that's great!

Now let's get down to the topic of the lesson - **UDP** .

Create two more files - **udp.c** and **udp.h** with the following contents

**udp.h** :

**#ifndef UDP\_H\_**

**#define UDP\_H\_**

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

**#include "stm32f1xx\_hal.h"**

**#include <string.h>**

**#include <stdlib.h>**

**#include <stdint.h>**

**#include "enc28j60.h"**

**#include "net.h"**

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

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

**#endif /\* UDP\_H\_ \*/**

**udp.c** :

**#include "udp.h"**

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

Add the function of reading UDP packets to the **udp.c** file of standard content

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

**uint8\_t udp\_read(enc28j60\_frame\_ptr \*frame, uint16\_t len)**

**{**

**uint8\_t res=0;**

**return res;**

**}**

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

Let's create a prototype for this function.

Let's connect the global string array

#include "udp.h"

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

**extern char str1[60];**

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

In the **net.h** file **below** connect our udp library

#include "arp.h"

**#include "udp.h"**

In the file **net.c** in function **ip\_read we** call our function in the body of the corresponding condition

else if(ip\_pkt->prt==IP\_UDP)

{

**udp\_read(frame,len);**

}

Let's return to the **udp.c** file and connect the global variable

#include "udp.h"

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

**extern UART\_HandleTypeDef huart1;**

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

In the function **udp\_read** we add the display in the terminal program of some header fields of packages, adding a link to the IP packet

uint8\_t res=0;

**sprintf(str1,"%02X:%02X:%02X:%02X:%02X:%02X-%02X:%02X:%02X:%02X:%02X:%02X; %d; ip\r\n",**

**frame->addr\_src[0],frame->addr\_src[1],frame->addr\_src[2],**

**frame->addr\_src[3],frame->addr\_src[4],frame->addr\_src[5],**

**frame->addr\_dest[0],frame->addr\_dest[1],frame->addr\_dest[2],**

**frame->addr\_dest[3],frame->addr\_dest[4],frame->addr\_dest[5],len);**

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

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

**sprintf(str1,"%d.%d.%d.%d-%d.%d.%d.%d udp request\r\n",**

**ip\_pkt->ipaddr\_src[0],ip\_pkt->ipaddr\_src[1],ip\_pkt->ipaddr\_src[2],ip\_pkt->ipaddr\_src[3],**

**ip\_pkt->ipaddr\_dst[0],ip\_pkt->ipaddr\_dst[1],ip\_pkt->ipaddr\_dst[2],ip\_pkt->ipaddr\_dst[3]);**

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

Now it's time for the **UDP** package .

In the **udp.h** file **,** add a structure for it

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

**typedef struct udp\_pkt {**

**uint16\_t port\_src;//порт отправителя**

**uint16\_t port\_dst;//порт получателя**

**uint16\_t len;//длина**

**uint16\_t cs;//контрольная сумма заголовка**

**uint8\_t data[];//данные**

**} udp\_pkt\_ptr;**

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

Back in our function **udp\_read** file **udp.c** and similarly displayed in the terminal program, UDP ports, as well as the length of the datagram. Well, at the same time we will display the incoming data as a string

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

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

**sprintf(str1,"%u-%u\r\n", be16toword(udp\_pkt->port\_src),be16toword(udp\_pkt->port\_dst));**

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

**HAL\_UART\_Transmit(&huart1,udp\_pkt->data,len-sizeof(udp\_pkt\_ptr),0x1000);**

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

Let's collect the code, we'll tell the controller and try to send the UDP package using the **netcat** utility , the Windows version of which is not difficult to find on the Internet.

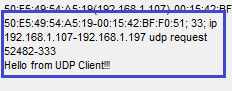
Before that, we'll run even the command line and the terminal program in addition to the Wireshark network packet analysis utility, having set up a filter for our module

image09

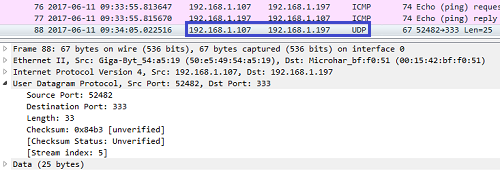
Let's pass some string to our module

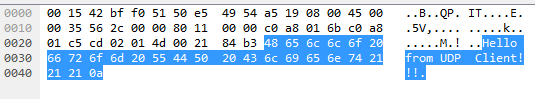
image10

Here is what we will see in the terminal program



And this is in Wireshark





It follows that the package went to us, and moreover, we got it.

Now we need to somehow answer this.

Add the response function to the UDP request

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

**uint8\_t udp\_reply(enc28j60\_frame\_ptr \*frame, uint16\_t len)**

**{**

**uint8\_t res=0;**

**return res;**

**}**

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

Call it in the function **udp\_read**

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

**udp\_reply(frame,len);**

return res;

We initialize in the **udp\_reply** function  pointers to packets and change the ports in reverse (source and destination)

uint8\_t res = 0;

**uint16\_t port;**

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

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

**port = udp\_pkt-> port\_dst ;**

**udp\_pkt-> port\_dst = udp\_pkt-> port\_src;**

**udp\_pkt-> port\_src = port;**

Add your own string as UDP package data

udp\_pkt->port\_src = port;

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

We re-measure the length of the packet, since the data field most likely changed and we put it in the corresponding header field

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

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

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

Do not forget about the checksum. You can of course calculate and simplify it by adding or subtracting the change in the length of the data, but we will still write a universal algorithm.

As the above, it was stated that the checksum in the **UDP** -Package is calculated in a special way, the vnesom some amendments in its calculation of the function in the file **net.c** .

First, we add one more argument of the package type. So far we agree that all the packets considered earlier will be of type **0** , and the **UDP** packet will be **1**

uint16\_t checksum(uint8\_t \*ptr, uint16\_t len**, uint8\_t type**)

{

We also need a prototype for this function, and we also need a prototype of the function **ip\_send** . Add them to **net.h**

void TIM\_PeriodElapsedCallback ( void );

**uint8\_t ip\_send (enc28j60\_frame\_ptr \* frame, uint16\_t len);**

**uint16\_t checksum ( uint8\_t \* ptr, uint16\_t len, uint8\_t type);**

We return to **net.c** and, since we changed the composition of the incoming arguments in the function of calculating the checksum, we add in the two places in the calls of this function the third argument

ip\_pkt->cs=checksum((void\*)ip\_pkt,sizeof(ip\_pkt\_ptr)**,0**);

icmp\_pkt->cs=checksum((void\*)icmp\_pkt,len**,0**);

In addition to the third argument in the function of calculating the checksum, add a response to **type 1**

uint32\_t sum=0;

**if(type==1)**

**{**

**sum+=IP\_UDP;**

**sum+=len-8;**

**}**

while(len>0)

I will explain a little, but it is immediately unclear. We add the protocol type identifier (at the moment 17), the length reduced by 8, since we will transfer the length increased by 8. This trick is necessary in order to hook along with the UDP header when transferring also the IP addresses that are At the end of the IP header, which precedes the UDP header in the packet. Like this!

Well, now let us return to the file **udp.c**  function **udp\_reply** and calculate checksum there, in advance of its zeroing

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

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

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

Accordingly, we move the pointer to the left, catching the network addresses of the source and the receiver from the IP header, but the length, we obviously increase by 8.

It remains for us to only send the packet to the recipient

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

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

return res;

We'll collect the code, we'll write the controller and send the line to the module again from the netcat utility. If everything is normal, then we get there an answer from the module of this kind

image15

Let's see our exchange also in the Wireshark network packet analysis utility

image16

As we can see, everything is normal and everything has been accepted. If there was something wrong, Wirechark would write a checksum error. So we are all normally calculated.

Thus, now we can receive and transmit UDP requests, which allows us to transmit conscious data (payload).