**Lesson 68**

**Part 1**

**LAN. ENC28J60**

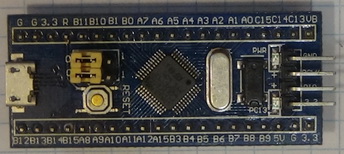
Today we connect the controller to the local network using a module on the **ENC28J60** chip  .

Since we  only used the **LAN** interface using the **AVR microcontroller** , I decided to make a similar lesson on**STM32** ., And also using the same module, since this module allows you to experience this or that **LAN** interface  from the inside, since for it STM ready there are no libraries.

This module looks like this



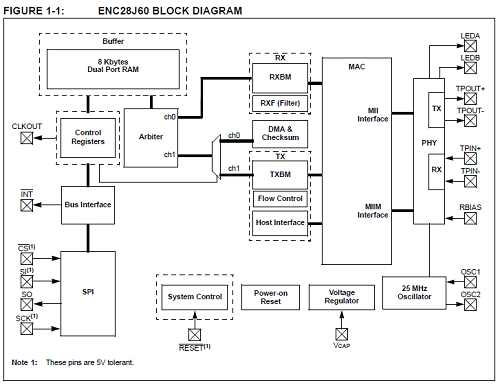
As the microcontroller was selected **STM32F103RCT6** , which I still did not give lessons, but unpacking among my videos on Youtube is available. But since there are no lessons, we will create the project anew. The controller is installed on a debug board that looks like this (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/03/image01.jpg)

To program our debug board we will use the ST-Link-V2 programmer, which is also unpacked, connected via the SWD interface. How to connect this programmer, there is also a video for unpacking [**here**](https://youtu.be/0YVPIruMz_s) .

The heart of this module is the same name chip **ENC28J60** , which is a ready-made network solution, which has both a physical and a channel level. Data exchange with the controller this chip carries out via the **SPI** bus . The physical layer is organized according to the **10BASE-T standard** . That is, the maximum data transfer speed is 10 megabits per second, which, although small by today's modern standards of data transfer speeds over the network, is enough for the microcontroller, and I think it will be more convenient than using USB, because, being on the network and having its own address and channel and network, the controller can access any computer on the local network as well as in the global one, which is relatively good.

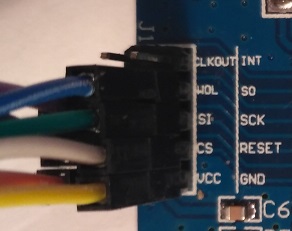
The internal structure of the chip is the following view (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/03/image02.png)

We see here that we have control registers, in which we will send certain commands to manage certain actions and settings. We also see a buffer of 8 kilobytes for receiving and sending data over the network.

The chip is delivered in various cases. On our module, it is housed in a housing designed for surface mounting.

To connect the module there is the following connector



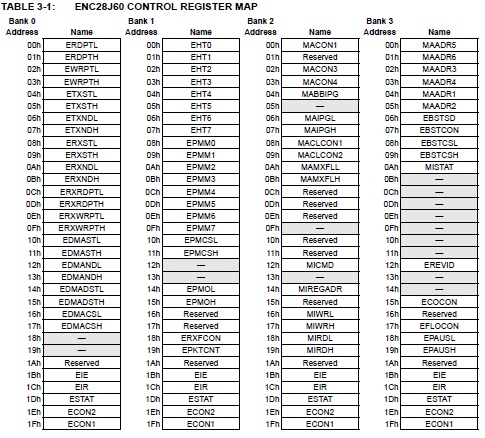
All the symbols of the contacts are visible on the right.

To the debug board we connect the module as follows

**ENC28J60 - STM32F103RCT6**  
**VCC - 3.3V   
GND - GND   
CS - PA4   
RESET - 3.3V**  
**SI - PA7 (MOSI)   
SCK - PA5   
SO - PA6 (MISO)**

Among other things, we'll connect a USART adapter to the board to monitor the data that will come from the network to the controller.

The registers in the microchip are organized as follows



We see here that besides the register addresses there are also banks, that is, the order of addressing of the registers is segmented, and starting from the 1Bh address, access to registers from banks does not depend, they are applied to all banks.

The main purpose of registers can be found by their initial letters in the abbreviation

E - Ethernet,

MA-MAC,

MII-MI.

Specifically with some registers we will get acquainted already when writing the source code, as it will be very much and by the time when we need one or another register, we will already forget its purpose.

Of course, to be able to program the transfer of data over the network, you need knowledge of network layers, OSI models and everything connected with the network, since I will not teach these things, there is a lot of information on these issues, I can only advise something , so contact if that's in the comments. But of course I will give some information, not without it.

In general, the network model is divided into several levels. The main ones are:

1. Application layer

2. Level of representation

3. Session layer

4. Transport level

5. Network layer

6. Link layer

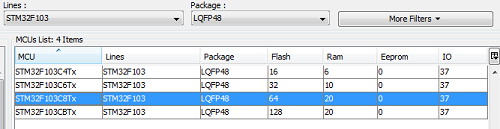
7. Physical layer

When transferring data from the program to a network wire, we go from top to bottom on the list above. First, we combine our data into some convenient sequences or arrays that are understandable for the application layer of the receiving party, then we encrypt them or somehow transform them to protect them from unauthorized use, then wrap them in a specific protocol that will help maintain the connection and not lose it (session level), then hook up another header from the front that will provide transmission to a certain port from another specific port, then wrap it in a protocol that will provide delivery to the redelonnomu device having a network address (IP), then more enveloping protocol that carries the MAC address of the physical device (the link layer),

In general, briefly somehow like this. We will become more familiar with the protocols as they are used.

Well, let's get a little distracted from this whole theory, create our project.

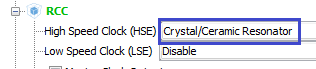
To do this, run the Cube MX project generator, create a new project and select our controller there (click on the image to enlarge the image)

[](http://narodstream.ru/wp-content/uploads/2017/03/Image05.png)

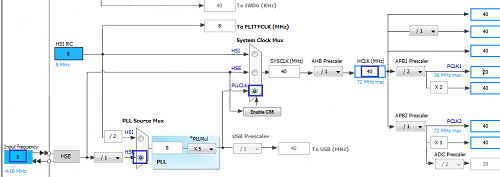
Turn on **SWD**

Image08

Then we connect a quartz resonator



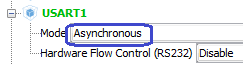
Clock Configuration is configured as follows

[](http://narodstream.ru/wp-content/uploads/2017/03/Image09.png)

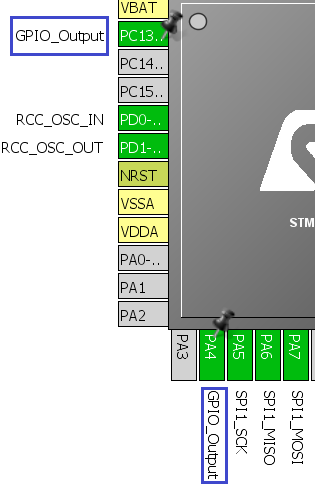
Let's return to "Pinout", turn on SPI

Image10

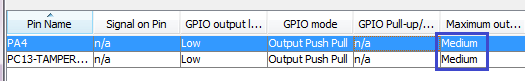
Also include USART



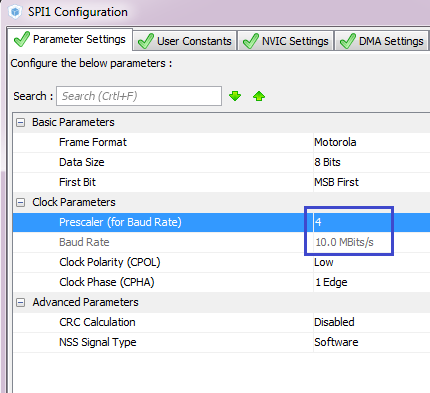
We also connect the ports of **PC13** and **PA4** to the output , since the red LED is connected to the pen, and to the second one we connect the Chip Select of our **SPI**



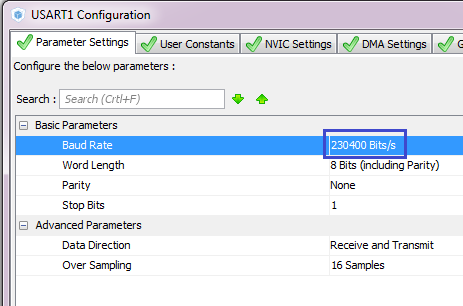
Going to **Configuration** , the first thing to do is include both ports of the feet, the above-mentioned **Medium** speed



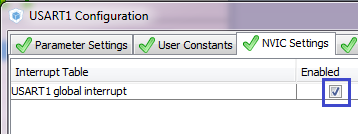
For SPI, configure the following parameters



And we will configure USART as follows



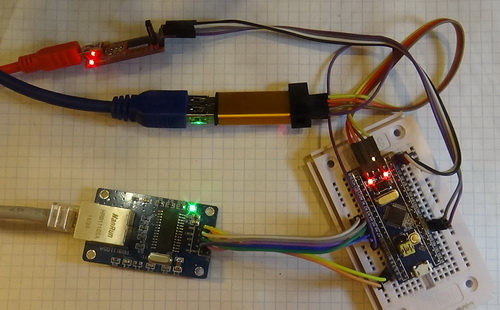
Just in case we turn on interrupts, we will not be superfluous



We will save the project for Keil5, and we will name it by the name of our used microcircuit unpretentious -**ENC28J60**.

Generate the project for Keil, run it, configure the programmer on autoreset and test the project to assemble.

We will connect all our modules by connecting them according to the pinout in the Cube MX, and see what we got (the white wire from USART can not be connected, it's already feeding itself)

[](http://narodstream.ru/wp-content/uploads/2017/03/Image17.jpg)

That is, USART is connected as follows:

**FDI - STM32F103RCT6**

**GND - GND**

**RX-A9**

**TX -   A10**

The **RESET pin of** the **ENC28J60** module  **should** preferably be connected to **3.3V** via a **10 kΩ** resistor .

In the [**next part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-2/) this lesson, we'll add some macros and functions to work with the ENC28J60 chip, and also work a bit on studying the technical documentation.

**Lesson 68**

**Part 2**

**LAN. ENC28J60**

In the [**previous part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-1/) this lesson we briefly got acquainted with the ENC28J60 chip, also created projects and made their settings.

Also create four more files: **net.c** , **net.h** , **enc28j60.c** and **enc28j60.h** and place them in the Inc and Src folders.

Immediately fill in these files with some standard macros and directives

File **net.h**

**#ifndef \_\_NET\_H  
#define \_\_NET\_H  
//--------------------------------------------------  
#include "stm32f1xx\_hal.h"  
#include <string.h>   
#include <stdlib.h>   
#include <stdint.h>**  
**#include "enc28j60.h"  
//--------------------------------------------------  
#endif /\* \_\_NET\_H \*/**

File **net.c**

**#include "net.h"  
//--------------------------------------------------**

File **enc28j60.h**

**#ifndef ENC28J60\_H\_  
#define ENC28J60\_H\_  
//--------------------------------------------------  
#include "stm32f1xx\_hal.h"  
#include <string.h>  
#include <stdlib.h>  
#include <stdint.h>  
//--------------------------------------------------  
#endif /\* ENC28J60\_H\_ \*/**

File **enc28j60.c**

**#include "enc28j60.h"  
//--------------------------------------------------**

Connect our libraries in **main.c**

**/\* USER CODE BEGIN Includes \*/  
#include "net.h"  
#include "enc28j60.h"  
/\* USER CODE END Includes \*/**

Well, and as always, first of all initialization.

Let's **go** to the file **enc28j60.c** and create a function

#include "enc28j60.h"  
**//--------------------------------------------------  
void enc28j60\_ini(void)  
{  
}  
//--------------------------------------------------**

Let's make a prototype for it in the header file, go to the file net.c and write there also an initialization function, in which we call the above function

#include "net.h"  
**//--------------------------------------------------  
void net\_ini(void)  
{  
 enc28j60\_ini();  
}  
//--------------------------------------------------**

Let's make for this function also a prototype in the header file net.h and call it in main ()

  /\* USER CODE BEGIN 2 \*/  
**net\_ini();**  
  /\* USER CODE END 2 \*/

Since we will only access the **SPI** interface in the **enc28j60.c**file , we add the following macros in the **enc28j60.h**file (we write macro macros for convenient control of the red LED)

#include <stdint.h>  
**//--------------------------------------------------  
#define CS\_GPIO\_PORT GPIOA                                        
#define CS\_PIN GPIO\_PIN\_4  
#define SS\_SELECT() HAL\_GPIO\_WritePin(CS\_GPIO\_PORT, CS\_PIN, GPIO\_PIN\_RESET)  
#define SS\_DESELECT() HAL\_GPIO\_WritePin(CS\_GPIO\_PORT, CS\_PIN, GPIO\_PIN\_SET)  
#define LD\_ON HAL\_GPIO\_WritePin(GPIOC, GPIO\_PIN\_13, GPIO\_PIN\_RESET); //RED  
#define LD\_OFF HAL\_GPIO\_WritePin(GPIOC, GPIO\_PIN\_13, GPIO\_PIN\_SET); //RED  
//--------------------------------------------------**

We return to the file **enc28j60.c**and write there a function to handle the error above the initialization function

**//--------------------------------------------------  
static void Error (void)  
{  
 LD\_ON;  
}  
//--------------------------------------------------**

Connect the SPI and USART buses in the file **enc28j60.c**

#include "enc28j60.h"  
**//--------------------------------------------------  
extern UART\_HandleTypeDef huart1;  
extern SPI\_HandleTypeDef hspi1;**

After the error handling function, we'll create functions for working with the SPI bus (we used similar functions when working with the accelerometer on the Discovery 4 board [**here**](http://narodstream.ru/stm-urok-41-podklyuchaem-akselerometr-lis3dsh-chast-1/) )

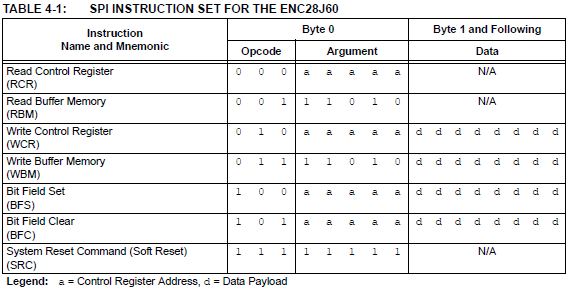
**//--------------------------------------------------  
static uint8\_t SPIx\_WriteRead(uint8\_t Byte)  
{  
  uint8\_t receivedbyte = 0;  
  if(HAL\_SPI\_TransmitReceive(&hspi1, (uint8\_t\*) &Byte, (uint8\_t\*) &receivedbyte, 1, 0x1000) != HAL\_OK)  
  {  
    Error();  
  }  
  return receivedbyte;  
}  
//--------------------------------------------------  
void SPI\_SendByte(uint8\_t bt)  
{  
 SPIx\_WriteRead(bt);  
}  
//--------------------------------------------------  
uint8\_t SPI\_ReceiveByte(void)  
{  
 uint8\_t bt = SPIx\_WriteRead(0xFF);  
 return bt; //вернём регистр данных  
}  
//--------------------------------------------------**

In the initialization function, turn off the LED

void enc28j60\_ini(void)  
{  
**LD\_OFF;**  
}

Now it is necessary to study and understand how commands and data are sent to the microcircuit, and also the data is read.

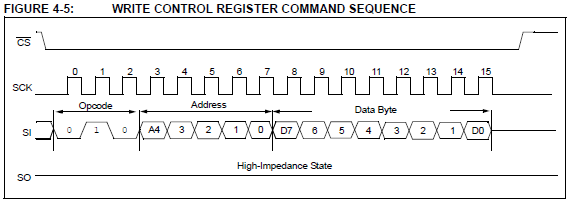
First, any command consists of an opcode and an argument



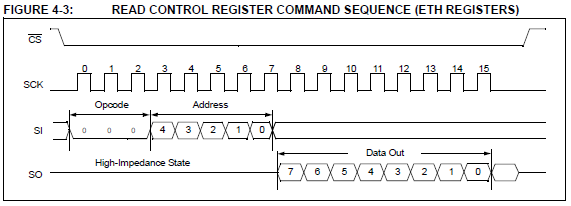
**The opcode** is a sort of command type pointer (read control register, write control register, read buffer, write buffer, set bit in register, clear bit in register, soft reset, it consists of the three highest bits of the command byte, and in the five least significant bytes (in the argument) are the rest of the command information (register address, bitmask, etc.).

Further, if necessary, a data byte is transmitted, or a data byte from the **MISO** bus is received with the read opcodes .

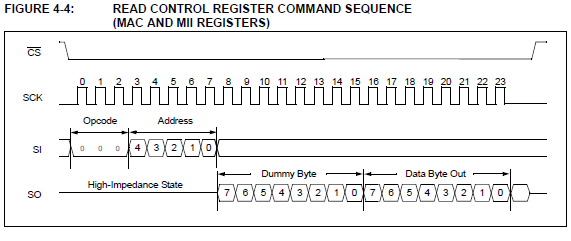
Now let's see how the data is transferred to the microcircuit



And also see how data is received from the control registers



If the MAC or MII registers are used, then reading occurs with skipping a false byte



In the header file enc28j60.h we will write a series of macros that we will need later, basically there will be addresses of the necessary registers, as well as some control bits. Also here we will add the desired physical address (MAC address) of our device. The address can be set any, since it is not assigned to the chip rigidly, that is, it is a very specified value. The main thing is that the address does not coincide with some other one in our network, and also according to some existing conclusions it is undesirable to write something in the very first byte, it's better to leave all the bits there empty, well, at least in the address it should not be put on the 40th bit.

**enc28j60.h**

Now, return to the implementation file enc28j60.c and write the function of writing the byte to the register

**//--------------------------------------------------  
void enc28j60\_writeOp(uint8\_t op,uint8\_t addres, uint8\_t data)  
{  
 SS\_SELECT();  
 SPI\_SendByte(op|(addres&ADDR\_MASK));  
 SPI\_SendByte(data);  
 SS\_DESELECT();  
}  
//--------------------------------------------------**

In the incoming arguments of the function, we will have a separate opcode, shifted accordingly in advance to its place in the three highest bits, the register address and the data byte proper.

**ADDR\_MASK** is a macro for the register address mask that will zero the first three bits, freeing up space for the opcode

Well, now, the read operation

**//--------------------------------------------------  
static uint8\_t enc28j60\_readOp(uint8\_t op,uint8\_t addres)  
{  
 uint8\_t result;  
 SS\_SELECT();  
 SPI\_SendByte(op|(addres&ADDR\_MASK));  
 SPI\_SendByte(0x00);  
 //пропускаем ложный байт  
  if(addres & 0x80) SPI\_ReceiveByte();  
 result=SPI\_ReceiveByte();  
  SS\_DESELECT();  
  return result;  
}  
//--------------------------------------------------**

In the [**next part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-3/) this lesson, we'll start writing code for the initialization of the chip, and also write a number of necessary functions for further work with the module.

**Lesson 68**

**Part 3**

# LAN. ENC28J60

In the [**previous part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-2/) this lesson, we added a few macros and functions to work with the ENC28J60 chip, and also worked a bit on studying the technical documentation.

We perform a "soft" reset in the initialization function and check that everything has rebooted

 LD\_OFF;  
**enc28j60\_writeOp(ENC28J60\_SOFT\_RESET,0,ENC28J60\_SOFT\_RESET);  
 HAL\_Delay(2);  
 //проверим, что всё перезагрузилось  
 while(!enc28j60\_readOp(ENC28J60\_READ\_CTRL\_REG,ESTAT)&ESTAT\_CLKRDY)  
 ;**

During the read operation, we read the **ESTAT** register for the installation of the **CLKRDY** bit .

Let's write the function of reading the buffer (see table 4.1 above)

**//--------------------------------------------------  
static void enc28j60\_readBuf(uint16\_t len,uint8\_t\* data)  
{  
 SS\_SELECT();  
 SPI\_SendByte(ENC28J60\_READ\_BUF\_MEM);  
 while(len--){  
  \*data++=SPIx\_WriteRead(0x00);  
 }  
 SS\_DESELECT();  
}  
//--------------------------------------------------**

Create a variable to store the number of the current bank

**//--------------------------------------------------  
static uint8\_t Enc28j60Bank;**

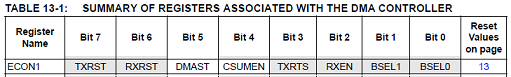
And now write the function of setting the current bank

**//--------------------------------------------------  
static void enc28j60\_SetBank(uint8\_t addres)  
{  
 if ((addres&BANK\_MASK)!=Enc28j60Bank)  
 {  
  enc28j60\_writeOp(ENC28J60\_BIT\_FIELD\_CLR,ECON1,ECON1\_BSEL1|ECON1\_BSEL0);  
  Enc28j60Bank = addres&BANK\_MASK;  
  enc28j60\_writeOp(ENC28J60\_BIT\_FIELD\_SET,ECON1,Enc28j60Bank>>5);  
 }  
}  
//--------------------------------------------------**

Let's try to understand what is happening here.

Well, firstly, if we already have the current bank equal to the one we want to establish, then we do nothing and we leave

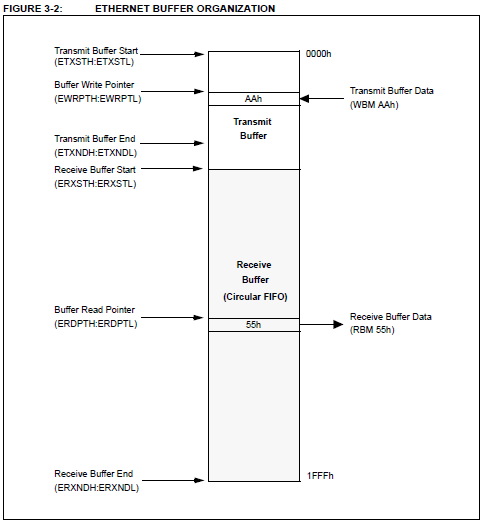
And then we register **ECON1** clear bits **BSEL1** and **BSEL0** , responsible for the bank, and then stored in the variable, our bank from the address, move the resulting value to the right by 5 points to the desired bits were in the minors, and set the appropriate bits in the same register



Then write the universal functions for reading and writing normal control registers

**//--------------------------------------------------  
static void enc28j60\_writeRegByte(uint8\_t addres,uint8\_t data)  
{  
 enc28j60\_SetBank(addres);  
 enc28j60\_writeOp(ENC28J60\_WRITE\_CTRL\_REG,addres,data);  
}  
//--------------------------------------------------  
static uint8\_t enc28j60\_readRegByte(uint8\_t addres)  
{  
 enc28j60\_SetBank(addres);  
 return enc28j60\_readOp(ENC28J60\_READ\_CTRL\_REG,addres);  
}  
//--------------------------------------------------**

The buffer, which works both for reading and writing, must also be configured with the help of registers



Since all the pointers are double-byte, we will create another function for writing data to such registers

**//--------------------------------------------------  
static void enc28j60\_writeReg(uint8\_t addres,uint16\_t data)  
{  
 enc28j60\_writeRegByte(addres, data);  
 enc28j60\_writeRegByte(addres+1, data>>8);  
}  
//--------------------------------------------------**

Configure the buffers in the initialization function

**//настроим буферы  
 enc28j60\_writeReg(ERXST,RXSTART\_INIT);  
 enc28j60\_writeReg(ERXRDPT,RXSTART\_INIT);  
 enc28j60\_writeReg(ERXND,RXSTOP\_INIT);  
 enc28j60\_writeReg(ETXST,TXSTART\_INIT);  
 enc28j60\_writeReg(ETXND,TXSTOP\_INIT);**

In the [**next part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-4/) this lesson, we will end the function of initializing the chip, and also write the function of receiving network packets.

**Lesson 68**

**Part 4**

# LAN. ENC28J60

In the [**previous part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-3/) this lesson we started writing the initialization code for the chip, and also wrote a number of necessary functions for further work with the module.

Turn on the broadcast, suddenly come in handy

**//Enable Broadcast  
 enc28j60\_writeRegByte(ERXFCON,enc28j60\_readRegByte(ERXFCON)|ERXFCON\_BCEN);**

You can also enable packet filtering with checksum checks using certain **EPMM0-EPMM7** and **EPMMCS registers** , but for now we will not do this, we want to see all the packets. If this is subsequently necessary, we will always be able to filter out. I just want to note that our chip is very good at filtering packets by mask and checksum.

Add to our file the global variable of our physical address

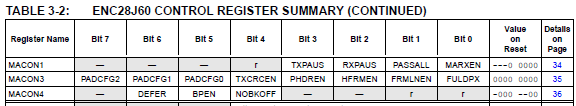
static uint8\_t Enc28j60Bank;   
**uint8\_t macaddr [6] = MAC\_ADDR;**

Well, accordingly, configure the channel level in the initialization

**//настраиваем канальный уровень  
 enc28j60\_writeRegByte(MACON1,MACON1\_MARXEN|MACON1\_TXPAUS|MACON1\_RXPAUS);  
 enc28j60\_writeRegByte(MACON2,0x00);  
 enc28j60\_writeOp(ENC28J60\_BIT\_FIELD\_SET,MACON3,MACON3\_PADCFG0|MACON3\_TXCRCEN|MACON3\_FRMLNEN);  
 enc28j60\_writeReg(MAIPG,0x0C12);  
 enc28j60\_writeRegByte(MABBIPG,0x12);//промежуток между фреймами  
 enc28j60\_writeReg(MAMXFL,MAX\_FRAMELEN);//максимальный размер фрейма  
 enc28j60\_writeRegByte(MAADR5,macaddr[0]);//Set MAC addres  
 enc28j60\_writeRegByte(MAADR4,macaddr[1]);  
 enc28j60\_writeRegByte(MAADR3,macaddr[2]);  
 enc28j60\_writeRegByte(MAADR2,macaddr[3]);  
 enc28j60\_writeRegByte(MAADR1,macaddr[4]);  
 enc28j60\_writeRegByte(MAADR0,macaddr[5]);**

Briefly about what is happening here.

We have four registers for configuring this level of **MACON1-MACON4** , although in the documentation only 3, the second is not



Therefore in the second we will send all zeros.

In the first register we will include the following bits: **MARXEN** , which allows receiving packets, as well as **TXPAUS** and  **RXPAUS** , which include hardware flow control.

In the third register, we turn on padding, or alignment. For this, there are three **PADCFG** bits , in which we will enable **001** to equalize the packet with zeros to 60 bytes and add a 4-byte checksum. Also here we include bits **FRMLNEN** , which automatically checks the length of the packets and **TXCRCEN** , which automatically adds a checksum to the packet.

Next are the standard values ​​of the **MAIPG** and **MABBIPG registers** , which set the full-duplex mode to certain delays.

Then **MAMXFL** , which sets the maximum **frame** size (frame).

Well, then, I think everyone understands that we are obliged to register our **MAC address** in the registers .

Then we work with the physical level.

It's not so simple.

It turns out that to put a value in one of the registers that control the physical layer ( **PHY** ), you need to do this not straightforward, since there is no such possibility, but through registers **MII** .

We have such a register **MIREGADR** , in which we first have to enter the value of the address of a register PHY. And then the data destined for this register, we put in the registers **MIWRL** (low byte) and **MIWRH** (high byte), and it is in this sequence. Then read the **MISTAT** register for a **BUSY** bit reset . Here and so.

Therefore, we write a separate function for writing the PHY register

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

**static void enc28j60\_writePhy(uint8\_t addres,uint16\_t data)**

**{**

**enc28j60\_writeRegByte(MIREGADR, addres);**

**enc28j60\_writeReg(MIWR, data);**

**while(enc28j60\_readRegByte(MISTAT)&MISTAT\_BUSY)**

**;**

**}**

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

Now configure the physical layer in the initialization

**enc28j60\_writeRegByte(MAADR0,macaddr[5]);**

**//настраиваем физический уровень**

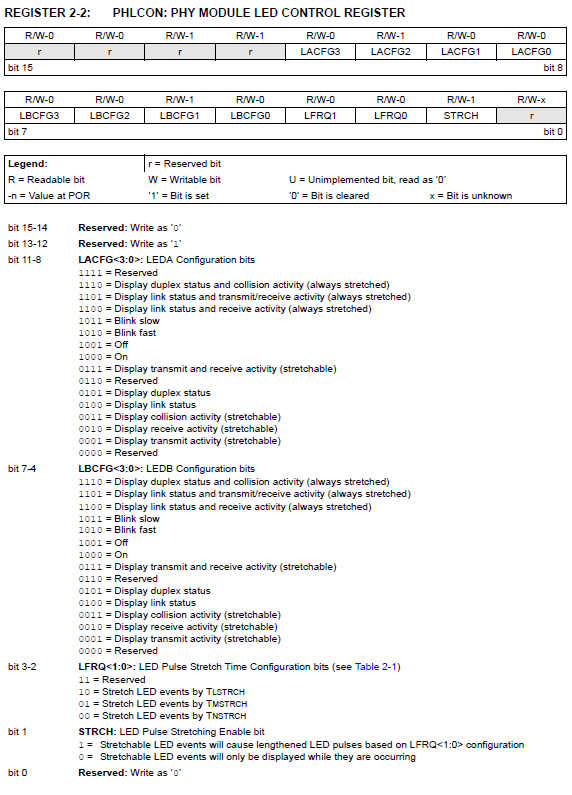
**enc28j60\_writePhy(PHCON2,PHCON2\_HDLDIS);//отключаем loopback**

**enc28j60\_writePhy(PHLCON,PHLCON\_LACFG2| //светодиоды**

**PHLCON\_LBCFG2|PHLCON\_LBCFG1|PHLCON\_LBCFG0|**

**PHLCON\_LFRQ0|PHLCON\_STRCH);**

Well, almost everything is clear from the comments. On one LED we have as many as 16 bits (separate register)



With the LEDs everything is basically clear.

Well, the final settings in the initialization

    PHLCON\_LFRQ0 | PHLCON\_STRCH);

**enc28j60\_SetBank (ECON1);**

**enc28j60\_writeOp (ENC28J60\_BIT\_FIELD\_SET, EIE, EIE\_INTIE | EIE\_PKTIE);**

**enc28j60\_writeOp (ENC28J60\_BIT\_FIELD\_SET, ECON1, ECON1\_RXEN); // allow receiving packets**

**}**

In the **EIE** register, **we** set the bits that allow global interrupts and receive packets waiting for an interrupt.

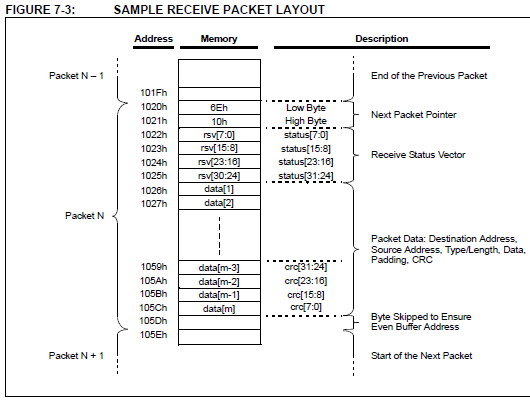
Also we will allow the reception of packages in general.

That's the whole initialization. At last!

Now we need to learn how to accept packets, since all the exchange on the local network is divided into packets.

The reception of packets in the chip occurs through the ring buffer.

Let's take a look at the example of receiving the package



The ENC28J60 writes the packet to the buffer as follows. First, 2 bytes with the pointer value to the next packet go, then 4 bytes with the receive status, then the actual data of the packet with the checksum.

Let's create a function for receiving the package in the file **enc28j60.c** and create a local variable in it to calculate the length of the packet and its subsequent return

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

**uint16\_t enc28j60\_packetReceive(uint8\_t \*buf,uint16\_t buflen)**

**{**

**}**

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

The **EPKTCNT** register stores the number of packets received so far, so we'll check it

uint16\_t len=0;

**if(enc28j60\_readRegByte(EPKTCNT)>0)**

**{**

**}**

Add a global variable in the file **enc28j60.c**

static uint8\_t Enc28j60Bank;

**static int gNextPacketPtr;**

In the ERDPT register, we set the pointer

if(enc28j60\_readRegByte(EPKTCNT)>0)

{

**enc28j60\_writeReg(ERDPT,gNextPacketPtr);**

Let's start reading the package. Declare the header structure for the header

enc28j60\_writeReg(ERDPT,gNextPacketPtr);

**//считаем заголовок**

**struct{**

**uint16\_t nextPacket;**

**uint16\_t byteCount;**

**uint16\_t status;**

**} header;**

Now, in fact, we consider the package header containing the exponent for the next packet, the number of bytes and the status of the packet

} header;

**enc28j60\_readBuf(sizeof header,(uint8\_t\*)&header);**

Let's write the value of the pointer to the next packet in the global variable

enc28j60\_readBuf(sizeof header,(uint8\_t\*)&header);

**gNextPacketPtr=header.nextPacket;**

Initialize the variable length packets, cutting off the checksum from it

gNextPacketPtr=header.nextPacket;

**len=header.byteCount-4;//remove the CRC count**

We shorten the length to the value specified in the input parameter

len=header.byteCount-4;//remove the CRC count

**if(len>buflen) len=buflen;**

Check the status and consider the buffer

if(len>buflen) len=buflen;

**if((header.status&0x80)==0) len=0;**

**else enc28j60\_readBuf(len, buf);**

We end the buffer with zero

else enc28j60\_readBuf(len, buf);

**buf[len]=0;**

Initialize the buffer pointer to the address of the next packet

buf[len]=0;

**if(gNextPacketPtr-1>RXSTOP\_INIT)**

**enc28j60\_writeReg(ERXRDPT,RXSTOP\_INIT);**

**else**

**enc28j60\_writeReg(ERXRDPT,gNextPacketPtr-1);**

The counter of the received packets is also reduced by 1 and we get out of the condition

  enc28j60\_writeReg(ERXRDPT,gNextPacketPtr-1);

**enc28j60\_writeOp(ENC28J60\_BIT\_FIELD\_SET,ECON2,ECON2\_PKTDEC);**

**}**

Return the length of the received packet

    enc28j60\_writeOp(ENC28J60\_BIT\_FIELD\_SET,ECON2,ECON2\_PKTDEC);

  }

**return len;**

}

In the [**next part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-5/) this lesson we will try to understand the structure of Ethernet frames, write the function of their reception and check this process in the terminal program, and also get acquainted with the ARP protocol.

**Lesson 68**

**Part 5**

# LAN. ENC28J60

In the [**previous part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-4/) this lesson, we finished writing the initialization function of the chip, and also wrote the function of receiving network packets.

In the file **net.c** we will create a function of constant network interrogation

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

**void net\_pool(void)**

**{**

**}**

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

Well, since we have a constant poll, we will call this function in the infinite loop of main (), creating a prototype in **net.h**

while (1)

{

**net\_pool();**

  /\* USER CODE END WHILE \*/

We know that the bit stream arriving at the chip on its physical level is subsequently divided into frames in the channel level, in English - frames.

Frames of themselves are quite a conscious flow, having a certain protocol.

At present, in most cases, the protocol layer of **Ethernet Version 2 is used** . It consists of the physical (MAC) address of the receiving device, the physical address of the sending device, 2 bytes or the length of the data field of the frame, or the identifier of the protocol (ARP, IP, etc.), the actual data, which will also have other protocols, and a checksum of 4 bytes, which we expect to learn a little later.

This is how the frame looks more clearly



Create a frame for the frame in **net.h**

#include "enc28j60.h"

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

**typedef struct enc28j60\_frame{**

**uint8\_t addr\_dest[6];**

**uint8\_t addr\_src[6];**

**uint16\_t type;**

**uint8\_t data[];**

**} enc28j60\_frame\_ptr;**

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

In the file **net.c** we add a global variable

#include "net.h"

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

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

Create two variables in our **net\_poll** function

void net\_pool(void)

{

**uint16\_t len;**

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

We will also create a function for reading the frame just above the function net\_poll

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

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

**{**

**}**

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

The protocol type in the frame is transmitted in an inverted form, not as usual, in the format **big endian** . Therefore, we write a definite define for the transformation in the file **net.h**

} enc28j60\_frame\_ptr;

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

**#define be16toword(a) ((((a)>>8)&0xff)|(((a)<<8)&0xff00))**

In the file **net.c** we add a global string array

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

**char str1[60]={0};**

Connect the **USART** bus to **net.c**

#include "net.h"

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

**extern UART\_HandleTypeDef huart1;**

Let's add the code to the function **eth\_read** , which will output to the terminal program the address of the source that sent the frame, the address of the receiver and the protocol type (while in digital form)

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

{

**if (len>=sizeof(enc28j60\_frame\_ptr))**

**{**

**sprintf(str1,"%02X:%02X:%02X:%02X:%02X:%02X-%02X:%02X:%02X:%02X:%02X:%02X; %d; %04Xrn",**

**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,be16toword(frame->type));**

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

**}**

}

In the file **enc28j60.h,** add the prototype function **enc28j60\_packetReceive**

void enc28j60\_ini(void);

**uint16\_t enc28j60\_packetReceive(uint8\_t \*buf,uint16\_t buflen);**

In the file net.c in the function of processing the reception and transmission from the **net\_poll** chip we **will** add the code for receiving information from the read buffer

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

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

**{**

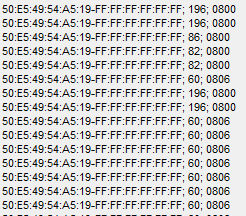
**eth\_read(frame,len);**

**}**

}

Finally, we'll collect the code and edit the controller, and then, I think everyone is already bored and want to see at least some result of our work.

Let's start the terminal program and see certain lines that will show us the source address that sent the broadcast request, also the broadcast address, since our physical address is not yet known to any network device (by the way, the module is connected to a common network in which there are several different devices) , and also we see the length of the frame and the protocol identifier



We are currently interested in two types of network protocols - **IP (0x0800)**  and **ARP (0x0806)** , and while we are more specific we will deal with **ARP** .

Therefore, add macros to the header file **net.h**

#define be16toword(a) ((((a)>>8)&0xff)|(((a)<<8)&0xff00))

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

**#define ETH\_ARP be16toword(0x0806)**

**#define ETH\_IP be16toword(0x0800)**

**Let's** fix the code in the function  **eth\_read a little** so that the type in the terminal program is displayed not in the digital but in the line view

if (len>=sizeof(enc28j60\_frame\_ptr))

{

**if(frame->type==ETH\_ARP)**

**{**

    sprintf(str1,"%02X:%02X:%02X:%02X:%02X:%02X-%02X:%02X:%02X:%02X:%02X:%02X; %d; **arp**rn",

    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);

**}**

**else if(frame->type==ETH\_IP)**

**{**

    sprintf(str1,"%02X:%02X:%02X:%02X:%02X:%02X-%02X:%02X:%02X:%02X:%02X:%02X; %d; **ip**rn",

    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],

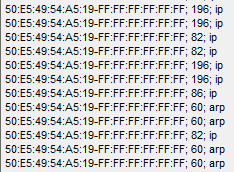
    len);

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

**}**

}

We'll collect the code and see



Now we can already clearly see what kind of protocol we have and who sent the shot.

The next task is to read the structure of the ARP header, and this is already the next level - not a channel, but a network one.

**ARP (Address Resolution Protocol)** or an address resolution protocol is a protocol that serves to allow a network device to determine someone's physical address (MAC) over a network address (IP). For this, the device sends a broadcast request (with the address of the receiver FF-FF-FF-FF-FF-FF, consisting of all units), which asks: "What physical address does the device have with that IP address?". This is required in order for devices to communicate on a physical level, which knows nothing about IP addresses. Therefore, the device sends such a request. It is received by all computers (or some other devices) on this local network, and the device that recognized its network address must send a response.

Now let's look at how this is done. **ARP** header structure :



In the [**next part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-6/) this lesson, we will write a structure for the ARP header, and also write a study function for this protocol, see some parts of the header of the received ARP request in the terminal program

**Lesson 68**

**Part 6**

# LAN. ENC28J60

In the [**previous part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-5/) this lesson, we figured out the structure of Ethernet frames, wrote the function of their reception and tested this process in the terminal program, and also got acquainted with the ARP protocol.

Since we now know the format of the **ARP** protocol , now we can work with it. To do this, we first add a structure for the **ARP** header

} enc28j60\_frame\_ptr;

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

**typedef struct arp\_msg{**

**uint16\_t net\_tp;**

**uint16\_t proto\_tp;**

**uint8\_t macaddr\_len;**

**uint8\_t ipaddr\_len;**

**uint16\_t op;**

**uint8\_t macaddr\_src[6];**

**uint8\_t ipaddr\_src[4];**

**uint8\_t macaddr\_dst[6];**

**uint8\_t ipaddr\_dst[4];**

**} arp\_msg\_ptr;**

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

Let's return now to the file **net.c** and add a function to read the arp package just above the frame reading function

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

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

**{**

**}**

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

We will return 1 if the module recognizes its address in the request.

Call this function in the **eth\_read** function

if(frame->type==ETH\_ARP)

{

  ...

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

**arp\_read(frame,len-sizeof(enc28j60\_frame\_ptr));**

Add a variable for the result and put a pointer to the package **arp** , for this we add the appropriate code to the function **arp\_read**

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

{

**uint8\_t res=0;**

**arp\_msg\_ptr \*msg=(void\*)(frame->data);**

Add some more macros to the file **net.h**

#define ETH\_IP be16toword(0x0800)

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

**#define ARP\_ETH be16toword(0x0001)**

**#define ARP\_IP be16toword(0x0800)**

**#define ARP\_REQUEST be16toword(1)**

**#define ARP\_REPLY be16toword(2)**

Also in the file **net.h** we add a macro to set the desired IP-address for our device

#include "enc28j60.h"

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

**#define IP\_ADDR {192,168,1,197}**

Add to net.c the global ip address variable

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

**uint8\_t ipaddr[4]=IP\_ADDR;**

In general, at the moment the address will be static. Maybe sometime we'll get to the DHCP request to the server to get the dynamic address, but for now it's exactly far to us.

Add to our function reading the ARP package a number of conditions for filtering unnecessary packets that do not meet our requirements, although while this filter is hollow, and also return the result

arp\_msg\_ptr \*msg=(void\*)(frame->data);

**if (len>=sizeof(arp\_msg\_ptr))**

**{**

**if ((msg->net\_tp==ARP\_ETH)&&(msg->proto\_tp==ARP\_IP))**

**{**

**if ((msg->op==ARP\_REQUEST)&&(!memcmp(msg->ipaddr\_dst,ipaddr,4)))**

**{**

**}**

**}**

**return res;**

**}**

In general, here we want to work only with the ARP package, and with the request, and also remove what does not correspond to our network address.

Now, in the **arp\_read** function **,**   add the condition type to the body of the request type, source addresses and receiver addresses, and set the result

if ((msg->op==ARP\_REQUEST)&&(!memcmp(msg->ipaddr\_dst,ipaddr,4)))

{

**sprintf(str1,"requestrnmac\_src %02X:%02X:%02X:%02X:%02X:%02Xrn",**

**msg->macaddr\_src[0],msg->macaddr\_src[1],msg->macaddr\_src[2],msg->macaddr\_src[3],msg->macaddr\_src[4],msg->macaddr\_src[5]);**

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

**sprintf(str1,"ip\_src %d.%d.%d.%drn",**

**msg->ipaddr\_src[0],msg->ipaddr\_src[1],msg->ipaddr\_src[2],msg->ipaddr\_src[3]);**

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

**sprintf(str1,"nmac\_dst %02X:%02X:%02X:%02X:%02X:%02Xrn",**

**msg->macaddr\_dst[0],msg->macaddr\_dst[1],msg->macaddr\_dst[2],msg->macaddr\_dst[3],msg->macaddr\_dst[4],msg->macaddr\_dst[5]);**

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

**sprintf(str1,"ip\_dst %d.%d.%d.%drn",**

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

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

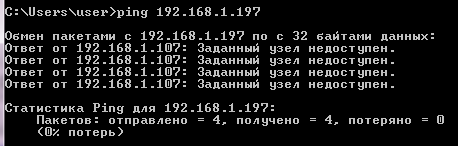
**res=1;**

}

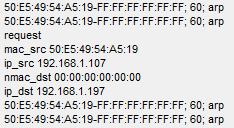
Well, everything is simple. Only there is one question. Why on earth would any device send us an ARP request, we did not tell anyone about our IP address.

And we will help him. We will use the ping utility built into any operating system. Although ping sends an ICMP request, but if it sees that the ARP address mapping table does not have our IP address, it first sends an ARP request.

That's why we'll send ping to the IP-address assigned by us from the command line, of course, after collecting the code and flashing the controller



We see that our ping board has not responded, but it's okay, we did not write an answer to ICMP requests, but to ARP, but we see this in our terminal program



We see both the physical and network address of the device that sent us the request (this is our main PC with address 192.168.1.89), as well as the network address of the receiver. Since the physical address of the receiver is not yet known, we get a null address.

In the [**next part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-7/) this lesson, we will write several functions for sending certain packages and check the response to ARP requests in the Wireshark program.

**Lesson 68**

**Part 7**

# LAN. ENC28J60

In the [**previous part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-6/) this lesson, we wrote a structure for the ARP header, and also wrote a study function for this protocol, saw some parts of the header of the received ARP request in the terminal program.

The next task before us is to respond to the ARP request.

But we remember that we wrote only the function of receiving Ethernet packets (or frames). And to answer something, we must also be able to send such packets. Therefore, we will have to return to the file enc28j60.c and add a function for sending packets

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

**void enc28j60\_packetSend(uint8\_t \*buf,uint16\_t buflen)**

**{**

**}**

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

Let's wait until the **TXRTS** bit in the **ECON1** register is **set** , which means that our transmitter is ready to transmit information

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

void enc28j60\_packetSend(uint8\_t \*buf,uint16\_t buflen)

{

**while(enc28j60\_readOp(ENC28J60\_READ\_CTRL\_REG,ECON1)&ECON1\_TXRTS)**

**{**

**{**

}

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

while(enc28j60\_readOp(ENC28J60\_READ\_CTRL\_REG,ECON1)&ECON1\_TXRTS)

{

**if(enc28j60\_readRegByte(EIR)& EIR\_TXERIF)**

**{**

**enc28j60\_writeOp(ENC28J60\_BIT\_FIELD\_SET,ECON1,ECON1\_TXRST);**

**enc28j60\_writeOp(ENC28J60\_BIT\_FIELD\_CLR,ECON1,ECON1\_TXRST);**

**}**

}

Add the function of writing to the buffer

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

**static void enc28j60\_writeBuf(uint16\_t len,uint8\_t\* data)**

**{**

**SS\_SELECT();**

**SPI\_SendByte(ENC28J60\_WRITE\_BUF\_MEM);**

**while(len--)**

**SPI\_SendByte(\*data++);**

**SS\_DESELECT();**

**}**

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

We continue the function of sending the package. Then write and send the package

      enc28j60\_writeOp(ENC28J60\_BIT\_FIELD\_CLR,ECON1,ECON1\_TXRST);

    }

  }

**enc28j60\_writeReg(EWRPT,TXSTART\_INIT);**

**enc28j60\_writeReg(ETXND,TXSTART\_INIT+buflen);**

**enc28j60\_writeBuf(1,(uint8\_t\*)"x00");**

**enc28j60\_writeBuf(buflen,buf);**

**enc28j60\_writeOp(ENC28J60\_BIT\_FIELD\_SET,ECON1,ECON1\_TXRTS);**

}

Add this function prototype in a header file and return to **net.c** . In the eth\_read **function,** we will correct the code slightly to send an ARP response only if the ARP request returns one

if(frame->type==ETH\_ARP)

{

  ...

**if(**arp\_read(frame,len-sizeof(enc28j60\_frame\_ptr))**)**

**{**

**}**

}

Add the physical address of our device to **net.c** by inheriting it from another module

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

**extern uint8\_t macaddr[6];**

Add a function to send a frame over **Ethernet** protocol somewhere higher, in which we will set the physical addresses of the source and the receiver, and then send the packet

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

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

**{**

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

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

**enc28j60\_packetSend((void\*)frame,len + sizeof(enc28j60\_frame\_ptr));**

**}**

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

We will also add the function of sending the ARP reply packet immediately after the function **eth\_send**

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

**void arp\_send(enc28j60\_frame\_ptr \*frame)**

**{**

**}**

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

In the arp\_send  **function,** enter the ARP response code in the ARP header

void arp\_send(enc28j60\_frame\_ptr \*frame)

{

**arp\_msg\_ptr \*msg = (void\*)frame->data;**

**msg->op = ARP\_REPLY;**

We also include physical addresses in the header

msg->op = ARP\_REPLY;

**memcpy(msg->macaddr\_dst,msg->macaddr\_src,6);**

**memcpy(msg->macaddr\_src,macaddr,6);**

Then zanesom **IP** URLs

memcpy(msg->macaddr\_src,macaddr,6);

**memcpy(msg->ipaddr\_dst,msg->ipaddr\_src,4);**

**memcpy(msg->ipaddr\_src,ipaddr,4);**

And at the end of the function we send the package

  memcpy(msg->ipaddr\_src,ipaddr,4);

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

}

In the eth\_read function, we will respond to the ARP request

if(arp\_read(frame,len-sizeof(enc28j60\_frame\_ptr)))

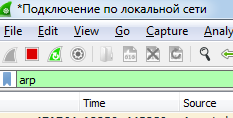
{

**arp\_send(frame);**

}

We will collect the code, we will sew the controller. In order to see our answer, it is desirable to install a free program Wireshark. This is a network packet analyzer.

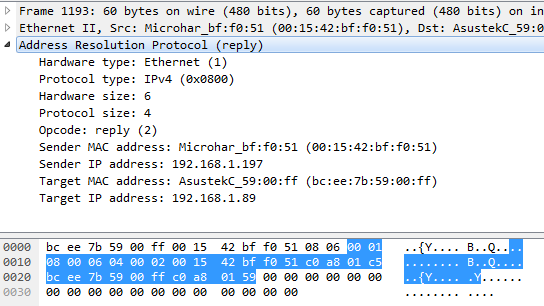
Install the filter in this program only on packages  **arp**



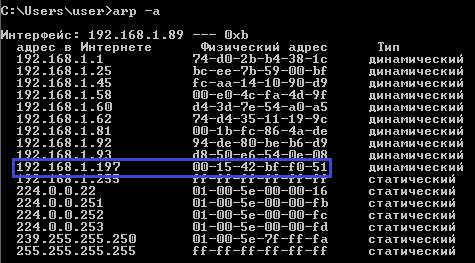
Propinue our IP address in the same way as before. We will not receive a response to the ping, but we will see an answer to the ARP request in the utility

image34

You can also open and view the structure of the ARP package



Also in the correspondence table of the computer's ARP addresses, we now see our data



In the [**next part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-8/) this lesson we will work on the IP protocol, and also learn how to calculate the checksum of the header and write a function for this.

**Lesson 68**

**Part 8**

# LAN. ENC28J60

In the [**previous part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-7/) this lesson we wrote several functions for sending certain packages and checked the response to ARP requests in the Wireshark program.

ARP-response we sent, I would like to answer pings, but it's not so simple. You need to learn how to work with the **Internet Control Message Protocol ( ICMP )** between the network layer and the transport layer. And to work with the ICMP protocol, you need to understand **IP** protocol . We'll start with it.

The IP protocol is more difficult than the ARP protocol, since it has many tasks before it. One of them is to find the route by which the packet should be sent, also the package can consist of several fragments that are in different frames. But nothing, we'll sort it out.

Above the function **eth\_read** we add another function to read the IP packet

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

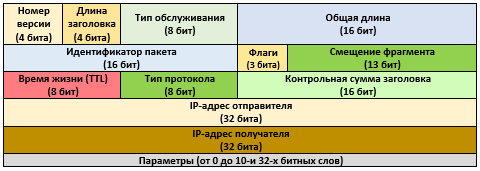
**uint8\_t ip\_read(enc28j60\_frame\_ptr \*frame, uint16\_t len)**

**{**

**}**

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

Here is the **IP** packet header format



The first field is the version number, in our case 4, since the version of the protocol we have is IPv4, in which the IP address consists of 4 bytes.

Next is the length of the header, measured in 4-byte values ​​or in double words (DWORD), so the IP header should be a multiple of 4 bytes. In our case without parameters and additional. options here is figure 5.

The type of service is practically not used at the moment.

The total length is the total length of the packet along with the data. It is important to us, because we do not know it because of the alignment of the packets. It is measured in bytes. The maximum value is 65535 bytes. But in practice it does not exceed 1500 because of the maximum length of the Ethernet frame.

The packet identifier, flags and fragment offset are used for fragmentation, we do not consider it yet.

The lifetime is the number of hops (jumps). After passing each router is reduced by one, so that any left packets do not walk forever over the network.

The protocol type is an example - TCP - 6, UDP - 17, ICMP - 1.

The header checksum is a cleverly calculated value that serves to verify the packet reaches the recipient. It is calculated on both the receiver and the source and is compared. Also, the checksum changes after passing through the router, because there the lifetime decreases by 1 and the field changes.

And further it is clear - the IP-addresses of the sender and the recipient.

Let's add a structure for the IP **packet to net.h**

} arp\_msg\_ptr;

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

**typedef struct ip\_pkt{**

**uint8\_t verlen;//версия протокола и длина заголовка**

**uint8\_t ts;//тип севриса**

**uint16\_t len;//длина**

**uint16\_t id;//идентификатор пакета**

**uint16\_t fl\_frg\_of;//флаги и смещение фрагмента**

**uint8\_t ttl;//время жизни**

**uint8\_t prt;//тип протокола**

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

**uint8\_t ipaddr\_src[4];//IP-адрес отправителя**

**uint8\_t ipaddr\_dst[4];//IP-адрес получателя**

**uint8\_t data[];//данные**

**} ip\_pkt\_ptr;**

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

We'll also write macros for protocol types

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

**#define IP\_ICMP 1**

**#define IP\_TCP 6**

**#define IP\_UDP 17**

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

Let's now work out a little with the calculation of the checksum of the headers. It is calculated almost the same for all types of headers and packets, so the function will be universal.

The general principle of the calculation is as follows. The header is divided into 16-bit values. They all add up. If the sum does not fit into a 16-bit value, which often happens, it is divided into two 16-bit numbers - the upper half-word and the lower one, and they also add up. If it does not fit again, repeat this operation until it gets into 16 bits. The resulting value is first converted to a big endian, then inverted, that is, instead of all units, we write zeros, and instead of zeros, one. There is a small nuance. Suddenly, the header length will consist of an odd number of bytes. Then the last byte is equalized by zero, that is, the existing one byte becomes the oldest one, and in the younger one all zeros are written. Something like this. It is possible to add zeros to an even length in advance, the entire headline, this is as you like. Also do not forget,

Let's write this function somewhere higher to be visible from other functions

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

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

**{**

**}**

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

**ptr** - header of the header, the checksum of which we are counting,

**len** is the length of the header.

Return the value of the calculated checksum.

Let's add a variable for the sum

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

{

**uint32\_t sum = 0;**

Add a conditional loop that tracks the ending of the header

uint32\_t sum=0;

**while(len>1)**

**{**

**}**

In the body of the conditional loop, calculate and add to the total amount of the next two-byte value

while(len>1)

{

**sum += (uint16\_t) (((uint32\_t)\*ptr<<8)|\*(ptr+1));**

As we see, in order to get a big endian, it is not necessary to use a macro and lose unnecessary machine cycles, just move the left one to the right and make the low byte at the earliest and add the high byte to it, which automatically becomes the youngest byte.

Further here in the body of the conditional cycle, we will move the pointer further by 2 points and reduce the value of the header length by 2

  sum += (uint16\_t) (((uint32\_t)\*ptr<<8)|\*(ptr+1));

**ptr+=2;**

**len-=2;**

}

We leave the body of the loop and further in our function, we check the length of the header for parity by checking the variable **len** for the remaining byte. If the byte really remains, then move it to the left, thereby at the same time free the right-hand side for zero-parity

  len-=2;

}

**if(len) sum+=((uint32\_t)\*ptr)<<8;**

Next, according to the condition of calculating the sum, we need to find out whether our bag has fit into a size of 16 bits. To do this, we will create another conditional loop, in the body of which we will constantly add the high and low parts of the sum to each other until the sum fits into a 16-bit size

if(len) sum+=((uint32\_t)\*ptr)<<8;

**while (sum>>16) sum=(uint16\_t)sum+(sum>>16);**

And at the end of the function, we only need to convert the amount into the big endian format, then invert it and return it from the function

  while (sum>>16) sum=(uint16\_t)sum+(sum>>16);

**return ~be16toword((uint16\_t)sum);**

}

Add to the function of reading the IP packet variable for the result, and also set the pointer to the package

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

{

**uint8\_t res=0;**

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

But since there is still a lot of source code to write in this function, we will postpone this matter to the [**next part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-9/) our lesson.

**Lesson 68**

**Part 9**

# LAN. ENC28J60

In the [**previous part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-8/) this lesson, we studied the IP header, wrote a structure for it, and learned how to calculate the checksum of the header and wrote a function for this.

Add the filtering according to the protocol version, the length of the header and the correspondence of the destination address to our IP, and return the result to the IP packet read function

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

**if((ip\_pkt->verlen==0x45)&&(!memcmp(ip\_pkt->ipaddr\_dst,ipaddr,4)))**

**{**

**}**

**return res;**

}

Let's calculate the size of the data in bytes

if((ip\_pkt->verlen==0x45)&&(!memcmp(ip\_pkt->ipaddr\_dst,ipaddr,4)))

{

**//длина данных**

**len = be16toword(ip\_pkt->len) - sizeof(ip\_pkt\_ptr);**

Let's compare the checksum from the field with the checksum with the calculated by us while in text form

len = be16toword(ip\_pkt->len) - sizeof(ip\_pkt\_ptr);

**sprintf(str1,"rnip\_cs 0x%04Xrn", ip\_pkt->cs);**

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

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

**sprintf(str1,"ip\_cs 0x%04Xrn", checksum((void\*)ip\_pkt,sizeof(ip\_pkt\_ptr)));**

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

Call our function in the **eth\_read** function

else if(frame->type==ETH\_IP)

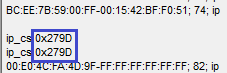
{

  ...

**ip\_read(frame,len-sizeof(ip\_pkt\_ptr));**

}

We will collect the code, we will impose the controller and check the incoming and calculated checksum again with the help of the ping utility



Everything converges. Excellent!

Now you can delete the comparison code. We will then compare in a different way

~~sprintf(str1,"rnip\_cs 0x%04Xrn", ip\_pkt->cs);~~

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

~~ip\_pkt->cs=0;~~

~~sprintf(str1,"ip\_cs 0x%04Xrn", checksum((void\*)ip\_pkt,sizeof(ip\_pkt\_ptr)));~~

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

We return to the function **ip\_read** and learn the type of protocol

len = be16toword(ip\_pkt->len) - sizeof(ip\_pkt\_ptr);

**if (ip\_pkt->prt==IP\_ICMP)**

**{**

**}**

**else if (ip\_pkt->prt==IP\_TCP)**

**{**

**}**

**else if (ip\_pkt->prt==IP\_UDP)**

**{**

**}**

Add the function of reading the **ICMP-** package

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

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

**{**

**}**

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

Call this function in the function **ip\_read**

if (ip\_pkt->prt==IP\_ICMP)

{

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

}

Now let's look at the **ICMP** packages .

In the function  **ip\_read,** we still add a variable for the result and return it, so to speak, for the future

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

{

**uint8\_t res=0;**

**return res;**

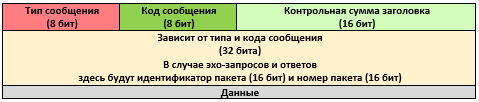
}

Put in this function while the pointer to the IP packet

uint8\_t res=0;

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

Here is the **ICMP** header format



There are several types of ICMP messages:

**0** - the echo-answer,

**3** - the destination node is unreachable,

**5** - route redirection,

**8** - echo request,

**9** - message about the router,

**10** - requesting a message about the router,

**11** - expiration of the package,

**12** - problems with the parameters,

**13** - request a timestamp,

**14** is the response time stamp.

Of all this variety of messages, we still need 2 - this is an **echo-reply (0)** and an **echo request (8)** , which are used when the ping utility is running.

The packet code in the cases of echo requests and answers is 0. In other cases there will be other codes, but we do not consider them within the framework of our lesson.

The packet checksum is calculated in the same way as in the case of IP, but not exactly. The checksum includes the entire package along with the data.

The next 2 fields are set by the host, and therefore we also do not bother with them. They are left in the echo-reply the same as they came.

The data field is usually a test set of bytes, which must also return unchanged.

ICMP packets are also used to diagnose routes in the trace.

In the net.h header file, add a structure for the header of the icmp package

} ip\_pkt\_ptr;

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

**typedef struct icmp\_pkt{**

**uint8\_t msg\_tp;//тип севриса**

**uint8\_t msg\_cd;//код сообщения**

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

**uint16\_t id;//идентификатор пакета**

**uint16\_t num;//номер пакета**

**uint8\_t data[];//данные**

**} icmp\_pkt\_ptr;**

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

Also add message types

#define IP\_UDP 17

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

**#define ICMP\_REQ 8**

**#define ICMP\_REPLY 0**

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

In the **icmp\_read** function in the **net.c**  file **,**  set the pointer to the **ICMP** packet

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

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

Filter the packet by length and type of message - echo request

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

**//Отфильтруем пакет по длине и типу сообщения - эхо-запрос**

**if ((len>=sizeof(icmp\_pkt\_ptr))&&(icmp\_pkt->msg\_tp==ICMP\_REQ))**

**{**

**}**

We will display a message in the terminal program that we have such a request

if ((len>=sizeof(icmp\_pkt\_ptr))&&(icmp\_pkt->msg\_tp==ICMP\_REQ))

{

**sprintf(str1,"icmp requestrn");**

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

}

We will collect the code, we will tell the controller and we will see the result

image40

Let's try to answer. Let's change the message request type to a response

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

**icmp\_pkt->msg\_tp=ICMP\_REPLY;**

We will calculate the checksum

icmp\_pkt->msg\_tp=ICMP\_REPLY;

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

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

Now we will create a function for sending an IP packet, since we do not need a separate function to send an ICMP packet, because we have already generated it. The function billet is similar to the function of sending a package

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

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

**{**

**uint8\_t res=0;**

**return res;**

**}**

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

In the [**next part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-10/) this lesson, we will finish the work on the answer to the ping of our ENC28J60 module.

**Lesson 68**

**Part 10**

# LAN. ENC28J60

In the [**previous part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-9/) this lesson we studied the ICMP protocol and wrote the code for receiving packets of this level.

Call this function to send an IP packet to icmp\_read

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

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

}

Let's return now to the function ip\_send and set the pointer to the IP packet

uint8\_t res=0;

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

Fill in the IP header

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

**//Заполним заголовок пакета IP**

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

**ip\_pkt->fl\_frg\_of=0;**

**ip\_pkt->ttl=128;**

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

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

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

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

And send the frame

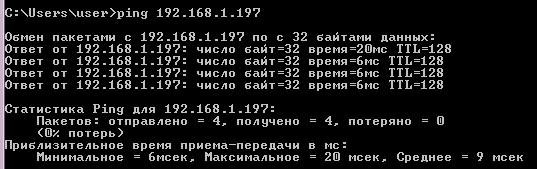
ip\_pkt->cs = checksum((void\*)ip\_pkt,sizeof(ip\_pkt\_ptr));

**//отправим фрейм**

**eth\_send(frame,len);**

return res;

We will collect the code. Let's sew the controller. Let's see the result



Everything is working!

The rest of the protocols we learn later someday, and then already and so our study has dragged on. Perhaps there will be UDP, TCP, DHCP.

In the meantime, all the creative successes. Thank you all for attention!