**Lesson 71**

**Part 1**

**LAN. ENC28J60. ARP**

Today we will continue the  topic started in [**lesson 68**](http://narodstream.ru/stm-urok-68-lan-enc28j60-chast-1/) on programming the LAN module on the **ENC28J60** chip , which allows us to study the programming of network protocols manually, while pursuing the goal of "Understanding the operation of the local network and all its levels and protocols from within - at a low level." If we achieve this goal, then in the future it will be much easier to navigate in the programming of local networks using ready-made stacks implemented in certain libraries.

Therefore, we will continue to study more deeply the ARP protocol, which allows you to exchange packets between devices at lower network levels, without using the upper layers, and, as we know from the previous session, allows you to find out the physical address of the device we are interested in, the local address of which we already know.

A quite logical question arises: "Why do we need this, we already did it in the last lesson?". And the question has an answer. We did it in one direction. That is, we provided this task only if the external device needed to know the physical address of our device. That is, we created a kind of ARP-server. But we also need an ARP-client, with which our module will already try to learn the network address of some external device. And the client in practice will probably even be needed more often, because without this we can not ask anything from other devices, since we do not know any physical address of external devices. That is, we can not even transfer anything, because, given the above, we have nowhere to transfer. So here is the answer. Therefore we continue.

We will create the project from the **ENC28J60** project and call it **ENC28J60\_ARP** .

But first, let's correct some of the shortcomings that were committed in the last lesson, and also improve our code slightly, for which we will take out all the code associated with the ARP requests and answers in a separate module.

Run the project in the  **Cube MX** . Nothing is corrected, the project will generate for the Keil, open it up, configure the programmer to avtorezet also connect our library files **enc28j60.c** and **net.c** .

We will try to collect and flash the project. If everything works fine, then we will continue our work.

First, go to the header file  **enc28j60.h** and add one more register to the 3 bank

#define MISTAT (0x0A|0x60|0x80)

**#define ECOCON (0x15|0x60)**

Let's **go** to the file  **enc28j60.c** and add the delay function in microseconds at the very top

uint8\_t macaddr[6]=MAC\_ADDR;

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

**\_\_STATIC\_INLINE void DelayMicro(\_\_IO uint32\_t micros)**

**{**

**micros \*= (SystemCoreClock / 1000000) / 5;**

**/\* Wait till done \*/**

**while (micros--) ;**

**}**

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

Now we'll go into the initialization function and perform some optimizing steps, which in my opinion have reduced the number of module hangs.

First of all, let's comment on the interrupt enable code

enc28j60\_SetBank(ECON1);

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

Further at the very end, we will halve the frequency of the generator and then wait a little

  enc28j60\_writeOp(ENC28J60\_BIT\_FIELD\_SET,ECON1,ECON1\_RXEN);//разрешаем приём пакетов

**//Включим делитель частоты генератора 2, то есть частота будет 12,5 MHz**

**enc28j60\_writeRegByte(ECOCON,0x02);**

**DelayMicro(15);**

**}**

According to the technical documentation there in general you can wait less than a microsecond, but we have nowhere to hurry and we will wait, just in case, more.

Now let's **go** to the function of sending packets  **enc28j60\_packetSend** and there we will also add some delay at the very end. It is not necessary, but in the loaded networks it worked better, which was confirmed by the experimental way

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

**//небольшая задержка, почему-то без неё не работает в загруженных сетях**

**HAL\_Delay(1);**

**}**

So far, with all the amendments.

Let's now put all the functionality of ARP into a separate module, for which we will create the files **arp.c**and **arp.h** and connect them to the project.

The initial contents of these files are standard:

**arp.c:**

**#include "arp.h"**

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

**arp.h:**

**#ifndef ARP\_H\_**

**#define ARP\_H\_**

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

**#include "stm32f1xx\_hal.h"**

**#include <string.h>**

**#include <stdlib.h>**

**#include <stdint.h>**

**#include "enc28j60.h"**

**#include "net.h"**

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

**#endif /\* ARP\_H\_ \*/**

Now we need to somehow connect this new library. And why somehow, because it's unusual, otherwise there will be mistakes.

Let's **go to the net.h** header file and connect our new module **to the bottom of the** file (this is very important)

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

**#include "arp.h"**

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

#endif /\* \_\_NET\_H \*/

Now, from the file **net.c** to the **arp.c** file, **we** first transfer all the code of the **arp\_read** function , of course, then deleting this code from the source file, also in the header file **arp.h,** we will create a prototype for this function. Also in the file **arp.c we** connect some global variables

#include "arp.h"

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

**extern UART\_HandleTypeDef huart1;**

**extern uint8\_t ipaddr[4];**

**extern uint8\_t macaddr[6];**

**extern char str1[60];**

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

The same thing we do with the **arp\_send** function and we also add a prototype to it. But since this function uses the call to the function **eth\_send** , which remains in another file, then we will also make a prototype in the file net.h **at the bottom of the** file before connecting the file arp.h

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

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

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

#include "arp.h"

Now we will consider how we will send ARP requests, that is, due to which the controller will generally receive such a command. But since we already have a USART module connected, let's use it, we will not connect any displays or buttons yet. That is, we will use the function of receiving data on USART and we will send these commands from the PC using the terminal program. Once we did this, but everything was forgotten. Here, let's at the same time and repeat how it's done.

Therefore, we now go to the file **net.h** and create there some structure with properties

} icmp\_pkt\_ptr;

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

**typedef struct USART\_prop{**

**uint8\_t usart\_buf[20];**

**uint8\_t usart\_cnt;**

**uint8\_t is\_ip;**

**} USART\_prop\_ptr;**

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

Using the type of our structure, create a global variable **called net.c**

char str1[60]={0};

**USART\_prop\_ptr usartprop;**

Now initialize the properties in the **net\_ini** initialization function **beforehand**

void net\_ini(void)

{

**usartprop.usart\_buf[0]=0;**

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

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

  HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"123456rn",8,0x1000);

Also here - in the file **net.c** - at the bottom we will add the function-interrupt handler after the reception of the specified number of bytes to the **USART** bus

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

**void UART1\_RxCpltCallback(void)**

**{**

**}**

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

Let's make a prototype in the header file, go to the main.c file and add there already an official handler function

/\* USER CODE BEGIN 4 \*/

**void HAL\_UART\_RxCpltCallback(UART\_HandleTypeDef \*huart)**

**{**

**if(huart==&huart1)**

**{**

**UART1\_RxCpltCallback();**

**}**

**}**

/\* USER CODE END 4 \*/

Also, add a global string array to main.c

/\* Private variables ---------------------------------------------------------\*/

**char str[20];**

/\* USER CODE END PV \*/

In main (), we call the USART receive function

net\_ini();

**HAL\_UART\_Receive\_IT(&huart1,(uint8\_t\*)str,1);**

/\* USER CODE END 2 \*/

Go back to the file net.c and connect there the same global string array

char str1[60]={0};

**extern char str[20];**

USART\_prop\_ptr usartprop;

In function  **UART1\_RxCpltCallback** we will write the received byte into a variable

void UART1\_RxCpltCallback ( void )

{

**uint8\_t b;**

**b = str [0];**

In the [**next part of the**](http://narodstream.ru/stm-urok-68-lan-enc28j60-arp-chast-2/) lesson we will write the code by receiving the desired IP address for the ARP request from the terminal program in a string form, then we convert this address into a numeric array and start writing the function of sending the ARP request to the network.

**Lesson 71**

**Part 2**

# LAN. ENC28J60. ARP

In the [**previous part of the**](http://narodstream.ru/stm-urok-68-lan-enc28j60-arp-chast-1/) lesson, we set up the project, corrected the mistakes of the last lesson, took out the functions of implementing the ARP protocol in a separate module and created the USART interrupt handler for the reception.

Now we start to process the result, changing all the statuses in time, and in the end we will call the reception from USART

  b = str[0];

**//если вдруг случайно превысим длину буфера**

**if (usartprop.usart\_cnt>20)**

**{**

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

**}**

**else if (b == 'a')**

**{**

**usartprop.is\_ip=1;//статус отрпавки ARP-запроса**

**}**

**else**

**{**

**usartprop.usart\_buf[usartprop.usart\_cnt] = b;**

**usartprop.usart\_cnt++;**

**}**

**HAL\_UART\_Receive\_IT(&huart1,(uint8\_t\*)str,1);**

}

I think there's nothing to explain, so everything is clear. The fact that there are a lot of lines is nothing, I simply used curly brackets everywhere, regardless of the number of lines of code, as long as we work only with ARP, and in the future, I think other protocols will be tightened, accordingly the statuses will appear and we will not have to add anything there is for the simplicity of writing the future code. And the symbol "a" we track to ensure that the entered line, we will finish it, to somehow track the end of this line. That is, we will enter the IP address of the device in the terminal, the MAC address of which we want to know.

Now let's catch the status change in the function **net\_poll** , creating in it a local four-element array

uint16\_t len;

**uint8\_t ip[4]={0};**

Next, after receiving the next packet, we begin to monitor the status in this function

    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);**

**usartprop.is\_ip = 0;**

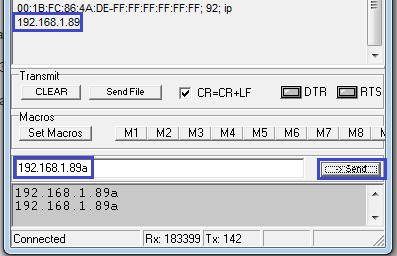
**usartprop.usart\_cnt=0;**

**}**

}

At the same time, we will output here the line that was received from it and again reset all our statuses in order to normally wait for the next line from the **USART** .

Let's collect the code, we'll tell the controller and try to enter something into the terminal program



As you can see, everything turned out for us. We are currently entering the ip-address of our PC or some other one and get it in our program as a string.

But we can not send the ARP request yet, since we now need to convert this address from the string into a four-element unsigned integer array.

To do this, we add in the same module the corresponding function somewhere above the function eth\_read, adding to it several local variables. The result of the function will be written to **ipextr**

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

**//функция преобразования строкового значения IP в 32-битное числовое**

**void ip\_extract(char\* ip\_str,uint8\_t len, uint8\_t\* ipextr)**

**{**

**uint8\_t offset = 0;**

**uint8\_t i;**

**char ss2[5] = {0};**

**char \*ss1;**

**int ch = '.';**

**}**

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

Now let's run through the function for the appearance of points

int ch = '.';

**for(i=0;i<3;i++)**

**{**

**ss1 = strchr(ip\_str,ch);**

**}**

Again, we use a new interesting function that monitors the entry of a character in the string added as the first argument, and then returns the address in the memory of the occurrences of that character. Only it is more convenient for us to work not with the address but with an offset relative to the beginning of the line with the IP address, so we calculate this offset

ss1 = strchr(ip\_str,ch);

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

Then we will copy another line of our line to the point, and then finish it with zero, as requested by the function that we will use later

offset = ss1-ip\_str+1;

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

**ss2[offset]=0;**

And then we, converting the above-generated string into a number, write it to the corresponding element of the array for the address

ss2[offset]=0;

**ipextr[i] = atoi(ss2);**

Next, we move the pointers of our line with the IP address and the length of this line accordingly, and then we leave the **for** loop

  ipextr[i] = atoi(ss2);

**ip\_str+=offset;**

**len-=offset;**

}

Now, after leaving the loop, we are already working with the last byte of the IP address, after which there is already no dot and also write it to the corresponding element of the returned array of the IP address

    len-=offset;

  }

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

**ss2[len]=0;**

**ipextr[3] = atoi(ss2);**

}

We continue to write the body of the function **net\_pool** , using the above function, inserting the code in the appropriate place in this function, also here we display the result of our function in the USART port in order to verify the correctness of writing the code of the above function

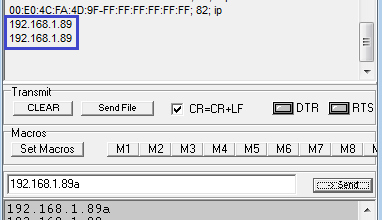
HAL\_UART\_Transmit(&huart1,(uint8\_t\*)"\r\n",2,0x1000);

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

**sprintf(str1,"%d.%d.%d.%d\r\n", ip[0],ip[1],ip[2],ip[3]);**

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

Let's check this by collecting the code and flashing the controller and then typing in the same string that you typed in the terminal program



The function works great.

Now go to the arp.c file and start writing the ARP request function there, because for this we have everything ready now

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

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

**{**

**uint8\_t i;**

**return 1;**

**}**

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

We add a prototype of this function and call it in net.c in **net\_pool** function . We'll write the function call instead of sending the ip address to USART, we do not need it there, since it was just a test of the function of converting the string value of an IP address to an integer array

~~sprintf(str1,"%d.%d.%d.%d\r\n", ip[0],ip[1],ip[2],ip[3]);~~

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

**arp\_request(ip);**

Also, we'll go into the file **net.h** and add there macros for zero and broadcast MAC addresses

#define IP\_ADDR {192,168,1,197}

**#define MAC\_BROADCAST {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF}**

**#define MAC\_NULL {0x00,0x00,0x00,0x00,0x00,0x00}**

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

In the [**next part of the**](http://narodstream.ru/stm-urok-68-lan-enc28j60-arp-chast-3/) lesson we will send an ARP request, and also start writing the function of populating the ARP table.

**Lesson 71**

**Part 3**

# LAN. ENC28J60. ARP

In the [**previous part of**](http://narodstream.ru/stm-urok-68-lan-enc28j60-arp-chast-2/)  this lesson, we wrote the code for receiving the desired IP address for the ARP request from the terminal program in a string form, then converted the address into a numeric array and started writing the function of sending the ARP request to the network.

We return to **arp.c** and connect the global variable of our buffer to again save on memory, and also add zero and broadcast MAC-aders

extern char str1[60];

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

**uint8\_t macbroadcast[6]=MAC\_BROADCAST;**

**uint8\_t macnull[6]=MAC\_NULL;**

Continue our function of sending an ARP request.

After we created the variable, we must first determine that in our ARP-table, which we will create a bit later, there is not yet a match record for the researched IP-address of the finished MAC-address, which we once defined and saved in the table . But while we do not have a table, we will skip this code, and we will add it a little later, because the table will be filled in a different way in another case, and, accordingly, we have nothing to take. Therefore, we will create a package for ARP-request and send it to the network

  uint8\_t i;

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

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

**msg->net\_tp = ARP\_ETH;**

**msg->proto\_tp = ARP\_IP;**

**msg->macaddr\_len = 6;**

**msg->ipaddr\_len = 4;**

**msg->op = ARP\_REQUEST;**

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

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

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

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

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

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

**frame->type = ETH\_ARP;**

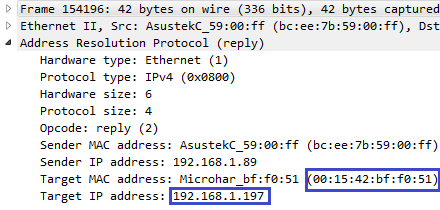
**enc28j60\_packetSend((void\*)frame,sizeof(arp\_msg\_ptr) + sizeof(enc28j60\_frame\_ptr));**

  return 1;

In principle, we can already verify the sending of the package using the Wireshark program. To do this, we'll compile the code, we'll tell the controller and we'll send the line with the address from the terminal program, before doing this, do not forget to start the Wireshark program and apply the arp filter there, just as we did in the **last lesson** . Here's what we need to see

image02

We see that our request came safely, and the computer sent a response, which we have not processed yet. Let's see that the answer was sent to us



We see in the quality of the receiver our physical and local addresses. Excellent!

Now we need to process the response of the PC.

For this zaydom file **arp.c** and function **arp\_read** we change the code here

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

{

  . . .

}

We will separate each condition from each other. It will be like this

**if(!memcmp(msg->ipaddr\_dst,ipaddr,4))**

**{**

**if(msg->op==ARP\_REQUEST)**

**{**

**. . .**

**}**

**}**

After the body of the condition of having an ARP request, we will create one more condition for having an ARP response. The code will be somewhat similar, only the variable **res** we will expose in 2

        res=1;

      }

**else if(msg->op==ARP\_REPLY)**

**{**

**sprintf(str1,"rnreplyrnmac\_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,"mac\_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=2;**

**}**

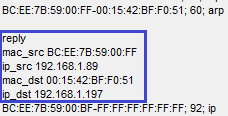
    }

  }

}

return res;

We will collect the code, we will tell the controller, we will send the same line in the terminal program, as a result we should observe here such here picture



Excellent! We received a response from a PC, or from any device that can respond to ARP requests.

Let's return to **net.c** and correct the function of reading Ethernet packets **eth\_read** . First we add a local variable to it

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

{

**uint8\_t res=0;**

Then fix the code here

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

{

  . . .

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

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

  if(res==1)

  {

    arp\_send(frame);

  }

  else if(res==2)

  {

  }

}

Now we need the function to populate the arp table. But first we need to create this table.

The ARP table stores the IP addresses of already known devices with their corresponding MAC addresses. The data of the table can be of different dimensions. Given our requests and our computing power, we will make this table into 5 records. The entry will store the IP address and MAC address of the device and the time in seconds that passed since the controller was turned on. Time is needed in order not to store too old records, since during this time the IP address can be assigned to another device. We will store the table in normal RAM. We do not need to store it in non-volatile memory due to uselessness. Also, such a memory is less fast and access to this table will take another time and the code will get harder. Of course, you can fully implement this mechanism, since we are able to work with such a memory.

First, create a structure for the table in the file **arp.h**

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

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

**typedef struct arp\_record{**

**uint8\_t ipaddr[4];**

**uint8\_t macaddr[6];**

**uint32\_t sec; //какое было количество секунд в переменной clock\_cnt, когда была сделана запись**

**} arp\_record\_ptr;**

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

Then create an array of 5 elements in the file **arp.c,** as well as another variable to store the current fill position in the table

uint8\_t macnull[6]=MAC\_NULL;

**arp\_record\_ptr arp\_rec[5];**

**uint8\_t current\_arp\_index=0;**

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

Now add the function of filling this table

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

**void arp\_table\_fill(enc28j60\_frame\_ptr \*frame)**

**{**

**uint8\_t i;**

**}**

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

We add a prototype to this function and call it in **net.c** in the **eth\_read** function in the place where we left the empty body condition

else if (res == 2)

{

**arp\_table\_fill (frame);**

}

Let's return to arp.c and continue to write our function to populate the ARP table.

Add a pointer to the ARP message

uint8\_t i;

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

Let's start adding a record

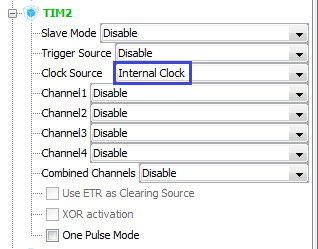
arp\_msg\_ptr \*msg = (void\*)frame->data;

**memcpy(arp\_rec[current\_arp\_index].ipaddr,msg->ipaddr\_src,4);**

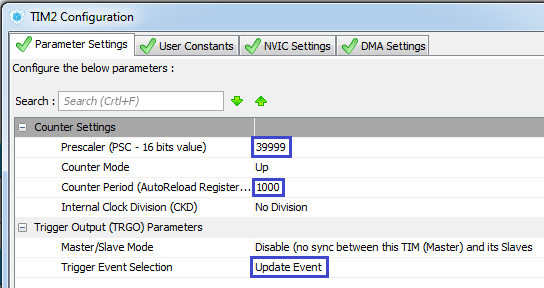
**memcpy(arp\_rec[current\_arp\_index].macaddr,msg->macaddr\_src,6);**

Now we have reached the point where we need to add the number of seconds to the row of our ARP table. After the controller power was turned on. You can use several methods - enable RTC, enable timer, use a variable that counts cycles, etc. I decided to use the timer, at the same time and repeat the work with the timer. Maybe later we will use time from the Internet, obtained with the help of NTP, but as long as the timer, since these are already other levels, and we have not reached them yet.

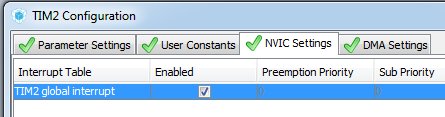
Save and close the project in Keil and go to the Cube MX project. Turn on there timer 2



Let's configure it in "Configuration" as follows



Enable interrupts



Generate the project for Keil, open it and just in case collect.

Let's return to **net.c** and add a global variable

uint8\_t ipaddr[4]=IP\_ADDR;

**uint32\_t clock\_cnt=0;//счетчик секунд**

Add the function-handler for the interrupt by the coincidence of this timer in the same file

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

**void TIM\_PeriodElapsedCallback(void)**

**{**

**//считаем секунды и записываем их в clock\_cnt**

**clock\_cnt++;**

**}**

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

Let's make a prototype for this function and add another real interrupt handler in the main.c file, in which we call our handler

**void HAL\_TIM\_PeriodElapsedCallback(TIM\_HandleTypeDef \*htim)**

**{**

**if(htim==&htim2)**

**{**

**TIM\_PeriodElapsedCallback();**

**}**

**}**

/\* USER CODE END 4 \*/

Also, do not forget to run our timer in **main ()**

HAL\_UART\_Receive\_IT(&huart1,(uint8\_t\*)str,1);

**HAL\_TIM\_Base\_Start\_IT(&htim2);**

/\* USER CODE END 2 \*/

In the [**next part of the**](http://narodstream.ru/stm-urok-68-lan-enc28j60-arp-chast-4/) lesson, we will finish the function of sending the ARP request, and also write a code that will not allow duplication and expiration of records in the ARP table.

**Lesson 71**

**Part 4**

# LAN. ENC28J60. ARP

In the [**previous part of the**](http://narodstream.ru/stm-urok-68-lan-enc28j60-arp-chast-3/) lesson, we sent an ARP request, and also started writing the function of populating the ARP table.

To check that our timer counts down seconds, you can add an LED flasher to this handler, but I already tested it, so we will not deal with it.

Let's return to the **arp.c** file and connect our meter

extern char str1[60];

**extern uint32\_t clock\_cnt;//счетчик секунд**

We continue the function **arp\_table\_fill** and add to the table entry the number of seconds that the timer counted

memcpy(arp\_rec[current\_arp\_index].macaddr,msg->macaddr\_src,6);

**arp\_rec[current\_arp\_index].sec = clock\_cnt;**

Further increase the position of the record in the table

arp\_rec[current\_arp\_index].sec = clock\_cnt;

**if(current\_arp\_index<4) current\_arp\_index++;**

**else current\_arp\_index=0;**

And at the end of the function let's see this table

  else current\_arp\_index=0;

**//смотрим ARP-таблицу**

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

**{**

**sprintf(str1,"%d.%d.%d.%d - %02X:%02X:%02X:%02X:%02X:%02X - %lurn",**

**arp\_rec[i].ipaddr[0],arp\_rec[i].ipaddr[1],arp\_rec[i].ipaddr[2],arp\_rec[i].ipaddr[3],**

**arp\_rec[i].macaddr[0],arp\_rec[i].macaddr[1],arp\_rec[i].macaddr[2],**

**arp\_rec[i].macaddr[3],arp\_rec[i].macaddr[4],arp\_rec[i].macaddr[5],**

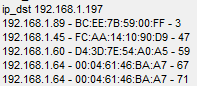
**(unsigned long)arp\_rec[i].sec);**

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

**}**

}

We will collect the code, we will tell the controller and try to send ARP-requests from the terminal program to various addresses that we know in our network



As we can see, the address mapping table is normally filled in, only if we send a request to an existing IP, it is added to us again. This is a mess. So let's return to **arp\_request** ARP request function now, as we wanted, and for now insert a loop there that will search through the records for finding the existing IP address

uint8\_t i;

**//проверим, может такой адрес уже есть в таблице ARP, а заодно и удалим оттуда просроченные записи**

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

**{**

**}**

In the body of the cycle we add condition

for(i=0;i<5;i++)

{

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

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

**{**

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

**}**

}

From the commentary it is quite clear what we have done here. We zeros all the memory allocated to the record.

Now, after this condition, we add the following, after the fulfillment of the body of the condition (not of the cycle)

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

}

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

**{**

**return 0;**

**}**

In the body of this condition, before we exit the function with return 0, we look at our table

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

{

**//смотрим ARP-таблицу**

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

**{**

**sprintf(str1,"%d.%d.%d.%d - %02X:%02X:%02X:%02X:%02X:%02X - %lurn",**

**arp\_rec[i].ipaddr[0],arp\_rec[i].ipaddr[1],arp\_rec[i].ipaddr[2],arp\_rec[i].ipaddr[3],**

**arp\_rec[i].macaddr[0],arp\_rec[i].macaddr[1],arp\_rec[i].macaddr[2],**

**arp\_rec[i].macaddr[3],arp\_rec[i].macaddr[4],arp\_rec[i].macaddr[5],**

**(unsigned long)arp\_rec[i].sec);**

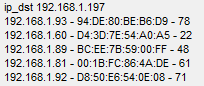
**HAL\_UART\_Transmit(&huart1,(uint8\_t\*)str1,strlen(str1),0x1000);**

**}**

  return 0;

}

We will collect the code, we will impose the controller and the program with the requests



The IP addresses in the table are no longer duplicated, no existing requests are sent to existing addresses. 12 hours I did not wait, but I tried the maximum number of seconds in the condition to change to a much smaller - obsolete addresses were deleted. So everything works.

Thus, we have now created a fully-fledged ARP client server, which not only responds to ARP requests, but also knows how to send them.