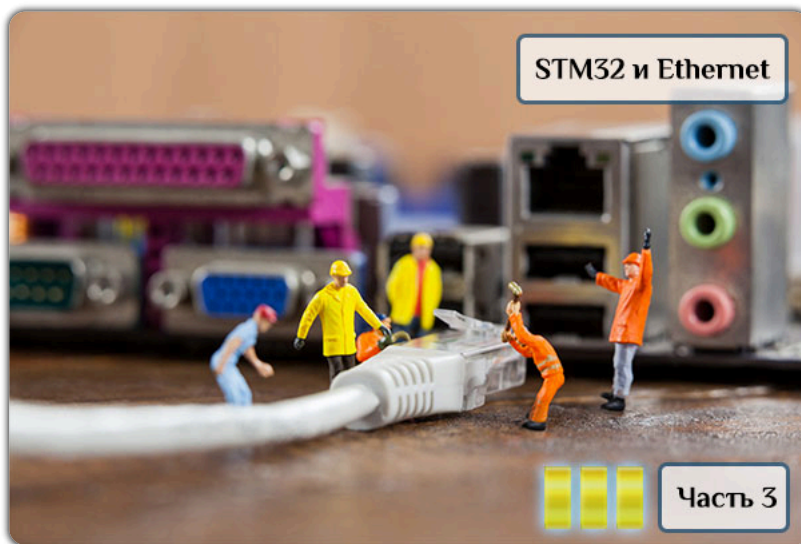


STM32 and Ethernet. Part 3. Channel level. ARP protocol.

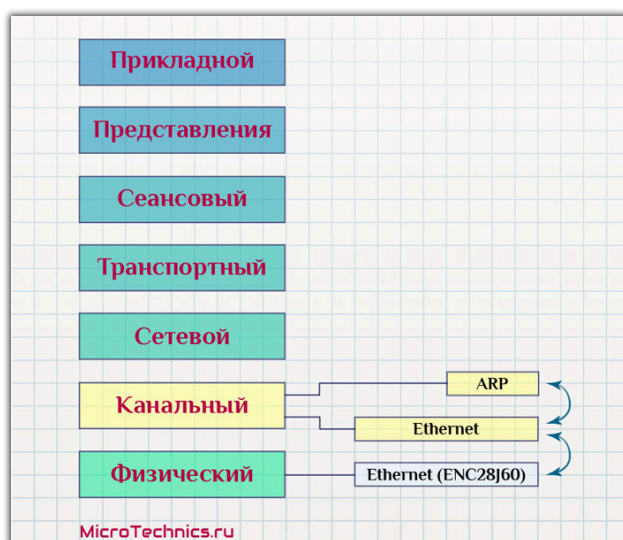


Aveal MT | 10/07/2021 | 0 | STM32, STM32CubeMx

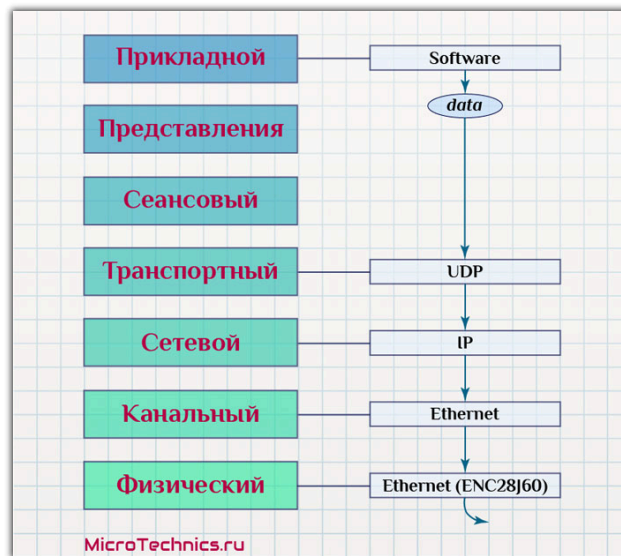


Good day everyone, this week will be Ethernet week, since it's time to reach some more or less final point 😊 We are moving up the protocol stack, today we need to disassemble the accepted Ethernet frames, generate responses and get involved in the meantime **ARP protocol**. Let's start about all this below.

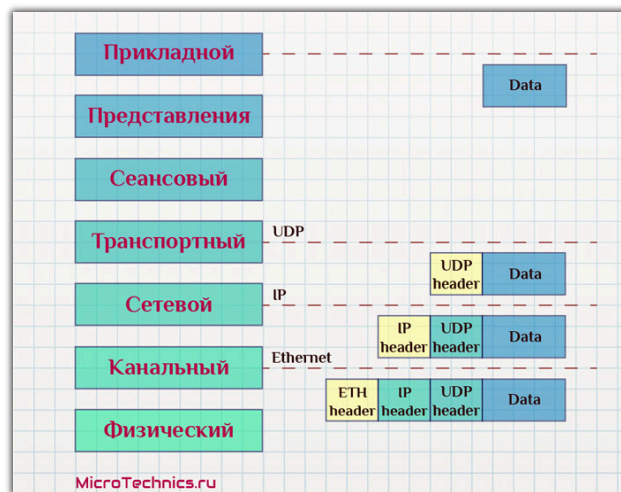
Summarizing the above and discussed earlier, we can say that today's activities will be entirely devoted to work at the channel level:



In general, behind the names of levels and the corresponding terminology - up/down the stack, the level above/below - the very essence of these processes, which is actually the simplest, is often lost. Let's look at the protocol stack:



At the application level, let us have N bytes of data that need to be output outside in the direction indicated by the arrow. This is what a typical task of sending data using a particular protocol stack looks like. So, physically, the process of passing data through the layers is just adding service information (headers) to it (to the data) for each of the protocols used:

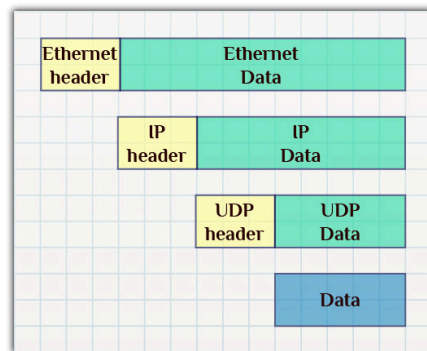


The reception process proceeds similarly. They took an Ethernet frame:



The format of the header is known, we analyze it and find out what kind of data is contained in the field *Ethernet Data*. For our example, this is an IP packet that also contains its own header. And the process of "unfolding" the received data continues in the same vein:





Thus, when implementing data reception over Ethernet, we will extract data from the received sequence of bytes by dropping the headers of protocols encountered along the path. When transmitting data, we will wrap the original payload data in headers with the service information of certain protocols. To understand the general meaning of ongoing processes, such an interpretation, albeit simplified, can be very useful.

The idea has been sorted out, we move on to implementation, and start by adding functionality to the project for working with Ethernet frames.

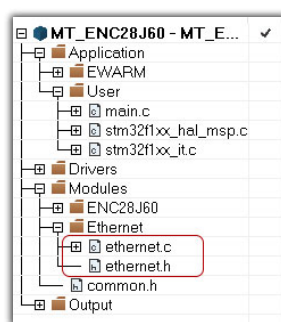
Processing of received Ethernet frames.

There is no point in dwelling on theory; the format of the package was probably discussed [10 times already](#):



We move on to the project, which we will use as a basis the version from [the previous part about Ethernet](#). We have fully implemented the work with ENC28J60; now we will directly process and prepare the data. We add two new files to the project:

- `ethernet.c`
- `ethernet.h`



The architecture will be structured as follows. Let's add a function:

```
1. void ETH_Process(ENC28J60_Frame* encFrame)
```

It takes as its only argument a pointer to an instance of the structure we added earlier to the draft `ENC28J60_Frame`. IN THE `main.c` we add a variable and, in the main loop of the program, a call to this very function:

```
1. /* USER CODE BEGIN PV */
2. ENC28J60_Frame frame;
3.
4. /* USER CODE END PV */
```

```
1. /* Infinite loop */
2. /* USER CODE BEGIN WHILE */
```

```

3. while(1)
4. {
5.     ETH_Process(&frame);
6.     /* USER CODE END WHILE */
7.
8.     /* USER CODE BEGIN 3 */
9. }
10. /* USER CODE END 3 */

```

The entire project, as well as the full file code, can traditionally be found at the end of the article, so you can connect the file `ethernet.h` and I don't dwell on similar points separately in the text.

Total, function `ETH_Process()` it is the heart of our program, so in it we must carry out all data processing. And first of all, let's announce the storage structure **Ethernet frames**:

```

1. typedef struct ETH_Frame
2. {
3.     uint8_t destMacAddr[MAC_ADDRESS_BYTES_NUM];
4.     uint8_t srcMacAddr[MAC_ADDRESS_BYTES_NUM];
5.     uint16_t etherType;
6.     uint8_t data[];
7. } ETH_Frame;

```

As you can see, the structure fields exactly correspond to the frame format, which is not surprising 👍 Now the minimum functionality is next `ETH_Process()`:

```

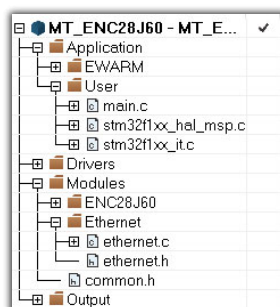
1. /*-----*/
2. void ETH_Process(ENC28J60_Frame* encFrame)
3. {
4.     uint16_t responseSize = 0;
5.     uint16_t requestSize = ENC28J60_ReceiveFrame(encFrame);
6.
7.     if(requestSize > 0)
8.     {
9.         ETH_Frame* ethFrame = (ETH_Frame*)encFrame->data;
10.        uint16_t etherType = ntohs(ethFrame->etherType);
11.    }
12. }
13.
14.
15.
16. /*-----*/

```

In a step-by-step approach:

- We receive the packet using the function intended for this purpose `ENC28J60_ReceiveFrame()`. It was created in the same one [previous article of the series](#).
- The function will return 0 if no new frames are found, which we check with the subsequent condition - `if(requestSize > 0)`.
- If a new frame is available, further processing follows, which at this stage consists only of extracting the field `EtherType` from accepted data.

A small and not very pleasant nuance - the network uses big endian, but we work with little endian, so there is nothing left to do but create macros for these transformations. Let's take them to a separate file (`common.h`), because in the future we will obviously need them from different modules/parts/files of the program. For the same reason, we will add variable declarations to this file that store MAC and IP addresses:



+ Common.h file

- `htons` - little endian to big endian - host to network short (2 bytes)
- `htonl` - little endian to big endian - host to network long (4 bytes)
- `ntohs` - big endian to little endian - network to host short (2 bytes)
- `ntohl` - big endian to little endian - network to host long (4 bytes)

IN THE `ethernet.h` let's connect:

```
1. #include "common.h"
```

So, you have accepted the Ethernet frame, then you should process it and send a response. Let's start, contrary to logic, by sending a response:

```
1.  /*-----*/
2.  static void ETH_Response(ETH_Frame* ethFrame, uint16_t len)
3.  {
4.      memcpy(ethFrame->destMacAddr, ethFrame->srcMacAddr, MAC_ADDRESS_BYTES_NUM);
5.      memcpy(ethFrame->srcMacAddr, macAddr, MAC_ADDRESS_BYTES_NUM);
6.
7.      ENC28J60_TransmitFrame((uint8_t*)ethFrame, len + sizeof(ETH_Frame));
8.  }
9.
10.
11.
12. /*-----*/
```

This type has a function for sending a response to the frame. One useful activity here is reordering MAC addresses. Indeed, the sender from the original frame now becomes the receiver. We now use our own as the sender's address **MAC address**, which is stored in the array `macAddr[]`. After that, we launch the response packet:

```
1. ENC28J60_TransmitFrame((uint8_t*)ethFrame, len + sizeof(ETH_Frame));
```

And now in `ETH_Process()` we have:

```
1.  /*-----*/
2.  void ETH_Process(ENC28J60_Frame* encFrame)
3.  {
4.      uint16_t responseSize = 0;
5.      uint16_t requestSize = ENC28J60_ReceiveFrame(encFrame);
6.
7.      if(requestSize > 0)
8.      {
9.          ETH_Frame* ethFrame = (ETH_Frame*)encFrame->data;
10.         uint16_t etherType = ntohs(ethFrame->etherType);
11.
12.         //Protocols processing should be there
13.
14.         if(responseSize > 0)
15.         {
16.             ETH_Response(ethFrame, responseSize);
17.         }
18.     }
19. }
20.
21.
22.
23. /*-----*/
```

What's important here is the process of generating a response packet, because directly in the send function we only change the MAC addresses in the header. It is understood that by pointer `ethFrame` the data intended to be sent is already located in the variable `responseSize` is the length of this data. That is, the complete process is structured as follows:

1. We accept a new frame using `ENC28J60_ReceiveFrame`.
2. We process the received data and generate a response packet, while using the same pointer to the same frame `ethFrame`, and also a variable `responseSize`, where we should put the size of the data in the response frame.
3. We send a response using `ETH_Response()`.

Step 2 is currently missing from the code; when we implement it, everything will finally fall into place.

ARP protocol.



But before moving on, you need to test what has been done. To do this, we will set it in the file `enc28j60.c` IP address of our board and then ping it from the command line. **IP address:**

```
1. uint8_t ipAddr[IP_ADDRESS_BYTES_NUM] = {169, 254, 191, 22};
```

```
PS C:\WINDOWS\system32> ping 169.254.191.22
```

Briefly about what will happen online during these actions...

So, the PC receives a signal to ping a specific IP address, but to send an Ethernet frame it needs to know **MAC address** from the target device, not **IP**. And then it comes into play **ARP protocol**, which is designed to obtain the device's MAC address from a known IP address. Thus, in this case, we have a typical situation where device-1 wants to send data to device-2 (in this role ENC28J60), knowing only its IP address.

To solve this problem, device-1 sends a broadcast request to the network, that is, an Ethernet frame in which the broadcast MAC address is indicated as the recipient - `FF:FF:FF:FF:FF:FF`. All devices on the network accept such packets, including our device-2. The essence of the package can be expressed in words as follows: "Device with IP `xxx.xxx.xxx.xxx`, tell your MAC address to the device with the MAC address `YY:YY:YY:YY:YY`". In our particular case `xxx.xxx.xxx.xxx` is the IP address that we set in the project and are trying to ping, and `YY:YY:YY:YY:YY` This is the address of the PC from which we execute the command `ping`.

As a result of the above actions, device-2 received **ARP request**, must perform the sending of a response containing its own MAC address. After which device-1 will be able to easily send Ethernet frames in the direction of the MAC address that has become known to it.

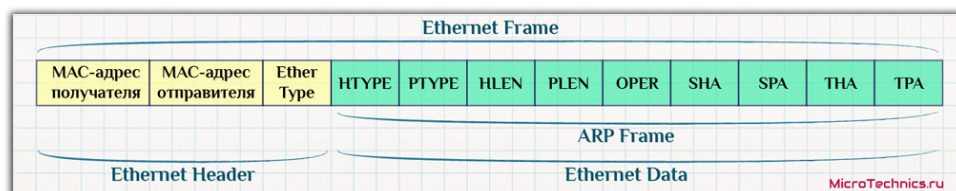
We have to do all this in the project, but for now we have a few more words about the structure of ARP packages:



Let's take a look at the fields:

- **HTYPE** This stores the code that matches the link layer protocol used. We have Ethernet, Ethernet - code 0x0001.
- **PTYPE** The code for the network protocol, IPv4, is 0x0800.
- **HLEN** - length *physical* addresses, number of bytes. This role is played by the MAC address, 6 bytes (0x06).
- **PLEN** - length *logical* addresses, number of bytes. This role is played by the IP address, 4 bytes (0x04).
- **OPERA** The operation code is 0x0001 - ARP request, 0x0002 - response to request.
- **SHA** - Sender Hardware Address - Sender MAC address.
- **SPA** - Sender Protocol Address - Sender IP address.
- **THA** - Target Hardware Address - Recipient's MAC address.
- **TPA** Target Protocol Address - The recipient's IP address.

The total byte sequence in the network is as follows:

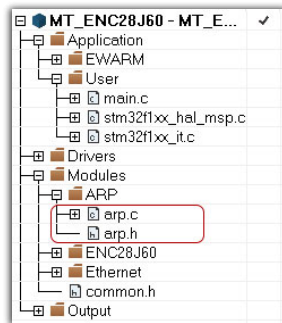


As we discussed at the very beginning of this article, the whole point of passing data through various layers of the OSI network model is, by and large, to add the headers of the protocols used to the "useful" data.

Today we will ensure that ARP requests are received and responses are sent; in the future we will need to add functionality for sending packets to other network nodes, which is implemented using ARP tables, but more on that today.

That's it, let's move on to the code. Adding 2 more files to the project:

- `arp.c`
- `arp.h`



Similar to what we did earlier, we will add a structure for the ARP protocol:

```

1. typedef struct ARP_Frame
2. {
3.     uint16_t hType;
4.     uint16_t pType;
5.     uint8_t hLen;
6.     uint8_t pLen;
7.     uint16_t opCode;
8.     uint8_t srcMacAddr[MAC_ADDRESS_BYTES_NUM];
9.     uint8_t srcIpAddr[IP_ADDRESS_BYTES_NUM];
10.    uint8_t destMacAddr[MAC_ADDRESS_BYTES_NUM];
11.    uint8_t destIpAddr[IP_ADDRESS_BYTES_NUM];
12. } ARP_Frame;

```

The fields are in the same order as in the diagram just above. We write in `arp.c` code for data processing:

```

1. /* Functions -----
2. uint16_t ARP_Process(ARP_Frame* arpFrame, uint16_t frameLen)
3. {
4.     uint16_t newFrameLen = 0;
5.
6.     if(memcmp(arpFrame->destIpAddr, ipAddress, IP_ADDRESS_BYTES_NUM) == 0)
7.     {
8.         if(arpFrame->opCode == ntohs(ARP_OP_CODE_REQUEST))
9.         {
10.            memcpy(arpFrame->destMacAddr, arpFrame->srcMacAddr, MAC_ADDRESS_BYTES_NUM);
11.            memcpy(arpFrame->srcMacAddr, macAddr, MAC_ADDRESS_BYTES_NUM);
12.
13.            memcpy(arpFrame->destIpAddr, arpFrame->srcIpAddr, IP_ADDRESS_BYTES_NUM);
14.            memcpy(arpFrame->srcIpAddr, ipAddress, IP_ADDRESS_BYTES_NUM);
15.
16.            arpFrame->opCode = htons(ARP_OP_CODE_RESPONSE);
17.            newFrameLen = frameLen;
18.        }
19.    }
20.
21.    return newFrameLen;
22. }
23.
24.
25.
26. /*-----*/

```

We understand it step by step and in detail... Arguments for the function:

- `arpFrame` - pointer to frame data. We must analyze this data and, if necessary, change it. Next, they will be used in the file `ethernet.c` when sending a response, but more on that later.
- `frameLen` is the full length of the ARP packet.

In a variable `newFrameLen` we will place the size of the "new" ARP frame that is intended to be sent. Looking ahead, in this case the length of the ARP request is exactly the same as the length of our response:

```

1. newFrameLen = frameLen;

```

We will perform a two-stage check of the received data:

- First, let's check that the recipient's IP address contained in the received data matches our IP, which is stored in the variable `ipAddr`:

```
1. if(memcmp(arpFrame->destIpAddr, ipAddr, IP_ADDRESS_BYTES_NUM) == 0)
```

- And secondly, let's check the operation code:

```
1. if(arpFrame->opCode == ntohs(ARP_OP_CODE_REQUEST))
```

Our permissible values are prudently defined in *arp.h*:

```
1. #define ARP_OP_CODE_REQUEST 0x0001
2. #define ARP_OP_CODE_RESPONSE 0x0002
```

Again, I omit some minor points in the text of the article for ease and integrity of reading. The full code and link to the project can always be found at the end of the article.

We begin to generate a response message. To do this, we swap the MAC and IP addresses of the sender and recipient:

```
1. memcpy(arpFrame->destMacAddr, arpFrame->srcMacAddr, MAC_ADDRESS_BYTES_NUM);
2. memcpy(arpFrame->srcMacAddr, macAddr, MAC_ADDRESS_BYTES_NUM);
3.
4. memcpy(arpFrame->destIpAddr, arpFrame->srcIpAddr, IP_ADDRESS_BYTES_NUM);
5. memcpy(arpFrame->srcIpAddr, ipAddr, IP_ADDRESS_BYTES_NUM);
```

And the final step is to replace the opcode: was `ARP_OP_CODE_REQUEST` - request, will `ARP_OP_CODE_RESPONSE` - the answer is:

```
1. arpFrame->opCode = htons(ARP_OP_CODE_RESPONSE);
```

At this stage in part **ARP protocol** everything is ready for practical use; all that remains is to add a call to the created function at the level above - that is, in the file *ethernet.c*:

```
1. /*-----*/
2. void ETH_Process(ENC28J60_Frame* encFrame)
3. {
4.     uint16_t responseSize = 0;
5.     uint16_t requestSize = ENC28J60_ReceiveFrame(encFrame);
6.
7.     if(requestSize > 0)
8.     {
9.         ETH_Frame* ethFrame = (ETH_Frame*)encFrame->data;
10.        uint16_t etherType = ntohs(ethFrame->etherType);
11.
12.        //ARP protocol
13.        if(etherType == ETH_FRAME_TYPE_ARP)
14.        {
15.            uint16_t arpFrameLen = requestSize - sizeof(ETH_Frame);
16.            responseSize = ARP_Process((ARP_Frame*)ethFrame->data, arpFrameLen);
17.        }
18.
19.        if(responseSize > 0)
20.        {
21.            ETH_Response(ethFrame, responseSize);
22.        }
23.    }
24. }
25.
26.
27.
28. /*-----*/
```

As you can see, in the space we have prudently allocated for processing received data, we place a couple of new lines - everything is clear, systematic and transparent:

```
1. //ARP protocol
2. if(etherType == ETH_FRAME_TYPE_ARP)
3. {
4.     uint16_t arpFrameLen = requestSize - sizeof(ETH_Frame);
5.     responseSize = ARP_Process((ARP_Frame*)ethFrame->data, arpFrameLen);
6. }
```



When working with other protocols, for example, IP, UDP, etc., the design architecture will remain unchanged, we will simply add a call to the corresponding handlers `ETH_Process()` 🍌

We assemble the project, flash it, launch it, and watch it. Let's breakpoint:

```

36  /*-----*/
37  void ETH_Process(ENC28J60_Frame* encFrame)
38  {
39      uint16_t responseSize = 0;
40      uint16_t requestSize = ENC28J60_ReceiveFrame(encFrame);
41
42      if (requestSize > 0)
43      {
44          ETH_Frame* ethFrame = (ETH_Frame*)encFrame->data;
45          uint16_t etherType = ntohs(ethFrame->etherType);
46
47          // ARP protocol
48          if (etherType == ETH_FRAME_TYPE_ARP)
49          {
50              uint16_t arpFrameLen = requestSize - sizeof(ETH_Frame);
51              responseSize = ARP_Process((ARP_Frame*)ethFrame->data, arpFrameLen);
52          }
53
54          ETH_Response(ethFrame, responseSize);
55      }
56  }
57
58
59
60  /*-----*/

```

Now, let's send the command from the console again:

```
1. ping 169.254.191.22
```

Once we hit a break, we evaluate what kind of data we have in the variable `frame`:

Expression	Value		Expression	Value	
frame	<struct>		[28]	0xA9	
nextPtr	0x069A		[29]	0xFE	169.254.191.234 SPA (Sender Protocol Address)
length	64		[30]	0xBF	
status	0x00C0		[31]	0xEA	
data	<array>		[32]	0x00	
[0]	0x00		[33]	0x17	THA (Target Hardware Address)
[1]	0x17	00:17:22:ED:A5:01 MAC-адрес получателя	[34]	0x22	
[2]	0x22		[35]	0xED	
[3]	0xED		[36]	0xA5	
[4]	0xA5		[37]	0x01	
[5]	0x01		[38]	0xA9	169.254.191.22 TPA (Target Protocol Address)
[6]	0x44		[39]	0xFE	
[7]	0x8A	44:8A:5B:ED:E9:54 MAC-адрес отправителя	[40]	0xBF	
[8]	0x5B		[41]	0x16	
[9]	0xED		[42]	0x00	
[10]	0xE9		[43]	0x00	
[11]	0x54		[44]	0x00	
[12]	0x08	EtherType	[45]	0x00	
[13]	0x06		[46]	0x00	
[14]	0x00		[47]	0x00	
[15]	0x01	HTYPE	[48]	0x00	
[16]	0x08		[49]	0x00	
[17]	0x00	PTYPE	[50]	0x00	
[18]	0x06		[51]	0x00	
[19]	0x04	HLEN	[52]	0x00	
[20]	0x00	PLEN	[53]	0x00	
[21]	0x01	OPER	[54]	0x00	
[22]	0x44		[55]	0x00	
[23]	0x8A		[56]	0x00	
[24]	0x5B	SHA (Sender Hardware Address)	[57]	0x00	
[25]	0xED		[58]	0x00	
[26]	0xE9		[59]	0x00	
[27]	0x54		[60]	0x73	

Everything is clearly in accordance with the principles and aspects that we have analyzed. To the team *pingof* course, there will be no answer, since the ICMP protocol (network layer) is responsible for this, and we will implement this in the next article of the cycle. The correctness of network messages can be checked using WireShark. We start capturing packets and repeat the command *pingwe* end up observing:

```

Micro-St_ed:e9:54 Broadcast ARP 42 Who has 169.254.191.22? Tell 169.254.191.234
Hanazede_ed:a5:01 Micro-St_ed:e9:54 ARP 60 169.254.191.22 is at 00:17:22:ed:a5:01
169.254.191.234 169.254.191.22 ICMP 74 Echo (ping) request id=0x0001, seq=132/33792, ttl=64 (no response found!)

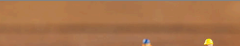
```

The PC sends a broadcast request with our IP address (line 1), in response we send it our MAC address (line 2). You can also get detailed information on each of the packages:



Well, as expected, ICMP is the answer to the command *ping* is missing. Everything worked out as planned, so we end today's article here, see you soon, in the next part!

- + Ethernet.h file



STM32 and Ethernet. Part 4.
Network...





MicroTechnics | STM32, Raspberry Pi...

4,562 followers



Follow on VK

The entrance

Registration

Search for...



Categories

Arduino

Qt

QML

Raspberry Pi

Algorithms

Sensors

Microcontrollers

AVR

MSP430

STM32

Neural networks

Miscellaneous



[Rebuses, tasks, charades](#)[Training courses](#)[STM32 from scratch](#)[STM32CubeMx](#)[Fundamentals of electronics](#)[Arduino lessons](#)[Electronics](#)

Community

[Arduino](#)[C++](#)[Sensors](#)[Microcontrollers](#)[STM32](#)[Developments](#)[Electronics](#)

Fresh entries

[NextPCB's DFM Tool.](#)[Where is the error in the diagram?](#)[STM32 and Ethernet. Part 5. Transport level. UDP protocol.](#)[How to add a library to Arduino IDE.](#)[How to accelerate Raspberry Pi 4 Model B.](#)