

ELECTRONICS



RASPBERRY PI



HOME



COMMUNITY

STM32 and Ethernet. Part 4. Network layer. IP and ICMP protocols.



Aveal MT

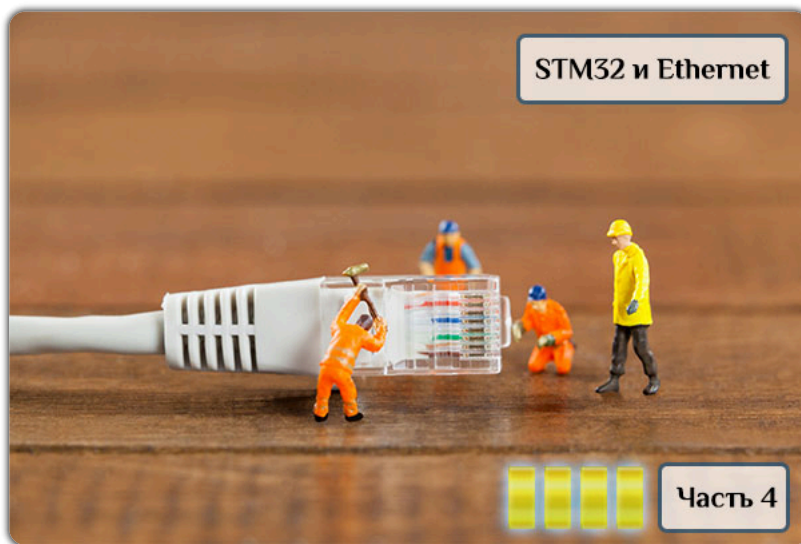
14/10/2021



2



STM32, STM32CubeMx



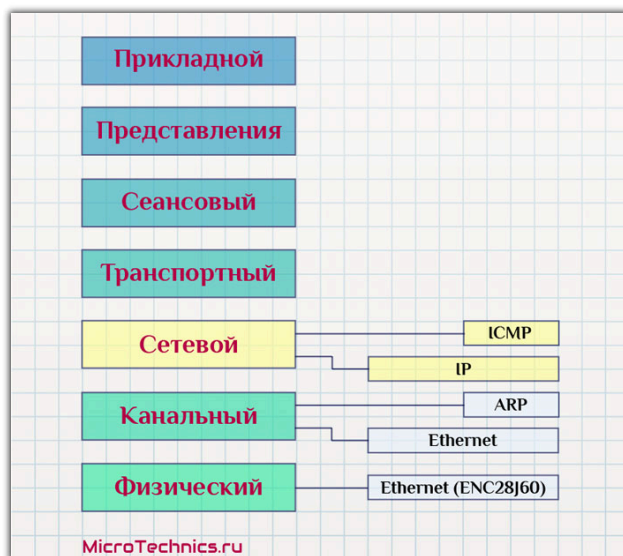
While memories of [previous project](#) we haven't sunk into oblivion, we'll start finishing what we started. In fact, we processed ARP requests received when sending a command *ping* our device. The logically correct step would be to implement a response to the above-mentioned command, which is what we are moving on to. We work according to the classical scheme - primary theoretical information, which will subsequently be accompanied by a practical example for **STM32** and **ENC28J60**.

A small contribution of additional systematization:

- [STM32 and Ethernet. Part 1. Connecting and configuring ENC28J60.](#)
- [STM32 and Ethernet. Part 2. ENC28J60. Reception and transfer of personnel.](#)
- [STM32 and Ethernet. Part 3. Channel level. ARP protocol.](#)

So, as we continue to move through the protocol stack to the upper layers, we are working at the network level today:





IP protocol.

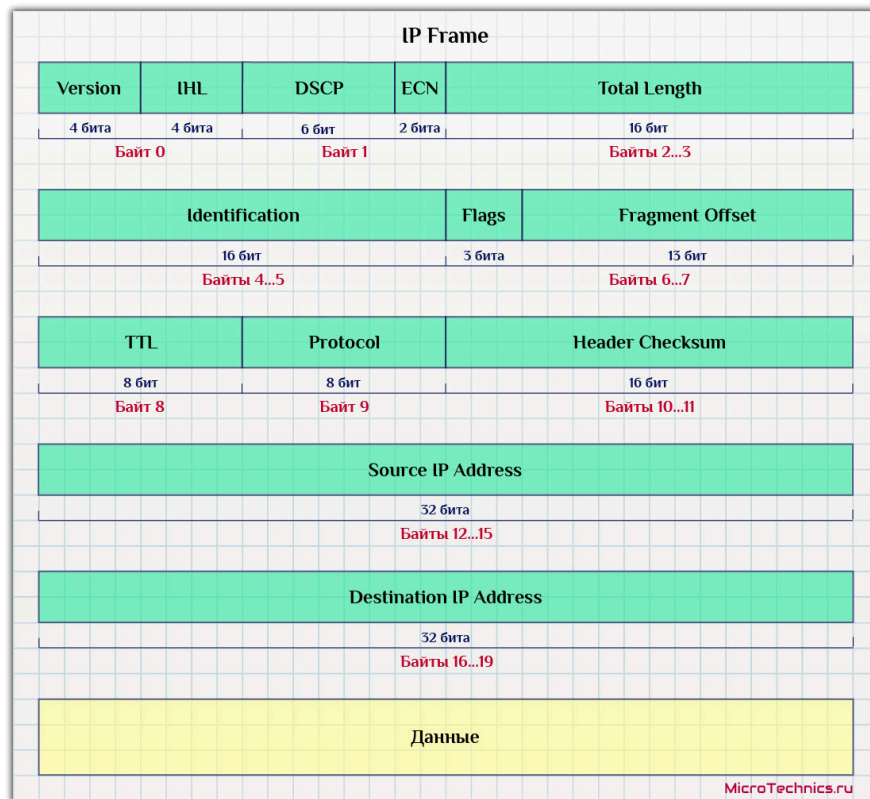
The IP protocol is key when it comes to the functioning of the Internet as a whole. It is this protocol that allows packets to circulate between networks, that is, it combines various network segments into one single Network. Whatever the number of intermediate routers between two network nodes, the protocol will deliver information to the address.

Meanwhile, the IP protocol itself does not guarantee reliable packet delivery, unlike, for example, the standard example of reliability - TCP.

An important property is the ability to fragment packets. If the size of the packet exceeds the maximum permissible for a particular node that is encountered along the path of the packet, then the packet is divided into several parts. Accordingly, each fragment contains its own identifier number for subsequent "assembly" of data together. Today we will not process fragmentation; we will leave it for the future (if necessary), since the size of our packages will be objectively small.

The IP packet format is based on the usual structure - header bytes followed by a semantic load in the form of data (the IPv4 version is considered):





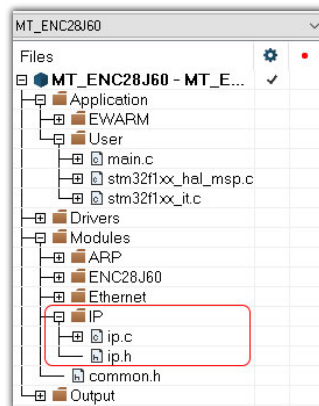
- **Version** The protocol version for IPv4 is 0x04.
- **IHL (Internet Header Length)** Is the length of the IP packet header in 32-bit words.
- **DSCP (Differentiated Services Code Point)** - this field was previously called *ToS (Type of Service)* - used to split packages into different service classes.
- **ECN (Explicit Congestion Notification)** - optional function - serves to inform nodes about network congestion.
- **Total Length** the full packet size in bytes. "Complete" implies that not only the data but also the header is included.
- **Identification** - id.
- **Flags** - three bits of flags:
 - Bit 0 (high) - does not carry a semantic load - is always zero.
 - Bit 1 - bit DF (Don't Fragment) - determines whether fragmentation as such is possible for a given packet.
 - Bit 2 (lowest) - bit MF (More Fragments) - for fragmented packets indicates whether the current part is the last.
- **Fragment Offset** - fragment offset - a 13-bit value that determines the position of the data of the current fragment relative to the original data packet. Moreover, the value is not in bytes, but in blocks of 8 bytes. For example, if the first fragment had 64 bytes of data (8 eight-byte blocks), then the next fragment will have this field as 8. The very first packet has an offset of 0.
- **TTL (Time To Live)** The packet lifetime is the maximum number of routers that a packet can pass along its path.
- **Protocol** - next layer network protocol id. If the IP data field of the packet encapsulates in itself **ICMP** packet, then the value of the field is 0x01, for **TCP** - 0x06, **UDP** - 0x11.
- **Header Checksum** - header checksum. Its calculation is feasible in the practical part just below.
- **Source IP Address** The sender's IP address.
- **Destination IP Address** The recipient's IP address.

The IP protocol package format has been disassembled and will be supplemented [current project](#) functionality for processing them.

We adhere to our highly structured architecture and start by adding two new files to the project:

- `ip.c`
- `ip.h`





The second step is to define a structure for working with IP frames. This action takes place in a file *ip.h*:

```

1. typedef struct IP_Frame
2. {
3.     uint8_t verHeaderLen;
4.     uint8_t diffServices;
5.     uint16_t len;
6.     uint16_t fragId;
7.     uint16_t fragOffset;
8.     uint8_t timeToLive;
9.     uint8_t protocol;
10.    uint16_t checksum;
11.    uint8_t srcIpAddr[IP_ADDRESS_BYTES_NUM];
12.    uint8_t destIpAddr[IP_ADDRESS_BYTES_NUM];
13.    uint8_t data data[];
14. } IP_Frame;

```

The fields are arranged in accordance with the order of their sequence discussed above. Let's immediately add a function to calculate the checksum:

```

1. /*-----*/
2. uint16_t IP_CalcChecksum(uint8_t* data, uint16_t len)
3. {
4.     uint32_t res = 0;
5.     uint16_t* ptr = (uint16_t*)data;
6.
7.     while(len > 1)
8.     {
9.         res += *ptr;
10.        ptr++;
11.        len -= 2;
12.    }
13.
14.    if(len > 0)
15.    {
16.        res += *(uint8_t*)ptr;
17.    }
18.
19.    while(res > 0xffff)
20.    {
21.        res = (res >> 16) + (res & 0xFFFF);
22.    }
23.
24.    return ~(uint16_t)res;
25. }
26.
27.
28.
29. /*-----*/

```

Basically, all the preparatory activity is complete, adding the main function for processing IP packets and generating responses:

```

1. /*-----*/
2. uint16_t IP_Process(IP_Frame* ipFrame, uint16_t frameLen)
3. {
4.     uint16_t newFrameLen = 0;
5.
6.     if(memcmp(ipFrame->destIpAddr, ipAddr, IP_ADDRESS_BYTES_NUM) == 0)
7.     {
8.         uint16_t rxChecksum = ipFrame->checksum;

```



```

9.     ipFrame->checksum = 0;
10.    uint16_t calcChecksum = IP_CalcChecksum((uint8_t*)ipFrame, sizeof(IP_Frame));
11.
12.    if(rxChecksum == calcChecksum)
13.    {
14.        uint16_t dataLen = frameLen - sizeof(IP_Frame);
15.        uint16_t newDataLen = 0;
16.
17.        if(ipFrame->protocol == IP_FRAME_PROTOCOL_ICMP)
18.        {
19.            newDataLen = ICMP_Process((ICMP_EchoFrame*)ipFrame->data, dataLen);
20.        }
21.
22.        ipFrame->len = htons(newDataLen + sizeof(IP_Frame));
23.
24.        ipFrame->fragId = 0;
25.        ipFrame->fragoffset = 0;
26.
27.        memcpy(ipFrame->destIpAddr, ipFrame->srcIpAddr, IP_ADDRESS_BYTES_NUM);
28.        memcpy(ipFrame->srcIpAddr, ipAddr, IP_ADDRESS_BYTES_NUM);
29.
30.        ipFrame->checksum = IP_CalcChecksum((uint8_t*)ipFrame, sizeof(IP_Frame));
31.
32.        newFrameLen = newDataLen + sizeof(ETH_Frame) + sizeof(IP_Frame);
33.    }
34. }
35.
36. return newFrameLen;
37. }
38.
39.
40.
41. /*-----*/

```

The general concept is the same as that used in the ARP protocol module. As an argument we get `frameLen` is the length of the frame IP. We will return the updated length corresponding to the sent packet. This value will be stored in `newFrameLen`.

We check that the specified IP in the packet matches our address:

```

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

```

If successful, we calculate the checksum. A small nuance is that before calculating, you need to reset the field with the accepted value, after saving it to a local variable for subsequent comparison:

```

1.  uint16_t rxChecksum = ipFrame->checksum;
2.  ipFrame->checksum = 0;
3.  uint16_t calcChecksum = IP_CalcChecksum((uint8_t*)ipFrame, sizeof(IP_Frame));
4.
5.  if(rxChecksum == calcChecksum)
6.  {
7.      // ...
8.  }

```

Again, in case of correctness, that is, coincidence of values, we continue to work with the received data. IN THE `frameLen` at the moment, the full length of the IP packet (including the header). Calculate the size of the data field (`dataLen`), which we will use in the future. Because we also have plans **ICMP protocol**, let's check the field `protocol` IP header:

```

1.  uint16_t dataLen = frameLen - sizeof(IP_Frame);
2.  uint16_t newDataLen = 0;
3.
4.  if(ipFrame->protocol == IP_FRAME_PROTOCOL_ICMP)
5.  {
6.      // ICMP processing should be there
7.  }

```

In this specially and prudently designated place we will subsequently place the processing of ICMP frames, but for now we will finish with IP... Assuming that `newDataLen` it will be changed subsequently in the ICMP part, we calculate the updated length of the full

2 IP packet and enter it in `ipFrame->len`:

```

1.  newFrameLen = newDataLen + sizeof(IP_Frame);
2.  ipFrame->len = htons(newFrameLen);

```

Packet fragmentation is not affected, so:

```

1.  ipFrame->fragId = 0;

```

```
2. ipFrame->fragoffset = 0;
```

We swap the IP addresses of the sender and recipient, or rather enter our address into the sender's IP - `ipAddr`. We finish the package by calculating the checksum:

```
1. memcpy(ipFrame->destIpAddr, ipFrame->srcIpAddr, IP_ADDRESS_BYTES_NUM);
2. memcpy(ipFrame->srcIpAddr, ipAddr, IP_ADDRESS_BYTES_NUM);
3.
4. ipFrame->checkSum = IP_CalcChecksum((uint8_t*)ipFrame, sizeof(IP_Frame));
```

That's all. Going down a level, to the file `ethernet.c`, into the function `ETH_Process()` and we use a similar principle to ARP to process IP packets:

```
1. //IP protocol
2. if(etherType == ETH_FRAME_TYPE_IP)
3. {
4.     responseSize = IP_Process((IP_Frame*)ethFrame->data, ethDataLen);
5. }
```

In total 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.        uint16_t ethDataLen = requestSize - sizeof(ETH_Frame);
12.
13.        //ARP protocol
14.        if(etherType == ETH_FRAME_TYPE_ARP)
15.        {
16.            responseSize = ARP_Process((ARP_Frame*)ethFrame->data, ethDataLen);
17.        }
18.
19.        //IP protocol
20.        if(etherType == ETH_FRAME_TYPE_IP)
21.        {
22.            responseSize = IP_Process((IP_Frame*)ethFrame->data, ethDataLen);
23.        }
24.
25.        if(responseSize > 0)
26.        {
27.            ETH_Response(ethFrame, responseSize);
28.        }
29.    }
30. }
31.
32.
33.
34. /*-----*/
```

We've sorted out the IP packets, and now we'll figure out the data they contain. And as already discussed, first of all, we will pay attention to ICMP.

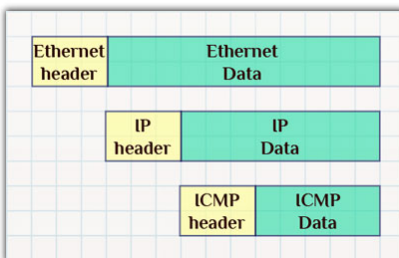
ICMP protocol.

ICMP is mainly used to signal the occurrence of exceptional situations - errors, unavailability of nodes, etc. In addition, what currently interests us first is **ICMP protocol** used to operate the utility *ping*. Actually, when trying to ping a node, an echo request is sent, and the presence or absence of an echo response determines the availability or unavailability of the target node 👍 Thus, we need to accept this very request and send the correct answer.

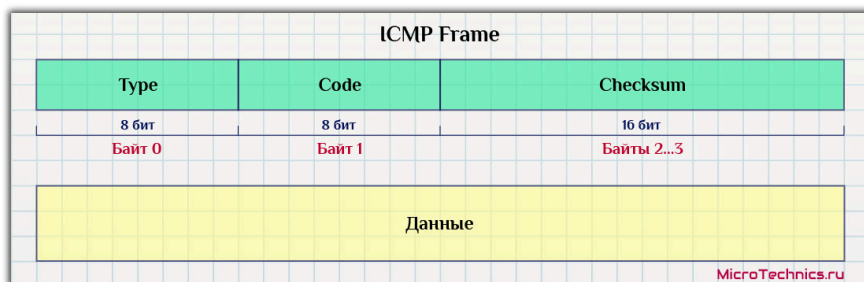
2

The essence of package formation remains the same as we discussed in [the previous part](#):





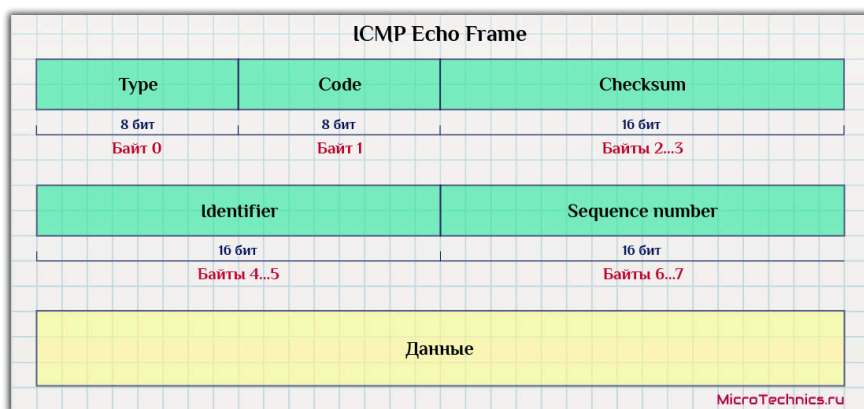
After processing the IP packet, we extract from it data that is already an ICMP packet, which also has its own header. The ICMP format of the package is as follows:



Again, we go through the header fields in the indicated order:

- *Type* - package type. At the moment, our area of interest is echo query - code 0x08, echo response - code 0x00.
- *Code* - in cooperation with the field *Type* gives complete information about the type of package. For both echo query and response, the code is 0x00. And, for example, if an ICMP packet signals that a node is unreachable, then: *Type* = 0x03, *Code* = 0x01. If the packet's meaning is that the port is unreachable, then this is the case: *Type* = 0x03, *Code* = 0x03. In general, this is the essence.
- *Checksum* - checksum. Calculated in the same way as for the IP package.

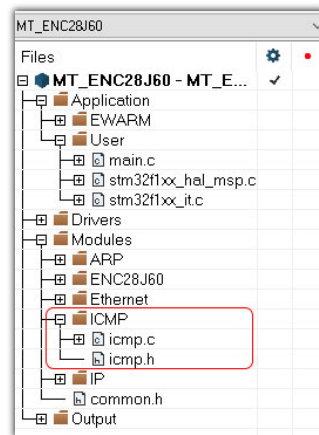
The header is followed by the data. Moreover, the data format may also differ for different field values *Type* and *Code*. Specifically, the ICMP format of echo requests and responses is



When generating a response to an echo request, we will only change the field *Type*. Again, we return to the project, add files, and define a structure for storing specialized data:

- *icmp.c*
- *icmp.h*





```

1. typedef struct ICMP_EchoFrame
2. {
3.     uint8_t type;
4.     uint8_t code;
5.     uint16_t checkSum;
6.     uint16_t id;
7.     uint16_t seqNum;
8.     uint8_t data data[];
9. } ICMP_EchoFrame;

```

Sticking to the established architecture, adding a function `ICMP_Process()` :

```

1. /*-----*/
2. uint16_t ICMP_Process(ICMP_EchoFrame* icmpFrame, uint16_t frameLen)
3. {
4.     uint16_t newFrameLen = 0;
5.
6.     uint16_t rxCheckSum = icmpFrame->checkSum;
7.     icmpFrame->checkSum = 0;
8.     uint16_t calcCheckSum = IP_CalcCheckSum((uint8_t*)icmpFrame, frameLen);
9.
10.    if(rxCheckSum == calcCheckSum)
11.    {
12.        if(icmpFrame->type == ICMP_FRAME_TYPE_ECHO_REQUEST)
13.        {
14.            icmpFrame->type = ICMP_FRAME_TYPE_ECHO_REPLY;
15.            icmpFrame->checkSum = IP_CalcCheckSum((uint8_t*)icmpFrame, frameLen);
16.            newFrameLen = frameLen;
17.        }
18.    }
19.
20.    return newFrameLen;
21. }
22.
23.
24.
25. /*-----*/

```

The algorithm is similar to the one discussed earlier for IP packets, so I won't even describe it separately (as a result, we have a processed packet ready for subsequent sending. All that remains is to call the currently added function in a pre-prepared place:

```

1. /*-----*/
2. uint16_t IP_Process(IP_Frame* ipFrame, uint16_t frameLen)
3. {
4.     uint16_t newFrameLen = 0;
5.
6.     if(memcmp(ipFrame->destIpAddr, ipAddr, IP_ADDRESS_BYTES_NUM) == 0)
7.     {
8.         uint16_t rxCheckSum = ipFrame->checkSum;
9.         ipFrame->checkSum = 0;
10.        uint16_t calcCheckSum = IP_CalcCheckSum((uint8_t*)ipFrame, sizeof(IP_Frame));
11.
12.        if(rxCheckSum == calcCheckSum)
13.        {
14.            uint16_t dataLen = frameLen - sizeof(IP_Frame);
15.            uint16_t newDataLen = 0;
16.
17.            if(ipFrame->protocol == IP_FRAME_PROTOCOL_ICMP)
18.            {

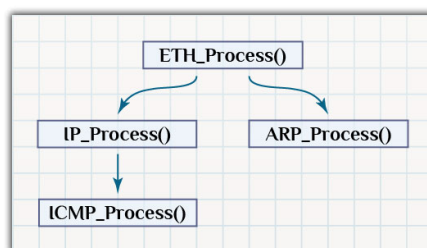
```

```

19.     newDataLen = ICMP_Process((ICMP_EchoFrame*)ipFrame->data, dataLen);
20. }
21.
22. newFrameLen = newDataLen + sizeof(IP_Frame);
23. ipFrame->len = htons(newFrameLen);
24.
25. ipFrame->fragId = 0;
26. ipFrame->fragoffset = 0;
27.
28. memcpy(ipFrame->destIpAddr, ipFrame->srcIpAddr, IP_ADDRESS_BYTES_NUM);
29. memcpy(ipFrame->srcIpAddr, ipAddr, IP_ADDRESS_BYTES_NUM);
30.
31. ipFrame->checksum = IP_CalcChecksum((uint8_t*)ipFrame, sizeof(IP_Frame));
32. }
33. }
34.
35. return newFrameLen;
36. }
37.
38.
39.
40. /*-----*/

```

Systematizing what is graphically implemented, we get:



We assemble the project, flash and analyze the result obtained, sending a command `ping` on our ip:

```
1. ping 169.254.191.22
```

```

Администратор: Windows PowerShell
PS C:\WINDOWS\system32> ping 169.254.191.22

Обмен пакетами с 169.254.191.22 по 32 байтами данных:
Ответ от 169.254.191.22: число байт=32 время=1мс TTL=64
Ответ от 169.254.191.22: число байт=32 время=1мс TTL=64
Ответ от 169.254.191.22: число байт=32 время=1мс TTL=64
Ответ от 169.254.191.22: число байт=32 время=1мс TTL=64

Статистика Ping для 169.254.191.22:
    Пакетов: отправлено = 4, получено = 4, потеряно = 0
    (0% потеря)
    Приблизительное время приема-передачи в мс:
        Минимальное = 1мсек, Максимальное = 2 мсек, Среднее = 1 мсек
PS C:\WINDOWS\system32>

```

Everything went according to plan, we observed successful responses from the node. Similarly, see WireShark:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	169.254.191.234	169.254.191.22	ICMP	74	Echo (ping) request id=0x0001, seq=339/21249, ttl=64 (reply in 2)
2	0.001400	169.254.191.22	169.254.191.234	ICMP	74	Echo (ping) reply id=0x0001, seq=339/21249, ttl=64 (request in 1)
4	1.006460	169.254.191.234	169.254.191.22	ICMP	74	Echo (ping) request id=0x0001, seq=340/21505, ttl=64 (reply in 5)
5	1.007879	169.254.191.22	169.254.191.234	ICMP	74	Echo (ping) reply id=0x0001, seq=340/21505, ttl=64 (request in 4)
6	2.027322	169.254.191.234	169.254.191.22	ICMP	74	Echo (ping) request id=0x0001, seq=341/21761, ttl=64 (reply in 7)
7	2.028732	169.254.191.22	169.254.191.234	ICMP	74	Echo (ping) reply id=0x0001, seq=341/21761, ttl=64 (request in 6)
8	3.041951	169.254.191.234	169.254.191.22	ICMP	74	Echo (ping) request id=0x0001, seq=342/22017, ttl=64 (reply in 9)
9	3.043249	169.254.191.22	169.254.191.234	ICMP	74	Echo (ping) reply id=0x0001, seq=342/22017, ttl=64 (request in 8)

The icing on the cake - let's look at the bytes received. Let's set a break point, for example, here:



```

25 uint16_t ICMP_Process(ICMP_EchoFrame* icmpFrame, uint16_t frameLen)
26 {
27     uint16_t newFrameLen = 0;
28
29     uint16_t rxChecksum = icmpFrame->checksum;
30     icmpFrame->checksum = 0;
31     uint16_t calcChecksum = IP_CalcChecksum((uint8_t*)icmpFrame, frameLen);
32
33     if (rxChecksum == calcChecksum)
34     {
35         if (icmpFrame->type == ICMP_FRAME_TYPE_ECHO_REQUEST)
36         {
37             icmpFrame->type = ICMP_FRAME_TYPE_ECHO_REPLY;
38             icmpFrame->checksum = IP_CalcChecksum((uint8_t*)icmpFrame, frameLen);
39             newFrameLen = frameLen;
40         }
41     }
42
43     return newFrameLen;
44 }

```

And then we analyze the available bytes:

Expression	Value		Expression	Value	
frame	<struct>		[28]	0xBF	
nextPtr	0x0654		[29]	0xEA	IP-адрес отправителя
length	78		[30]	0xA9	169.254.191.22
status	0x00C0		[31]	0xFE	IP-адрес получателя
data	<array>""		[32]	0xBF	
[0]	0x00		[33]	0x16	Тип (эхо-запрос) Code
[1]	0x17	00:17:22:ED:A5:01	[34]	0x08	Checksum
[2]	0x22	MAC-адрес получателя	[35]	0x00	Identifier
[3]	0xED		[36]	0x00	
[4]	0xA5	44:8A:5B:ED:E9:54	[37]	0x00	Sequence number
[5]	0x01	MAC-адрес отправителя	[38]	0x00	
[6]	0x44		[39]	0x01	
[7]	0x8A		[40]	0x01	
[8]	0x5B		[41]	0x57	
[9]	0xED		[42]	0x61	Данные
[10]	0xE9		[43]	0x62	
[11]	0x54		[44]	0x63	
[12]	0x08	EtherType	[45]	0x64	
[13]	0x00		[46]	0x65	
[14]	0x45	Version + IHL	[47]	0x66	
[15]	0x00	DSCP + ECN	[48]	0x67	
[16]	0x00	Total Length	[49]	0x68	
[17]	0x3C		[50]	0x69	
[18]	0x49	Identification	[51]	0x6A	
[19]	0x57	Flags + Fragment Offset	[52]	0x6B	
[20]	0x00	TTL	[53]	0x6C	
[21]	0x00	Protocol (ICMP)	[54]	0x6D	
[22]	0x40	Checksum	[55]	0x6E	
[23]	0x01		[56]	0x6F	
[24]	0x00		[57]	0x70	
[25]	0x00		[58]	0x71	
[26]	0xA9	169.254.191.234	[59]	0x72	
[27]	0xFE		[60]	0x73	

The checksum fields are zero because we reset them to zero in the code. Actually, that's all, we'll end here for today 🙌

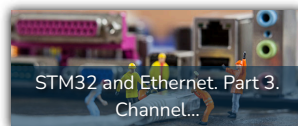
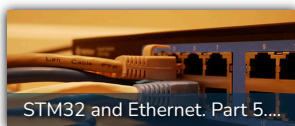
+ ip.c file

+ ip.h file

+ The icmp.c file

+ File icmp.h

Project reference - [MT_ENC28J60_Part_4](#)



2



- < STM32 and Ethernet. Part 3. Channel level. ARP protocol.
- > Display based on SSD1306 controller. Library for STM32.



B I U

⚡ 🔥 Old ones ▼



⌚ 4 years ago

↩ Reply



⌚ 3 years ago

↩ Reply

MicroTechnics | STM32, Raspberry Pi.

4,562 followers

Follow on VK

2



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.

© 2013 - 2026 MicroTechnics.ru | Privacy Policy | Site rules | Images and materials on the site are protected by copyright

