

STM32 and Ethernet. Part 2. ENC28J60. Reception and transfer of personnel.

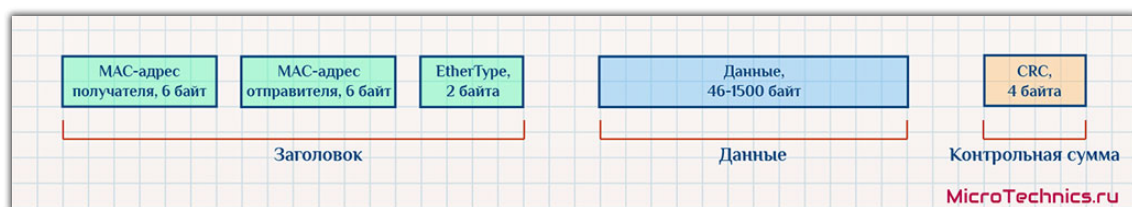


Aveal MT | 09/01/2021 | 2 | STM32, STM32CubeMx



Greetings to everyone, we continue with **STM32 and Ethernet**, this is going to be part 2 (part 1 is available by [link](#)). And the task will be simple – you need to finish off the driver **ENC28J60**. We did most of it, but the most important process – sending and receiving personnel was not covered. This is our plan for today 👍

Ethernet frame format we already [they sorted it outlet's](#) refresh the list of fields in memory:

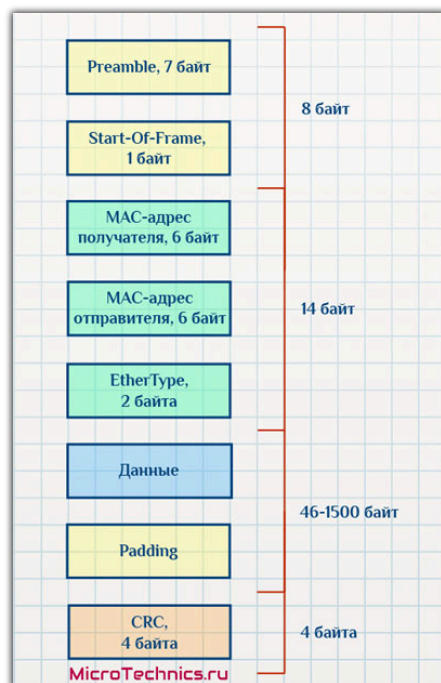


- MAC address of the recipient
- MAC address of the sender
- EtherType
- Data
- CRC

2



In addition, for the IEEE 802.3 standard, fields are added to it when a packet is transferred to a line *Preamble* (7 bytes) and *Start-Frame* (1 byte):



And in addition, we are observing a field here *Padding*, which is attached to the transmitted data. It has a simple functional purpose. According to the standard, the frame length should not be less than 64 bytes (including 4 bytes of checksum), which consist of:

- 6 bytes - MAC address of the recipient
- 6 bytes - Sender's MAC address
- 2 bytes - EtherType
- Data
- 4 bytes - CRC

That is why our diagram shows that the minimum data size is 46 bytes ($64 - (6 + 6 + 2 + 4)$). And it is because of this that an additional field is required *Padding*, which is a set of zero bytes. If the total frame length for a given data set is less than 64 bytes, then it is added *Padding* with enough bytes to get the total of 64 bytes required by the standard.

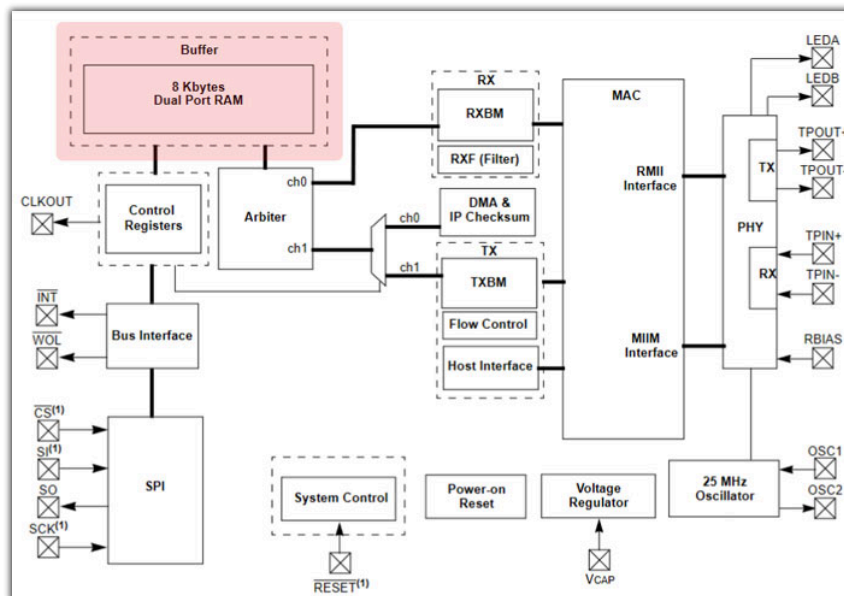
Among its capabilities, the ENC28J60 controller also has hardware support for the mentioned parts of the package:

- Preamble
- Start-of-Frame
- Padding
- CRC

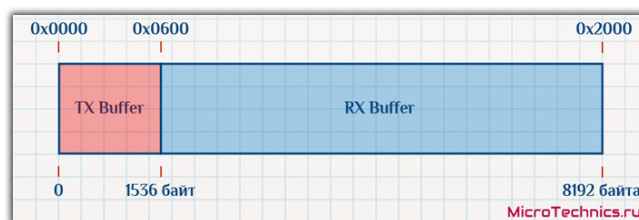
That is, when sending **ENC28J60** it can add these fields to the frame, and upon reception, accordingly, "withdraw". Although *Padding* and the checksum will nevertheless be written to the receiving buffer, so that they are available for analysis if necessary.

Both receiving and sending frames to ENC28J60 are carried out using ring buffers located in RAM memory. In total, 8 KB of memory is available for these purposes:





The distribution of memory between the receiving and transmitting buffers is entrusted to us. In the draft from the previous article, I did the following as an example:



Set accordingly in defines in the file `enc28j60.h`:

```
1. #define ENC28J60_TX_BUF_START 0x0000
2. #define ENC28J60_RX_BUF_START 0x0600
3. #define ENC28J60_RX_BUF_END 0x1FFF
```

To control these processes, ENC28J60 provides 4 pairs of registers:

- *ETXSTL/ETXSTH* – start address of the transfer buffer
- *ETXNDL/ETXNDH* – address of the end of the buffer for transmission
- *ERXSTL/ERXSTH* – address of the start of the buffer to receive
- *ERXNDL/ERXNDH* – address of the end of the buffer to receive

The last two of them are used to set the area for the Rx buffer, and everything else is allocated to the Tx buffer by default. In registers *ETXST* and *ETXND* during transmission, we will record the addresses of the initial and final bytes of a specific one **ethernet frames**.which we are going to transmit. We will clearly see it in the implementation process.

However, ENC28J60 does not check the addresses specified in *ETXST* and *ETXND*.The control controller should take care of this if the RX buffer addresses overlap.

Summarizing the above, in the initialization function we set the boundary values for the receiving buffer:

```
1. //Rx/Tx buffers
2. WriteControlRegPair(ERXSTL, ENC28J60_RX_BUF_START);
3. WriteControlRegPair(ERXNDL, ENC28J60_RX_BUF_END);
```

2

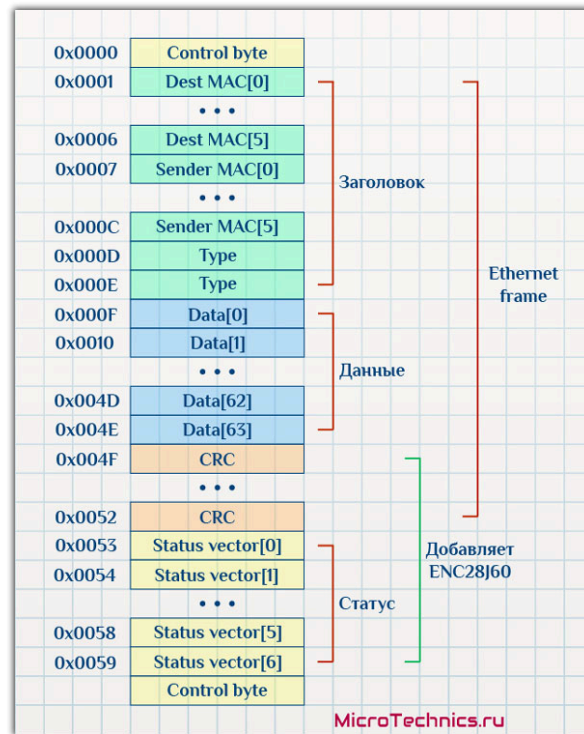
Ethernet frame transmission.

The buffering has been sorted out, we move on, immersing ourselves in the sending mechanism to fully understand the theoretical aspects considered. So, to send a frame, you need to:

- Check that the previous transmission process is completed using a bit *TXRTS* register *ECON1*.

- Write to register *ETXST* free memory address where we will subsequently place the data of the transmitted frame. This register, like most others, is a complementary pair of registers *ETXSTL/ETXSTH*, but I will omit this point in the text and write simply *ETXST* for harmony and brevity of the story.
- With the help of the team **Write Buffer Memory** we put the data in the buffer. In the code we have a function for this `static void WriteBufferMem(uint8_t *data, uint16_t size)`.
- According to the size of the transmitted data, we write it to the register *ETXND* final address.
- We start the process by setting the bit *TXRTS* register *ECON1*.

Let's move on to a more specific example. Let's say 64 bytes of data need to be transmitted, then placing the information in the buffer will look like this:



Our starting address is set to `ENC28J60_TX_BUF_START(0x0000)` .. and we use it. Next, we write an additional control byte. I'll tell you more about it:



As you can see, there are 4 significant bits that enable or disable this or that functionality. As usual, a detailed description of the bits is in [datasheet](#), I won't upload the article with a banal translation. Only in essence and nothing superfluous, namely, let's pay attention to the bit *POVERRIDE*. If set to "1", ENC28J60 will be guided by the values contained in the remaining three bits of the check byte when transmitting (*PHUGREEN*, *PPADEN*, *PCRCEN*).

If in *POVERRIDE* write "0", then the settings saved in the register will be used *MACON3*. That is, similar settings are present in two places, and bit *POVERRIDE* decides which ones to use.

We will take the second path – in *POVERRIDE* let's write "0". Accordingly, the remaining bits of this register no longer play a role in this situation; we will reset them too. A in register *MACON3* we have already saved the necessary settings that we set when initializing the module in `enc28j60.c`:

2. 1. `WriteControlReg(MACON3, MACON3_PADCFG0_BIT | MACON3_TXCRCEN_BIT | MACON3_FRMLNEN_BIT);`



In particular, for example, bit *PADCFG0* allows ENC28J60 to add the required zero bytes if the frame size is too small, and *TXCRCEN* – adds a checksum to the transmitted packet.

That's all about the check byte. Summary – it will be 0x00 and settings will be taken from *MACON3*.

Next, we put the sent data into memory **ethernet frames, including both the frame header and 64 bytes of data**. This is completed by a set of status bytes, which ENC28J60 also adds on its own.

We finish this part by creating a function in the file *enc28j60.c*. In the end, all we had to do was implement what we discussed in the code:

```

1.  /*-----*/
2.  void ENC28J60_TransmitFrame(uint8_t *data, uint16_t size)
3.  {
4.      while((ReadControlReg(ECON1) & ECON1_TXRTS_BIT) != 0)
5.      {
6.          if((ReadControlReg(EIR) & EIR_TXERIF_BIT) != 0)
7.          {
8.              BitFieldSet(ECON1, ECON1_TXRST_BIT);
9.              BitFieldClear(ECON1, ECON1_TXRST_BIT);
10.         }
11.     }
12.
13.     WriteControlRegPair(EWRPTL, ENC28J60_TX_BUF_START);
14.
15.     uint8_t controlByte = 0x00;
16.     WriteBufferMem(&controlByte, 1);
17.     WriteBufferMem(data, size);
18.
19.     WriteControlRegPair(ETXSTL, ENC28J60_TX_BUF_START);
20.     WriteControlRegPair(ETXNDL, ENC28J60_TX_BUF_START + size);
21.
22.     BitFieldSet(ECON1, ECON1_TXRTS_BIT);
23. }
24.
25.
26.
27. /*-----*/

```

There is only one nuance here. Bit *TXRTS* may not reset in hardware when a transmission error occurs, so it is recommended to check the bit *TXERIF* register *EIR*. If the bit is in "1", it signals an error occurred in the previous transmission process. In this case, we install and reset the bit sequentially *TXRST*. Then we proceed to fill the buffer with data.

Ethernet frame reception.

Next up is a technique that will allow us to complete the driver for **ENC28J60** and move on to practical processing of real personnel. We have already initialized the receiving buffer:

```

1. WriteControlRegPair(ERXSTL, ENC28J60_RX_BUF_START);
2. WriteControlRegPair(ERXNDL, ENC28J60_RX_BUF_END);

```

ENC28J60 also includes hardware filtering of received messages. Settings for this functionality – in register *ERXFCON*:

R/W-1	R/W-0	R/W-1	R/W-0	R/W-0	R/W-0	R/W-0	R/W-1
UCEN	ANDOR	CRCEN	PMEN	MPEN	HTEN	MCEN	BCEN
bit 7							
							bit 0

IN THE [datasheet](#) there is not only a description, but also a graphical diagram describing the operation of the filter mechanism, so I won't copy it. In our initialization, we did not change the value of this register, leaving it in default – 0xA1 (0b10100001). But if necessary, you can select specific optimal settings for a specific task, all through this register.

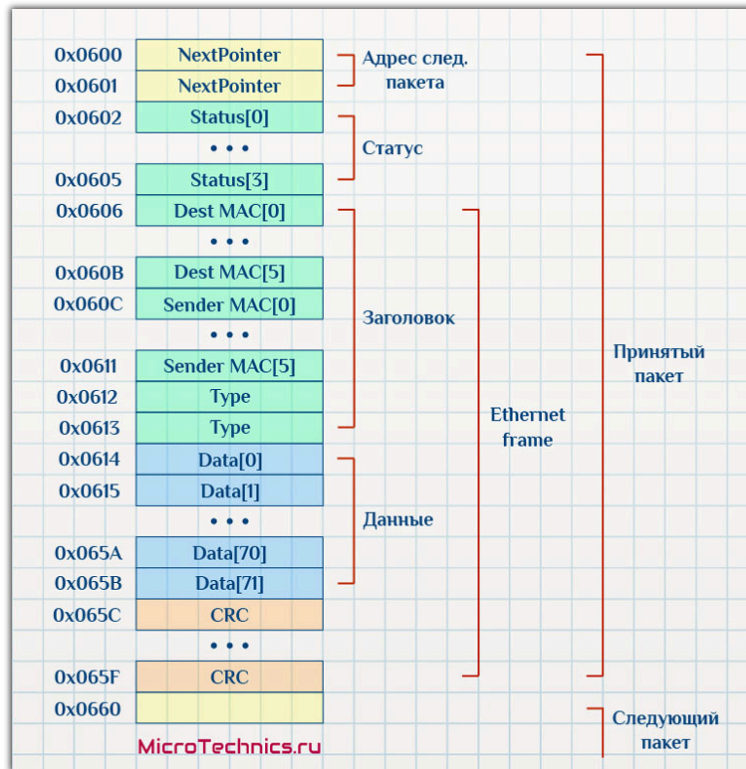
So, we have configured the addresses and filtering. Since I do not use ENC28J60 interrupts to track changes in its state, all I need to do to start receiving is set the bit *RXEN* the same register *ECON1*. What we do in our function:

```

1.  /*-----*/
2.  void ENC28J60_StartReceiving()
3.  {
4.      BitFieldSet(ECON1, ECON1_RXEN_BIT);
5.  }
6.
7.
8.
9.  /*-----*/

```

After execution, all received frames (those that passed the hardware filtering) will be sent to the ring buffer in the corresponding memory area:



Besides the data bytes themselves `Data[]` and the header will contain additional service information:

- Pointer to next packet (2 bytes)
- Status bits (4 bytes in total)
- The last 4 bytes contain the checksum value
- The packet address in ENC28J60 must always be even, so a 1-byte offset (an extra byte after CRC) can be added to guarantee parity

The status bits are responsible for the following:

Bit	Field	Description
31	Zero	0
30	Receive VLAN Type Detected	Current frame was recognized as a VLAN tagged frame.
29	Receive Unknown Opcode	Current frame was recognized as a control frame but it contained an unknown opcode.
28	Receive Pause Control Frame	Current frame was recognized as a control frame containing a valid pause frame opcode and a valid destination address.
27	Receive Control Frame	Current frame was recognized as a control frame for having a valid type/length designating it as a control frame.
26	Dribble Nibble	Indicates that after the end of this packet, an additional 1 to 7 bits were received. The extra bits were thrown away.
25	Receive Broadcast Packet	Indicates packet received had a valid broadcast address.
24	Receive Multicast Packet	Indicates packet received had a valid multicast address.
23	Received Ok	Indicates that at the packet had a valid CRC and no symbol errors.
22	Length Out of Range	Indicates that frame type/length field was larger than 1500 bytes (type field).
21	Length Check Error	Indicates that frame length field value in the packet does not match the actual data byte length and specifies a valid length.
20	CRC Error	Indicates that frame CRC field value does not match the CRC calculated by the MAC.
19	Reserved	
18	Carrier Event Previously Seen	Indicates that at some time since the last receive, a carrier event was detected. The carrier event is not associated with this packet. A carrier event is activity on the receive channel that does not result in a packet receive attempt being made.
17	Reserved	
16	Long Event/Drop Event	Indicates a packet over 50,000 bit times occurred or that a packet was dropped since the last receive.
15-0	Received Byte Count	Indicates length of the received frame. This includes the destination address, source address, type/length, data, padding and CRC fields. This field is stored in little-endian format.

We get that the 2 least significant status bytes are nothing more than the length of the received packet. And, looking ahead, we will add the structure to the file `enc28j60.h`. Here, the order of the fields corresponds exactly to the sequence of received data:

```
1. typedef struct ENC28J60_Frame
2. {
3.     uint16_t nextPtr;
4.     uint16_t length;
5.     uint16_t status;
6.     uint8_t data[ENC28J60_FRAME_DATA_MAX];
7.     uint32_t checksum;
8. } ENC28J60_Frame;
```

To organize the operation of the ring buffer, two pointers are used - `ERXRDPT` and `ERXWRPT`. Upon reaching the end of the buffer, ENC28J60 automatically moves to the beginning. `ERXWRPT` stores the address where the receiver will start saving the next frame when it arrives. A `ERXRDPT` on the contrary, it indicates the memory location from which we will retrieve the received data.

To register `ERDPT` we will enter the address from which we want to read the data. If bit `AUTOINC` in the `ECON2` set (and it is set by default), the value will be incremented automatically when reading the information.

In the code to read data from the buffer, we already have a function `void ReadBufferMem(uint8_t *data, uint16_t size)`. Now let's implement it **receiving Ethernet frames** in accordance with the aspects we have considered. First, the full code, then we go through it in detail:

```
1. /*-----*/
2. uint16_t ENC28J60_ReceiveFrame(ENC28J60_Frame* frame)
3. {
4.     uint16_t dataSize = 0;
5.     uint8_t packagesNum = ReadControlReg(EPKTCNT);
6.
7.     if(packagesNum > 0)
8.     {
9.         WriteControlRegPair(ERDPTL, curPtr);
10.
11.         ReadBufferMem((uint8_t*) frame, ENC28J60_HEADER_SIZE);
12.
13.         curPtr = frame->nextPtr;
14.
15.         if((frame->status & ENC28J60_FRAME_RX_OK_MASK) != 0)
16.         {
17.             dataSize = frame->length - ENC28J60_CRC_SIZE;
18.
19.             if(dataSize > ENC28J60_FRAME_DATA_MAX)
20.             {
21.                 dataSize = ENC28J60_FRAME_DATA_MAX;
22.             }
23.
24.             ReadBufferMem((uint8_t*) &(frame->data[0]), dataSize);
25.             ReadBufferMem((uint8_t*) &(frame->checksum), ENC28J60_CRC_SIZE);
26.         }
27.
28.         uint16_t nextPtr = frame->nextPtr - 1;
29.         if(nextPtr > ENC28J60_RX_BUF_END)
30.         {
31.             nextPtr = ENC28J60_RX_BUF_END;
32.         }
33.
34.         WriteControlRegPair(ERXRDPTL, nextPtr);
35.         BitFieldSet(ECON2, ECON2_PKTDEC_BIT);
36.     }
37.
38.     return dataSize;
39. }
40.
41.
42.
43. /*-----*/
```

We start by declaring a variable to store the size of the received packet, which the function will subsequently return, and we also get it from the register `EPKTCNT` the number of packets received so far is

```
1. uint16_t dataSize = 0;
2. uint8_t packagesNum = ReadControlReg(EPKTCNT);
```

Since ENC28J60 has another known address parity/oddness bug, we will additionally store the current pointer value in a special variable:

```
1. static uint16_t curPtr = ENC28J60_RX_BUF_START;
```

Accordingly, the address initially corresponds to the beginning of the receiving buffer. We write this saved address to the register *ERDPTL*:

```
1. WriteControlRegPair(ERDPTL, curPtr);
```

At this stage, everything is ready to read data from the buffer. As we found out a little earlier, in addition to information data, service information will be stored in the buffer. Therefore, we will read into an already added structure of the type *ENC28J60_Frame* ., a pointer to which we take as an argument in the receive function *ENC28J60_ReceiveFrame(ENC28J60_Frame* frame)* . We read 6 bytes (*ENC28J60_HEADER_SIZE*):

```
1. ReadBufferMem((uint8_t*)frame, ENC28J60_HEADER_SIZE);
```

They fall accordingly in *frame->nextPtr* , *frame->length* and *frame->status* . Let's check with the status bits that everything is fine with the received packet:

```
1. if((frame->status & ENC28J60_FRAME_RX_OK_MASK) != 0)
```

If the condition is met, then we continue processing. We get the frame size and limit it if the maximum size we set is exceeded *ENC28J60_FRAME_DATA_MAX* . That's it, then read the long-awaited data and 4 bytes of checksum (*ENC28J60_CRC_SIZE*):

```
1. ReadBufferMem((uint8_t*)The &(frame->data data[0]), dataSize);
2. ReadBufferMem((uint8_t*)The &(frame->checkSum), ENC28J60_CRC_SIZE);
```

The data has been received, all that remains is to service the ring buffer mechanism for further uninterrupted operation.

And then comes the mentioned nuance-bug ENC28J60. For an even value in *ERXRDP* data corruption may occur, so we write a value reduced by 1 to this register (it will be odd, since in *frame->nextPtr* an even value is guaranteed, since the packet start address is always even in the controller):

```
1. uint16_t nextPtr = frame->nextPtr - 1;
2. if(nextPtr > ENC28J60_RX_BUF_END)
3. {
4.     nextPtr = ENC28J60_RX_BUF_END;
5. }
6.
7. WriteControlRegPair(ERXRDP, nextPtr);
```

At the same time, we check that the pointer has not gone beyond the buffer. A in *curPtr* we already have (*curPtr = frame->nextPtr*) we have placed the correct address of the next packet, which we will use for further data reception.

Final step – let ENC28J60 know that we have received the package. To do this, write "1" to the bit *PKTDEC* register *ECON2*:

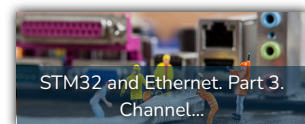
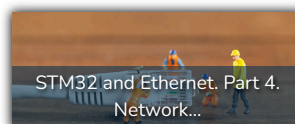
```
1. BitFieldSet(ECON2, ECON2_PKTDEC_BIT);
```

It's hard to believe, but that's where the reception and transmission of packets ends. In the next article, let's go further with **Ethernet**, see you later 🍷

+ File enc28j60.c

+ File enc28j60.h

Project link - [MT_ENC28J60_Part_2](#)



< Connecting the ST7735-based display to the STM32 microcontroller.

> Display based on ST7735 and STM32. Image output.

 Subscribe



Присоединиться к обсуждению

B *I* U           

2 COMMENTS

  Old ones



Igor

4 years ago

Tell me, what transmission speed did you manage to achieve?

Reply



Aveal MT

Administrator

Author

Reply to comment Igor

4 years ago

Yes, I only checked for functionality; I don't have any current tasks at all where Ethernet is involved. So what happens next is up to readers (If there is a desire to protest and share the results in the community, then there will be fire, it would be interesting.

Reply

 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.

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

