

FORUM

STM32

ELECTRONICS

RASPBERRY PI

STM32 and Ethernet. Part 1. Connecting and configuring ENC28J60.

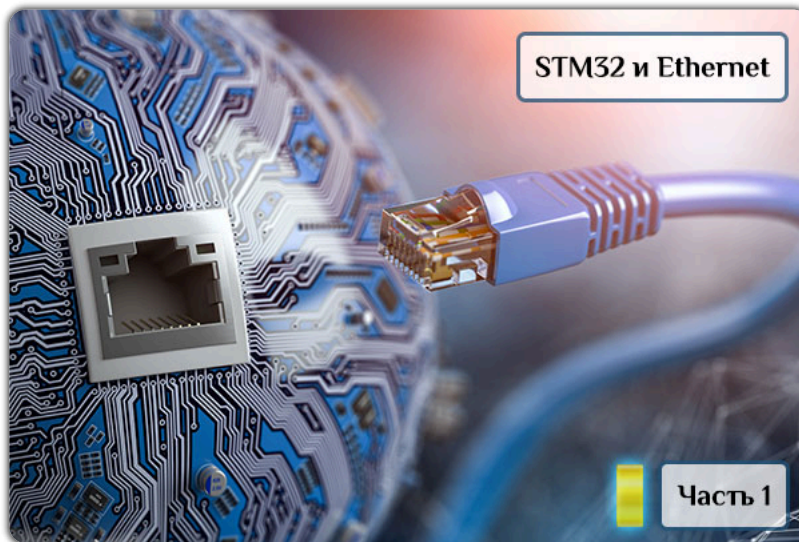


Aveal MT

17/08/2021

23

STM32, STM32CubeMx

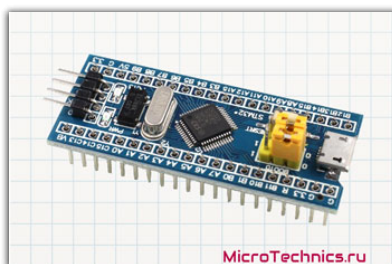


Good day everyone! We begin the long-awaited series of articles on practical work with [Ethernet](#). Let's go from the lowest level across the entire protocol stack - ARP, IP, UDP, TCP... We'll take it traditionally as a target platform **STM32** and as a physical layer chip, the Ethernet controller **ENC28J60**, as one of the most common and accessible.

In this first article, we will look at the physical connection, initialization of interfaces, ENC28J60 configuration, and, perhaps, we will write a driver for working with the above-mentioned controller. I will describe everything in as much detail as possible, but if questions suddenly arise, welcome to the comments, or even better, contact us at [forum](#).

Connecting ENC28J60.

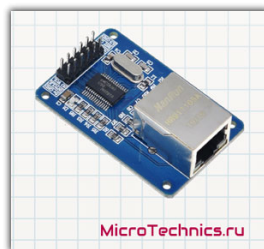
So, STM32... I will take the STM32F103C8T6, again due to its incredible prevalence and popularity, which we can be grateful for the Blue Pill board:



23



For ENC28J60, I have one module, among others:



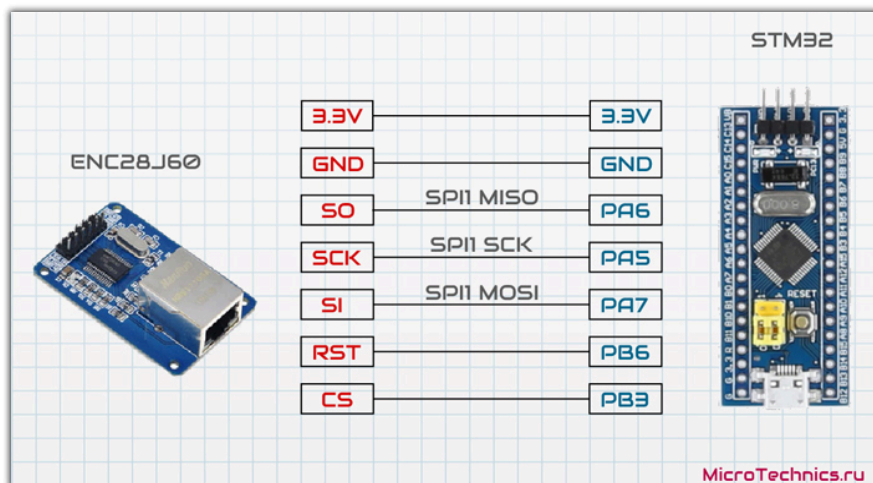
We will write code based on the possibility of using it with any other hardware, so the choice in this case is not critical.

Let's recall the OSI network model (I'll be linking to an OSI article here shortly):



The lowest level, physical, in this case it is **10BASE-T** fully takes over the ENC28J60. We move to a level higher - the link layer, which is responsible for transmitting frames between network nodes, is based on the use of MAC addresses. It will also be partly taken over by the ENC28J60, fortunately the possibilities allow. But we will implement work at the levels above programmatically - for example, at the network level, there will be an IP protocol, at the transport level - UDP/TCP, but we will not get ahead of ourselves. Let's return to connecting the modules.

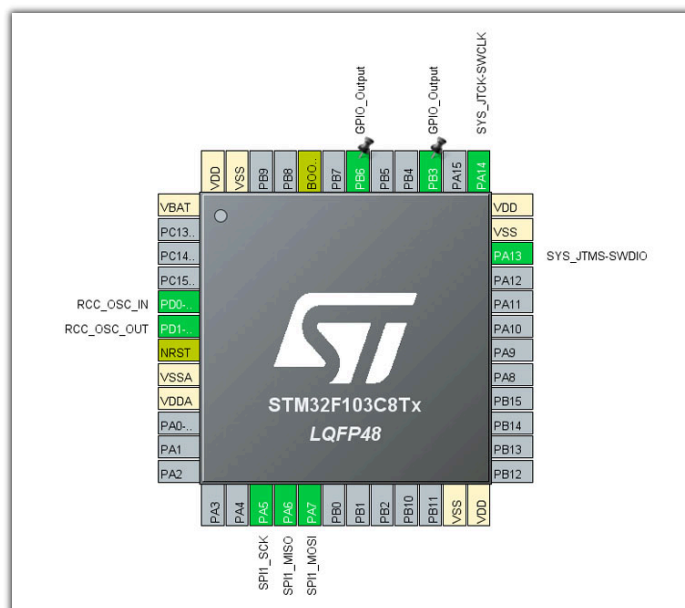
And we will connect the controller via the interface **SPI**. In fact, the following is a diagrammatic representation:



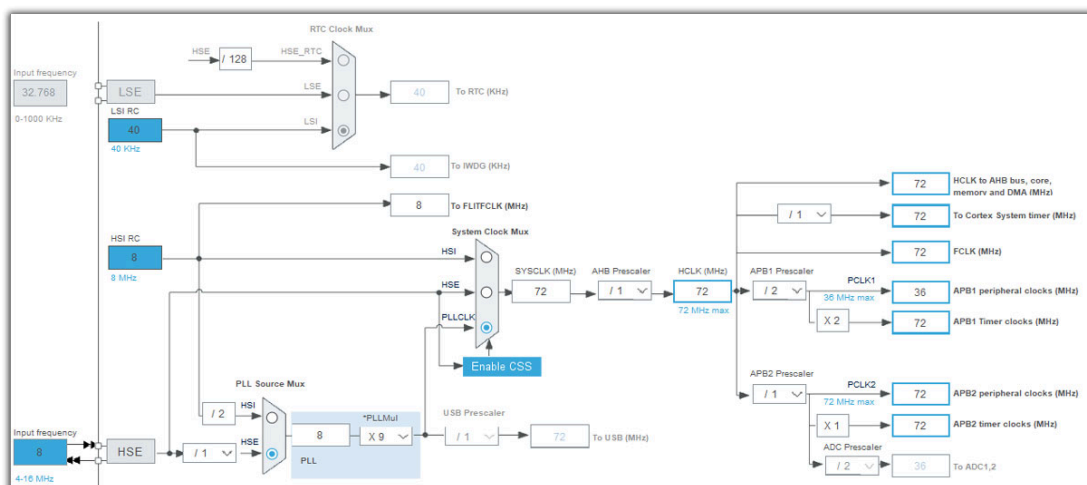
23



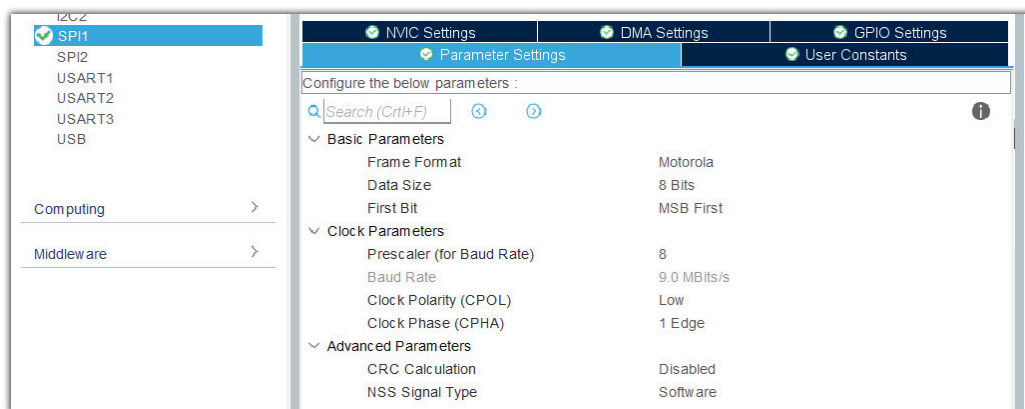
It happens differently in different projects, but in this case I don't see the slightest reason not to use STM32CubeMx and HAL, so create a new project and immediately turn on the "basic" things, such as external clocking and SWD interface. In addition, a couple of pins for ENC28J60, in this case, are Chip Select, Reset and, of course, SPI1 signals:



I have an external 8 MHz quartz resonator, so in the clock window:



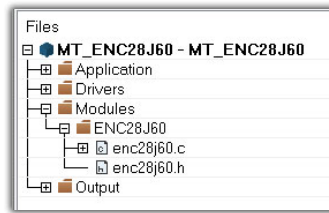
Finally, of course, we set up the SPI. As shown in the diagram above, we use legs PA5 - SPI SCK, PA6 - SPI MISO and PA7 - SPI MOSI. The settings for the ENC28J60 are as follows



After this, we can move on to generating a project; we won't change anything else in CubeMx.

We will immediately set adequate structuring in the project, which we will adhere to as it, the project, develops. Let's create a folder *Modules* in which we will place our files to work with this or that protocol or device. And immediately add a subfolder, as well as a

header and a file with the source code for the driver **ENC28J60**, which we will implement:



The entire project can, as always, be downloaded at the end of the article, and at the end of the article I will also place the full code of these files for the most convenient reading. In the meantime, we are starting to sort it out step by step.

Initialization and work with ENC28J60.

Let's start with the basic functions for sending and receiving SPI data, which are accompanied by the installation and reset of the Chip Select signal:

```

1.  /*-----*/
2.  staticvoidSetCS(ENC28J60_CS_State state)
3.  {
4.      HAL_GPIO_WritePin(ENC28J60_CS_PORT, ENC28J60_CS_PIN, (GPIO_PinState)state);
5.  }
6.
7.
8.
9.  /*-----*/

```

The function arguments, as well as the STM32 ports used for Chip Select and Reset, are declared in the header file *enc28j60.h*:

```

1.  #define ENC28J60_CS_PORT GPIOB
2.  #define ENC28J60_CS_PIN GPIO_PIN_3
3.
4.  #define ENC28J60_RESET_PORT GPIOB
5.  #define ENC28J60_RESET_PIN GPIO_PIN_6
6.
7.
8.  typedefenum
9.  {
10.     CS_LOW = 0,
11.     CS_HIGH = 1,
12. } ENC28J60_CS_State;

```

Then, sequentially, the functions for writing/sending bytes, writing multiple bytes, and reading bytes are:

```

1.  /*-----*/
2.  staticvoidWriteByte(uint8_t data)
3.  {
4.      HAL_StatusTypeDef res = HAL_SPI_Transmit(&hspi1, &data, 1, ENC28J60_SPI_TIMEOUT);
5.  }
6.
7.
8.  /*-----*/
9.  staticvoidWriteBytes(uint8_t* data, uint16_t size)
10. {
11.     HAL_StatusTypeDef res = HAL_SPI_Transmit(&hspi1, data, size, ENC28J60_SPI_TIMEOUT);
12. }
13.
14.
15. /*-----*/
16. staticuint8_tReadByte()
17. {
18.     uint8_t txData = 0x00;
19.     uint8_t rxData = 0x00;
20.     HAL_StatusTypeDef res = HAL_SPI_TransmitReceive(&hspi1, &txData, &rxData, 1, ENC28J60_SPI_TIMEOUT);
21.     return rxData;
22. }
23.
24.
25.
26. /*-----*/

```



ENC28J60 supports a standard mechanism for interacting with an external controller - by reading and writing values to registers responsible for a particular function. All registers fall into three main groups:

- control registers
- registers for working with Ethernet buffer
- and PHY registers

Today our focus will shift to working with the first of these groups. The managing registers are distributed among 4 banks:

TABLE 3-1: ENC28J60 CONTROL REGISTER MAP

Bank 0		Bank 1		Bank 2		Bank 3	
Address	Name	Address	Name	Address	Name	Address	Name
00h	ERDPTL	00h	EHT0	00h	MACON1	00h	MAADR1
01h	ERDPPTH	01h	EHT1	01h	MACON2	01h	MAADR0
02h	EWRPTL	02h	EHT2	02h	MACON3	02h	MAADR3
03h	EWRPTH	03h	EHT3	03h	MACON4	03h	MAADR2
04h	ETXSTL	04h	EHT4	04h	MABBIPG	04h	MAADR5
05h	ETXSTH	05h	EHT5	05h	—	05h	MAADR4
06h	ETXNDL	06h	EHT6	06h	MAIPGL	06h	EBSTSD
07h	ETXNDH	07h	EHT7	07h	MAIPGH	07h	EBSTCON
08h	ERXSTL	08h	EPMM0	08h	MACLCON1	08h	EBSTCSL
09h	ERXSTH	09h	EPMM1	09h	MACLCON2	09h	EBSTCSH
0Ah	ERXNDL	0Ah	EPMM2	0Ah	MAMXFLL	0Ah	MISTAT
0Bh	ERXNDH	0Bh	EPMM3	0Bh	MAMXFLH	0Bh	—
0Ch	ERXRDPTL	0Ch	EPMM4	0Ch	Reserved	0Ch	—
0Dh	ERXRDPTH	0Dh	EPMM5	0Dh	MAPHSUP	0Dh	—
0Eh	ERXWRPTL	0Eh	EPMM6	0Eh	Reserved	0Eh	—
0Fh	ERXWRPTH	0Fh	EPMM7	0Fh	—	0Fh	—
10h	EDMASTL	10h	EPMCSL	10h	Reserved	10h	—
11h	EDMASTH	11h	EPMCSH	11h	MICON	11h	—
12h	EDMANDL	12h	—	12h	MICMD	12h	EREVID
13h	EDMANDH	13h	—	13h	—	13h	—
14h	EDMADSTL	14h	EPMDL	14h	MIREGADR	14h	—
15h	EDMADSTH	15h	EPMDH	15h	Reserved	15h	ECOCON
16h	EDMACSL	16h	EWOLIE	16h	MIWRL	16h	Reserved
17h	EDMACSH	17h	EWOLIR	17h	MIWRH	17h	EFLOCON
18h	—	18h	ERXFCON	18h	MIRDL	18h	EPAUSL
19h	—	19h	EPKTCNT	19h	MIRDH	19h	EPAUSH
1Ah	Reserved	1Ah	Reserved	1Ah	Reserved	1Ah	Reserved
1Bh	EIE	1Bh	EIE	1Bh	EIE	1Bh	EIE
1Ch	EIR	1Ch	EIR	1Ch	EIR	1Ch	EIR
1Dh	ESTAT	1Dh	ESTAT	1Dh	ESTAT	1Dh	ESTAT
1Eh	ECON2	1Eh	ECON2	1Eh	ECON2	1Eh	ECON2
1Fh	ECON1	1Fh	ECON1	1Fh	ECON1	1Fh	ECON1

Accordingly, if we have bank 0 active, then when writing a register with address 0x00 we will change the value of the register *ERDPTL*, if bank 1 - then *EHT0*. At the end of each bank there are five identical registers that are common, that is, no matter which bank is active, these addresses (0x1B - 0x1F) will belong to the same registers *EIE*, *EIR*, *ESTAT*, *ECON2*, *ECON1*.

The first letters of the registers indicate the identity of the registers:

- E - Ethernet registers
- MA - MAC Register Group
- MII - MII register group

This may seem very confusing, opaque and incomprehensible, but everything will fall into place as we solve practical problems. So don't get too hung up on it.

I will not give a description of all the registers and their bits, since this will be a banal translation of datasheet; this is of no use, except for increasing the volume of the article.

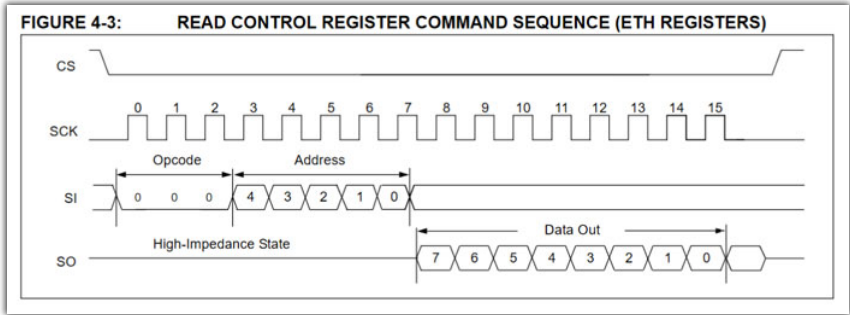
Work with **ENC28J60** built on the following format: the controller sends a one-byte command followed by data. Depending on the type of command, it is either followed by a data transfer to ENC28J60 or a data read from it. There are 7 of these commands in total:



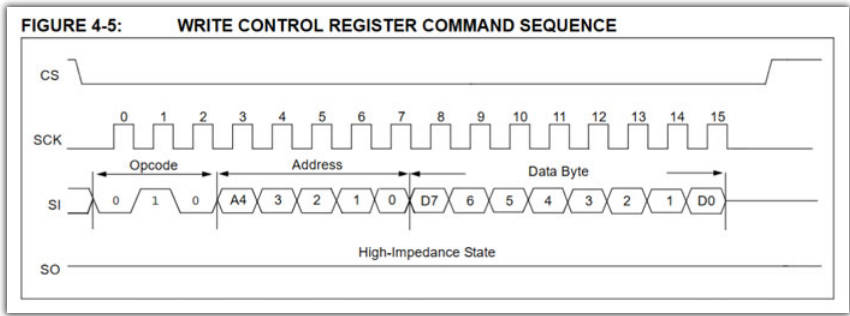
Instruction Name and Mnemonic	Byte 0		Byte 1 and Following
	Opcode	Argument	Data
Read Control Register (RCR)	0 0 0	a a a a a	N/A
Read Buffer Memory (RBM)	0 0 1	1 1 0 1 0	N/A
Write Control Register (WCR)	0 1 0	a a a a a	d d d d d d d d
Write Buffer Memory (WBM)	0 1 1	1 1 0 1 0	d d d d d d d d
Bit Field Set (BFS)	1 0 0	a a a a a	d d d d d d d d
Bit Field Clear (BFC)	1 0 1	a a a a a	d d d d d d d d
System Command (Soft Reset) (SC)	1 1 1	1 1 1 1 1	N/A

Legend: a = control register address, d = data payload.

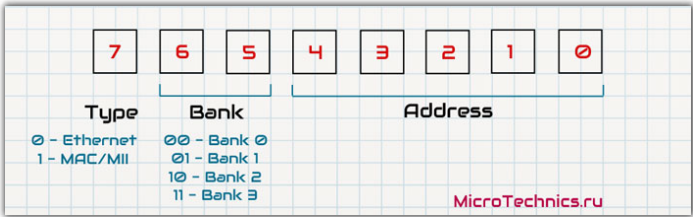
Accordingly, as indicated here, the command byte consists of code *OpCode* (3 bits) and 5 bits that are either responsible for the register address from the table above or have a fixed value for some commands. Let's look at the command to read the control register, for example *Read Control Register*):



Everything is exactly as we expected - a command byte (3 bits of opcode + 5 bits of address), followed by data with a register value transmitted from ENC28J60. The write command is the same, except that after the command, we transfer the data to the ENC28J60 to write to the register:



In general, it's a simple model, we implement it practically. First, let's identify all existing registers. The nuance here is that in addition to the address, each register also has its own type (Ethernet, MAC, MII), and also belongs to some bank. Therefore, to describe each of the registers, we will use 8 bits in the following way:



- 5 least significant bits - register address

- 2 bits for storing bank number information - *0b00 - Bank0, 0b01 - Bank1, 0b10 - Bank2, 0b11 - Bank3*
- and the most significant bit is a flag indicating that the register belongs to either an Ethernet group (bit is 0) or to one of the MAC or MII groups (bit is 1)

This is implemented in the code according to the following scheme. Let me remind you that the full code of the files and the entire draft will be at the end of the article, but here, to maintain the integrity of the presentation, there will only be excerpts to convey the main essence. I will be happy to answer any additional questions 👍

So, in the header file are defined `enum` for banks and register types:

```
1. typedef enum
2. {
3.     BANK_0,
4.     BANK_1,
5.     BANK_2,
6.     BANK_3,
7. } ENC28J60_RegBank;
8.
9. typedef enum
10. {
11.     ETH_REG,
12.     MAC_MII_REG,
13. } ENC28J60_RegType;
```

And a set of defines:

```
1. #define ENC28J60_REG_BANK_OFFSET 5
2. #define ENC28J60_REG_TYPE_OFFSET 7
3.
4. #define ENC28J60_REG_BANK_MASK 0x60
5. #define ENC28J60_REG_TYPE_MASK 0x80
6. #define ENC28J60_REG_ADDR_MASK 0x1F
7.
8. #define ENC28J60_BANK_0_BITS (BANK_0 << ENC28J60_REG_BANK_OFFSET)
9. #define ENC28J60_BANK_1_BITS (BANK_1 << ENC28J60_REG_BANK_OFFSET)
10. #define ENC28J60_BANK_2_BITS (BANK_2 << ENC28J60_REG_BANK_OFFSET)
11. #define ENC28J60_BANK_3_BITS (BANK_3 << ENC28J60_REG_BANK_OFFSET)
12. #define ENC28J60_BANK_COMMON_BITS (BANK_0 << ENC28J60_REG_BANK_OFFSET)
13.
14. #define ENC28J60_COMMON_REGS_ADDR 0x1B
15.
16. #define ENC28J60_ETH_REG_BIT (ETH_REG << ENC28J60_REG_TYPE_OFFSET)
17. #define ENC28J60_MAC_MII_REG_BIT (MAC_MII_REG << ENC28J60_REG_TYPE_OFFSET)
```

All these definitions follow the register description scheme that we have analyzed. For the bank number we have the 5th and 6th of 8 bits, respectively `ENC28J60_REG_BANK_OFFSET = 5` and `ENC28J60_REG_BANK_MASK = 0x60` (*0b01100000*). Similarly, for the bit that is responsible for the register type (we have it 7th).

Also, for those 5 registers that do not belong to a specific bank, we will use the same bit value as for the zero bank (`ENC28J60_BANK_COMMON_BITS`). But to identify them, we will not use the bit values, as for other registers, but the address value itself. Because we know that these unique registers are located at addresses older than `ENC28J60_COMMON_REGS_ADDR(0x1B)`.

Bit masks will help extract bank and register type data from the 8-bit value. As a result, the definition of each of our registers will look like this. Let's take a few banks for clarity:

```
1. #define EIE (0x1B | ENC28J60_ETH_REG_BIT | ENC28J60_BANK_COMMON_BITS)
2.
3. #define EWRPTL (0x02 | ENC28J60_ETH_REG_BIT | ENC28J60_BANK_0_BITS)
4.
5. #define MACON4 (0x03 | ENC28J60_MAC_MII_REG_BIT | ENC28J60_BANK_2_BITS)
6.
7. #define EBSTCSH (0x09 | ENC28J60_ETH_REG_BIT | ENC28J60_BANK_3_BITS)
```

The result is an elegant register detection system: **Register = Address | Type | Bank**, where:

23

- Address - the address value from the table
- Type - `ENC28J60_ETH_REG_BIT` or `ENC28J60_MAC_MII_REG_BIT`
- Bank - `ENC28J60_BANK_0_BITS` - `ENC28J60_BANK_3_BITS`

To work with certain register bits, we determine their positions. Take the ECON1 register:

```
1. #define ECON1_TXRST_BIT (1 << 7)
```

```

2. #define ECON1_RXRST_BIT (1 << 6)
3. #define ECON1_DMAST_BIT (1 << 5)
4. #define ECON1_CSUMEN_BIT (1 << 4)
5. #define ECON1_TXRTS_BIT (1 << 3)
6. #define ECON1_RXEN_BIT (1 << 2)
7. #define ECON1_BSEL1_BIT (1 << 1)
8. #define ECON1_BSEL0_BIT (1 << 0)

```

That's it, let's move on to creating functions. A group of functions that extract data about the register's type, bank, and address from the 8-bit value:

```

1. /*-----*/
2. static ENC28J60_RegType getRegType(uint8_t reg)
3. {
4.     ENC28J60_RegType type = (ENC28J60_RegType)((reg & ENC28J60_REG_TYPE_MASK)>> ENC28J60_REG_TYPE_OFFSET);
5.     return type;
6. }
7.
8.
9.
10. /*-----*/
11. static ENC28J60_RegBank getRegBank(uint8_t reg)
12. {
13.     ENC28J60_RegBank bank = (ENC28J60_RegBank)((reg & ENC28J60_REG_BANK_MASK)>> ENC28J60_REG_BANK_OFFSET);
14.     return bank;
15. }
16.
17.
18.
19. /*-----*/
20. static uint8_t getRegAddr(uint8_t reg)
21. {
22.     uint8_t addr = (reg & ENC28J60_REG_ADDR_MASK);
23.     return addr;
24. }
25.
26.
27.
28. /*-----*/

```

Before any register transactions, we will need to check the current active bank in ENC28J60, the number of which is stored in the two least significant bits of the register *ECON1*:

ECON1	TXRST	RXRST	DMAST	CSUMEN	TXRTS	RXEN	BSEL1	BSEL0	0000 0000	15
-------	-------	-------	-------	--------	-------	------	-------	-------	-----------	----

And if we need a register from another bank, then the bank should be switched. And in order not to subtract this value from the register every time, we will store the current bank number in the variable:

```

1. static ENC28J60_RegBank curBank = BANK_0;

```

Bank verification function:

```

1. /*-----*/
2. static void CheckBank(uint8_t reg)
3. {
4.     uint8_t regAddr = getRegAddr(reg);
5.     if(regAddr < ENC28J60_COMMON_REGS_ADDR)
6.     {
7.         ENC28J60_RegBank regBank = getRegBank(reg);
8.         if(curBank != regBank)
9.         {
10.            uint8_t econ1Addr = getRegAddr(ECON1);
11.
12.            //Clear bank bits
13.            SetCS(CS_LOW);
14.            WriteCommand(BIT_FIELD_CLEAR, econ1Addr);
15.            WriteByte(ECON1_BSEL1_BIT | ECON1_BSEL0_BIT);
16.            SetCS(CS_HIGH);
17.
18.            //Set bank bits
19.            SetCS(CS_LOW);
20.            WriteCommand(BIT_FIELD_SET, econ1Addr);
21.            WriteByte(regBank);
22.            SetCS(CS_HIGH);

```

```

23.
24.     curBank = regBank;
25. }
26. }
27. }
28.
29.
30.
31. /*-----*/

```

If the register bank passed by the argument does not match the current one, we switch the bank using the function `WriteCommand()` c, which we'll discuss below. Moreover, this entire verification mechanism is needed only for registers whose address values are numerically smaller `ENC28J60_COMMON_REGS_ADDR`.

Let's remember the 7 commands that ENC28J60 provides to us and define them:

```

1. typedef enum
2. {
3.     READ_CONTROL_REG,
4.     READ_BUFFER_MEM,
5.     WRITE_CONTROL_REG,
6.     WRITE_BUFFER_MEM,
7.     BIT_FIELD_SET,
8.     BIT_FIELD_CLEAR,
9.     SYSTEM_RESET,
10.    COMMANDS_NUM,
11. } ENC28J60_Command;

```

The command opcodes, in turn, are in the array:

```

1. static uint8_t commandOpCodes[COMMANDS_NUM] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x07};

```

The command byte sending function is:

```

1. /*-----*/
2. static void WriteCommand(ENC28J60_Command command, uint8_t argData)
3. {
4.     uint8_t data = 0;
5.     data = (commandOpCodes[command] << ENC28J60_OP_CODE_OFFSET) | argData;
6.     WriteByte(data);
7. }
8.
9.
10.
11. /*-----*/

```

And based on this function, we will create all commands. Let's look at a couple of them, reading the register:

```

1. /*-----*/
2. static uint8_t ReadControlReg(uint8_t reg)
3. {
4.     uint8_t data = 0;
5.     ENC28J60_RegType regType = getRegType(reg);
6.     uint8_t regAddr = getRegAddr(reg);
7.     CheckBank(reg);
8.
9.     SetCS(CS_LOW);
10.    WriteCommand(READ_CONTROL_REG, regAddr);
11.
12.    if(regType == MAC_MII_REG)
13.    {
14.        ReadByte();
15.    }
16.    data = ReadByte();
17.
18.    SetCS(CS_HIGH);
19.    return data;
20. }
21.
22.
23.
24. /*-----*/

```

23



The argument is the value of the register in the format we defined to describe the controller registers. From this value we then get the bank and the type of register. Note that for MAC/MII registers you need to read an additional byte that does not carry any semantic load, this is how ENC28J60 works.

Reset command:

```

1.  /*-----*/
2.  static void SystemReset()
3.  {
4.      SetCS(CS_LOW);
5.      WriteCommand(SYSTEM_RESET, ENC28J60_RESET_COMMAND_ARG);
6.      SetCS(CS_HIGH);
7.
8.      curBank = BANK_0;
9.      HAL_Delay(1);
10. }
11.
12.
13.
14. /*-----*/

```

Here is the final list of those 7 function commands to interact with ENC28J60:

```

1. void BitFieldSet(uint8_t reg, uint8_t regData);
2. void BitFieldClear(uint8_t reg, uint8_t regData);
3. uint8_t ReadControlReg(uint8_t reg);
4. void WriteControlReg(uint8_t reg, uint8_t regData);
5. void WriteBufferMem(uint8_t *data, uint16_t size);
6. void ReadBufferMem(uint8_t *data, uint16_t size);
7. static void SystemReset();

```

But you won't be able to limit yourself to this. First, many registers are a complementary pair, meaning the 16-bit value is stored in 2 registers (for example *EWRPTL* and *EWRPTH*). The most significant 8 bits are in *EWRPTH*, the least significant bits are in *EWRPTL*. So we will need to work with them as well:

```

1.  /*-----*/
2.  static void WriteControlRegPair(uint8_t reg, uint16_t regData)
3.  {
4.      WriteControlReg(reg, (uint8_t)regData);
5.      WriteControlReg(reg + 1, (uint8_t)(regData >> 8));
6.  }
7.
8.
9.
10. /*-----*/
11. static uint16_t ReadControlRegPair(uint8_t reg)
12. {
13.     uint16_t data = 0;
14.     data = (uint16_t)ReadControlReg(reg) | ((uint16_t)ReadControlReg(reg + 1) << 8);
15.     return data;
16. }
17.
18.
19.
20. /*-----*/

```

And secondly, there is a separate group of PHY registers that are not related to the banks that we discussed, which have their own address space. We simply define them by the following addresses:

```

1. #define PHCON1 (0x00)
2. #define PHSTAT1 (0x01)
3. #define PHID1 (0x02)
4. #define PHID2 (0x03)
5. // ...

```

The peculiarity of these registers is that they cannot be worked with directly, but only through *MIREGADR* and *MIRDL/MIRDH*:

```

1.  /*-----*/
2.  static uint16_t ReadPhyReg(uint8_t reg)
3.  {
4.      uint16_t data = 0;
5.      uint8_t regAddr = getRegAddr(reg);
6.
7.      WriteControlReg(MIREGADR, regAddr);
8.      BitFieldSet(MICMD, MICMD_MIIRD_BIT);
9.
10.     while((ReadControlReg(MISTAT) & MISTAT_BUSY_BIT) != 0);
11.
12.     BitFieldClear(MICMD, MICMD_MIIRD_BIT);
13.     data = ReadControlRegPair(MIRDL);
14. }

```



```

15.     return data;
16. }
17.
18.
19.
20. /*-----*/
21. static void WritePhyReg(uint8_t reg, uint16_t regData)
22. {
23.     uint8_t regAddr = getRegAddr(reg);
24.
25.     WriteControlReg(MIREGADR, regAddr);
26.     WriteControlRegPair(MIWRL, regData);
27.
28.     while((ReadControlReg(MISTAT) & MISTAT_BUSY_BIT) != 0);
29. }
30.
31.
32.
33. /*-----*/

```

To read the PHY register, enter its address in *MIREGADR* and we set the bit *MICMD_MIIRD_BIT* in register *MICMD*. Bit *MISTAT_BUSY_BIT* register *MISTAT* signals that the operation has been completed and we expect it to be reset. After this we clean it *MICMD_MIIRD_BIT* and read the PHY value of the register from the pair *MIRDL/MIRDH*. In general, everything is clear in code.

The database for working with ENC28J60 is completely ready, so we will now complete the starting article of the loop with the initialization of the module, which we will use in subsequent projects:

```

1. /*-----*/
2. void ENC28J60_Init()
3. {
4.     HAL_GPIO_WritePin(ENC28J60_RESET_PORT, ENC28J60_RESET_PIN, GPIO_PIN_RESET);
5.     HAL_Delay(50);
6.     HAL_GPIO_WritePin(ENC28J60_RESET_PORT, ENC28J60_RESET_PIN, GPIO_PIN_SET);
7.     HAL_Delay(50);
8.
9.     SystemReset();
10.
11.     //Rx/Tx buffers
12.     WriteControlRegPair(ERXSTL, ENC28J60_RX_BUF_START);
13.     WriteControlRegPair(ERXNDL, ENC28J60_RX_BUF_END);
14.
15.     WriteControlRegPair(ERDPTL, ENC28J60_RX_BUF_START);
16.
17.     //MAC address
18.     WriteControlReg(MAADR1, macAddr[0]);
19.     WriteControlReg(MAADR2, macAddr[1]);
20.     WriteControlReg(MAADR3, macAddr[2]);
21.     WriteControlReg(MAADR4, macAddr[3]);
22.     WriteControlReg(MAADR5, macAddr[4]);
23.     WriteControlReg(MAADR6, macAddr[5]);
24.
25.     WriteControlReg(MACON1, MACON1_TXPAUS_BIT | MACON1_RXPAUS_BIT | MACON1_MARXEN_BIT);
26.     WriteControlReg(MACON3, MACON3_PADCFG0_BIT | MACON3_TXCRCEN_BIT | MACON3_FRMLNEN_BIT);
27.
28.     WriteControlRegPair(MAIPGL, ENC28J60_NBB_PACKET_GAP);
29.     WriteControlReg(MABBIPG, ENC28J60_BB_PACKET_GAP);
30.
31.     WriteControlRegPair(MAMXFLL, ENC28J60_FRAME_DATA_MAX);
32.
33.     //PHY Resistors
34.     WritePhyReg(PHCON2, PHCON2_HLDIS_BIT);
35.
36.     ENC28J60_StartReceiving();
37. }
38.
39.
40.
41. /*-----*/

```

23

Again, I won't write down the bits copying this information from the datasheet, if necessary, ask questions in the comments, I'll be happy to help. In the next article, we will look at the receiving and transmitting buffers that will be used to receive and send Ethernet frames.

+ File enc28j60.c

Project reference: [MT_ENC28J60_Part_1](#)



- › Manchester code. Part 1. Data encoding.



B I U $\frac{1}{2}$ $\frac{2}{3}$ $\frac{3}{4}$ " </> {} [+]

⚡ 🔥 Old ones ▼



⌚ 4 years ago

↩ Reply



Author

⌚ 4 years ago

↩ Reply



🕒 4 years ago

↩ Reply



Author

🕒 4 years ago

↩ Reply

23

**Heinrich**

🕒 4 years ago

Oh, is there STM32 with built-in Gigabit Ethernet? Or at worst, an external module, but Gigabit?

➔ Reply

**Rodion**

🕒 3 years ago

Hello! Please tell me why in the function **ReadPhyReg** the MIRD_L least significant byte is read, when is the most significant byte read, and when is it two? The documentation says that the data lies in MIRD_L and MIRD_H registers.

➔ Reply

**Aveal MT**

Administrator

Author

🗨️ Reply to comment [Rodion](#)

🕒 3 years ago

Good afternoon! There through **ReadControlRegPair()** a couple of registers are read at once.

➔ Reply

**Rodion**

🗨️ Reply to comment [Aveal MT](#)

🕒 3 years ago

Thanks, I figured it out

➔ Reply

**Timur**

🕒 3 years ago

Hello everyone, I found a small error in the code:

```
static void WriteBytes(uint8_t* data, uint8_t size){
    HAL_StatusTypeDef res = HAL_SPI_Transmit(&hspi1, data, size, ENC28J60_SPI_TIMEOUT);
}
```

the function accepts the size of a packet of the form uint8_t size for subsequent sending to ENC via SPI, and this limits the packet size to 255 bytes, corrected to uint16_t size, the packet was sent with all 1000 bytes of data correctly.

➔ Reply

**Aveal MT**

Administrator

Author

🗨️ Reply to comment [Timur](#)

🕒 3 years ago

Super! Damn, just projects, everything will have to be re-uploaded (

➔ Reply

**Ivan**

🕒 2 years ago

Tell me, with Ethernet built-in, is there a ready-made board at level F405 and above?

➔ Reply

23

**Aveal MT**

Administrator

Author

🗨️ Reply to comment [Ivan](#)

🕒 2 years ago

How much higher? I have it lying down [NUCLEO-H743ZI](#):



→ Reply

P

Pavel

Reply to comment [Aveal MT](#)
2 years ago

Does IPv6 work on this board? On STM32F74x, STM32F76x silicone bugs in the IPv6 part

→ Reply



Aveal MT

Administrator

Author

Reply to comment [Pavel](#)
2 years ago

It depends on the stack, not on the chip

→ Reply

U

usr

Reply to comment [Aveal MT](#)
1 year ago

not a fact! the call depends on what levels are implemented in the "network chip", if in **enc28j60** everything is done "with your hands", then **W5500** almost all routines are implemented in the chip itself

→ Reply



Aveal MT

Administrator

Author

Reply to comment [usr](#)
1 year ago

Here, "chip" meant a microcontroller (STM32xx).

→ Reply

U

usr

Reply to comment [Aveal MT](#)
1 year ago

total, calls:

- from the iron implementation of the network chip
- from the software implementation of the network stack

→ Reply



Aveal MT

Administrator

Author

Reply to comment [usr](#)
1 year ago

That's right, but I haven't heard about the hardware bugs of the microcontroller itself, so I asked.

→ Reply

U

usr

Reply to comment [Aveal MT](#)
1 year ago

23



what kind of bugs are mentioned?

silicone bugs in the IPv6 part

➔ Reply

A**Alexander**

🕒 1 year ago

Hello, in all circuits, power is connected to 3.3V. Why did you connect to 5 V?

➔ Reply

**Aveal MT****Administrator****Author**

➔ Reply to comment [Alexander](#)

🕒 1 year ago

Good afternoon! Wrong in the picture looks like... Thank you, I'll correct it now.

➔ Reply

B**Vyacheslav**

🕒 8 months ago

Hello. Maybe someone can tell you how to properly configure USB and ETH collaboration on LAN8720. Problem: when adding ETH to a project, USB stops working (project on stm32h723zgt6)

➔ Reply

**Aveal MT****Administrator****Author**

➔ Reply to comment [Vyacheslav](#)

🕒 7 months ago

Good afternoon! I'm away now, I can't give you any advice... If it is still relevant, when I return, I will be glad to help.

➔ Reply

**MicroTechnics | STM32, Raspberry Pi...**

4,562 followers



23

Follow on VK



[The entrance](#)[Registration](#)

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](#)

23

[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.](#)