Ethernet Communication in Microcontroller Systems

Technical Report · January 2017

DOI: 10.13140/RG.2.2.27380.35206

1 author:

Davor Antonic
Antonic Ltd.
15 PUBLICATIONS **28** CITATIONS

Ethernet Communication in Microcontroller Systems[1]

Davor Antonic
Faculty of Chemical Engineering and Technology
davor@antonic.hr

January 2017

Abstract *– As a communication protocol for low cost microcontroller systems Ethernet is becoming superior alternative to traditional communication protocols as RS-232, RC-485, or CAN. High speed, long range and moderate cost, associated with availability of dedicated chips that implements Physical and Data Link layers make Ethernet the protocol of choice for many applications. This paper describes development of Ethernet interface for the Atmel AVR XMEGA microcontroller, based on Microchip's ENCx24J600 Ethernet Controller. Hardware implementation, software functions supporting ENCx24J600 and implementation of UDP protocol are explained in detail.*

Table of Contents

[1] This work has been fully supported by Croatian Science Foundation under the project IP-2013-11-8825

1. INTRODUCTION

Traditionally in low cost microcontroller systems simple communication protocols like RS-232 or RS-485 are used. Although simple to implement they are characterized by low speed, short distance (for RS-232) and lack of communication and synchronization protocols. CAN is more advanced, but more complicated to implement. It's nondestructive bitwise arbitration protocol imposes hard limit on speed-length product. E.g. maximum CAN speed at 40 m link is 1Mbit/s.

Ethernet offers numerous advantages. Data rate of 100 Mbit/s or more is easily achieved, distance is practically unlimited, and standardization and widespread support makes possible communication with a broad spectrum of systems and devices. Implementation complexity could be overcome by using specialized Ethernet controller chip, which implements Physical and Data Link layers of OSI model.

This paper deals with Microchip's ENCx24J600 Ethernet Controller (1) with integrated MAC and 10/100Base-T PHY. It provides connection to host microcontroller through 14 Mbit/s SPI interface or high throughput parallel interface. ENCx24J600 Ethernet Controller is implemented on Atmel AVR XMEGA-A3BU Xplained evaluation microcontroller system. Schematic, implementation of various ENCX24J600 functions and UDP protocol implementation are explained in detail.

Using complex libraries like (2) masks implementation details and allows the programmer to focus to the project goals. On the other hand, approaching hardware at the level of registers and individual bits gains deep insight into the chip functionality, which leads to more efficient and smaller code. Sometimes it will be simpler to implement required functions than to dig through complex library of which only few percent of provided functionality is needed.

Provided code and explanations are supplement to the ENCx24J600 Data Sheet, intended to clarify ENCx24J600 Ethernet controller features and programming. They are free to use without restrictions, as stated in the associated code files (3).

2. ENCx24J600 ETHERNET CONTROLLER

2.1. Device overview

The ENCx24J600 (1) is stand-alone Ethernet controller with Serial Peripheral Interface (SPI) or flexible parallel interface for communication with host microcontroller. Maximum data rate for SPI is 14 Mbit/s and for parallel interface in demultiplexed, 16-bit mode up to 160 Mbit/s. The only difference between the ENC424J600 (44 pin) and ENC624J600 (64 pin) devices are the number of parallel interface options they support. SPI functionality is the same for both devices, so ENC424J600 is preferable choice for SPI interface.

ENCx24J600 implements all of the IEEE 802.3 specifications applicable to 10Base-T and 100Base-TX Ethernet. Many optional features, such as auto-negotiation are supported. ENCx24J600 incorporate eleven software configurable receive filters to discard unwanted frames. Each device is preprogrammed with unique nonvolatile MAC address. During initialization it is copied to writable registers, which means that user application could assign different MAC address if required.

24-Kbyte on-chip RAM buffer, organized as 12K 16-bit words is available for transmit and receive buffer and for general purpose storage if desired. Internal 16-bit DMA controller supports memory copy operation and hardware checksum calculations.

ENCx24J600 incorporates three different cryptographic security engines, which perform encryptions, decryptions and mathematical computations most commonly used for network security functions. The engines implemented are Modular Exponentiation, MD5 and SHA-1 Hashing and AES.

2.2. Serial Peripheral Interface (SPI)

SPI is a synchronous serial communication interface used for short distance high speed communication (4). It is the master-slave architecture with single master. SPI requires three lines for communication: SCLK (Serial Clock), MOSI (Master Output - Slave Input) and MISO (Master Input - Slave Output). Additionally, for each slave device master should provide active low SS (Slave Select) signal. Master always initiates write cycle, which generates SCLK and outputs data bit-by-bit at MOSI line. Slave device should simultaneously put data sending to master at MISO line, synchronous with SCLK. There are four SPI modes that differ in SCLK polarity and phase in relation to data lines.

2.2.1. Physical implementation

Corresponding SPI lines on ENCx24J600 are named SCK, SO, SI and CS. The SPI port operates as a SPI slave port in Mode 0,0 (SCK is idle in a logic low state and data is clocked in on rising clock edges). The active low CS pin must be asserted while any operation is performed and return to inactive state when finished.

2.2.2. SPI instruction set

The ENCx24J600 SPI interface supports 47 different instructions for accessing various ENCx24J600 registers. Although full functionality can be achieved with only six instructions, using additional instructions will improve system performance. In described implementation following twelve SPI instructions are implemented:

Mnemonic	Length (bytes)	Description
SETETHRST	1	Issues System Reset by setting ETHRST (ECON2<4>)
SETEIE	1	Enable Ethernet Interrupts by setting INT (ESTAT<15>)
CLREIE	1	Disable Ethernet Interrupts by clearing INT (ESTAT<15>)
DMACKSUM	1	Configures and starts DMA checksum operation (sets ECON1<5:2> to 1000)
SETTXRTS	1	Sends an Ethernet packet by setting TXRTS (ECON1<1>)
RCRU	4	Read Control Register Unbanked – reads content of addressed 16-bit Special Function Register (SFR). Byte 1 – opcode, Byte 2 – address, Byte 3 – returned low byte, Byte 4 – returned high byte
WCRU	4	Write Control Register Unbanked – writes to addressed 16-bit Special Function Register (SFR). Byte 1 – opcode, Byte 2 – address, Byte 3 – low byte, Byte 4 – high byte
BFSU(*)	4	Bit Field Set Unbanked – Bytes 3 and 4 contains mask used to set corresponding bit in addressed SFR. If mask bit is '1', corresponding SFR bit will be set, if it is '0', SFR bit remains unchanged.
BFCU(*)	4	Bit Field Clear Unbanked – Bytes 3 and 4 contains mask used to clear corresponding bit in addressed SFR. If mask bit is '1', corresponding SFR bit will be cleared, if it is '0', SFR bit remains unchanged.
WGPWRPT	3	Write General Purpose Buffer Write Pointer (EGPWRPT).
WGPDATA	*N*	Writes data to General Purpose Buffer. Byte is written indirectly to address pointed by EGPWRPT and EGPWRPT is incremented.
RRXDATA	*N*	Reads data from Circular RX FIFO Buffer. Byte is fetched indirectly from address pointed by ERXDATA and ERXDATA is incremented.

(*) *It is advised to use BFS and BFC instructions to modify control registers, instead of reading register into host controller, modify it and write it back to ENCx24J600 (RCR followed by WCR). Such operation is not atomic, meaning that ENCx24J600 may change content of the register between read and write-back operations. BFS and BFC instructions are atomic, requires half of SPI cycles and results in simpler code.*

2.3. Memory organization

All memory in ENCx24J600 is implemented as volatile RAM, functionally divided into four areas:

- Special Function Registers (SFRs)
- PHY Special Function Registers
- Cryptographic Data Memory
- SRAM Buffer

The PHY SFRs configure, control and provide status information for the PHY module. They are located inside the PHY module so they are not directly accessible through the I/O interface. They can normally be left at default values.

The cryptography data memory is used to store key and data used by Cryptographic Security Engines. It can only be accessed through the DMA module.

Memory mapping depends on the selected I/O interface. For the SPI interface there are three memory address spaces:

- The SFR area – directly accessible 160 bytes linear memory space

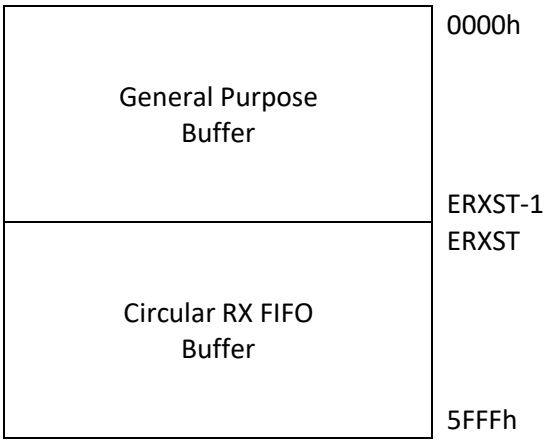
- The main memory area – linear, byte-addressable space of 32 Kbytes, where the first 24 Kbytes (0000h – 5FFFh) is implemented as the SRAM buffer. Through the SPI interface buffer can be accessed only indirectly, using several SFRs as memory pointers and virtual data window registers. Area between 7800h and 7C4Fh is assigned to Cryptographic Data and the rest is unimplemented.
- The PHY register area – linear, word-addressable memory space of 32 words accessible by the MIIM interface.

2.3.1. Special function registers

All ENCx24J600 functions are configured through 58 Special Function Registers (SFR's). For faster access they are organized in four banks, but it is also possible to address them directly, which requires additional SPI cycle. E.g. WCR banked instruction requires three bytes instead of four bytes WCRB, because the register banked address is inside the op-code byte. If wrong bank is active, additional one-byte Bank Select instruction is required. SFR's are detail explained in (1), c. 3.2.

2.3.2. SRAM Buffer

The SRAM buffer is a 12K word x 16-bit memory used for receive and transmit packet buffering and general purpose storage by the host microcontroller. Depending on application requirements size of receive buffer and general purpose / transmit buffer can be adjusted through the ERXST register. The default value of ERXST is 5340h, which allocates 21,312 bytes to general purpose buffer and 3,264 bytes to the RX buffer, which is sufficient to store two maximum length Ethernet frames



Since there is no dedicated transmit buffer, general purpose buffer is used to prepare packets to be transmitted. Host application should gradually build the packet with available data and request transmission when the packet is fully built. It is also possible to modify and (re)transmit packet directly from the receive buffer. The receive buffer is organized as circular FIFO buffer. After the memory at address 5FFFh is written to, the hardware will automatically wrap around and write the next byte of received data to the ERXST address.

For the SPI interface indirect access to the SRAM buffer is the only method available. ENCx24J600 provides three pairs of Read/Write pointers and three associated 8-bit data registers through which the SRAM data is read or written. Following the read/write operation, the appropriate pointer is automatically incremented in hardware. All pointers can be used to access any address within the

SRAM buffer. They differ from each other based on their address wrapping behavior. Indirect access and circular buffer configuration and behavior are explained in detail in (1), c. 3.5.5.

2.4. Initialization

Before using ENCx24J600 it must be initialized. SFR's contains meaningful default values, which considerably simplifies initialization procedure. In associate code ENCx24J600 is initialized as follows:

```
Wait for ENC SPI interface to initialize
Wait for stable clock
Reset ENCx24J600 and wait at least 25µs for the reset to take place
   and the SPI interface to begin functioning again
Confirm that the System Reset took place (check that EUDAST returned
   to default value of 0000h)
Wait at least 256 us for PHY initialization
Enable Ethernet, LED stretching, automatic MAC Address transmission,
   transmit and receive logic
Initialize 'NextPacketPointer' variable to ERXST
Disable reception of broadcast (ff-ff-ff-ff-ff-ff) frames (only frames
   having correct MAC address will be accepted)(*)
Enable reception
Enable packet received interrupt
```

(*) This option should be disabled for the implementation of the ARP protocol

2.5. Receiving packets

Incoming Ethernet frames are written to the circular receive buffer. Upon frame reception ENCx24J600 increments the Packet Counter bits, PKTCNT (ESTAT<7:0>), which indicates the number of pending frames.

Procedure of reading arrived packets from the RX buffer:

```
Set Receive Buffer Read Pointer (ERXRDPT) to the value of
   'NextPacketPointer' variable
Read first two bytes of the packet, which are the address of the next
   packet and write them to 'NextPacketPointer' variable
Read Receive Status Vector (next six bytes), extract the length of the
   frame
Read Ethernet (and higher level protocol) header
Read data
Update RXTAIL pointer to NextPacketPointer – 2
Decrement PKTCNT by asserting ECON1.PKTDEC
```

Above procedure is intended to be executed on Packet Received Interrupt. If polling is used instead of the interrupt, procedure should wait for packet reception (for the condition PKTCNT>0).

2.1. Transmitting packets

The packet to be transmitted is defined by the Transmit Data Start Pointer (ETXST), and the Transmit Buffer Length Pointer (ETXLEN). The ETXLEN bytes from the address indicated by ETXST will be transmitted. If the end of the general purpose buffer is encountered, the operation will wrap around

to the beginning of the general purpose buffer space (0000h). ENCx24J600 could be configured to automatically insert the source MAC address into the transmitted byte stream. The value of ETXLEN only indicates the number of bytes to read from memory, which means that the ENCx24J600 will automatically insert padding bytes to satisfy minimum packet length constraint, calculate and append the CRC field and optionally insert the source MAC address.

Procedure for preparing and transmitting the packet:

```
Set General Purpose Buffer Write Pointer (EGPWRPT) to the address
   where the packet will be constructed, set transmission start
   address (ETXST) to the same value
Set number of bytes into ETXLEN
Construct Ethernet (and higher level protocol) header
Write header and data to General Purpose Buffer through the EGPDATA
   register
Wait for the transmit subsystem to be ready (ongoing transmission
   completed)(*)
Start transmission
```

(*) Optionally Transmit Done Interrupt could be used. Since ENCx24J600 at 100Mbit/s will transmit data faster than the host microcontroller could prepare the next packet that is usually not benefitial.

2.2. Receive Filters

To minimize the number of frames that the host controller must process, ENCx24J600 incorporates eleven different software configurable receive filters to discard unwanted frames. Default settings are to accept only Broadcast frames and frames specifically addressed to the local MAC address. Invalid frames and those destined for other nodes will be automatically rejected. Additionally, in initialization procedure described in 2.4, Broadcast Collection Filter is disabled, accepting only frames specifically addressed to the local MAC address, thus restricting incoming traffic to absolute minimum.

2.3. Interrupts

The ENCx24J600 implements ten interrupt sources:

- Cryptographic Security Engines operations completion (four interrupts)
- PHY Link Status Change
- RX Packet Pending
- DMA Operation Completed
- Transmit Done
- Transmit Abort
- Receive Abort
- Packet Counter Full

If particular interrupt is disabled, host controller can check the condition by reading corresponding flag in Ethernet Interrupt Flag Register (EIR). To enable particular interrupt, corresponding bit in Ethernet Interrupt Enable Register (EIE) should be set, as well as the INT Global Interrupt Enable bit.

2.4. DMA Controller

ENCx24J600 incorporates a Direct Memory Access (DMA) controller, performing following functions:

- Copying data within the SRAM buffer
- Copying data between the SRAM buffer and the Cryptographic Engines data buffer
- Calculating a 16-bit checksum over a block of data, compatible with the checksum used in standard protocols, such as IP, UDP and TCP.

In described implementation DMA controller is used to calculate the checksum of data part of the UDP frame:

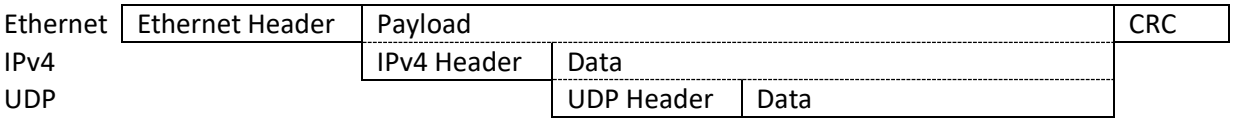
```
Set EDMAST to the start address
Set EDMALEN to the length of the input data
Execute DMACKSUM SPI command (clear DMACPY (ECON1<4>) to prevent a
   copy operation, clear DMANOCS (ECON1<2>) to select a checksum
   calculation, clear DMACSSD (ECON1<3>) to use the default seed of
   0000h, set DMAST to initiate the operation)
. . .  // programmatically calculate checksum of UDP pseudoheader
Wait for DMA to finish data checksum calculation (DMAST (ECON1<5>)
   cleared)
Read checksum from EDMACS
Combine with programmatically calculated checksum of UDP pseudoheader
```

3. UDP protocol

User datagram protocol (UDP) (5) uses a simple connectionless transmission model. It is defined inside Data segment of underlying protocol, e.g. IPv4. UDP provides source and destination port numbers for addressing different functions at the source and destination of the datagram and checksum for checking data integrity. Additionally, underlying protocol defines source and destination IP addresses and base Ethernet protocol defines MAC addresses.

Although UDP provides no guarantee of datagram delivery, ordering, or duplicate protection, it is suitable for purposes where error checking and correction is either not necessary or is performed in the application. Since there is no need for sending confirmation packets, exceptionally high data rates are possible, especially if connected systems are at the same local network segment.

Encapsulation of UDP datagram into lower layer packets is presented in the following figure:



3.1. Ethernet frame

Ethernet frame (6) is transported within the Ethernet packet at the physical layer, which starts with the seven bytes Preamble and Start Frame Delimiter byte, and ends with twelve bytes Interpacket

Gap. It defines source and destination through the MAC addresses and IP addresses are defined in higher level protocols.

Field Name	Size (bytes)	Description
MAC destination	6	Destination MAC address
MAC source	6	Source MAC address. Could be inserted automatically by the ENCx24J600.
Ethertype / Length	2	In IEEE 802.3 it is size of payload in bytes and in Ethernet II type frame it defines encapsulated protocol. Values of 1500 and below mean that it is used to indicate the size of the payload, while values of 1536 and above indicate that it is used as an EtherType, to signal which protocol is encapsulated in the payload of the frame.
Payload	46-1500	If contains less than 46 bytes could be automatically padded with zeroes by the ENCx24J600.
Frame check sequence	4	Cyclic redundancy check (CRC), could be calculated and appended automatically by the ENCx24J600. For incoming packets ENCx24J600 will check CRC and could automatically reject invalid frames.

3.2. IPv4 packet

Ethernet frame having Ethertype 0x0800 encapsulate Internet Protocol version 4 packet (7). IPv4 packet is fully defined inside Payload of Ethernet frame. Contains header and data section and has no data checksum or any other footer after the data section. Typically the link layer encapsulates IP packets in frames with a CRC footer that detects most errors, and higher layer checksum detects most other errors.

Field Name	Size (bytes)	Description
Version and IHL	1	**Version** (bit 4-7) is a protocol version field. For IPv4 it is equal to 4. **Internet Header Length** (bit 0-3) contains number of 32-bit words in the header. Since an IPv4 header may contain a variable number of options, this field specifies the size of the header, which also coincides with the offset to the data. It is five if Options are not used.
DSCP and ECN	1	**Differentiated Services Code Point** (bit 2-7) defines network traffic type. If not used should be 0. **Explicit Congestion Notification** (bit 0-1) allows end-to-end notification of network congestion without dropping packets. If not used should be 0.
Total Length	2	Defines the entire packet size, including header and data, in bytes.
Identification	2	Identification field is primarily used for uniquely identifying the group of fragments of single IP datagram.
Flags and Fragment Offset	2	**Flags** (bit 13-15) are used to control or identify fragments. **Fragment Offset** (bit 0-12) is measured in units of eight-byte blocks. It is 13 bits long and specifies the offset of a

		particular fragment relative to the beginning of the original unfragmented IP datagram.
Time To Live	1	Limit of a datagram's lifetime. It is specified in seconds, but in practice the field is used as a hop count.
Protocol	1	Defines the higher level protocol used in the data portion of the IP datagram. E.g. UDP protocol number is 11h. Standard Internet Protocol Numbers could be found in (8).
Header Checksum	2	It is used for error-checking of the header, calculated as the 16-bit one's complement of the one's complement sum of all 16-bit words in the header (9), (10).
Source IP Address	4	IPv4 address of the sender of the packet.
Destination IP Address	4	IPv4 address of the receiver of the packet.
Options	variable	Usually not used.
Data	variable	Content is interpreted based on the value of the *Protocol* field.

3.3. UDP datagram

UDP datagram structure is defined in (5).

Field Name	Size (bytes)	Description
Source Port	2	Identifies the sender's port. It should be zero if not used.
Destination Port	2	Identifies the receiver's port.
Length	2	The length in bytes of the UDP header and UDP data.
Checksum	2	May be used for error-checking of the header and data. It should be zero if not used. It is calculated the same way as the IPv4 Header Checksum.
Data	variable	

3.4. Address resolution

To use a higher level protocol like UDP it is necessary to establish mapping of Internet layer addresses into link layer addresses (IP to MAC address). General solution is to implement Address Resolution Protocol (ARP), but for static configuration, e.g. where known microcontroller modules are statically connected to the PC computer, ARP table can be created manually.

To establish communication between Windows PC and microcontroller board, following steps are required:

1. Assign IP addresses to PC and microcontroller

To assign IP address to PC Ethernet adapter, in 'IPv4 protocol properties' enter IP address. In sample code it is assumed that IP address 192.168.1.10 is assigned to PC Ethernet adapter.

IP address of microcontroller system is assigned programmatically. In sample code it is assumed that IP address 192.168.1.11 is assigned to the microcontroller.

2. Determine PC Ethernet adapter MAC address

To be able to send packets to the PC, microcontroller should know the PC Ethernet adapter MAC address. To determine it, open 'Command Prompt' and issue command '`ipconfig /all`'.

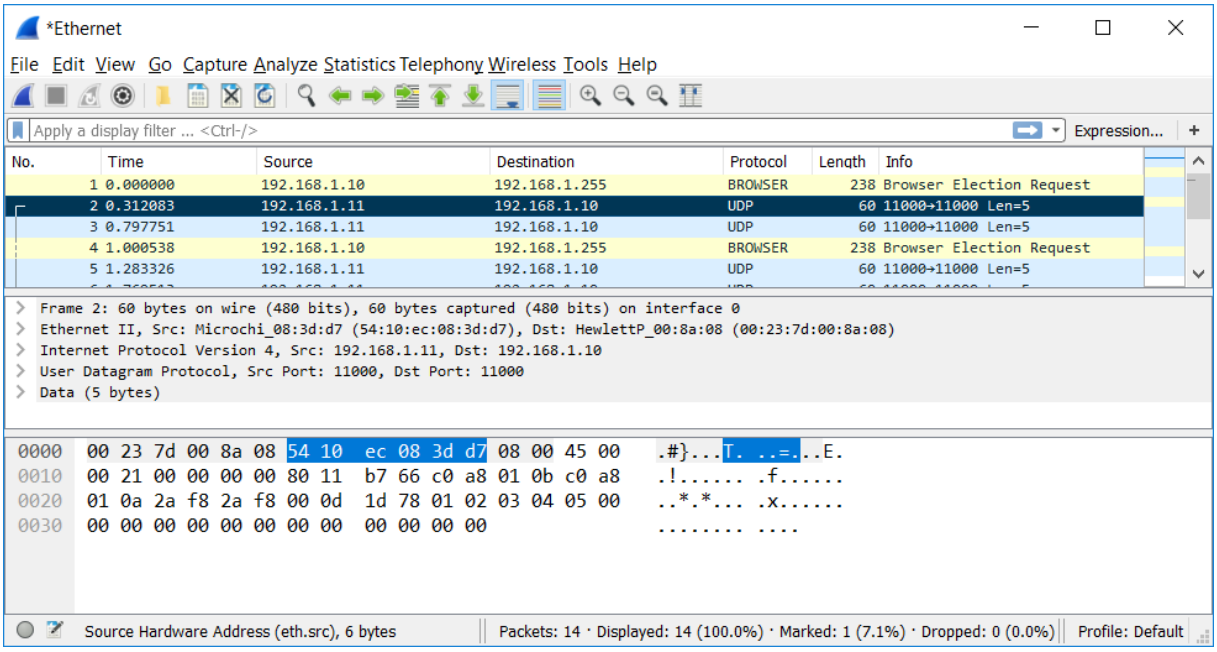
```
Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . :
   Description . . . . . . . . . . . : Intel(R) 82567LM Gigabit Network Connection
   Physical Address. . . . . . . . . : 00-23-7D-00-8A-08
   DHCP Enabled. . . . . . . . . . . : No
   Autoconfiguration Enabled . . . . : Yes
   Link-local IPv6 Address . . . . . : fe80::3c69:2652:538c:7c17%9(Preferred)
   IPv4 Address. . . . . . . . . . . : 192.168.1.10(Preferred)
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . :
   DHCPv6 IAID . . . . . . . . . . . : 50340733
   DHCPv6 Client DUID. . . . . . . . : 00-01-00-01-1E-28-27-A8-00-23-7D-00-8A-08
   DNS Servers . . . . . . . . . . . : fec0:0:0:ffff::1%1
                                       fec0:0:0:ffff::2%1
                                       fec0:0:0:ffff::3%1
   NetBIOS over Tcpip. . . . . . . . : Enabled
```

MAC address of the PC Ethernet adapter is 00-23-7D-00-8A-08h.

3. Determine microcontroller's Ethernet controller MAC address

ENCx24J600 Ethernet Controller has unique MAC address. To determine it, send packet using `ENC_SendUDPFrame` function (5.2.4) and inspect it with Wireshark network analyzer (11):



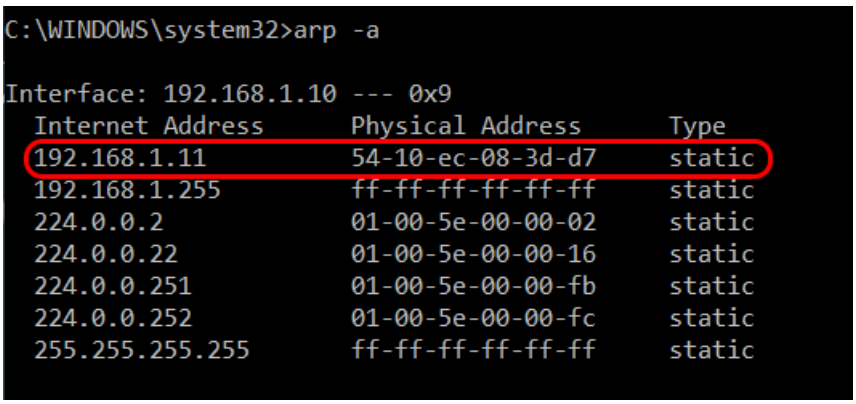
Locate UDP datagram sent from microcontroller (192.168.1.11) to PC (192.168.1.10). In this case MAC address of microcontroller's Ethernet controller is 54-10-EC-08-3D-D7h, where first three bytes identifies the Microchip Technology Inc. as a producer of ENCx24J600.

4. Assign microcontroller's MAC address to IP address

To be able to communicate with microcontroller through the IP address it is necessary to define static ARP entry that links IP to MAC address. Open 'Command prompt' in administrator mode (right click and select 'Run as administrator') and issue command:

```
netsh interface ipv4 add neighbors 9 192.168.1.11 54-10-ec-08-3d-d7
```

That will link IP address 192.168.1.11 to MAC address 54-10-EC-08-3D-D7 for interface number '9' (interface number could be found in 'Interface List' section displayed by `route print` command). To check that the entry is successfully added issue the `arp -a` command:

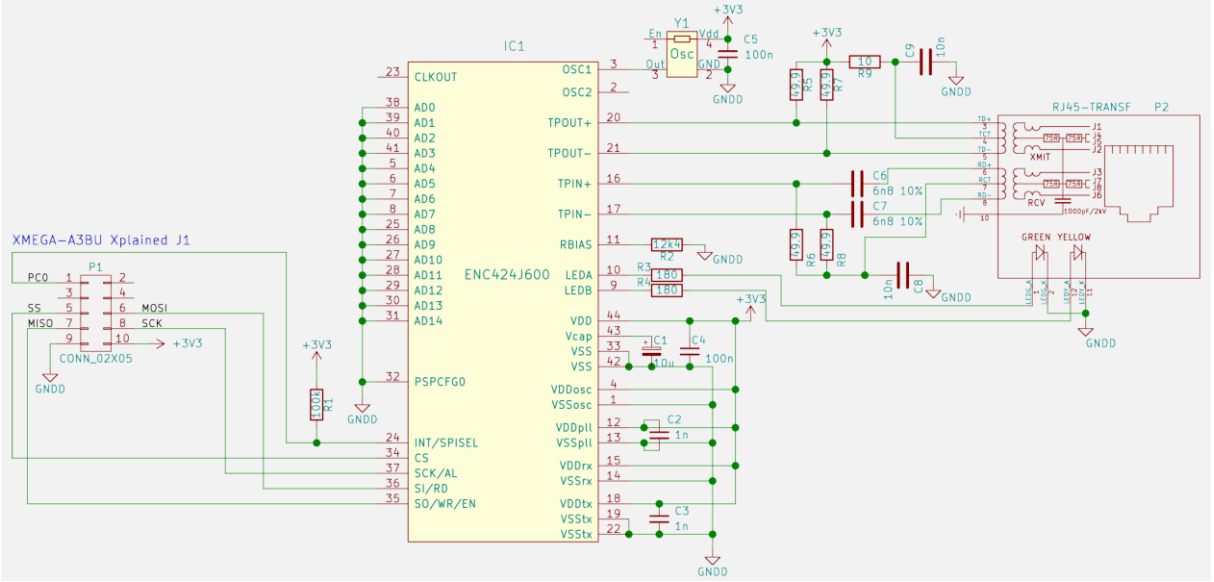


Added entry will lasts until the next system restart, so it is necessary to execute the `netsh` command after each system restart, e.g. through the batch file.

4. Ethernet module for XMEGA A3BU Xplained

The Atmel AVR XMEGA®-A3BU Xplained evaluation kit (12) is a hardware platform based on the Atmel ATxmega256A3BU 8-bit microcontroller. ENCx24J600 Ethernet controller is easily interfaced to the Xplained kit requiring four lines at PORT C SPI interface available at connector J1. Ethernet Signal Pins and External Magnetics are connected according to specifications in ENCx24J600 Data Sheet (1).

Ethernet module is designed as a daughterboard to the Xplained kit, connected to J1 connector. Schematic and PCB are designed in KiCad (13) . KiCad project (schematic and PCB) is available for free download at (3).



5. UDP protocol implementation at AVR XMEGA microcontroller

5.1. SPI protocol implementation

Header file SPI.h defines bit masks for SPI interface pins and three commands: SPI_WAIT that waits for current byte transmission to finish, and SPI_CS_ON and SPI_CS_OFF for asserting and clearing the Slave Select line (ENCx24J600 CS line, active low).

```
#define SPI_SS_bm               0x10 // bit mask for the SS pin
#define SPI_MOSI_bm             0x20 // bit mask for the MOSI pin
#define SPI_MISO_bm             0x40 // bit mask for the MISO pin
#define SPI_SCK_bm              0x80 // bit mask for the SCK pin

#define SPI_WAIT        while(!(SPIC.STATUS & SPI_IF_bm))       // wait for assertion of IF (transmit/
                                                                // receive completed)
#define SPI_CS_ON       PORTC.OUTCLR = SPI_SS_bm                // assert SS
#define SPI_CS_OFF      PORTC.OUTSET = SPI_SS_bm                // deassert SS
```

SPI.c contains function SPIC_Init() that configures microcontroller Port C, enables SPI master mode 0 and set SPI clock frequency to 8MHz. Transmission of one byte takes only 1µs, so polling through SPI_WAIT instruction is used instead of interrupt. Since interrupts are enabled, CPU could service an interrupt routine while waiting for completion of SPI transfer, e.g. to fetch data from the A/D converter at specified intervals.

```
void SPIC_Init()
{
        // configure SS, MOSI and SCK as output.
        PORTC.DIR = SPI_SS_bm | SPI_MOSI_bm | SPI_SCK_bm;
        SPI_CS_OFF;

        // SPI_C - Master mode 00, Clk_per / 4 (8MHz)
        SPIC.CTRL= SPI_PRESCALER_DIV4_gc | SPI_ENABLE_bm | SPI_MASTER_bm | SPI_MODE_0_gc;
}
```

To write one byte of data to SPI slave the code should write to SPI DATA register. That starts SPI clock generation and the hardware shifts the eight bits into the selected slave synchronous to the clock signal. After shifting one byte, the SPI clock generator stops and the SPI interrupt flag is set (SPI_IF bit

in SPI STATUS register). Interrupt Flag is automatically cleared by hardware when executing the corresponding interrupt vector. If polling is used, to clear the Interrupt Flag, software should first read the STATUS register and then access the DATA register. Therefore at the end of SPI instruction it is necessary to perform the dummy read of DATA register. That is not required if the next SPI cycle follows immediately, because writing to the DATA register will have the same effect.

```
SPIC.DATA = data;
SPI_WAIT;
(dummy = SPIC.DATA;)
```

Each SPI cycle is essentially bidirectional, which means that for reading data from slave, master should initiate write cycle. Since slave will ignore the data, some dummy value (e.g. zero) is normally used. By writing to SPI DATA register master starts clock generation and slave uses the clock to assert data to the MISO line. Master shifts data synchronously to the clock into input shift register from where they are transferred into the DATA register upon cycle completion.

```
SPIC.DATA = DUMMY;
SPI_WAIT;
data = SPIC.DATA;
```

5.2. ENCx24J600 functions

5.2.1. Implemented ENCx24J600 SPI Instructions

List of implemented SPI instructions are presented in 2.2.2. They are implemented through one to *N* SPI cycles, where first cycle is always the instruction op-code. Slave Select (ENCx24J600 CS) is asserted during all SPI cycles belonging to single instruction.

As an example of single byte instruction System Reset (SETETHRST) instruction is presented. To execute the instruction at ENCx24J600 it is sufficient to send the instruction op-code, which is CAh.

```
// ENCx24J600 System reset
void ENC_SETETHRST()
{
        uint8_t dummy;

        SPI_CS_ON;

        SPIC.DATA = 0xca;       // op code
        SPI_WAIT;
        dummy = SPIC.DATA;

        SPI_CS_OFF;
}
```

Example of four-byte instruction is Read Control Register Unbanked (RCRU), consisting of two write cycles to transfer op-code (20h) and Special Function Register's address, followed by two read cycles for fetching low and high byte of addressed register.

```
// Read Control Register Unbanked
// Fetches content of 16-bit ENCx24J600 register. Common registers are defined
// in ENCx24J600.h
uint16_t ENC_RCRU(uint8_t addr)
{
        uint8_t dummy, hi, lo;

        SPI_CS_ON;

        SPIC.DATA = 0x20;       // op code
```

```
        SPI_WAIT;
        dummy = SPIC.DATA;

        SPIC.DATA = addr;       // register address
        SPI_WAIT;
        dummy = SPIC.DATA;

        SPIC.DATA = DUMMY;
        SPI_WAIT;
        lo = SPIC.DATA;

        SPIC.DATA = DUMMY;
        SPI_WAIT;
        hi = SPIC.DATA;

        SPI_CS_OFF;

        return lo + (hi<<8);
}
```

Example of the *N*-byte instruction is Write EGPDATA (WGPDATA). Instruction op-code (2Ah) is followed by arbitrary number of bytes written to consecutive locations in General Purpose Buffer addressed by General Purpose Buffer Write Pointer (EGPWRPT) register. After sending last byte master must inactivate the CS line. To avoid additional level of data buffering, *N*-byte instructions WGPDATA and RRXDATA (Read from Receive Buffer Data Register) are not implemented as separate functions. They are incorporated into functions for receiving and transmitting UDP datagram, to interpret and construct the datagram on the fly.

5.2.2. Initialization

Initialization procedure ENC_Init() is described in 2.4.

```
// Initialize ENCx24J600 according to procedure provided in (1), chapter 8
// Assumes SPI interface on Port C, interrupt line connected to Pin 0. SPI should be initialized
// before calling ENC_Init (function SPIC_Init in SPI.c)
int8_t ENC_Init()
{
        // Wait for ENC SPI interface to initialize
        do
        {
                ENC_WCRU(EUDAST,0x1234);
        } while (ENC_RCRU(EUDAST) != 0x1234);

        // Wait for stable clock
        while (!(ENC_RCRU(ESTAT) & ENC_ESTAT_CLKRDY_bm));

        // Reset
        ENC_SETETHRST();
        _delay_us(50);

        // Check that EUDAST returned to default value
        if (ENC_RCRU(EUDAST) != 0x0000) return ERR;

        // wait at least 256 us for PHY initialization
        _delay_us(500);


        // Enable Ethernet, LED stretching, automatic MAC Address transmission, TX and RX logic
        ENC_WCRU(ECON2,0xe000);

        // Initialize 'NextPacketPointer' to ERXST
        NextPacketPointer = ENC_RCRU(ERXST);

        // Disable reception of broadcast (ff-ff-ff-ff-ff-ff) frames - only frames having correct MAC
        // address will be accepted
        ENC_BFCU(ERXFCON, ENC_ERXFCON_BCEN_bm);

        // Enable reception
```

```
        ENC_BFSU(ECON1, ENC_ECON1_RXEN_bm);

        // Interrupt control - PORT C, Pin 0
        PORTC.PIN0CTRL = PORT_OPC_PULLUP_gc | PORT_ISC_FALLING_gc;      // falling edge
        PORTC.INT0MASK = PIN0_bm;
        PORTC.INTCTRL = PORT_INT0LVL_MED_gc;                            // medium priority

        PMIC.CTRL |= PMIC_MEDLVLEN_bm;                          // enable medium level interrupts

        // Enable ENC interrupts
        ENC_SETEIE();

        return OK;
}
```

5.2.3. Reading UDP datagram

ENCx24J600 is configured to generate interrupt upon reception of incoming frame. Since INT line is connected to PORT C.0, interrupt service routine should be placed in ISR(PORTC_INT0_vect) function.

```
ISR(PORTC_INT0_vect)
{
        ENC_CLREIE();   // disable ENC interrupts (INT line goes inactive)

        // read packet
        if(ENC_RdUDPFrame(SourceAddr, DestAddr, &SourcePort, &DestPort, &Len, &data) == OK)
        {       // if it is correct UDP frame, send it back
                ENC_SendUDPFrame(uC_IPAddr, PC_IPAddr, PC_MACAddr, 11000, 11000, 0, Len, data);
                free(data);     // free allocated memory
        }

        ENC_SETEIE();   // enable ENC interrupts (if interrupt is pending INT line goes active again)
}
```

As advised in (1), c. 13, in interrupt service routine host controller should clear the Global Interrupt Enable bit, INTIE (EIE<15>). This causes the interrupt pin to return to the non-asserted (high) state. At the end of interrupt service routine host controller should set the INTIE bit to re-enable interrupts. If new interrupt request is pending, new falling edge will occur on the INT line, which will be recognized by the host controller interrupt system.

ENC_RdUDPFrame function is defined as follows:

```
int8_t ENC_RdUDPFrame(uint8_t *SourceAddr, uint8_t *DestAddr, uint16_t *SourcePort, uint16_t *DestPort,
        uint16_t *Len, uint8_t **Data)
```

Function parameters:

Name	Definition	Description
SourceAddr	uint8_t SourceAddr[4]	Source IP address. Each element contains one of four IP address segments.
DestAddr	uint8_t DestAddr[4]	Destination IP address.
SourcePort	uint16_t* SourcePort	Source port number.
DestPort	uint16_t* DestPort	Destination port number.
Len	uint16_t* Len	Number of Data bytes.
Data	uint8_t** Data	Address of pointer to Data array. Array is dynamically allocated, so it is necessary to call *free(Data)* after the data has been processed.

Function returns completion status as defined in ENCx24J600.h.

Function ENC_RdUDPFrame follows procedure defined in 2.5 and sequentially reads and parses Ethernet frame, IPv4 packet and UDP datagram as described in chapters 3.1, 3.2 and 3.3.

5.2.4. Preparing and sending UDP datagram

Before sending, datagram should be fully constructed inside the General Purpose Buffer. Function ENC_SendUDPFrame constructs Ethernet frame, IPv4 packet and UDP datagram as described in chapters 3.1, 3.2 and 3.3. It also calculates both IPv4 header checksum and UDP datagram checksum.

ENC_SendUDPFrame function is defined as follows:

```
void ENC_SendUDPFrame(uint8_t *SourceIPAddr, uint8_t *DestIPAddr, uint8_t *DestMACAddr,
        uint16_t SourcePort, uint16_t DestPort, uint16_t BuffAddr, uint16_t Len, uint8_t *data)
```

Function parameters:

Name	Definition	Description
SourceIPAddr	uint8_t SourceAddr[4]	Source IP address. Each element contains one of four IP address segments.
DestIPAddr	uint8_t DestAddr[4]	Destination IP address.
DestMACAddr	uint8_t DestMACAddr[6]	Destination MAC address.
SourcePort	uint16_t SourcePort	Source port number. Should be zero if not used.
DestPort	uint16_t DestPort	Destination port number.
BuffAddr	uint16_t	Start address in General Purpose Buffer from where the frame will be constructed(*).
Len	uint16_t Len	Number of Data bytes.
Data	uint8_t *Data	Address of Data array.

(*) Application should define at least two different areas in General Purpose Buffer, to be able to construct the next frame while ENCx24J600 is sending the previous one.

5.2.5. Checksum calculation

Although checksum field is defined as optional in RFC 768 Internet Standard, PC computer running operating system such as MS Windows will reject datagrams not having a valid checksum, so the ENC_SendUDPFrame function implements UDP checksum calculation. UDP checksum is not checked for incoming datagrams. Faulty frames will be rejected by ENCx24J600 and between PC computer and microcontroller on local network segment there is no other possible source of UDP data corruption.

UDP checksum is calculated for UDP pseudoheader, which in addition to the UDP header contains some fields from the IPv4 header, and the UDP data. Since UDP pseudoheader is not continuous in ENCx24J600 General Purpose Buffer, it's checksum is calculated in code. In parallel, UDP data checksum is calculated by ENCx24J600 DMA, which is capable of processing 100Mbytes/s. After completion of DMA checksum calculation partial checksums are unified and returned. Algorithm for checksum calculation is explained in detail in (10).

```
// Calculate UDP checksum
// UDP pseudoheader checksum is calculated inside function and data checksum is calculated using
ENCx24J600
// DMA controller. Algorithm is described in (1) and (2)
// Parameters:
//                      Header          - UDP pseudoheader
//                      HLen            - pseudoheader length
```

```
//                      DataStartAddr   - start address of data in general purpose buffer
//                      DLen            - data length in bytes
uint16_t GenerateUDPChecksum(uint8_t *Header, uint16_t HLen, uint16_t DataStartAddr, uint16_t DLen)
{
        volatile uint32_t sum = 0;
        uint8_t carry;
        volatile uint16_t headSum, dataSum, checksum;

        if (DLen > 0)   // if data field is empty skip calculation od data checksum
        {
                // Initialize DMA calculation of data checksum:
                // Set EDMAST to the start address
                ENC_WCRU(EDMAST, DataStartAddr);
                // Set EDMALEN to the length of the input data
                ENC_WCRU(EDMALEN, DLen);
                // Clear DMACPY (ECON1<4>) to prevent a copy operation.
                // Clear DMANOCS (ECON1<2>) to select a checksum calculation.
                // Clear DMACSSD (ECON1<3>) to use the default seed of 0000h.
                // Set DMAST to initiate the operation
                ENC_DMACKSUM();
        }

        // Calculate Header checksum. ENC simultaneously calculates data checksum.
        for(uint8_t i = 0; i < HLen; i+=2)
        {
                sum += ((uint16_t)(Header[i])<<8) + Header[i+1];
        }
        // add carry
        while (((carry = (sum & 0xffff0000) >> 16)) != 0)
        {
                sum &= 0x0000ffff;
                sum += carry;
        }
        headSum = sum;

        if (DLen > 0)
        {
                // Wait for ENC DMA to finish data checksum calculation
                while (ENC_RCRU(ECON1) & ENC_ECON1_DMAST_bm);

                // Read data checksum
                dataSum = ENC_RCRU(EDMACS);     // LO and HI byte swapped !?
                uint8_t lo = dataSum & 0x00ff;
                uint8_t hi = dataSum >> 8;
                dataSum = ~((lo<<8) + hi);
        }
        else dataSum = 0;

        // Combine header and data checksum
        sum = (uint32_t)headSum + dataSum;      // 32 bit sum, otherwise carry will be discarded
        // add carry
        while (((carry = (sum & 0xffff0000) >> 16)) != 0)
        {
                sum &= 0x0000ffff;
                sum += carry;
        }
        sum = ~sum;
        checksum = sum != 0 ? sum : 0xffff;     // positive zero should be converted to negative zero

        return checksum;
}
```

6. Sending and receiving UDP datagrams on PC computer

Sample application is developed under MS Windows operating system and .NET framework (version 4.5), using Visual Studio 2015 development environment and C# programming language. Application defines UDP datagrams destination and source as IPEndPoint objects. For the target microcontroller it contains IP address of 192.168.1.11 and Port 11000, and for receiving datagrams from any source

IP address is set to IPAddress.Any (defined as 0.0.0.0) and the port is also 11000. The main function starts function UDPListen, defined to execute asynchronously using thread from the thread pool, and then each second displays number of transmitted/received datagrams.

UDPListen initialize content of UDP datagram and in a loop sends the datagram and waits for the reception.

```
using System;
using System.Diagnostics;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

public class UDPListener
{
    private const int TotalPackets = 100000;

    private const int UDPPort = 11000;

    // sending to target microcontroller
    private static IPEndPoint microConEP = new IPEndPoint(IPAddress.Parse("192.168.1.11"), UDPPort);
    // receiveing UDP packet from any source
    private static IPEndPoint groupEP = new IPEndPoint(IPAddress.Any, UDPPort);

    private static int noPackets = 0;
    private static bool finished = false;


    private static void UDPListen()
    {
        Task.Run(async () =>
        {
            using (var udpClient = new UdpClient(UDPPort))
            {
                byte[] sendBytes = new byte[512];
                for (int i = 0; i < 512; i++) sendBytes[i] = (byte)(i % 256);

                // Send and receive 100,000 datagrams. If sent or received datagram is corrupted
                // application will lock.
                for (int i = 0; i < TotalPackets; i++)
                {   // send datagram and wait for reception
                    udpClient.Send(sendBytes, sendBytes.Length, microConEP);
                    UdpReceiveResult receivedResults = await udpClient.ReceiveAsync();

                    noPackets++;
                }

                finished = true;
            }
        });
    }



    public static void Main()
    {
        noPackets = 0;

        UDPListen();
        Console.WriteLine("Started ...");

        Stopwatch sw = new Stopwatch();

        sw.Start();
        while (!finished)
        {
            Thread.Sleep(1000);
            Console.WriteLine("Transceived: {0} datagrams", noPackets);
        }
```

```
            sw.Stop();
            Console.WriteLine("\nRound-trip time per datagram: {0:F3} ms",
                (double)sw.ElapsedMilliseconds / TotalPackets);
            Console.ReadLine();
        }
    }
```

The application calculates the average time needed for the datagram to reach the target microcontroller and return. In this case it is 2.6 milliseconds.

Total length of UDP datagram having 512 bytes payload is 554 bytes. Transferring 554 bytes on 100 Mbit/s link takes approximately 45µs. To transfer that data to / from microcontroller through the 8 Mbit/s SPI interface takes additional 554µs, which gives 600µs, or total round-trip time of 1.2ms. Difference to measured 2.6ms comes from the PC side, probably due to Windows OS latency.

```
Transceived: 97989 datagrams
Transceived: 98377 datagrams
Transceived: 98761 datagrams
Transceived: 99151 datagrams
Transceived: 99537 datagrams
Transceived: 99928 datagrams
Transceived: 100000 datagrams

Round-trip time per datagram: 2,621 ms
```

It is interesting to analyze the case where the microcontroller continuously sends UDP frames to the PC. Since the time to prepare and send the frame is 600µs for 512 bytes of useful data, maximal theoretical throughput is 853 Kbyte/s, or 8.5 Mbit/s. That utilizes the 100 Mbit/s link less than 10%, which leaves space to connect up to ten microcontroller modules communicating at maximum speed. There will be no lost datagrams at the PC side, because Ethernet controller has sufficient buffering. It is easy to test it by slightly modify the test applications at PC and microcontroller.

7. Conclusion

Described Ethernet communication subsystem is developed for the Potentiostat - Galvanostat - Supercapacitor/Battery Tester (14), which consists of eight independent microcontroller channels connected to the single board PC computer. The main reason for choosing Ethernet was to support future system upgradability, e.g. Ethernet will easily support modules with 1Msamples/s acquisition rates, which will be impossible for any other communication protocol.

Although system is developed for short-range closed-system communication, with minor modifications of provided code it is easy to develop application for sending data to the other continent. There is enough memory in a system (large ENC buffer and 16Kbyte XMEGA RAM) to implement complex protocols like ARP and TCP. Data security could be implemented through the ENCx24J600 encryption engines.

I would like to hear about the developments based on provided code and schematic, bugs you find, and questions you may have, so feel free to contact me.

8. Literature

1. *ENC424J600/624J600 Data Sheet - Stand-Alone 10/100 Ethernet Controller with SPI or Parallel Interface.* s.l. : Microchip, 2010.

2. Microchip's TCP/IP Stacks. *Microchip.* [Online] Microchip. [Cited: January 12, 2017.] http://www.microchip.com/SWLibraryWeb/product.aspx?product=TCPIPSTACK.

3. **Antonic, Davor.** ENCx24J600 Ethernet module. *Antonic.* [Online] January 19, 2017. http://antonic.hr/Projects/Potentiostat/Ethernet%20module/.

4. Serial Peripheral Interface Bus. *Wikipedia.* [Online] December 28, 2016. [Cited: January 11, 2017.] https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus.

5. User Datagram Protocol. *Wikipedia.* [Online] December 5, 2016. [Cited: December 23, 2016.] https://en.wikipedia.org/wiki/User_Datagram_Protocol.

6. Ethernet frame. *Wikipedia.* [Online] December 18, 2016. [Cited: December 23, 2016.] https://en.wikipedia.org/wiki/Ethernet_frame.

7. IPv4. *Wikipedia.* [Online] December 18, 2016. [Cited: December 23, 2016.] https://en.wikipedia.org/wiki/IPv4.

8. List of IP protocol numbers. *Wikipedia.* [Online] November 27, 2016. [Cited: December 24, 2016.] https://en.wikipedia.org/wiki/List_of_IP_protocol_numbers.

9. IPv4 header checksum. *Wikipedia.* [Online] March 23, 2016. [Cited: December 24, 2016.] https://en.wikipedia.org/wiki/IPv4_header_checksum.

10. **Braden, R., Borman, D. and Partridge, C.** RFC 1071 - Computing the Internet checksum. *faqs.org.* [Online] September 1988. http://www.faqs.org/rfcs/rfc1071.html.

11. Wireshark. [Online] [Cited: November 15, 2016.] https://www.wireshark.org/.

12. XMEGA-A3BU Xplained. *Atmel.* [Online] Microchip - Atmel. [Cited: January 15, 2017.] http://www.atmel.com/tools/XMEGA-A3BUXPLAINED.aspx.

13. KiCad EDA - A Cross Platform and Open Source Electronics Design Automation Suite. *KiCad EDA.* [Online] [Cited: January 15, 2017.]

14. **Antonic, Davor.** Potentiostat - Galvanostat - Supercapacitor/Battery Tester. *Antonic.* [Online] November 15, 2016. http://antonic.hr/Projects/Potentiostat/.

15. **Horvat, Goran, Sostaric, Damir and Balkic, Zoran.** Cost-effective Ethernet Communication for Low Cost Microcontroller Architecture. *International Journal of Electrical and Computer Engineering Systems.* 2012, Vol. 3, 1.