**W5100.h – работа с микросхемой на низком уровне(платформозависимый файл)**

\*Инициализация SPI происходит в функции sysInit() файла main.c.

*uint8\_t* SPItransfer(*uint8\_t* data) – передача байта по SPI (без выбора ведомого)

*uint8\_t* W5100write(*uint16\_t* \_addr, *uint8\_t* \_data) – запись байта данных в микрасхему по адресу \_addr

*uint8\_t* W5100read(*uint16\_t* \_addr) – чтение байта данных по адресу \_addr

*uint16\_t* W5100readBlock(*uint16\_t* \_addr, *uint8\_t* \*\_buf, *uint16\_t* \_len) – чтение блока данных длинной \_len в буффер \_buf. Начало данных в микросхеме находится в \_addr.

execCmdSn(*uint8\_t* s, *uint8\_t* \_cmd) – выполнить команду \_cmd для сокета s (функция блокирующая, но поскольку микросхема выполняет команды быстро, то ее иногда целесообразней использовать чем КА)

*uint8\_t* W5100\_SetMac(*uint16\_t* oct1, *uint16\_t* oct2, *uint16\_t* oct3, *uint16\_t* oct4, *uint16\_t* oct5, *uint16\_t* oct6) – записать Mac адрес в микросхему

*uint8\_t* W5100\_SetIP(*uint16\_t* oct1, *uint16\_t* oct2, *uint16\_t* oct3, *uint16\_t* oct4) – записать IP адрес в микросхему

*uint8\_t* W5100\_SetGateway(*uint16\_t* oct1, *uint16\_t* oct2, *uint16\_t* oct3, *uint16\_t* oct4) –записать адрес шлюза в микросхему

*uint8\_t* W5100\_SetMask(*uint16\_t* oct1, *uint16\_t* oct2, *uint16\_t* oct3, *uint16\_t* oct4) – записать маску подсети в микросхему

void W5100\_init() – инициализация микросхемы (ресет питания, аппаратный ресет, програмный ресет, задание размерв буфферов для чтения и записи, заполнение массивов базовых адресов)

Макросы

#define \_\_GP\_REGISTER8(name, address)

void write##name(uint8\_t \_data)

*uint8\_t* read##name()

#define \_\_GP\_REGISTER16(name, address)

void write##name(uint16\_t \_data)

*uint16\_t* read##name()

#define \_\_GP\_REGISTER\_N(name, address, size)

*uint16\_t* write##name(uint8\_t \*\_buff)

*uint16\_t* read##name(uint8\_t \*\_buff)

где name может быть MR, IR, IMR, RCR, RMSR, TMSR, PATR, PTIMER, PMAGIC) – для удобства работы с регистрами микросхемы

Работа с сокетами

*uint8\_t* readSn(*uint8\_t* \_s, *uint16\_t* \_addr);

*uint8\_t* writeSn(*uint8\_t* \_s, *uint16\_t* \_addr, *uint8\_t* \_data);

*uint16\_t* readSnBlock(*uint8\_t* \_s, *uint16\_t* \_addr, *uint8\_t* \*\_buf, *uint16\_t* len);

*uint16\_t* writeSnBlock(*uint8\_t* \_s, *uint16\_t* \_addr, *uint8\_t* \*\_buf, *uint16\_t* len);

-запись значений в регистры управления сокетами или в память буферов чтения/записи

Макросы:

#define \_\_SOCKET\_REGISTER8(name, address) \

static inline void write##name(uint8\_t \_s, *uint8\_t* \_data)

static inline *uint8\_t* read##name(uint8\_t \_s)

#define \_\_SOCKET\_REGISTER16(name, address)

static void write##name(uint8\_t \_s, *uint16\_t* \_data)

static *uint16\_t* read##name(uint8\_t \_s)

#define \_\_SOCKET\_REGISTER\_N(name, address, size)

static *uint16\_t* write##name(uint8\_t \_s, *uint8\_t* \*\_buff)

static *uint16\_t* read##name(uint8\_t \_s, *uint8\_t* \*\_buff)

где ##name может быть SnMR, SnCR, SnIR, SnPROTO – для удобства работы с регистрами сокетов

*uint8\_t* execCmdS0\_Automat(*uint8\_t* event,*uint8\_t* \_cmd) – автомат, реализующий отправку и проверку выполнения команды для сокета 0

*uint8\_t* execCmdS1\_Automat(*uint8\_t* event,*uint8\_t* \_cmd) – автомат, реализующий отправку и проверку выполнения команды для сокета 1

*uint8\_t* execCmdS2\_Automat(*uint8\_t* event,*uint8\_t* \_cmd) – автомат, реализующий отправку и проверку выполнения команды для сокета 2

void W5100\_writeS0DPORT(*uint16\_t* port) – запись порта, в который необходимо отправиить пакет из сокета 0

void W5100\_writeS0DIPR(*uint8\_t*\* addr) – запись адреса, по которому необходимо отправиить пакет из сокета 0

void W5100\_writeS1DPORT(*uint16\_t* port) – запись порта, в который необходимо отправиить пакет из сокета 1

void W5100\_writeS1DIPR(*uint8\_t*\* addr) – запись адреса, по которому необходимо отправиить пакет из сокета 1

void W5100\_writeS2DPORT(*uint16\_t* port) – запись порта, в который необходимо отправиить пакет из сокета 2

void W5100\_writeS2DIPR(*uint8\_t*\* addr) – запись адреса, по которому необходимо отправиить пакет из сокета 2

void W5100\_send\_data\_processing\_offset(*uint8\_t* s, *uint16\_t* data\_offset, const *uint8\_t* \*data, *uint16\_t* len) – записывает данные из буфера data, длинной len в буффер передатчика сокета s микросхемы, учитывая сдвиг data\_offset.

*uint16\_t* bufferData(*uint8\_t* s, *uint16\_t* offset, const *uint8\_t*\* buf, *uint16\_t* len) – обертка над void W5100\_send\_data\_processing\_offset(*uint8\_t* s, *uint16\_t* data\_offset, const *uint8\_t* \*data, *uint16\_t* len), которая не дает записать в буффер больше, чем в нем свободного места на данный момент

**udp\_automat.h –работа с UDP**

*uint16\_t* \_remaining[4] – масив, в котором указывается количество не вычитанных байт из сокета

*uint8\_t* \_remoteIP[4] – масив адресов отправителей последнего сообщения

!нужно доработать для каждого сокета???

*uint16\_t* \_remotePort[4] - масив портов отправителей последнего принятого сообщения

*uint16\_t* \_offset, \_offset1, \_offset2 – текущее значение смещения для каждого сокета

*uint8\_t* UDPbegin0\_Automat(*uint8\_t* event,*uint16\_t* port) – КА, реализующий неблокирующее открытие UDP сервера на аппаратном сокете 0

*uint8\_t* UDPbegin1\_Automat(*uint8\_t* event,*uint16\_t* port) – КА, реализующий неблокирующее открытие UDP сервера на аппаратном сокете 1

*uint8\_t* UDPbegin2\_Automat(*uint8\_t* event,*uint16\_t* port) – КА, реализующий неблокирующее открытие UDP сервера на аппаратном сокете 2

int startUDP(*uint8\_t* s, *uint8\_t*\* addr, *uint16\_t* port) – записывает порт и адрес адресата сообщения в соответствующие регистры сокета

int \_dhcpUdpSocket\_beginPacket(*uint8\_t*\* ip, *uint16\_t* port) – начало сообщения от сокета 0

(установка текущего смещения в 0, запись адреса и порта адресата в соответствующие регистры)

int \_UDPtoSerialSocket\_beginPacket(*uint8\_t*\* ip, *uint16\_t* port)– начало сообщения от сокета 1

(установка текущего смещения в 0, запись адреса и порта адресата в соответствующие регистры)

int \_Socket2\_beginPacket(*uint8\_t*\* ip, *uint16\_t* port) – начало сообщения от сокета 1

(установка текущего смещения в 0, запись адреса и порта адресата в соответствующие регистры)

*uint16\_t* \_dhcpUdpSocket\_write(const *uint8\_t* \*buffer, *uint16\_t* size) – запись данных размером size из массива buffer в память буфера отправки сокета 0 микросхемы

*uint16\_t* \_UDPtoSerialSocket\_write(const *uint8\_t* \*buffer, *uint16\_t* size) – запись данных размером size из массива buffer в память буфера отправки сокета 1 микросхемы

*uint16\_t* \_Socket2\_write(const *uint8\_t* \*buffer, *uint16\_t* size) ) – запись данных размером size из массива buffer в память буфера отправки сокета 2 микросхемы

*uint8\_t* \_UDPtoSerialSocket\_endPacket\_Automat(*uint8\_t* event) – автомат, который выполняет отправку сообщения сокетом 1

*uint8\_t* \_dhcpUdpSocket\_endPacket\_Automat(*uint8\_t* event) – автомат, который выполняет отправку сообщения сокетом 0

*uint8\_t* \_Socket2\_endPacket\_Automat(*uint8\_t* event) - автомат, который выполняет отправку сообщения сокетом 2

*uint8\_t* Socket2\_sendmessage\_Automat(*uint8\_t* event, *uint8\_t*\* remote\_ip, *uint16\_t* remote\_port, *uint8\_t*\* data,*uint16\_t* \_size) – автомат, который посылает сообщение длинной size из буффера data на remote\_ip, remote\_port.

void read\_data(*uint8\_t* s, volatile *uint16\_t* src, volatile *uint8\_t* \*dst, *uint16\_t* len) – читает данные длинной len из буфера приема сокета s микросхемы начиная с src, в буфер dst

void W5100\_recv\_data\_processing(*uint8\_t* s, *uint8\_t* \*data, *uint16\_t* len) - читает данные длинной len из буфера приема сокета s микросхемы начиная с значения в Socket RX Read Pointer, в буфер dst, в конце устанавливает новое значение

*uint16\_t* W5100\_getRXReceivedSize(*uint8\_t* s) – проверить количество принятых байт

*int16\_t* recv(*uint8\_t* s, *uint8\_t* \*buf, *int16\_t* len) – проверяет, есть ли чтото в буфере приема, если есть читает все и подает команду в сокет что считывание завершено с новым указателем на начало несчитанных данных

*int16\_t* recvAvailable(*uint8\_t* s) – возвращает количество байт в буфере приема

int UDP\_read(*uint8\_t* \_sock) – читает один байт из буфера приема и перемещает указатель

int UDP\_parsePacket(*uint8\_t* \_sock) - дочитываются данные прошлого сообщения(данные нигде не сохраняются), после этого проверяется, есть ли принятые данные, если есть то считывается адрес отправителя, порт, и количество байт в пакете

int \_dhcpUdpSocket\_parsePacket() - UDP\_parsePacket(*uint8\_t* \_sock) для сокет 0

int \_UDPtoSerialSocket\_parsePacket() - UDP\_parsePacket(*uint8\_t* \_sock) для сокет 1

int \_dhcpUdpSocket\_readBlock(unsigned char\* buffer, size\_t len) – читает первыей блок длинной len в buffer из памяти приема сокета 0, если len больше remaining, читает remaining

int \_SNMPSocket\_read(unsigned char\* buffer, size\_t len) ) – читает первыей блок длинной len в buffer из памяти приема сокета 1, если len больше remaining, читает remaining

*uint16\_t* \_dhcpUdpSocket\_remotePort() – возвращает порт приславшего последнее сообщение в сокет 0

*uint16\_t* \_snmpUdpSocket\_remotePort()– возвращает порт приславшего последнее сообщение в сокет 1

int \_dhcpUdpSocket\_read() – читает байт из памяти приема сокета 0

*uint16\_t* \_dhcpUdpSocket\_available() – проверяет, есть ли чтото в буфере приема сокета 0

*uint16\_t* \_SNMPUdpSocket\_available() – проверяет, есть ли чтото в буфере приема сокета 1

*uint8\_t* UDPtoSerial\_sendmessage\_Automat(*uint8\_t* event, *uint8\_t*\* remote\_ip, *uint16\_t* remote\_port, *uint8\_t*\* data,*uint16\_t* \_size) – автомат, который посылает сообщение длинной size из буффера data на remote\_ip, remote\_port.

**DHCP\_Automat.h – клиент DHCP**

void send\_DHCP\_MESSAGE(*uint8\_t* messageType, *uint16\_t* secondsElapsed) – заполняет буфер отправки сообщением DHCP, тип сообщения задается в messageType.

*uint8\_t* send\_DHCP\_MESSAGE\_Automat(*uint8\_t* event, *uint8\_t* messageType, *uint16\_t* secondsElapsed) – заполняет буфер отправки, отправляет сообщение

*uint8\_t* parseDHCPResponse(unsigned long responseTimeout, *uint32\_t*\* transactionId) – ожидает и разбирает ответ от сервера

responseTimeout – время ожидания ответа, *uint32\_t*\* transactionId указатель на переменную в которую ложится ID транзакции

long random1(long howbig), long random2(long howsmall, long howbig) – функции для генерации случайной ID транзакции.

*uint8\_t* DHCP\_automat(*uint8\_t* event) – автомат, который выполняет запрос IP адреса, парсит ответы, записывает адрес в регистры микросхемы

**SNMP\_Automat.h – агент SNMP**

-построена на основе Agentuino.h

Выполнена, в отличии от Agentuino.h в виде конечного автомата, что позволяет избавится от блокирования

typedef struct {

SNMP\_PDU\_TYPES type;

*int32\_t* version;

*int32\_t* requestId;

SNMP\_ERR\_CODES error;

*int32\_t* errorIndex;

SNMP\_OID OID;

SNMP\_VALUE VALUE;

}SNMP\_PDU; - структура для хранения данных сообщения SNMP

SNMP\_PDU\_TYPES type – тип сообщение (GET, SET, GETNext)

*int32\_t* version; - версия протокола

*int32\_t* requestId; - id запроса

SNMP\_ERR\_CODES error; - код аварии

*int32\_t* errorIndex; - индекс аварии

SNMP\_OID OID; - OID переменной

SNMP\_VALUE VALUE; - значение переменной

void SNMP\_OID\_toString(SNMP\_OID oid, char \*buffer) –преобразовывает OID в строку

void SNMP\_OID\_fromString(SNMP\_OID \*oid, const char \*buffer) – преобразовывает строку в OID

Функции декодеры (преобразовывают данные из сообщения в данные, которые можно присвоить переменным):

SNMP\_ERR\_CODES decodeOctetStringToString(SNMP\_VALUE \*snmp\_value, char \*value, size\_t max\_size) – преобразовывает строку октетов в массив char

SNMP\_ERR\_CODES decodeInt16toInt16(SNMP\_VALUE \*snmp\_value, *int16\_t* \*value) – Инт(snmp) в инт

SNMP\_ERR\_CODES decodeInt16toHoldReg(SNMP\_VALUE \*snmp\_value, *int16\_t* \*value) инт32 в инт16

SNMP\_ERR\_CODES decodeInt32ToInt32(SNMP\_VALUE \*snmp\_value, *int32\_t* \*value) инт32 в инт 32

SNMP\_ERR\_CODES decodeUint32CounterTimeticksGaugeToUint32(SNMP\_VALUE \*snmp\_value, *uint32\_t* \*value) инт32, тайм тики в инт 32

SNMP\_ERR\_CODES decodeIpAddressToByteArray(SNMP\_VALUE \*snmp\_value, byte \*value) – IP адрес в масив байтов

SNMP\_ERR\_CODES decodeBoolToUint8(SNMP\_VALUE \*snmp\_value, *uint8\_t* \*value) – бит в байт

Функции кодеры (преобразовывают данные из переменных, в данные протокола SNMP):

SNMP\_ERR\_CODES encodeStringToOctetString(SNMP\_VALUE \*snmp\_value, SNMP\_SYNTAXES syn, const char \*value) – строку в строку октетов

SNMP\_ERR\_CODES encodeInt16ToInt(SNMP\_VALUE \*snmp\_value, SNMP\_SYNTAXES syn, const *int16\_t* value) – инт16 в инт

SNMP\_ERR\_CODES encodeInt32ToInt32(SNMP\_VALUE \*snmp\_value, SNMP\_SYNTAXES syn, const *int32\_t* value) – инт32 в инт32

SNMP\_ERR\_CODES encodeUint32ToUint32(SNMP\_VALUE \*snmp\_value, SNMP\_SYNTAXES syn, const *uint32\_t* value) - encode's an uint32 to uint32, counter, time-ticks, gauge, opaque syntax

SNMP\_ERR\_CODES encodeByteArrayToIpAddress(SNMP\_VALUE \*snmp\_value, SNMP\_SYNTAXES syn, const byte \*value) - encode's an ip-address byte array to ip-address, NSAP-address, opaque syntax

SNMP\_ERR\_CODES encodeUint8ToBool(SNMP\_VALUE \*snmp\_value, SNMP\_SYNTAXES syn, const *uint8\_t* value) - encode's a boolean to boolean, opaque syntax

SNMP\_ERR\_CODES encodeUint64ToCounter64(SNMP\_VALUE \*snmp\_value, SNMP\_SYNTAXES syn, const *uint64\_t* value) - encode's an uint64 to counter64, opaque syntax

SNMP\_API\_STAT\_CODES SNMP\_begin() – настройка работы по SNMP

\_getCommName = "public";

\_setCommName = "public"; - пароли для community

typedef struct {

const char\* oid;

SNMP\_SYNTAXES syn;

void \* variable;

*uint8\_t* variableType;

*uint8\_t* AccessLevel;

}SNMP\_FIELD;

Структура, для хранения переменной, задаваемой по SNMP

SNMP\_FIELD SNMP\_VARS[] – массив структур всех переменных

unsigned int GetOIDCount(const char\* str) – возвращает количество чисел в OID 1.3.6.1.4.1.2017.13.16.0 – 10

unsigned int GetNVal\_FromOID\_String(unsigned int n, const char\* str) – возвращает значение n-го числа OID

int CompareOIDS(const char\* str1,const char\* str2) – сравнивает два OID кода и определяет который из них выше в лексикографической последовательности

SNMP\_ERR\_CODES Encode(void\* C\_value, *uint8\_t* C\_type, SNMP\_VALUE \*snmp\_value, SNMP\_SYNTAXES syn ) – функция обертка над кодерами, которая выбирает нужный кодер в зависимости от типа

SNMP\_ERR\_CODES Decode(SNMP\_VALUE \*snmp\_value, SNMP\_SYNTAXES syn, void\* C\_value, *uint8\_t* C\_type) – функция обертка над декодерами, которая выбирает нужный декодер в зависимости от типа

SNMP\_API\_STAT\_CODES SNMP\_requestPdu(SNMP\_PDU \*pdu) – функция анализа принятого сообщения, сообщение парсится и сохраняется в \*pdu

SNMP\_API\_STAT\_CODES SNMP\_responsePdu(SNMP\_PDU \*pdu) – функция, формирующая сообщение ответ на сообщение, которое находится в \*pdu

SNMP\_API\_STAT\_CODES SNMP\_responsePduGetBulk(SNMP\_PDU \*pdu) – (для SNMP>1)функция, формирующая ответ на запрос GET\_Bulk (нуждается в доработке)

void pduReceived() – функция, в которой обрабатывается принятое по udp сообщение, если оно оказывается корректным SNMP запросом то на него формируется ответ

*uint8\_t* SNMP\_Automat(*uint8\_t* event) – автомат, который реализует работу SNMP агента

*uint8\_t* SNMP\_VARS\_Processing(char\* oid, SNMP\_PDU\* pdu) – функция, которая сверяет OID переменной-привязки из запроса с таблицей переменных устройства, и передает его кодеру или декодеру

**Modbus\_TCP\_Automat.h –** Modbus\_TCP slave

#define MAX\_BUF 1024 – размер буфера в ОЗУ под сообщение (максимальный размер сообщения)

Конфигурация, которая указывает что слейв будет работать на сокете 3

#define TXBUFADDR 0x5800 // W5100 Send Buffer Base Address

#define RXBUFADDR 0x7800 // W5100 Read Buffer Base Address

#define RX\_BUF\_MASK 0x7FF

#define TX\_BUF\_MASK 0x7FF

#define WebserverSocket\_MR 0x0700 // Socket 0: Mode Register Address

#define WebserverSocket\_CR 0x0701 // Socket 0: Command Register Address

#define WebserverSocket\_IR 0x0702 // Socket 0: Interrupt Register Address

#define WebserverSocket\_SR 0x0703 // Socket 0: Status Register Address

#define WebserverSocket\_PORT 0x0704 // Socket 0: Source Port: 0x0404 to 0x0405

#define WebserverSocket\_TX\_FSR 0x0720 // Socket 0: Tx Free Size Register: 0x0420 to 0x0421

#define WebserverSocket\_TX\_RD 0x0722 // Socket 0: Tx Read Pointer Register: 0x0422 to 0x0423

#define WebserverSocket\_TX\_WR 0x0724 // Socket 0: Tx Write Pointer Register: 0x0424 to 0x0425

#define WebserverSocket\_RX\_RSR 0x0726 // Socket 0: Rx Received Size Pointer Register: 0x0425 to 0x0427

#define WebserverSocket\_RX\_RD 0x0728 // Socket 0: Rx Read Pointer: 0x0428 to 0x0429

#define WebserverSocketNum 3

// WebserverSocket\_SR values

#define SOCK\_CLOSED 0x00 // Closed

#define SOCK\_INIT 0x13 // Init state

#define SOCK\_LISTEN 0x14 // Listen state

#define SOCK\_SYNSENT 0x15 // Connection state

#define SOCK\_SYNRECV 0x16 // Connection state

#define SOCK\_ESTABLISHED 0x17 // Success to connect

#define SOCK\_FIN\_WAIT 0x18 // Closing state

#define SOCK\_CLOSING 0x1A // Closing state

#define SOCK\_TIME\_WAIT 0x1B // Closing state

#define SOCK\_CLOSE\_WAIT 0x1C // Closing state

#define SOCK\_LAST\_ACK 0x1D // Closing state

#define SOCK\_UDP 0x22 // UDP socket

#define SOCK\_IPRAW 0x32 // IP raw mode socket

#define SOCK\_MACRAW 0x42 // MAC raw mode socket

#define SOCK\_PPPOE 0x5F // PPPOE socket

// WebserverSocket\_MR values

#define MR\_CLOSE 0x00 // Unused socket

#define MR\_TCP 0x01 // TCP

#define MR\_UDP 0x02 // UDP

#define MR\_IPRAW 0x03 // IP LAYER RAW SOCK

#define MR\_MACRAW 0x04 // MAC LAYER RAW SOCK

#define MR\_PPPOE 0x05 // PPPoE

#define MR\_ND 0x20 // No Delayed Ack(TCP) flag

#define MR\_MULTI 0x80 // support multicating

// WebserverSocket\_CR values

#define CR\_OPEN 0x01 // Initialize or open socket

#define CR\_LISTEN 0x02 // Wait connection request in tcp mode(Server mode)

#define CR\_CONNECT 0x04 // Send connection request in tcp mode(Client mode)

#define CR\_DISCON 0x08 // Send closing reqeuset in tcp mode

#define CR\_CLOSE 0x10 // Close socket

#define CR\_SEND 0x20 // Update Tx memory pointer and send data

#define CR\_SEND\_MAC 0x21 // Send data with MAC address, so without ARP process

#define CR\_SEND\_KEEP 0x22 // Send keep alive message

#define CR\_RECV 0x40 // Update Rx memory buffer pointer and receive data

* Константы взятые из даташита

#define TCP\_PORT 502 – порт на котором будет работать сервер модбас

*uint16\_t* Webserver\_recv\_size(void) – возвращает размер принятого сообщения

void WebserverSocket\_close() – закрыть сокет

*uint8\_t* WebserverSocketOpen(*uint8\_t* sock,*uint8\_t* eth\_protocol,*uint16\_t* tcp\_port) – открыть сокет

eth\_protocol – протокол

tcp\_port – порт

*uint8\_t* Webserver\_listen() – перевести сокет в режим слушания

*uint16\_t* Webserver\_recv(*uint8\_t* \*buf,*uint16\_t* buflen) – считать данные из буфера микросхемы в ОЗУ

void WebserverSocket\_disconnect() – закрыть соединение

*uint16\_t* Webserver\_send(const *uint8\_t* \*buf,*uint16\_t* buflen) – послать данные (записать в буфер отправки и подать команду SEND)

enum MB\_FC {

MB\_FC\_NONE = 0,

MB\_FC\_READ\_COILS = 1,

MB\_FC\_READ\_DISCRETE\_INPUTS = 2,

MB\_FC\_READ\_REGISTERS = 3,

MB\_FC\_READ\_INPUT\_REGISTERS = 4,

MB\_FC\_WRITE\_COIL = 5,

MB\_FC\_WRITE\_REGISTER = 6,

MB\_FC\_WRITE\_MULTIPLE\_COILS = 15,

MB\_FC\_WRITE\_MULTIPLE\_REGISTERS = 16

}; - перечисление кодов функций

void SendErrorWrongAddress(*uint8\_t* \*buf) – функция, которая выполняет посылку сообщения о неверном адресе

void ModbusTCPRun(*uint8\_t* FC) – анализирует пакет и отправляет ответ

*uint8\_t* SetFC(int fc) – вытягивает из пакета номер функции

*uint8\_t* Modbus\_TCP\_Automat(*uint8\_t* event) – автомат, реализующий работу Modbus\_TCP