

# **CODESYS V2.3**

Программная реализация обмена через сокет

02.03.2020

Версия 1.0

## Оглавление

<b>1. Введение .....</b>	<b>4</b>
<b>2. Блокирующий и неблокирующий режимы .....</b>	<b>5</b>
<b>3. Описание библиотеки SysLibSockets.lib и её функций .....</b>	<b>5</b>
3.1. SysSockCreate .....	7
3.2. SysSockBind .....	8
3.3. SysSockListen .....	8
3.4. SysSockAccept .....	9
3.5. SysSockRecv и SysSockSend. Протокол TCP .....	10
3.6. SysSockRecvFrom и SysSockSendTo. Протокол UDP .....	11
3.7. SysSockShutdown .....	13
3.8. SysSockClose .....	13
3.9. SysSockSetOption .....	14
3.10. SysSockGetOption .....	15
3.11. SysSockHtonl .....	15
3.12. SysSockHtons .....	16
3.13. SysSockNtohl .....	16
3.14. SysSockNtohs .....	17
3.15. SysSockConnect .....	17
3.16. SysSockGetHostName .....	18
3.17. SysSockIoctl .....	19
3.18. SysSockInetAddr .....	19
3.19. SysSockInetNtoa .....	20
<b>4. TCP обмен .....</b>	<b>21</b>
4.1. Реализация TCP-сервера и TCP-клиента .....	21
4.2. Реализация TCP-сервера .....	22
4.2.1. Переменные PLC_PRG (TCP сервер) .....	22
4.2.2. Создание сокета SERVER_STATE_CREATE .....	23
4.2.3. Связка сокета с портом SERVER_STATE_BIND .....	24
4.2.4. Задание максимального количества соединений SERVER_STATE_LISTEN .....	24
4.2.5. Ожидание соединения с клиентом SERVER_STATE_ACCEPT .....	24
4.2.6. Закрытие серверного сокета SERVER_STATE_CLOSE .....	26
4.3. Описание ФБ fbClientSockets .....	26
4.3.1. Ожидание подключения CLIENT_STATE_IDLE .....	27
4.3.2. Чтение сообщения клиента CLIENT_STATE_READ .....	27
4.3.3. Ответ клиенту CLIENT_STATE_SEND .....	28

4.3.4.	Закрытие клиентского сокета CLIENT_STATE_CLOSE .....	29
<b>4.4.</b>	<b>Подпрограмма StopPrg</b> .....	<b>30</b>
<b>4.5.</b>	<b>Визуализация (TCP сервер)</b> .....	<b>30</b>
<b>4.6.</b>	<b>Реализация TCP-клиента</b> .....	<b>31</b>
4.6.1.	Переменные программы PLC_PRG (TCP клиент).....	31
4.6.2.	Ожидание команды для обмена с сервером CLIENT_STATE_IDLE.....	32
4.6.3.	Создание сокета CLIENT_STATE_CREATE.....	32
4.6.4.	Подключение к серверу CLIENT_STATE_CONNECT .....	33
4.6.5.	Отправка сообщения серверу CLIENT_STATE_SEND.....	33
4.6.6.	Получение сообщения от сервера CLIENT_STATE_READ .....	35
4.6.7.	Закрытие сокета CLIENT_STATE_CLOSE.....	35
<b>4.7.</b>	<b>Визуализация (TCP клиент)</b> .....	<b>36</b>
<b>5.</b>	<b>UDP обмен</b> .....	<b>37</b>
<b>5.1.</b>	<b>Реализация UDP-сервера и UDP-клиента</b> .....	<b>37</b>
<b>5.2.</b>	<b>Реализация UDP-сервера</b> .....	<b>38</b>
5.2.1.	Переменные PLC_PRG (UDP сервер).....	38
5.2.2.	Создание сокета SERVER_STATE_CREATE .....	39
5.2.3.	Связка сокета с портом SERVER_STATE_BIND.....	39
5.2.4.	Чтение сообщения клиента SERVER_STATE_READ.....	39
5.2.5.	Ответ клиенту SERVER_STATE_SEND .....	40
5.2.6.	Закрытие сокета SERVER_STATE_CLOSE .....	41
<b>5.3.</b>	<b>Подпрограмма StopPrg</b> .....	<b>41</b>
<b>5.4.</b>	<b>Визуализация (UDP сервер)</b> .....	<b>42</b>
<b>5.5.</b>	<b>Реализация UDP-клиента</b> .....	<b>42</b>
5.5.1.	Переменные программы PLC_PRG (UDP клиент) .....	43
5.5.2.	Ожидание команды для обмена с сервером CLIENT_STATE_IDLE.....	44
5.5.3.	Создание сокета CLIENT_STATE_CREATE.....	44
5.5.4.	Отправка сообщения серверу CLIENT_STATE_SEND.....	44
5.5.5.	Получение сообщения от сервера CLIENT_STATE_READ .....	46
5.5.6.	Закрытие сокета CLIENT_STATE_CLOSE.....	46
<b>5.6.</b>	<b>Визуализация (UDP клиент)</b> .....	<b>46</b>

## 1. Введение

Данный документ описывает программную реализацию передачи данных с помощью сетевых протоколов UDP и TCP для контроллеров ОВЕН, программируемых в среде CODESYS V2.3, с помощью библиотеки **SysLibSockets.lib**.

Обмен по протоколу TCP реализован в проектах:

1. TCP Сервер
2. TCP Клиент

Обмен по протоколу UDP реализован в проектах:

3. UDP Сервер
4. UDP Клиент

Контроллеры ОВЕН, программируемые в среде CODESYS V2.3 и имеющие интерфейс Ethernet, разделяются на 2 линейки: [ПЛК110/160 M02](#) и [ПЛК1xx](#). По умолчанию, в линейке ПЛК110/160 M02 сокет работает в блокирующем режиме, а в ПЛК1xx в неблокирующем, поэтому работа с библиотекой для ПЛК разных линеек отличается.

**Таблица 1.1 – Ключевые отличия линеек ПЛК110/160 M02 и ПЛК1xx**

Параметр	ПЛК110/160 M02	ПЛК1xx
Режим работы сокета по умолчанию	Блокирующий	Неблокирующий
Количество сокетов по TCP, не более	36	15
Количество сокетов по UDP, не более	36	4

## 2. Блокирующий и неблокирующий режимы

В блокирующем режиме все операции, производимые с сокетом, являются синхронными, а в неблокирующем – асинхронными. Например, в блокирующем режиме функция чтения завершает свою работу только после получения данных, что, соответственно, делает время цикла ПЛК непрогнозируемым. В неблокирующем режиме вызов любой функции не останавливает цикл ПЛК.

## 3. Описание библиотеки SysLibSockets.lib и её функций

Библиотека SysLibSockets.lib поддерживает работу с сокетами по TCP/IP и UDP.

Функции библиотеки и совместимость их для линеек контроллеров ПЛК110/160 M02 и ПЛК1xx представлены в табл.3.1.

Таблица 3.1 – Функции библиотеки SysLibSockets.lib

Функция	ПЛК110/160 M02	ПЛК1xx
<a href="#">SysSockCreate</a>	✓	✓
<a href="#">SysSockBind</a>	✓	✓
<a href="#">SysSockListen</a>	✓	✓
<a href="#">SysSockAccept</a>	✓	
<a href="#">SysSockRecv</a> и <a href="#">SysSockSend (TCP)</a>	✓	✓
<a href="#">SysSockRecvFrom</a> и <a href="#">SysSockSendTo (UDP)</a>	✓	✓
<a href="#">SysSockShutdown</a>	✓	✓
<a href="#">SysSockClose</a>	✓	✓
<a href="#">SysSockSetOption</a>	✓	
<a href="#">SysSockGetOption</a>	✓	
<a href="#">SysSockHtonl</a>	✓	
<a href="#">SysSockHtons</a>	✓	
<a href="#">SysSockNtohl</a>	✓	
<a href="#">SysSockNtohs</a>	✓	
<a href="#">SysSockConnect</a>	✓	✓
<a href="#">SysSockGetHostName</a>	✓	
<a href="#">SysSockIoctl</a>	✓	
<a href="#">SysSockInetAddr</a>	✓	
<a href="#">SysSockInetNtoa</a>	✓	
<a href="#">SysSockGetHostByName</a>		
<a href="#">SysSockSelect</a>		
<a href="#">SysSockSetIPAddress</a>		
<a href="#">SysSockGetLastError</a>		

✓ – функция поддерживается

В библиотеке в списке глобальных переменных объявлены константы для входных аргументов функций (см. Рисунок 3.1).

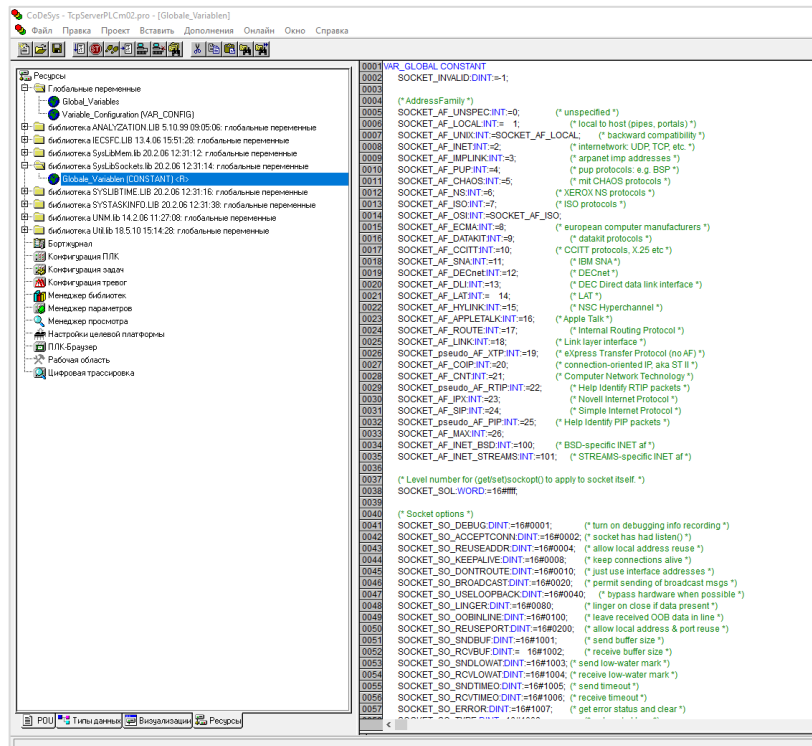


Рисунок 3.1 – Константы входных аргументов функций библиотеки SysLibSockets.lib

Часть функций библиотеки ссылаются на структуры (см. Рисунок 3.2).

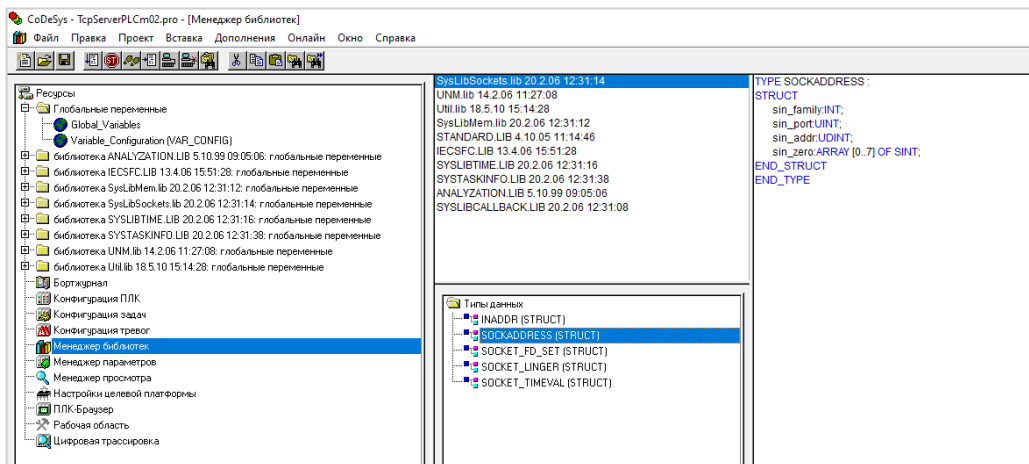


Рисунок 3.2 – Структуры библиотеки SysLibSockets.lib

Таблица 3.2 – Описание структур библиотеки SysLibSockets.lib

Структура	Описание
INADDR	Структура используется только в функции <a href="#">SysSockInetNtoa</a> .
SOCKADDRESS	Структура для хранения адреса. См. описание в <a href="#">табл.3.3</a>
SOCKET_FD_SET	Структура используется в функции <a href="#">SysSockSelect</a> , которая не поддерживается
SOCKET_LINGER	Используется в функции <a href="#">SysSockSetOption</a> для аргумента SO_LINGER.
SOCKET_TIMEVAL	Структура используется в функции <a href="#">SysSockSelect</a> , которая не поддерживается

Таблица 3.3 – Описание структуры SOCKADDRESS

Переменная	Тип	Описание
sin_family	INT	Семейство протоколов. Входные аргументы: - SOCKET_AF_INET (для семейства IPv4);
sin_port	UINT	Порт
sin_addr	UDINT	IP-адрес, к которому будет привязан сокет. Входные аргументы: - SOCKET_INADDR_ANY (все адреса локального хоста) – <a href="#">TCP и UDP сервер</a> (SysSockBind);  - 32-х битное значение IP-адреса – <a href="#">TCP клиент</a> (SysSockConnect) и <a href="#">UDP клиент</a> (SysSockRecvFrom/SysSockSendTo)
sin_zero	ARRAY [0..7] OF SINT	Дополнение до размера структуры SOCKADDRESS. Поле должно содержать массив нулей и служит только для увеличения размера структуры до стандартных 16 байт. Переменную не обязательно использовать в проекте.

Далее опишем все поддерживаемые функции в библиотеке.

### 3.1. SysSockCreate

Функция создает новый сокет и возвращает для него идентификатор (handle).

Переменная	Тип	Описание
<b>Входы</b>		
diAddressFamily	DINT	Семейство протоколов создаваемого сокета. Возможные значения: - SOCKET_AF_INET (для семейства IPv4);
diType	DINT	Определение типа создаваемого сокета. Возможные значения: - SOCK_STREAM (поточковый сокет для протокола TCP); - SOCK_DGRAM (датаграммный сокет для протокола UDP).
diProtocol	DINT	Протокол сокета. Возможные значения: - SOCKET_IPPROTO_TCP (для типа SOCK_STREAM); - SOCKET_IPPROTO_UDP (для типа SOCK_DGRAM).
<b>Выход</b>		
SysSockCreate	DINT	Идентификатор сокета

Пример:

```
// Объявление
```

```
diSocket: DINT; (*дескриптор сокета*)
```

```
// Код программы
```

```
diSocket := SysSockCreate(SOCKET_AF_INET, SOCK_STREAM, SOCKET_IPPROTO_TCP);
```

```
// Комментарии
```

Семейство IPv4, потоковый сокет, протокол TCP.

При попытке создать сокет в системе, где достигнуто максимальное число сокетов функция SysSockCreate вернёт -1 (SOCKET\_INVALID).

Максимальное количество сокетов для ПЛК110/160 M02 – 36 по TCP/UDP, для ПЛК1xx – 15 по TCP или 4 по UDP.

### 3.2. SysSockBind

Функция SysSockBind используется только в случае реализации на ПЛК сервера и привязывает сокет к IP адресу и порту. Для привязки функция SysSockBind ссылается на структуру [SOCKADDRESS](#), в которой хранится заданный адрес и порт.

Переменная	Тип	Описание
<b>Входы</b>		
diSocket	DINT	Дескриптор сокета, полученный от функции SysSockCreate
pSockAddr	DWORD	Указатель на переменную типа SOCKADDRESS
diSockAddrSize	DINT	Размер структуры SOCKADDRESS в байтах
<b>Выход</b>		
SysSockBind	BOOL	Результат функции

Пример для TCP/UDP сервера:

```
// Объявление
```

```
diSocket:          DINT; (*дескриптор сокета*)
```

```
stServerSettings: SOCKADDRESS; (*структура для хранения адреса сокета*)
```

```
xBinded:          BOOL; (*результат функции SysSockBind*)
```

```
wPort:           WORD; (*порт сокета*)
```

```
// Код программы
```

```
stServerSettings.sin_family := SOCKET_AF_INET;
```

```
stServerSettings.sin_addr := SysSockHtonl(SOCKET_INADDR_ANY);
```

```
stServerSettings.sin_port := SysSockHtons(wPort);
```

```
xBinded := SysSockBind(diSocket, ADR(stServerSettings), sizeof(stServerSettings) );
```

```
// Комментарии
```

Функция SysSockBind после выполнения возвращает TRUE. Для корректного заполнения структуры SOCKADDRESS для линейки ПЛК110/160 M02 следует использовать функции [SysSockHtonl](#) и [SysSockHtons](#).

#### **Внимание!**

Для ПЛК110/160 M02 каждый новый сокет должен иметь свой уникальный порт (502, 503 и т.д.).

### 3.3. SysSockListen

Функция SysSockListen задаёт максимальное количество входящих соединений. Функция используется при настройке обмена по протоколу TCP.

Переменная	Тип	Описание
<b>Входы</b>		
diSocket	DINT	Дескриптор сокета
diMaxConnections	DINT	Максимальное количество входящих соединений
<b>Выход</b>		
SysSockListen	BOOL	Результат функции



Пример:

```
// Объявление
```

```
diSocket:          DINT; (*дескриптор сокета*)
```

```
diMaxConnections: DINT:=5; (*максимальное количество входящих соединений*)
```

```
xListened:        BOOL; (*результат функции SysSockListen*)
```

```
// Код программы
```

```
xListened := SysSockListen(diSocket, diMaxConnections) );
```

```
// Комментарии
```

После выполнения возвращает TRUE – актуально только для линейки ПЛК110/160 M02. В линейке ПЛК1xx функция SysSockListen всегда возвращает FALSE.

#### Примечание:

В линейке ПЛК1xx для каждого клиента следует создать свой сокет с помощью функции [SysSockCreate](#). Максимальное количество сокетов для ПЛК1xx – 15 по TCP или 4 по UDP, а максимальное количество входящих соединений для одного сокета (diMaxConnections) – 1.

Для ПЛК110/160 M02 максимальное количество сокетов – 36 по TCP или UDP. По протоколу TCP в ПЛК110/160 M02 есть возможность создавать дескриптор клиентского сокета для установленного соединения с помощью функции [SysSockAccept](#), поэтому сокеты можно разделить на серверные и клиентские. Серверные сокеты создаются функцией SysSockCreate, а клиентские – SysSockAccept. Распределять 36 сокетов можно в разных пропорциях между серверными и клиентскими сокетами. Рассмотрим 2 примера по максимальному распределению сокетов для протокола TCP:

- Функцией SysSockCreate создан 1 сокет, который обрабатывает 35 входящих соединений (клиентских сокетов).
- Функцией SysSockCreate создано 6 сокетов, каждый сокет обрабатывает по 5 входящих соединений (клиентских сокетов).

### 3.4. SysSockAccept

При подключении клиента функция SysSockAccept создает дескриптор клиентского сокета для установленного соединения – актуально только для ПЛК110/160 M02. Функция используется при настройке обмена по протоколу TCP.

Для линейки ПЛК1xx соединение с клиентом определяется косвенным путем с помощью функции [SysSockRead](#), т.к. сокет по умолчанию в неблокирующем режиме.

Переменная	Тип	Описание
<b>Входы</b>		
diSocket	DINT	Дескриптор серверного сокета
pSockAddr	DWORD	Указатель на переменную типа <a href="#">SOCKADDRESS</a>
piSockAddrSize	DINT	Указатель на размер структуры SOCKADDRESS
<b>Выход</b>		
SysSockAccept	DINT	Дескриптор клиентского сокета для установленного соединения

Пример:

```
// Объявление
```

```
diSocket:          DINT; (*дескриптор серверного сокета*)
```

```

diClientSocket: DINT; (*дескриптор клиентского сокета*)
stSettings:     SOCKADDRESS; (*структура для хранения адреса сокета*)
// Код программы
diClientSocket := SysSockAccept(diSocket, ADR(stSettings), SIZEOF(stSettings) );

```

### 3.5. SysSockRecv и SysSockSend. Протокол TCP

Функция SysSockRecv выполняет прием данных и возвращает число считанных байт.

Максимальный буфер приема – 1500 байт.

Переменная	Тип	Описание
<b>Входы</b>		
diSocket	DINT	Дескриптор клиентского сокета
pbyBuffer	DWORD	Указатель на буфер принимаемых данных
diBufferSize	DINT	Размер буфера в байтах
diFlags	DINT	Флаг определяет параметры приема данных. Входные аргументы: 0 – не поддерживаются.
<b>Выход</b>		
SysSockRecv	DINT	Количество считанных байт

Пример:

```

// Объявление
diClientSocket: DINT; (*дескриптор клиентского сокета*)
abyRead:       ARRAY [1..c_iBufferSize] OF BYTE; (*переменная для приема сообщения*)
diRecvBytes:   DINT; (*количество считанных байт*)
c_diFlags:     DINT:=0; (*константа, флаг*)
c_iBufferSize: INT:=10; (*константа, размер буфера приёма*)
// Код программы
diRecvBytes := SysSockRecv(diClientSocket, ADR(abyRead), SIZEOF(abyRead), c_diFlags);
// Комментарии

```

Для приема можно использовать любые типы данных (например, переменные типа STRING).

Функция SysSockSend выполняет передачу данных и возвращает число переданных байт.

Максимальный буфер передачи – 1500 байт.

Переменная	Тип	Описание
<b>Входы</b>		
diSocket	DINT	Дескриптор клиентского сокета
pbyBuffer	DWORD	Указатель на буфер для передачи данных
diBufferSize	DINT	Размер буфера
diFlags	DINT	Флаг определяет параметры передачи данных. Входные аргументы: 0 – не поддерживаются.
<b>Выходы</b>		
SysSockSend	DINT	Количество переданных байт

Пример:

```
// Объявление
diClientSocket: DINT; (*дескриптор клиентского сокета*)
abySend: ARRAY [1..c_iBufferSize] OF BYTE; (*переменная для передачи сообщения*)
diSendBytes: DINT; (*количество переданных байт*)
c_diFlags: DINT:=0; (*константа, флаг*)
c_iBufferSize: INT:=10; (*константа, размер буфера передачи*)
```

```
// Код программы
```

```
diSendBytes := SysSockSend(diClientSocket, ADR(abySend), SIZEOF(abySend), c_diFlags);
```

```
// Комментарии
```

Для передачи можно использовать любые типы данных (например, переменные типа STRING).

### 3.6. SysSockRecvFrom и SysSockSendTo. Протокол UDP

Функция SysSockRecvFrom выполняет прием данных и возвращает число считанных байт.

Максимальный буфер приема – 1500 байт.

Переменная	Тип	Описание
<b>Входы</b>		
diSocket	DINT	Дескриптор клиентского сокета
pbyBuffer	DWORD	Указатель на буфер принимаемых данных
diBufferSize	DINT	Размер буфера
diFlags	DINT	Флаг определяет параметры приема данных. Входные аргументы: 0 – не поддерживаются.
pSockAddr	DWORD	Указатель на переменную типа SOCKADDRESS
diSockAddrSize	DINT	Размер структуры SOCKADDRESS
<b>Выход</b>		
SysSockRecvFrom	DINT	Количество считанных байт

Пример:

```
// Объявление
stClientSettings: SOCKADDRESS; (*структура для хранения адреса сокета*)
dwIPAddr: DWORD:=16#0A00060A; (*IP-адрес сервера: 10.0.6.10*)
wPort: WORD; (*порт сокета*)
diClientSocket: DINT; (*дескриптор клиентского сокета*)
abyRead: ARRAY [1..c_iBufferSize] OF BYTE; (*переменная для приема сообщения*)
diRecvBytes: DINT; (*количество считанных байт*)
c_diFlags: DINT:=0; (*константа, флаг*)
c_iBufferSize: INT:=10; (*константа, размер буфера приёма*)

// Код программы
stClientSettings.sin_family := SOCKET_AF_INET;
```

```
stClientSettings.sin_addr := SysSockHtonl(dwIPAddr);
```

```
stClientSettings.sin_port := SysSockHtons(wPort);
```

```
diRecvBytes := SysSockRecvFrom(diClientSocket, ADR(abyRead), SIZEOF(abyRead), c_diFlags,  
ADR(stClientSettings), SIZEOF(stClientSettings) );
```

```
// Комментарии
```

Для приема можно использовать любые типы данных (например, переменные типа STRING).

Функция SysSockSendTo выполняет передачу данных и возвращает число переданных байт. Максимальный буфер передачи – 1500 байт.

Переменная	Тип	Описание
<b>Входы</b>		
diSocket	DINT	Дескриптор клиентского сокета
pbyBuffer	DWORD	Указатель на буфер для передачи данных
diBufferSize	DINT	Размер буфера
diFlags	DINT	Флаг определяет параметры передачи данных. Входные аргументы: 0 – не поддерживаются.
pSockAddr	DWORD	Указатель на переменную типа SOCKADDRESS
diSockAddrSize	DINT	Размер структуры SOCKADDRESS
<b>Выход</b>		
SysSockSendTo	DINT	Количество переданных байт

Пример:

```
// Объявление
```

```
stClientSettings: SOCKADDRESS; (*структура для хранения адреса сокета*)
```

```
dwIPAddr: DWORD:=16#0A00060A; (*IP-адрес сервера: 10.0.6.10*)
```

```
wPort: WORD; (*порт сокета*)
```

```
diClientSocket: DINT; (*дескриптор клиентского сокета*)
```

```
abySend: ARRAY [1..c_iBufferSize] OF BYTE; (*переменная для передачи сообщения*)
```

```
diSendBytes: DINT; (*количество переданных байт*)
```

```
c_diFlags: DINT:=0; (*константа, флаг*)
```

```
c_iBufferSize: INT:=10; (*константа, размер буфера передачи*)
```

```
// Код программы
```

```
stClientSettings.sin_family := SOCKET_AF_INET;
```

```
stClientSettings.sin_addr := SysSockHtonl(dwIPAddr);
```

```
stClientSettings.sin_port := SysSockHtons(wPort);
```

```
diSendBytes := SysSockSend(diClientSocket, ADR(abySend), SIZEOF(abySend), c_diFlags,  
ADR(stSettings), SIZEOF(stSettings) );
```

```
// Комментарии
```

Для передачи можно использовать любые типы данных (например, переменные типа STRING).

### 3.7. SysSockShutdown

Функция SysSockShutdown выполняет запрет передачи и приема данных.

Переменная	Тип	Описание
<b>Входы</b>		
diSocket	DINT	Дескриптор сокета
diHow	DINT	Аргумент определяет тип запрещаемых действий: 0 – отключение приема сообщений; 1 – отключение отправки сообщения; 2 – отключение приема и отправки сообщения
<b>Выход</b>		
SysSockShutdown	BOOL	Результат функции

Пример:

```
// Объявление
```

```
diSocket: DINT; (*дескриптор сокета*)
```

```
xShutdowned: BOOL; (*результат функции SysSockShutdown *)
```

```
c_diHow: DINT:=2; (*константа, определяет тип запрещаемых действий*)
```

```
// Код программы
```

```
xShutdowned := SysSockShutdown(diSocket, c_diHow);
```

```
// Комментарии
```

После выполнения возвращает TRUE.

### 3.8. SysSockClose

Функция SysSockClose закрывает сокет.

Переменная	Тип	Описание
<b>Вход</b>		
diSocket	DINT	Дескриптор сокета
<b>Выход</b>		
SysSockClose	BOOL	Результат функции

Пример:

```
// Объявление
```

```
diSocket: DINT; (*дескриптор сокета*)
```

```
xSockClosed: BOOL; (*результат функции SysSockClose*)
```

```
// Код программы
```

```
xSockClosed := SysSockClose(diSocket);
```

```
// Комментарии
```

После выполнения возвращает TRUE. В линейке ПЛК1хх всегда возвращает FALSE.

### 3.9. SysSockSetOption

Функция SysSockSetOption устанавливает параметры, связанные с сокетом. Функция не поддерживается в линейке ПЛК1хх.

Переменная	Тип	Описание
<b>Входы</b>		
hSocket	DINT	Дескриптор сокета
diLevel	DINT	Уровень, на котором находится опция. Поддержан уровень для работы с сокетами: - SOL_SOCKET
diOption	DINT	Опция, которой нужно передать значение. Входные аргументы: SO_NBIO=0x1014 – установить сокет в неблокирующий режим SO_DEBUG=0x00001 – включить запись отладочной информации SO_REUSEADDR=0x00004 – разрешить повторное использование локального адреса SO_KEEPALIVE=0x00008 – поддерживать соединение SO_DONTROUTE=0x00010 – использовать адреса интерфейса SO_LINGER=0x00080 – задержка закрытия при наличии данных SO_WINSIZE=0x00400 – установить параметр окна масштабирования SO_TIMESTAMP=0x00800 – установить опцию отметки времени TCP SO_BROADCAST=0x01000 – большое начальное окно перегрузки TCP SO_NOBROADCAST=0x04000 – подавление медленного запуска в этом сокете
diOptionValue	DWORD	Указатель на переменную, в которую после выполнения функции будет записано значение для запрашиваемой опции
diOptionLength	DWORD	Размер в байтах переменной
<b>Выход</b>		
SysSockSetOption	BOOL	Результат функции

Пример:

// Объявление

diSocket: DINT; (\*дескриптор сокета\*)

c\_diOption: DINT:=16#1014; (\*константа, опция SO\_NBIO переводит в неблокирующий режим \*)

diOptionValue: DINT; (\*значение для запрашиваемой опции\*)

// Программа

xSetOption := SysSockSetOption(diSocket, SOCKET\_SOL, c\_diOption, ADR(diOptionValue),  
SIZEOF(diOptionValue));

// Комментарии

После выполнения возвращает TRUE. Детальное описание аргументов дано в справочных системах соответствующих ОС.

### 3.10. SysSockGetOption

Функция SysSockGetOption считывает значение параметра, связанного с сокетом. Функция не поддерживается в линейке ПЛК1хх.

Переменная	Тип	Описание
<b>Входы</b>		
diSocket	DINT	Дескриптор сокета
diLevel	DINT	Уровень, на котором находится опция. Входные аргументы: - SOL_SOCKET
diOption	DINT	Опция, которой нужно передать значение. Входные аргументы: SO_ERROR=0x1007 – получить статус ошибки SO_RXDATA=0x1011 – получить количество байт recv SO_TXDATA=0x1012 – получить количество байт send
diOptionValue	DWORD	Указатель на переменную, в которую после выполнения функции будет записано значение для запрашиваемой опции
diOptionLength	DWORD	Размер в байтах переменной
<b>Выход</b>		
SysSockGetOption	BOOL	Результат функции

Пример:

// Объявление

diSocket: DINT; (\*дескриптор сокета\*)

c\_diSoError: DINT:=16#1007; (\*константа, опция SO\_ERROR позволяет получить статус ошибки\*)

diOption: DINT; (\*после выполнения функции будет записано значение для запрашиваемой опции \*)

// Программа

xGetOption := SysSockGetOption(diSocket, SOCKET\_SOL, c\_diSoError, ADR(diOption),  
SIZEOF(diOption) );

// Комментарии

После выполнения возвращает TRUE.

### 3.11. SysSockHtonl

Функция SysSockHtonl (Host to Network Long) конвертирует переменную типа DWORD в соответствии с порядком байт в сетях TCP/IP. В линейке ПЛК1хх функция не конвертирует байты и возвращает входное значение.

Переменная	Тип	Описание
<b>Вход</b>		
dwHost	DWORD	Значение для конвертирования
<b>Выход</b>		
SysSockHtonl	DWORD	Результат функции

Пример:

// Объявление

dwIPAddr: DWORD:= 16#0A021478; (\*IP-адрес формата DWORD (например: 16#0A021478)\*)

stSettings: [SOCKADDRESS](#); (\*структура для хранения адреса сокета\*)

// Код программы

```
stSettings.sin_addr := SysSockHtonl(dwIPAddr);
```

// Комментарии

После преобразования в переменную stSettings.sin\_addr запишется 16#7814020A.

В линейке ПЛК1хх конвертация в соответствии с порядком байт в сетях TCP/IP выполняется автоматически.

### 3.12. SysSockHtons

Функция SysSockHtons (Host to Network Short) конвертирует переменную типа WORD в соответствии с порядком байт в сетях TCP/IP. В линейке ПЛК1хх функция не конвертирует байты и возвращает входное значение.

Переменная	Тип	Описание
<b>Вход</b>		
wHost	WORD	Значение для конвертирования
<b>Выход</b>		
SysSockHtons	WORD	Результат функции

Пример:

// Объявление

```
wPort: WORD:= 16#01F6; (*порт сокета, например, 502*)
```

stSettings: [SOCKADDRESS](#); (\*структура для хранения адреса сокета\*)

// Код программы

```
stSettings.sin_port := SysSockHtons(wPort);
```

// Комментарии

После преобразования в переменную stSettings.sin\_port запишется 16#F601.

В линейке ПЛК1хх конвертация в соответствии с порядком байт в сетях TCP/IP выполняется автоматически.

### 3.13. SysSockNtohL

Функция SysSockNtohL (Network to Host Long) конвертирует из порядка байт в сетях TCP/IP в переменную типа DWORD. В линейке ПЛК1хх функция не конвертирует байты и возвращает входное значение.

Переменная	Тип	Описание
<b>Вход</b>		
dwNet	DWORD	Значение для конвертирования
<b>Выход</b>		
SysSockNtohL	DWORD	Результат функции

Пример:

// Объявление



```

dwIPAddr: DWORD; (*IP-адрес формата DWORD*)
stSettings: SOCKADDRESS; (*структура для хранения адреса сокета*)
// Код программы
dwIPAddr := SysSockNtohl(stSettings.sin_addr); (*допустим переменная равна 16#7814020A*)
// Комментарии
После преобразования в переменную dwIPAddr запишется 16#0A021478 (10.2.20.120).

```

### 3.14. SysSockNtohs

Функция SysSockNtohs (Network to Host Short) конвертирует из порядка байт в сетях TCP/IP в переменную типа WORD. В линейке ПЛК1xx функция не конвертирует байты и возвращает входное значение.

Переменная	Тип	Описание
<b>Вход</b>		
wNet	WORD	Значение для конвертирования
<b>Выход</b>		
SysSockNtohs	WORD	Результат функции

Пример:

```

// Объявление
wPort: WORD; (*порт сокета*)
stSettings: SOCKADDRESS; (*структура для хранения адреса сокета*)
// Код программы
wPort := SysSockNtohs(stSettings.sin_port); (*допустим переменная равна 16#F601*)
// Комментарии
После преобразования в переменную wPort запишется 16#01F6 (502).

```

### 3.15. SysSockConnect

Функция SysSockConnect выполняет подключение к серверному сокету по IP-адресу и порту.

Переменная	Тип	Описание
<b>Входы</b>		
diSocket	DINT	Дескриптор клиентского сокета
pSockAddr	DWORD	Указатель на переменную типа SOCKADDRESS
diSockAddrSize	DINT	Размер структуры SOCKADDRESS
<b>Выход</b>		
SysSockConnect	BOOL	Результат функции (всегда возвращает FALSE)

Пример:

```

// Объявление
stClientSettings: SOCKADDRESS; (*структура для хранения адреса сокета*)

```

```

dwIPAddr:    DWORD:=16#0A00060A; (*IP-адрес сервера: 10.0.6.10*)
wPort:      WORD; (*порт сокета*)
diSocket:   DINT; (*дескриптор сокета*)
xConnected:  BOOL; (*результат функции SysSockConnect*)

// Код программы
stClientSettings.sin_family := SOCKET_AF_INET;
stClientSettings.sin_addr := SysSockHtonl(dwIPAddr);
stClientSettings.sin_port := SysSockHtons(wPort);

xConnected := SysSockConnect(diSocket, ADR(stClientSettings), sizeof(stClientSettings) );

// Комментарии

```

В неблокирующем режиме факт установки соединения можно определить только косвенным путем, используя функции [SysSockSend](#) и [SysSockRecv](#). После выполнения функция SysSockConnect всегда возвращает FALSE.

### 3.16. SysSockGetHostName

Функция SysSockGetHostName считывает имя контроллера. Функция не поддерживается в линейке ПЛК1хх.

Переменная	Тип	Описание
<b>Входы</b>		
stHostName	STRING	Имя хоста машины
diNameLength	DINT	Максимальный размер имени хоста, в байтах
<b>Выход</b>		
SysSockGetHostName	BOOL	Результат функции

Пример:

```

// Объявление
xResultGetHostName:  BOOL; (*результат функции SysSockGetHostName*)
sNamePLC:            STRING; (*переменная для записи имени хоста машины*)

// Код программы
xResultGetHostName := SysSockGetHostName(ADR(sNamePLC), sizeof(sNamePLC) );

// Комментарии

```

После выполнения возвращает TRUE.

Например, при использовании ПЛК110-24.32.К-М M02 в переменную sNamePLC будет записано '110-32'.

### 3.17. SysSockIoctl

Функция SysSockIoctl поддерживает только команду SOCKET\_FIONREAD для контроллеров ПЛК110/160 M02. Эта команда позволяет собирать входящие сообщения в один буфер. Затем можно вывести весь собранный буфер через функцию приема [SysSockRecv/SysSockRecvFrom](#)

Переменная	Тип	Описание
<b>Входы</b>		
hSocket	DINT	Дескриптор сокета
diCommand	DINT	Команда, которую нужно применить к сокету. Допустимые команды: - SOCKET_FIONREAD (собирает входящие сообщения в один буфер)
piParameter	DWORD	Указатель на параметр, который указывает объем данных в буфере в байтах.
<b>Выход</b>		
SysSockIoctl	DINT	Результат функции

Пример:

```
// Объявление
```

```
diSocket: DINT; (*дескриптор сокета*)
```

```
diParameter: DINT; (*переменная содержит размер входящего сообщения в байтах*)
```

```
diReturn: DINT; (*при поступлении данных возвращает 1*)
```

```
// Код программы
```

```
diReturn := SysSockIoctl(diSocket, SOCKET_FIONREAD, ADR(diParameter) );
```

```
// Комментарии
```

Получить объем входных данных можно до функции приема сообщения [SysSockRecv/SysSockRecvFrom](#). После вызова функции [SysSockRecv/SysSockRecvFrom](#) переменная diParameter, в которой хранится количество накопленных байт, очищается и в буфер принимаемых данных записывается буфер накопившихся данных.

### 3.18. SysSockInetAddr

Функция SysSockInetAddr преобразует строку в формат типа DWORD. Функция не поддерживается в линейке ПЛК1xx.

Переменная	Тип	Описание
<b>Вход</b>		
sIPAddr	STRING	IP-адрес
<b>Выход</b>		
SysSockInetAddr	DWORD	Результат функции

Пример:

```
//Объявление
```

```
stSettings: SOCKADDRESS; (*структура типа SOCKADDRESS*)
```

```
sInetIP: STRING:= '10.0.6.11'; (*переменная для записи IP*)
```

```
dwResultInetAddr: DWORD; (*возвращает преобразованный IP-адрес типа STRING в формат типа DWORD*)
```

```
//Код программы
dwResultInetAddr:=SysSockInetAddr(sInetIP);
// Комментарии
В переменную dwResultInetAddr запишется 16#0A00060B
```

### 3.19. SysSockInetNtoa

Функция SysSockInetNtoa преобразует сетевой адрес Интернета IPv4 в строку. Функция не поддерживается в линейке ПЛК1хх.

Переменная	Тип	Описание
<b>Входы</b>		
pInAddr	INADDR	Указатель на структуру INADDR
sIPAddr	STRING	Интернет IP-адрес
dilPAddrSize	DINT	Размер IP-адреса
SysSockInetNtoa	BOOL	Результат функции в Codesys

Пример:

```
// Объявление
stSettings:          INADDR; (*структура типа INADDR*)
xResultInetNtoa:    BOOL; (*результат функции SysSockInetNtoa в Codesys*)
sIPInet:            STRING; (*переменная для получения IP*)
byNumberBytes:     BYTE; (*длина переменной sIPInet*)
dwIpInet:          DWORD := 16#0A00060B; (*переменная, содержащая адрес IPv4*)
pbyResultInetNtoa: POINTER TO BYTE; (*указатель на BYTE для получения корректного
результата функции SysSockInetNtoa*)
// Код программы
stSettings.S_addr := dwIpInet;
xResultInetNtoa := SysSockInetNtoa(stSettings, sIPInet, SIZEOF(sIPInet) ); (*в переменную sIPInet
запишется '10.0.6.11' *)
pbyResultInetNtoa:=ADR(xResultInetNtoa);
byNumberBytes:= pbyResultInetNtoa^;
```

// Комментарии

Реализация библиотеки и Codesys не совпадает. Функция SysSockInetNtoa в Codesys возвращает BOOL, а в библиотеке – BYTE. Для получения корректного результата на выходе функции следует преобразовать переменную типа BOOL (xResultInetNtoa) в переменную типа BYTE (byNumberBytes) через указатель.

## 4. TCP обмен

### 4.1. Реализация TCP-сервера и TCP-клиента

Примеры доступны для скачивания:

- [ПЛК110/160 M02 TCP](#)
- [ПЛК1xx TCP](#)

Протокол TCP является одним из основных протоколов Интернета. Он обеспечивает надежную, упорядоченную и проверенную передачу данных между клиентом и сервером.

Основные характеристики протокола TCP:

- *надежность* — TCP управляет подтверждением, повторной передачей и таймаутом сообщений. Производятся многочисленные попытки доставить сообщение. Если оно потеряется по пути, сервер вновь запросит потерянную часть. В TCP нет ни пропавших данных, ни (в случае многочисленных таймаутов) разорванных соединений;
- *упорядоченность* — если два сообщения отправлены последовательно, первое сообщение достигнет приложения-получателя первым. Если участки данных прибывают в неверном порядке, TCP отправляет неупорядоченные данные в буфер до тех пор, пока все данные не могут быть упорядочены и переданы приложению;
- *тяжеловесность* — TCP необходимо три пакета для установки сокет-соединения перед тем, как отправить данные. TCP следит за надежностью и перегрузками;
- *потокость* — данные читаются как поток байтов, не передается никаких особых обозначений для границ сообщения или сегментов.

На рис.4.1 представлена структурная схема взаимодействия TCP сервера и клиента.

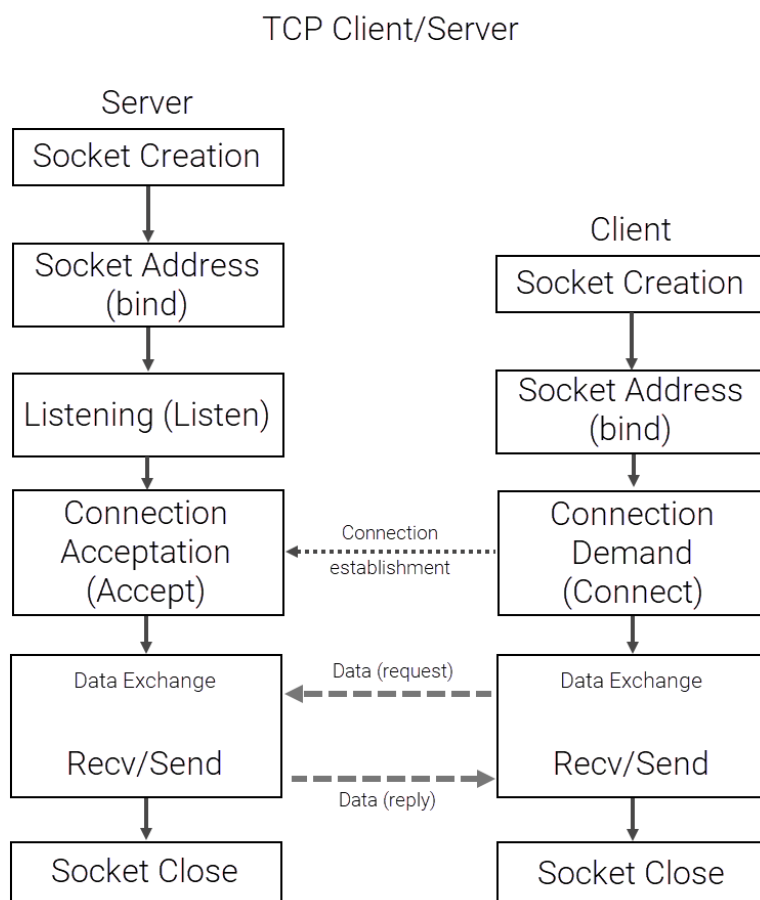


Рисунок 4.1 - Схема взаимодействия TCP сервера и клиента

## 4.2. Реализация TCP-сервера

Алгоритм работы сервера имеет следующие этапы:

1. Создание сокета (Socket Creation)
2. Связка сокета с портом (bind)
3. Включение прослушки (Listen)
4. Ожидание соединения с клиентом (Асепт)
5. Запрос/ответ (Recv/Send)
6. Закрытие сокета (Socket Close)

Данный алгоритм легко представить в виде последовательности шагов, выполняемых с помощью оператора CASE. Для удобства свяжем номера шагов с символьными именами через перечисления SERVER\_STATE и CLIENT\_STATE.

```
TYPE SERVER_STATE :
(
  SERVER_STATE_CREATE      := 0,      (*создание сокета*)
  SERVER_STATE_BIND        := 1,      (*связываем сокет с портом*)
  SERVER_STATE_LISTEN      := 2,      (*включаем прослушивание*)
  SERVER_STATE_ACCEPT      := 3,      (*ожидаем соединения от клиента*)
  SERVER_STATE_CLOSE       := 4       (*закрытие сокета*)
);
END_TYPE
```

Рисунок 4.2.1 - Перечисление SERVER\_STATE (PLC\_PRG)

```
TYPE CLIENT_STATE :
(
  CLIENT_STATE_IDLE        := 0,      (*состояние ожидания*)
  CLIENT_STATE_READ        := 1,      (*получаем запрос от клиента*)
  CLIENT_STATE_SEND        := 2,      (*отправляем ответ клиенту*)
  CLIENT_STATE_CLOSE       := 3       (*закрытие клиентского сокета*)
);
END_TYPE
```

Рисунок 4.2.2 - Перечисление CLIENT\_STATE (ФБ CLIENTSOCKETS)

Далее опишем каждый шаг в PLC\_PRG (п. 4.2) и функциональном блоке [ФБ CLIENTSOCKETS](#) (п. 4.3).

### 4.2.1. Переменные PLC\_PRG (TCP сервер)

На рис.4.2.3 представлены переменные, которые используются в программе PLC\_PRG.

```

PROGRAM PLC_PRG
VAR
    wPort:          WORD := 502;                (*порт сокета*)
    eState:         SERVER_STATE;              (*текущее состояние выполнения программы*)
    aeStatus:       ARRAY [1..c_iMaxConnections] OF SOCKET_STATUS; (*состояние сокета в символьном виде*)
    hServerSocket:  HANDLE;                    (*идентификатор серверного сокета*)
    hClientSocket:  HANDLE;                    (*идентификатор клиентского сокета*)

    ahClientSocket: ARRAY [1..c_iMaxConnections] OF HANDLE; (*идентификаторы клиентских сокетов*)
    iClientIterator: INT;                      (*счетчик подключаемых клиентов*)
    i:              INT := 1;                  (*счетчик переключения массива ФБ*)

    stServerSettings: SOCKET_ADDRESS;          (*структура для хранения адреса сокета*)
    xBinded:          BOOL;                    (*результат функций SysSockBind*)
    xListened:       BOOL;                    (*результат функций SysSockListen*)

    fbTon:          TON;                       (*таймер задержки включения*)
    tSockClose:     TIME := T#2m;             (*время задержки закрытия серверного сокета*)

    fbClientSockets: ARRAY [1..c_iMaxConnections] OF CLIENT_SOCKETS; (*массив ФБ для обмена с клиентами*)
    iCurrentConnections: INT;                 (*количество подключенных клиентов*)

    asReadFromClient: ARRAY [1..c_iMaxConnections] OF STRING(10); (*прием сообщения от клиента*)
    asSendToClient:   ARRAY [1..c_iMaxConnections] OF STRING(10); (*отправка сообщения клиенту*)

    diOption:       DINT := 0;                (*значение для запрашиваемой опции*)
END_VAR

VAR CONSTANT
    c_iMaxConnections: INT := 3;              (*максимальное количество клиентов (максимально возможное - 36)*)

    (*аргументы сокета*)
    c_diSoNbio:     DINT := 16#1014;          (*параметр SO_NBIО переводит в неблокирующий режим*)
    c_diHow:        DINT := 2;                (*тип запрещаемых действий*)
END_VAR

```

Рисунок 4.2.3 – Переменные программы PLC\_PRG для TCP сервера

#### 4.2.2. Создание сокета SERVER\_STATE\_CREATE

При старте программы происходит инициализация сервера. С помощью функции [SysSockCreate](#) создается сокет и возвращается для него системный идентификатор (handle). Данная функция в качестве входных параметров принимает аргументы, задающие тип и протокол сокета.

```

SERVER_STATE_CREATE: (*создаем сокет*)

    hServerSocket := SysSockCreate(SOCKET_AF_INET, SOCKET_STREAM, SOCKET_IPPROTO_TCP);

    IF hServerSocket <> SOCKET_INVALID THEN

        SysSockSetOption(hServerSocket, SOCKET_SOL, c_diSoNbio, ADR(diOption), SIZEOF(diOption)); (*перевод в неблокирующий режим*)

        eState := SERVER_STATE_BIND;
    ELSE
        eState := SERVER_STATE_CLOSE; (*если сокет не создался, то переходим на закрытие и пробуем снова*)
    END_IF

```

Рисунок 4.2.4 - Создание сокета SERVER\_STATE\_CREATE

Линейку контроллеров ПЛК1хх М02 необходимо переводить в [неблокирующий режим](#) с помощью функции [SysSockSetOption](#) для того, чтобы при вызове любой функции контроль над программой сразу возвращался.

**Внимание!** Контроллеры ПЛК1хх переводить в неблокирующий режим не нужно, так как они по умолчанию настроены на этот режим.

### 4.2.3. Связка сокета с портом SERVER\_STATE\_BIND

Сокет сервера привязывается к определенному IP-адресу и порту с помощью функции [SysSockBind](#). Для привязки к определенному IP-адресу функция SysSockBind ссылается на структуру SOCKADDRESS, в которой хранятся заданный адрес сокета для привязки.

```
SERVER_STATE_BIND:    (*связываем сокет с портом*)

(*определяем параметры*)
stServerSettings.sin_family := SOCKET_AF_INET;          (*тип сокета*)
stServerSettings.sin_addr  := SysSockHtonl(SOCKET_INADDR_ANY); (*принимать со всех адресов*)
stServerSettings.sin_port  := SysSockHtons(wPort);      (*используемый порт*)

xBinded := SysSockBind(hServerSocket, ADR(stServerSettings), sizeof(stServerSettings));

IF xBinded THEN
    eState := SERVER_STATE_LISTEN;
ELSE
    eState := SERVER_STATE_CLOSE;
END_IF
```

Рисунок 4.2.5 – Связка сокета с портом SERVER\_SOCK\_BIND

### 4.2.4. Задание максимального количества соединений SERVER\_STATE\_LISTEN

После привязки к адресу функция [SysSockListen](#) задает максимальное количество входящих соединений.

```
SERVER_STATE_LISTEN: (*включаем прослушивание входящих соединений*)

xListened := SysSockListen(hServerSocket, INT_TO_DINT(c_iMaxConnections));

IF xListened THEN
    eState := SERVER_STATE_ACCEPT;
ELSE
    eState := SERVER_STATE_CLOSE;
END_IF
```

Рисунок 4.2.6 – Связка сокета с портом SERVER\_SOCK\_BIND

#### Примечание:

В линейке ПЛК1хх функция SysSockListen всегда возвращает FALSE.

### 4.2.5. Ожидание соединения с клиентом SERVER\_STATE\_ACCEPT

После того как сервер включает режим прослушивания, он переходит в рабочий режим и ждет входящие соединения от клиентов. Как только клиент подключается к сокету сервера, с помощью функции [SysSockAccept](#) создается системный идентификатор клиентского сокета hClientSocket и соединение считается открытым.



```

SERVER_STATE_ACCEPT: (*ждедем соединения от клиента*)

hClientSocket := SysSockAccept(hServerSocket, ADR(stServerSettings), SIZEOF(stServerSettings));

IF hClientSocket <> SOCKET_INVALID THEN

    iClientIterator := 0;
    FOR iClientIterator := 1 TO c_iMaxConnections DO

        IF ahClientSocket[iClientIterator] = SOCKET_INVALID THEN

            ahClientSocket[iClientIterator] := hClientSocket;
            RETURN;
        ELSE
            ; (*все клиенты заняты*)
        END_IF

    END_FOR

END_IF

iCurrentConnections := 0;
FOR i := 1 TO c_iMaxConnections DO

    fbClientSockets[i]
    (
        hClientSocket := ahClientSocket[i],
        sSendToClient := asSendToClient[i],
        sReadFromClient => asReadFromClient[i],
        wStatus => aeStatus[i]
    );
    (*если входящих соединений нет, то запускаем таймер на закрытие серверного сокета*)
    IF ahClientSocket[i] = SOCKET_INVALID THEN
        fbTon(IN := TRUE, PT := tSockClose);

        IF fbTon.Q THEN
            eState := SERVER_STATE_CLOSE;
            fbTon(IN := FALSE);
        END_IF

    ELSE
        fbTon(IN := FALSE);
    END_IF

    IF ahClientSocket[i] <> SOCKET_INVALID THEN
        iCurrentConnections := iCurrentConnections + 1; (*определяем количество подключенных клиентов*)
    END_IF

END_FOR

```

**Рисунок 4.2.7 – Ожидание входящих соединений SERVER\_STATE\_ACCEPT**

С помощью цикла FOR и переменной iClientIterator выполняется распределение клиентов по свободным сокетами.

Для обмена с клиентами создан массив из ФБ [fbClientSockets](#).

В случае отсутствия входящих соединений по истечении времени tSockClose происходит переход на шаг SERVER\_STATE\_CLOSE, где выполняется закрытие серверного сокета.

В переменную iCurrentConnections записывается количество подключенных клиентов.

## 4.2.6. Закрытие серверного сокета SERVER\_STATE\_CLOSE

Перед закрытием сокета корректно выполнять запрет приема и передачи функцией [SysSockShutDown](#). Закрытие сокета выполняется с помощью функции [SysSockClose](#).

```
SERVER_STATE_CLOSE:    (*закрываем сокет*)

    SysSockShutDown(hServerSocket, c_diHow);    (*запрет приема и передачи сокета*)
    SysSockClose(hServerSocket);                (*закрытие сокета*)

    eState := SERVER_STATE_CREATE;
```

Рисунок 4.2.8 – Закрытие серверного сокета SERVER\_STATE\_CLOSE

## 4.3. Описание ФБ fbClientSockets

На рис. 4.3.1 представлены переменные, которые используются в функциональном блоке fbClientSockets. В функциональном блоке происходит обмен данными с подключенным клиентом.

В рамках данного примера для обмена используются переменные типа STRING, но в общем случае можно использовать любые типы данных (например, массив байт).

```
FUNCTION_BLOCK CLIENTSOCKETS

VAR_IN_OUT
    hClientSocket:    HANDLE;    (*идентификатор клиентского сокета*)
END_VAR

VAR_INPUT
    sSendToClient:    STRING(10);    (*переменная для отправки сообщения клиенту*)
END_VAR

VAR_OUTPUT
    sReadFromClient:  STRING(10);    (*переменная приема сообщения от клиента*)
    wStatus:          WORD;          (*переменная для передачи значение в eStatus*)
END_VAR

VAR
    eState:          CLIENT_STATE;    (*текущее состояние*)
    diRecvBytes:     DINT;            (*количество принятых байт*)
    diSendBytes:     DINT;            (*количество отправленных байт*)
    dwStatus:        DWORD;          (*переменная нужна для корректного получения состояния сокета*)
    fbTon:           TON;            (*таймер задержки включения*)
    tRecvClient:     TIME := T#20s;    (*время ожидания сообщения от клиента*)
    tSendClient:     TIME := T#2ms;    (*время передачи сообщения от клиенту*)
END_VAR

VAR CONSTANT
    (*аргументы сокета*)
    c_diFlags:       DINT := 0;        (*способ вызова функции*)
    c_diHow:         DINT := 2;        (*тип запрещаемых действий*)
    c_diSoError:     DINT := 16#1007; (*параметр SO_ERROR выводит статус ошибки*)
END_VAR
```

Рисунок 4.3.1 – Переменные для ФБ fbClientSockets

### 4.3.1. Ожидание подключения CLIENT\_STATE\_IDLE

Если системный идентификатор клиентского сокета создан, то переходим на чтение сообщения клиента.

```
CLIENT_STATE_IDLE:      (*ожидаем подключение*)

IF hClientSocket <> SOCKET_INVALID THEN
  eState := CLIENT_STATE_READ;
END_IF
```

Рисунок 4.3.2 – CLIENT\_STATE\_IDLE

### 4.3.2. Чтение сообщения клиента CLIENT\_STATE\_READ

Сервер принимает сообщение с помощью функции [SysSockRecv](#).

После получения сообщения выполняется переход к ответу на шаг CLIENT\_STATE\_SEND. Переменная tRecvClient – таймаут ожидания данных от клиента.

```
CLIENT_STATE_READ:      (*читаем сообщение клиента*)

diRecvBytes := SysSockRecv(hClientSocket, ADR(sReadFromClient), SIZEOF(sReadFromClient), c_diFlags);

IF diRecvBytes > 0 THEN (*если получили от клиента > 0 байт, готовим ответ*)
  eState := CLIENT_STATE_SEND;
  fbTon(IN := FALSE);
END_IF

fbTon(IN := TRUE, PT := tRecvClient);

IF fbTon.Q THEN          (*если запросов от клиентов не получаем, запускаем таймер*)
  eState := CLIENT_STATE_CLOSE;
  fbTon(IN := FALSE);
END_IF

SysSockGetOption(hClientSocket, SOCKET_SOL, c_diSoError, ADR(dwStatus), SIZEOF(dwStatus));
wStatus := DWORD_TO_WORD(dwStatus);
```

Рисунок 4.3.3 – Чтение сообщения клиента CLIENT\_STATE\_READ

На шаге приема сообщения добавлена функция [SysSockGetOption](#). Функция возвращает состояние клиентского сокета в переменную dwStatus. Для удобства числовые номера состояния свяжем с символьными именами с помощью перечисления SOCKET\_STATUS.

```

TYPE SOCKET_STATUS : (*статус сокета*)
(
  IP_ERR_EMPTY           := 16#0000,    (*Статус пуст*)
  IP_ERR_MISC           := 16#FFFF,    (*Разные ошибки, которые не имеют конкретный код ошибки*)
  IP_ERR_TIMEDOUT       := 16#FFFE,    (*Время операции истекло*)
  IP_ERR_ISCONN         := 16#FFFD,    (*Сокет уже подключен*)
  IP_ERR_OP_NOT_SUPP    := 16#FFFC,    (*Операция не поддерживается для выбранного сокета*)
  IP_ERR_CONN_ABORTED   := 16#FFFB,    (*Соединение было прервано*)
  IP_ERR_WOULD_BLOCK    := 16#FFFA,    (*Сокет находится в необлокирующем режиме*)
  IP_ERR_CONN_REFUSED   := 16#FFF9,    (*Соединение отклонено одноранговым узлом*)
  IP_ERR_CONN_RESET     := 16#FFF8,    (*Соединение было сброшено*)
  IP_ERR_NOT_CONN       := 16#FFF7,    (*Сокет не подключен*)
  IP_ERR_ALREADY        := 16#FFF6,    (*Сокет уже находится в запрошенном состоянии*)
  IP_ERR_IN_VAL         := 16#FFF5,    (*Переданное значение для конфигурации не действительно*)
  IP_ERR_MSG_SIZE       := 16#FFF4,    (*Сообщение слишком большое для отправки*)
  IP_ERR_PIPE           := 16#FFF3,    (*Сокет не находится в правильном состоянии для данной операции*)
  IP_ERR_DEST_ADDR_REQ := 16#FFF2,    (*Адрес не указан*)
  IP_ERR_SHUTDOWN       := 16#FFF1,    (*Соединение было закрыто как только все данные были получены после обнаружения запроса*)
  IP_ERR_NO_PROTO_OPT   := 16#FFF0,    (*Неизвестная опция сокета для setsockopt () или getsockopt ()*)
  IP_ERR_NO_MEM         := 16#FFEE,    (*Недостаточно памяти*)
  IP_ERR_ADDR_NOT_AVAIL := 16#FFED,    (*Неизвестный путь для отправки в указанный адрес*)
  IP_ERR_ADDR_IN_USE    := 16#FFEC,    (*Сокет уже соединен с адресом и портом*)
  IP_ERR_IN_PROGRESS    := 16#FFEA,    (*Операция все еще продолжается*)
  IP_ERR_NO_BUF         := 16#FFE9,    (*Внутренний буфер недоступен*)
  IP_ERR_NOT SOCK       := 16#FFE8,    (*Сокет не был открыт или уже был закрыт*)
  IP_ERR_FAULT          := 16#FFE7,    (*Общая ошибка при неудачной операции*)
  IP_ERR_NET_UNREACH    := 16#FFE6,    (*Нет доступа к нужной сети*)
  IP_ERR_PARAM          := 16#FFE5,    (*Неверный параметр для функции*)
  IP_ERR_LOGIC          := 16#FFE4,    (*Логическая ошибка, которая не должна была произойти*)
  IP_ERR_NOMEM          := 16#FFE3,    (*Системная ошибка: нет памяти для данной операции*)
  IP_ERR_NOBUFFER       := 16#FFE2,    (*Системная ошибка: нет внутреннего буфера для данной операции*)
  IP_ERR_RESOURCE       := 16#FFE1,    (*Системная ошибка: недостаточно свободных ресурсов*)
  IP_ERR_BAD_STATE      := 16#FFE0,    (*Сокет находится в неожиданном состоянии*)
  IP_ERR_TIMEOUT        := 16#FFDF,    (*Тайм-аут запрашиваемой операции*)
  IP_ERR_NO_ROUTE       := 16#FFDC,    (*Пункт назначения недоступен*)
  IP_ERR_TRIAL_LIMIT    := 16#FF80,    (*Пробный лимит превышен*)
);
END_TYPE

```

Рисунок 4.3.4 – Статусы сокета (опция SO\_ERROR)

### 4.3.3. Ответ клиенту CLIENT\_STATE\_SEND

Сервер передает сообщение с помощью функции [SysSockSend](#). Передаваемое сообщение пишем в переменную sSendToClient.

После передачи сообщения переходим обратно на чтение на шаг CLIENT\_STATE\_READ, иначе по истечении времени tSendClient выполняем закрытие клиентского сокета на шаге CLIENT\_STATE\_CLOSE.

Таймаут tSendClient может сработать в случае, если клиент закрыл свой сокет, не дожидаясь ответа. Также чем больше сообщение, тем больше время на передачу, поэтому переменную таймаута можно использовать для контроля ответа по времени.

```

CLIENT_STATE_SEND:    (*отправляем ответ*)

diSendBytes := SysSockSend(hClientSocket, ADR(sSendToClient), LEN(sSendToClient), c_diFlags);

IF diSendBytes > 0 THEN
  eState := CLIENT_STATE_READ;
  fbTon(IN := FALSE);
ELSE
  fbTon(IN := TRUE, PT := tSendClient);

  IF fbTon.Q THEN
    eState := CLIENT_STATE_CLOSE;
    fbTon(IN := FALSE);
  END_IF
END_IF

```

Рисунок 4.3.5 – Ответ клиенту CLIENT\_STATE\_SEND

Поскольку в рамках примера передается переменная типа STRING, то для определения размера пересылаемых данных используется оператор LEN.

#### 4.3.4. Закрытие клиентского сокета CLIENT\_STATE\_CLOSE

Для остановки приема и передачи используется функция [SysSockShutdown](#), а закрытие выполняется с помощью функции [SysSockClose](#).

Также на шаге закрытия клиентского сокета используется функция SysMemSet из библиотеки [SysLibMem.lib](#) для очищения сообщения клиента.

Если не делать чистку сообщения, то следующее сообщение клиента будет накладываться на предыдущее. Например, клиент прислал сообщение 'Hello world', потом прислал 'World'. В итоге получится сообщение 'World world'.

```

CLIENT_STATE_CLOSE:    (*закрываем клиентский сокет*)

SysSockShutdown(hClientSocket, c_diHow);    (*запрет приема и передачи сокета*)
SysSockClose(hClientSocket);                (*закрытие сокета*)

hClientSocket := SOCKET_INVALID;
wStatus      := 0;

SysMemSet(ADR(sReadFromClient), 0, SIZEOF(sReadFromClient)); (*очищаем сообщение клиента*)

eState := CLIENT_STATE_IDLE;

```

Рисунок 4.3.6 – Закрытие клиентского сокета CLIENT\_STATE\_CLOSE

#### 4.4. Подпрограмма StopPrg

В проекте добавлена подпрограмма StopPrg (рис.4.5.1) для закрытия созданных сокетов в случае сброса работы программы. Сброс программы может потребоваться, когда необходимо начать цикл заново не перезагружая контроллер.

Например, сбросить программу можно через CODESYS меню Вставка | Сброс холодный. Холодный сброс заново инициализирует все переменные, включая RETAIN.

```
(*в случае сброса работы программы выполняем закрытие серверного и клиентских сокетов*)
FOR k := 0 TO PLC_PRG.c_iMaxConnections DO
  SysSockShutdown(PLC_PRG.ahClientSocket[k], c_diHow);
  SysSockClose(PLC_PRG.ahClientSocket[k]);
END_FOR

SysSockShutdown(PLC_PRG.hServerSocket, c_diHow);
SysSockClose(PLC_PRG.hServerSocket);
```

Рисунок 4.4.1 – Подпрограмма StopPrg

Для привязки подпрограммы к системному событию нужно перейти в конфигурацию задач, Системные события, поставить галочку возле stop и выбрать подпрограмму (нажать F2 под вызываемым POU).

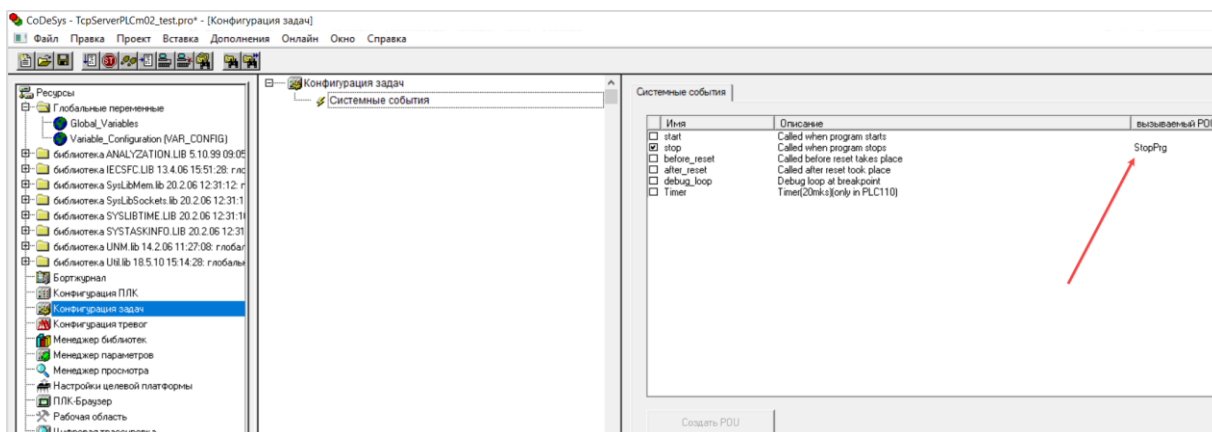


Рисунок 4.4.2 – Привязка подпрограммы StopPrg к проекту

#### 4.5. Визуализация (TCP сервер)

Проект включает в себя экран визуализации с названием Visu (рис.3.2.15). Она содержит параметры для отображения: порт, максимальное количество входящих соединений, количество подключенных клиентов и запрос клиента. Отправляемое сообщение клиенту можно ввести с экрана визуализации. В примере максимальное количество входящих соединений – 3, поэтому имеет три поля для отправки и получения сообщения.



Рисунок 4.5.1 – Внешний вид экрана визуализации

#### 4.6. Реализация TCP-клиента

Алгоритм работы клиента имеет следующие этапы:

1. Создание сокета (Socket Creation)
2. Подключение (Connect)
3. Запрос/ответ (Send/Recv)
4. Закрытие сокета (Socket Close)

Данный алгоритм легко представить в виде последовательности шагов, выполняемых с помощью оператора CASE. Для удобства свяжем номера шагов с символьными именами через перечисления CLIENT\_STATE.

```

TYPE CLIENT_STATE :
(
  CLIENT_STATE_IDLE      := 0,      (*ожидание*)
  CLIENT_STATE_CREATE    := 1,      (*создание сокета*)
  CLIENT_STATE_CONNECT   := 2,      (*подключаемся к серверному сокету*)
  CLIENT_STATE_SEND      := 3,      (*отправляем запрос серверу*)
  CLIENT_STATE_READ      := 4,      (*получаем ответ от сервера*)
  CLIENT_STATE_CLOSE     := 5,      (*закрытие сокета*)
);
END_TYPE

```

Рисунок 4.6.1 – Объявление перечисления CLIENT\_STATE

Далее опишем каждый шаг.

##### 4.6.1. Переменные программы PLC\_PRG (TCP клиент)

На рис. 4.6.2 представлены переменные, которые используются в программе.

В рамках данного примера для обмена используются переменные типа STRING, но в общем случае можно использовать любые типы данных (например, массив байт).



```

PROGRAM PLC_PRG
VAR
(*переменные задания*)
xStart:      BOOL;                (*запуск выполнения обмена с сервером*)
wPort:      WORD := 502;          (*порт сокета*)
sIPv4:      STRING(15) := '10.2.11.119'; (*IP сервера*)

(*внутренние переменные*)
eState:     CLIENT_STATE;        (*текущее состояние выполнения программы*)
eStatus:    SOCKET_STATUS;       (*состояние сокета в символьном виде*)
dwIPAddr:   DWORD;               (*IP сервера*)
dwStatus:   DWORD;               (*переменная нужна для корректного получения состояния сокета*)
hClientSocket: HANDLE;           (*идентификатор клиентского сокета*)
stClientSettings: SOCKADDRESS;   (*структура для хранения адреса сокета*)

(*переменные для отправки сообщения серверу*)
sSendToServer: STRING(10);       (*переменная для отправки сообщения серверу*)
diSendBytes: DINT;               (*количество отправленных байт*)

(*переменные для получения ответа от сервера*)
sReadFromServer: STRING(16);     (*переменная приема сообщения от сервера*)
diRecvBytes: DINT;               (*количество принятых байт*)

(*таймер задержки и уставки*)
fbTon:      TON;                 (*таймер задержки включения*)
tRecvServer: TIME := T#1s;       (*ожидание на прием сообщения от серверу*)
tSendServer: TIME := T#2ms;      (*ожидание на отравку сообщения серверу*)
tSockClose: TIME := T#2ms;       (*задержка между запретом/передачей и закрытием сокета*)

diOption:   DINT := 0;           (*значение для запрашиваемой опции*)
END_VAR

VAR CONSTANT
(*аргументы сокета*)
c_diFlags:  DINT := 0;           (*способ вызова функции*)
c_diSoNbio: DINT := 16#1014;     (*параметр SO_NBIO переводит в неблокирующий режим*)
c_diSoError: DINT := 16#1007;   (*параметр SO_ERROR выводит статус ошибки*)
c_diHow:    DINT := 2;           (*тип запрещаемых действий*)
END_VAR

```

Рисунок 4.6.2 – Переменные программы PLC\_PRG для TCP клиента

#### 4.6.2. Ожидание команды для обмена с сервером CLIENT\_STATE\_IDLE

Когда переменная xStart = TRUE выполняется однократный проход по всем этапам алгоритма работы клиента.

```

CLIENT_STATE_IDLE:
IF xStart THEN (*запуск выполнения обмена с сервером*)
    eState := CLIENT_STATE_CREATE;
    xStart := FALSE;
END_IF

```

Рисунок 4.6.3 – Шаг CLIENT\_STATE\_IDLE

#### 4.6.3. Создание сокета CLIENT\_STATE\_CREATE

С помощью функции [SysSockCreate](#) создается сокет и возвращается для него системный идентификатор (handle). Данная функция в качестве входных параметров принимает аргументы, задающие тип и протокол сокета.



```

CLIENT_STATE_CREATE:  (*создаем сокет*)

hClientSocket  := SysSockCreate(SOCKET_AF_INET, SOCKET_STREAM, SOCKET_IPPROTO_TCP);

IF hClientSocket <> SOCKET_INVALID THEN
  (*переводим в неблокирующий режим*)
  SysSockSetOption(hClientSocket, SOCKET_SOL, c_diSoNbio, ADR(c_diOption), SIZEOF(c_diOption) );

  eState := CLIENT_STATE_CONNECT;
ELSE
  eState := CLIENT_STATE_CLOSE;
END_IF

```

Рисунок 4.6.4 – Создание сокета CLIENT\_STATE\_CREATE

Линейку контроллеров ПЛК110/160 M02 необходимо переводить в [неблокирующий режим](#) с помощью функции [SysSockSetOption](#) для того, чтобы при вызове любой функции контроль над программой сразу возвращался.

**Внимание!** Контроллеры ПЛК1xx переводить в неблокирующий режим не нужно, так как они по умолчанию настроены на этот режим.

#### 4.6.4. Подключение к серверу CLIENT\_STATE\_CONNECT

Подключение к серверу выполняется с помощью функции [SysSockConnect](#). Функция ссылается на структуру [SOCKADDRESS](#), в которой хранится заданный адрес и порт сервера.

```

stClientSettings.sin_family := SOCKET_AF_INET;          (*тип сокета*)
stClientSettings.sin_port  := SysSockHtons(wPort);     (*порт*)

dwIPAddr := IP_DECODE(sIPV4);

stClientSettings.sin_addr  := SysSockHtonl(dwIPAddr);  (*IP сервера*)

SysSockConnect(hClientSocket, ADR(stClientSettings), SIZEOF(stClientSettings) );

eState := CLIENT_STATE_SEND;
(*в неблокирующем режиме факт установки соединения можно определить только косвенным путем,
используя функции SysSockSend и SysSockRecv*)

```

Рисунок 4.6.5 – Подключение к серверу CLIENT\_STATE\_CONNECT

#### 4.6.5. Отправка сообщения серверу CLIENT\_STATE\_SEND

Клиент передает сообщение с помощью функции [SysSockSend](#). Передаваемое сообщение необходимо записать в переменную sSendToServer.

Если сообщение передали (функция вернула количество переданных байт), то переходим к чтению на шаг CLIENT\_STATE\_READ. Иначе если клиент не смог отправить сообщение в течении времени tSendServer, то переходим к закрытию на шаг CLIENT\_STATE\_CLOSE.

Таймаут tSendServer может сработать в случае, если сервер закрыл свой сокет, не дожидаясь ответа. Также чем больше сообщение, тем больше времени на передачу, поэтому переменную таймаута можно использовать для контроля ответа по времени.

```

CLIENT_STATE_SEND:    (*отправляем сообщение серверу*)

    diSendBytes := SysSockSend(hClientSocket, ADR(sSendToServer), LEN(sSendToServer), c_diFlags);

    IF diSendBytes > 0 THEN
        eState := CLIENT_STATE_READ;
        fbTon(IN := FALSE);
    ELSE

        fbTon(IN := TRUE, PT := tSendServer);

        IF fbTon.Q THEN
            eState := CLIENT_STATE_CLOSE;
            fbTon(IN := FALSE);
        END_IF

    END_IF

    SysSockGetOption(hClientSocket, SOCKET_SOL, c_diSoError, ADR(dwStatus), SIZEOF(dwStatus) );
    eStatus := DWORD_TO_WORD(dwStatus); (*состояние сокета*)

```

Рисунок 4.6.6 – Отправка сообщения CLIENT\_STATE\_SEND

Поскольку в рамках примера передается переменная типа STRING, то для определения размера пересылаемых данных используется оператор LEN.

На шаге передачи сообщения добавлена функция [SysSockGetOption](#). Она возвращает состояние сокета в переменную типа DWORD dwStatus. Для удобства числовые номера состояния свяжем с символьными именами с помощью перечисления SOCKET\_STATUS.

```

TYPE SOCKET_STATUS : (*статус сокета*)
(
    IP_ERR_EMPTY           := 16#0000,    (*Статус пуст*)
    IP_ERR_MISC           := 16#FFFF,    (*Разные ошибки, которые не имеют конкретный код ошибки*)
    IP_ERR_TIMEDOUT       := 16#FFFE,    (*Время операции истекло*)
    IP_ERR_ISCONN         := 16#FFFD,    (*Сокет уже подключен*)
    IP_ERR_OP_NOT_SUPP    := 16#FFFC,    (*Операция не поддерживается для выбранного сокета*)
    IP_ERR_CONN_ABORTED  := 16#FFFB,    (*Соединение было прервано*)
    IP_ERR_WOULD_BLOCK   := 16#FFFA,    (*Сокет находится в необлокирующем режиме*)
    IP_ERR_CONN_REFUSED  := 16#FFF9,    (*Соединение отклонено одноранговым узлом*)
    IP_ERR_CONN_RESET    := 16#FFF8,    (*Соединение было сброшено*)
    IP_ERR_NOT_CONN      := 16#FFF7,    (*Сокет не подключен*)
    IP_ERR_ALREADY       := 16#FFF6,    (*Сокет уже находится в запрошенном состоянии*)
    IP_ERR_IN_VAL        := 16#FFF5,    (*Переданное значение для конфигурации не действительно*)
    IP_ERR_MSG_SIZE      := 16#FFF4,    (*Сообщение слишком большое для отправки*)
    IP_ERR_PIPE          := 16#FFF3,    (*Сокет не находится в правильном состоянии для данной операции*)
    IP_ERR_DEST_ADDR_REQ := 16#FFF2,    (*Адрес не указан*)
    IP_ERR_SHUTDOWN      := 16#FFF1,    (*Соединение было закрыто как только все данные были получены после обнаружения запроса*)
    IP_ERR_NO_PROTO_OPT  := 16#FFF0,    (*Неизвестная опция сокета для setsockopt () или getsockopt ()*)
    IP_ERR_NO_MEM        := 16#FFEE,    (*Недостаточно памяти*)
    IP_ERR_ADDR_NOT_AVAIL := 16#FFED,    (*Неизвестный путь для отправки в указанный адрес*)
    IP_ERR_ADDR_IN_USE   := 16#FFEC,    (*Сокет уже соединен с адресом и портом*)
    IP_ERR_IN_PROGRESS   := 16#FFEA,    (*Операция все еще продолжается*)
    IP_ERR_NO_BUF        := 16#FFE9,    (*Внутренний буфер недоступен*)
    IP_ERR_NOT_SOCKET    := 16#FFE8,    (*Сокет не был открыт или уже был закрыт*)
    IP_ERR_FAULT         := 16#FFE7,    (*Общая ошибка при неудачной операции*)
    IP_ERR_NET_UNREACH   := 16#FFE6,    (*Нет доступа к нужной сети*)
    IP_ERR_PARAM         := 16#FFE5,    (*Неверный параметр для функции*)
    IP_ERR_LOGIC         := 16#FFE4,    (*Логическая ошибка, которая не должна была произойти*)
    IP_ERR_NOMEM         := 16#FFE3,    (*Системная ошибка: нет памяти для данной операции*)
    IP_ERR_NOBUFFER      := 16#FFE2,    (*Системная ошибка: нет внутреннего буфера для данной операции*)
    IP_ERR_RESOURCE      := 16#FFE1,    (*Системная ошибка: недостаточно свободных ресурсов*)
    IP_ERR_BAD_STATE     := 16#FFE0,    (*Сокет находится в неожиданном состоянии*)
    IP_ERR_TIMEOUT       := 16#FFDF,    (*Тайм-аут запрашиваемой операции*)
    IP_ERR_NO_ROUTE      := 16#FFDC,    (*Пункт назначения недоступен*)
    IP_ERR_TRIAL_LIMIT   := 16#FFD0,    (*Пробный лимит превышен*)
);
END_TYPE

```

Рисунок 4.6.7 – Статусы сокета (опция SO\_ERROR)

#### 4.6.6. Получение сообщения от сервера CLIENT\_STATE\_READ

Клиент принимает сообщение с помощью функции [SysSockRecv](#). После шага CLIENT\_STATE\_READ переходим к закрытию клиентского сокета на шаг CLIENT\_STATE\_CLOSE.

```
CLIENT_STATE_READ:      (*получаем ответ от сервера*)

    diRecvBytes := SysSockRecv(hClientSocket, ADR(sReadFromServer), SIZEOF(sReadFromServer), c_diFlags);

    IF diRecvBytes > 0 THEN (*ответ получили, обмен прекращаем*)
        eState := CLIENT_STATE_CLOSE;
        fbTon(IN := FALSE);
    ELSE
        fbTon(IN := TRUE, PT := tRecvServer);

        IF fbTon.Q THEN
            eState := CLIENT_STATE_CLOSE;
            fbTon(IN := FALSE);
            (*ошибка: не удалось получить ответ*)
        END_IF
    END_IF
```

Рисунок 4.6.8 – Получение сообщения от сервера CLIENT\_STATE\_READ

#### 4.6.7. Закрытие сокета CLIENT\_STATE\_CLOSE

Перед закрытием сокета корректно выполнять запрет приема и передачи функцией [SysSockShutDown](#). Закрытие сокета выполняется с помощью функции [SysSockClose](#).

```
CLIENT_STATE_CLOSE:    (*закрываем сокет*)

    SysSockShutdown(hClientSocket, c_diHow);      (*запрет приема и передачи*)

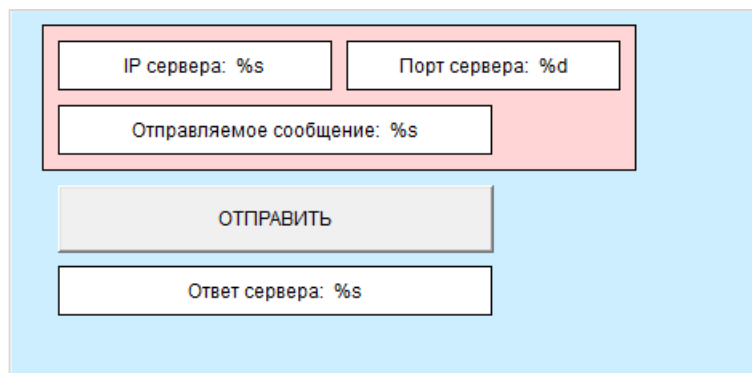
    fbTon(IN := TRUE, PT := tSockClose);        (*желательно разделять запрет приема/передачи и закрытие*)

    IF fbTon.Q THEN
        SysSockClose(hClientSocket);            (*закрытие сокета*)
        eState := CLIENT_STATE_IDLE;
        fbTon(IN := FALSE);
    END_IF
```

Рисунок 4.6.9 – Закрытие сокета CLIENT\_STATE\_CLOSE

#### 4.7. Визуализация (TCP клиент)

Проект включает в себя экран визуализации с названием Visu (рис.4.7.1). Она содержит параметры для записи: IP-адреса сервера, порта и отправляемое сообщение серверу. А также параметр чтения - ответ от сервера. Обмен с сервером выполняется по команде через кнопку ОТПРАВИТЬ. Обмен прекращается после получения ответа от сервера или по истечении таймаута tRecvServer.



The image shows a graphical user interface for a TCP client visualization. It consists of a light blue rectangular area containing several white rectangular boxes with black borders. At the top, there are three input fields: 'IP сервера: %s', 'Порт сервера: %d', and 'Отправляемое сообщение: %s'. Below these is a grey button labeled 'ОТПРАВИТЬ'. At the bottom, there is an output field labeled 'Ответ сервера: %s'.

Рисунок 4.7.1 – Внешний вид экрана визуализации

## 5. UDP обмен

### 5.1. Реализация UDP-сервера и UDP-клиента

Примеры доступны для скачивания:

- [ПЛК110/160 M02 UDP](#)
- [ПЛК1xx UDP](#)

UDP (User Datagram Protocol) – простой протокол транспортного уровня модели OSI, не подразумевающий установки выделенного соединения между сервером и клиентом. Связь достигается путём передачи информации в одном направлении от источника к получателю без проверки готовности получателя.

Основные характеристики протокола UDP:

- *ненадёжность* — когда сообщение посылается, неизвестно, достигнет ли оно точки назначения или потеряется по пути. Нет таких понятий, как подтверждение, повторная передача, таймаут;
- *неупорядоченность* — если два сообщения отправлены одному получателю, то порядок их достижения цели не может быть предугадан;
- *легковесность* — никакого упорядочивания сообщений, никакого отслеживания соединений и т. д. UDP – это небольшой транспортный уровень, разработанный на IP;
- *использование датаграмм* — пакеты посылаются по отдельности и проверяются на целостность только в том случае, если они прибыли. Пакеты имеют определенные границы, которые соблюдаются после получения, то есть операция чтения на сокете-получателе выдаст сообщение таким, каким оно было изначально послано;
- *отсутствие контроля перегрузок* — UDP сам по себе не избегает перегрузок. Для приложений с большой пропускной способностью возможно вызвать коллапс перегрузок, если только они не реализуют меры контроля на прикладном уровне.

На рис.5.1 представлена структурная схема взаимодействия UDP сервера и клиента.

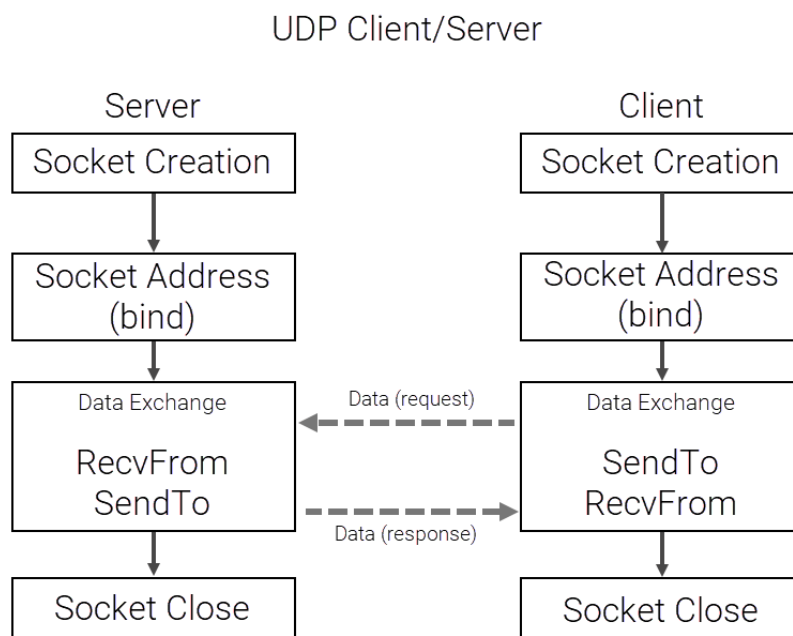


Рисунок 5.1 – Схема взаимодействия UDP сервера и клиента

## 5.2. Реализация UDP-сервера

Алгоритм работы сервера имеет следующие этапы:

1. Создание сокета (Socket Creation)
2. Связка сокета с портом (bind)
3. Запрос/ответ (RecvFrom/SendTo)
4. Закрытие сокета (Socket Close)

Данный алгоритм легко представить в виде последовательности шагов, выполняемых с помощью оператора CASE. Для удобства свяжем номера шагов с символьными именами через перечисления SERVER\_STATE.

```
TYPE SERVER_STATE :
(
    SERVER_STATE_CREATE      := 0,      (*создание сокета*)
    SERVER_STATE_BIND        := 1,      (*связываем сокет с портом*)
    SERVER_STATE_READ        := 2,      (*получаем запрос от клиента*)
    SERVER_STATE_SEND        := 3,      (*отправляем ответ клиенту*)
    SERVER_STATE_CLOSE       := 4,      (*закрытие сокета*)
);
END_TYPE
```

Рисунок 5.2.1 – Перечисление SERVER\_STATE

Далее опишем каждый шаг.

### 5.2.1. Переменные PLC\_PRG (UDP сервер)

На рис. 5.2.2 представлены переменные, которые используются в программе PLC\_PRG.

В рамках данного примера для обмена используются переменные типа STRING, но в общем случае можно использовать любые типы данных (например, массив байт).

```
PROGRAM PLC_PRG
VAR
    wPort:          WORD := 502;          (*порт сокета*)
    eState:         SERVER_STATE;        (*текущее состояние выполнения программы*)
    eStatus:        SOCKET_STATUS;       (*состояние сокета*)
    dwStatus:       DWORD;               (*переменная нужна для корректного получения состояния сокета*)
    hServerSocket:  HANDLE;              (*идентификатор серверного сокета*)
    stServerSettings: SOCKADDRESS;       (*структура для хранения адреса сокета*)
    xBinded:        BOOL;                (*результат функции SysSockBind*)

    (*переменные для получения ответа от клиента*)
    diRecvBytes:    DINT;                (*количество принятых байт*)
    sReadFromClient: STRING(10);        (*переменная приема сообщения от клиента*)

    (*переменные для отправки сообщения клиенту*)
    sSendToClient:  STRING(10);          (*переменная для отправки сообщения клиенту*)
    diSendBytes:    DINT;                (*количество отправленных байт*)

    (*таймер задержки и уставки*)
    fbTon:          TON;                 (*таймер задержки включения*)
    tRecvClient:    TIME := T#20s;       (*ожидание получения ответа от клиента*)
    tSendClient:    TIME := T#2ms;       (*время передачи сообщения от клиенту*)
    tSockClose:     TIME := T#2ms;       (*задержка между запретом/передачей и закрытием сокета*)

    diOption:       DINT := 0;           (*значение для запрашиваемой опции*)
END_VAR

VAR CONSTANT
    (*аргументы сокета*)
    c_diSoNbio:     DINT := 16#1014;     (*параметр SO_NBIO переводит в неблокирующий режим*)
    c_diSoError:    DINT := 16#1007;     (*параметр SO_ERROR выводит статус ошибки*)
    c_diHow:        DINT := 2;           (*тип запрещаемых действий*)
    c_diFlags:      DINT := 0;           (*способ вызова функции*)
END_VAR
```

Рисунок 5.2.2 – Переменные PLC\_PRG для UDP сервера

## 5.2.2. Создание сокета SERVER\_STATE\_CREATE

При старте программы происходит инициализация сервера. С помощью функции [SysSockCreate](#) создается сокет и возвращается для него системный идентификатор (handle). Данная функция в качестве входных параметров принимает аргументы, задающие тип и протокол сокета.

```
SERVER_STATE_CREATE:  (*создаем сокет*)

    hServerSocket := SysSockCreate(SOCKET_AF_INET, SOCKET_DGRAM, SOCKET_IPPROTO_UDP);

    IF hServerSocket <> SOCKET_INVALID THEN
        (*перевод в неблокирующий режим*)
        SysSockSetOption(hServerSocket, SOCKET_SOL, c_diSoNbio, ADR(diOption), SIZEOF(diOption) );

        eState := SERVER_STATE_BIND;
    ELSE
        eState := SERVER_STATE_CLOSE;
    END_IF
```

Рисунок 5.2.3 – Создание сокета SERVER\_STATE\_CREATE

Линейку контроллеров ПЛК1хх M02 необходимо переводить в [неблокирующий](#) режим с помощью функции [SysSockSetOption](#) для того, чтобы при вызове любой функции контроль над программой сразу возвращался.

**Внимание!** Контроллеры ПЛК1хх переводить в неблокирующий режим не нужно, так как они по умолчанию настроены на этот режим.

## 5.2.3. Связка сокета с портом SERVER\_STATE\_BIND

Сокет сервера привязывается к определенному IP-адресу и порту с помощью функции [SysSockBind](#). Для привязки к определенному IP-адресу функция SysSockBind ссылается на структуру SOCKADDRESS, в которой хранятся заданный адрес и порт сокета для привязки.

```
SERVER_STATE_BIND:  (*связываем сокет с портом*)

    (*определяем параметры*)
    stServerSettings.sin_family := SOCKET_AF_INET;                (*тип сокета*)
    stServerSettings.sin_addr  := SysSockHtonl(SOCKET_INADDR_ANY); (*принимать со всех адресов*)
    stServerSettings.sin_port   := SysSockHtons(wPort);           (*используемый порт*)

    xBinded := SysSockBind(hServerSocket, ADR(stServerSettings), SIZEOF(stServerSettings) );

    IF xBinded THEN
        eState := SERVER_STATE_READ;
    ELSE
        eState := SERVER_STATE_CLOSE;
    END_IF
```

Рисунок 5.2.4 – Связка сокета с портом SERVER\_STATE\_BIND

## 5.2.4. Чтение сообщения клиента SERVER\_STATE\_READ

Сервер принимает сообщение с помощью функции [SysSockRecvFrom](#).

После получения сообщения выполняется переход к ответу на шаг SERVER\_STATE\_SEND. Переменная tRecvClient – таймаут ожидания данных от клиента. По истечении таймаута сокет закрывается на шаге SERVER\_STATE\_CLOSE.



```

SERVER_STATE_READ: (*читаем сообщение клиента*)

diRecvBytes := SysSockRecvFrom(hServerSocket, ADR(sReadFromClient), SIZEOF(sReadFromClient), c_diFlags, ADR(stServerSettings), SIZEOF(stServerSettings));

IF diRecvBytes > 0 THEN (*если получили от клиента > 0 байт, то готовим ответ*)
    eState := SERVER_STATE_SEND;
    fbTon(IN := FALSE);
END_IF

fbTon(IN := TRUE, PT := tRecvClient); (*если запросов от клиентов не получаем, запускаем таймер*)

IF fbTon.Q THEN
    eState := SERVER_STATE_CLOSE;
    fbTon(IN := FALSE);
END_IF

SysSockGetOption(hServerSocket, SOCKET_SOL, c_diSoError, ADR(dwStatus), SIZEOF(dwStatus));
eStatus := DWORD_TO_WORD(dwStatus); (*состояние сокета*)

```

Рисунок 5.2.5 – Чтение сообщения клиента SERVER\_STATE\_READ

На шаге приема сообщения добавлена функция [SysSockGetOption](#). Функция возвращает состояние сокета в переменную dwStatus. Для удобства числовые номера состояния свяжем с символьными именами с помощью перечисления SOCKET\_STATUS.

```

TYPE SOCKET_STATUS : (*статус сокета*)
(
    IP_ERR_EMPTY           := 16#0000, (*Статус пуст*)
    IP_ERR_MISC           := 16#FFFF, (*Разные ошибки, которые не имеют конкретный код ошибки*)
    IP_ERR_TIMEDOUT       := 16#FFFE, (*Время операции истекло*)
    IP_ERR_ISCONN         := 16#FFFD, (*Сокет уже подключен*)
    IP_ERR_OP_NOT_SUPP    := 16#FFFC, (*Операция не поддерживается для выбранного сокета*)
    IP_ERR_CONN_ABORTED   := 16#FFFB, (*Соединение было прервано*)
    IP_ERR_WOULD_BLOCK    := 16#FFFA, (*Сокет находится в необлокирующем режиме*)
    IP_ERR_CONN_REFUSED   := 16#FFF9, (*Соединение отклонено одноранговым узлом*)
    IP_ERR_CONN_RESET     := 16#FFF8, (*Соединение было сброшено*)
    IP_ERR_NOT_CONN       := 16#FFF7, (*Сокет не подключен*)
    IP_ERR_ALREADY        := 16#FFF6, (*Сокет уже находится в запрошенном состоянии*)
    IP_ERR_IN_VAL         := 16#FFF5, (*Переданное значение для конфигурации не действительно*)
    IP_ERR_MSG_SIZE       := 16#FFF4, (*Сообщение слишком большое для отправки*)
    IP_ERR_PIPE           := 16#FFF3, (*Сокет не находится в правильном состоянии для данной операции*)
    IP_ERR_DEST_ADDR_REQ  := 16#FFF2, (*Адрес не указан*)
    IP_ERR_SHUTDOWN       := 16#FFF1, (*Соединение было закрыто как только все данные были получены после обнаружения запроса*)
    IP_ERR_NO_PROTO_OPT   := 16#FFF0, (*Неизвестная опция сокета для setsockopt () или getsockopt (*)*)
    IP_ERR_NO_MEM         := 16#FFEE, (*Недостаточно памяти*)
    IP_ERR_ADDR_NOT_AVAIL := 16#FFED, (*Неизвестный путь для отправки в указанный адрес*)
    IP_ERR_ADDR_IN_USE    := 16#FFEC, (*Сокет уже соединен с адресом и портом*)
    IP_ERR_IN_PROGRESS    := 16#FFEA, (*Операция все еще продолжается*)
    IP_ERR_NO_BUF         := 16#FFE9, (*Внутренний буфер недоступен*)
    IP_ERR_NOT_SOCKET     := 16#FFE8, (*Сокет не был открыт или уже был закрыт*)
    IP_ERR_FAULT          := 16#FFE7, (*Общая ошибка при неудачной операции*)
    IP_ERR_NET_UNREACH    := 16#FFE6, (*Нет доступа к нужной сети*)
    IP_ERR_PARAM          := 16#FFE5, (*Неверный параметр для функции*)
    IP_ERR_LOGIC          := 16#FFE4, (*Логическая ошибка, которая не должна была произойти*)
    IP_ERR_NOMEM          := 16#FFE3, (*Системная ошибка: нет памяти для данной операции*)
    IP_ERR_NOBUFFER       := 16#FFE2, (*Системная ошибка: нет внутреннего буфера для данной операции*)
    IP_ERR_RESOURCE       := 16#FFE1, (*Системная ошибка: недостаточно свободных ресурсов*)
    IP_ERR_BAD_STATE      := 16#FFE0, (*Сокет находится в неожиданном состоянии*)
    IP_ERR_TIMEOUT        := 16#FFDF, (*Тайм-аут запрашиваемой операции*)
    IP_ERR_NO_ROUTE       := 16#FFDC, (*Пункт назначения недоступен*)
    IP_ERR_TRIAL_LIMIT    := 16#FF80 (*Пробный лимит превышен*)
);
END_TYPE

```

Рисунок 5.2.6 – Статусы сокета (опция SO\_ERROR)

### 5.2.5. Ответ клиенту SERVER\_STATE\_SEND

Сервер передает сообщение с помощью функции [SysSockSendTo](#). Передаваемое сообщение пишем в переменную sSendToClient.

После передачи сообщения переходим обратно на чтение на шаг SERVER\_STATE\_READ, иначе по истечении времени tSendClient выполняем закрытие сокета на шаге SERVER\_STATE\_CLOSE.



Таймаут tSendClient может сработать в случае, если клиент закрыл свой сокет, не дожидаясь ответа. Также чем больше сообщение, тем больше время на передачу, поэтому переменную таймаута можно использовать для контроля ответа по времени.

```

SERVER_STATE_SEND:  (*отправляем ответ клиенту*)

diSendBytes := SysSockSendTo(hServerSocket, ADR(sSendToClient), LEN(sSendToClient), c_diFlags, ADR(stServerSettings), SIZEOF(stServerSettings) );

IF diSendBytes > 0 THEN
  eState := SERVER_STATE_READ;
  fbTon(IN := FALSE);
ELSE
  fbTon(IN := TRUE, PT := tSendClient);

  IF fbTon.Q THEN
    eState := SERVER_STATE_CLOSE;
    fbTon(IN := FALSE);
  END_IF
END_IF

```

Рисунок 5.2.7 – Ответ клиенту SERVER\_STATE\_SEND

Поскольку в рамках примера передается переменная типа STRING, то для определения размера пересылаемых данных используется оператор LEN.

### 5.2.6. Закрытие сокета SERVER\_STATE\_CLOSE

Для остановки приема и передачи используется функция [SysSockShutdown](#), а закрытие выполняется с помощью функции [SysSockClose](#).

```

SERVER_STATE_CLOSE:  (*закрываем сокет*)

SysSockShutdown(hServerSocket, c_diHow);      (*запрет приема и передачи сокета*)

fbTon(IN := TRUE, PT := tSockClose);         (*желательно разделять запрет приема/передачи и закрытие*)

IF fbTon.Q THEN
  SysSockClose(hServerSocket);               (*закрытие сокета*)
  eState := SERVER_STATE_CREATE;
  fbTon(IN := FALSE);
END_IF

```

Рисунок 5.2.8 – Закрытие сокета SERVER\_STATE\_CLOSE

## 5.3. Подпрограмма StopPrg

В проекте добавлена подпрограмма StopPrg (рис.5.3.1) для закрытия созданных сокетов в случае сброса работы программы. Сброс программы может потребоваться, когда необходимо начать цикл заново не перезагружая контроллер.

Например, сбросить программу можно через CODESYS меню Вставка | Сброс холодный. Холодный сброс заново инициализирует все переменные, включая RETAIN.

```

(*в случае сброса работы программы выполняем закрытие сокета*)
SysSockShutdown(PLC_PRG.hServerSocket, PLC_PRG.c_diHow);
SysSockClose(PLC_PRG.hServerSocket);

```

Рисунок 5.3.1 – Подпрограмма StopPrg

Для привязки подпрограммы к системному событию нужно перейти в конфигурацию задач, Системные события, поставить галочку возле stop и выбрать подпрограмму (нажать F2 под вызываемым POU).

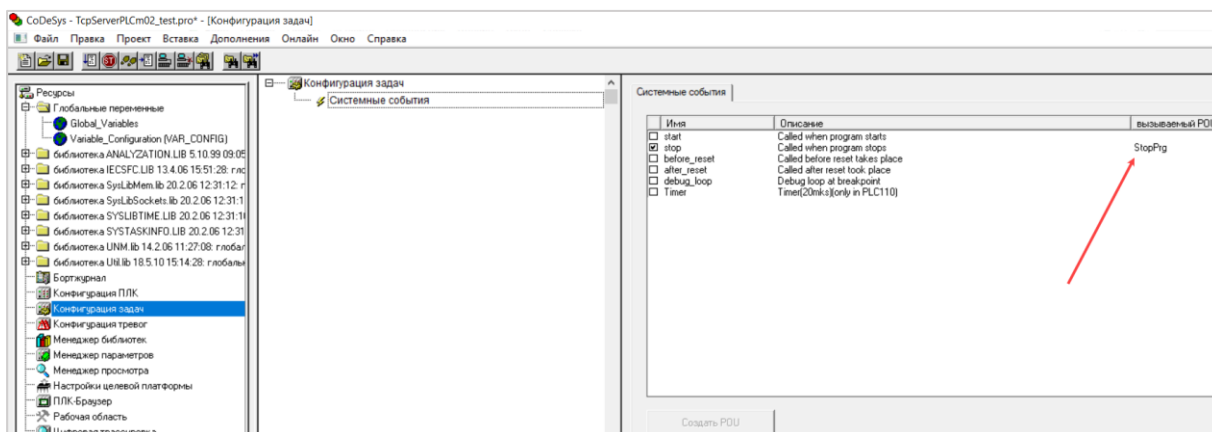


Рисунок 5.3.2 – Привязка подпрограммы StopPrg к проекту

#### 5.4. Визуализация (UDP сервер)

Проект включает в себя экран визуализации с названием Visu (рис.5.4.1). Она содержит параметры для отображения: порт и крайний запрос клиента. Отправляемое сообщение клиенту можно ввести с экрана визуализации.

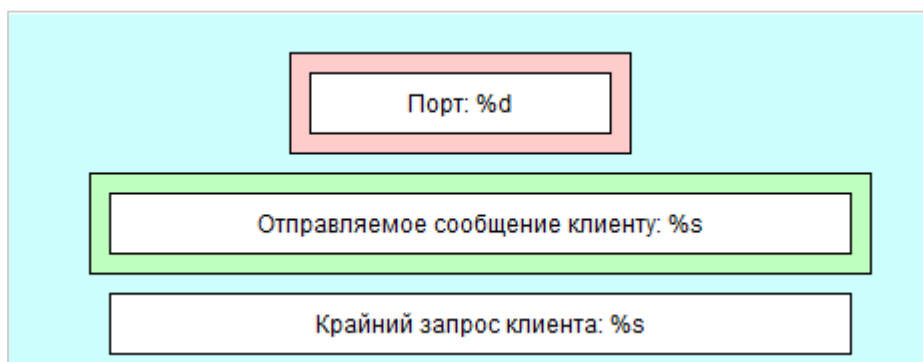


Рисунок 5.4.1 – Внешний вид экрана визуализации

#### 5.5. Реализация UDP-клиента

Алгоритм работы клиента имеет следующие этапы:

1. Создание сокета (Socket Creation)
2. Запрос/ответ (Send/Recv)
3. Закрытие сокета (Socket Close)

Данный алгоритм легко представить в виде последовательности шагов, выполняемых с помощью оператора CASE. Для удобства свяжем номера шагов с символьными именами через перечисления CLIENT\_STATE.

```

TYPE CLIENT_STATE :
(
  CLIENT_STATE_IDLE      := 0,      (*ожидание*)
  CLIENT_STATE_CREATE    := 1,      (*создание сокета*)
  CLIENT_STATE_SEND      := 2,      (*отправляем запрос серверу*)
  CLIENT_STATE_READ      := 3,      (*получаем ответ от сервера*)
  CLIENT_STATE_CLOSE     := 4       (*закрытие сокета*)
);
END_TYPE

```

Рисунок 5.5.1 – Объявление перечисления CLIENT\_STATE

Далее опишем каждый шаг.

### 5.5.1. Переменные программы PLC\_PRG (UDP клиент)

На рис. 5.5.2 представлены переменные, которые используются в программе.

В рамках данного примера для обмена используются переменные типа STRING, но в общем случае можно использовать любые типы данных (например, массив байт).

```

PROGRAM PLC_PRG
VAR
(*переменные задания*)
  xStart:      BOOL;                (*запуск выполнения обмена с сервером*)
  wPort:       WORD := 502;         (*порт сокета*)
  sIPAddr:     STRING := '10.2.11.119'; (*IP сервера*)

(*внутренние переменные*)
  eState:      CLIENT_STATE;        (*текущее состояние выполнения программы*)
  hClientSocket: HANDLE;            (*идентификатор клиентского сокета*)
  stClientSettings: SOCKADDRESS;    (*структура для хранения адреса сокета*)
  eStatus:     SOCKET_STATUS;       (*состояние сокета в символьном виде*)
  dwStatus:    DWORD;               (*переменная нужна для корректного получения состояния сокета*)

(*переменные для отправки сообщения сервера*)
  diSendBytes: DINT;                (*количество отправленных байт*)
  sSendToServer: STRING(10);        (*переменная для отправки сообщения серверу*)

(*переменные для получения ответа от сервера*)
  diRecvBytes: DINT;                (*количество принятых байт*)
  sReadFromServer: STRING(16);      (*переменная приема сообщения от сервера*)

(*таймеры задержки включения*)
  fbTon:       TON;                 (*таймер задержки включения*)
  tRecvServer: TIME := T#1s;        (*ожидание на прием сообщения от серверу*)
  tSendServer: TIME := T#2ms;       (*ожидание на отравку сообщения серверу*)
  tSockClose:  TIME := T#2ms;       (*задержка между запретом/передачей и закрытием сокета*)

  diOption:    DINT := 0;           (*значение для запрашиваемой опции*)
END_VAR

VAR CONSTANT
(*аргументы сокета*)
  c_diSoNbio:  DINT := 16#1014;     (*параметр SO_NBIO переводит в неблокирующий режим*)
  c_diSoError: DINT := 16#1007;     (*параметр SO_ERROR выводит статус ошибки*)
  c_diHow:     DINT := 2;           (*тип запрещаемых действий*)
  c_diFlags:   DINT := 0;           (*способ вызова функции*)
END_VAR

```

Рисунок 5.5.2 – Переменные программы PLC\_PRG для UDP клиента

## 5.5.2. Ожидание команды для обмена с сервером CLIENT\_STATE\_IDLE

Когда переменная `xStart = TRUE` выполняется однократный проход по всем этапам алгоритма работы клиента.

```
CLIENT_STATE_IDLE:
  IF xStart THEN      (*запуск выполнения обмена с сервером*)
    eState := CLIENT_STATE_CREATE;
    xStart := FALSE;
  END_IF
```

Рисунок 5.5.3 – Шаг CLIENT\_STATE\_IDLE

## 5.5.3. Создание сокета CLIENT\_STATE\_CREATE

С помощью функции [SysSockCreate](#) создается сокет и возвращается для него системный идентификатор (`handle`). Данная функция в качестве входных параметров принимает аргументы, задающие тип и протокол сокета.

```
CLIENT_STATE_CREATE:  (*создаем сокет*)

  hClientSocket := SysSockCreate(SOCKET_AF_INET, SOCKET_DGRAM, SOCKET_IPPROTO_UDP);

  IF hClientSocket <> SOCKET_INVALID THEN
    (*переводим в неблокирующий режим*)
    SysSockSetOption(hClientSocket, SOCKET_SOL, c_diSoNbio, ADR(diOption), SIZEOF(diOption));

    eState := CLIENT_STATE_SEND;
  ELSE
    eState := CLIENT_STATE_CLOSE;
  END_IF

  stClientSettings.sin_family := SOCKET_AF_INET;      (*тип сокета*)
  stClientSettings.sin_port := SysSockHtons(wPort);  (*порт сервера*)
  stClientSettings.sin_addr := SysSockInetAddr(sIPAddr); (*IP сервера*)
```

Рисунок 5.5.4 – Создание сокета CLIENT\_STATE\_CREATE

Линейку контроллеров ПЛК1хх M02 необходимо переводить в [неблокирующий режим](#) с помощью функции [SysSockSetOption](#) для того, чтобы при вызове любой функции контроль над программой сразу возвращался.

**Внимание!** Контроллеры ПЛК1хх переводить в неблокирующий режим не нужно, так как они по умолчанию настроены на этот режим.

## 5.5.4. Отправка сообщения серверу CLIENT\_STATE\_SEND

Клиент передает сообщение с помощью функции [SysSockSend](#). Передаваемое сообщение необходимо записать в переменную `sSendToServer`.

Если сообщение передали (функция вернула количество переданных байт), то переходим к чтению на шаг CLIENT\_STATE\_READ. Иначе если клиент не отправляет сообщение в течении времени `tSendServer`, то переходим к закрытию на шаг CLIENT\_STATE\_CLOSE.

Таймаут `tSendServer` может сработать в случае, если сервер закрыл свой сокет, не дожидаясь ответа. Также чем больше сообщение, тем больше времени на передачу, поэтому переменную таймаута можно использовать для контроля ответа по времени.

```

CLIENT_STATE_SEND:    (*отправляем сообщение серверу*)

    diSendBytes := SysSockSendTo(hClientSocket, ADR(sSendToServer), LEN(sSendToServer), c_diFlags, ADR(stClientSettings), sizeof(stClientSettings));

    IF diSendBytes > 0 THEN
        eState := CLIENT_STATE_READ;
        fbTon(IN:=FALSE);
    ELSE
        fbTon(IN:=TRUE, PT:=tSendServer);

        IF fbTon.Q THEN
            eState := CLIENT_STATE_CLOSE;
            fbTon(IN:=FALSE);
        END_IF
    END_IF

    SysSockGetOption(hClientSocket, SOCKET_SOL, c_diSoError, ADR(dwStatus), sizeof(dwStatus));
    eStatus := DWORD_TO_WORD(dwStatus); (*состояние сокета*)

```

**Рисунок 5.5.5 – Отправка сообщения CLIENT\_STATE\_SEND**

Поскольку в рамках примера передается переменная типа STRING, то для определения размера пересылаемых данных используется оператор LEN.

На шаге передачи сообщения добавлена функция [SysSockGetOption](#). Она возвращает состояние сокета в переменную типа DWORD dwStatus. Для удобства числовые номера состояния свяжем с символьными именами с помощью перечисления SOCKET\_STATUS.

```

TYPE SOCKET_STATUS : (*статус сокета*)
(
    IP_ERR_EMPTY           := 16#0000,    (*Статус пуст*)
    IP_ERR_MISC           := 16#FFFF,    (*Разные ошибки, которые не имеют конкретный код ошибки*)
    IP_ERR_TIMEDOUT       := 16#FFFE,    (*Время операции истекло*)
    IP_ERR_ISCONN         := 16#FFFD,    (*Сокет уже подключен*)
    IP_ERR_OP_NOT_SUPP    := 16#FFFC,    (*Операция не поддерживается для выбранного сокета*)
    IP_ERR_CONN_ABORTED   := 16#FFFB,    (*Соединение было прервано*)
    IP_ERR_WOULD_BLOCK    := 16#FFFA,    (*Сокет находится в неблокирующем режиме*)
    IP_ERR_CONN_REFUSED   := 16#FFF9,    (*Соединение отклонено одноранговым узлом*)
    IP_ERR_CONN_RESET     := 16#FFF8,    (*Соединение было сброшено*)
    IP_ERR_NOT_CONN       := 16#FFF7,    (*Сокет не подключен*)
    IP_ERR_ALREADY        := 16#FFF6,    (*Сокет уже находится в запрошенном состоянии*)
    IP_ERR_IN_VAL         := 16#FFF5,    (*Переданное значение для конфигурации не действительно*)
    IP_ERR_MSG_SIZE       := 16#FFF4,    (*Сообщение слишком большое для отправки*)
    IP_ERR_PIPE           := 16#FFF3,    (*Сокет не находится в правильном состоянии для данной операции*)
    IP_ERR_DEST_ADDR_REQ  := 16#FFF2,    (*Адрес не указан*)
    IP_ERR_SHUTDOWN       := 16#FFF1,    (*Соединение было закрыто как только все данные были получены после обнаружения запроса*)
    IP_ERR_NO_PROTO_OPT   := 16#FFF0,    (*Неизвестная опция сокета для setsockopt () или getsockopt ()*)
    IP_ERR_NO_MEM         := 16#FFEE,    (*Недостаточно памяти*)
    IP_ERR_ADDR_NOT_AVAIL := 16#FFED,    (*Неизвестный путь для отправки в указанный адрес*)
    IP_ERR_ADDR_IN_USE    := 16#FFEC,    (*Сокет уже соединен с адресом и портом*)
    IP_ERR_IN_PROGRESS    := 16#FFEA,    (*Операция все еще продолжается*)
    IP_ERR_NO_BUF         := 16#FFE9,    (*Внутренний буфер недоступен*)
    IP_ERR_NOT_SOCKET     := 16#FFE8,    (*Сокет не был открыт или уже был закрыт*)
    IP_ERR_FAULT          := 16#FFE7,    (*Общая ошибка при неудачной операции*)
    IP_ERR_NET_UNREACH    := 16#FFE6,    (*Нет доступа к нужной сети*)
    IP_ERR_PARAM          := 16#FFE5,    (*Неверный параметр для функции*)
    IP_ERR_LOGIC          := 16#FFE4,    (*Логическая ошибка, которая не должна была произойти*)
    IP_ERR_NOMEM          := 16#FFE3,    (*Системная ошибка: нет памяти для данной операции*)
    IP_ERR_NOBUFFER       := 16#FFE2,    (*Системная ошибка: нет внутреннего буфера для данной операции*)
    IP_ERR_RESOURCE       := 16#FFE1,    (*Системная ошибка: недостаточно свободных ресурсов*)
    IP_ERR_BAD_STATE      := 16#FFE0,    (*Сокет находится в неожиданном состоянии*)
    IP_ERR_TIMEOUT        := 16#FFDF,    (*Тайм-аут запрашиваемой операции*)
    IP_ERR_NO_ROUTE       := 16#FFDC,    (*Пункт назначения недоступен*)
    IP_ERR_TRIAL_LIMIT    := 16#FF80,    (*Пробный лимит превышен*)
);
END_TYPE

```

**Рисунок 5.5.6 – Статусы сокета (опция SO\_ERROR)**

### 5.5.5. Получение сообщения от сервера CLIENT\_STATE\_READ

Сокет принимает сообщение с помощью функции [SysSockRecv](#). После шага CLIENT\_STATE\_READ переходим к закрытию клиентского сокета на шаг CLIENT\_STATE\_CLOSE.

```
CLIENT_STATE_READ:    (*получаем ответ от сервера*)

    diRecvBytes      := SysSockRecvFrom(hClientSocket, ADR(sReadFromServer), SIZEOF(sReadFromServer), c_diFlags, ADR(stClientSettings), SIZEOF(stClientSettings));

    IF diRecvBytes > 0 THEN (*ответ получили, закрываем сокет*)
        eState := CLIENT_STATE_CLOSE;
        fbTon(IN:=FALSE);
    ELSE
        fbTon(IN:=TRUE, PT:=tRecvServer);

        IF fbTon.Q THEN
            eState := CLIENT_STATE_CLOSE;
            fbTon(IN:=FALSE);
            (*ошибка: не удалось получить ответ*)
        END_IF
    END_IF
```

Рисунок 5.5.7 – Получение сообщения от сервера CLIENT\_STATE\_READ

### 5.5.6. Закрытие сокета CLIENT\_STATE\_CLOSE

Перед закрытием сокета корректно выполнять запрет приема и передачи функцией [SysSockShutDown](#). Закрытие сокета выполняется с помощью функции [SysSockClose](#).

```
CLIENT_STATE_CLOSE:    (*закрываем сокет*)

    SysSockShutDown(hClientSocket, c_diHow);    (*запрет приема и передачи*)

    fbTon(IN := TRUE, PT := tSockClose);    (*желательно разделять запрет приема/передачи и закрытие*)

    IF fbTon.Q THEN
        SysSockClose(hClientSocket);    (*закрытие сокета*)
        eState := CLIENT_STATE_IDLE;
        fbTon(IN := FALSE);
    END_IF
```

Рисунок 5.5.8 – Шаг CLIENT\_STATE\_CLOSE

## 5.6. Визуализация (UDP клиент)

Проект включает в себя экран визуализации с названием Visu (рис.5.6.1). Она содержит параметры для записи: IP-адреса сервера, порта и отправляемое сообщение серверу. А также параметр чтения - ответ от сервера. Обмен с сервером выполняется по команде через кнопку ОТПРАВИТЬ. Обмен прекращается после получения ответа от сервера.

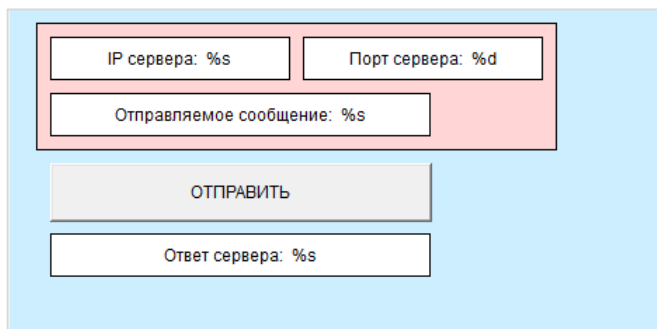


Рисунок 5.6.1 – Внешний вид экрана визуализации