

2018

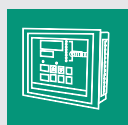


# СПК

## Реализация обмена через сокет

Руководство для продвинутых пользователей

Версия: 1.0  
Дата: 15.02.2018



## Оглавление

1. Цель и структура документа .....	4
2. Основные сведения о работе с сокетами.....	5
2.1. Общая информация о сокетах.....	5
2.2. Серверы и клиенты.....	6
2.3. Протокол UDP .....	7
2.4. Протокол TCP .....	8
2.5. Вопросы информационной безопасности .....	8
2.6. Средства отладки.....	8
2.7. Средства для работы с сокетами в CODESYS .....	9
3. Библиотека CAA Net Base Services.....	10
3.1. Добавление библиотеки в проект CODESYS.....	10
3.2. Структуры и перечисления .....	11
3.2.1. Структура NBS.IP_ADDR .....	11
3.2.2. Перечисление NBS.ERROR.....	11
3.3. ФБ работы с протоколом UDP .....	12
3.3.1. ФБ NBS.UDP_Peer.....	12
3.3.2. ФБ NBS.UDP_Receive.....	13
3.3.3. ФБ NBS.UDP_Send .....	14
3.3.4. ФБ NBS.UDP_ReceiveBuffer.....	15
3.3.5. ФБ NBS.UDP_SendBuffer .....	16
3.4. ФБ работы с протоколом TCP .....	17
3.4.1. ФБ NBS.TCP_Server.....	17
3.4.2. ФБ NBS.TCP_Connection.....	18
3.4.3. ФБ NBS.TCP_Client.....	19
3.4.4. ФБ NBS.TCP_Read .....	20
3.4.5. ФБ NBS.TCP_Write .....	21
3.4.6. ФБ NBS.TCP_ReadBuffer .....	22
3.4.7. ФБ NBS.TCP_WriteBuffer .....	23
3.5. Дополнительные функции.....	24
3.5.1. Функция NBS.IPSTRING_TO_UDINT .....	24
3.5.2. Функция NBS.IPSTRING_TO_UDINT .....	24
3.5.3. Функция NBS.IS_MULTICAST_GROUP.....	25

4. Примеры работы с библиотекой CAA Net Base Services .....	26
4.1. Краткое описание примеров .....	26
4.2. Реализация UDP-сервера и UDP-клиента .....	27
4.2.1. Основная информация.....	27
4.2.2. Реализация UDP-сервера .....	27
4.2.3. Реализация UDP-клиента .....	31
4.3. Реализация TCP-сервера и TCP-клиента .....	34
4.3.1. Основная информация.....	34
4.3.2. Реализация TCP-сервера .....	34
4.3.3. Реализация TCP-клиента .....	39
4.4. Работа с примером.....	42
4.5. Рекомендации и замечания .....	43
Приложение. Листинг примера.....	44
A1. UDP-сервер .....	44
A.1.1. Перечисление SERVER_STATE.....	44
A.1.2. Функция MIRROR .....	44
A.1.3. Программа PLC_PRG.....	45
A2. UDP-клиент .....	48
A.2.1. Перечисление CLIENT_STATE.....	48
A.2.2. Программа PLC_PRG.....	48
B1. TCP-сервер.....	51
B.1.1. Перечисление SERVER_STATE .....	51
B.1.2. Структура CONNECTION.....	51
B.1.3. Функция MIRROR .....	52
B.1.4. Программа PLC_PRG.....	53
B2. TCP-клиент.....	56
B.2.1. Перечисление CLIENT_STATE .....	56
B.2.2. Программа PLC_PRG.....	56

## 1. Цель и структура документа

Одним из современных трендов промышленной автоматизации является повсеместное внедрение интерфейса [Ethernet](#) и использование для обмена данными между устройствами протоколов, основанных на стеке [TCP/IP](#) – Modbus TCP, KNX, MQTT, SNMP и др.

Другой тенденцией является расширение коммуникационных задач ПЛК: помимо опроса устройств и передачи данных на верхний уровень (в OPC-серверы или SCADA-системы) возникает потребность в передаче файлов (например, по FTP), синхронизации данных с серверами точного времени (NTP), рассылке сообщений по электронной почте (SMTP/POP3) и т.д.

Некоторые ПЛК имеют готовые компоненты, предназначенные для решения конкретных задач. Такие компоненты просты и удобны в использовании, но зачастую требуют покупки отдельной лицензии. Кроме того, набор доступных компонентов далеко не всегда соответствует потребностям пользователя.

Среда **CODESYS 3.5** предоставляет возможность работы с [сетевыми сокетами](#), что позволяет программисту реализовать свой собственный протокол обмена поверх стандартных протоколов **UDP** или **TCP**. Для этого требуется:

- понимание основ сетевого взаимодействия систем;
- хорошие навыки программирования на языке ST;
- спецификация реализуемого протокола.

В [п. 2](#) приведена основная информация о работе с сокетами.

В [п. 3](#) приведено описание библиотеки **CAA Net Base Services**.

В [п. 4](#) рассмотрены примеры использования библиотеки.

Документ рекомендуется читать строго последовательно.

## 2. Основные сведения о работе с сокетами

### 2.1. Общая информация о сокетах

[Сокет](#) – это программный интерфейс, который обеспечивает обмен данными между процессами. Данный документ посвящен сетевым сокетам – которые позволяют организовать обмен данными между процессами, выполняющимися на разных устройствах. Примером такого процесса может являться пользовательская программа, выполняемая ПЛК. Формат передачи данных между двумя устройствами зависит от используемого [протокола обмена](#).

С точки зрения пользователя сокет представляет собой пару «IP-адрес – порт». [IP-адрес](#) позволяет идентифицировать сетевой адаптер конкретного устройства, а [порт](#) – конкретное приложение этого устройства. Примером таких приложений, например, могут быть Modbus TCP Slave и web-сервер, обслуживающий web-визуализацию. Фактически порт представляет собой целое число в диапазоне **1...65535**. В большинстве случаев порт стандартизирован на уровне используемого протокола обмена. В [данной статье](#) приведен список портов, используемых различными протоколами, реализованными поверх **UDP** и **TCP**. Приложение может использовать несколько портов, но каждый порт в отдельно взятый момент времени может использоваться только одним приложением.

Среда исполнения **CODESYS V3** в процессе своей работы использует следующие порты ПЛК:

Табл. 2.1. Список портов ПЛК, используемых средой исполнения **CODESYS V3**

Порт	Описание
1740-1743	Выделены под задачи среды исполнения (UDP).
11740	Выделены под задачи среды исполнения (TCP).
1217	Codesys Gateway (TCP) – шлюз для связи среды исполнения и среды программирования.
8080	Сервер web-визуализации.

Также стоит отметить порт **502**, который традиционно используется протоколом **Modbus TCP**.

Итак, тезисно подведем итоги данного подпункта:

- сокет характеризуется **IP-адресом** и **портом**. Зная их (а также протокол), можно организовать обмен данными с конкретным приложением конкретного устройства;
- среда исполнения **CODESYS** в процессе своей работы использует определенные порты ПЛК. Не следует пытаться занимать их другими процессами.

## 2.2. Серверы и клиенты

Большинство сетевых протоколов основано на архитектуре «[клиент – сервер](#)». Фактически клиент и сервер являются приложениями, выполняемые на различных (в определенных случаях – на одном и том же) устройствах. Сокеты также разделяются на серверные и клиентские.

Сервер ожидает запросов от клиента, и при их получении выполняет заданные операции – после чего, при необходимости, отправляет клиенту ответ. Сервер не может являться инициатором обмена. Простейшим примером сервера и клиента являются web-сервер и web-браузер. Следует отметить, что один сервер может обслуживать множество клиентов.

Архитектура «клиент – сервер» достаточно близка к архитектуре «ведущий – ведомый» (Master – Slave), используемой при обмене данными по последовательной линии связи (RS-232/RS-485). Принципиальным отличием является то, что при сетевом обмене нет явного ограничения на число активных устройств (при использовании последовательных интерфейсов в каждый момент времени активным является только одно устройство, регулирующее уровень сигнала на линии связи).

В рамках данного документа мы рассмотрим реализацию сервера и клиента для протоколов UDP и TCP. Основная информация об этих протоколах приведена в следующих пунктах.

## 2.3. Протокол 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. К основным характеристикам протокола относятся:

- *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.

Как упоминалось выше, пакеты имеют определенные границы. Если размер пакета превышает эти границы, то он разбивается на несколько отдельных пакетов (фрагментируется). Не всё сетевое оборудование поддерживает работу с фрагментированными UDP-пакетами.

Не вдаваясь в технические подробности отметим, что для предотвращения фрагментации размер данных в пакете не должен превышать **1432 байт**, а для уверенности в том, что пакет сможет быть принят любым устройством – **508 байт**.

Протокол UDP поддерживает следующие схемы маршрутизации:

- [Unicast](#) – передача данных конкретному устройству;
- [Multicast](#) – передача данных группе устройств. Для этого устройство должно быть подписано на Multicast-группу, которая характеризуется IP-адресом. Для мультивещания зарезервирована подсеть **224.0.0.0 – 239.255.255.255**, при этом выделенные для частного использования адреса начинаются с **239.0.0.0**;
- [Broadcast](#) – передача данных всем устройствам данного сегмента сети. Для этого должен использоваться последний IP-адрес сегмента. Например, при отправке UDP-пакета на адрес **10.2.11.255**, он будет доставлен устройствам с адресами **10.2.11.1 – 10.2.11.254**.



Рис. 2.1. Схемы маршрутизации UDP

## 2.4. Протокол TCP

[TCP](#) (Transmission Control Protocol) – один из основных протоколов интернета, предназначенный для управления передачей данных. Протокол TCP выполняет функции протокола транспортного уровня [модели OSI](#). Сети и подсети, в которых совместно используются протоколы TCP и IP, называются сетями TCP/IP. К основным характеристикам протокола относятся:

- *Надёжность* — 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 надёжностью и перегрузками;
- *Потоковость* — данные читаются как поток байтов, не передается никаких особых обозначений для границ сообщения или сегментов.

Принципиальным отличием TCP от UDP является необходимость установки соединения перед началом обмена данными.

## 2.5. Вопросы информационной безопасности

В рамках данного документа не рассматриваются вопросы информационной безопасности и защищенной передачи данных. В качестве источника информации по этому вопросу можно использовать документ [CODESYS Security Whitepaper](#) и раздел [Security](#) сайта CODESYS.

## 2.6. Средства отладки

В процессе отладки ПО, реализующего сетевой обмен, удобно использовать анализатор трафика [Wireshark](#) и TCP/UDP-терминал [Hercules](#) (для эмуляции сервера и клиента).



## 2.7. Средства для работы с сокетами в CODESYS

В сети можно найти множество материалов по программированию сокетов на различных языках программирования. В качестве примера отметим [эту статью](#). Работа с сокетами в Codesys происходит по тем же общим принципам.

В среде **CoDeSys 2.3** для работы с сокетами используется библиотека [SysLibSockets](#). Она содержит типичные функции, которые можно найти в подобных библиотеках для любого языка программирования (например, C) – connect(), bind(), accept() и т.д.

Хорошим источником информации по ее применению являются статьи [Войцеха Гомолка](#):

- [CoDeSys and Ethernet communication: The concept of Sockets and basic Function Blocks for communication over Ethernet. Part 1: UPD Client/Server](#)
- [The concept of Sockets and basic Function Blocks for communication over Ethernet. Part 2: TCP Server and TCP Client](#)

В среде **CODESYS 3.5** аналогом этой библиотеки является библиотека **SysSocket**. Пример работы с ней описан Михаилом Шевцовым ([ПК Пролог](#)) в видеоуроке [Программирование сокетов в CODESYS V3](#).

Применение данной библиотеки может оказаться затруднительным для пользователей, не имеющих опыта работы с сокетами, и потребует определенных затрат времени даже для тех, у кого подобный опыт есть. Это стало одной из причин разработки и включения в состав CODESYS V3 библиотеки **CAA Net Base Services**. Эта библиотека реализована на более высоком уровне абстракции и представляет собой обвязку вокруг стандартных функций работы с сокетами, предоставляя пользователю удобный и емкий программный интерфейс. Для создания сетевой части серверного или клиентского приложения в данном случае достаточно будет использовать всего несколько функциональных блоков. Описание и примеры использования этой библиотеки приведены в данном документе.

### 3. Библиотека CAA Net Base Services

#### 3.1. Добавление библиотеки в проект CODESYS

Библиотека **CAA Net Base Service** используется для обмена данными по протоколам UDP и TCP.

Для добавления библиотеки в проект **CODESYS** в **Менеджере библиотек** нажмите кнопку **Добавить** и выберите библиотеку **CAA Net Base Services**:

**Обратите внимание**, что версия библиотеки не должна превышать версию таргет-файла контроллера. В противном случае корректная работа контроллера не гарантируется.

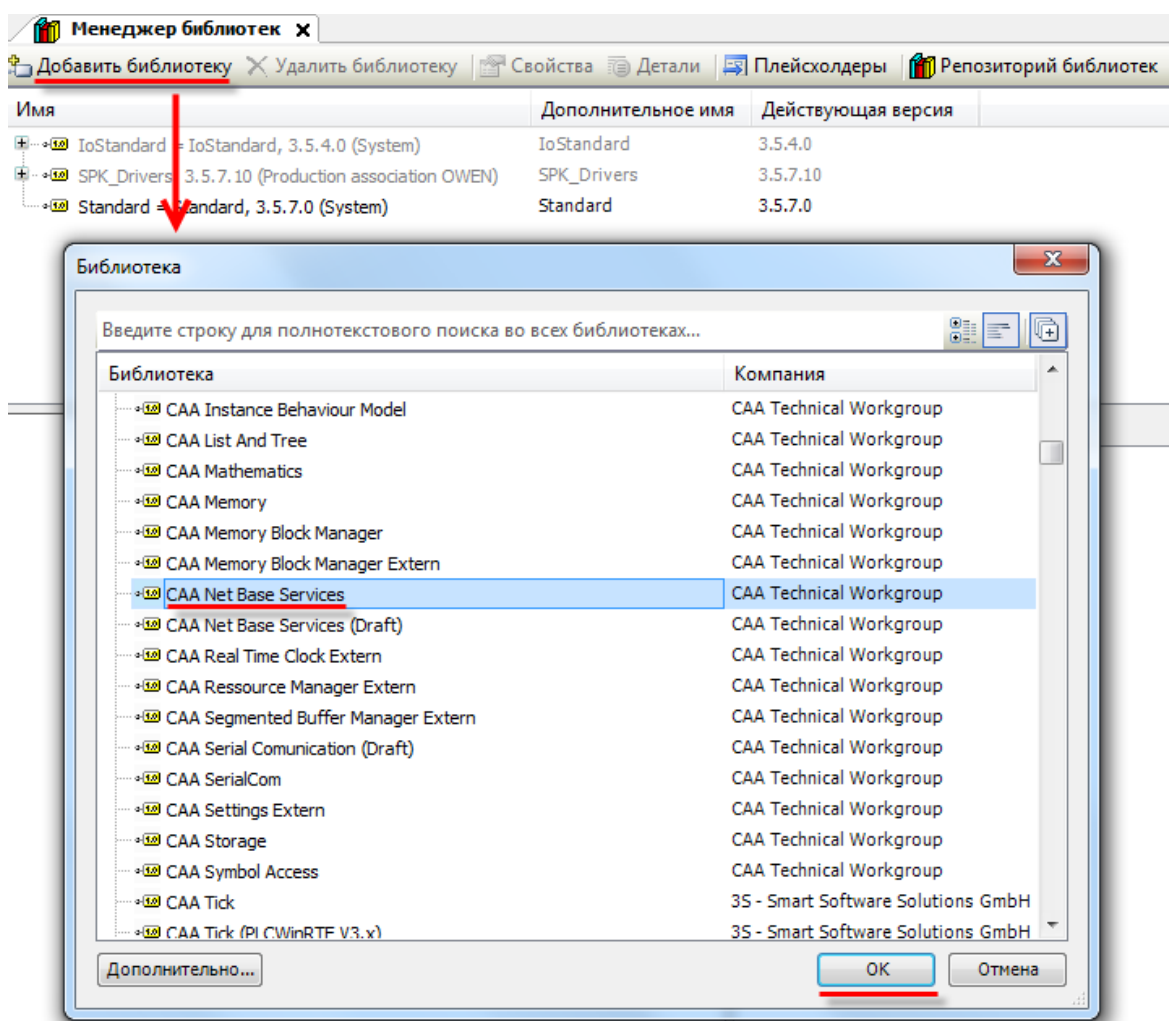


Рис. 3.1. Добавление библиотеки **CAA Net Base Services** в проект **CODESYS**

**Обратите внимание**, что при объявлении экземпляров ФБ библиотеки необходимо перед их названием указывать префикс NBS. (пример: **NBS.TCP\_Server**).

## 3.2. Структуры и перечисления

### 3.2.1. Структура NBS.IP\_ADDR

Структура **NBS.IP\_ADDR** описывает IP-адрес устройства.

Название	Тип данных	Описание
sAddr	STRING(80)	IP-адрес устройства в виде строки (например, '10.2.11.10').

### 3.2.2. Перечисление NBS.ERROR

Перечисление **NBS.ERROR** описывает ошибки, которые могут возникнуть при использовании ФБ библиотеки.

Название	Значение	Описание
NO_ERROR	0	Нет ошибок.
TIME_OUT	6001	Истек лимит времени для данной операции.
INVALID_ADDR	6002	На вход ФБ подан некорректный IP-адрес.
INVALID_HANDLE	6003	На вход ФБ подан некорректный дескриптор (handle).
INVALID_DATAPOINTER	6004	На вход ФБ подан некорректный указатель.
INVALID_DATASIZE	6005	На вход ФБ подан некорректный размер данных.
UDP_RECEIVE_ERROR	6006	Ошибка получения данных по UDP.
UDP_SEND_ERROR	6007	Ошибка передачи данных по UDP.
UDP_SEND_NOT_COMPLETE	6008	Передача по UDP не была завершена – возможно, были отправлены не все данные.
UDP_OPEN_ERROR	6009	Ошибка создания UDP-сокета.
UDP_CLOSE_ERROR	6010	Ошибка закрытия UDP-сокета.
TCP_SEND_ERROR	6011	Ошибка передачи данных по TCP.
TCP_RECEIVE_ERROR	6012	Ошибка получения данных по TCP.
TCP_OPEN_ERROR	6013	Ошибка создания TCP-сокета.
TCP_CONNECT_ERROR	6014	Ошибка сервера при обработке соединения клиента.
TCP_CLOSE_ERROR	6015	Ошибка закрытия TCP-сокета.
TCP_SERVER_ERROR	6016	Ошибка TCP-сервера.
WRONG_PARAMETER	6017	ФБ вызван с некорректными аргументами.
TCP_NO_CONNECTION	6019	Превышен лимит подключений к серверу.

### 3.3. ФБ работы с протоколом UDP

#### 3.3.1. ФБ NBS.UDP\_Peer

Функциональный блок **NBS.UDP\_Peer** создает UDP-сокеты и возвращает его дескриптор (**handle**), который используется для операций получения (ФБ [NBS.UDP\\_Receive](#), [NBS.UDP\\_ReceiveBuffer](#)) и передачи данных (ФБ [NBS.UDP\\_Send](#), [NBS.UDP\\_SendBuffer](#)).

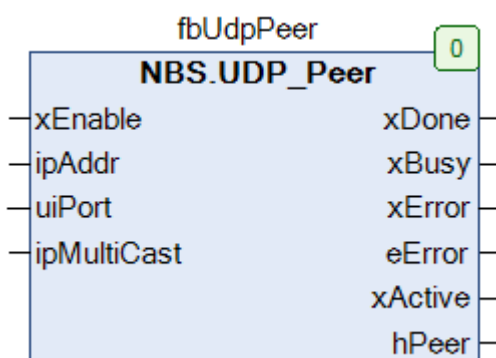


Рис. 3.2. Внешний вид ФБ **NBS.UDP\_Peer** на языке **CFC**

Название	Тип данных	Описание
<b>Входные переменные</b>		
<b>xEnable</b>	BOOL	Вход управления блоком. Пока он имеет значение <b>TRUE</b> – блок находится в работе.
<b>ipAddr</b>	<a href="#">NBS.IP_ADDR</a>	IP-адрес устройства, на котором создается сокет.
<b>uiPort</b>	UINT	Номер порта сокета.
<b>uiMultiCast</b>	<a href="#">NBS.IP_ADDR</a>	IP-адрес мультикаст-группы. <i>В текущих версиях CODESYS мультикаст не поддерживается.</i>
<b>Выходные переменные</b>		
<b>xDone</b>	BOOL	Флаг завершения работы блока. Принимает значение <b>TRUE</b> на один цикл.
<b>xBusy</b>	BOOL	Флаг «ФБ в процессе работы».
<b>xError</b>	BOOL	Флаг ошибки. Принимает значение <b>TRUE</b> при возникновении ошибки.
<b>eError</b>	<a href="#">NBS.ERROR</a>	Статус работы ФБ (или имя ошибки).
<b>xActive</b>	BOOL	Флаг «сокеты успешно открыты». Пока он имеет значение <b>TRUE</b> – сокет открыт, и его можно использовать.
<b>hPeer</b>	NBS.CAA.HANDLE	Дескриптор открытого сокета устройства.

### 3.3.2. ФБ NBS.UDP\_Receive

Функциональный блок **NBS.UDP\_Receive** используется для получения данных. Прослушиваемый порт задается при создании UDP-сокета с помощью ФБ [NBS.UDP\\_Peer](#).

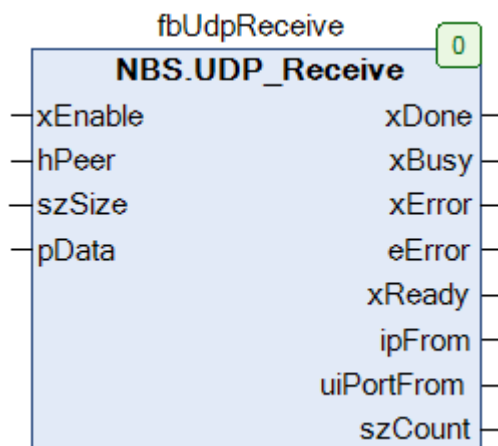


Рис. 3.3. Внешний вид ФБ **NBS.UDP\_Receive** на языке **CFC**

Название	Тип данных	Описание
<b><i>Входные переменные</i></b>		
<b>xEnable</b>	BOOL	Вход управления блоком. Пока он имеет значение <b>TRUE</b> – блок находится в работе.
<b>hPeer</b>	NBS.CAA.HANDLE	Дескриптор сокета устройства, полученный от ФБ NBS.UDP_PEER.
<b>szSize</b>	NBS.CAA.SIZE	Максимально допустимый размер получаемых данных в байтах. Может быть указан помощью оператора <b>SIZEOF</b> .
<b>pData</b>	NBS.CAA.PVOID	Начальный адрес для размещения принятых данных. Может быть указан с помощью оператора <b>ADR</b> .
<b><i>Выходные переменные</i></b>		
<b>xDone</b>	BOOL	Флаг завершения работы блока. Принимает значение <b>TRUE</b> на один цикл.
<b>xBusy</b>	BOOL	Флаг «ФБ в процессе работы».
<b>xError</b>	BOOL	Флаг ошибки. Принимает значение <b>TRUE</b> при возникновении ошибки.
<b>eError</b>	<a href="#">NBS.ERROR</a>	Статус работы ФБ (или имя ошибки).
<b>xReady</b>	BOOL	Флаг «данные получены». Принимает значение <b>TRUE</b> на один цикл при получении нового пакета данных.
<b>ipFrom</b>	<a href="#">NBS.IP_ADDR</a>	IP-адрес отправителя.
<b>uiPortFrom</b>	UINT	Номер порта отправителя.
<b>szCount</b>	NBS.CAA.SIZE	Размер принятых данных в байтах.

### 3.3.3. ФБ NBS.UDP\_Send

Функциональный блок **NBS.UDP\_Send** используется для отправки данных на заданный IP-адрес/порт.

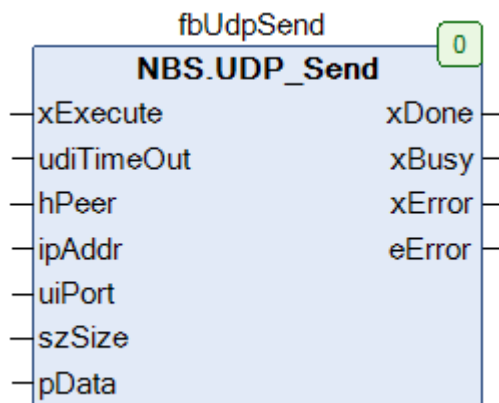


Рис. 3.4. Внешний вид ФБ **NBS.UDP\_Send** на языке **CFC**

Название	Тип данных	Описание
<b>Входные переменные</b>		
<b>xExecute</b>	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
<b>udiTimeOut</b>	UDINT	Допустимое время операции (в мкс). Значение <b>0</b> означает, что время выполнения ФБ не ограничивается.
<b>hPeer</b>	NBS.CAA.HANDLE	Дескриптор сокета устройства, полученный от ФБ <a href="#">NBS.UDP_PEER</a> .
<b>ipAddr</b>	<a href="#">NBS.IP_ADDR</a>	IP-адрес получателя.
<b>uiPort</b>	UINT	Номер порта получателя.
<b>szSize</b>	NBS.CAA.SIZE	Размер отправляемых данных в байтах. Может быть указан помощью оператора <b>SIZEOF</b> .
<b>pData</b>	NBS.CAA.PVOID	Начальный адрес отправляемых данных. Может быть указан с помощью оператора <b>ADR</b> .
<b>Выходные переменные</b>		
<b>xDone</b>	BOOL	Флаг завершения работы блока. Принимает значение <b>TRUE</b> на один цикл.
<b>xBusy</b>	BOOL	Флаг «ФБ в процессе работы».
<b>xError</b>	BOOL	Флаг ошибки. Принимает значение <b>TRUE</b> при возникновении ошибки.
<b>eError</b>	<a href="#">NBS.ERROR</a>	Статус работы ФБ (или имя ошибки).

### 3.3.4. ФБ NBS.UDP\_ReceiveBuffer

Функциональный блок **NBS.UDP\_ReceiveBuffer** используется для получения данных. Прослушиваемый порт задается при создании UDP-сокета с помощью ФБ [NBS.UDP\\_Peer](#). В отличие от ФБ [NBS.UDP\\_Receive](#) данный блок не копирует данные по указателю, а возвращает дескриптор буфера, в котором они были размещены. Для работы с буфером используется библиотека **CAA SegBufMan**. Этот способ является менее ресурсозатратным, но более сложным в использовании.

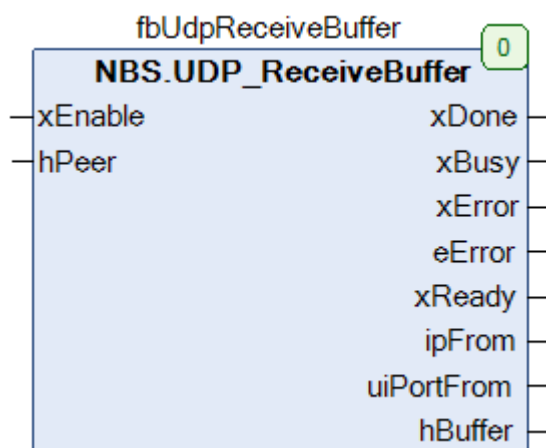


Рис. 3.5. Внешний вид ФБ **NBS.UDP\_ReceiveBuffer** на языке CFC

Название	Тип данных	Описание
<b>Входные переменные</b>		
<b>xEnable</b>	BOOL	Вход управления блоком. Пока он имеет значение <b>TRUE</b> – блок находится в работе.
<b>hPeer</b>	NBS.CAA.HANDLE	Дескриптор сокета устройства, полученный от ФБ <a href="#">NBS.UDP_PEER</a> .
<b>Выходные переменные</b>		
<b>xDone</b>	BOOL	Флаг завершения работы блока. Принимает значение <b>TRUE</b> на один цикл.
<b>xBusy</b>	BOOL	Флаг «ФБ в процессе работы».
<b>xError</b>	BOOL	Флаг ошибки. Принимает значение <b>TRUE</b> при возникновении ошибки.
<b>eError</b>	<a href="#">NBS.ERROR</a>	Статус работы ФБ (или имя ошибки).
<b>xReady</b>	BOOL	Флаг «данные получены». Принимает значение <b>TRUE</b> на один цикл при получении нового пакета данных.
<b>ipFrom</b>	<a href="#">NBS.IP_ADDR</a>	IP-адрес отправителя.
<b>uiPortFrom</b>	UINT	Номер порта отправителя.
<b>hBuffer</b>	NBS.CAA.HANDLE	Дескриптор буфера принятых данных.

### 3.3.5. ФБ NBS.UDP\_SendBuffer

Функциональный блок **NBS.UDP\_SendBuffer** используется для передачи данных. В отличие от ФБ [NBS.UDP\\_Send](#) данный блок не копирует данные по указателю, а принимает на вход дескриптор буфера, в котором они размещены. Для работы с буфером используется библиотека **CAA SegBufMan**. Этот способ является менее ресурсозатратным, но более сложным в использовании.

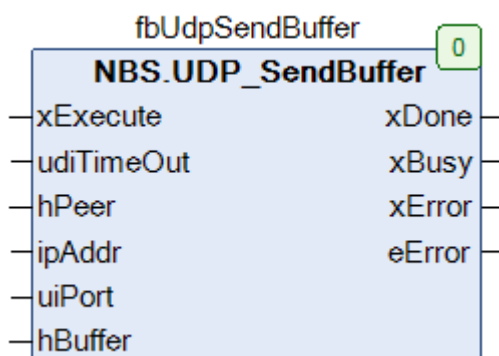


Рис. 3.6. Внешний вид ФБ **NBS.UDP\_SendBuffer** на языке **CFC**

Название	Тип данных	Описание
<b>Входные переменные</b>		
<b>xExecute</b>	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
<b>udiTimeOut</b>	UDINT	Допустимое время операции (в мкс). Значение <b>0</b> означает, что время выполнения ФБ не ограничивается.
<b>hPeer</b>	NBS.CAA.HANDLE	Дескриптор сокета устройства, полученный от ФБ <a href="#">NBS.UDP_PEER</a> .
<b>ipAddr</b>	<a href="#">NBS.IP_ADDR</a>	IP-адрес получателя.
<b>uiPort</b>	UINT	Номер порта получателя.
<b>hBuffer</b>	NBS.CAA.HANDLE	Дескриптор буфера отправляемых данных.
<b>Выходные переменные</b>		
<b>xDone</b>	BOOL	Флаг завершения работы блока. Принимает значение <b>TRUE</b> на один цикл.
<b>xBusy</b>	BOOL	Флаг «ФБ в процессе работы».
<b>xError</b>	BOOL	Флаг ошибки. Принимает значение <b>TRUE</b> при возникновении ошибки.
<b>eError</b>	<a href="#">NBS.ERROR</a>	Статус работы ФБ (или имя ошибки).



### 3.4. ФБ работы с протоколом TCP

#### 3.4.1. ФБ NBS.TCP\_Server

Функциональный блок **NBS.TCP\_Server** создает серверный TCP-сокет и возвращает его дескриптор (**handle**), который используется для обработки соединений с помощью ФБ [NBC.TCP\\_Connection](#).

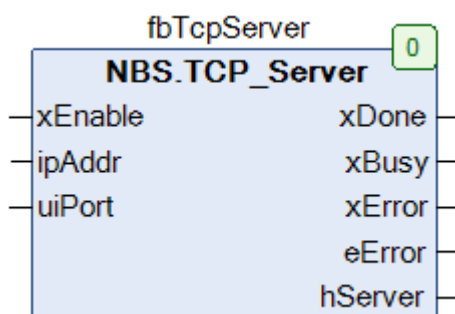


Рис. 3.7. Внешний вид ФБ **NBS.TCP\_Server** на языке **CFC**

Название	Тип данных	Описание
<b>Входные переменные</b>		
<b>xEnable</b>	BOOL	Вход управления блоком. Пока он имеет значение <b>TRUE</b> – блок находится в работе.
<b>ipAddr</b>	<a href="#">NBS.IP_ADDR</a>	IP-адрес устройства, на котором создается сокет.
<b>uiPort</b>	UINT	Номер порта сервера.
<b>Выходные переменные</b>		
<b>xDone</b>	BOOL	Флаг завершения работы блока. Принимает значение <b>TRUE</b> на один цикл.
<b>xBusy</b>	BOOL	Флаг «ФБ в процессе работы».
<b>xError</b>	BOOL	Флаг ошибки. Принимает значение <b>TRUE</b> при возникновении ошибки.
<b>eError</b>	<a href="#">NBS.ERROR</a>	Статус работы ФБ (или имя ошибки).
<b>hServer</b>	NBS.CAA.HANDLE	Дескриптор сервера.

### 3.4.2. ФБ NBS.TCP\_Connection

Функциональный блок **NBS.TCP\_Connection** используется для обработки одного клиента, подключенного к TCP-серверу. ФБ принимает на вход дескриптор блока [NBS.TCP\\_Server](#) и возвращает дескриптор подключения, который используется для операций получения (ФБ [NBS.TCP\\_Read](#), [NBS.TCP\\_ReadBuffer](#)) и передачи данных (ФБ [NBS.TCP\\_Write](#), [NBS.TCP\\_WriteBuffer](#)).

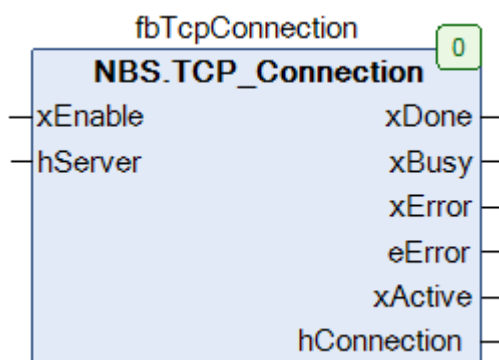


Рис. 3.8. Внешний вид ФБ **NBS.TCP\_Connection** на языке CFC

Название	Тип данных	Описание
<b>Входные переменные</b>		
<b>xEnable</b>	BOOL	Вход управления блоком. Пока он имеет значение <b>TRUE</b> – блок находится в работе.
<b>hServer</b>	NBS.CAA.HANDLE	Дескриптор сервера.
<b>Выходные переменные</b>		
<b>xDone</b>	BOOL	Флаг завершения работы блока. Принимает значение <b>TRUE</b> на один цикл.
<b>xBusy</b>	BOOL	Флаг «ФБ в процессе работы».
<b>xError</b>	BOOL	Флаг ошибки. Принимает значение <b>TRUE</b> при возникновении ошибки.
<b>eError</b>	<a href="#">NBS.ERROR</a>	Статус работы ФБ (или имя ошибки).
<b>xActive</b>	BOOL	Флаг активности соединения. Если он имеет значение <b>TRUE</b> – то к серверу подключен клиент.
<b>hConnection</b>	NBS.CAA.HANDLE	Дескриптор подключения.

### 3.4.3. ФБ NBS.TCP\_Client

Функциональный блок **NBS.TCP\_Connection** создает клиентский TCP-сокет и возвращает дескриптор подключения, который используется для операций получения (ФБ [NBS.TCP\\_Read](#), [NBS.TCP\\_ReadBuffer](#)) и передачи данных (ФБ [NBS.TCP\\_Write](#), [NBS.TCP\\_WriteBuffer](#)).

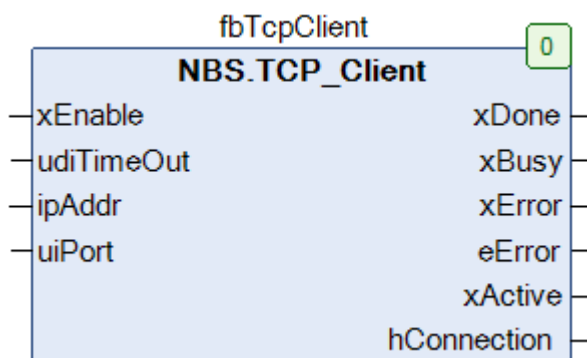


Рис. 3.9. Внешний вид ФБ **NBS.TCP\_Client** на языке CFC

Название	Тип данных	Описание
<b>Входные переменные</b>		
<b>xEnable</b>	BOOL	Вход управления блоком. Пока он имеет значение <b>TRUE</b> – блок находится в работе.
<b>udiTimeOut</b>	UDINT	Допустимое время операции (в мкс). Значение <b>0</b> означает, что время выполнения ФБ не ограничивается.
<b>ipAddr</b>	<a href="#">NBS.IP_ADDR</a>	IP-адрес сервера.
<b>uiPort</b>	UINT	Номер порта сервера.
<b>Выходные переменные</b>		
<b>xDone</b>	BOOL	Флаг завершения работы блока. Принимает значение <b>TRUE</b> на один цикл.
<b>xBusy</b>	BOOL	Флаг «ФБ в процессе работы».
<b>xError</b>	BOOL	Флаг ошибки. Принимает значение <b>TRUE</b> при возникновении ошибки.
<b>eError</b>	<a href="#">NBS.ERROR</a>	Статус работы ФБ (или имя ошибки).
<b>xActive</b>	BOOL	Флаг «сокет успешно открыт». Пока он имеет значение <b>TRUE</b> – сокет открыт, и его можно использовать.
<b>hConnection</b>	NBS.CAA.HANDLE	Дескриптор подключения.

### 3.4.4. ФБ NBS.TCP\_Read

Функциональный блок **NBS.TCP\_Read** используется для получения данных в заданном подключении. На вход блока подается дескриптор подключения с выхода ФБ [NBS.TCP\\_Connection](#) (если получатель данных – сервер) или [NBS.TCP\\_Client](#) (если получатель данных – клиент).

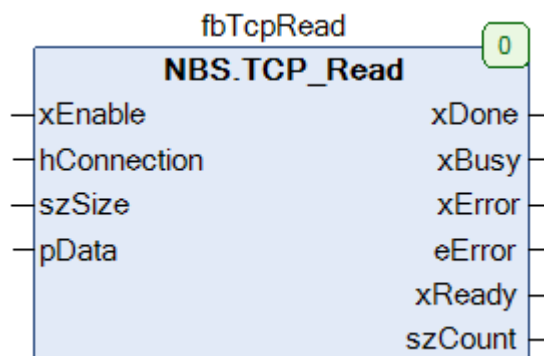


Рис. 3.10. Внешний вид ФБ **NBS.TCP\_Read** на языке CFC

Название	Тип данных	Описание
<b>Входные переменные</b>		
<b>xEnable</b>	BOOL	Вход управления блоком. Пока он имеет значение <b>TRUE</b> – блок находится в работе.
<b>hPeer</b>	NBS.CAA.HANDLE	Дескриптор сокета устройства.
<b>szSize</b>	NBS.CAA.SIZE	Максимально допустимый размер получаемых данных в байтах. Может быть указан помощью оператора <b>SIZEOF</b> .
<b>pData</b>	NBS.CAA.PVOID	Начальный адрес для размещения принятых данных. Может быть указан с помощью оператора <b>ADR</b> .
<b>Выходные переменные</b>		
<b>xDone</b>	BOOL	Флаг завершения работы блока. Принимает значение <b>TRUE</b> на один цикл.
<b>xBusy</b>	BOOL	Флаг «ФБ в процессе работы».
<b>xError</b>	BOOL	Флаг ошибки. Принимает значение <b>TRUE</b> при возникновении ошибки.
<b>eError</b>	<a href="#">NBS.ERROR</a>	Статус работы ФБ (или имя ошибки).
<b>xReady</b>	BOOL	Флаг «данные получены». Принимает значение <b>TRUE</b> на один цикл при получении нового пакета данных.
<b>ipFrom</b>	<a href="#">NBS.IP_ADDR</a>	IP-адрес отправителя.
<b>uiPortFrom</b>	UINT	Номер порта отправителя.
<b>szCount</b>	NBS.CAA.SIZE	Размер принятых данных в байтах.

### 3.4.5. ФБ NBS.TCP\_Write

Функциональный блок **NBS.TCP\_Write** используется для передачи данных в заданном подключении. На вход блока подается дескриптор подключения с выхода ФБ [NBS.TCP\\_Connection](#) (если получатель данных – сервер) или [NBS.TCP\\_Client](#) (если получатель данных – клиент).

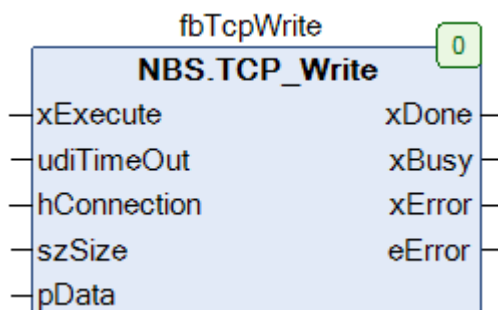


Рис. 3.11. Внешний вид ФБ **NBS.TCP\_Write** на языке **CFC**

Название	Тип данных	Описание
<b>Входные переменные</b>		
<b>xExecute</b>	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
<b>udiTimeOut</b>	UDINT	Допустимое время операции (в мкс). Значение <b>0</b> означает, что время выполнения ФБ не ограничивается.
<b>hPeer</b>	NBS.CAA.HANDLE	Дескриптор сокета устройства.
<b>ipAddr</b>	<a href="#">NBS.IP_ADDR</a>	IP-адрес получателя.
<b>uiPort</b>	UINT	Номер порта получателя.
<b>szSize</b>	NBS.CAA.SIZE	Размер отправляемых данных в байтах. Может быть указан помощью оператора <b>SIZEOF</b> .
<b>pData</b>	NBS.CAA.PVOID	Начальный адрес отправляемых данных. Может быть указан с помощью оператора <b>ADR</b> .
<b>Выходные переменные</b>		
<b>xDone</b>	BOOL	Флаг завершения работы блока. Принимает значение <b>TRUE</b> на один цикл.
<b>xBusy</b>	BOOL	Флаг «ФБ в процессе работы».
<b>xError</b>	BOOL	Флаг ошибки. Принимает значение <b>TRUE</b> при возникновении ошибки.
<b>eError</b>	<a href="#">NBS.ERROR</a>	Статус работы ФБ (или имя ошибки).

### 3.4.6. ФБ NBS.TCP\_ReadBuffer

Функциональный блок **NBS.TCP\_ReadBuffer** используется для получения данных в заданном подключении. На вход блока подается дескриптор подключения с выхода ФБ [NBS.TCP\\_Connection](#) (если получатель данных – сервер) или [NBS.TCP\\_Client](#) (если получатель данных – клиент).

В отличие от ФБ [NBS.TCP\\_Read](#) данный блок не копирует данные по указателю, а возвращает дескриптор буфера, в котором они были размещены. Для работы с буфером используется библиотека **CAA SegBufMan**. Этот способ является менее ресурсозатратным, но более сложным в использовании.

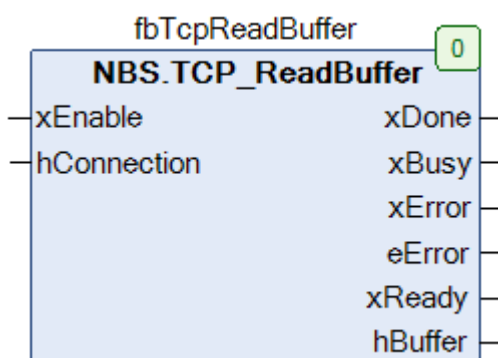


Рис. 3.12. Внешний вид ФБ **NBS.TCP\_ReadBuffer** на языке **CFC**

Название	Тип данных	Описание
<b>Входные переменные</b>		
<b>xEnable</b>	BOOL	Вход управления блоком. Пока он имеет значение <b>TRUE</b> – блок находится в работе.
<b>hPeer</b>	NBS.CAA.HANDLE	Дескриптор сокета устройства, полученный от ФБ NBS.UDP_PEER.
<b>Выходные переменные</b>		
<b>xDone</b>	BOOL	Флаг завершения работы блока. Принимает значение <b>TRUE</b> на один цикл.
<b>xBusy</b>	BOOL	Флаг «ФБ в процессе работы».
<b>xError</b>	BOOL	Флаг ошибки. Принимает значение <b>TRUE</b> при возникновении ошибки.
<b>eError</b>	<a href="#">NBS.ERROR</a>	Статус работы ФБ (или имя ошибки).
<b>xReady</b>	BOOL	Флаг «данные получены». Принимает значение <b>TRUE</b> на один цикл при получении нового пакета данных.
<b>ipFrom</b>	<a href="#">NBS.IP_ADDR</a>	IP-адрес отправителя.
<b>uiPortFrom</b>	UINT	Порт отправителя.
<b>hBuffer</b>	NBS.CAA.HANDLE	Дескриптор буфера принятых данных.

### 3.4.7. ФБ NBS.TCP\_WriteBuffer

Функциональный блок **NBS.TCP\_WriteBuffer** используется для отправки данных в заданном подключении. На вход блока подается дескриптор подключения с выхода ФБ [NBS.TCP\\_Connection](#) (если получатель данных – сервер) или [NBS.TCP\\_Client](#) (если получатель данных – клиент).

Функциональный блок **NBS.TCP\_WriteBuffer** используется для получения данных. В отличие от ФБ [NBS.TCP\\_Write](#) данный блок не копирует данные по указателю, а принимает на вход дескриптор буфера, в котором они размещены. Для работы с буфером используется библиотека **CAA SegBufMan**. Этот способ является менее ресурсозатратным, но более сложным в использовании.

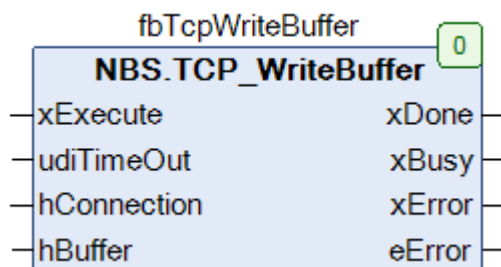


Рис. 3.13. Внешний вид ФБ **NBS.TCP\_WriteBuffer** на языке **CFC**

Название	Тип данных	Описание
<b>Входные переменные</b>		
<b>xExecute</b>	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
<b>udiTimeOut</b>	UDINT	Допустимое время операции (в мкс). Значение 0 означает, что время выполнения ФБ не ограничивается.
<b>hPeer</b>	NBS.CAA.HANDLE	Дескриптор сокета устройства, полученный от ФБ NBS.UDP_PEER.
<b>ipAddr</b>	<a href="#">NBS.IP_ADDR</a>	IP-адрес получателя.
<b>uiPort</b>	UINT	Номер порта получателя.
<b>hBuffer</b>	NBS.CAA.HANDLE	Дескриптор буфера отправляемых данных.
<b>Выходные переменные</b>		
<b>xDone</b>	BOOL	Флаг завершения работы блока. Принимает значение <b>TRUE</b> на один цикл.
<b>xBusy</b>	BOOL	Флаг «ФБ в процессе работы».
<b>xError</b>	BOOL	Флаг ошибки. Принимает значение <b>TRUE</b> при возникновении ошибки.
<b>eError</b>	<a href="#">NBS.ERROR</a>	Статус работы ФБ (или имя ошибки).

### 3.5. Дополнительные функции

#### 3.5.1. Функция NBS.IPSTRING\_TO\_UDINT

Функция **NBS.IPSTRING\_TO\_UDINT** конвертирует строковое представление IP-адреса в бинарное ('10.2.11.10' --> 16#0A02B00A).

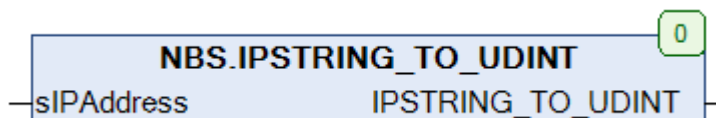


Рис. 3.14. Внешний вид функции **NBS.IPSTRING\_TO\_UDINT** на языке **CFC**

Название	Тип данных	Описание
<i>Входные переменные</i>		
stIPAddress	<a href="#">NBS.IP_ADDR</a>	Строковое представление IP-адреса.
<i>Выходные переменные</i>		
IPSTRING_TO_UDINT	UDINT	Бинарное представление IP-адреса.

#### 3.5.2. Функция NBS.IPSTRING\_TO\_UDINT

Функция **NBS.UDINT\_TO\_IPSTRING** конвертирует бинарное представление IP-адреса в строковое (16#0A02B00A --> '10.2.11.10').

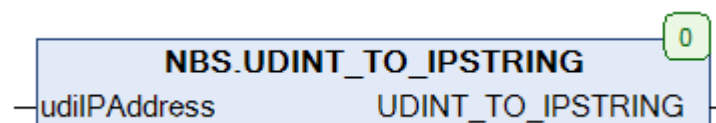


Рис. 3.15. Внешний вид функции **NBS. UDINT\_TO\_IPSTRING** на языке **CFC**

Название	Тип данных	Описание
<i>Входные переменные</i>		
udiIPAddress	UDINT	Бинарное представление IP-адреса.
<i>Выходные переменные</i>		
UDINT_TO_IPSTRING	<a href="#">NBS.IP_ADDR</a>	Строковое представление IP-адреса.



### 3.5.3. Функция NBS.IS\_MULTICAST\_GROUP

Функция **NBS.IS\_MULTICAST\_GROUP** возвращает **TRUE**, если указанный IP-адрес является адресом Multicast-группы.

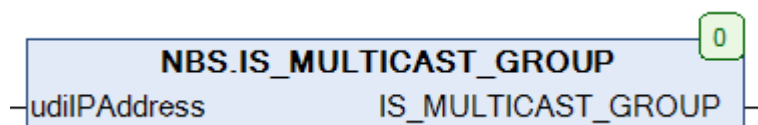


Рис. 3.16. Внешний вид функции **NBS.IS\_MULTICAST\_GROUP** на языке **CFC**

Название	Тип данных	Описание
<i>Входные переменные</i>		
<b>udilIPAdress</b>	UDINT	Бинарное представление IP-адреса.
<i>Выходные переменные</i>		
<b>IS_MULTICAST_GROUP</b>	BOOL	Флаг «Multicast-адрес».

## 4. Примеры работы с библиотекой CAA Net Base Services

### 4.1. Краткое описание примеров

В данной главе описываются принципы работы с библиотекой **CAA Net Base Services** на примере решения простейшей задачи:

1. Клиент отправляет на сервер строку данных;
2. Сервер получает эти данные и отправляет клиенту инвертированную строку.

Соответственно, при отправлении на сервер строки 'hello' клиент получит в ответ строку 'olleh'.

В [п. 4.2](#) приводится пример решения этой задачи с использованием протокола **UDP**, а в [п. 4.3](#) – с использованием протокола **TCP**.

Примеры созданы в среде **CODESYS 3.5 SP7 Patch4** и подразумевают запуск на виртуальном контроллере **CODESYS Control Win V3**, который входит в состав среды **CODESYS** и представляет собой программную эмуляцию реального контроллера, запускаемую на ПК с ОС семейства Windows. Для полноценной работы с примерами потребуются два виртуальных контроллера, запущенных на ПК, находящихся в одной локальной сети. Пользователь также может запустить примеры на других устройствах, изменив таргет-файл в проекте CODESYS (**ПКМ** на узел **Device** – **Обновить устройство**).

Каждый пример содержит два приложения (для сервера и клиента). Для загрузки в контроллер конкретного приложения необходимо нажать **ПКМ** на узел **Application** и выбрать команду **Установить активное приложение**.

Запуск виртуального контроллера выполняется с помощью иконки на **панели задач** Windows. При необходимости запустить несколько экземпляров виртуального контроллера на одном ПК следует использовать соответствующий ярлык в **меню Пуск (Все программы – 3S CODESYS – Codesys Control WinV3 – Codesys Control Win V3)**.

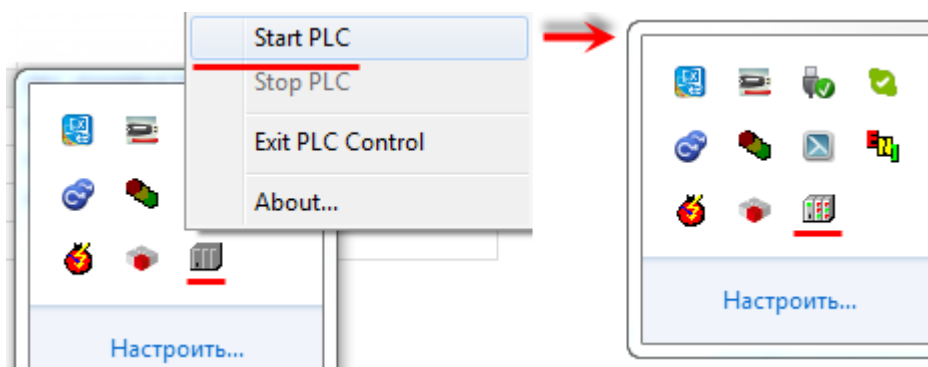


Рис. 4.1. Запуск виртуального контроллера

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

### 4.2.1. Основная информация

В данном примере рассматривается реализация UDP-сервера и UDP-клиента.

Решаемая задача описана в [п. 4.1](#).

Листинг примера приведен в приложениях [A1](#) (сервер) и [A2](#) (клиент).

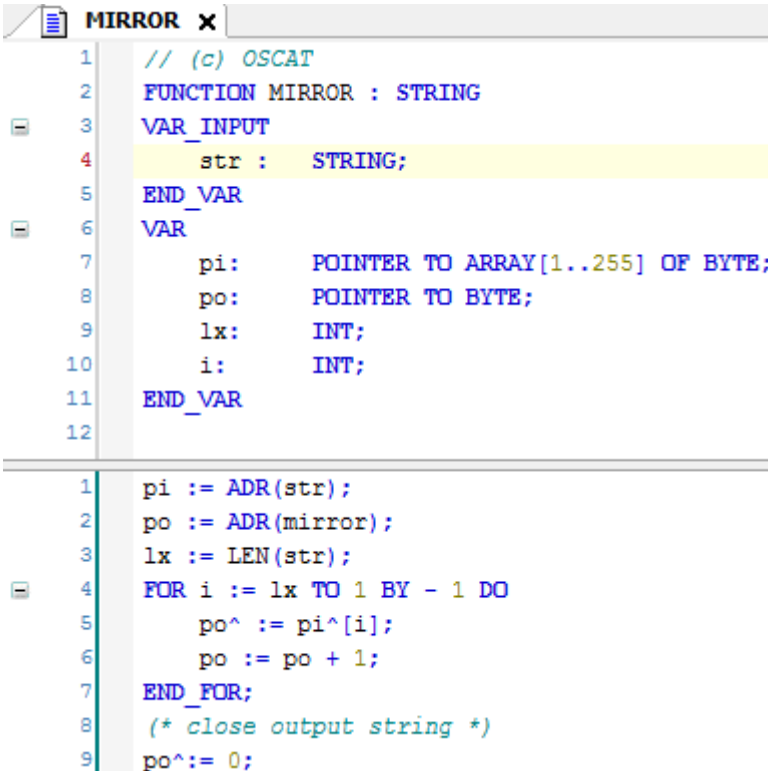
Пример доступен для скачивания: [Example UDP v3.projectarchive](#)

### 4.2.2. Реализация UDP-сервера

Еще раз напомним условие решаемой задачи – необходимо реализовать UDP-сервер, который будет получать от клиента строку данных, и возвращать ему инвертированную строку.

Начнем с самого простого – написания функции инверсии строки. Такая функция уже есть в свободно распространяемой библиотеке **OSCAT**. Библиотека доступна для скачивания на сайте [oscat.de](#), а также на сайте компании **Овен** в разделе [CODESYS V3/Библиотеки](#). Библиотека OSCAT имеет открытые исходные коды, поэтому во многих случаях удобнее копировать ее функции и ФБ в пользовательский проект (вместо добавления через **Менеджер библиотек**).

Функция инверсии строки называется **MIRROR**. Скопируем ее в свой проект и удалим константы **STRING\_LENGTH**, определяющие максимальную длину строк (чтобы не копировать из библиотеки дополнительные POU):



```
1 // (c) OSCAT
2 FUNCTION MIRROR : STRING
3 VAR_INPUT
4     str : STRING;
5 END_VAR
6 VAR
7     pi:   POINTER TO ARRAY[1..255] OF BYTE;
8     po:   POINTER TO BYTE;
9     lx:   INT;
10    i:    INT;
11 END_VAR
12
13 pi := ADR(str);
14 po := ADR(mirror);
15 lx := LEN(str);
16 FOR i := lx TO 1 BY - 1 DO
17     po^ := pi^[i];
18     po := po + 1;
19 END_FOR;
20 (* close output string *)
21 po^ := 0;
```

Рис. 4.2.1. Код функции **MIRROR**

Алгоритм работы сервера можно представить следующим образом:

1. Создание сокета;
2. Ожидание запроса от клиента и извлечение данных из полученного запроса;
3. Отправка ответа клиенту;
4. Возвращение на шаг 2.

Как можно заметить, этот алгоритм легко представить в виде последовательности шагов, выполняемых с помощью оператора **CASE**. В качестве меток оператора **CASE** можно использовать обычные числа (0, 1, 2 и т.д.) – но это затруднит чтение программы. Поэтому объявим перечисление **SERVER\_STATE** (**Application – Добавление объекта – DUT – Перечисление**), в котором свяжем номера шагов с символьными именами.

```
SERVER_STATE x
1 // шаг состояния сервера
2 {attribute 'strict'}
3 TYPE SERVER_STATE :
4 (
5     CREATE := 0,
6     LISTEN := 10,
7     SEND := 20
8 );
9 END_TYPE
10
```

Рис. 4.2.2. Объявление перечисления **SERVER\_STATE**

Объявим в программе **PLC\_PRG** следующие переменные:

```
PLC_PRG x
1 PROGRAM PLC_PRG
2 VAR
3     sClientString: STRING; // Строка, полученная от клиента
4     sInverseString: STRING; // Строка, отправляемая клиенту (инверсия полученной)
5
6     eState: SERVER_STATE; // Шаг состояния сервера
7
8     fbUdpPeer: NBS.UDP_Peer; // фБ создания UDP-пира
9     fbUdpRecieve: NBS.UDP_Receive; // фБ получения данных
10    fbUdpSend: NBS.UDP_Send; // фБ отправки данных
11
12    uiPortServer: UINT := 4711; // Порт сервера
13
14    stIpClient: NBS.IP_ADDR; // IP-адрес клиента
15    uiPortClient: UINT; // Порт клиента
16 END_VAR
17
```

Рис. 4.2.3. Объявление переменных программы **PLC\_PRG**

**Обратите внимание** на переменную **uiPortServer** – она определяет номер порта сервера.

Код программы выглядит следующим образом:

```
1  CASE eState OF
2
3
4  SERVER_STATE.CREATE:  // создаем UDP-пира на заданном порту
5
6      fbUdpPeer
7      (
8          xEnable      := TRUE,
9          ipAddr       := ,
10         uiPort        := uiPortServer,
11         ipMultiCast :=
12     );
13
14     IF fbUdpPeer.xActive THEN
15         eState      := SERVER_STATE.LISTEN;
16     ELSIF fbUdpPeer.xError THEN
17         ; // обработка ошибок
18     END_IF
19
20
21     SERVER_STATE.LISTEN:  // слушаем заданный порт, ожидая запрос от клиента
22
23
24     fbUdpRecieve
25     (
26         xEnable := TRUE,
27         hPeer   := fbUdpPeer.hPeer,
28         pData   := ADR(sClientString),
29         szSize  := SIZEOF(sClientString)
30     );
31
32     // если получены данные - извлекаем адрес и порт клиента,
33     // ...и подготавливаем ответ
34     IF fbUdpRecieve.xReady THEN
35         stIpClient   := fbUdpRecieve.ipFrom;
36         uiPortClient := fbUdpRecieve.uiPortFrom;
37
38         sInverseString := MIRROR(sClientString);
39
40         eState      := SERVER_STATE.SEND;
41     ELSIF fbUdpRecieve.xError THEN
42         ; // обработка ошибок
43     END_IF
44
45
46     SERVER_STATE.SEND:  // отправляем данные клиенту
47
48     fbUdpSend
49     (
50         xExecute := TRUE ,
51         hPeer    := fbUdpPeer.hPeer ,
52         ipAddr   := stIpClient ,
53         uiPort   := uiPortClient ,
54         pData    := ADR(sInverseString),
55         szSize   := SIZEOF(sInverseString)
56     );
57
58     // если данные успешно отправлены - продолжаем слушать порт, ожидая следующего запроса
59     IF fbUdpSend.xDone THEN
60         fbUdpSend(xExecute:=FALSE);
61         eState := SERVER_STATE.LISTEN;
62     ELSIF fbUdpSend.xError THEN
63         ; // обработка ошибок
64     END_IF
65
66     END_CASE
```

Рис. 4.2.4. Код программы PLC\_PRG

На шаге **CREATE** с помощью экземпляра ФБ [NBS.UDP\\_Peer](#) происходит создание серверного UDP-сокета на порту, номер которого определяется значением переменной **uiPortServer**. В данном примере используется порт **4711** – он был выбран произвольным образом. Результатом успешного создания сокета является получение его дескриптора (**hPeer**), который будет использоваться для приема и передачи данных на следующих шагах. Если сокет успешно создан (**xActive=TRUE**), то происходит переход на шаг **LISTEN**.

На шаге **LISTEN** с помощью экземпляра ФБ [NBS.UDP\\_Receive](#) происходит прослушивание порта и ожидание запроса от клиента. Если получен запрос (**xReady=TRUE**), то выполняются следующие операции:

- Копирование IP-адреса и номера порта клиента, отправившего запрос, в переменные **stIpClient** и **uiPortClient**;
- Инверсия (см. функцию **MIRROR**) полученной от клиента строки (**sClientString**) с записью результата в переменную **sInverseString**;
- Переход на шаг **SEND**.

На шаге **SEND** с помощью экземпляра ФБ [NBS.UDP\\_Send](#) происходит отправление ответа клиенту на заданный IP-адрес (**stIpClient**) и порт (**uiPortClient**). Ответ представляет собой строку **sInverseString**. После завершения операции (**xDone=TRUE**) происходит переход на шаг **LISTEN** для ожидания следующего запроса.

### 4.2.3. Реализация UDP-клиента

Теперь приступим к реализации UDP-клиента. Его задача – отправить запрос на сервер и получить ответ.

Как и в случае с сервером, алгоритм работы клиента представим в виде последовательности шагов, выполняемых с помощью оператора **CASE**. Для использования символьных имен в качестве меток оператора **CASE** объявим перечисление **CLIENT\_STATE** (**Application – Добавление объекта – DUT – Перечисление**), в котором свяжем номера шагов с символьными именами.

```
CLIENT_STATE x
1 // шаг состояния клиента
2 {attribute 'strict'}
3 TYPE CLIENT_STATE :
4 (
5     CREATE := 0,
6     WAITING := 10,
7     SEND := 20,
8     RECIEVE := 30
9 );
10 END_TYPE
```

Рис. 4.2.5. Объявление перечисления **CLIENT\_STATE**

В программе **PLC\_PRG** объявим следующие переменные:

```
PLC_PRG x
1 PROGRAM PLC_PRG
2 VAR
3     sClientString:   STRING      := 'Hello world'; // Строка, отправляемая клиентом
4     sInverseString: STRING;      // Строка, получаемая от сервера
5
6     eState:         CLIENT_STATE; // Шаг состояния клиента
7
8     fbUdpPeer:     NBS.UDP_Peer; // ФБ создания UDP-пира
9     fbUdpReceive: NBS.UDP_Receive; // ФБ получения данных
10    fbUdpSend:     NBS.UDP_Send; // ФБ отправки данных
11
12    stIpServer:    NBS.IP_ADDR := (sAddr:='10.2.5.229'); // IP-адрес сервера (измените его на адрес вашего сервера)
13
14    uiPortClient:  UINT      := 3000; // Порт клиента
15    uiPortServer: UINT      := 4711; // Порт сервера
16
17    xSend:        BOOL;
18
19    fbSendTrig:   R_TRIG; // Триггер записи
20    fbResponseTimeout: TON; // Таймер ожидания ответа
21 END_VAR
```

Рис. 4.2.6. Объявление переменных программы **PLC\_PRG**

**Обратите внимание** на переменные:

- **stIpServer** – содержит IP-адрес сервера, с которым работает клиент;
- **uiPortServer** – содержит номер порта сервера, с которым работает клиент;
- **uiPortClient** – содержит номер порта клиента.

Код программы будет выглядеть следующим образом:

```
1 CASE eState OF
2
3
4 CLIENT_STATE.CREATE: // создаем UDP-пир на заданном порту
5
6 fbUdpPeer
7 (
8     xEnable := TRUE,
9     ipAddr := ,
10    uiPort := uiPortClient,
11    ipMultiCast := ,
12 );
13
14 IF fbUdpPeer.xActive THEN
15     eState := CLIENT_STATE.WAITING;
16 ELSIF fbUdpPeer.xError THEN
17     ; // обработка ошибок
18 END_IF
19
20
21 CLIENT_STATE.WAITING: // ожидаем команды на запись
22
23 fbSendTrig (CLK:=xSend);
24
25 IF fbSendTrig.Q THEN
26     eState := CLIENT_STATE.SEND;
27 END_IF
28
29
30 CLIENT_STATE.SEND: // отправляем запрос серверу
31
32 fbUdpSend
33 (
34     xExecute := TRUE,
35     hPeer := fbUdpPeer.hPeer,
36     ipAddr := stIpServer,
37     uiPort := uiPortServer,
38     pData := ADR(sClientString),
39     szSize := SIZEOF(sClientString)
40 );
41
42
43 IF fbUdpSend.xDone THEN
44     fbUdpSend(xExecute:=FALSE);
45     fbResponseTimeout(IN:=FALSE);
46     eState := CLIENT_STATE.RECEIVE;
47 ELSIF fbUdpSend.xError THEN
48     ; // обработка ошибок
49 END_IF
50
51
52 CLIENT_STATE.RECEIVE: // получаем ответ от сервера
53
54 // запускаем таймер ожидания ответа
55 fbResponseTimeout(IN:=TRUE, PT:=T#1S);
56
57 fbUdpReceive
58 (
59     xEnable := TRUE,
60     hPeer := fbUdpPeer.hPeer,
61     pData := ADR(sInverseString),
62     szSize := SIZEOF(sInverseString)
63 );
64
65 // если данные получены - ожидаем следующей команды на запись
66 IF fbUdpReceive.xReady OR fbResponseTimeout.Q THEN
67     eState := CLIENT_STATE.WAITING;
68 ELSIF fbUdpReceive.xError THEN
69     ; // обработка ошибок
70 END_IF
71
72 END_CASE
```

Рис. 4.2.7. Код программы PLC\_PRG



На шаге **CREATE** с помощью экземпляра ФБ [NBS.UDP\\_Peer](#) происходит создание клиентского UDP-сокета на порту, номер которого определяется значением переменной **uiPortClient**. В данном примере используется порт **3000** – он был выбран произвольным образом. Результатом успешного создания сокета является получение его дескриптора (**hPeer**), который будет использоваться для приема и передачи данных на следующих шагах. Если сокет успешно создан (**xActive=TRUE**), то происходит переход на шаг **WAITING**.

На шаге **WAITING** происходит ожидание команды отправления запроса на сервер. Команда обрабатывается через триггер, чтобы предотвратить циклическую отправку запросов на сервер (запрос будет отправляться однократно по переднему фронту команды). После получения команды (**xSend=TRUE**) следует переход на шаг **SEND**.

На шаге **SEND** с помощью экземпляра ФБ [NBS.UDP\\_Send](#) происходит отправление запроса на заданный IP-адрес (**stIpServer**) и порт (**uiPortServer**). В рамках рассматриваемого примера UDP-сервер имеет IP-адрес **10.2.5.229** и порт **4711**. Номер порта соответствует порту, указанному при создании сокета на сервере (см. рис. 4.2.3). Запрос представляет собой строку **sClientString**. Если запрос успешно отправлен (**xDone=TRUE**), то происходит сброс таймера ожидания ответа и переход на шаг **RECEIVE**.

На шаге **RECEIVE** с помощью экземпляра ФБ [NBS.UDP\\_Receive](#) происходит получение ответа от сервера и запись его в строку **slInverseString**. Если ответ получен (**xReady=TRUE**) или время ожидания истекло (**fbResponseTimeout.Q=TRUE**), то выполняется переход на шаг **WAITING** для ожидания команды отправки следующего запроса.

Также создадим в проекте клиента простую визуализацию:

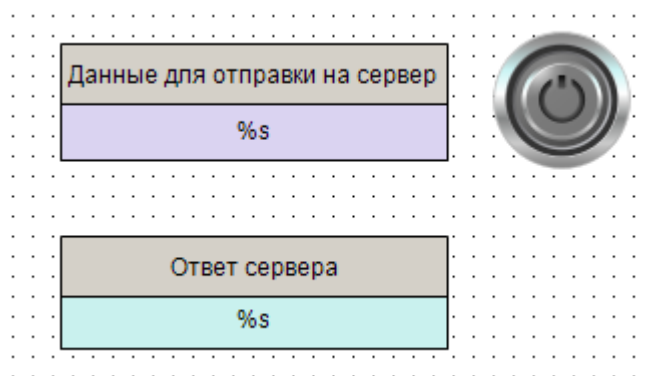


Рис. 4.2.8. Внешний вид визуализации клиента

К элементу **Данные для отправки на сервер** привязана переменная **sClientString** и настроена возможность ее изменения (вкладка **InputConfiguration** - **OnMouseClicked** – действие **Записать переменную**). К элементу **Ответ сервера** привязана переменная **slInverseString**. К переключателю (тип действия **Клавиша изображения**) привязана переменная **xSend** – она принимает значение **TRUE** при нажатии на элемент и **FALSE** – при его отпускании.

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

### 4.3.1. Основная информация

В данном примере рассматривается реализация TCP-сервера и TCP-клиента.

Решаемая задача описана в [п. 4.1](#).

Листинг примера приведен в приложениях [Б1](#) (сервер) и [Б2](#) (клиент).

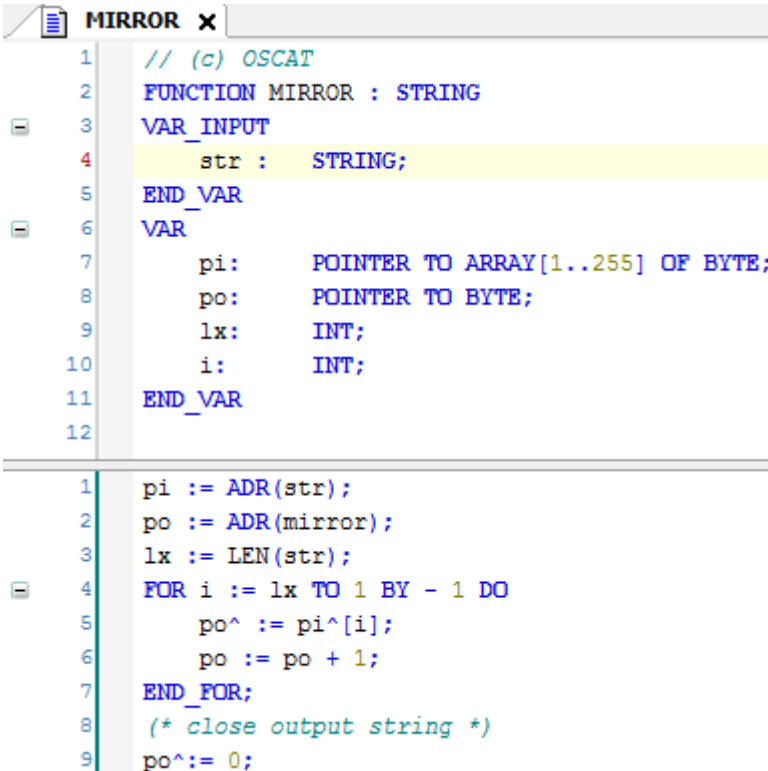
Пример доступен для скачивания: [Example TCP v3.projectarchive](#)

### 4.3.2. Реализация TCP-сервера

Еще раз напомним условие решаемой задачи – необходимо реализовать TCP-сервер, который будет получать от клиента строку данных, и возвращать ему инвертированную строку.

Начнем с самого простого – написания функции инверсии строки. Такая функция уже есть в свободно распространяемой библиотеке **OSCAT**. Библиотека доступна для скачивания на сайте [oscat.de](#), а также на сайте компании **Овен** в разделе [CODESYS V3/Библиотеки](#). Библиотека OSCAT имеет открытые исходные коды, поэтому во многих случаях удобнее копировать ее функции и ФБ в пользовательский проект (вместо добавления через **Менеджер библиотек**).

Функция инверсии строки называется **MIRROR**. Скопируем ее в свой проект функции и удалим константы **STRING\_LENGTH**, определяющие максимальную длину строк (чтобы не копировать из библиотеки дополнительные POU):



```
MIRROR x
1 // (c) OSCAT
2 FUNCTION MIRROR : STRING
3 VAR_INPUT
4   str : STRING;
5 END_VAR
6 VAR
7   pi:   POINTER TO ARRAY[1..255] OF BYTE;
8   po:   POINTER TO BYTE;
9   lx:   INT;
10  i:    INT;
11 END_VAR
12
1 pi := ADR(str);
2 po := ADR(mirror);
3 lx := LEN(str);
4 FOR i := lx TO 1 BY - 1 DO
5   po^ := pi^[i];
6   po := po + 1;
7 END_FOR;
8 (* close output string *)
9 po^ := 0;
```

Рис. 4.3.1. Код функции **MIRROR**

Алгоритм работы сервера можно представить следующим образом:

1. Создание сокета;
2. Создание обработчиков для клиентов;
3. Ожидание запросов от клиентов и извлечение данных из полученных запросов;
4. Отправка ответов клиентам;
5. Возвращение на шаг 3.

Как можно заметить, этот алгоритм легко представить в виде последовательности шагов, выполняемых с помощью оператора **CASE**. В качестве меток оператора **CASE** можно использовать обычные числа (0, 1, 2 и т.д.) – но это затруднит чтение программы. Поэтому объявим перечисление **SERVER\_STATE** (**Application – Добавление объекта – DUT – Перечисление**), в котором свяжем номера шагов с символьными именами.

```
SERVER_STATE x
1 // шаг состояния сервера
2 {attribute 'strict'}
3 TYPE SERVER_STATE :
4 (
5     CREATE := 0,
6     LISTEN := 10,
7     SEND := 20
8 );
9 END_TYPE
10
```

Рис. 4.3.2. Объявление перечисления **SERVER\_STATE**

TCP-сервер может одновременно обслуживать нескольких клиентов. Создадим структуру **CONNECTION**, которая содержит переменные и ФБ, необходимые для этого.

```
CONNECTION x
1 // структура параметров соединения
2 TYPE CONNECTION :
3 STRUCT
4     eState:          SERVER_STATE;          // Шаг состояния сервера
5
6     fbTcpConnection: NBS.TCP_Connection;    // ФБ обработки подключения
7     fbTcpRead:       NBS.TCP_Read;         // ФБ чтения данных
8     fbTcpWrite:      NBS.TCP_Write;       // ФБ записи данных
9
10    sClientString:   STRING;               // Строка, которую клиент отправляет на сервер
11    sInverseString:  STRING;              // Строка, которую клиент получает от сервер
12
13    fbAddClient:     R_TRIG;               // Триггер установки соединения
14 END_STRUCT
15 END_TYPE
```

Рис. 4.3.3. Объявление структуры **CONNECTION**

Объявим в программе **PLC\_PRG** следующие переменные:

```
1 PROGRAM PLC_PRG
2 VAR
3     fbTcpServer:      NBS.TCP_Server;           // ФБ TCP-сервера
4     astClients:      ARRAY [1..usiMaxConnections] OF CONNECTION; // Массив структур для обработки подключений
5
6     uiPortServer:    UINT           := 4711;
7
8     usiActiveClientCounter: USINT;           // Число подключенных клиентов
9
10    i:                INT;                // Счетчик для цикла
11 END_VAR
12
13 VAR CONSTANT
14     usiMaxConnections: USINT           := 3; // Максимальное число подключенных клиентов
15 END_VAR
```

Рис. 4.3.4. Объявление переменных программы **PLC\_PRG**

**Обратите внимание** на переменные:

- **uiPortServer** – содержит номер порта сервера;
- **usiMaxConnections** – определяет максимальное число клиентов, которые могут быть подключены к серверу.

Код программы выглядит следующим образом:

```

1 // создаем сервер на заданном порту
2 fbTcpServer
3 (
4     xEnable := TRUE,
5     ipAddr  := ,
6     uiPort  := uiPortServer
7 );
8
9 IF fbTcpServer.xError THEN
10     ; // обработка ошибок
11 END_IF
12
13 // создаем обработчики подключений для клиентов
14 FOR i:=1 TO usiMaxConnections DO
15     astClients[i].fbTcpConnection
16     (
17         xEnable:=fbTcpServer.xBusy,
18         hServer:=fbTcpServer.hServer
19     );
20
21 IF astClients[i].fbTcpConnection.xError THEN
22     ; // обработка ошибок
23 END_IF
24
25 // отслеживаем подключение клиента
26 astClients[i].fbAddClient(CLK:=astClients[i].fbTcpConnection.xActive);
27
28 // регистрируем подключение нового клиента
29 IF astClients[i].fbAddClient.Q THEN
30     usiActiveClientCounter := usiActiveClientCounter + 1;
31 END_IF
32
33 // регистрируем отключение одного из клиентов
34 IF astClients[i].fbTcpConnection.xDone THEN
35     usiActiveClientCounter := usiActiveClientCounter - 1;
36 END_IF
37
38
39 CASE astClients[i].eState OF
40
41
42     SERVER_STATE.CREATE: // проверяем, что подключился клиент
43
44     IF astClients[i].fbTcpConnection.xActive THEN
45         astClients[i].eState:=SERVER_STATE.LISTEN;
46     END_IF
47
48
49     SERVER_STATE.LISTEN: // получаем данные от клиента
50
51     astClients[i].fbTcpRead
52     (
53         xEnable := astClients[i].fbTcpConnection.xActive,
54         hConnection := astClients[i].fbTcpConnection.hConnection,
55         pData := ADR(astClients[i].sClientString),
56         szSize := SIZEOF(astClients[i].sClientString)
57     );
58
59 // если получен запрос от клиента - подготавливаем ответ
60 IF astClients[i].fbTcpRead.xReady THEN
61     astClients[i].sInverseString:=MIRROR(astClients[i].sClientString);
62     astClients[i].eState:=SERVER_STATE.SEND;
63 ELSIF astClients[i].fbTcpRead.xError THEN
64     ; // обработка ошибок
65 END_IF
66
67
68     SERVER_STATE.SEND: // отправляем ответ клиенту
69
70     astClients[i].fbTcpWrite
71     (
72         xExecute := TRUE,
73         hConnection := astClients[i].fbTcpConnection.hConnection,
74         pData := ADR(astClients[i].sInverseString),
75         szSize := SIZEOF(astClients[i].sInverseString)
76     );
77
78 // если ответ успешно отправлен - продолжаем слушать порт, ожидая следующего запроса
79 IF astClients[i].fbTcpWrite.xDone THEN
80     astClients[i].fbTcpWrite(xExecute:=FALSE);
81     astClients[i].eState:=SERVER_STATE.LISTEN;
82 ELSIF astClients[i].fbTcpWrite.xError THEN
83     ; // обработка ошибок
84 END_IF
85
86 END_CASE
87
88 END_FOR

```

Рис. 4.3.5. Код программы PLC\_PRG

В первых строках программы с помощью экземпляра ФБ [NBS.TCP\\_Server](#) происходит создание серверного TCP-сокета на порту, номер которого определяется значением переменной **uiPortServer**. В данном примере используется порт **4711** – он был выбран произвольным образом. Результатом успешного создания сокета является получение его дескриптора (**hServer**), который будет использоваться обработчиками клиентов.

В цикле **FOR** происходит последовательная обработка клиентов, подключенных к серверу (напомним, максимальное число клиентов определяется значением переменной **usiMaxConnections**). В процессе обработки выполняются следующие операции:

- вызов экземпляров ФБ [NBS.TCP\\_Connection](#) для обработки клиентов. При подключении клиента создается дескриптор (**hConnection**), который будут использовать ФБ получения ([NBS.TCP\\_Read](#)) и передачи данных ([NBS.TCP\\_Write](#));
- подсчет числа клиентов, подключенных к серверу (при подключении клиента выход **xActive** принимает значение **TRUE**, при отключении – на выходе **xDone** генерируется единичный импульс);
- обмен данными, разбитый на отдельные шаги через оператор **CASE**.

На шаге **CREATE** проверяется, подключен ли клиент к серверу. Если подключен (**xActive=TRUE**), то следует переход на шаг **LISTEN**.

На шаге **LISTEN** с помощью экземпляра ФБ [NBS.TCP\\_Read](#) происходит прослушивание порта и ожидание запроса от клиента. Если получен запрос (**xReady=TRUE**), то выполняются следующие операции:

- Инверсия (см. функцию **MIRROR**) полученной от клиента строки (**sClientString**) с записью результата в переменную **sInverseString**;
- Переход на шаг **SEND**.

На шаге **SEND** с помощью экземпляра ФБ [NBS.TCP\\_Write](#) происходит отправление ответа клиенту. Ответ представляет собой строку **sInverseString**. После завершения операции (**xDone=TRUE**) происходит переход на шаг **LISTEN** для ожидания следующего запроса.

### 4.3.3. Реализация TCP-клиента

Теперь приступим к реализации TCP-клиента. Его задача – отправить запрос на сервер и получить ответ.

Как и в случае с сервером, алгоритм работы клиента представим в виде последовательности шагов, выполняемых с помощью оператора **CASE**. Для использования символьных имен в качестве меток оператора **CASE** объявим перечисление **CLIENT\_STATE** (**Application – Добавление объекта – DUT – Перечисление**), в котором свяжем номера шагов с символьными именами.

```
CLIENT_STATE x
1 // шаг состояния клиента
2 {attribute 'strict'}
3 TYPE CLIENT_STATE :
4 (
5     CREATE := 0,
6     WAITING := 10,
7     SEND := 20,
8     RECIEVE := 30
9 );
10 END_TYPE
```

Рис. 4.3.6. Объявление перечисления **CLIENT\_STATE**

В программе **PLC\_PRG** объявим следующие переменные:

```
PLC_PRG x
1 PROGRAM PLC_PRG
2 VAR
3     sClientString:   STRING      := 'Hello world';      // Строка, отправляемая клиентом
4     sInverseString: STRING;      // Строка, получаемая от сервера
5
6     eState:         CLIENT_STATE; // Шаг состояния клиента
7
8     fbTcpClient:    NBS.TCP_Client; // ФБ создания TCP-клиента
9     fbTcpRead:      NBS.TCP_Read;   // ФБ чтения данных
10    fbTcpWrite:     NBS.TCP_Write;   // ФБ записи данных
11
12    stIpServer:     NBS.IP_ADDR      := (sAddr:='10.2.5.229'); // IP-адрес сервера
13    uiPortServer:   UINT             := 4711;           // Порт сервера
14
15    xSend:          BOOL;            // Команда отправки запроса на сервер
16
17    fbSendTrig:     R_TRIG;          // Триггер записи
18    fbResponseTimeout: TON;         // Таймер ожидания ответа
19 END VAR
```

Рис. 4.3.7. Объявление переменных программы **PLC\_PRG**

**Обратите внимание** на переменные:

- **stIpServer** – содержит IP-адрес сервера, с которым работает клиент;
- **uiPortServer** – содержит номер порта сервера, с которым работает клиент.

Код программы будет выглядеть следующим образом:

```
1  CASE eState OF
2
3  CLIENT_STATE.CREATE:  // создаем TCP-клиента
4
5      fbTcpClient
6      (
7          xEnable      := TRUE,
8          ipAddress    := stIpServer,
9          uiPort       := uiPortServer,
10         );
11
12     IF fbTcpClient.xActive THEN
13         eState := CLIENT_STATE.WAITING;
14     ELSIF fbTcpClient.xError THEN
15         ; // обработка ошибок
16     END_IF
17
18
19     CLIENT_STATE.WAITING:  // ожидаем команды на запись
20
21     fbSendTrig(CLK:=xSend);
22
23     IF fbSendTrig.Q THEN
24         eState := CLIENT_STATE.SEND;
25     END_IF
26
27
28     CLIENT_STATE.SEND:  // отправляем запрос серверу
29
30     fbTcpWrite
31     (
32         xExecute      := TRUE,
33         hConnection   := fbTcpClient.hConnection,
34         pData         := ADR(sClientString),
35         szSize        := SIZEOF(sClientString)
36     );
37
38     IF fbTcpWrite.xDone THEN
39         fbTcpWrite(xExecute:=FALSE);
40         fbResponseTimeout(IN:=FALSE);
41         eState := CLIENT_STATE.RECEIVE;
42     ELSIF fbTcpWrite.xError THEN
43         ; // обработка ошибок
44     END_IF
45
46
47     CLIENT_STATE.RECEIVE:  // получаем ответ от сервера
48
49     // запускаем таймер ожидания ответа
50     fbResponseTimeout(IN:=TRUE, PT:=T#1S);
51
52     fbTcpRead
53     (
54         xEnable      := TRUE,
55         hConnection  := fbTcpClient.hConnection,
56         pData        := ADR(sInverseString),
57         szSize       := SIZEOF(sInverseString)
58     );
59
60     // если данные получены - ожидаем следующей команды на запись
61     IF fbTcpRead.xReady OR fbResponseTimeout.Q THEN
62         eState := CLIENT_STATE.WAITING;
63     ELSIF fbTcpRead.xError THEN
64         ; // обработка ошибок
65     END_IF
66 END CASE
```

Рис. 4.3.8. Код программы PLC\_PRG



На шаге **CREATE** с помощью экземпляра ФБ [NBS.TCP\\_Client](#) происходит создание клиентского TCP-сокета для работы с сервером, который имеет IP-адрес **stIpServer** и номер порта **uiPortServer**. В данном примере используется порт **3000** – он был выбран произвольным образом. Результатом успешного создания сокета является получение дескриптора соединения (**hConnection**), который будет использоваться для получения и передачи данных на следующих шагах. Если сокет успешно создан (**xActive=TRUE**), то происходит переход на шаг **WAITING**.

На шаге **WAITING** происходит ожидание команды отправления запроса на сервер. Команда обрабатывается через триггер, чтобы предотвратить циклическую отправку запросов на сервер (запрос будет отправляться однократно по переднему фронту команды). После получения команды (**xSend=TRUE**) следует переход на шаг **SEND**.

На шаге **SEND** с помощью экземпляра ФБ [NBS.TCP\\_Write](#) происходит отправление запроса серверу. Запрос представляет собой строку **sClientString**. Если запрос успешно отправлен (**xDone=TRUE**), то происходит переход на шаг **RECEIVE**.

На шаге **RECEIVE** с помощью экземпляра ФБ [NBS.TCP\\_Read](#) происходит получение ответа от сервера и запись его в строку **sInverseString**. Если ответ получен (**xReady=TRUE**) или время ожидания истекло (**fbResponseTimeout.Q=TRUE**), то выполняется переход на шаг **WAITING** для ожидания команды отправки следующего запроса.

Также создадим в проекте клиента простую визуализацию:

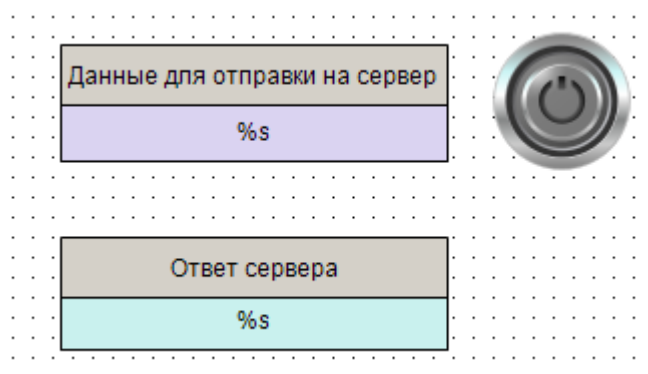


Рис. 4.3.9. Внешний вид визуализации клиента

К элементу **Данные для отправки на сервер** привязана переменная **sClientString** и настроена возможность ее изменения (вкладка **InputConfiguration** - **OnClick** – действие **Записать переменную**). К элементу **Ответ сервера** привязана переменная **sInverseString**. К переключателю (тип действия **Клавиша изображения**) привязана переменная **xSend** – она принимает значение **TRUE** при нажатии на элемент и **FALSE** – при его отпускании.

#### 4.4. Работа с примером

1. Каждый пример содержит два приложения – **Server** и **Client**.

В приложениях отредактируйте нужным образом:

- IP-адрес сервера (переменная **stIpServer** в приложении **Client**);
- Номер порта сервера (переменная **uiPortServer** в обоих приложениях);
- Номер порта клиента (переменная **uiPortClient** в приложении **Client** – *только для примера UDP*).

Запустите два виртуальных контроллера на ПК, подключенных к одной локальной сети (можно использовать один ПК с несколькими сетевыми картами). Для загрузки в контроллер конкретного приложения необходимо нажать **ПКМ** на узел **Application** и выбрать команду **Установить активное приложение**. Предварительно рекомендуется выполнить команду **Сброс заводской** из меню **Онлайн**.

При необходимости вы можете изменить таргет-файлы приложений, чтобы запустить их на нужных вам устройствах (**ПКМ** на узел **Device** – **Обновить устройство**).

2. В приложении **Client** перейдите на страницу визуализации.

Введите строку данных, которая будет отправлена на сервер, и нажмите кнопку.

В поле **Ответ сервера** должна появиться инвертированная строка.

Если этого не происходит – следует проверить корректность сетевых настроек устройств, на которых запускаются проекты, и уточнить особенности настроек сети (например, на маршрутизаторах могут быть заблокированы какие-то порты).

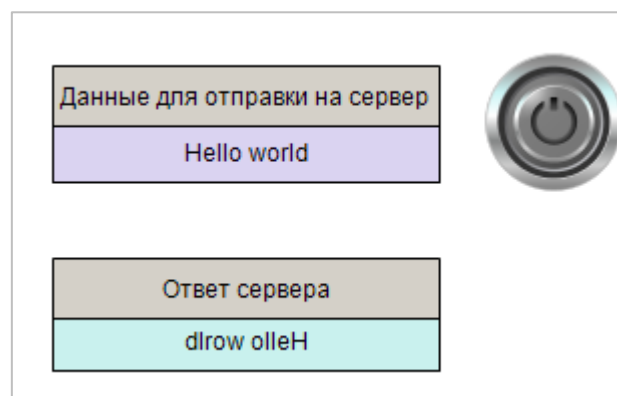


Рис. 4.4.1. Работа с примером

#### 4.5. Рекомендации и замечания

Ниже перечислены основные тезисы и рекомендации по разработке программ, работающих с сокетами, использованные в данном документе.

- ФБ и программы, реализующие обмен, разбиваются на шаги, которые выполняются через оператор **CASE**.
- Для того чтобы сделать прозрачным переходы между шагами, можно использовать **перечисления**.
- Переход к следующему шагу должен происходить только после окончания предыдущего. Контроль окончания шага, в частности, может осуществляться с помощью выходов соответствующих ФБ (**xDone**, **xActive**, **xReady** и т.д.).

Следует также отметить ряд моментов, оставшихся за пределами примеров документа:

- В рамках примера рассматривается обмен данными между сервером и клиентом с помощью обычных текстовых строк. Для реализации конкретного протокола потребуется его спецификация, описывающая форматы и последовательности запросов и ответов.
- В большинстве случаев требуется тщательная обработка ошибок. Контролируйте выходы **xError** и **eError** соответствующих ФБ. См. описание кодов ошибок в [п. 3.2.2](#).

## Приложение. Листинг примера

### A1. UDP-сервер

#### A.1.1. Перечисление SERVER\_STATE

```
// шаг состояния сервера
{attribute 'strict'}
TYPE SERVER_STATE :
(
    CREATE      := 0,
    LISTEN     := 10,
    SEND       := 20
);
END_TYPE
```

#### A.1.2. Функция MIRROR

```
// (c) OSCAT
FUNCTION MIRROR : STRING
VAR_INPUT
    str : STRING;
END_VAR
VAR
    pi: POINTER TO ARRAY [1..255] OF BYTE;
    po: POINTER TO BYTE;
    lx: INT;
    i: INT;
END_VAR
```

```
(*
version 1.1    29. mar. 2008
programmer    hugo
tested by     tobias
```

this function reverses an input string.

```
*)
```

```

pi    :=    ADR(str);
po    :=    ADR(mirror);
lx    :=    LEN(str);

FOR i := lx TO 1 BY -1 DO
    po^ := pi^[i];
    po  := po + 1;
END_FOR;
(* close output string *)
po^:= 0;

```

```

(* revision histroy
hm    4. feb. 2008    rev 1.0
      original release

hm    29. mar. 2008  rev 1.1
      changed STRING to STRING(STRING_LENGTH)
*)

```

### A.1.3. Программа PLC\_PRG

```

PROGRAM PLC_PRG
VAR
    sClientString: STRING;      // Строка, полученная от клиента
    sInverseString: STRING;    // Строка, отправляемая клиенту (инверсия полученной)

    eState:        SERVER_STATE; // Шаг состояния сервера

    fbUdpPeer:    NBS.UDP_Peer;  // ФБ создания UDP-пира
    fbUdpReceive: NBS.UDP_Receive; // ФБ получения данных
    fbUdpSend:    NBS.UDP_Send;  // ФБ отправки данных

    uiPortServer:  UINT := 4711; // Порт сервера

    stIpClient:   NBS.IP_ADDR;   // IP-адрес клиента
    uiPortClient:  UINT;         // Порт клиента
END_VAR

```

## CASE eState OF

```
SERVER_STATE.CREATE:      // создаем UDP-пира на заданном порту

    fbUdpPeer
    (
        xEnable      := TRUE,
        ipAddr       := ,
        uiPort       := uiPortServer,
        ipMultiCast  :=
    );

    IF fbUdpPeer.xActive THEN
        eState := SERVER_STATE.LISTEN;
    ELSIF fbUdpPeer.xError THEN
        ; // обработка ошибок
    END_IF

SERVER_STATE.LISTEN: // слушаем заданный порт, ожидая запрос от клиента

    fbUdpReceive
    (
        xEnable:=      TRUE,
        hPeer  :=      fbUdpPeer.hPeer,
        pData  :=      ADR(sClientString),
        szSize :=      SIZEOF(sClientString)
    );

    // если получены данные - извлекаем адрес и порт клиента,
    // ...и подготавливаем ответ
    IF fbUdpReceive.xReady THEN
        stIpClient      :=      fbUdpReceive.ipFrom;
        uiPortClient    :=      fbUdpReceive.uiPortFrom;

        sInverseString :=      MIRROR(sClientString);

        eState          :=      SERVER_STATE.SEND;
    ELSIF fbUdpReceive.xError THEN
        ; // обработка ошибок
    END_IF
```

```
SERVER_STATE.SEND: // отправляем данные клиенту
```

```
fbUdpSend  
(  
    xExecute      := TRUE ,  
    hPeer         := fbUdpPeer.hPeer ,  
    ipAddr        := stIpClient ,  
    uiPort        := uiPortClient ,  
    pData         := ADR(sInverseString),  
    szSize        := SIZEOF(sInverseString)  
);
```

```
// если данные успешно отправлены –  
// ...продолжаем слушать порт, ожидая следующего запроса
```

```
IF fbUdpSend.xDone THEN  
    fbUdpSend(xExecute:=FALSE);  
    eState := SERVER_STATE.LISTEN;  
ELSIF fbUdpSend.xError THEN  
    ; // обработка ошибок  
END_IF
```

```
END_CASE
```

## A2. UDP-клиент

### A.2.1. Перечисление CLIENT\_STATE

```
// шаг состояния клиента
{attribute 'strict'}
TYPE CLIENT_STATE :
(
    CREATE      := 0,
    WAITING     := 10,
    SEND        := 20,
    RECEIVE     := 30
);
END_TYPE
```

### A.2.2. Программа PLC\_PRG

```
PROGRAM PLC_PRG
VAR
    sClientString: STRING := 'Hello world';           // Строка, отправляемая клиентом
    sInverseString: STRING;                           // Строка, получаемая от сервера

    eState: CLIENT_STATE;                             // Шаг состояния клиента

    fbUdpPeer: NBS.UDP_Peer;                           // ФБ создания UDP-пира
    fbUdpReceive: NBS.UDP_Receive;                     // ФБ получения данных
    fbUdpSend: NBS.UDP_Send;                           // ФБ отправки данных

    // IP-адрес сервера (измените его на адрес вашего сервера)
    stIpServer: NBS.IP_ADDR := (sAddr := '10.2.5.229');

    uiPortClient: UINT := 3000;                       // Порт клиента
    uiPortServer: UINT := 4711;                       // Порт сервера

    xSend: BOOL;

    fbSendTrig: R_TRIG;                               // Триггер записи
    fbResponseTimeout: TON;                           // Таймер ожидания ответа
END_VAR
```



## CASE eState OF

CLIENT\_STATE.CREATE:       // создаем UDP-пира на заданном порту

```
fbUdpPeer
(
    xEnable      := TRUE,
    ipAddr       := ,
    uiPort       := uiPortClient,
    ipMultiCast  := ,
);

IF fbUdpPeer.xActive THEN
    eState := CLIENT_STATE.WAITING;
ELSIF fbUdpPeer.xError THEN
    ; // обработка ошибок
END_IF
```

CLIENT\_STATE.WAITING:       // ожидаем команды на запись

```
fbSendTrig(CLK:=xSend);

IF fbSendTrig.Q THEN
    eState := CLIENT_STATE.SEND;
END_IF
```

CLIENT\_STATE.SEND:         // отправляем запрос серверу

```
fbUdpSend
(
    xExecute      := TRUE,
    hPeer         := fbUdpPeer.hPeer,
    ipAddr        := stIpServer,
    uiPort        := uiPortServer,
    pData         := ADR(sClientString),
    szSize        := SIZEOF(sClientString)
);

IF fbUdpSend.xDone THEN
    fbUdpSend(xExecute:=FALSE);
    fbResponseTimeout(IN:= FALSE);
    eState := CLIENT_STATE.RECEIVE;
ELSIF fbUdpSend.xError THEN
    ; // обработка ошибок
END_IF
```

```

CLIENT_STATE.RECEIVE:      // получаем ответ от сервера

// запускаем таймер ожидания ответа
fbResponseTimeout(IN:= TRUE, PT:= T#1s);

fbUdpReceive
(
    xEnable:=      TRUE,
    hPeer  :=      fbUdpPeer.hPeer,
    pData  :=      ADR(sInverseString),
    szSize :=      SIZEOF(sInverseString)
);

// если данные получены - ожидаем следующей команды на запись
IF fbUdpReceive.xReady OR fbResponseTimeout.Q THEN
    eState := CLIENT_STATE.WAITING;
ELSIF fbUdpReceive.xError THEN
    ; // обработка ошибок
END_IF
END_CASE

```

## Б1. TCP-сервер

### Б.1.1. Перечисление SERVER\_STATE

```
// шаг состояния сервера
{attribute 'strict'}
TYPE SERVER_STATE :
(
    CREATE      :=    0,
    LISTEN      :=    10,
    SEND        :=    20
);
END_TYPE
```

### Б.1.2. Структура CONNECTION

```
// структура параметров соединения
TYPE CONNECTION :
STRUCT
    eState:          SERVER_STATE;          // Шаг состояния сервера

    fbTcpConnection: NBS.TCP_Connection;    // ФБ обработки подключения
    fbTcpRead:       NBS.TCP_Read;         // ФБ чтения данных
    fbTcpWrite:      NBS.TCP_Write;        // ФБ записи данных

    sClientString:  STRING;               // Строка, которую клиент отправляет на сервер
    sInverseString: STRING;               // Строка, которую клиент получает от сервера

    fbAddClient:    R_TRIG;               // Триггер установки соединения
END_STRUCT
END_TYPE
```

### Б.1.3. Функция MIRROR

```
// (c) OSCAT
FUNCTION MIRROR : STRING
VAR_INPUT
    str :    STRING;
END_VAR
VAR
    pi:     POINTER TO ARRAY [1..255] OF BYTE;
    po:     POINTER TO BYTE;
    lx:     INT;
    i:      INT;
END_VAR
```

```
(*
version 1.1    29. mar. 2008
programmer    hugo
tested by     tobias
```

this function reverses an input string.

\*)

```
pi    :=    ADR(str);
po    :=    ADR(mirror);
lx    :=    LEN(str);

FOR i := lx TO 1 BY -1 DO
    po^ := pi^[i];
    po  := po + 1;
END_FOR;
(* close output string *)
po^:= 0;
```

```
(* revision history
hm    4. feb. 2008    rev 1.0
        original release

hm    29. mar. 2008    rev 1.1
        changed STRING to STRING(STRING_LENGTH)
*)
```

#### Б.1.4. Программа PLC\_PRG

```
PROGRAM PLC_PRG
VAR
    fbTcpServer:  NBS.TCP_Server;      // ФБ TCP-сервера

    // Массив структур для обработки подключений
    astClients:   ARRAY [1..usiMaxConnections] OF CONNECTION;
    uiPortServer: UINT                  := 4711;

    usiActiveClientCounter:  USINT;    // Число подключенных клиентов

    i:               INT;             // Счетчик для цикла
END_VAR

VAR CONSTANT
    // Максимальное число подключенных клиентов
    usiMaxConnections:  USINT := 3;
END_VAR
```

```

// создаем сервер на заданном порту
fbTcpServer
(
    xEnable:= TRUE,
    ipAddr := ,
    uiPort := uiPortServer
);

IF fbTcpServer.xError THEN
    ; // обработка ошибок
END_IF

// создаем обработчики подключений для клиентов
FOR i:=1 TO usiMaxConnections DO
    astClients[i].fbTcpConnection
    (
        xEnable:=fbTcpServer.xBusy,
        hServer:=fbTcpServer.hServer
    );

    IF astClients[i].fbTcpConnection.xError THEN
        ; // обработка ошибок
    END_IF

    // отслеживаем подключение клиента
    astClients[i].fbAddClient(CLK:=astClients[i].fbTcpConnection.xActive);

    // регистрируем подключение нового клиента
    IF astClients[i].fbAddClient.Q THEN
        usiActiveClientCounter := usiActiveClientCounter + 1;
    END_IF

    // регистрируем отключение одного из клиентов
    IF astClients[i].fbTcpConnection.xDone THEN
        usiActiveClientCounter := usiActiveClientCounter - 1;
    END_IF

```

```
CASE astClients[i].eState OF
```

```
SERVER_STATE.CREATE:      // проверяем, что подключился клиент
```

```
    IF astClients[i].fbTcpConnection.xActive THEN
        astClients[i].eState:=SERVER_STATE.LISTEN;
    END_IF
```

```
SERVER_STATE.LISTEN: // получаем данные от клиента
```

```
    astClients[i].fbTcpRead
    (
        xEnable      :=    astClients[i].fbTcpConnection.xActive,
        hConnection  :=    astClients[i].fbTcpConnection.hConnection,
        pData        :=    ADR(astClients[i].sClientString),
        szSize       :=    SIZEOF(astClients[i].sClientString)
    );
```

```
    // если получен запрос от клиента - подготавливаем ответ
```

```
    IF astClients[i].fbTcpRead.xReady THEN
        astClients[i].sInverseString:=MIRROR(astClients[i].sClientString);
        astClients[i].eState:=SERVER_STATE.SEND;
    ELSIF astClients[i].fbTcpRead.xError THEN
        ; // обработка ошибок
    END_IF
```

```
SERVER_STATE.SEND: // отправляем ответ клиенту
```

```
    astClients[i].fbTcpWrite
    (
        xExecute     :=    TRUE,
        hConnection  :=    astClients[i].fbTcpConnection.hConnection,
        pData        :=    ADR(astClients[i].sInverseString),
        szSize       :=    SIZEOF(astClients[i].sInverseString)
    );
```

```
    // если ответ успешно отправлен - продолжаем слушать порт, ожидая следующего запроса
```

```
    IF astClients[i].fbTcpWrite.xDone THEN
        astClients[i].fbTcpWrite(xExecute:=FALSE);
        astClients[i].eState:=SERVER_STATE.LISTEN;
    ELSIF astClients[i].fbTcpWrite.xError THEN
        ; // обработка ошибок
    END_IF
```

```
END_CASE
```

```
END_FOR
```

## Б2. TCP-клиент

### Б.2.1. Перечисление CLIENT\_STATE

```
// шаг состояния клиента
{attribute 'strict'}
TYPE CLIENT_STATE :
(
    CREATE      := 0,
    WAITING     := 10,
    SEND        := 20,
    RECEIVE     := 30
);
END_TYPE
```

### Б.2.2. Программа PLC\_PRG

```
PROGRAM PLC_PRG
VAR
    sClientString: STRING := 'Hello world';           // Строка, отправляемая клиентом
    sInverseString: STRING;                          // Строка, получаемая от сервера

    eState: CLIENT_STATE;                            // Шаг состояния клиента

    fbTcpClient: NBS.TCP_Client;                     // ФБ создания TCP-клиента
    fbTcpRead: NBS.TCP_Read;                         // ФБ чтения данных
    fbTcpWrite: NBS.TCP_Write;                       // ФБ записи данных
    // IP-адрес сервера (измените его на адрес вашего сервера)
    stIpServer: NBS.IP_ADDR := (sAddr := '10.2.5.229');

    uiPortClient: UINT := 3000;                      // Порт клиента
    uiPortServer: UINT := 4711;                     // Порт сервера

    xSend: BOOL;

    fbSendTrig: R_TRIG;                             // Триггер записи
    fbResponseTimeout: TON;                         // Таймер ожидания ответа
END_VAR
```



## CASE eState OF

CLIENT\_STATE.CREATE:       // создаем TCP-клиента

```
fbTcpClient
(
    xEnable      := TRUE,
    ipAddr       := stIpServer,
    uiPort       := uiPortServer,
);

IF fbTcpClient.xActive THEN
    eState := CLIENT_STATE.WAITING;
ELSIF fbTcpClient.xError THEN
    ; // обработка ошибок
END_IF
```

CLIENT\_STATE.WAITING:       // ожидаем команды на запись

```
fbSendTrig(CLK:=xSend);

IF fbSendTrig.Q THEN
    eState := CLIENT_STATE.SEND;
END_IF
```

CLIENT\_STATE.SEND:         // отправляем запрос серверу

```
fbTcpWrite
(
    xExecute      := TRUE,
    hConnection   := fbTcpClient.hConnection,
    pData         := ADR(sClientString),
    szSize        := SIZEOF(sClientString)
);

IF fbTcpWrite.xDone THEN
    fbTcpWrite(xExecute:=FALSE);
    fbResponseTimeout(IN:= FALSE);
    eState := CLIENT_STATE.RECEIVE;
ELSIF fbTcpWrite.xError THEN
    ; // обработка ошибок
END_IF
```

```

CLIENT_STATE.RECEIVE:      // получаем ответ от сервера

// запускаем таймер ожидания ответа
fbResponseTimeout(IN:= TRUE, PT:= T#1s);

fbTcpRead
(
    xEnable      :=    TRUE,
    hConnection  :=    fbTcpClient.hConnection,
    pData        :=    ADR(sInverseString),
    szSize       :=    SIZEOF(sInverseString)
);

// если данные получены - ожидаем следующей команды на запись
IF fbTcpRead.xReady OR fbResponseTimeout.Q THEN
    eState      :=    CLIENT_STATE.WAITING;
ELSIF fbTcpRead.xError THEN
    ; // обработка ошибок
END_IF
END_CASE

```