УТВЕРЖДАЮ

Директор ООО «Континент ЭТС»

_____ А.А. Алексеев

«04» 12 2024 г.

КОМПЛЕКС ТЕХНИЧЕСКИХ СРЕДСТВ ИЗМЕРИТЕЛЬНЫЙ

UZOLA PRO100/ UZOLA PRO300

Руководство программиста

МПВР.421457.000И3

СОГЛАСОВАНО

Начальник отдела АСУТП

_____ А.Н. Вовк

«04» 12 2024 г.

Подп. и дата | Инв. № дубл. | Взам. инв. № | Подп. и дата | Инв. № подл.

Перв. примен.

Справ. №

Подп. и дата

Инв. № дубл.

Взам. инв. №

Подп. и дата

Инв. № подл.

Содержание

Обозначения и сокращения

МП – модуль процессорный;

МВВ – модуль ввода-вывода;

КТСИ – комплекс технических средств измерительный;

Введение

Настоящее руководство программиста содержит информацию, необходимую для начала работы в среде CODESYS V3.5. Процесс ознакомления со средой CODESYS V3.5 описан применительно к КТСИ Пролог/ Прогресс.

1 Назначение и условия применения

1.1 Назначение

В данный момент (2024 г.) компания UZOLA выпускает две линейки КТСИ программируемых в среде CODESYS V3.5.

Таблица 1 - Список КТСИ UZOLA, программируемых в CODESYS V3.5

Линейка КТСИ	Требуемая для программирования версия CODESYS (для заводских прошивок)
UZOLA PRO100	3.5
UZOLA PRO300	3.5

КТСИ имеет блочно-модульную структуру, включающую в себя модули различного типа. Модули являются основным элементом КТСИ и, в зависимости от типа, выполняют ту или иную функцию.

1.2 Меры безопасности при эксплуатации

1.2.1 К эксплуатации КТСИ допускаются лица, изучившие руководство по эксплуатации изделия и прошедшие инструктаж о соблюдении правил безопасности при работе с электроустановками.

1.2.2 Персонал, выполняющий работы по эксплуатации КТСИ, должен иметь квалификацию не ниже квалификационной группы III по ПТЭЭП.

1.3 Подготовка изделия к использованию

1.3.1 Перед началом проведения работ по эксплуатации КТСИ следует ознакомиться с эксплуатационной документацией на изделие, требованиями безопасности, а также другими нормативными и эксплуатационными документами и строго ими руководствоваться.

1.3.2 Перед началом проведения работ следует проверить комплектность КТСИ и провести его внешний осмотр в следующей последовательности:

- проверить целостность соединительных кабелей и проводников;
- проверить состояние соединителей;
- осмотреть поверхности и разъемы корпусов модулей.

При этом, наружные поверхности не должны иметь вмятин, трещин, царапин, дефектов покрытия и загрязнений, влияющих на работу модулей.

1.4 Подключение модулей КТСИ

1.4.1 Подключение модулей КТСИ производить, руководствуясь положениями п.2.4 документа «Руководство по эксплуатации» на КТСИ.

1.4.2 Подать напряжение питания 24 В постоянного тока на общую электрическую шину. Питание подаётся через процессорный модуль (рисунки 1, 2).

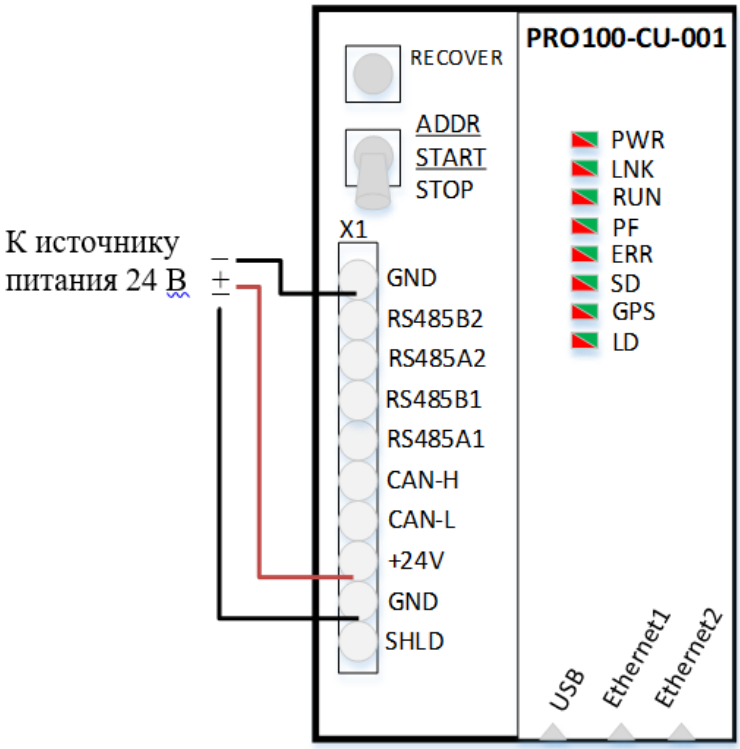


Рисунок 1 – Подача напряжения питания на общую электрическую шину через процессорный модуль UZOLA PRO100

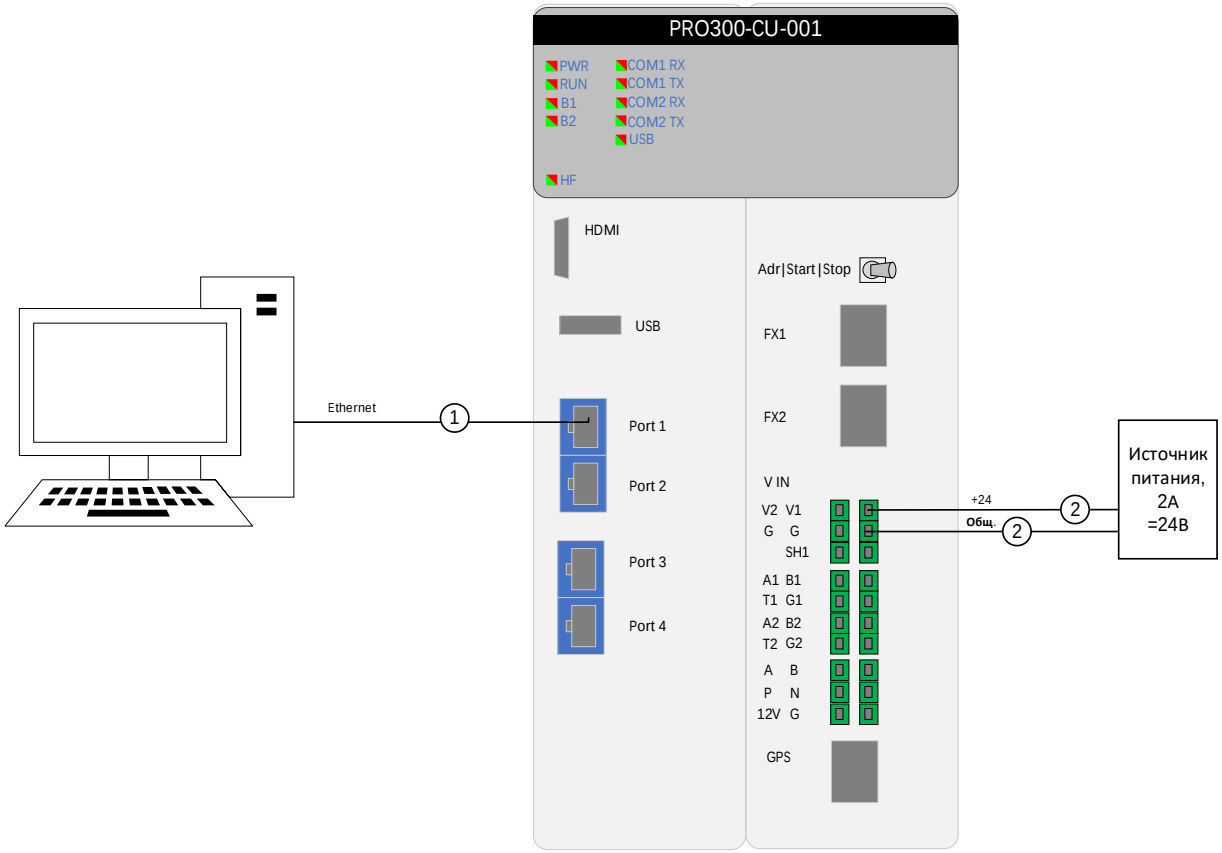


Рисунок 2 - Подача напряжения питания на общую электрическую шину через процессорный модуль UZOLA PRO300

1.4.3 Подключить МП к компьютеру по интерфейсу Ethernet (см. рисунки 1, 2).

1.4.4 Произвести сопряжение МВВ с МП.

1.5 Указания по включению и работе

1.5.1 Перед включением КТСИ следует убедиться в правильности и полноте выполнения подключений в соответствии с пунктом 1.4.

1.5.2 Включение КТСИ производить, руководствуясь положениями п.2.5 документа «Руководство по эксплуатации» на КТСИ.

1.5.3 Управление КТСИ осуществляется при помощи специализированного программного обеспечения CODESYS через компьютер, подключенный к процессорному модулю через канал связи Ethernet.

1.6 Порядок выключения и демонтажа после окончания работ

1.6.1 Выключение и демонтаж КТСИ производить, руководствуясь положениями п.2.6 документа «Руководство по эксплуатации» на КТСИ.

2 Установка и настройка CODESYS

2.1 Подключение процессорного модуля к ПО CODESYS

Подключение процессорного модуля к среде разработки и программирования КТСИ CODESYS (далее IDE) производить в соответствии с п.2.7 «Руководства по эксплуатации» на КТСИ.

2.2 Разработка проекта в IDE CODESYS

Для создания проектов и написания программы пользователя для КТСИ используется IDE CODESYS.

2.2.1 После выполнения всех пунктов 2.1 необходимо добавить необходимые в проекте модули ввода/вывода и используемые протоколы. Для их подключения необходимо выполнить последовательность действий:

а) открыть (если необходимо) дерево устройств: меню «Вид» -> вкладка «Устройства» или комбинация клавиш <Alt>+<0> (на рисунке 3 показан пример);

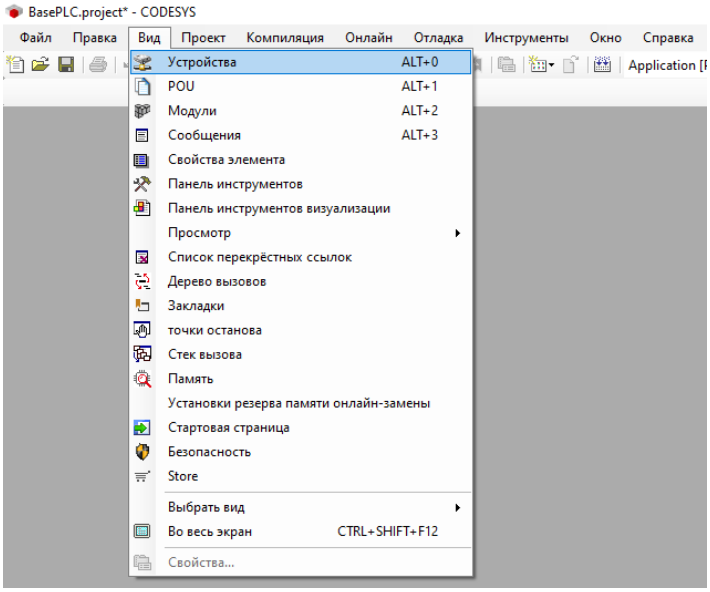


Рисунок 3

б) нажать ПКМ на добавленный ранее МП КТСИ и выпадающем списке выбрать «Добавить устройство…» (на рисунке 4 показан пример);

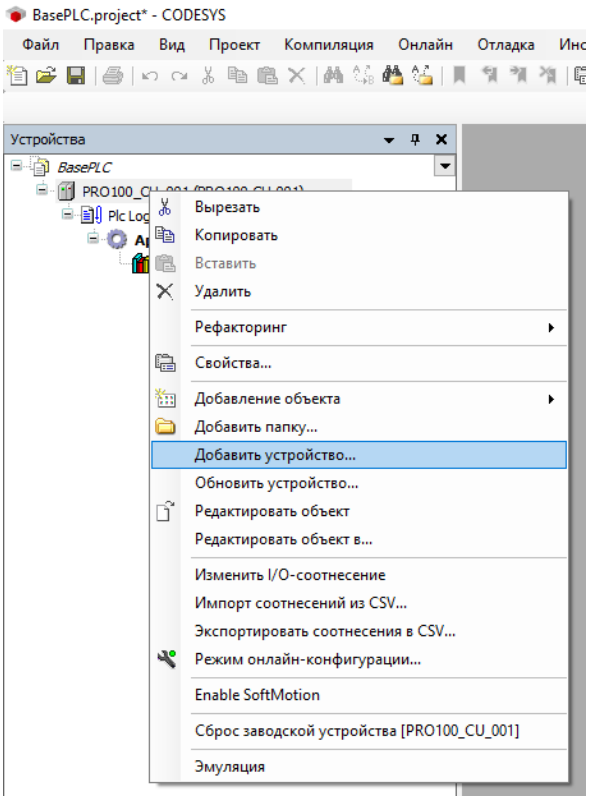


Рисунок 4

в) в появившемся окне выбрать из доступных нужный интерфейс или протокол для связи с периферией (на рисунке 5 показан пример);

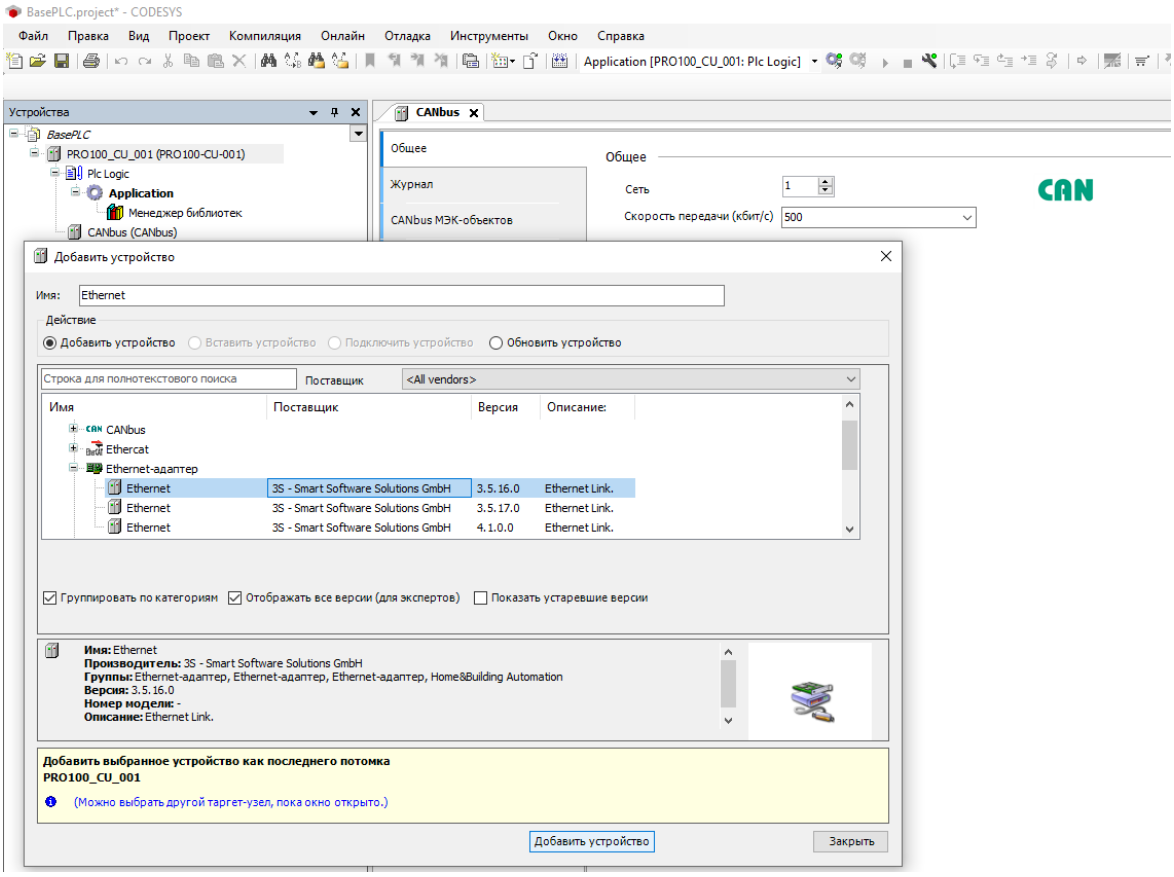


Рисунок 5

г) после перейти к настройке связи по интерфейсу/протоколу между МП и устройствами (на рисунке 6 показан пример).

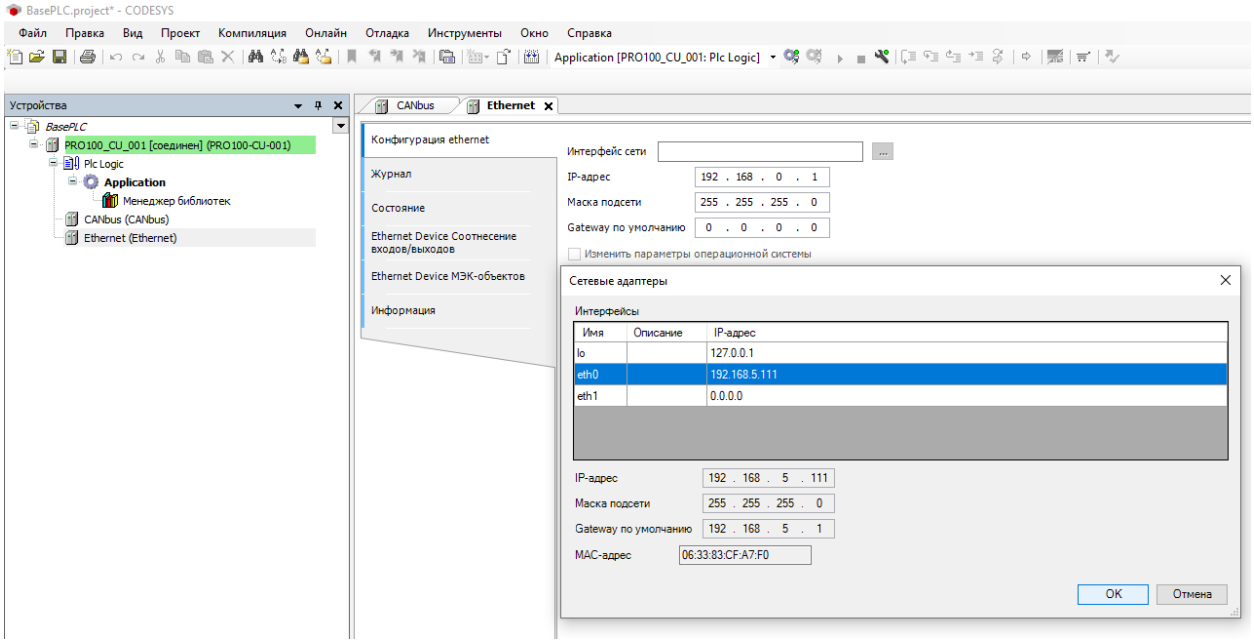


Рисунок 6

Можно воспользоваться сконфигурированным демонстрационным проектом

(BasePLC) из дистрибутива, поставляемого с КТСИ.

Внимание! При загрузке демонстрационного проекта без изменений настроек и состава модулей в КТСИ без подключенных модулей ввода/вывода, будут появляться сообщения об ошибках в дереве устройств и индивидуальных окнах устройств во вкладках «Журнал».

На рисунке 7 показан пример добавления интерфейса CAN. При этом, следует выполнить следующие действия:

– нажать ПКМ на Device (наш КТСИ);

– выбрать «Промышленные сети»;

– выбрать CANbus;

– нажав ПКМ, добавить CANopen Manager;

– далее в CANopen_Manager выбрать «ведомое устройство»;

– переход к настройке CAN интерфейса.

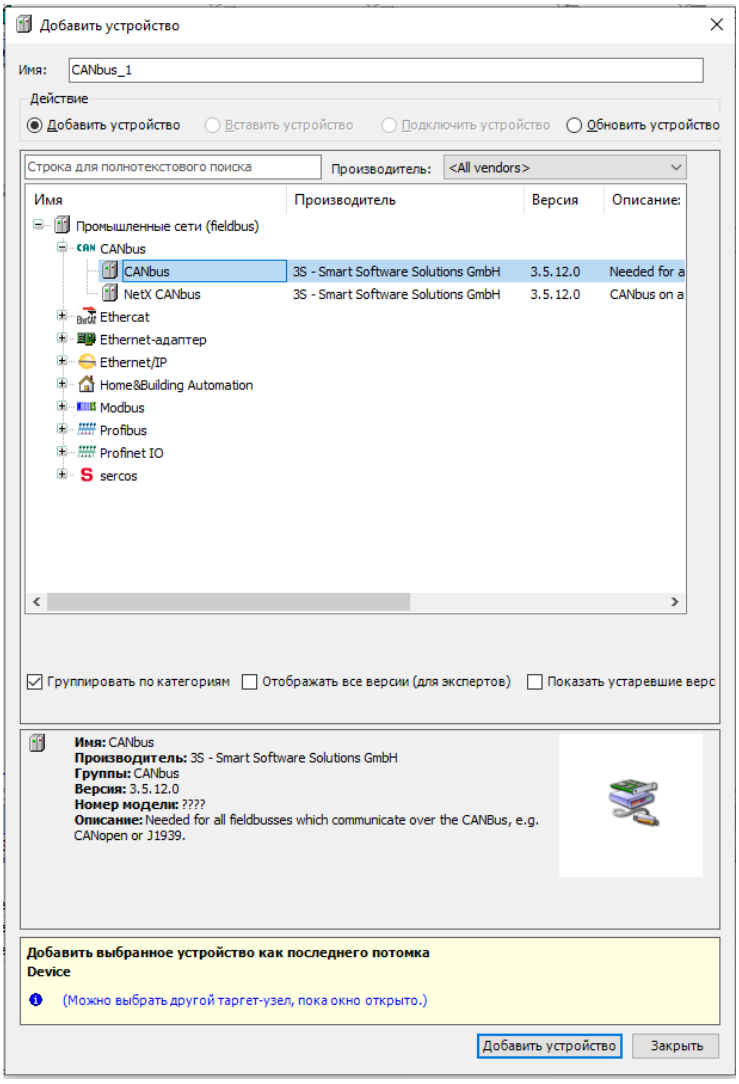


Рисунок 7 – Интерфейсы и протоколы Codesys

2.3 Связь с процессорным модулем

В КТСИ UZOLA PRO100 реализованы два независимых контроллера Ethernet (10/100 и 10/100/1000 base-TX). Коммуникационные порты Ethernet расположены снизу МП. Порт Ethernet1(в среде CODESYS – Eth0) имеет адрес по умолчанию – 192.168.1.110, Ethernet2 (в среде CODESYS – Eth1) имеет адрес по умолчанию – 192.168.5.150. Переопределение адресов портов осуществляется посредством ПО «Конфигуратор КТСИ Узола» (п.4.5 «Руководство конфигурирования КТСИ Узола»).

Примечание - Одновременная работа обоих портов Ethernet в одной сети некорректна.

В КТСИ UZOLA PRO300 реализованы шесть независимых контроллеров Ethernet 10/100/1000 base-TX, 4 из которых выведены на верхнюю часть МП. Ethernet может использоваться как независимо (для подключения к внешним устройствам), так и для подключения к одному устройству по двум линиям с резервированием канала связи. Порт Ethernet может использоваться как независимо (для подключения к внешним устройствам), так и для подключения к одному устройству по двум линиям с резервированием канала связи. Порт Port1 (в среде CODESYS – Eth4) имеет статичный адрес – 192.168.4.3/24, порт Port2 (в среде CODESYS – Eth5) имеет статичный адрес – 192.168.5.3/24, порт Port3 (в среде CODESYS – Eth2) имеет статичный адрес – 192.168.2.3/24, порт Port4 (в среде CODESYS – Eth3) имеет статичный адрес – 192.168.3.3/24.

2.4 Работа модулей в системе КТСИ

2.4.1 Работа контроллера в целом и программная модель

В КТСИ UZOLA PRO100 обмен между процессорными модулем и модулями ввода-вывода осуществляется по CAN – шине на скорости 500 Кбод. Используется протокол обмена CANopen с элементами профилей DS-301, DS-401.

Процессорный модуль является ведущим устройством, модули ввода-вывода являются ведомыми устройствами. Каждому ведомому устройству присваивается свой адрес (NodeID). Нумерация адресов начинается с 1 и увеличивается слева направо, т.е. ближайший к процессорному модулю модуль ввода-вывода должен иметь адрес 1, следующий справа модуль ввода-вывода должен иметь адрес 2 и т.д.

Параметры, которые может принимать или передавать модуль ввода-вывода (также называемые «словарь объектов»), содержатся в файле описания устройства (*.eds). Каждому типу модуля ввода-вывода соответствует свой словарь объектов; надо также

иметь ввиду что при смене версии программного обеспечения модуля ввода-вывода словарь объектов может измениться. Объекты словаря обычно делят на PDO и SDO. С точки зрения программной модели, объекты PDO – это параметры модуля ввода-вывода, которые передаются инициативно, а объекты SDO – это параметры модуля ввода-вывода, которые передаются по запросу.

При работе протокола CANopen на ведущем устройстве, каждому объявленному ведомому устройству присваивается СОСТОЯНИЕ в соответствии с машиной переходов протокола. Основные состояния: NOT_AVIAIL, UNKNOWN, RESET, INIT, PRE_OPERATIONAL, OPERATIONAL, STOPPED. Основные рабочие состояния: INIT, PRE_OPERATIONAL, OPERATIONAL; остальные состояния имеют служебный характер или возникают при обработке ошибок.

После установления первичной связи с ведомым устройством, этому устройству присваивается состояние INIT. В состоянии INIT осуществляется запрос идентификационных данных ведомого устройства. Если возвращаемые идентификационные данные совпадают с идентификационными данными словаря объектов, соответствующему данному устройству, осуществляется переход в состояние PRE_OPERATIONAL.

В состоянии PRE_OPERATIONAL запрещена передача PDO, разрешена передача SDO. В частности, в этом состоянии мастер передает ведомому устройству все SDO, которым при создании проекта был присвоен статус «передать при инициализации». Как правило, эти SDO содержат настроечные и конфигурационные параметры. После передачи таких SDO мастер дает ведомому устройству команду на переход в состояние OPERATIONAL.

В состоянии OPERATIONAL работа модуля ввода-вывода по назначению осуществляется в полном объеме.

В КТСИ UZOLA PRO300 информационный обмен осуществляется по сети Ethernet по протоколу Ethercat на скорости до 100 Мб/с. Процессорный модуль является ведущим устройством, модули ввода-вывода являются ведомыми устройствами. Направление передачи считается от процессорного модуля к модулю ввода-вывода: данные, которые передаются от процессорного модуля к модулю ввода-вывода называются выходными; а данные, которые передаются от модуля ввода-вывода, называется входными. Протокол Ethercat обеспечивает очень маленькую задержку между запросом данных и их получением.

В общем случае, используемая сеть имеет топологию «кольцо». Пакет Ethercat сначала отправляется от процессорного модуля к первому модулю ввода-вывода, потом от первого модуля ввода-вывода ко второму модулю ввода-вывода и т.д. От последнего модуля ввода-вывода пакет возвращается на процессорный модуль. Передаваемый пакет

содержит выходные и входные данные для всех модулей ввода-вывода, находящихся в кольце. Получив пакет, модуль ввода-вывода «на лету» считывает адресованные ему данные и записывает в пакет данные, которые он направляет процессорному модулю, и передает видоизмененный пакет дальше по кольцу. Таким образом, осуществляется минимальная задержка при передаче данных.

При обмене передаются как служебные данные, так и пользовательские данные. Для передачи пользовательских данных используется тип протокола CoE (CAN over Ethecat). Каждому типу модуля ввода-вывода соответствует файл описания устройства (*.xml), в котором определены все пользовательские переменные (также называемые «объекты»). Среди пользовательских переменных выделяются, так называемые, «данные процесса» - эти данные передаются в каждом пакете.

Индексация объектов в файле описания устройства осуществляется по определенным правилам.

Индекс вида 0x6nnx назначается для входных данные процесса, индекс вида 0x7nnx назначается для выходных данных процесса, индекс 0x8nnx назначается для конфигурационных данных модуля.

При работе протокола CoE на ведущем устройстве, каждому объявленному ведомому устройству присваивается СОСТОЯНИЕ в соответствии с машиной переходов протокола. Основные состояния: INIT, PRE_OPERATIONAL, SAFE_OPERATIONAL OPERATIONAL.

После установления первичной связи с ведомым устройством, этому устройству присваивается состояние INIT. В состоянии INIT осуществляется запрос идентификационных данных ведомого устройства, а обмен пользовательскими данными не производится. Если возвращаемые идентификационные данные совпадают с идентификационными данными словаря объектов, соответствующему данному устройству, осуществляется переход в состояние PRE_OPERATIONAL.

В состоянии PRE_OPERATIONAL данные процесса не передаются. Мастер передает ведомому устройству конфигурационные данные, объявленные пользователем при создании проекта. После передачи конфигурационных данных мастер дает ведомому устройству команду на переход в состояние SAFE_OPERATIONAL.

Для каждого типа модуля ввода-вывода в комплект поставки входит файл описания устройств соответствующей версии, а также тестовый проект, с помощью которого можно осуществлять настройку и техническое обслуживание модуля. Словарь объектов, описание PDO, SDO и список SDO, рекомендуемых для начальной инициализации, приведены в разделах настоящего руководства, посвященным описанию соответствующих типов

модулей ввода-вывода.

Последовательность подключения к проекту модулей ввода-вывода в общем случае следующая:

– провести нумерацию используемых модулей ввода-вывода в соответствии с порядком их расположения на шине контроллера;

– включить в проект CANopen менеджер, выбрать сеть «1», установить скорость обмена 500 кбит/с;

– в репозитарий устройств добавить файлы описания устройств используемых модулей ввода-вывода;

– добавить в CANopen менеджер устройства (модули ввода-вывода), при этом ID узла должен соответствовать порядковому номеру модуля ввода-вывода;

– для каждого модуля определить номенклатуру и значения SDO, передаваемые модулю в процессе инициализации;

– для использования в прикладной программе параметров модуля ввода-вывода, определенных как PDO, они должны быть присвоены переменным прикладной программы;

– для доступа из прикладной программы к параметрам модуля ввода-вывода, определенным как SDO, используются специальные функциональные блоки CODESYS: SDO_READ4; SDO_READ_DATA; SDO_WRITE4; SDO_WRITE_DATA.

Блок SDO_READ4 используется для чтения конкретного объекта из словаря объектов. Может быть прочитано до 4 байт. Данные записываются в файл. Если необходимо прочитать более 4 байт, используйте SDO_READ. Если объект может быть прочитан без ошибок, становится и содержит прочитанные данные в порядке байтов Little Endian. Если ошибка возникает, становится неравной 0. В случае прерывания SDO устанавливается значение CANOPEN_KERNEL_ERROR.
CANOPEN_KERNEL_OTHER_ERROR и содержит соответствующий код прерывания (в формате Little Endian), как определено в CiA 301.

Блок SDO_READ_DATA используется для чтения конкретного объекта из словаря объектов. Если объект может быть прочитан без ошибок, становится и содержит прочитанные данные в порядке байтов Little Endian. Если ошибка возникает, становится неравной 0. В случае прерывания SDO устанавливается значение CANOPEN_KERNEL_ERROR. CANOPEN_KERNEL_OTHER_ERROR и содержит соответствующий код прерывания (в формате Little Endian), как определено в CiA.

Блок SDO_WRITE4 используется для записи конкретного объекта в словарь объектов. Можно записать до 4 байт. Данные предоставляются в формате. Если необходимо

записать более 4 байт, используйте SDO_WRITE. Если было записано без ошибки, становится. должен быть в порядке байтов Little Endian. Если ошибка возникает, становится неравной 0. В случае прерывания SDO устанавливается значение CANOPEN_KERNEL_ERROR. CANOPEN_KERNEL_OTHER_ERROR и содержит соответствующий код прерывания (в формате Little Endian), как определено в CiA 301.

Блок SDO_WRITE_DATA используется для записи конкретного объекта в словарь объектов. Если было записано без ошибки, становится. должен быть в порядке байтов Little Endian. Если ошибка возникает, становится неравной 0. В случае прерывания SDO устанавливается значение CANOPEN_KERNEL_ERROR.
CANOPEN_KERNEL_OTHER_ERROR и содержит соответствующий код прерывания (в формате Little Endian), как определено в CiA 301.

2.4.2 Работа контроллера по протоколу Modbus TCP, Modbus RTU

2.4.2.1 Для работы КТСИ по протоколу Modbus TCP необходимо добавить в проект Ethernet-адаптер: нужно щёлкнуть ПКМ на название КТСИ в дереве устройств -> Добавить устройство -> Ethernet-адаптер, 3.5.16.0 (рисунок 8).

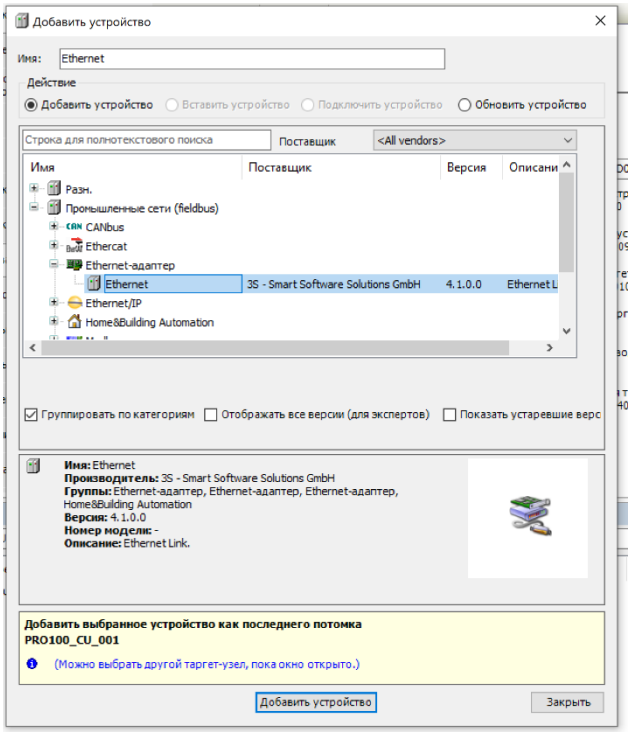


Рисунок 8

2.4.2.2 Добавить к адаптеру протокол Modbus (если Master - рисунок 9, если Slave - рисунок 10).

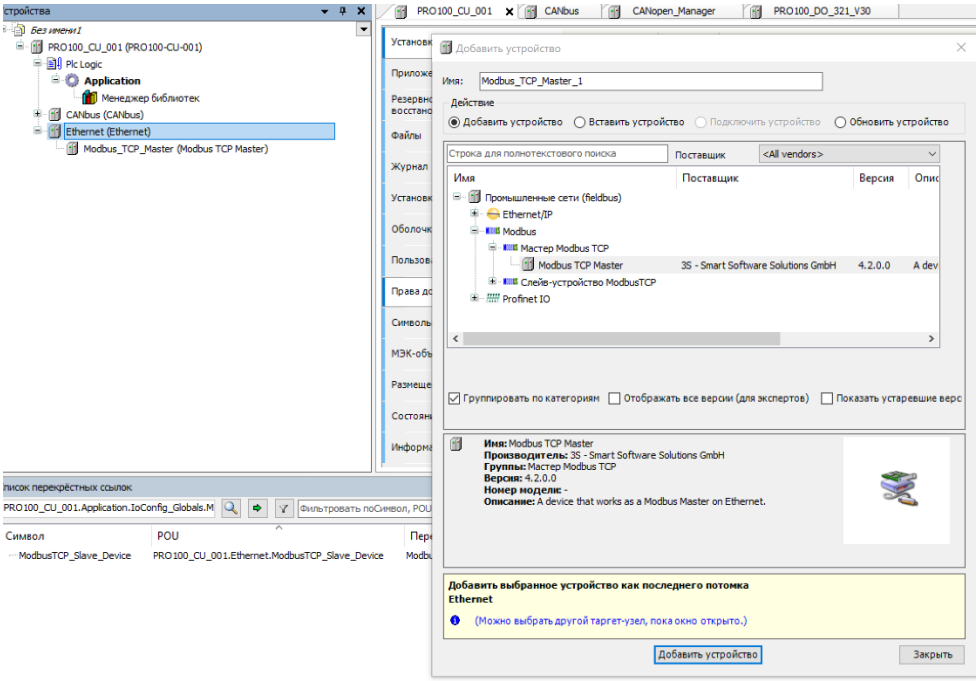


Рисунок 9

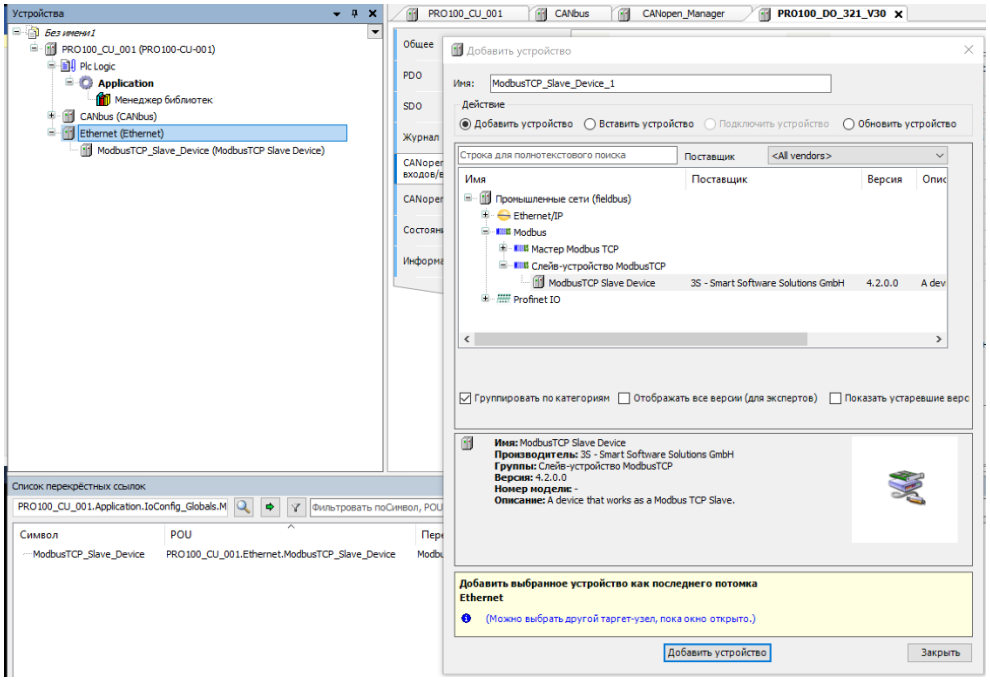


Рисунок 10

2.4.2.3 При использовании КТСИ в режиме Master:

а) приступить к настройке обмена со Slave'ом: двойной клик ЛКМ на модуле Ethernet, выбрать во вкладке конфигурации интерфейс eth0 или eth1 (eth1 и eth2 на корпусе CPU КТСИ соответственно, рисунок 11);

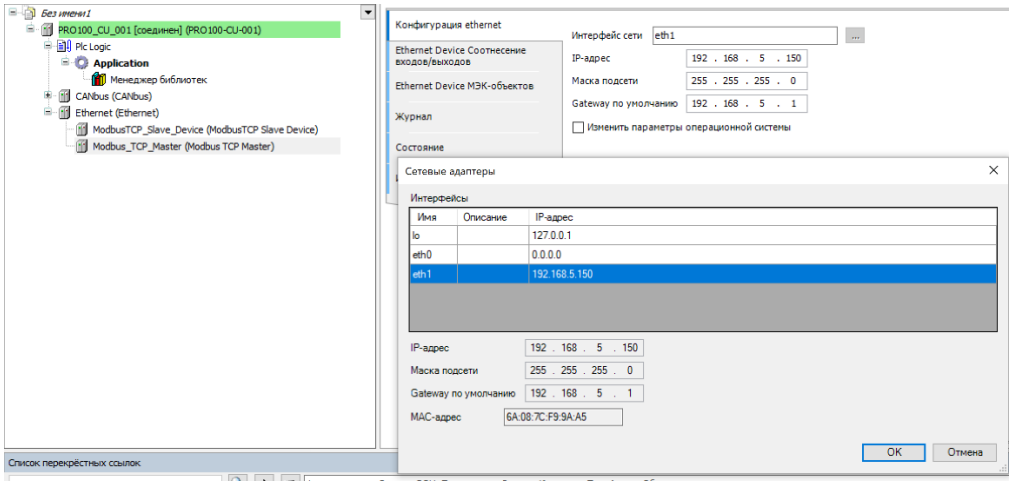


Рисунок 11

б) настроить поведение Modbus TCP Master: щёлкнуть дважды ЛКМ на модуле Modbus TCP Master в дереве устройств. Нужно установить «флажок-«галочка» «автоподключение», можно настроить тайминги сбоев связи. Также в этом окне можно отслеживать состояние подключение и передачи онлайн при выполнении программы пользователя (рисунок 12);

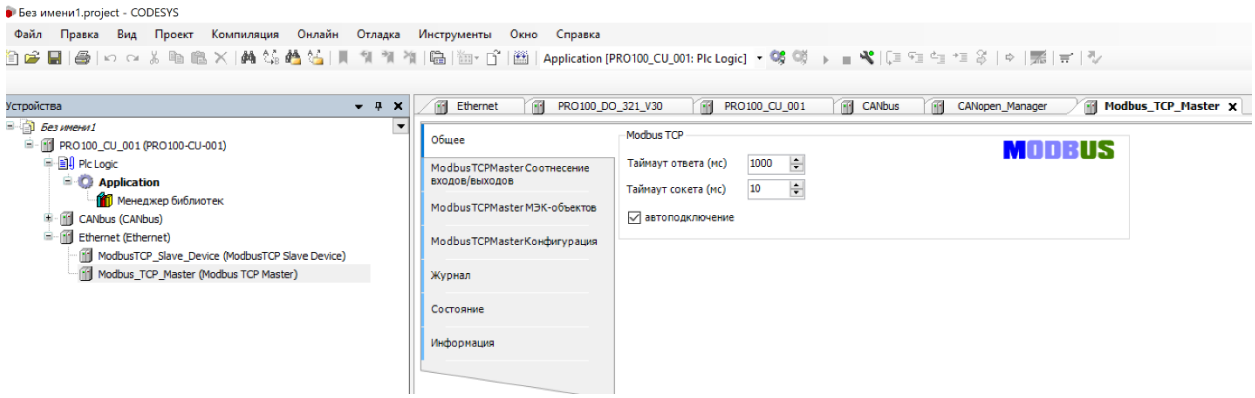


Рисунок 12

При использовании КТСИ в режиме Slave:

а) на вкладке «Страница конфигурации» указать Slave-порт (по умолчанию 502), указать величину массивов регистров временного хранения, входных регистров, где будут хранится опрашиваемые данные;

б) на вкладке «Modbus TCP Slave Соотнесение входов/выходов» отобразятся адреса переменных, соотнесенных с опрашиваемыми каналами в Slave устройстве. Их можно назначить переменным и использовать в программе пользователя без вызова каких-либо дополнительных ФБ.

По протоколу Modbus RTU необходимо добавить в устройство Modbus_COM (рисунок 13).

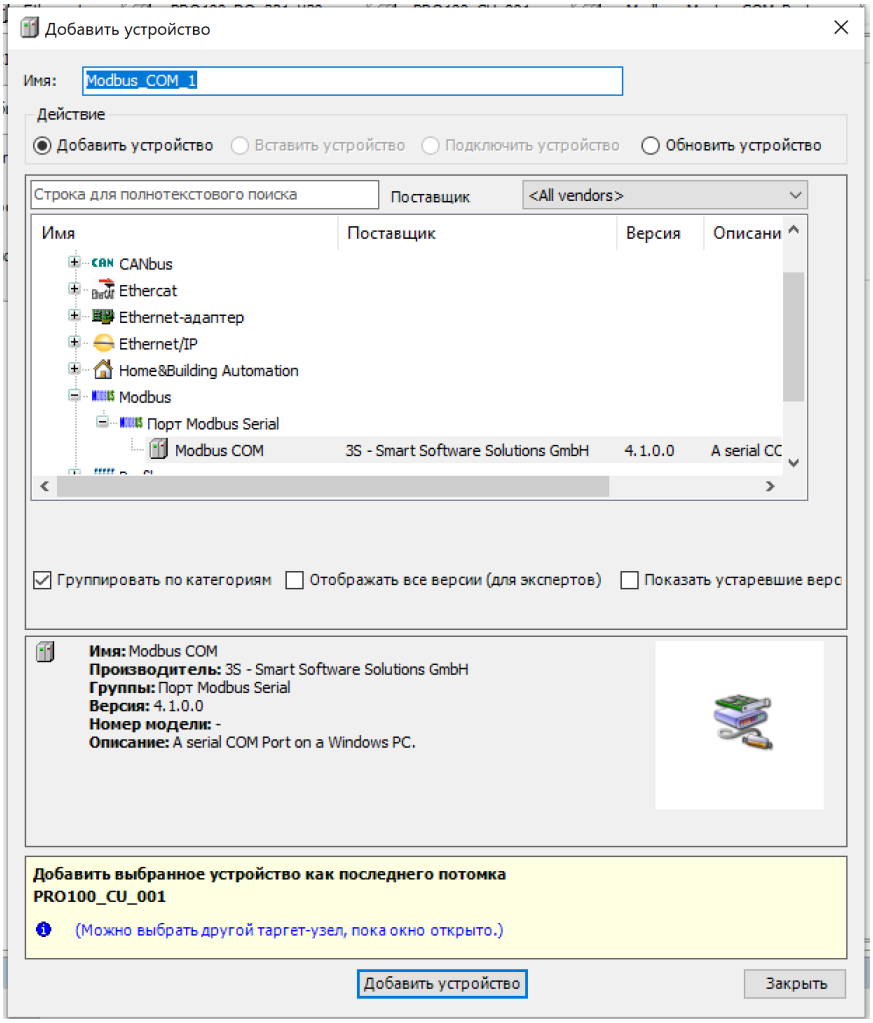


Рисунок 13

Во вкладке «Общие» этого устройства настроить номер COM-порта, скорость передачи, четность биты информации, стоповые биты. Далее нужно добавить, в зависимости от того как настраивается порт – мастером или слейвом, устройство Modbus_Master_Devce или Modbus_Serial_Device (рисунок 14).

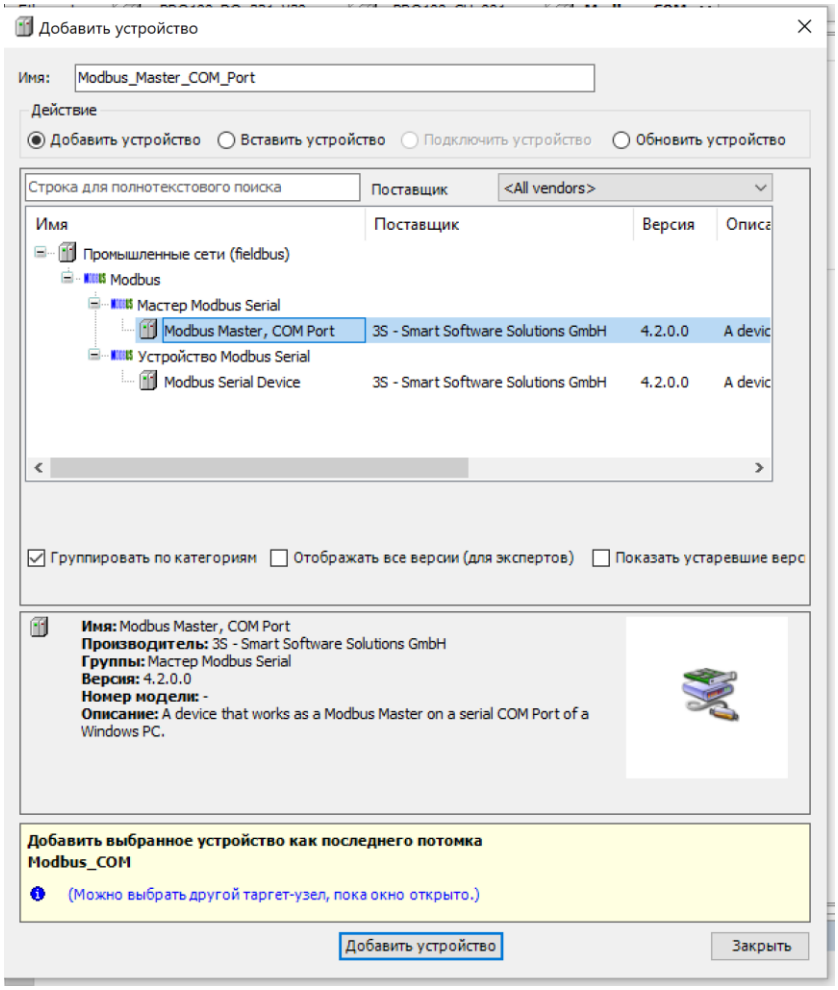


Рисунок 14

Настройка этих модулей практически не отличается от настройки по протоколу Modbus TCP (рисунок 15, рисунок 16).

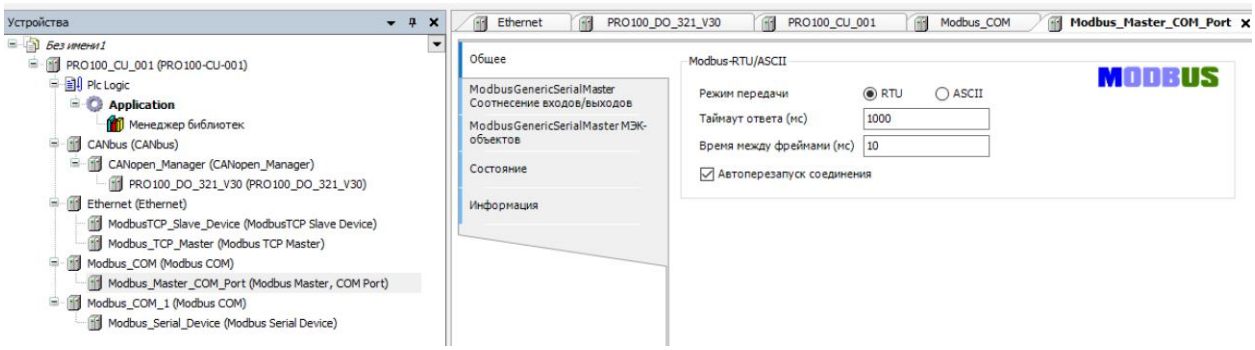


Рисунок 15

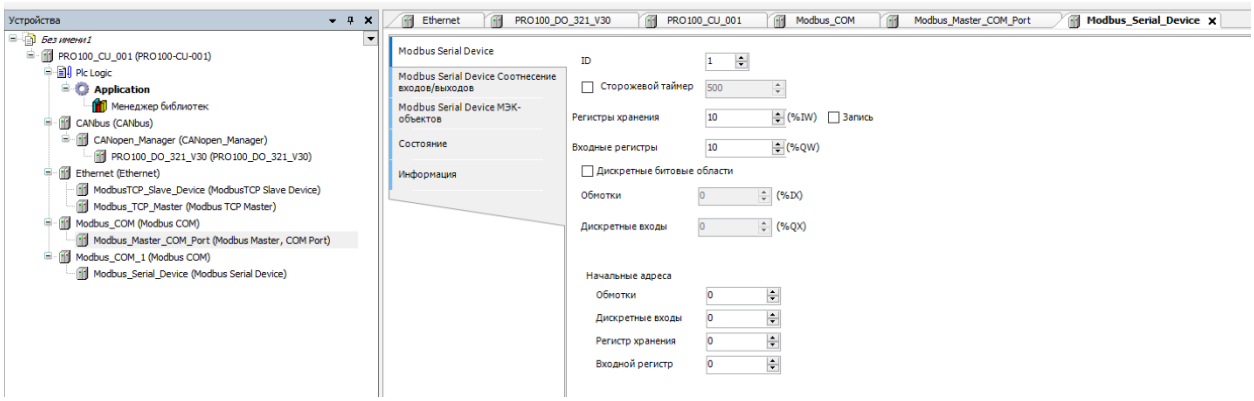


Рисунок 16

В CODESYS имеется набор библиотек для работы с Modbus (IoDrvModbus, IoDrvModbusSerialSlave, IoDrvModbusTCP), можно настроить работу через них. Добавление этих библиотек происходит автоматически при создании Modbus устройств (Master или Slave). Можно добавить необходимые вручную через Менеджер библиотек в дереве устройств. Там же есть описание библиотеки.

2.4.3 Настройка OPC UA

2.4.3.1 Настройка OPC UA происходит следующим образом (если в проекте уже есть POU с набором переменных): в дереве проектов ПКМ на узле Aplication -> выбрать «Добавление объекта» -> «Символьная конфигурация».

2.4.3.2 В появившемся окне выбрать «Поддержка функций OPC UA», можно дополнительно включить комментарии в XML.

2.4.3.3 В пункте «размещение данных на стороне клиента» выбрать «Совместимость» (рисунок 17).

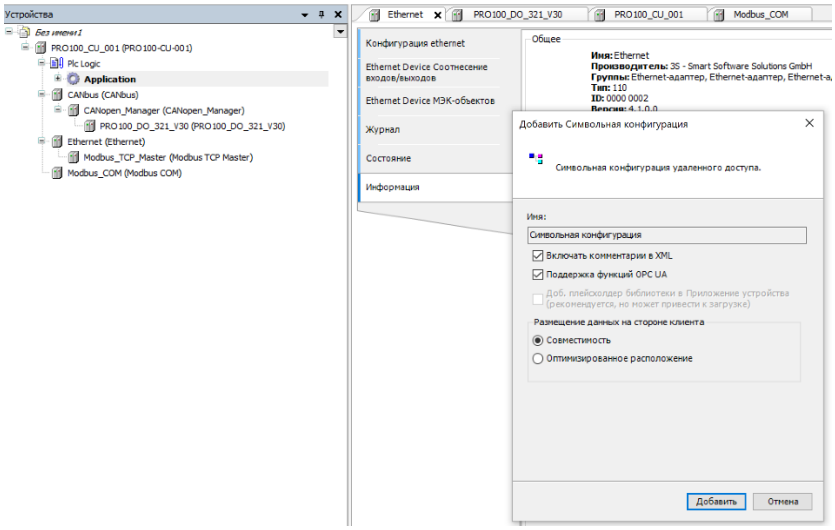


Рисунок 17

2.4.3.4 Щёлкнуть дважды ЛКМ на объекте «Символьная конфигурация» в дереве проектов -> в появившемся окне выполнить компиляцию после выбрать POU переменные, из которых вы планируете передавать на ВУ и повторно провести компиляцию уже всего проекта (рисунок 18).

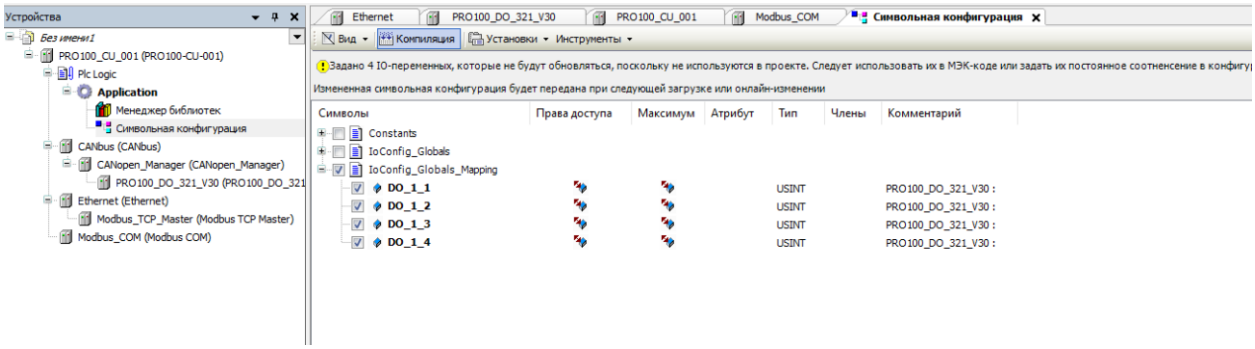


Рисунок 18

2.4.3.5 Проект можно заливать в КТСИ. Подключение к OPC UA серверу из вне происходит по адресу, выбранному для связи (eth0 или eth1) порт 4940.

2.4.3.6 Возможна настройка, путем добавления файлов описания устройств (*.eds) в IDE и потом подключения их в проект. По образу и подобию модулей ввода/вывода.

2.5 Сеть CANopen

2.5.1 Структура сообщений CANopen

CAN-сообщение содержит идентификатор (COB ID), код длины данных (DLC) и до 8 байтов данных:

COB ID	**Длина**	**Байт 1**	**Байт 2**	**Байт 3**	**Байт 4**	**Байт 5**	**Байт 6**	**Байт 7**	**Байт 8**
11 бит	X	x	x	x	x	x	x	x	X

11-битный идентификатор (COB-ID) состоит из 4-битного кода функции (FC) и 7-битного номера узла (Node ID)

Код функции (FC)				**Номер узла (Node ID)**					

Код функции определяет тип и приоритет сообщения. Меньший COB ID даёт более высокий приоритет.

Таблица типов сообщений и соответствующих COB ID:

Таблица 2

Тип сообщения	COB ID	Описание
Широковещательные сообщения		
NMT	0	Объект управления сетью для инициализации контроля сети
SYNC	80h	Объект для синхронизации устройств CAN-сети
Сообщения, адресуемые к узлам		
Emergency	80h + Node ID	Критические объекты для сообщений об ошибках
PDO1 (передача)	180h + Node ID	1й объект данных процесса (Process Data Object) для передачи данных в режиме реального времени
PDO1 (передача)	280h + Node ID	2й объект данных процесса
SDO (передача)	580h + Node ID	Служебный объект (Service Data Object) для чтения параметров через объектный словарь
SDO (приём)	600h + Node ID	Служебный объект (Service Data Object) для изменения параметров через объектный словарь
Heartbeat	700h + Node ID	Объект тактовых сообщений (Heartbeat message), периодически вызываемый для поддержания протокола проверки работоспособности устройств (Node Guarding Protocol)

2.5.2 Передача данных процесса

Объект данных процесса (PDO) используется для передачи данных в режиме реального времени одному или нескольким потребителям.

Структура PDO-сообщения:

	COB ID	Длина	Байт 1	Байт 2	Байт 3	Байт 4
PDO1	180h + node ID	4	LSB	x	x	MSB
PDO2	280h + node ID	4	LSB	x	x	MSB

Байты Байт1 – Байт4 содержат 32-битовое беззнаковое значение позиции с учётом смещения.

Объект PDO может передаваться в синхронном или асинхронном режиме. Для синхронизации устройств служит объект синхронизации SYNC, который периодически передаётся синхронизирующим приложением. Объект SYNC представлен как предопределённый и имеет двухбайтовую структуру:

Байт 1	Байт 2
COB-ID = 80h	0

Распределение синхронных (Synchronous PDO) и асинхронных (Asynchronous PDO) объектов передачи данных во времени показано на рисунке 19:

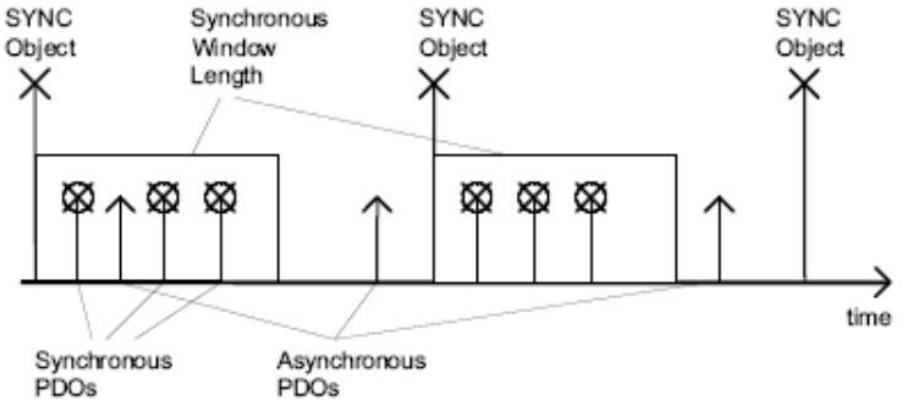


Рисунок 19

Синхронный режим задействован, когда объект 1800h / 1801h (для PDO1 / PDO2) субиндекс 2 имеет значение от 1 до 240 (F0h). При значении 1 данные передаются на каждое принятое SYNC сообщение; при другом значении n передача происходит на каждое n-е SYNC сообщение. Если значение 1800h/1801h-2 равно 0, то PDO1/PDO2 передаётся только один раз после каждого изменения кода синхронно с SYNC (независимо от значения объекта 2800h/2801h, см. далее).

Асинхронный режим задействован, когда объект 1800h / 1801h субиндекс 2 содержит величину FEh. В этом режиме PDO передаются в цикле. Время цикла в миллисекундах определяется объектом 1800h / 1801h субиндекс 5. Если содержимое объекта 0, то PDO не передаётся.

Примеры различных режимов передачи PDO1 / PDO2 показаны в таблице:

Таблица 3

1800h / 1801h		**2800h/ 2801h**	**Описание**
суб. 2	**суб. 5**		
Асинхронный режим			
FEh	0	x	Циклическая передача PDO выключена
FEh	4	0	Циклическая передача каждые 4мс
FEh	3	5	Циклическая передача каждые 3мс, но только 5 раз после изменения позиции
FEh	7	1	Циклическая передача каждые 7мс один раз при изменении позиции

1800h / 1801h		2800h/ 2801h	Описание
суб. 2	суб. 5		
Синхронный режим			
0	x	x	Передача PDO происходит один раз при каждом изменении значения угла
2	x	0	Передача PDO на каждое 2е SYNC сообщение
3	x	12h	Передача 18 раз (=12h) по каждому 3 SYNC сообщению
Установки PDO1 по умолчанию			
FEh	203h	0	Циклическая передача каждые 515мс (203h)
Установки PDO2 по умолчанию			
2h	100h	0	Передача по каждому 2 SYNC сообщению

Передача PDO также выключена, когда 31й бит объекта 1800h/1801h субиндекс 1 равен 1. Запись в 1800/1801-1 влияет только на этот бит.

Для ограничения количества передач PDO используются объекты 2800h / 2801h. Если значение равно 0, то передача PDO происходит как описано выше. Если объект 2800h / 2801h имеет ненулевое значение, то передача данных происходит соответствующее количество раз после каждого изменения позиции или после команды NMT «сброс». Если значение равно 1, то передача происходит однократно при каждом изменении позиции.

2.5.3 Обмен сервисными данными (работа с объектным словарём)

Все параметры устройства хранятся в объектном словаре.

Адреса параметров (индексы) стандартизированы, они могут быть прочитаны или изменены путём чтения/записи сервисных объектов (Service Data Object – SDO).

SDO сообщение имеет структуру:

COB-ID	Длина	Байт 1	Байт 2	Байт 3	Байт 4	Байт 5	Байт 6	Байт 7	Байт 8
11-бит	0..4	Команда	Индекс		Субиндекс	Параметр			
			LSB	MSB		LSB	…	…	MSB

COB-ID равен 580h+**nodeID** для передачи Устройство → Мастер или **600h+nodeID** для передачи Мастер → Устройство.

DLC – длина кода параметра в байтахПоиск и устранение отказов.

Байт 1 – Код команды, определяет действие с параметром:

Таблица 4

Команда	Описание
22h	Установка параметра устройства
42h	Запрос величины параметра
43h, 4Bh, 4Fh	Запрос величины параметра
60h	Подтверждение устройством изменения параметра
80h	Сообщение аварийного завершения, также «мастеру» передаётся сообщение об ошибке

Индекс и **Субиндекс** соответствуют значениям в объектном словаре.

Величина Параметр состоит из 0…4 байтов В случае ошибки передаётся сообщение об аварийном завершении (**SDO abort message**), которое имеет структуру:

COB-ID	Длина	Байт 1	Байт 2	Байт 3	Байт 4	Байт 5	Байт 6	Байт 7	Байт 8
580h + node ID	8	80h	Индекс		Суб-индекс	Байт ошибки 0	Байт ошибки 1	Байт ошибки 2	Байт ошибки 3

Индекс и субиндекс соответствуют запрашиваемому объекту.

Поддерживаются следующие сообщения:

0504 0001 Спецификатор команды неизвестен или не работает

0601 0001 Попытка чтения только записываемого объекта

0601 0002 Попытка записи только читаемого объекта

0602 0000 Объект не существует в Словаре объектов

0606 0000 Неудачный доступ к объекту из-за аппаратной ошибки

0609 0011 Субиндекс не существует

0609 0030 Величина вне допустимого диапазона (для записи)

0800 0000 Общая ошибка

0800 0020 Неверная подпись “load” или “save”

2.5.4 Служба аварийных сообщений

Внутренние ошибки устройства или ошибки связи вызывают аварийное сообщение, которое имеет структуру:

COB-ID	Длина	Байт 1	Байт 2	Байт 3	Байт 4	Байт 5	Байт 6	Байт 7	Байт 8
80h+node ID	**8**	Код ошибки		Регистр ошибок 1001h	Аварии 6503h		Предупреждения 6505h		-

Байты 2..1: Код ошибки:

0000 Ошибка сброшена или ошибки отсутствуют

1000 Общая ошибка

5530 Ошибка энергонезависимой памяти (EEPROM)

6010 Сброс по сторожевому таймеру

7320 Ошибка позиции (напр., выходного кода датчика угла)

7510 Ошибка связи по CAN-линии (Bus off)

8130 Ошибка от охранного протокола NodeGuard

Байт 3: Регистр ошибок

Бит 0 Общая ошибка

Бит 4 Ошибка связи

Бит 5 Ошибка охранного протокола NodeGuard

Бит 7 Ошибка памяти EEPROM

Байты 5..4: Аварии

Бит 0 Недействительный код положения

Бит 1 Аппаратная ошибка

Байты 7..6: Предупреждения

Бит 2 Сброс по сторожевому таймеру

С позиции рассмотрения ошибок устройство может находиться в двух состояниях: A (ошибок нет) или B (ошибки появлялись и не устранены):

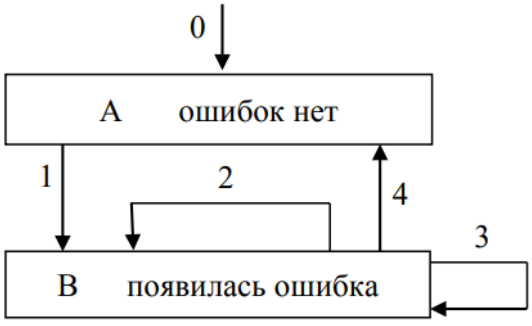


Рисунок 20

В зависимости от переходов между этими состояниями будут передаваться следующие *аварийные сообщения*:

0. После инициализации устройства, в случае отсутствия ошибок, устройство переходит в состояние A (нет ошибок), и сообщения об ошибке не посылаются.

1. Обнаружена ошибка, указанная в первых трёх байтах *аварийного сообщения*. Устройство переходит в состояние B. Передаётся *аварийное сообщение* с соответствующим кодом ошибки (см. выше), содержащее регистр ошибок (объект 1001h). Код ошибки записывается в объект 1003h – массив ошибок.

2. Одна, но не последняя, ошибка устранена. Передаётся *аварийное сообщение*, содержащее код 0000 (ошибка сброшена) в поле кода ошибок, а также оставшиеся ошибки и предупреждения в соответствующих полях.

3. Обнаружена новая (не первая ошибка). Устройство остаётся в состоянии B и передаёт *аварийное сообщение* с соответствующим кодом ошибки. Новый код ошибки заносится в верхний элемент массива ошибок (объект 1003h).

4. Все ошибки устранены. Устройство переходит в состояние отсутствия ошибок A и передаёт *аварийное сообщение* с кодом ошибки 0000.

2.5.5 Протокол проверки узла

Этот протокол (Node guarding protocol) используется для определения ошибок в сети при помощи удалённого запроса. NMT мастер делает удалённый запрос устройства через регулярные отрезки времени – период опроса (Guard time). Отклик устройства содержит код состояния узла и переключаемый бит и имеет следующую структуру:

COB-ID	**Длина**	**Байт 1**
700+node ID	1	Переключаемый бит (7) + Состояние узла (6..0)

Состояние узла может принимать значения:

4 останов (Stopped)

5 Работа (Operational)

127 Подготовительный режим (Pre-operational)

Переключаемый бит инициируется значением 0 при старте устройства или переходе в режим «Node Guarding» и меняет значение при каждой посылке.

Контрольное время (Node Life Time) определяется как произведение интервала Guard Time и множителя Lifetime Factor. Если устройство не принимает сообщение от NMT-мастера в течение контрольного времени, то посредством службы Emergency Service генерируется информация об ошибке, которая хранится в объекте 1003h. При приёме NMT-сообщения ошибка сбрасывается.

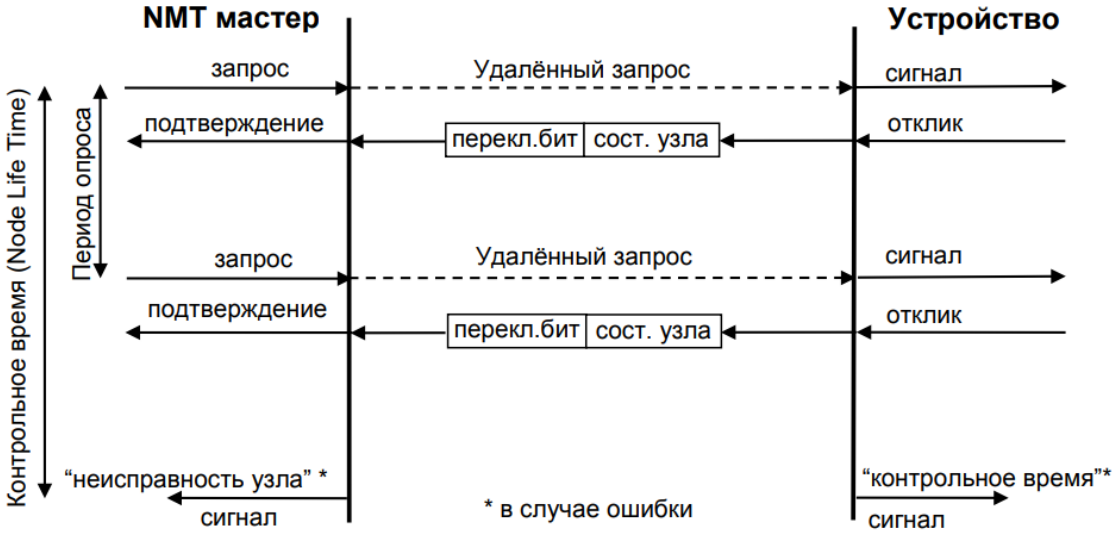


Рисунок 21

Протокол проверки узла использует следующие объекты:

Таблица 5

Объект	Параметр	Тип	Описание
100Ch	Период опроса (Guard time)	U16	0 – Протокол проверки узла не используется 1..FFFFh – период опроса, мсек
100Dh	Множитель (Lifetime factor)	U32	0 – Протокол проверки узла не используется 1..FFh – множитель

2.5.6 Протокол проверки связи

Протокол определяет службу контроля над ошибками (Error Control Service) без необходимости использования удалённых запросов. Устройство периодически посылает сообщения (Heartbeat messages) следующей структуры:

COB-ID	Длина	Байт 1
700+node ID	1	Состояние узла

Состояние узла может принимать значения:

4 останов (Stopped)

5 Работа (Operational)

127 Подготовительный режим (Pre-operational)

Замечание. Если на шине появляется удалённое сообщение (RTR) с совпадающим COB-ID, то устройство начинает посылать сообщения с пустым байтом состояния узла. Один или более потребителей принимают сообщения. Отношения между передатчиком и

потребителем конфигурируются посредством двух объектов:

Таблица 6

Объект	Параметр	Тип	Описание
1016h суб1	Контрольное время потребителя (Consumer heartbeat time)	U32	Определяет допустимый интервал ожидания посылок в мсек. Должен быть больше периода посылок передатчика
1017h	Период посылок передатчика (Producer heartbeat time)	U16	Определяет время в мсек циклических посылок передатчика. Если равно 0, то протокол не используется

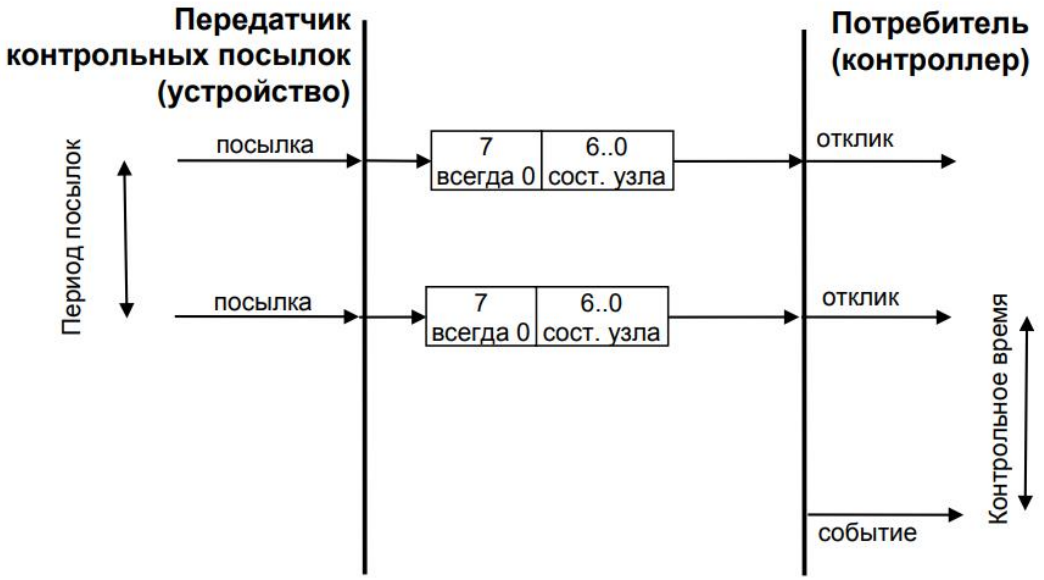


Рисунок 22

Невозможно использовать одновременно оба протокола контроля работоспособности в одном устройстве.

Если объекты 100Ch, 100Dh, а также 1017h не равны 0, то приоритетным является протокол Heartbeat. Т.е. при задании ненулевого периода (1017h) протокол NodeGuard отключается. Для его включения нужно обнулить 1017h, а затем задать ненулевые значения 100Ch и 100Dh.

Приоритет Heartbeat соблюдается также при инициализации устройства после перезапуска или включения. В этом случае включение протокола определяется по наличию ненулевых значений в указанных объектах, сохранённых в EEPROM.

2.6 Реализация обмена через сокеты в среде CODESYS

2.6.1 Основные сведения о работе с сокетами

2.6.1.1 Общая информация о сокетах

Сокет – это программный интерфейс, который обеспечивает обмен данными между процессами. Сетевые сокеты позволяют организовать обмен данными между процессами, которые выполняются на разных устройствах. Примером такого процесса может являться пользовательская программа, выполняемая КТСИ. 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 этом разделе 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.

Список портов, используемых средой CODESYS и сервисами контроллера, приведен в руководстве CODESYS V3.5. FAQ

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 КТСИ.

Не следует пытаться занимать их другими процессами.

2.6.1.2 Общая информация о сокетах

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 в случае их получения выполняет заданные операции – после чего, в случае 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 (в случае использования последовательных интерфейсов в каждый момент времени активным является только одно устройство, управляющее уровнем напряжения на линии связи). В настоящем руководстве рассматривается реализация сервера и клиента для протоколов UDP и TCP.

2.6.1.3 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 базе протокола 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.

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 передачи должен использоваться последний IP-адрес сегмента. Например, в случае отправки UDP-пакета на адрес 10.2.11.255, он будет доставлен устройствам с адресами 10.2.11.1 – 10.2.11.254.

Рисунок 23 – Схемы маршрутизации UDP

2.6.1.4 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:

• 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;

• 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

2.6.1.5 Вопросы информационной безопасности

В рамках данного документа не рассматриваются вопросы информационной безопасности и защищенной передачи данных. В качества источника информации по этому вопросу можно использовать документ CODESYS Security Whitepaper и раздел Security сайта CODESYS.

2.6.1.6 Средства отладки

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

2.6.1.7 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.

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 настоящем руководстве.

2.6.2 Библиотека CAA Net Base Services

2.6.2.1 Добавление библиотека в проект CODESYS

Библиотека CAA Net Base Services используется для обмена данными по протоколам UDP и TCP. Для добавления библиотеки в проект CODESYS в Менеджере библиотек следует нажать кнопку Добавить и выбрать библиотеку CAA Net Base Services.

Примечание - Версия библиотеки не должна превышать версию таргет-файла контроллера. В противном случае корректная работа контроллера не гарантируется.

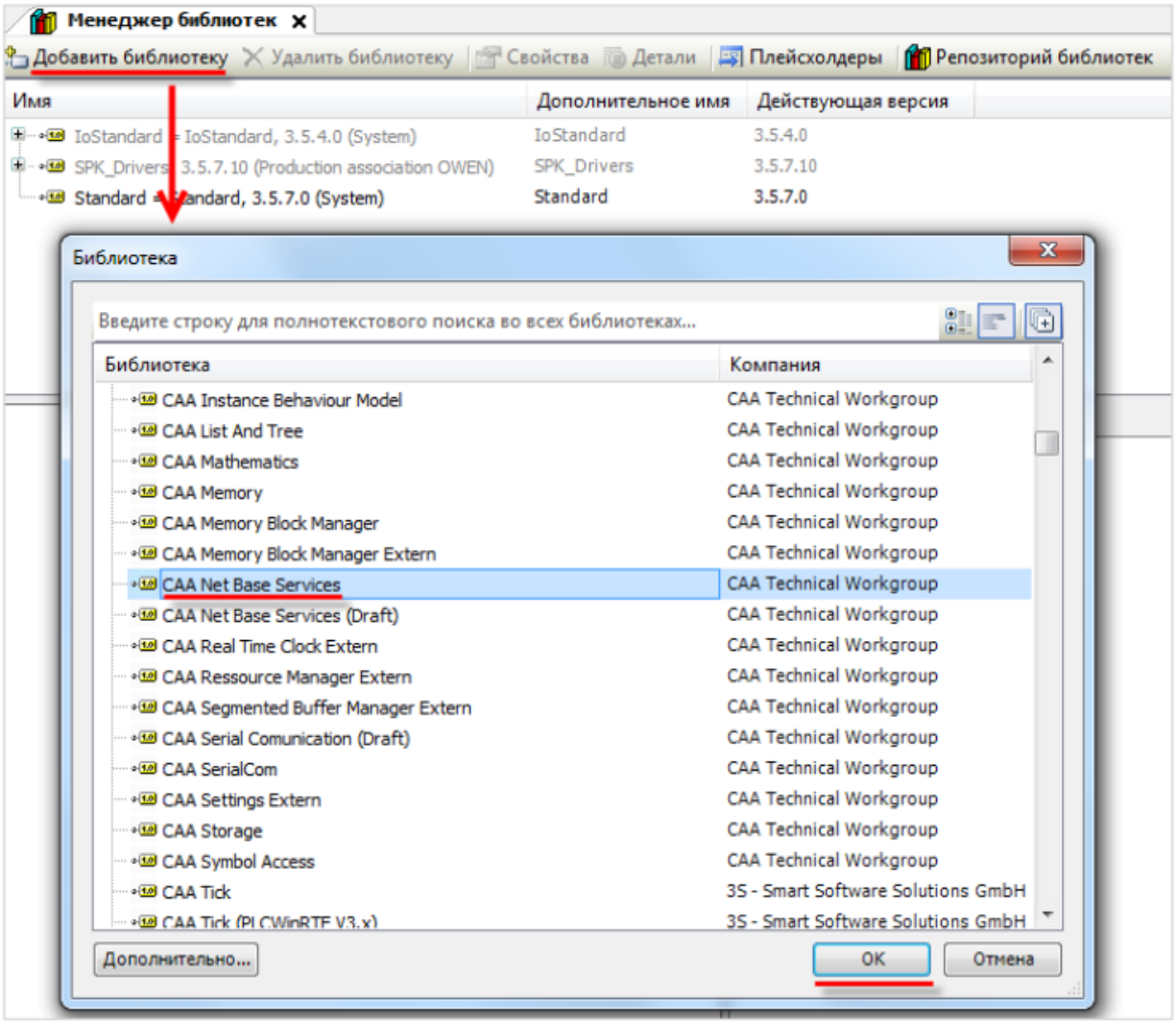


Рисунок 24 - Добавление библиотеки CAA Net Services в проект CODESYS

Примечания:

1 При объявлении экземпляров ФБ библиотеки следует перед их названием указывать префикс NBS (пример: NBS.TCP_Server).

2 Следует различать библиотеки CAA Net Base Services (рассматриваемую в данном документе) и NetBaseServices (без префикса CAA). Библиотека NetBaseServices появилась в более поздних версиях CODESYS. Она является более функциональной (в частности, поддерживается защищенная передача данных с использованием протокола

TLS) и более сложной в использовании.

2.6.2.2 Структуры и перечисления

2.6.2.2.1 Структура NBS.IP_ADDR

Структура NBS.IP_ADDR описывает IP-адрес.

Таблица 7 - Переменные структуры NBS.IP_ADDR

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

2.6.2.2.2 Перечисления NBS.ERROR

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

Таблица 8

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

2.6.2.3 ФБ работы с протоколом UDP

2.6.2.3.1 ФБ NBS.UDP_Peer

Функциональный блок NBS.UDP_Peer создает UDP-сокет 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). Если вход ipAddr оставлен пустым, то сокет связывается со всеми интерфейсами контроллера (используется специальное значение ‘0.0.0.0’).

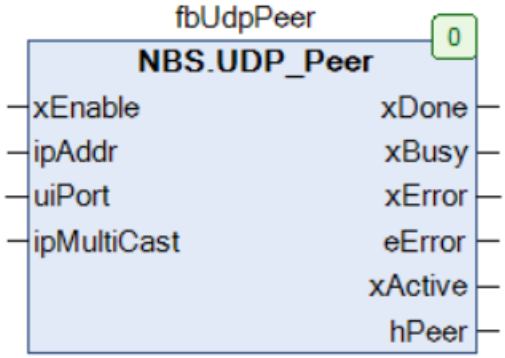


Рисунок 25 – Внешний вид ФБ NBS.UDP_Peer на языке CFC

Таблица 9

Название	Тип данных	Описание
Входные переменные		
xEnable	BOOL	Вход управления блоком. Пока он имеет значение TRUE – блок находится в работе
ipAddr	NBS.IP_ADDR	IP-адрес интерфейса, на котором создается сокет
uiPort	UINT	Номер порта
uiMultiCast	NBS.IP_ADDR	IP-адрес мультикаст-группы
Выходные переменные		
xDone	BOOL	Флаг завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение TRUE в случае возникновения ошибки
eError	NBS.ERROR	Статус работы ФБ (или имя ошибки)
xActive	BOOL	Флаг «сокет успешно открыт». Пока он имеет значение TRUE – сокет открыт, и его можно

		использовать
hPeer	NBS.CAA.HANDLE	Дескриптор открытого сокета устройства

2.6.2.3.2 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.

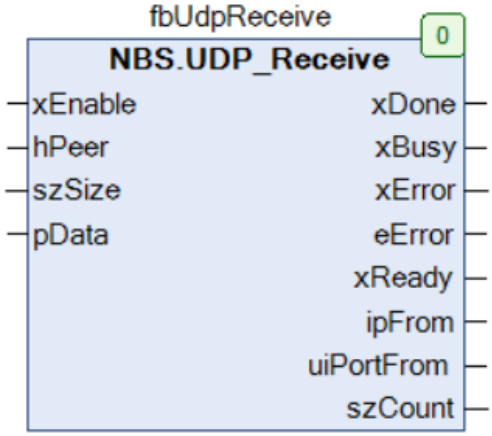


Рисунок 26 - Внешний вид ФБ NBS.UDP_Receive на языке CFC

Таблица 10

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

2.6.2.3.3 ФБ NBS.UDP_Send

Функциональный блок NBS.UDP_Send используется для отправки данных на заданный IP-адрес/порт.

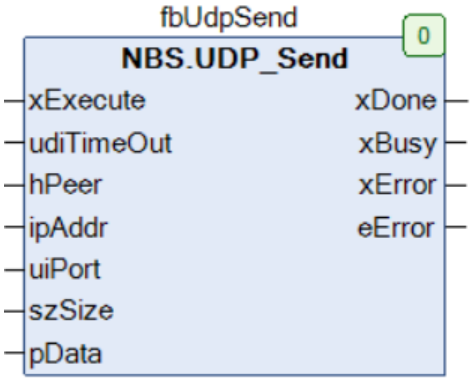


Рисунок 27 – Внешний вид ФБ NBS.UDP_Send на языке CFC

Таблица 11 - Описание входов и выходов ФБ NBS. UDP_Send

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

2.6.2.3.4 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.

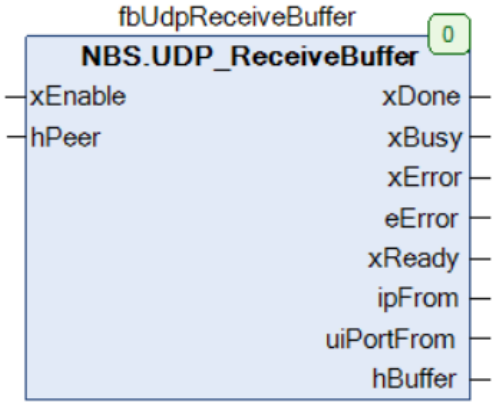


Рисунок 28 – Внешний вид ФБ NBS.UDP_ReceiveBuffer на языке CFC

Таблица 12 - Описание входов и выходов ФБ NBS. UDP_ReceiveBuffer

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

2.6.2.3.5 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.

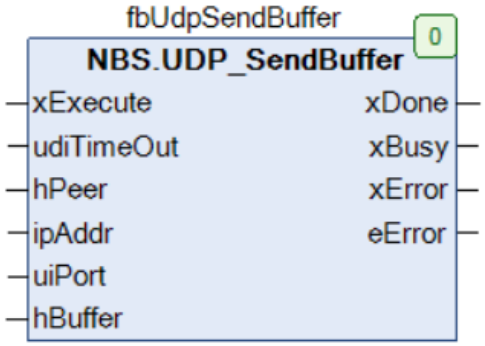


Рисунок 29 – Внешний вид ФБ NBS.UDP_SendBuffer на языке CFC

Таблица 13 - Описание входов и выходов ФБ NBS. UDP_SendBuffer

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

2.6.2.4 ФБ работы с протоколом TCP

2.6.2.4.1 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. Если вход ipAddr оставлен пустым, то сокет связывается со всеми интерфейсами контроллера (используется специальное значение ‘0.0.0.0’).

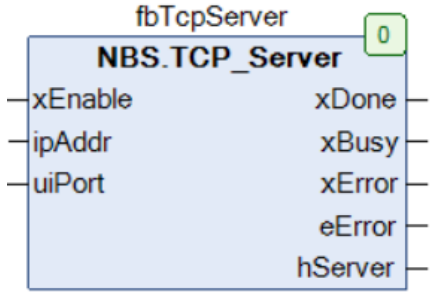


Рисунок 30 – Внешний вид ФБ NBS.TCP_Server на языке CFC

Таблица 14

Название	Тип данных	Описание
Входные переменные		
xEnable	BOOL	Вход управления блоком. Пока он имеет значение TRUE – блок находится в работ
ipAddr	NBS.IP_ADDR	IP-адрес интерфейса, на котором создается сокет
uiPort	UINT	Номер порта сервера
Выходные переменные		
xDone	BOOL	Флаг завершения работы блока. Принимает значение TRUE на один цикл
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение TRUE в случае возникновения ошибки
eError	NBS.ERROR	Статус работы ФБ (или имя ошибки)
hServer	NBS.CAA.HANDLE	Дескриптор сервера

2.6.2.4.2 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).

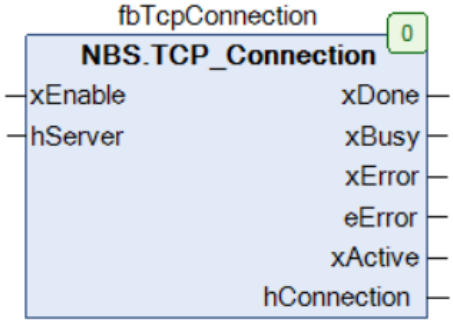


Рисунок 31 – Внешний вид ФБ NBS.TCP_Connection на языке CFC

Таблица 15 – Описание входов и выходов ФБ NBS. TCP_Connection

Название	Тип данных	Описание
Входные переменные		
xEnable	BOOL	Вход управления блоком. Пока он имеет значение TRUE – блок находится в работе
hServer	NBS.CAA.HANDLE	Дескриптор сервера
Выходные переменные		
xDone	BOOL	Флаг завершения работы блока. Принимает значение TRUE в случае разрыва соединения со стороны клиента. Повторное соединение будет невозможно до перезапуска блока через вход xEnable
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение TRUE в случае возникновения ошибки
eError	NBS.ERROR	Статус работы ФБ (или имя ошибки)
xActive	BOOL	Флаг активности соединения. Если он имеет значение TRUE – то к серверу подключен клиент. Принимает значение FALSE в случае разрыва соединения
hConnection	NBS.CAA.HANDLE	Дескриптор подключения (0 – соединение не установлено или разорвано)

2.6.2.4.3 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).

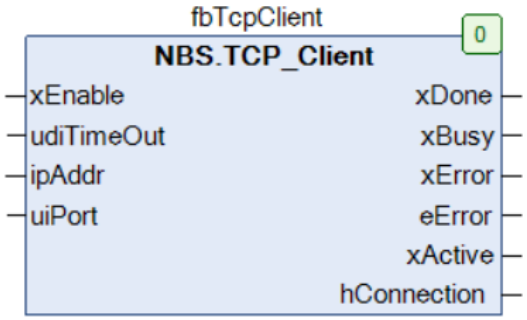


Рисунок 32 – Внешний вид ФБ NBS.TCP_Client на языке CFC

Таблица 16 - Описание входов и выходов ФБ NBS. TCP_Client

Название	Тип данных	Описание
Входные переменные		
xEnable	BOOL	Вход управления блоком. Пока он имеет значение TRUE – блок находится в работе
udiTimeOut	UDINT	Допустимое время операции (в мкс). Значение 0 означает, что время выполнения ФБ не ограничивается
ipAddr	NBS.IP_ADDR	IP-адрес сервера
uiPort	UINT	Номер порта сервера
Выходные переменные		
xDone	BOOL	TRUE – сервер закрыл соединение. Для повторного соединения требуется создать передний фронт на входе xEnable. Значение выхода обновляется только после запроса в рамках данного соединения
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение TRUE в случае возникновения ошибки
eError	NBS.ERROR	Статус работы ФБ (или имя ошибки)
xActive	BOOL	Флаг «сокет успешно открыт». Пока он имеет значение TRUE – сокет открыт, и его можно использовать
hConnection	NBS.CAA.HANDLE	Дескриптор подключения

2.6.2.4.4 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).

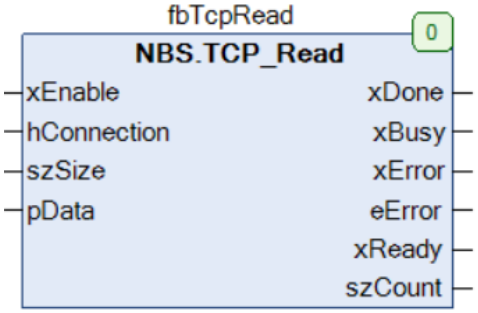


Рисунок 33 – Внешний вид ФБ NBS.TCP_Read на языке CFC

Таблица 17 - Описание входов и выходов ФБ NBS. TCP_Read

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

2.6.2.4.5 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).

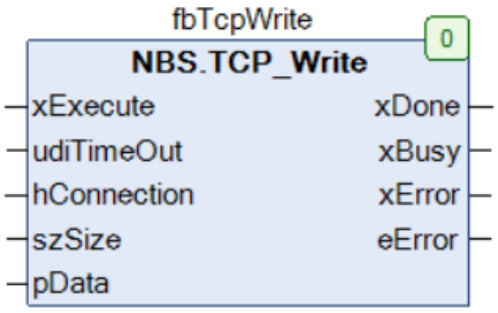


Рисунок 34 – Внешний вид ФБ NBS.TCP_Write на языке CFC

Таблица 18 -Описание входов и выходов ФБ NBS. TCP_Write

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

2.6.2.4.6 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.

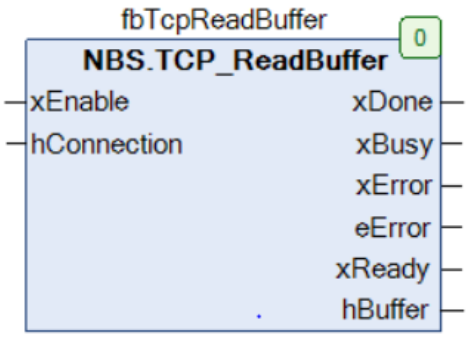


Рисунок 35 – Внешний вид ФБ NBS.TCP_ReadBuffer на языке CFC

Таблица 19 - Описание входов и выходов ФБ NBS. TCP_ReadBuffer

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

2.6.2.4.7 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.

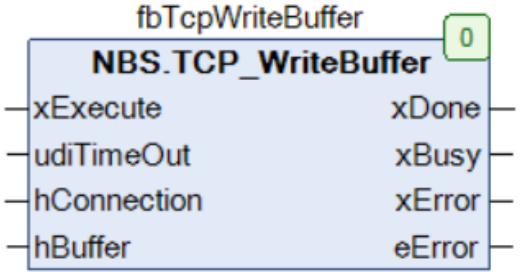


Рисунок 36 – Внешний вид ФБ NBS.TCP_WriteBuffer на языке CFC

Таблица 20 - Описание входов и выходов ФБ NBS. TCP_WriteBuffer

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
udiTimeOut	UDINT	Допустимое время операции (в мкс). Значение 0 означает, что время выполнения ФБ не ограничивается
hPeer	NBS.CAA.HANDLE	Дескриптор сокета устройства
ipAddr	NBS.IP_ADDR	IP-адрес получателя
uiPort	UINT	Номер порта получателя
hBuffer	NBS.CAA.HANDLE	Дескриптор буфера отправляемых данных
Выходные переменные		
xDone	BOOL	Флаг завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение TRUE в случае возникновения ошибки
eError	NBS.ERROR	Статус работы ФБ (или имя ошибки)

2.6.2.5 Дополнительные функции

2.6.2.5.1 Функция NBS.IPSTRING_TO_UDINT

Функция NBS.IPSTRING_TO_UDINT конвертирует строковое представление IP-адреса в бинарное (’10.2.11.10’ --> 16#0A02B00A).

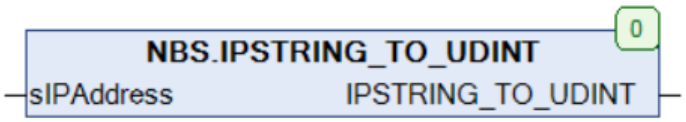


Рисунок 37 – Внешний вид функции NBS.IPSTRING_TO_UDINT на языке CFC

Таблица 21 - Описание входов и выходов функции NBS. IPSTRING_TO_UDINT

Название	Тип данных	Описание
Входные переменные		
stIPAdress	NBS.IP_ADDR	Строковое представление IP-адреса
Выходные переменные		
IPSTRING_TO_UDINT	UDINT	Бинарное представление IP-адреса

2.6.2.5.2 Функция NBS.IPSTRING_TO_UDINT

Функция NBS.UDINT_TO_IPSTRING конвертирует бинарное представление IP-адреса в строковое (16#0A02B00A --> ’10.2.11.10’).

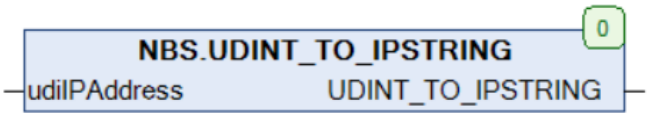


Рисунок 38 – Внешний вид функции NBS. UDINT_TO_IPSTRING на языке CFC

Таблица 22 - Описание входов и выходов функции NBS. UDINT_TO_IPSTRING

Название	Тип данных	Описание
Входные переменные		
udiIPAdress	UDINT	Бинарное представление IP-адреса
Выходные переменные		
UDINT_TO_IPSTRING	NBS.IP_ADDR	Строковое представление IP-адреса

2.6.2.5.3 Функция NBS.IS_MULTICAST_GROUP

Функция NBS.IS_MULTICAST_GROUP возвращает TRUE, если указанный IP-адрес является адресом Multicast-группы.

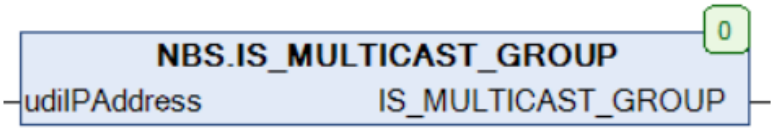


Рисунок 39 – Внешний вид функции NBS.IS_MULTICAST_GROUP на языке CFC

Таблица 23 – Описание входов и выходов функции NBS. IS_MULTICAST_GROUP

Название	Тип данных	Описание
Входные переменные		
udiIPAdress	UDINT	Бинарное представление IP-адреса
Выходные переменные		
S_MULTICAST_GROUP	BOOL	Флаг «Multicast-адрес»

2.6.3 Примеры работы с библиотекой CAA Net Base Services

2.6.3.1 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, в случае отправления на сервер строки ‘hello’ клиент получит в ответ строку ‘olleh’.

Примеры созданы в среде CODESYS V3.5 SP17 Patch 3 и подразумевают запуск на виртуальном контроллере CODESYS Control Win V3, который входит в состав 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 следует нажать ПКМ на узел Application и выбрать команду Установить активное приложение (или использовать выпадающий список на панели меню).

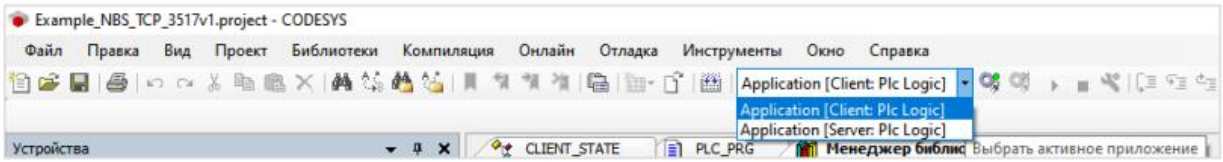


Рисунок 40 – Выбор активного приложения в проекте CODESYS

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

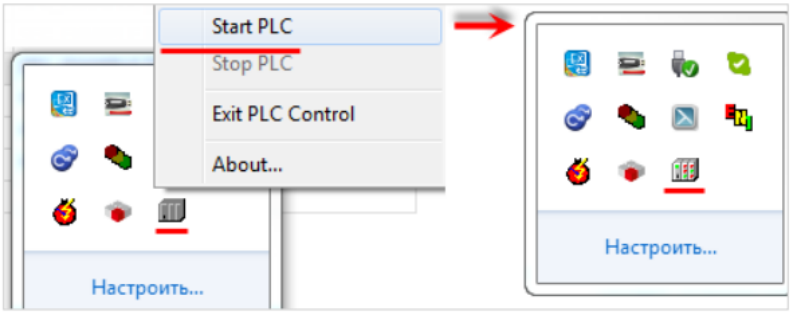


Рисунок 41 – Запуск виртуального контроллера

2.6.3.2 Реализация UDP-сервера и UDP-клиента

2.6.3.2.1 Основная информация

В данном примере рассматривается реализация UDP-сервера и UDP-клиента. Решаемая задача описана в п. 2.6.3.1.

2.6.3.2.2 Реализация UDP-сервера

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

Сначала следует создать функцию 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 рекомендуется копировать ее функции и ФБ в пользовательский проект (вместо добавления через Менеджер библиотек).

Функция инверсии строки называется MIRROR. Функцию следует скопировать в свой проект и удалить константы STRING_LENGTH, определяющие максимальную длину строк (чтобы не копировать из библиотеки дополнительные POU):

```
// (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


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;
```

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.

Данный 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 следует объявить перечисление SERVER_STATE (Application – Добавление объекта – DUT – Перечисление), в котором свяжем номера шагов с символьными именами.

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

Затем следует объявить в программе PLC_PRG переменные:

```
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
```

Примечание - Переменная uiPortServer определяет номер порта сервера.

Код программы выглядит следующим образом:

```
CASE eState OF

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

      fbUdpPeer
      (
         xEnable       := TRUE,
         ipAddr        := ,
         uiPort        := uiPortServer,
         // в рамках примера multicast не используется
         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);

        // очищаем буфер приема
        MEM.MemFill(ADR(sClientString), SIZEOF(sClientString), 0);

        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   := TO_UDINT(LEN(sInverseString) )
      );

      // если данные были 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
```

На шаге CREATE с помощью экземпляра ФБ NBS.UDP_Peer происходит создание серверного UDPсокета 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;

• очистка переменной sClientString с помощью функции MemFill из библиотеки CAA Memory. Это позволяет избежать «склеивания» строк, полученных от клиента (так как данные из запроса клиента копируются по указателю, и если размер данных нового запроса меньше, чем в предыдущем – то в буфере окажется смесь старых и новых данных);

• 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.

Примечание - Обратите внимание, что экземпляру ФБ NBS.UDP_Receive на вход szSize передает размер буфера приема (вычисленный помощью оператора SIZEOF), а экземпляру ФБ NBS.UDP_Send на этот вход передается число отправляемых байт (вычисленное с помощью функции LEN из библиотеки Standard – то есть передаются только символы строки без лишних «нулевых» байтов).

2.6.3.2.3 Реализация UDP-клиента

Задача UDP-клиента. – отправить запрос на сервер и получить ответ.

Как и в случае с сервером, алгоритм работы клиента представляется в виде последовательности шагов, выполняемых с помощью оператора CASE. Для использования символьных имен в качестве меток оператора CASE следует объявить перечисление CLIENT_STATE (Application – Добавление объекта – DUT – Перечисление), в котором номера шагов связываются с символьными именами.

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

Затем следует объявить в программе PLC_PRG переменные:

```
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

   stIpServer:     NBS.IP_ADDR := (sAddr:='10.2.8.133');    // IP-адрес сервера
                                  // (измените его на адрес вашего сервера)

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

   xSend:          BOOL;       // Команда отправки запроса
   fbSendTrig:     R_TRIG;     // Триггер отправки запроса
   fbResponseTimeout: TON;     // Таймер ожидания ответа
END_VAR
```

Переменные программы, определяющие настройки сервера и клиента:

• 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:

```
CASE eState OF

   CLIENT_STATE.CREATE:   // создаем UDP-пира на заданном порту

      fbUdpPeer
      (
         xEnable          := TRUE,
         ipAddr           := ,
         uiPort           := uiPortClient,
         // в рамках примера multicast не используется
         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   := TO_UDINT(LEN(sClientString) )
      );

      IF fbUdpSend.xDone THEN
         fbUdpSend(xExecute:=FALSE);
         fbResponseTimeout(IN:=FALSE);
         // очищаем буфер приема
        MEM.MemFill(ADR(sInverseString), SIZEOF(sInverseString), 0);
        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 или нет ответа - 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
```

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 10.2.8.133 и порт 4711. Номер порта соответствует порту, указанному при создании сокета на сервере). Запрос представляет собой строку sClientString. Если запрос успешно отправлен (xDone=TRUE), то происходит сброс таймера ожидания ответа, очистка буфера приема (ее необходимость поясняется в п. 4.2.2) 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 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.

Примечание - Обратите внимание, что экземпляру ФБ NBS.UDP_Receive на вход szSize передает размер буфера приема (вычисленный помощью оператора SIZEOF), а экземпляру ФБ NBS.UDP_Send на этот вход передается число отправляемых байт (вычисленное с помощью функции LEN из библиотеки Standard – то есть передаются только символы строки без лишних «нулевых» байтов).

Пример простой визуализации проекта:

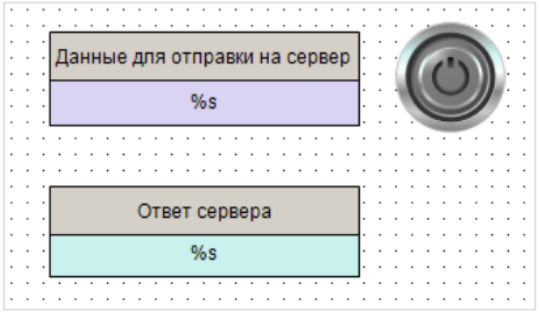


Рисунок 42 – Внешний вид визуализации клиента

К элементу Данные для отправки на сервер привязана переменная sClientString и настроена возможность ее изменения (вкладка InputConfiguration – OnMouseClick – 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.

2.6.3.3 Реализация TCP-сервера и TCP-клиента

2.6.3.3.1 Основная информация

В данном примере рассматривается реализация UDP-сервера и UDP-клиента. Решаемая задача описана в п. 2.6.2.6.1.

2.6.3.3.2 Реализация TCP-сервера

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

Сначала следует создать функцию инверсии строки. Такая функция уже есть в свободно распространяемой библиотеке OSCAT. Библиотека OSCAT имеет открытые исходные коды, поэтому во многих случаях рекомендуется копировать ее функции и ФБ в пользовательский проект (вместо добавления через Менеджер библиотек).

Функция инверсии строки называется MIRROR. Функцию следует скопировать в свой проект и удалить константы STRING_LENGTH, определяющие максимальную длину строк (чтобы не копировать из библиотеки дополнительные POU):

```
// (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


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;
```

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.

Данный 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 следует объявить перечисление SERVER_STATE (Application – Добавление объекта – DUT – Перечисление), в котором свяжем номера шагов с символьными именами.

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

Затем следует объявить в программе PLC_PRG переменные:

```
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
```

Примечание - Переменная uiPortServer определяет номер порта сервера.

Код программы выглядит следующим образом:

```
CASE eState OF

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

      fbUdpPeer
      (
         xEnable       := TRUE,
         ipAddr        := ,
         uiPort        := uiPortServer,
         // в рамках примера multicast не используется
         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);

        // очищаем буфер приема
        MEM.MemFill(ADR(sClientString), SIZEOF(sClientString), 0);

        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   := TO_UDINT(LEN(sInverseString) )
      );

      // если данные были 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
```

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;

• очистка переменной sClientString с помощью функции MemFill из библиотеки CAA Memory. Это позволяет избежать «склеивания» строк, полученных от клиента (так как данные из запроса клиента копируются по указателю, и если размер данных нового запроса меньше, чем в предыдущем – то в буфере окажется смесь старых и новых данных);

• 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.

Примечание - Обратите внимание, что экземпляру ФБ NBS.UDP_Receive на вход szSize передает размер буфера приема (вычисленный помощью оператора SIZEOF), а экземпляру ФБ NBS.UDP_Send на этот вход передается число отправляемых байт (вычисленное с помощью функции LEN из библиотеки Standard – то есть передаются только символы строки без лишних «нулевых» байтов.

2.6.3.3.3 Реализация TCP-клиента

Задача TCP-клиента – отправить запрос на сервер и получить ответ. Как и в случае с сервером, алгоритм работы клиента можно представить в виде последовательности шагов, выполняемых с помощью оператора CASE. Для использования символьных имен в качестве меток оператора CASE следует объявить перечисление CLIENT_STATE (Application – Добавление объекта – DUT – Перечисление), в котором номера шагов связываются с символьными именами.

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

Затем следует объявить в программе PLC_PRG переменные:

```
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

   stIpServer:           NBS.IP_ADDR := (sAddr:='10.2.8.133');      // IP-адрес сервера
   uiPortServer:         UINT := 4711;                              // Порт сервера

   xSend:                BOOL;            // Команда отправки запроса на сервер

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

Переменные программы, определяющие настройки сервера:

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

Код программы будет выглядеть следующим образом:

```
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      := TO_UDINT(LEN(sClientString) )
      );

      IF fbTcpWrite.xDone THEN
         fbTcpWrite(xExecute := FALSE);
         fbResponseTimeout(IN := FALSE);
         // очищаем буфер приема
         MEM.MemFill(ADR(sInverseString), SIZEOF(sInverseString), 0);
         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 или нет ответа - 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
```

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 сброс таймера ожидания ответа, очистка буфера приема (ее необходимость поясняется в п. 4.3.2) и 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.

Пример простой визуализации проекта:

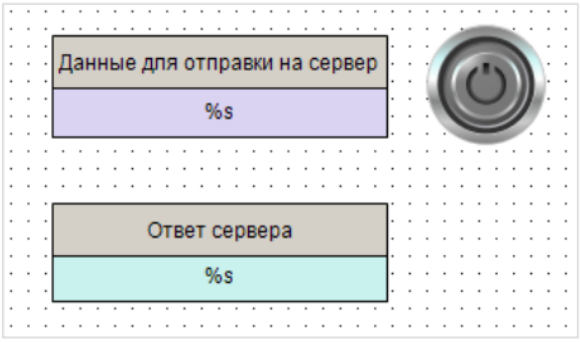


Рисунок 43 – Внешний вид визуализации клиента

К элементу Данные для отправки на сервер привязана переменная sClientString и настроена возможность ее изменения (вкладка InputConfiguration – OnMouseClick – 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.

Примечание - Обратите внимание, что экземпляру ФБ NBS.TCP_Read на вход szSize передает размер буфера приема (вычисленный помощью оператора SIZEOF), а экземпляру ФБ NBS.TCP_Write на этот вход передается число отправляемых байт (вычисленное с помощью функции LEN из библиотеки Standard – то есть передаются только символы строки без лишних «нулевых» байтов).

2.6.3.4 Работа с примером

2.6.3.4.1 Каждый пример содержит два приложения – Server и Client. В приложениях следует отредактировать:

• 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).

Сначала следует запустить 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». В случае необходимости можно изменить таргет-файлы приложений, чтобы запустить их на нужных устройствах (нажать ПКМ на узел Device – Обновить устройство).

2.6.3.4.2 В приложении Client следует перейти на страницу визуализации. Затем ввести строку данных, которая будет отправлена на сервер, и нажать кнопку. В поле Ответ сервера должна появиться инвертированная строка. Если строка не появляется – следует проверить корректность сетевых настроек устройств, на которых запускаются проекты, и уточнить особенности настроек сети (например, на маршрутизаторах могут быть заблокированы какие-то порты).

Рисунок 44 – Работа с примером

2.6.3.5 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.

• 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.

2.7 Язык Structured Text стандарта МЭК 61131-3

2.7.1 Структура программы

Основная структура программы, написанная на ST:

```
PROGRAM PLC_PRG
  VAR
    a : BOOL;
  END_VAR

  a := TRUE;

  REPEAT
    a := FALSE;
  UNTIL a = FALSE;
  END_REPEAT;

END_PROGRAM;
```

Программа начинается и заканчивается ключевыми словами PROGRAM и END_PROGRAM, а все, что между этими словами, программа ST.

Далее в примерах ключи PROGRAM и END_PROGRAM опущены, но они подразумеваются.

Открывающими и закрывающими ключами также могут быть:

• CONFIGURATION и END_CONFIGURATION для определения конфигурации;

• FUNCTION_BLOCK и END_FUNCTION_BLOCK для определения функционального блока;

• FUNCTION и END_FUNCTION для определения функции;

• И еще пара десятков подобных.

2.7.2 Порядок исполнения программы

Программа исполняется строка за строкой, пока не достигнет ключа завершения программы END_PROGRAM. Но ключ END_PROGRAM не остановит всю программу после ее исполнения. Как только программа достигнет этого ключа, она начнет исполняться снова. И так бесконечно в цикле, раз за разом.

Контроль управления программой осуществляет связанные с ней задачи. Как правило, основная программа исполняется циклически. Исключения составляют программы, которые вызываются однократно, привязанные к определенным событиям.

Основная программа работает по принципу работы циклов FOR или REPEAT и никогда не останавливается. Если отобразить языком С как работает программа, то получится примерно следующее:

```
InitVariables();
while(1)
{
  ReadInputs();

  PLC_PRG();
  WriteOutputs();
}
```

Последовательность работы, как правило, неизменна для всех КТСИ.

При старте КТСИ однократно исполнить InitVariables();, где инициализируются все переменные, которые определены во всех программах, глобальные переменные, функциональные блоки (ФБ), сетевые переменные а также входы и выходы КТСИ. Под них отводится и резервируется память к которой они привязываются автоматически, если не указано точного адреса памяти для привязки.

Регистры входов и выходов особые, они физически привязаны или к ножкам процессора, или к коммуникационной шине, и изменение значений этих регистров приведет к изменению физического состояния выхода КТСИ. И наоборот - подача сигнала на вход КТСИ приведет к изменению значения регистра входа.

Затем в цикле исполнить саму программу КТСИ с определенным интервалом. Это цикл, который никогда не прерывается, потому что 1 в while(1) всегда будет возвращать TRUE. Внутри этого цикла выполнить следующее:

- Сначала загрузить все состояния входов КТСИ в отведенные для них переменные. Другие - локальные и глобальные переменные, помнят свои значения с предыдущего цикла.

- Выполнить саму программу КТСИ.

- Присвоить выходам значения, обработанных программой, переменных связанных с выходами.

ST исполняется последовательно, строка за строкой. Следующая строка не будет исполнена, пока предыдущая не закончит исполнение. Такой стиль называется синхронным или блокирующим.

Примечание - исключения составляют функциональные блоки, программа не будет ждать окончания их работы, чтобы перейти к следующей строке.

Рассмотрим следующий пример:

```
VAR
  a AT %QX0.0.1 : BOOL;
END_VAR

a := TRUE;
act1()
a := FALSE;
act1()
a := TRUE;
```

Предположим, есть некая переменная a, привязанная к физическому выходу КТСИ по адресу %QX0.0.1. В теле программы поменять ее состояние с TRUE на FALSE и обратно. Между этим исполнить какие-то подпрограммы act1(). Допустим, время исполнения подпрограммы act1() 500ms. Так как нужно вызвать эту подпрограмму 2 раза, значит что время одного цикла КТСИ будет как минимум 1000ms.

Выход КТСИ не будет менять свое состояние в FALSE на 500ms каждый цикл работы контроллера, так как назначения выходов происходит только тогда, когда программа завершит работу, а значит, выходу или адресу %QX0.0.1 будет присвоено только то значение, которое будет в переменной a на момент завершения программы, а все промежуточные значения будут проигнорированы.

Это дает возможность использовать определенный паттерн программирования.

Например, вместо:

```
IF b > 10 THEN
  a := TRUE;
ELSE
  a := FALSE;
END_IF
```

можно смело сделать:

```
a := FALSE;
IF b > 10 THEN
  a := TRUE;
END_IF
```

Конечно, в данном случае идеальным преобразованием будет просто одна строка a := (b > 10);, но вот какой принцип я хочу продемонстрировать этим примером.

Разница в примерах в том, что в первом из них переменной a присваивается значение FALSE только если переменная b меньше 10. А во втором примере переменной a присваивается значение FALSE каждый цикл программы, хоть и на короткий промежуток времени, не зависимо от значения переменной b, даже если b больше 10.

Не важно, какой промежуток времени прошел между присвоениями переменной a разных значений, выход КТСИ не будет менять свое состояние. Подробнее об этом поговорим, изучая условия.

2.7.3 Элементы высокого уровня

На схеме ниже можно увидеть элементы высокого уровня и то, как они относятся друг ко другу. Эти элементы можно программировать на ST.

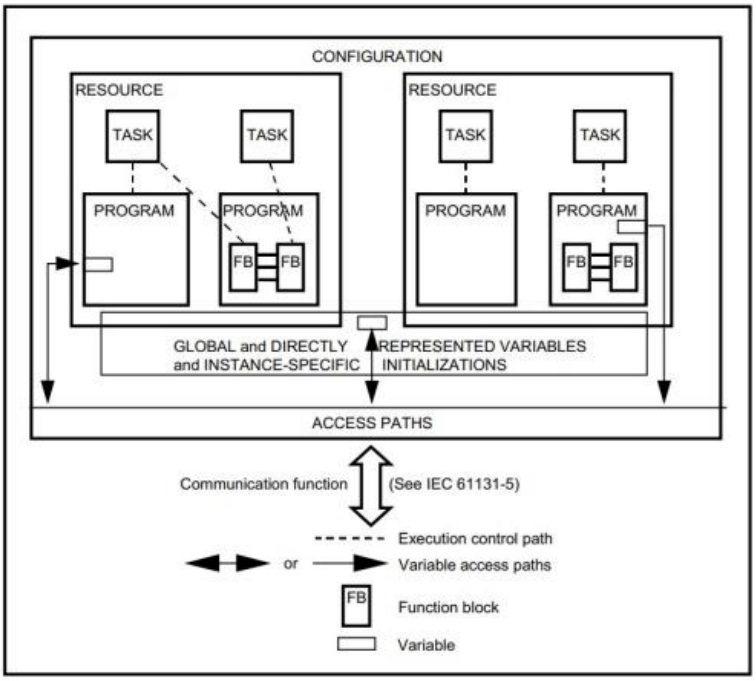


Рисунок 45

Элемент CONFIGURATION - это элемент языка, который соответствует системе КТСИ. Конфигурация содержит один или несколько элементов RESOURCE, каждый из которых содержит задачи TASK, которые запускают программы PROGRAM.

Например:

```
CONFIGURATION DefaultCfg
  VAR_GLOBAL
    Start_Stop AT %IX0.0: BOOL
    ON_OFF     AT %QX0.0: BOOL;
  END_VAR
  RESOURCE Resource1
    TASK NewTask  (INTERVAL := t#20ms);
    PROGRAM Main WITH NewTask : PLC_PRG;
  END_RESOURCE
END_CONFIGURATION
```

Обычно производители КТСИ генерируют файл конфигурации автоматически, а для его изменения создают визуальный интерфейс. Так что, скорее всего, вам напрямую не придется столкнуться с работой с подобными файлами. Например, вызов задачи в Codesys.

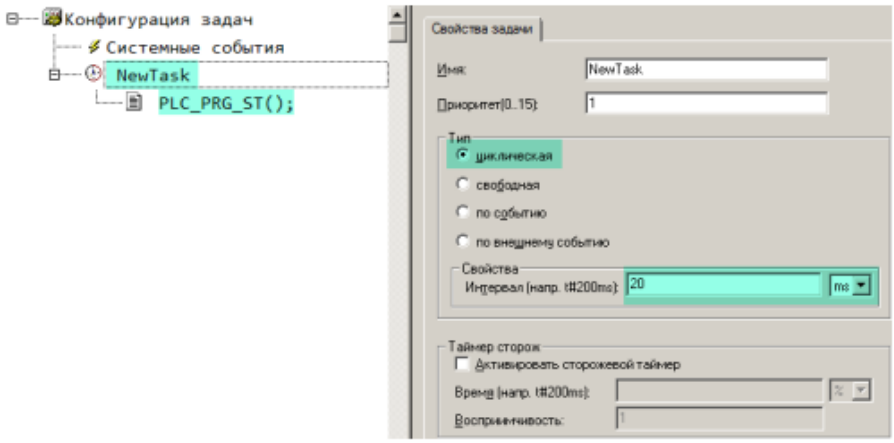


Рисунок 46

В конфигурации выглядит вот так:

```
TASK NewTask  (INTERVAL := t#20ms);
PROGRAM Main WITH NewTask : PLC_PRG_ST;
```

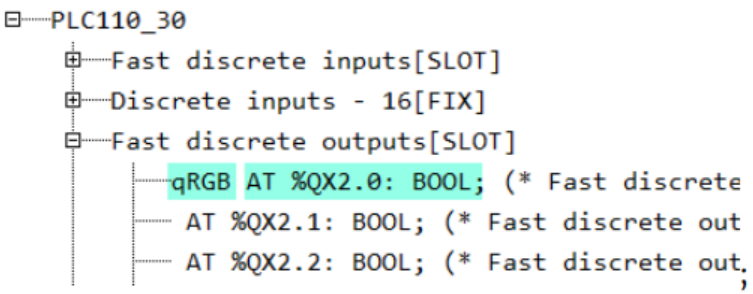
Вы можете сами определить время цикла. Например, 20ms. Это значит, что если программа закончится раньше, чем через 20ms, то контроллер подождет, пока это время истечет, чтобы запустить следующий цикл программы. Можно параллельно вызвать несколько программ, по разным циклам или событиям. Хотя параллельный запуск программ поддерживается не всеми IDE или не все CPU способны с этим справиться. Поэтому, если вам нужно запустить больше одной программы параллельно, обязательно узнайте у производителя КТСИ, что их продукция поддерживает подобную возможность.

Примечание - Время исполнения нужно настраивать таким образом, чтобы его точно хватило на выполнение программы. Для этого с помощью соответствующих средств IDE нужно обязательно определить время исполнения программы.

2.7.4 Глобальные переменные

В конфигурации объявляются глобальные переменные. Например, в CoDeSys

присваиваем имя qRGB для выхода контроллера.



При компиляции программы конфигурация будет создана автоматически, и эта переменная будет добавлена в нее как глобальная.

```
VAR_GLOBAL
  qRGB AT %QX2.0: BOOL;
END_VAR
```

2.7.5 Синтаксис

Синтаксис ST или любого другого языка - это набор правил написания инструкций, определяющий значение и форму языка. Синтаксис определяет то, каким образом и в какой последовательности тот или иной символ или их набор будет интерпретирован в программе.

Разные символы, такие как точка с запятой ;, двоеточие : и даже невидимый символ новой строки - все имеет свое значение и цель. Что-то является оператором, что-то функцией, что-то определением, и все вместе формирует программу.

Далее рассмотрим все отдельные части синтаксиса в деталях. Есть и общие правила:

- Каждая строка, инструкция заканчивается точкой с запятой ;. ST весь состоит из инструкций и ; нужны чтобы их разделять.

- ST не чувствителен к регистру, хотя это хорошая практика - использовать регистры для простоты чтения программы.

- Пробел не имеет никакой функции, но его следует использовать для форматирования программы, для простоты ее чтения.

2.7.6 Компилятор

Прежде чем закачать программу на КТСИ, IDE переведет ST (или любой другой язык МЭК 61131) в машинный код напрямую, или иногда используя промежуточный язык. В большинстве случаев таким языком является IL, но иногда и сам ST является промежуточным языком для компиляции графических языков в машинный код.

Это будет сделано компилятором. Другими словами, вы используете ST для объяснения компилятору того, что вы хотите, а компилятор использует синтаксис ST для того, чтобы вас понять.

Например, если он увидит символ ; то поймет, что достиг конца инструкции. Компилятор прочтет инструкцию до конца и только потом превратит его в команду для исполнения.

2.7.7 Комментарии

В программе при использовании языка ST можно написать текст, который не будет исполнен. Эта возможность используется для вставки комментариев.

```
// Комментарий одной строкой

a := 10; (* Комментарий в конце строки *)
a := 10; /* Комментарий в конце строки */
a := 10; // Комментарий в конце строки

(* Многострочный
комментарий *)

/* Многострочный
комментарий */
```

Комментарии полезны не только для того чтобы комментировать код и писать описания идей, заложенных в алгоритмы логики. Комментарии можно использовать, чтобы закомментировать участок кода и сделать его не исполняемым без его удаления. Например, есть код для отладки или проверки программы во время симуляции, но который не нужен при запуске рабочей программы на КТСИ и который вы не хотите удалять и оставить для отладки в случае поломки.

Если у вас хорошая память, то вы можете комментировать минимально или вообще не комментировать. Но у памяти есть свойство забывать. Моя рекомендация - использовать инструмент комментирования в меру. Не обязательно описывать всю идею. Иногда, когда разбираешься в чужом коде, достаточно лишь одной строки, в которой изложен принцип, скрывающийся за текущим выражением. Привычка комментировать код обязательно сбережет вам несколько часов времени и несколько десятков нервных клеток.

2.7.8 Инструкции

ST состоит из инструкций (statement).

Инструкция – это команда для КТСИ о том, что ему делать.

Рассмотрим первую инструкцию. В качестве примера возьмем объявление переменной, что само по себе так же является инструкцией.

```
a : BOOL;
```

Компилятор прочтет это, увидит в конце ;, и поймет, что это инструкция. Закончив анализ самой инструкции, поймет, что нужно объявить переменную типа BOOL (булев тип) и отделить для нее место в памяти на один бит.

Еще примеры инструкций:

```
a := TRUE;
```

или

```
ton1(IN := TRUE, PT := T#500ms);
```

Так как пустые символы не расцениваются языком ST, то не обязательно начинать новую инструкцию с новой строки. В одной строке может быть несколько инструкций разделенных символом ;.

```
a := TRUE; ton1(IN := TRUE, PT := T#500ms);
```

А так же инструкции могут быть разбиты на несколько строк.

```
a := TRUE;

ton1(
        IN := TRUE,
        PT := T#500ms

);
```

Вызов функции или функционального блока тоже считается инструкцией. Даже полная конструкция условия от IF до END_IF, считается инструкцией, хотя и может содержать инструкции внутри себя.

2.7.9 Выражения

Выражение (expression) – это конструкция языка ST, которая состоит из операндов и операторов и производит значение определенного типа.

Операнд – это объект, над которым оператор производит действие.

Например:

```
a := (b - c) * 100 + ADD(b, c);
```

- инструкция a := (b - c) * 100 + ADD(b, c);

- выражение (b - c) * 100 + ADD(b, c)

- операнды b, c, 100 и ADD(b, c). Функции или, скорее, производные функций, тоже считаются операндами.

- операторы (,), -, * и ADD, так как все функции можно считать оператором, если они используются в выражении.

2.7.10 Операторы

Операторы в ST можно разбить на 4 группы. Каждая группа имеет свою специфику и производит специфичный тип данных.

2.7.10.1 Арифметические операторы (Arithmetic)

Все арифметические операторы, как правило, применяются для записи и осуществления математических вычислений. Результатом их использования всегда будет проведение математических вычислений и получение соответствующего числового значения.

- + сложение (add);
- – вычитание (subtract/negate);
- * умножение (multiply);
- ** возведение в степень (exponent);
- / деление (divide);
- MOD остаток от деления (modulo divide).

```
a := 15 MOD 4 (* результат 3 *)
```

2.7.10.2 Операторы Сравнения (Relational)

Для сравнения или определения отношения между значениями двух переменных или выражений. Результат всегда будет булевым, TRUE или FALSE.

- = равно (EQ);
- < меньше чем (LT);
- <= меньше чем или равно (LE);
- > больше чем (GT);
- >= больше чем или равно (GE);
- <> не равно (NE).

Использовать можно как символы, так и буквенные обозначения. Последние 2 строки в примере выполняют одно и тоже и их результат будет одинаковый.

```
a := 15;
b := 15;
c := a = b;  (* результат TRUE *)
c := a EQ b; (* результат TRUE *)
```

2.7.10.3 Логические операторы (Logical)

Используются, если нужно, работая с булевыми значениями, реализовывать логические выражения. Результат также всегда будет булевым, TRUE или FALSE.

- & и AND логическое И;
- OR логическое ИЛИ;
- XOR ИЛИ с отрицанием;
- NOT отрицание или негатив, инвертирование переменной. Из TRUE в FALSE, либо наоборот.

```
a := TRUE;
b := FALSE;
c := a OR b;      (* результат TRUE *)
c := a AND b;     (* результат FALSE *)
c := NOT a OR b; (* результат FALSE *)
```

Логические операторы часто используются в сочетании с операторами сравнения. Так как оператор сравнения производит булево значение, то далее это значение можно использовать в получении результатов логических операций. Проверяем, что значение b находится между значениями a и c.

```
a := 10;
b := 20;
c := 30;
d := ((b > a ) AND (b < C)); (* результат TRUE *)
```

2.7.10.4 Битовые операторы (Bitwise)

Битовые операторы вычисляют результат побитно. Это значит, что простая логическая операция производится для каждого бита отдельно. Логические и битовые операторы выглядят одинаково и применяются автоматически. Если хотя бы один операнд в выражении будет булевым, то автоматически будет произведено вычисление булевым логическим оператором, а если оба операнда в выражении будут битовыми, как BYTE, WORD, DWORD, и т.д., то будет произведено битовое вычисление. Результат - это число, сумма битовых операций.

- & и AND логическое И;
- OR логическое ИЛИ;
- XOR ИЛИ с отрицанием;
- NOT отрицание или негатив, инвертирование переменной. Из TRUE в FALSE, либо наоборот.

```
var1 := (2#1001_0011 AND 2#1000_1010) (* Результат 2#1000_0010 *)
```

Так как у оператора AND с двух сторон байтовая переменная, применяется битовый оператор. Если вы помните выполнение сложения столбиком, то он работает по тому же принципу. Сначала оператором AND сравнивается первый бит в одном и в другом операторе и сохраняется в первый бит результата. И так - бит за битом.

- 2#1001_0011 (147);
- 2#1000_1010 (138);
- 2#1000_0010 (130).

Как понятно из примера, в производном 2-х байт оператором AND, битами равными 1 будут те, где они равны 1 в правом И в левом операнде, или верхнем и нижнем, если мы смотрим на это как на вычисление столбцом.

2.7.10.5 Числовые Операторы (Numeric)

Числовое операторы производят основные математические вычисления. Результат всегда будет числом типа ANY_REAL, поэтому присваивать значения нужно переменным типа ANY_REAL или использовать конвертацию. Исключение составляет оператор ABS. Он вернет целое число ANY_INT.

- ABS возвращает абсолютное число "модуль" значения a := ABS(4.5); будет 4;
- ACOS возвращает арккосинус;
- ASIN возвращает арксинус;
- ATAN возвращает арктангенс;
- COS возвращает косинус;
- EXP возводит в степень;
- EXPT возводит в степень с указанием степени;
- LN возвращает натуральный логарифм;
- LOG возвращает логарифм;
- SIN возвращает синус;
- SQRT возвращает квадратный корень;
- TAN возвращает тангенс.

Эти операторы работают со следующими группами типов переменных: ANY_NUM (ANY_INT, ANY_REAL).

```
cr := 20;                (* радиус круга *)
cs := 3.14 * EXPT(cr, 2); (* площадь круга Пи Эр квадрат *)
```

2.7.10.6 Операторы выбора (Selection)

- SEL - бинарный выбор. Бинарный, потому что переменная, по которой этот выбор осуществляется, должна быть булева типа. В примере это b.

```
a: = SEL(b, in1, in2);
```

Если b будет равно TRUE, то a будет присвоено in2, а если b будет FALSE, то a будет присвоено in1. Переменные in* могут быть любого типа.

- MAX - возвращает большее число из переданных. Стандарт не ограничивает количество передаваемых значений, но если вы используете CoDeSys вы наткнетесь на ограничение в 2 значения. Тоже самое касается следующего оператора.

```
a := 10;
b := 20;
c := MAX(a, b); (* Результат будет 20 *)
```

- MIN - возвращает меньшее число из переданных 2-х.
- LIMIT - ограничение значений переменной по минимуму и максимуму.

```
a := LIMIT(10, b, 20);
```

В данном примере, если b находится между 10 и 20, то a будет присвоено значение b. Если значение b меньше минимума 10, то a будет присвоено 10, а если значение b будет больше максимума 20, то a будет присвоено 20.

- MUX - мультиплексор. Это хитрая штука - не сразу понятно, для чего нужна, но со временем вы сами увидите, куда ее вставить. Но как правило, в текстовом языке, этот оператор применяется очень редко. Я ни разу не применял.

```
a := MUX(k, in0, in1, in2.........);
```

a и in0 - in* должны быть одного типа. k может быть ANY_INT, ANY_BIT. a будет присвоено значение in* по номеру в переменной k.

```
k : 1;
a := MUX(k, 10, 20, 30);
```

a будет равно 20, так как k начинается с 0-го индекса.

2.7.10.7 Операторы смещения бита (Bitshift)

- SHL - SHR - побитное смещение влево или вправо на заданное количество бит.

```
a := 2#0001_0100; (* 20 в десятичной системе *)
b := SHL(a, 1);   (* 2#0010_1000 или 40 в десятичной *)
c := SHR(a, 2);   (* 2#0000_0101 или 5 в десятичной *)
```

Если сместить все биты влево на 1, то в b будет 2#0010_1000 или 40 в десятичной системе. Если сместить на 2 бита вправо, то с будет 2#0000_0101 или 5 в десятичной системе.

Примечание – Если длина входной переменной меньше, чем предполагаемое смещение, то можно получить некорректные данные.

Подробнее. Для большей очевидности, допустим, у BYTE 2#1000_0000, это 128 в десятичной системе.

```
VAR
    a: BYTE := 2#1000_0000;
    b: WORD;
END_VAR
b := SHL(a, 1);                (* Результат 0 *)
b := SHL(BYTE_TO_WORD(a), 1); (* Результат 256 *)
```

Если сдвинуть на 1 бит влево SHL, то первая единица уйдет за пределы, а справа новый бит заполнится нулем и мы получим 2#0000_0000 или 0. А, возможно, мы хотели получить 2#0000_0001_0000_0000 или 256. Автоматически этого не произойдет, даже если мы присваиваем результат переменной типа WORD. Нужно обязательно предварительно преобразовать переменную перед операцией смещения.

- ROL - ROR это ротация влево или вправо. То же побитное смещение, только скрывающийся вправо или влево бит назначается новому биту слева или справа. Представьте, что число закольцовано и оно просто крутится.

2.7.10.8 Другие операторы

Все ниже перечисленные операторы, не описаны стандартом МЭК, но некоторые ведущие IDE, такие как CoDeSys, TwinCAT, их поддерживают.

- INDEXOF - с помощью этого оператора можно узнать индекс программного объекта(POU).

```
a := INDEXOF(TON_1);
```

- SIZEOF - позволяет узнать размер или количество байт, отведенных для этой переменной.

```
VAR
    ar: ARRAY[0..4] OF INT;
    c: INT;
END_VAR

c := SIZEOF(ar);
(* результат будет 10. По 2 байта на каждый элемент массива *)
```

- ADR - возвращает адрес байта в памяти КТСИ, где хранится входная переменная. В основном, этот оператор используется для привязки переменной к указателю. Подробнее этот оператор будет рассмотрен в главе об указателях.
- ADRINST - возвращает экземпляр ФБ или POU.
- ^ - снятие косвенности (Dereferencing) или разыменовывание. Используется для получения значения переменной, на которую ссылается указатель.
- BITADR - возвращает адрес бита.

2.7.11 Общие принципы работы операторов

2.7.11.1 Старшинство

Операторы применяются к операндам последовательно, по определению старшинства. Самый старший оператор в выражении будет применен первым, далее - младшие, и так до конца выражения. Операторы одинакового старшинства применяются слева направо.

Например, есть операнды a, b и c типа INT со значениями 1, 2 и 4, соответственно. Тогда выражение a + b - c * (c / b) будет оценено следующей последовательностью:

- скобки (c / b) имеют высшее старшинство, поэтому будут посчитаны первыми (4 - 2) и получится 2;

- умножение c*(2) будет следующим, потому что имеет следующее старшинство 4*(2) и получится 8;

- сложение a + b и вычитание b - 8 имеют одинаковое старшинство и поэтому будут исполняться по очереди слева направо. Сначала сложение 1 + 2, получится 3, а потом вычитание 3 - 8, получится -5.

Таблица 23 - Таблица операторов по старшинству

№	Описание	Пример	Старшинство
1	Скобки	A + (B / C)	11
2	Вызов функции	ABS(A), ADD(X, Y)	10
3	Снятие косвенности ^	a^	9
4	Унарный минус	-A	8
5	Унарный плюс	+A	8
6	Отрицание NOT	NOT A	8
7	Возведение в степень **	A ** B	7
8	Умножение *	A * B	6

№	Описание	Пример	Старшинство
9	Деление /	A / B	6
10	Остаток от деления	A MOD B	6
11	Сложение	A + B	5
12	Вычитание	A - B	5
13	Сравнение (, <=, >=)	A < B	4
14	Сравнение (=)	A = B	4
15	Неравенство <>	A <> B	4
16а	Булево &	A & B	3
16б	Булево AND	A AND B	3
17	Булево XOR	A XOR B	2
18	Булево OR	A OR B	1

2.7.11.2 Два операнда

```
(a EQ b) = (a AND b)
```

Выражение (a EQ b) будет оценен первым, а (a AND b) вторым. Это и есть выражение само по себе со своими операторами и операндами, но внутри другого выражения рассматривается как единое целое с производным результатом, а это уже операнд. Другими словами, это выражение, но для оператора сравнения = - операнд.

2.7.11.3 Сокращенные вычисления

Сокращенные вычисления - это когда выражения с булевыми переменными оцениваются только до точки, где результат уже понятен. Например:

```
IF (a > b) AND (b > 10) THEN
    a := b;
END_IF
```

Оцениваем сравнение AND. Это значит, что операнды слева и справа должны быть равны TRUE. Если хотя бы один из операндов вернет FALSE, то оценивать другие операнды уже не имеет смысла, так как их конъюнкция все равно будет FALSE.

Например, если операнд слева, который оценивается первым, (a > b) вернет FALSE, то уже нет смысла оценивать операнд справа, (b > 10). Ведь даже если (b > 10) будет TRUE,

этого все равно не достаточно, чтобы условие сработало. Поэтому анализ выражения прекратится на (a > b). Далее пример с оператором OR:

```
IF (a > b) OR (b > 10) THEN
    a := b;
END_IF
```

Если (a > b) будет равен TRUE, то уже не нужно оценивать следующий операнд, ведь результат сравнения все равно будет TRUE.

К сожалению, подобные сокращенные вычисления не оговорены в стандарте ST. В ST выражения как справа так и слева AND, OR оцениваются все условия всегда. Но нам все равно важно, в какой последовательности условия вписываются в выражение, по следующим причинам:

1 зная, что в некоторых других языках это работает, это поможет выработать хорошую практику написания кода и стиль;

2 в ExST существует альтернатива в виде операторов AND_THEN и OR_ELSE.

Операторы AND_THEN и OR_ELSE не являются частью стандарта МЭК 61131-3 а являются расширением языка ST. Обычно на него ссылаются как на ExST (Extended ST). К радости, эти операторы внедрены многими вендорами. Это CoDeSys 3.5 а значит и все основанные на CoDeSys IDE как например e!COCKPIT от WAGO, TwinCat от Beckhoff, Machine Expert от Schneider и другие ведущие вендоры. Одним словом, шансы высоки, что в той IDE в которой вы будет работать, будет включена поддержка этих операторов.

Если в условиях есть функции, которые выполняют сложные расчеты, их можно располагать последними, и тогда в половине случаев их вычисления даже не будут проводиться.

Например, имеется выражение с логическим оператором, где используется функция MYCOUNT(c, d), расчет которой занимает 1 миллисекунду.

```
IF MYCOUNT(c, d) AND_THEN a THEN
    b := TRUE;
END_IF
```

Данная конструкция будет занимать 1 миллисекунду времени каждый цикл КТСИ, не важно а будет TRUE или FALSE. А если развернуть операнды:

```
IF a AND_THEN MYCOUNT(c, d) THEN
    b := TRUE;
END_IF
```

В этом случае, если а будет равно FALSE, проверка условия займет 1 микросекунду или 0.001 миллисекунды, так как MYCOUNT(c, d) даже не будет вычисляться.

Правило здесь простое, в выражениях сравнения оператором AND_THEN, первыми

ставим те операнды, которые чаще всего возвращают FALSE чтобы исключить дальнейший анализ выражения, а в выражениях сравнения оператором OR_ELSE, первым ставим тот, что чаще всего вернет TRUE.

2.7.12 Условия

2.7.12.1 IF

IF позволяет программе принимать решения - при каких условиях и что требуется исполнить. Я считаю, что IF - наиболее интуитивно понятная конструкция условия.

```
IF [булево выражение] THEN
    <инструкции>;
ELSIF [булево выражение] THEN
    <инструкции>;
ELSE
    <инструкции>;
END_IF;
```

Вся конструкция блока от IF до END_IF сама по себе является инструкцией, хотя и содержит инструкции внутри себя, так что символ конца инструкции ; можно поставить после END_IF. Но, как правило, это не обязательно. После THEN, ELSE и ELSIF символ ; не ставится.

Рассмотрим пример. Если температура на входе rT1 поднялась выше 100 градусов, подаем сигнал на аварийный выход xAlert1:

```
IF rT1 > 100.0 THEN
    xAlert1 := TRUE;
ELSE
    xAlert1 := FALSE;
END_IF
```

2.7.12.1.1 Паттерны IF

Так как конструкция IF самая важная для любой логики, то все паттерны в основном крутятся вокруг этой конструкции.

2.7.12.1.1.1 Усреднение

Применив паттерн усреднения к нашему примеру, код можно сократить до:

```
xAlert1 := (rT1 > 100.0);
```

Данный паттерн можно применить везде, где внутри условий идет назначение одних и тех же булевых переменных. Применять подобный паттерн можно достаточно часто, так как результат выражения в условии конструкции IF булев и ее можно напрямую назначить всем булевым переменным.

Паттерн усреднения можно использовать не везде, так как внутри конструкций IF могут быть не только булевы значения, но и некая логика, что делает невозможным полностью избавиться от конструкции IF.

2.7.12.1.1.2 Предварительное отрицание

Еще один паттерн, "паттерн предварительного отрицания".

```
xAlert1 := FALSE;
IF rT1 > 100.0 THEN
    xAlert1 := TRUE;
END_IF
```

Идея этого паттерна в том, что сбрасываются все булевы переменные, с которыми работаем внутри конструкции IF, в FALSE перед условием, а потом используем только условия, которые изменят переменную в TRUE. Или наоборот - в зависимости от того, какой приоритет у данной переменной. Такой подход допустим, так как не произойдет отключение выхода на миллисекунды, если переменная привязана к выходу.

Преимущества этого паттерна: код легче читать, он содержит меньше строк, его работу проще понимать.

2.7.12.1.1.3 Примеры паттернов

Сравним 2 примера из реального кода управления установки сублимационной сушки. Оба они делают одно и тоже: включают компрессор охладителя и нужный клапан режима - на холод или на тепло.

2.7.12.1.1.3.1 Логический код

Под логическим подразумевается ход мысли: как перевести в код условие, сказанное словами. Имеются переменные:

- xCoolerOn - Сигнал с панели на включение охладителя
- xSpIndCooler - Индикатор на панели работы охладителя
- xIRKF - Сигнал с реле контроля фаз
- qCooler - Выход, включающий охладитель
- qValveCoolerHeat - Выход, режим работы охладителя на нагрев
- qValveCoolerCool - Выход, режим работы охладителя на охлаждение

- xSpCoolerModeHeat - Условие с панели, режим работы охладителя - нагрев или охлаждение

Допустим, есть условие, которое словами можно выразить так:

Если подан сигнал с панели и реле контроля фаз в порядке, то включаем охладитель, а если нет - выключаем охладитель. Если включаем охладитель и режим установлен на охлаждение, то переключаем на холод; в другом случае - на тепло.

Вот как это описывается в коде:

```
(* Если подан сигнал с панели и реле контроля фаз в порядке, *)
IF xCoolerOn AND xIRKF THEN

    (* то включаем охладитель *)
    qCooler := xSpIndCooler := TRUE;

    (* Если включаем охладитель и режим установлен на
    охлаждение, то переключаем на холод *)
    IF xSpCoolerModeHeat = FALSE THEN
        qValveCoolerCool := TRUE;  (* Режим холод *)
        qValveCoolerHeat := FALSE; (* Режим нагрев *)
     (* в другом случае на тепло *)
     ELSE
         qValveCoolerHeat := TRUE;  (* Режим нагрев *)
         qValveCoolerCool := FALSE; (* Режим холод *)
     END_IF

(* а если нет, то выключаем охладитель *)
ELSE
    qCooler := xSpIndCooler := FALSE;
    qValveCoolerCool := qValveCoolerHeat := FALSE;
END_IF;
```

Посмотрите на комментарий в коде: это в точности наше условие. На самом деле, был скопирован текст из условия, чтобы сделать комментарии. Код написан так, как мы словами высказываем само условие.

2.7.12.1.1.3.2 Паттерн предварительного отрицания

Тот же код, но с примененным паттерном предварительного отрицания.

```
qValveCoolerCool := qValveCoolerHeat :=
qCooler := xSpIndCooler := FALSE;

IF xCoolerOn AND xIRKF THEN
    qCooler := xSpIndCooler := TRUE;

    IF xSpCoolerModeHeat THEN
        qValveCoolerHeat := TRUE; (* Режим нагрев *)
    ELSE
        qValveCoolerCool := TRUE; (* Режим холод *)
    END_IF
END_IF;
```

Сначала первый пример может показаться более информативным и логичным, потому что напрямую отображает ход мысли. Но обратите внимание, что второй пример написан с применением паттерна предварительного отрицания. Если принять это в расчет, очевидно, что второй пример намного прозрачнее и легче. Даже буквально - в нем меньше строк.

В коде первого примера читаем в первом условии qCooler := xSpIndCooler := TRUE; И сразу вопрос: а когда они будут FALSE? Где в коде это условие находится? Нужно пересмотреть весь код программы, чтобы найти ответ и убедиться, что эта переменная нигде больше не назначается. Это небольшой пример, но при работе с файлом в 2000 строк будет совсем непросто прослеживать логику, даже если вы являетесь автором программы. И тем более, если программу писали не вы.

Но если посмотреть на то же условие qCooler := xSpIndCooler := TRUE; в примере с предварительным отрицанием, не возникает никаких вопросов. Происходит понимание того, что это и есть то условие, при котором эти переменные включатся. Во всех остальных случаях они будут равны FALSE, так как уже известно или имеется в виду, что все переменные входят в условие в состоянии FALSE.

Мы не ищем, где эти переменные будут отключаться, а подразумеваем их отключение в начале любой логики. Во втором примере состояние переменных FALSE - это входное условие, а в первом - вопрос, на который нужно получить ответ.

2.7.12.1.1.3.3 Паттерн усреднения

Если сюда применить еще и паттерн усреднения, то код сократится еще больше.

```
qValveCoolerCool := qValveCoolerHeat := qCooler := xSpIndCooler := FALSE;

IF xCoolerOn AND xIRKF THEN
    qCooler := xSpIndCooler := TRUE;
    qValveCoolerHeat := (xSpCoolerModeHeat);     (* Режим нагрев *)
    qValveCoolerCool := (NOT xSpCoolerModeHeat); (* Режим холод *)
END_IF;
```

Как видите, здесь и предварительное отрицание, и усреднение. Подобная техника придет не сразу, но с опытом вы будете писать все более лаконичный и стабильный код. Прочитав и поняв концепт подобных паттернов, вы в любом случае быстрее придете к подобному способу мышления.

Чем меньше строк, тем проще понять код. Меньше его реальный откомпилированный размер, занимаемый в памяти устройства, следовательно, теоретически, он должен работать быстрее.

2.7.12.1.1.3.4 Паттерн усреднения

Еще один важный паттерн - минимизация вложенности. Часто одно условие исполняется внутри другого условия, внутри которого могут быть еще условия. Большая вложенность условий часто делает код неудобочитаемым, а отступы уменьшают видимую часть кода по ширине.

К счастью, паттерн минимизации вложенности может улучшить ситуацию в большинстве случаев. Он работает, если условие является общим для всего POU, а такое не редкость.

Основная идея этого паттерна - использование ключа RETURN для прерывания кода POU. Ведь IF делает то же самое - исключает часть логики.

Любую конструкцию, где дополнительная вложенность условий находится в ELSE:

```
IF (A) THEN
    // Логика 1
ELSE
    IF (B) THEN
        // Логика 2
    END_IF
    // Логика 3
END_IF
```

можно преобразовать в:

```
IF (A) THEN
    // Логика 1
    RETURN;
END_IF

IF (B) THEN

    // Логика 2
END_IF
// Логика 3
```

А конструкцию, где дополнительная вложенность условий находится в основном условии IF:

```
IF (A) THEN
    IF (B) THEN
        // Логика 1
    END_IF
    // Логика 2
ELSE
    // Логика 3
END_IF
```

можно преобразовать, вывернув логику основного условия:

```
IF (NOT A) THEN
    // Логика 3
    RETURN;
END_IF

IF (B) THEN
    // Логика 1
END_IF
// Логика 2
```

Данный паттерн применим только в том случае, если вся бизнес-логика POU заключена в одном условие.

2.7.12.1.1.3.5 Пример минимизации вложенности

Например, код функционального блока таймера с памятью, который я писал для сублимационной установки. Не обязательно вникать в бизнес-логику этого примера, просто посмотрите на структуру конструкций IF.

```
IF pt > T#0s THEN
   IF RS OR (TimeWorked > PT) THEN
      TimeWorked := T#0s;
```

```
            TON1(IN := FALSE);
        ELSE
            IF NOT TON1.Q AND NOT IN THEN
                TimeWorked := TimeWorked + TON1.ET;
            END_IF

            TON1(IN := IN, PT := SEL(IN, T#0MS, PT - TimeWorked), Q => Q);

            TW := TimeWorked + TON1.ET;
            TP := REAL_TO_WORD(TIME_TO_REAL(TW) * 100.0 / TIME_TO_REAL(PT));
        END_IF
    ELSE
        Q := TRUE;
    END_IF
```

Здесь вложенность на 3 уровня - один IF включает в себя другой. Но мы сможем от нее полностью избавиться.

Так как весь код таймера заключен в одно условие, мы можем убрать первую вложенность. Вложенные условия находятся в основном условии IF, а значит, нам нужно вывернуть логику:

```
IF pt = T#0S THEN
    Q := TRUE;
    RETURN;
END_IF

IF RS OR (TimeWorked > PT) THEN
    TimeWorked := T#0S;
    TON1(IN := FALSE);
ELSE
    IF NOT TON1.Q AND NOT IN THEN
        TimeWorked := TimeWorked + TON1.ET;
    END_IF

    TON1(IN := IN, PT := SEL(IN, T#0MS, PT - TimeWorked), Q => Q);

    TW := TimeWorked + TON1.ET;
    TP := REAL_TO_WORD(TIME_TO_REAL(TW) * 100.0 / TIME_TO_REAL(PT));
END_IF
```

Посмотрите, как радикально изменился код программы, не изменив логики. Представьте, если бы вложенность была еще больше или было больше самих условий. Подобный паттерн часто помогает превратить почти нечитаемый код в прекрасный предмет искусства.

2.7.12.2 CASE

CASE - это конструкция для группировки нескольких условий, объединенных

зависимостью от значения одной переменной.

```
CASE <Var1> OF
    <value1>:<instruction1>
    <value2>:<instruction2>
    <value3, value4, value5>:<instruction3>
    <value6..value10>:<instruction4>
{ELSE
    <ELSE-instruction>}
END_CASE;
```

Переменная условия <Var1> должна быть из группы типов ANY_NUM или перечислением. То, что в фигурных скобках, {} не обязательно. Условие может быть одно <value1>. Условий может быть несколько <value1, value2>, перечисленных через запятую. Условие может определять промежуток от ... до <value1. . value2>.

Если условие IF объединено по значению одной и той же переменной, как например:

```
IF Svetofor = 1 THEN
    sColor := "Green";
ELSIF Svetofor = 2 THEN
    sColor := "Yellow";
ELSIF Svetofor = 3 THEN
    sColor := "Red";
ELSE
    sColor := "White";
END_IF
```

подобную конструкцию можно заменить на CASE

```
CASE Svetofor OF
    1: sColor := "Green";
    2: sColor := "Yellow";
    3: sColor := "Red";

ELSE
     sColor := "White";
END_CASE
```

В этом случае читаемость кода будет лучше.

Вместо числовых индексов можно использовать перечисления. Например, если у вас есть свой тип перечисление.

```
TYPE
    enumSV : (green := 1, orange := 2, red := 3);
END_TYPE
```

то CASE может выглядеть более интуитивно:

```
VAR
    Svetofor : enumSV;
END_VAR

CASE Svetofor OF
    green:  sColor := "Green";
    orange: sColor := "Yellow";
    red:    sColor := "Red";
ELSE
    sColor := "White";
END_CASE
```

Такой блок читается еще проще.

Вот другой простой пример использования CASE, преобразующий номер ошибки в сообщение об ее уровне.

```
VAR
    iErrorCode : INT;
    sMsg : STRING;
END_VAR

CASE iErrorCode OF
(* С сотой по двухсотую это - предупреждения *)
100..200:
    sMsg := 'Предупреждение';

(* Эти конкретные три номера - ошибки *)

340, 350, 370:
    sMsg := 'Ошибка';

(* Все коды от -100 до -1 критические ошибки*)
neg100..neg1:
    sMsg := 'Критическая ошибка';
ELSE
    sMsg := 'Ошибок Нет';
END_CASE;
```

2.7.12.2.1 Последовательности

Одной из самых сильных сторон CASE является возможность использовать его для создания пошаговых операций, последовательностей или для фиксирующего вызова ФБ.

2.7.12.3 Циклы

Циклы - это уникальное преимущество ST, которое отсутствует в других графических языках. Именно циклы обеспечивают гибкость работы в текстовом языке.

В этой теме на примерах вы увидите элементы, с которыми мы еще не познакомились - массивы, структуры. Было очень не просто решить какую тему поставить первой, переменные или циклы. В теме о переменных, в примерах есть циклы и наоборот. Я решил, что важнее сначала понять циклы, а если примеры тут будут не совсем понятны, пропустите их, они станут более понятными, когда мы закончим изучать переменные.

Цикл FOR, как правило, используется для прохождения по переменным типа ARRAY. Его также можно применять в алгоритмах, где присутствует пошаговый счет.

Отличие цикла FOR в том, что максимальное количество итераций, которые он исполнит, заранее известно.

```
FOR <выражение> TO <int> BY <int> DO
  ...
END_FOR;
```

Инструкция BY <int> не является обязательной и применяется только в том случае, если увеличение индекса должно быть больше чем 1.

2.7.12.4 Массивы

Допустим, имеется несколько температурных датчиков, каждый из которых регулирует свой выход нагревателя.

Рассмотрим пример, где имеется 3 массива: один с уставками aPV, другой - с показателями датчиков aTSensors, последний массив aOuts привязан к выходам КТСИ.

Включаем выход, если температура упала ниже уставки.

```
VAR
    aPV:       ARRAY[1..5] OF REAL := [
        34.5, 32.0, 28.4, 27.0, 35.8
    ];
    aTSensors: ARRAY [1..5] OF REAL;
    aOuts:     ARRAY[1..5] OF BOOL;
    iCount:    INT := 0;
END_VAR

FOR iCount := 1 TO 5 DO
    aOuts[iCount] := (aTSensors[iCount] < aPV[iCount]);
END_FOR
```

2.7.12.4.1 Пошаговые вычисления

Автору этой книги, к сожалению, не известны случаи применения данной технологии в реальности. Тем не менее, исходя из того, что подобные примеры представлены во всех инструкциях и мануалах, здесь он также публикуется. Возможно, у

читателей возникнут идеи того, где и как это можно использовать.

```
VAR
    count, sum, I : INT;
END_VAR

count := 1;
sum := 0;

FOR I := 1 TO 50 BY 2 DO
    sum   := ADD(sum, count);
    count := ADD(count, 1);
END_FOR;
```

В данном примере I счетчик цикла. Цикл будет работать 24 раза до 49, так как инкрементация будет по 2 или через один. 1, 3, 5, 7, ... 47, 49.

Основная конструкция цикла WHILE.

```
WHILE <выражение булево> DO
    <инструкции>
END_WHILE;
```

Цикл WHILE исполняет инструкции внутри цикла, пока условие между WHILE и DO будет равно TRUE. Как только оно изменится на FALSE, работа цикла прекратится и программа перейдет к исполнению строк программы или инструкциям, написанным после цикла.

Отличие цикла WHILE от FOR в том, что итерации могут исполняться непредсказуемое число раз, до тех пор, пока условие в выражении между WHILE и DO будет равно TRUE.

Например, мы хотим найти и удалить из строки все символы пробела. Мы заранее не знаем сколько их будет, поэтому мы будем искать символ, удалять его, и потом искать опять, чтобы убедиться что больше символов пробела не осталось в строке.

Весь пример может быть пока не совсем понятен, но главное тут принцип, что мы продолжаем искать пока не удалим все символы, не важно сколько их, и не зная заранее их количества.

```
VAR
    (* Строка для обработки *)
    sStr: STRING[20] := 'this is test';
    (* Позиция найденного символа *)
    iPos: INT;
END_VAR

(* Ищем символ пробела в строке. Если найдем то iPos будет > 0 *)
iPos := FIND(sStr, ' ');
WHILE iPos > 0 DO
    (* Удаляем символ пробела из строки, и сохраняем в туже переменную *)
    sStr := DELETE(sStr, 1, iPos);

    (* Снова ищем символ пробела *)
    iPos := FIND(sStr, ' ');
END_WHILE;
```

2.7.12.5 Массивы

Можно использовать WHILE и для того, чтобы проходить по массиву. Когда это может быть полезно? Например, это производится до тех пор, пока не встретится определенное значение.

Например, прохождение по массиву производится до тех пор, пока не встретится значение температуры в любом элементе массива, превышающее уставку.

```
VAR
    axTemps:  ARRAY[1..10] OF REAL;
    rTempMax: REAL := 32.5;
    bCount:   BYTE;
END_VAR

bCount := 1;
WHILE axTemps[bCount] < rTempMax AND bCount <= 10 DO
    heat(bCount); (* Какой-то контроль нагрева *)
    bCount := bCount + 1;
END_WHILE;
```

Основная конструкция цикла REPEAT.

```
REPEAT
    <инструкции>
UNTIL <выражение булево>
END_REPEAT;
```

Цикл REPEAT работает, пока условие UNTIL равно FALSE.

Цикл REPEAT применяется для тех же задач, что и WHILE, с той лишь разницей, что в REPEAT инструкции исполнятся хотя бы один раз. Как мы видим, выражение условия

UNTIL здесь стоит в конце, поэтому мы сначала исполняем инструкции в REPEAT, а потом проверяем условие. Значит, инструкции внутри REPEAT будут исполнены хотя бы один первый раз, даже если условие UNTIL вернет TRUE. Иногда это бывает необходимо.

Оператор CONTINUE, если поддерживается, то всеми тремя циклами: FOR, WHILE и REPEAT.

Этот оператор пропускает одну итерацию и переходит к следующей. Предположим, есть структура нагревателей.

```
TYPE Heater:
    STRUCT
        enable: BOOL; (* Вкл/Откл нагреватель *)
        SV: REAL;     (* Set Value - Уставка *)
        PV: REAL;     (* Processed Value - Текущее значение *)
        Delta: REAL;  (* Гистерезис *)
    END_STRUCT
END_TYPE
```

Управляем только теми нагревателями, которые включены.

```
VAR
    sHeaters: ARRAY[1..10] OF Heater;
    bCount: BYTE;
END_VAR

FOR bCount := 1 TO 10 DO
    IF sHeaters[bCount].enable = FALSE THEN
        CONTINUE;
    END_IF;

    PID(sHeaters[bCount]); (* Какой-то контроль нагрева *)
END_FOR
```

В данном примере функция регулирования PID(sHeaters[bCount]) исполнится только если свойство структуры enable равно TRUE. А если нет, то CONTINUE прервет текущую итерацию цикла и перейдет к следующей.

Оператор EXIT, если поддерживается, то всеми тремя циклами: FOR, WHILE и REPEAT

Обычно оператор EXIT в других языках останавливает исполнение всей программы. В ST оператор EXIT прерывает работу цикла.

Предположим, имеется такая же структура, что и в примере с CONTINUE, но теперь стоит задача прервать цикл FOR, как только любой из нагревателей нагрелся до 50 градусов.

```
FOR bCount := 1 TO 10 DO
    IF sHeaters[bCount].PV > 50.0 THEN
        EXIT;
    END_IF;

    PID(sHeaters[bCount]);
END_FOR
```

В этом примере все последующие итерации FOR прекратятся, когда будет вызвано EXIT.

Примечание - При использовании циклов нужно быть очень аккуратным.

2.7.12.6 Открытый цикл

Все циклы работают синхронно, или другой термин - блокирующий цикл. Это значит, что программа не перейдет к исполнению следующей строки, пока не закончит исполнение предыдущей.

При этом, основная программа КТСИ тоже работает в цикле. После исполнения всей программы, КТСИ начинает исполнение программы заново.

Получается, что все итерации любого цикла FOR, WHILE и REPEAT должны быть завершены в одном цикле КТСИ прежде, чем КТСИ запустит новый цикл основной программы. Другими словами, КТСИ не завершит цикл основной программы, пока не завершиться цикл FOR, WHILE или REPEAT, который вы написали.

Например, расположим в программе подобный код:

```
WHILE TRUE DO
    ;
END_WHILE;
```

Это приведет к зависанию КТСИ. Так как подобный цикл никогда не прервется, то и КТСИ не перейдет к исполнению следующий строки за циклом, а значит, цикл основной программы никогда не будет завершен.

Каждый цикл должен быть закрыт, иными словами, в нем должен быть код, обеспечивающий выход из работы цикла.

2.7.12.7 Переменные

Переменные представляют собой средства идентификации объектов данных, содержание которых может изменяться. Например, это данные, связанные с входами, выходами или памятью КТСИ.

2.7.12.7.1 Объявления переменных

Прежде, чем рассмотрим типы переменных, разберемся, как переменные объявляются, чтобы стали понятны следующие примеры:

> **Заметка**
>
> В разных программах IDE переменные называются по-разному. Например, в CoDeSys это **переменные** (variables), в SIMATIC STEP 7 это **теги** (tags), а в Studio 5000 Logix Designer для Allen Bradley это **символы** (symbols).

Объявление делается в верхней части программы, в области введения переменных, между ключевыми словами VAR и END_VAR.

```
PROGRAM demo
    VAR
        a : BOOL;
    END_VAR
    // Программа тут
END_PROGRAM;
```

Переменной можно присвоить значение по умолчанию. Оно будет присвоено при инициализации переменной один раз - на старте программы.

```
a : BOOL := TRUE;
```

Можно при помощи ключа AT явно привязывать переменные к точным адресам памяти КТСИ.

```
a AT %IX0.0.1 : BOOL; (* Переменная a с привязкой к адресу *)
```

Объявление можно разделить на 4 части.

```
[имя] AT [адрес] : [тип] := [значение]; (* [комментарий] *)
```

1 [имя] - Имя переменной;

2 AT - Ключ привязки переменной к адресу памяти или I/O;

3 [адрес] - Адрес ячейки памяти, где хранить переменную;

4 : - Оператор объявления типа;

5 [тип] - Тип переменной;

6 := Оператор присвоения значения;

7 ; Определение окончания инструкции;

8 (* *) Комментарий.

2.7.12.8 Присвоение имени

Имя переменной может содержать только символы латинского алфавита любого регистра, символ _ и цифры 0-9. Любой другой символ приведет к ошибке.

Есть и правила, которые не обязательны к исполнению, но считаются общепринятыми, хорошей практикой.

Принято присваивать префикс при объявлении переменных, сокращение от типа самой переменной:

- BOOL - x - например: xStart
- BYTE - b - например: bStep
- WORD - w - например: wTime
- INT - i - например: iVal
- USINT - usi - например: usiChannel
- ARRAY - a - где это префикс переменной элемента массива, например: awTmp массив из WORD

Вот ещё несколько примеров объявления переменных:

```
axList: ARRAY[1..5] OF BOOL;
xStart_1: BOOL;
xStart_Motor1: BOOL;
wMotor_Speed: WORD;
```

Для переменных с физических входов и выходов, я присваиваю еще один префикс DI_, DO_, AI_ и AO_. Например, если у нас есть аналоговый вход типа WORD, измеряющий давление, то переменная будет такой: AI_wPressure. Здесь в объявлении виден и префикс AI_, и w. Во-первых, таким образом можно быстро получить список всех переменных в подсказке, просто введя первый префикс, во-вторых, видно не только, что это вход или выход, но и тип этого входа или выхода - дискретный или аналоговый. Ведь аналоговый вход или выход может быть REAL, а дискретный вход или выход может быть WORD.

2.7.12.9 Области видимости

2.7.12.9.1 Локальные переменные

Чаще других в программах объявляются локальные внутренние переменные. Это все, что размещено между VAR и END_VAR. Эти переменные доступны для чтения и записи только внутри той POU, в которой они были определены.

2.7.12.9.2 Глобальные переменные

Это тип переменных, которые видны во всех POU. Как правило, такие переменные размещаются между ключами VAR_GLOBAL и END_VAR.

Глобальные переменные объявляются в конфигурации между CONFIGURATION и END_CONFIGURATION. Для видимости глобальных переменных в POU нужно объявить те же переменные внутри POU в VAR_EXTERNAL, но в некоторых IDE как например CoDeSys этого не требуется.

2.7.12.9.3 Входные и выходные

Внешний объект - это POU, в котором вызывается тот POU, о котором идет речь. Например, если говорится о функции SQRT, для нее внешним объектом будет POU, в котором она вызывается.

По значению. Передача переменной по значению значит, что в POU передается только значение переменной и не существует связи между внешним объектом и вызываемым POU. После присвоения переменной ее можно менять внутри POU, но это не изменит переменную внешнего объекта, так как было передано просто значение.

По ссылке. Передача переменной по ссылке значит, что передается не значение переменной из внешнего объекта, а ссылка на эту переменную. То есть, если внутри POU поменяли значение подобной переменной, ее значение во внешнем объекте тоже изменится, так как было передано не само значение, а ссылка на значение переменной внешнего объекта.

Входные и выходные переменные позволяют определить переменные, которым можно присвоить значения, и таким образом передать их в POU из внешнего объекта в момент вызова POU, и переменные, которые вернут значения после вызова POU и передадут их значения во внешний объект.

```
VAR
    a : BOOL;
END_VAR
VAR_INPUT
    IN : BOOL;
    PT : TIME;
END_VAR
VAR_OUTPUT
    Q : BOOL;
END_VAR
```

Входные переменные VAR_INPUT передаются из внешнего объекта по значению, они являются локальными для POU, в котором определены. Их нельзя изменить внутри POU. То есть, присвоить им иное значение возможно, но переменная внешнего объекта, из которой это значение было передано, не изменится.

Выходные переменные VAR_OUTPUT можно менять в POU, их значения передаются далее во внешний объект.

Проходные переменные VAR_IN_OUT. Им можно присвоить значения из внешнего

объекта, их можно изменять внутри POU, и их значения передадутся далее во внешний объект. Можно смотреть на них, как на входные переменные, передающиеся по ссылке.

Эти типы определения переменных обычно используется при написании функций и ФБ при создании программы, если предполагается ее вызов из другой программы.

Например, функция, которая считает площадь круга:

```
FUNCTION rCircleP : REAL
    VAR_INPUT
        rRadius  : REAL;
    END_VAR
    VAR CONSTANT
        pi : REAL := 3.14159265359;
    END_VAR

    rCircleP := pi * (rRadius ** 2);
END_FUNCTION
```

2.7.12.9.4 Внешние переменные

Внешние переменные определяются при помощь ключа VAR_EXTERNAL. При определении внешних переменных им нельзя назначать значения по умолчанию. Это связано с тем, что этот ключ, скорее, не определяет новые переменные, а ссылается на них и делает их доступными, как локальные, для текущего POU.

Большинство IDE автоматически делают глобальными все переменные, определенные в VAR_GLOBAL. Но если писать программу на ST без помощи IDE, то для того, чтобы сделать глобальную переменную видимой в POU, нужно ее определить, как внешнюю. Например, имеется глобальная переменная:

```
CONFIGURATION
    VAR_GLOBAL
        iNum : INT := 10;
    END_VAR
END_CONFIGURATION
```

Тогда в программе нужно было бы обозначить ее как внешнюю, прежде чем мы получим к ней доступ.

```
PROGRAM
    VAR_EXTERNAL
        iNum : INT;
    END_VAR
    VAR
        iNum2 : INT := 20;
        iSum : INT;
    END_VAR
    iSum := iNum + INum2;
END_PROGRAM
```

2.7.12.9.5 Открытые переменные

Вряд ли вы скоро сможете воспользоваться этой функцией, но сейчас есть возможность открывать доступ к коммуникационным серверам, описанным в 5-ом разделе стандарта (МЭК 61131-5), к переменным, привязанным к входам, выходам или внутренним переменным. Для этого используется ключ VAR_ACCESS. Но в тех IDE, где имеется поддержка МЭК 61131-5, подобное можно делать в графическом интерфейсе, и нет необходимости писать подобный код. Но понимать, как это работает изнутри, все равно полезно. Доступ к переменной должен определяться полным путем к переменной, начиная с имени POU, и разделяя путь . точкой:

```
VAR_ACCESS
    SHARE1 : RESOURCE1.%IX1.1 : BOOL READ_ONLY;
    SHARE2 : RESOURCE1.PROGRAM1.iCount : UINT READ_WRITE;
END_VAR
```

Не будем уделять этому много внимания, так как в настоящее время данная функция не имеет практического применения, хотя это интересно и полезно знать для общего кругозора.

2.7.12.9.6 Список всех областей видимости переменных

Таблица 24

Ключ	Описание
VAR	Локальная для POU
VAR_TEMP	Временное хранилище для переменных которые не сохраняют свои значения между вызовами POU, в котором объявлены
VAR_INPUT	Назначается извне, не может быть изменена внутри POU
VAR_OUTPUT	Назначается из POU во внешний объект
VAR_IN_OUT	Назначается извне, может быть изменена в POU
VAR_EXTERNAL	Назначается из VAR_GLOBAL и может меняться в POU
VAR_GLOBAL	Глобальная переменная, видимая в любом POU
VAR_ACCESS	Определение уровня доступа к переменной
VAR_CONFIG	Инициализация, специфичная экземпляру

2.7.12.10 Дополнительные ключи

Ключ CONSTANT определяет группу переменных, как переменные защищенные от изменений, константы. Такую переменную нельзя переназначить. Пример с Pi подходит как нельзя лучше. Это неизменяемое число.

```
VAR CONSTANT
    PI : REAL := 3.14;
    c_MAX_LENGTH : INT := 100;

END_VAR
```

Желательно всегда использовать константы там, где это имеет смысл, чтобы защититься от случайных изменений значения этой переменной. Принято именовать константы большими буквами. Часто ставят префикс c_ к константе. Это улучшает читабельность кода. Сразу видно, что это константа, а не перечисление, например. Легко найти нужную константу при автозаполнении, просто введя c_ - выпадет список со всеми константами.

Ключ RETAIN позволяет сделать переменную энергонезависимой. Обычно NON_RETAIN не используется, так как по умолчанию все переменные создаются энергозависимыми. Если необходимо, чтобы какая-то переменная сохранила свое значение после сбоя программы или выключения КТСИ, можно использовать ключ RETAIN:

```
VAR RETAIN
    xError:    BOOL := TRUE;
    uiCounter: UINT := 0;
END_VAR
```

Здесь нужно быть осторожным. Как правило, ресурс подобной памяти в КТСИ ограничен количеством циклов перезаписи. Не желательно использовать переменные, которые часто меняют свое значение.

AT позволяет привязать переменную к точному адресу памяти контроллера. Обычно это делается для привязки переменных к адресам входов и выходов КТСИ. AT можно использовать, чтобы явно задавать адрес хранения переменной в памяти КТСИ.

```
VAR
    rTemperature AT %ID0.0.0: REAL;
END_VAR
```

Адресация начинается со знака % и далее идут один или два буквенных символа.

Первый символ ссылается на адресное положение переменной. Этот символ всегда должен присутствовать.

Таблица 25

Сим.	Адрес
I	Входы КТСИ
Q	Выходы КТСИ
M	Внутренняя память

Второй символ обозначает размер переменной.

Таблица 26

Сим	Адрес
X или без	Один бит BOOL
B	Один байт (8 bits) размер BYTE
W	Слово (16 bits) размер WORD
D	Двойное слово (32 bits) DWORD
L	Большое (quad) слово (64bits) размер LWORD

Адресное пространство отображается числами. Это может быть и 0.0 - 9.9 и 0.0.0.0 - 9.9.9.9 или просто 0 - 9999 без точек. Количество разрядов (чисел разделенных точкой) зависит от разных производителей и объема отведенной производителем области памяти, к которой мы обращаемся. Разряды могут быть не только в диапазоне 0-9, но и в диапазоне 0-255.

Ответ на вопрос, в каких рамках можно обращаться к адресам памяти, нужно искать в руководствах производителей контроллеров, которые вы программируете. Определяются эти разряды в конфигурации CONFIGURATION, в ключе VAR_CONFIG ... END_VAR.

2.7.12.11 Преобразование типов

Практически все IDE предоставляют способ преобразования одного типа переменной в другой что часто необходимо. Практически всегда имя функции преобразования строится на одном принципе.

[FROM]_TO_[TO]()

Где FROM это тип данных который мы опускаем в функцию и TO это тип данных который из нее возвращается. Таким образом можно легко узнать имя любой функции. Например нам нужно число INT преобразовать в REAL. Функция преобразования будет INT_TO_REAL().

В CoDeSys 3.5 можно опустить FROM и просто использовать конструкцию TO_[TO] () в нашем примере это будет TO_REAL() а тип входного значения будет определён

автоматически. А если быть точным, для CoDeSys определение типа входной переменной вообще не обязательно как мы поймем изучая указатели.

2.7.12.12 Типы переменных

Типы данных делятся на 3 основные категории.

> **Внимание**
>
> Правильный подбор типов переменных поможет использовать память ПЛК наиболее эффективно и писать более производительные, менее ресурсоемкие программы.

2.7.12.12.1 Одноэлементные переменные

Когда создается переменная с именем, то в памяти контроллера под нее отводится место и это имя привязывается к отведенной для нее ячейки памяти (адресу). Это делается автоматически. Как сообщалось ранее, при объявлении переменной можно указать к какой области памяти ее привязать при помощи ключа AT.

```
VAR
    rTemperature AT %ID0.0.0: REAL;
END_VAR
```

Если ключа AT не использовать, то переменной будет присвоен адрес %* автоматически или лучше сказать динамически. Поэтому часто, переменные в разных IDE называются - символьный адрес так как по сути имя переменной в программе это ссылка на адрес ячейки памяти с использованием символов алфавита а не адрес памяти напрямую как например %MW0.

Можно не только привязывать адреса к именам переменных, но и напрямую использовать адреса в коде программы.

```
IF %IX0.1.2 THEN
    %QX0.0.1 := TRUE;
END_IF
```

Одноэлементные переменные напрямую обращаются к зонам памяти.

Этот пример говорит о том, что если на дискретном входе КТСИ 0.1.2 будет сигнал, нужно включить выход КТСИ 0.0.1.

Обращаться можно не только к входам и выходам КТСИ, и не только к булевым значениям. Можно обращаться и к внутренней памяти. Например, обратиться к ячейке памяти размером WORD можно через символ %MW0.

```
%MW0 := 4585;
```

Важно

Существует опасность перезаписи значений при использовании одноэлементных типов.

Суть потенциальной проблемы в том, что размер адресации X, B, W, D, L не отводится в разных ячейках памяти. То есть, %MW0, %MB0, %MX0 - все начинаются с одного и того же адреса памяти. По сути, %MB0 и %MB1 - это первый во второй байт %MW0, а сам %MW0 - первый из двух регистров %MD0.

Если, например, сделать так:

```
%MW0 := 15; (* побитно будет 0000 0000 0000 1111 *)
%MB1 := 15; (* побитно будет 0000 1111 *)
```

то в результате %MW0 будет равно 855 в байтах 0000 1111 0000 1111, так как ячейке памяти %MB1 было присвоено 15, а это второй байт %MW0.

2.7.12.12.2 Группы элементарных типов данных

- General (общие);
- Integers (целочисленные);
- Floating point (числа с плавающей точкой);
- Strings (строки);
- Bit strings (битовые строки);
- Time (время);

В каждой группе элементарных типов есть несколько типов данных, определенных

в МЭК 61131-3:

2.7.12.12.2.1 Общие типы (general types)

Каждый из простых типов, описанных далее в этом разделе, относится так же и к общему типу ANY. Вот иерархия общих типов и элементарных типов, к которым они относятся.

```
ANY
    ANY_DERIVED
    ANY_ELEMENTARY
        ANY_MAGNITUDE
            ANY_NUM
                ANY_REAL
                    LREAL
                    REAL
                ANY_INT
                    LINT, DINT, INT, SINT
                    ULINT, UDINT, UINT, USINT
            ANY_DURATION
                TIME
                LTIME
        ANY_BIT
            LWORD, DWORD, WORD, BYTE, BOOL
        ANY_CHARS
            ANY_STRING
                STRING
                WSTRING
            ANY_CHAR
                CHAR
                WCHAR
        ANY_DATE
            DATE_AND_TIME
            DATE, TIME_OF_DAY
```

Стандарт это не оговаривает, но некоторые IDE позволяют в некоторых случаях, применять к переменным некоторые общие типы. Тогда, в теории, переменная сможет принимать значения из любой переменной дочернего типа. Например, объявив переменную типа ANY_REAL, можно назначать ей переменные дочерних типов, таких как REAL так и LREAL.

```
VAR
    iCount : ANY_NUM;
    a: INT;
    b: USINT;
END_VAR

iCount := a;
iCount := b;
```

Такой подход хорош при написании функций или функциональных блоков, где точно не известно, какой тип данных поступит в функцию, поэтому общий тип подходит как нельзя лучше. К сожалению, в связи с непростой логикой и определенными сложностями в реализации, не все вендоры внедрили и поддерживают данную возможность.

2.7.12.12.2.2 Integer: (целочисленный)

Таблица 27

Тип данных МЭК	Формат	Значение	Кол. байт
SINT	Short Integer	-128 ... 127	1
USINT	Unsigned Short Integer	0 ... 255	1
INT	Integer	-32768 ... 32767	2
UINT	Unsigned Integer	0 ... 2^16-1	2
DINT	Double Integer	-2^31 ... 2^31-1	4
UDINT	Unsigned Double Integer	0 ... 2^32-1	4
LINT	Long Integer	-2^63 ... 2^63-1	8
ULINT	Unsigned Long Integer	0 ... 2^64-1	8

Обратим внимание на то, что целочисленные типы данных с приставкой U что ссылается на слово Unsigned (без знака) не принимают отрицательных значений. Всегда стоит использовать тип с приставкой U, если известно, что значение не будет отрицательным.

Пример определения переменных целочисленного типа с назначением в коде:

```
VAR
    usiStage : USINT := 0;
    diMotorStepsCounter : DINT := 1_254;
END_VAR

usiStage := 14;
```

Назначение можно делать как в десятичном формате, так и в других форматах (описано позже). Обратите внимание на символ _. Его можно использовать в числах, а так же в типах время. Обычно его используют как разделитель для удобства чтения длинных чисел. Во время компиляции программы, этот символ игнорируется.

К переменным числового типа можно применить все арифметические операторы, такие как +, -, *, /, MOD и т.д., а если преобразовать в Floating point, то и все числовые операторы, такие как ABS, SQRT, COS, SIN и т.д. Но, к сожалению, в стандарт МЭК 61131-3 не входят, как обязательные, некоторые полезные математические функции. Мы их рассмотрим, изучая числа с плавающей точкой.

EVEN вычисляет, является ли число четным или нет. Если четное, то вернет TRUE, а если нет - вернет FALSE.

```
FUNCTION EVEN : BOOL
    VAR_INPUT
        in : DINT;
    END_VAR
    EVEN := NOT in.0;
END_FUNCTION
```

2.7.12.12.2.3 Floating point: (числа с плавающей точкой)

Таблица 28

Тип данных МЭК	Формат	Значение
REAL	Real Numbers	±10^±38
LREAL	Long Real Numbers	±10^±308

Назначения можно присваивать следующим образом:

```
VAR
    rNum : REAL;
END_VAR

rNum := 4.5;
rNum := 1.64e+009;
```

Стандарт не предусматривает большого количества математических и арифметических функций, которые в других языках являются базовыми. Рассмотрим несколько примеров полезных функций для работы с данными типа ANY_REAL.

FLOOR округляет значение и возвращает ближайшее целое значение, которое меньше или равно X.

floor(3.14) = 3 floor(-3.14) = -4

```
FUNCTION FLOOR : DINT
    VAR_INPUT
        X : REAL;
    END_VAR

    FLOOR := REAL_TO_DINT(X);
    IF DINT_TO_REAL(FLOOR) > X THEN
        FLOOR := FLOOR - 1;
    END_IF;
END_FUNCTION
```

CEIL округляет значение и возвращает ближайшее целое значение, которое больше или равно X.

ceil(3.14) = 4 ceil(-3.14) = -3

```
FUNCTION CEIL : DINT
    VAR_INPUT
        X : REAL;
    END_VAR
    CEIL := REAL_TO_DINT(X);
    IF DINT_TO_REAL(CEIL) < X THEN
        CEIL := CEIL + 1;
    END_IF;
END_FUNCTION
```

EXP10 вычисляет с основанием степени 10.

exp10(2) = 100 exp10(3) = 1000

```
FUNCTION EXP10 : REAL
    VAR_INPUT
        X : REAL;
    END_VAR

    EXP10 := EXP(X * 2.30258509299405);
END_FUNCTION
```

EXPN возводит в степень X^N. Хотя в стандарте есть оператор EXPT, по заявлению OSCAT, данный алгоритм работает в 30 раз быстрее в CoDeSys.

```
FUNCTION EXPN : REAL
    VAR_INPUT
        X : REAL;
        N : INT;
    END_VAR
    VAR
    sign: BOOL;
END_VAR
```

```
    sign := N.15;
    N := ABS(N);
    IF N.0 THEN EXPN := X; ELSE EXPN := 1.0; END_IF;
    N := SHR(N, 1);
    WHILE N > 0 DO
        X := X * X;
        IF N.0 THEN EXPN := EXPN * X; END_IF;
        N := SHR(N, 1);
    END_WHILE;
    IF sign THEN EXPN := 1.0 / EXPN; END_IF;
END_FUNCTION
```

ROUND округляет число с плавающей точкой до указанного количества знаков после запятой.

round(12.23456789, 2) = 12.23

```
FUNCTION ROUND : REAL
    VAR_INPUT
        r: REAL;  (* Число для обработки *)
        p: USINT; (* Точность после запятой *)
    END_VAR

    ROUND := DWORD_TO_REAL(REAL_TO_DWORD(r * EXP10(p))) / EXP10(p)
END_FUNCTION
```

CMP сравнивает 2 входных числовых переменных, совпадают ли их первые цифры. Количество сравниваемых цифр передается в параметре N.

cmp(3.141516, 3.141517, 6) вернет TRUE.

```
FUNCTION CMP : BOOL
    VAR_INPUT
        X, Y : REAL;
        N : INT;
    END_VAR
    VAR
        tmp : REAL;
    END_VAR
    tmp := ABS(x);
    IF tmp > 0.0 THEN
        tmp := EXP10(INT_TO_REAL(FLOOR(LOG(tmp))-N+1));
    ELSE
        tmp := EXP10(tmp);
    END_IF;
    CMP := ABS(X - Y) < tmp;
END_FUNCTION
```

MODR вычисляет остаток от деления для чисел с плавающей запятой.

modr(5.5, 2.5) = 0.5

```
FUNCTION MODR : REAL
    VAR_INPUT
        IN : REAL;
        DIVI : REAL;
    END_VAR

    IF divi = 0.0 THEN
        MODR := 0.0;
    ELSE
        MODR := in - DINT_TO_REAL(FLOOR(in / divi)) * divi;
    END_IF;
END_FUNCTION
```

DEG конвертирует радианы в градусы. Эта функция нужна для работы со временем и определения времени восхода и заката солнца.

```
FUNCTION DEG : REAL
    VAR_INPUT
        rad : REAL;
    END_VAR
    DEG := MODR(57.29577951308232 * RAD, 360.0);
END_FUNCTION
```

D_TRUNC усекает число с плавающей запятой в целое число DINT. 1.5 станет 1, а -1.5 станет -1. Данная функция необходима, потому что REAL_TO_DINT на разных системах может вернуть разный результат.

```
FUNCTION D_TRUNC : DINT
    VAR_INPUT
        X : REAL;
    END_VAR
    D_TRUNC := REAL_TO_DINT(X);
    IF X > 0.0 THEN
        IF DINT_TO_REAL(D_TRUNC) > X THEN
            D_TRUNC := D_TRUNC - 1;
        END_IF;
    ELSE
        IF DINT_TO_REAL(D_TRUNC) < X THEN
            D_TRUNC := D_TRUNC + 1;
        END_IF;
    END_IF;
END_FUNCTION
```

FRACT возвращает дробную часть числа с плавающей точкой.

fract(3.14) = 0.14

```
FUNCTION FRACT : REAL
    VAR_INPUT
        x : REAL;
    END_VAR

    IF ABS(x) < 2.0E9 THEN
        FRACT := x - DINT_TO_REAL(D_TRUNC(x));
    ELSE
        FRACT := 0.0;
    END_IF;
END_FUNCTION
```

RDM вычисляет псевдо-случайное число. Чтобы сгенерировать число, функция считывает таймер КТСИ и создает число с плавающей точкой от 0 до 1. Чтобы использовать эту функцию больше, чем один раз, в одном цикле КТСИ, нужно ее вызывать с разным значением входного параметра last. В него проще внести прошлое число, чтобы исключить повторения.

```
FUNCTION RDM : REAL
    VAR_INPUT
        last : REAL;
    END_VAR
    VAR
        tn : DWORD;
        tc : INT;
    END_VAR
    tn := TIME_TO_DWORD(TIME());
    tc := BIT_COUNT(tn);
    tn.31 := tn.2;
    tn.30 := tn.5;
    tn.29 := tn.4;
    tn.28 := tn.1;
    tn.27 := tn.0;
    tn.26 := tn.7;
    tn.25 := tn.6;
    tn.24 := tn.3;
    tn := ROL(tn, BIT_COUNT(tn)) OR 16#80000001;
    tn := tn MOD 71474513 + INT_TO_DWORD(tc + 77);
    RDM := FRACT(DWORD_TO_REAL(tn) / 10000000.0 *
        (math.E - LIMIT(0.0, last, 1.0))
    );
END_FUNCTION
```

В приведенном выше примере

math.E = 2.71828182845904523536028747135266249 - константа Эйлера.

Генерирование случайных чисел является достаточно частой задачей. Так как на

выходе у нас 0-1, то чтобы получить число в необходимом нам диапазоне, нужно сделать линейное масштабирование. Например, чтобы получить случайное число от 1 до 100.

```
rMyRandom := SCALE_R(RDM(0.5), 0, 1, 1, 100);
```

2.7.12.12.2.4 Strings: (строковые переменные)

Таблица 29

Тип данных МЭК	Формат	Значение
STRING	Символьная строка	'Моя строка'
WSTRING	Символьная строка двухбайтная	"Моя строка"
CHAR	Однобайтный символ	'A'
WCHAR	Двухбайтный символ	"A"

Примеры объявления строковых переменных.

```
VAR
    sMessage1 : STRING  := 'Motor Stopped!';
    sMessage2 : WSTRING := "Двигатель запущен!";
END_VAR
```

Одиночные кавычки, как правило, определяют строку нуля или более однобайтных символов символьной таблицей ISO/IEC 10646-1. В однобайтных строках комбинация из 3 символов - доллара $ и последующих за ним 2-х шестнадцатеричных символов, например '$AE', определяют код 8-битного символа. Могут использоваться любые HEX символы.

Двойные кавычки, как правило, определяют строку нуля или более двухбайтных символов символьной таблицей ISO/IEC 10646-1. В двубайтных строках комбинация из 5 символов - доллара $ и последующих за ним четырех шестнадцатеричных символов "$00C4", определяют код 16-битного символа.

Таблица 29 - Символьно-строковые литералы

№	Описание	Пример
Однобайтовые символы и строки символов с ' '		
1a	Пустая строка (длины ноль)	''
1b	Строка длины один или символ CHAR, содержащий единственный символ	'A'
1c	Строка длины один или символ CHAR, содержащий символ пробела	' '
1d	Строка длины один или символ CHAR, содержащий символ одиночной	'$' '

№	Описание	Пример
	кавычки	
1e	Строка длины один или символ CHAR, содержащий символ двойной кавычки	' " '
1f	Поддержка двухсимвольных комбинаций таблицы ниже	'RL'
1h	Строка длины пять символов $1.00	'$$1.00'
1g	Поддержка представления символа со знаком доллара '$' и двумя шестнадцатеричными цифрами	'$0A'
Двухбайтовые символы или символьные строки с " "		
2a	Пустая строка (длины ноль)	""
2Ь	Строка длины один или символ WCHAR, содержащий единственный символ	"A"
2c	Строка длины один или символ WCHAR, содержащий символ пробела	" "
2d	Строка длины один или символ WCHAR, содержащий символ одиночной кавычки	" ' "
2e	Строка длины один или символ WCHAR, содержащий символ двойной кавычки	"$""
2f	Поддержка двухсимвольных комбинаций таблицы ниже	"RL"
2h	Строка длины пять символов $1.00	"$$1.00"
2g	Поддержка представления символа с знаком доллара '$' и четырьмя шестнадцатеричными цифрами	"$00C4"

Таблица 29- Двухсимвольные комбинации в символьных строках

№	Описание	Комбинация
1	Знак доллара	$$
2	Единичная кавычка	$'
3	Двойная кавычка	$"
4	Перевод строки	$L или $l
5	Новая строка	$N или $n
6	Прогон (перевод) страницы	$P или $p
7	Возврат каретки	$R или $r
8	Табуляция	$T или $t

- Комбинация $' действительна только внутри строковых литералов с одиночными кавычками.

- Комбинация $" действительна только внутри строковых литералов с двойными кавычками.

2.7.12.13 Time: (время)

Время является одним из трех измерений, в которых мы существуем, поэтому работать со временем приходится очень часто. Ведь время - сама сущность бытия. Задач, использующих время, не просто много - не будет преувеличением сказать, что все до единой задачи в КТСИ решаются с использованием времени.

Одни задачи используют событийную структуру - от одного условия к другому, не применяя переменных типа "время", не делая их измерений или вычислений, или использования таймеров. Например:

```
VAR
    a, b: BOOL;
END_VAR

IF a THEN
    b := TRUE;
ELSE
    b := FALSE;
END_IF
```

Для начала рассмотрим типы данных, определяющих время, установленные стандартом МЭК 61131-3. Их всего 4 типа:

- TIME – интервал времени. Можно использовать префикс T для определения значения константой. Состоит из полей дней (d), часов (h), минут (m), секунд (s) и миллисекунд (ms). Порядок полей должен быть таким, в каком они перечислены. Менять его нельзя, но можно пропускать ненужные элементы.

```
VAR
    t1, t2, t3 : TIME;
    ton1: TON;
END_VAR
t1 := T#1h2m20s100ms;
t2 := T#20m100ms;
t2 := T#1h_20s;
ton1(IN := TRUE, PT := T#300s);
```

Значения могут превышать границы естественного диапазона, так как это не время дня, а просто интервал времени. Например, можно указать минут больше, чем 60.

```
T#120m (* Это 2 часа *)
```

Но это не относится к младшим полям или следующим, а только к первому полю. Так, например, следующее будет ошибкой. Если есть поле часа, то младшее поле минут не может превышать естественного номера 60.

```
T#1h120m (* Ошибка *)
```

Для улучшения восприятия или читабельности кода можно использовать символ _ для разделения полей. Как и в числовых типах, этот символ не учитывается, а используется для визуального разделения.

```
T#5h_30m_40s;
```

- TIME_OF_DAY – время дня. Можно использовать префикс TOD для определения значения константой. TOD не надо путать с T - они принципиально разные. Время в TOD ограничено диапазоном от 0 до 23:59:59.999, а в T оно не ограничено. В TOD мы не можем превышать допустимые пределы каждого поля. Если это секунды, то значение может быть только от 0 до 59. Если часы, то от 0 до 23.

TOD хранит точку во времени, а T хранит промежуток времени или время работы какого-то процесса.

TOD состоит из полей: час (HH), минуты (MM), секунды(SS), дробная часть секунд - миллисекунды (sss) Эти поля разделяются знаком :. Одни должны идти в строгом порядке как перечислены. Первыми будут всегда часы и т.д.

```
VAR
    tod1, tod2, tod3 : TIME_OF_DAY;
END_VAR
tod1 := TOD#15;           (* 15:00:00 *)
tod2 := TOD#15:30;        (* 15:30:00 *)
tod3 := TOD#16:25:45;     (* 16:25:45 *)
tod3 := TOD#16:25:45.123; (* 16:25:45.123 *)
```

- DATE – дата или календарное число. Можно определить при помощи символа D. Имеет структуру ГГГГ-ММ-ДД. Все числа имеют естественный диапазон.

```
VAR
    d1 : DATE;
END_VAR
d1 := D#2017-09-14; (* 14 Сентября 2017 *)
```

DATE_AND_TIME – дата или календарное число с временем. Можно определить при помощи символа DT. Это похоже на совмещение типа DATE и TIME_OF_DAY. Его формат может быть непривычен для нас, но он определен и принят во многих странах международным стандартом ISO 8601.

```
VAR
    dt1 : DATE_AND_TIME;
END_VAR

(* 11 Декабря 2017 в 19:25 когда я писал эту статью. *)
dt1 := DT#2017-12-11-19:25:00;
```

К сожалению, стандартом МЭК 61131-3 не предусмотрено, каким образом такие типы данных хранятся, и каждый производитель осуществляет их хранение по-своему.

Есть 2 основных способа того, как можно было бы хранить данные.

1 Структура - данные хранятся в виде переменной структуры, в которой отдельно хранится год, отдельно - месяц и т.д. Такой способ удобен для конечного разработчика, но неудобен в вычислениях.

2 Числовой - данные хранятся в виде числа, что является более распространенным способом.

Примечание – Все переменные типа «дата» и «время» хранятся в регистре памяти типа DWORD.

В языках программирования общепринято брать за дату отсчета 0 часов 1 января 1970 года.

Данные типа DATE_AND_TIME и DATE хранятся в виде количества секунд с этой даты.

Длительность TIME и TIME_OF_DAY имеют дискретность в 1 мс. Максимальная длительность, которую могут хранить переменные типа TIME, составляет примерно 1193 часа или почти 50 суток. Именно столько миллисекунд можно сохранить в 4-х байтах переменной размера DWORD.

Посмотрим, как можно получить текущую дату и время. Рассмотрим 3 разных способа:

1 RTC - (Real Time Clock) или часы реального времени. Это стандартный ФБ стандарта МЭК 61131-3. Этот ФБ в стандарте не рекомендован к исполнению, а служит только примером. Это связано с некоторыми сложностями, которые мы рассмотрим ниже.

```
VAR
    dtCurrent: DT;
    fbRTC: RTC;
END_VAR;

fbRTC(EN := TRUE, PDT := DT#2000-12-12-12:00:00, CDT => dtCurrent);
```

Здесь переменной dtCurrent будет определено значение текущего времени в формате DATE_AND_TIME. Важно понимать, что как только на EN появится сигнал, переменной CDT будет присвоено то, что было передано на PDT, и только потом значение начнет

отсчет. Другими словами, нужна какая-то система, которая будет хранить точку отсчета. Неудобно постоянно инициализировать этот блок при каждом включении.

Недостаток этого блока в том, что если сделать несколько экземпляров блока, не связанных между собой, то они могут считать по-разному и иметь отклонения относительно друг друга.

Решение лежит в реализации аппаратных RTC, что и сделано на многих КТСИ. При наличии аппаратного RTC, ФБ может инициализироваться от его значений, если PDT не указан. Аппаратные часы RTC хороши тем, что сохраняют ход во время отключения КТСИ в небольшой статической памяти. Но аппаратные часы нужно обязательно настроить при первом включении.

2 SysLibTime – специальная библиотека в CoDeSys.

```
VAR
    stTime: SysTime64;
    stTimeExt: SystemTimeDate;
    fbCurTime: CurTimeEx;
END_VAR

stTimeExt.dwHighMsec    := 0;
stTimeExt.dwLowMSecs    := 0;
stTimeExt.Year          := 0;
stTimeExt.Month         := 0;
stTimeExt.Day           := 0;
stTimeExt.Hour          := 0;
stTimeExt.Minute        := 0;
stTimeExt.Second        := 0;
stTimeExt.Milliseconds  := 0;
stTimeExt.DayOfWeek     := 0;

IF Save THEN
    stTimeExt.Year      := spTimeYear;
    stTimeExt.Month     := spTimeMonth;
    stTimeExt.Day       := spTimeDay;
    stTimeExt.HOUR      := spTimeHour;
    stTimeExt.Minute    := spTimeMinute;
END_IF;

fbCurTime(SystemTime := stTime, TimeDate := stTimeExt);
```

На выходе stTimeExt - это структура, которая содержит данные даты и времени в отдельных переменных. В начале каждого цикла мы инициализируем переменную stTimeExt и сбрасываем все в 0. Почему? Потому что если в этой переменной есть значения, то fbCurTime может расценить их так, как будто вы назначаете новое время. Сбросив все в

0, мы убеждаемся, что будем получать текущее время, а не устанавливать его.

Мы видим здесь пример изменения значения аппаратных часов времени. Надо заметить что в стандартном исполнении библиотеки, подобный функционал не предусматривается, а возможен на КТСИ Овен, с доработанной ими библиотекой. Предположим, что в переменных spTime*** хранятся раздельно часы, минуты, год, месяц и т.д., которые нужно назначить, и если мы подадим сигнал на Save, то произойдет назначение времени.

Эта функция реализована на многих КТСИ, но у ней есть недостаток - отсутствие выходной переменной типа DT, даты и времени. Делать дальнейшие вычисления, которые мы рассмотрим далее, будет сложнее. Мы увидим, как из этих данных создать переменную типа DT или любого другого типа времени.

3 SysLibRtc – специальная библиотека в CoDeSys.

Ее использование самое простое и с самым удобным результатом.

```
VAR
    dtCurrent: DT;
END_VAR

dtCurrent := SysRtcGetTime(TRUE);
```

Теперь dtCurrent содержит дату и время в формате DT, и мы можем использовать это в математических вычислениях, приведенных ниже.

Смешивать в выражениях переменные разного типа нельзя. При необходимости применяются операции преобразования типов. Преобразования из одного типа времени в другой осуществляются при помощи функций преобразования, которые, как правило, присутствуют в любой IDE.

```
VAR
    dt1 : DATE_AND_TIME := DT#2017-11-08-19:25:00;
    tod1: TIME_OF_DAY;
    t1:   TIME;
    d1:   DATE;
END_VAR

tod1 := DATE_AND_TIME_TO_TOD(dt1);
d1   := DATE_AND_TIME_TO_DATE(dt1);
t1   := TIME_OF_DAY_TO_TIME(tod1);
```

Общий принцип преобразования - это вызов функции по структуре [ИЗ

ФОРМАТА]_TO_[В ФОРМАТ].

Часто мы получаем время в разбитом формате - как в примере получения времени через библиотеку SysLibTime. Она хранит месяц, год, час, минуты и т.д. по отдельности, в разных переменных. Как их преобразовать в одну переменную времени DT?

Есть 2 способа. Один - через математику, путем получения нужного числа секунд или миллисекунд, второй - через строки.

Пример через математику:

Упаковка часов, минут, секунд и миллисекунд в TIME

```
FUNCTION HMS_TO_TIME : TIME
    VAR_INPUT
        h, m, sec, ms : UINT;
    END_VAR

    HMS_TO_TIME := DWORD_TO_TIME(
        ((h * 60 + m) * 60 + sec) * 1000 + ms);
END_FUNCTION

PROGRAM
    VAR
        tMyTime: TIME;
    END_VAR

    (* tMyTime = T#10h20m *)
    tMyTime := HMS_TO_TIME(10, 20, 0, 0);
END_PROGRAM
```

Все очень просто: переводим часы h в минуты и прибавляем m минуты. Общее число минут переводим в секунды, умножением на 60 и прибавляем секунды sec, то же делаем и для миллисекунд. Получаем количество миллисекунд и преобразуем в тип TIME.

Пример через строку: упаковка года, месяца, дня, часов, минут, секунд и миллисекунд в DT

```
FUNCTION YMDHMS_TO_DT : DT
    VAR_INPUT
        y, mn, d, h, m, sec, ms : UINT;
    END_VAR
    VAR
        sTemp: STRING;
    END_VAR

    sTemp := CONCAT("DT#", UINT_TO_STRING(y));
    sTemp := CONCAT(sTemp, "-");
    sTemp := CONCAT(sTemp, UINT_TO_STRING(mn));
    sTemp := CONCAT(sTemp, "-");
    sTemp := CONCAT(sTemp, UINT_TO_STRING(d));
    sTemp := CONCAT(sTemp, "-");
    sTemp := CONCAT(sTemp, UINT_TO_STRING(h));
    sTemp := CONCAT(sTemp, ":");
    sTemp := CONCAT(sTemp, UINT_TO_STRING(m));
    sTemp := CONCAT(sTemp, ":");
    sTemp := CONCAT(sTemp, UINT_TO_STRING(s));

    YMDHMS_TO_DT := STRING_TO_DT(sTemp);
END_FUNCTION
```

В примере сначала методом конкатенации создаем строку. В нашем случае получается строка типа DT#2000-01-01-12:00:00. Потом преобразуем эту строку в тип DT стандартной функцией преобразования типов STRING_TO_DT.

Единственный его недостаток в том, что он требует больше времени на обработку процессором и увеличивает общее время одного цикла КТСИ. С другой стороны, если операций с датами немного, то это не существенно. Хотя разница в скорости математического способа и строкового примерно в 4 раза, речь идет о микросекундах. Например, приведенный нами пример займет 20 микросекунд, а математический 5 микросекунд. Это все еще 50 раз за одну миллисекунду. Если делать тысячи операций с датами, тогда это может существенно отразиться на производительности, и, если это критично, следует использовать математический метод.

Может, в этом конкретном примере строк получилось больше, но далее мы увидим, что такой способ более элегантный и краткий. Я предпочитаю всегда использовать именно его.

Все математические примеры далее приведены для понимания основных принципов расчета времени, в качестве альтернативы я покажу, как ту же задачу решать строковым способом.

Еще пример, как упаковать год, месяц день в переменную типа DATA, но уже математическим способом.

Здесь есть сложность. Для получения формата DATA нужно вычислить количество дней от нулевой даты до указанной. Для этого необходимо узнать, сколько дней прошло с 1-го января 1970-го года. Но мы не можем просто умножить количество лет на 365 дней в году, ведь у нас есть високосные годы.

Давайте сначала сделаем функцию, которая посчитает, сколько прошло дней до указанного года.

```
FUNCTION DAYS_TILL_YEAR : UINT
    VAR_INPUT
        y: UINT;
    END_VAR

    (* Получаем количество лет прошло с 1970го года *)
    y := y - 1970;

    DAYS_TILL_YEAR := y * 365 + ((y+1)/4) -
        ((y+69)/100) + ((y+369)/400);
END_FUNCTION
```

- y * 365 - узнаем, сколько дней прошло от 1970-го года, если бы у нас все года имели одно и тоже количество дней.
- ((y+1)/4) - Прибавляем по дню на каждый високосный год. Первый за этот период високосный год был в 1973 г., так что y+1 позволит получить целое число количества високосных годов с 1970 г. до y.
- ((y+69)/100) и ((y+369)/400) - отнимаем число лет, кратное 100 и не кратное 400. Эти поправки включаются в 2001 году. Так как 2000 год был високосным, мы его не исключаем.

Понадобится еще одна функция, чтобы определить количество дней от начала года до указанного месяца.

```
FUNCTION DAYS_TILL_MONTH : UINT
    VAR_INPUT
        y, m: UINT;
    END_VAR
    VAR
        days: ARRAY[1..12] OF UINT :=
            0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334;
    END_VAR

    DAYS_TILL_MONTH := days[m] +
        BOOL_TO_INT(m > 2 AND IS_LEAP_YEAR(y));
END_FUNCTION
```

- days[m] возвращает нужное число, так как массив days содержит нужный ответ, но без учета високосного года.
- BOOL_TO_INT(m > 2 AND IS_LEAP_YEAR(y)) добавляет единицу, если год високосный и февраль уже прошел.

Обратите внимание, что эта функция будет работать, если m указывает от 1 до 12.

Теперь можно лаконично перевести год, месяц и день в формат DATE.

```
FUNCTION YMD_TO_DATE : DATE
    VAR_INPUT
        y, m, d: UINT;
    END_VAR

    YMD_TO_DATE := DWORD_TO_DATE(
        86400 * (DAYS_TILL_YEAR(y) + DAYS_TILL_MONTH(y, m) + d - 1)
    );
END_FUNCTION
```

Смотрите, как легко, в одной функции, с этим справился бы строковый метод.

```
FUNCTION YMD_TO_DATE : DATE
    VAR_INPUT
        y, m, d: UINT;
    END_VAR
    VAR
        sTemp: STRING;
    END_VAR


    sTemp := CONCAT("D#", UINT_TO_STRING(y));
    sTemp := CONCAT(sTemp, "-");
    sTemp := CONCAT(sTemp, UINT_TO_STRING(m));
    sTemp := CONCAT(sTemp, "-");
    sTemp := CONCAT(sTemp, UINT_TO_STRING(d));

    YMD_TO_DATE := STRING_TO_DATE(sTemp);
END_FUNCTION
```

Создаем строку типа D#2000-01-01 и конвертируем в дату. Самое главное - здесь не обязательно понимать все тонкости расчета времени математическим способом, вести учет високосных лет. Это все равно, что просто напечатать дату символами, а КТСИ ее сам распознает.

Иногда нужно наоборот – из переменной типа время получить отдельно часы, минуты и т.д.

Пример - извлечение часов, минут, секунд и миллисекунд из TIME:

```
FUNCTION TIME_TO_ELEMENTS : BOOL
    VAR_INPUT
        tTime : TIME;
    END_VAR
    VAR_OUTPUT
        h, m, sec, ms : UINT;
    END_VAR
    VAR
        tTemp : UINT;
    END_VAR

    (* Переводим временной промежуток в количество миллисекунд *)
    tTemp := TIME_TO_UINT(tTime);

    (* 1000 миллисекунд в секунде. Остаток от деления и будет
    остаток в  миллисекундах *)
    ms := tTime MOD 1000;

    (* Переводим временной промежуток в количество секунд *)
    tTemp := tTemp / 1000;

    (* 60 секунд в минуте. Делим наш промежуток в секундах на 60 и
    получаем количество полных минут, а остаток от деления будут
    лишние секунды. *)
    sec := tTemp MOD 60;

    (* Переводим временной промежуток в количество минут *)
    tTemp := tTemp / 60;

    (* 60 минут в часе, а tTemp количество минут. Остаток от деления
    лишние минуты. *)
    m := tTemp MOD 60;

    (* Делим количество минут на 60 и получаем полное количество часов *)
    h := tTemp / 60;

    TIME_TO_ELEMENTS := TRUE;
END_FUNCTION
```

Обратите внимание, что здесь используется функция и имеется несколько выходов, что не типично для функции. Вероятно, удобней было бы создать несколько функции по одному выходу на функцию, как TIME_TO_HOUR, TIME_TO_SECOND, и т.д. Либо нужно создать функцию, которая вернет структуру, содержащую по отдельности все элементы времени.

Если нужно узнать, сколько часов, мы смотрим на строку времени 12:35 на часах и выделяем 12 как часы. То же самое можно сделать и тут. Вот пример решения этой задачи строковым способом:

```
FUNCTION TIME_TO_ELEMENTS : BOOL
    VAR_INPUT
        tTime : TIME;
    END_VAR
    VAR_OUTPUT
        h, m, sec : UINT;
    END_VAR
    VAR
        sTemp: STRING;
    END_VAR

    (* Теперь время имеет строку типа 'TOD#12:12:59' *)
    sTemp := TOD_TO_STRING(TIME_TO_TOD(tTime));

    h   := STRING_TO_UINT(MID(sTemp, 5, 2));
    m   := STRING_TO_UINT(MID(sTemp, 8, 2));
    sec := STRING_TO_UINT(MID(sTemp, 11, 2));

    TIME_TO_ELEMENTS := TRUE;
END_FUNCTION
```

Пример - извлечение года, месяца и дня из DATE:

Извлечение года, месяца и дня из DATE - задача сложней, чем упаковка, которую мы рассмотрели выше в функции YMD_TO_DATE.

Хотя мы и знаем полное число суток, трудно рассчитать сколько в них полных лет или месяцев, так как все они имеют переменную длительность. Сначала будем определять год или месяц приблизительно, с возможной ошибкой на один больше. Затем, обратным преобразованием проверим результат на наличие ошибки. В этом примере я покажу, как использовать структуру, чтобы можно было вернуть функцией одно значение и не создавать VAR_OUTPUT для возврата множественных значений. Сначала объявим структуру, которая будет содержать по отдельности год, месяц и день:

```
TYPE YMD:
    STRUCT
        y: UINT;
        m: UINT;
        d: UINT;
    END_STRUCT
END_TYPE
```

Создадим функцию, которая вернёт эту структуру. Таким образом, возвращаем только одну переменную, при этом, она содержит 3 значения.

```
FUNCTION DATE_TO_YMD : YMD
    VAR_INPUT
        dat: DATE;
    END_VAR
    VAR
        uiDays, uiDayInYear: UINT;
    END_VAR

    (* Получаем число дней с 1970г *)
    uiDays := DWORD_TO_UINT(DATE_TO_DWORD(dat) / 86400);

    (* Грубо вычисляем год *)
    DATE_TO_YMD.y := uiDays / 365 + 1970;

    (* Сверяем количество полученных дней с обратным преобразованием,
       и если обратное преобразование больше, значит у нас на один
       год больше, чем нужно. *)
    IF uiDays < DAYS_TILL_YEAR(DATE_TO_YMD.y) THEN
        DATE_TO_YMD.y := DATE_TO_YMD.y - 1;
    END_IF;

    (* Определяем, сколько дней остаток от начала найденного года,
    чтобы посчитать месяцы *)
    uiDayInYear := uiDays - DAYS_TILL_YEAR(DATE_TO_YMD.y) + 1;

    (* Грубо считаем количество месяцев в остатке дней *)
    DATE_TO_YMD.m := MIN(uiDayInYear/29 + 1, 12);

    (* Сверяем количество полученных месяцев с обратным
    преобразованием,и если обратное преобразование больше или равно,
    значит у нас на один месяц больше чем нужно. *)
    IF uiDayInYear <= DAYS_TILL_MONTH(DATE_TO_YMD.y, DATE_TO_YMD.m) THEN
        DATE_TO_YMD.m := DATE_TO_YMD.m - 1;
    END_IF;

    (* Осталось вычислить остаток дней. Дни нам проверять не нужно *)
    DATE_TO_YMD.d := uiDayInYear -
        DAYS_TILL_MONTH(DATE_TO_YMD.y, DATE_TO_YMD.m);

END_FUNCTION
```

Обратим внимание, что здесь мы еще используем DAYS_TILL_YEAR и DAYS_TILL_MONTH, написанные раньше.

Далее, на радость нам, представляю строковый способ той же функции. Мало того, что он весьма компактный, так еще не нужно ничего вычислять и проверять. Жертвуем мы всего 10 микросекунд или 0.01 миллисекунды:

```
FUNCTION DATE_TO_YMD : YMD
    VAR_INPUT
        dat: DATE;
    END_VAR
    VAR
        sTemp: STRING;
    END_VAR

    (* Получаем дату строкой 'D#2000-01-01' *)
    sTemp := DATE_TO_STRING(TIME_TO_TOD(tTime));

    DATE_TO_YMD.y := STRING_TO_UINT(MID(sTemp, 3, 4));
    DATE_TO_YMD.m := STRING_TO_UINT(MID(sTemp, 8, 2));
    DATE_TO_YMD.d := STRING_TO_UINT(MID(sTemp, 11, 2));
END_FUNCTION
```

Если вы поймете принцип, который лежит в основе работы строкового метода как вычленения, так и формирования переменных, то сильно облегчите себе жизнь.

Проще всего делать вычисления с типом TIME. Можно выполнять присваивание, сравнение, сложение, вычитание, умножение и деление на число, использовать их в стандартных ограничителях и мультиплексорах:

```
VAR
    t1, t2, t3 : TIME;
END_VAR

t2:= T#0d;
t1 := t2 - T#2m; (* 0мин – 2мин = -2мин*)
t1 := t1 + T#5m; (* -2мин + 5мин = 3мин*)
t1 := t1/2; (* 3мин/2 = 1мин30сек *)
t3 := LIMIT(T#2s, t1, T#30s); (* t3 = 30сек *)
```

При проведении вычислений нужно обратить внимание на 3 момента:

1 Могут возникать отрицательные длительности. Допустимы они или нет, зависит от проекта и задачи, но нужно об этом помнить.

2 Общая длина временного промежутка ограничена, и может произойти переполнение. Убедитесь, что в результате вычислений вы не выходите за 1193 часа.

3 Нельзя делить TIME на TIME. Можно делить TIME на число. Если нужно поделить TIME на TIME, нужно преобразовать одно TIME в число.

```
t1 := t2 / TIME_TO_DWORD(t3);
```

Делать вычисления с TIME_OF_DAY сложнее из-за его функциональной принадлежности. Это всего лишь отметка времени дня в сутках, а сутки - это 24 часа. Имеет смысл или получить разницу между двумя отметками времени, что даст TIME, или

провести сравнение. Операции умножения или деления для времени суток не имеют смысла.

Для осуществления сравнений можно использовать стандартные операторы GT, <, SEL, и т.д.

```
MyTOD := SEL(CurTOD > TOD#22:00, TOD#08:00, TOD#18:00);
```

Пример по сравнению текущего времени дня:

Создадим функцию определения, что текущее время находится в промежутке между двумя отметками времени TIME_OF_DAY. Данное вычисление делается путем несложного сравнения, до тех пор, пока у нас не будет ситуации, когда промежуток выйдет из диапазона одних суток. Например, сравнить, если текущее время в промежутке между 20:00 вечера и 2:00 ночи следующих суток. Функция должна учесть подобные обстоятельства.

```
FUNCTION TOD_IS_BETWEEN : BOOL
    VAR_INPUT
        CurrTime: TIME_OF_DAY; (* текущее время *)
        FromTime: TIME_OF_DAY; (* нижняя граница *)
        ToTime:   TIME_OF_DAY; (* верхняя граница *)
    END_VAR

    TOD_IS_BETWEEN := (
        FromTime > ToTime
        AND (
            CurrTime > FromTime OR CurrTime < ToTime
        )
    ) OR (
        FromTime < ToTime
        AND (
            CurrTime < ToTime AND CurrTime > FromTime
        )
    );
END_FUNCTION
```

Первая часть (FromTime > ToTime AND (CurrTime > FromTime OR CurrTime < ToTime)) - это сравнение, при условии, что FromTime > ToTime, а значит, имеется переход из одних суток в другие, поэтому сравнение будет через оператор OR - (CurrTime > FromTime OR CurrTime < ToTime).

Вторая часть сравнения (FromTime < ToTime AND (CurrTime < ToTime AND CurrTime > FromTime)) проверяет, что диапазон расположен в пределах одних суток, тогда сравнение делается через оператор AND.

Пример по вычислению разницы отметок времени:

При вычислении разницы возникает такая же проблема. Разница вычисляется легко,

если диапазон отметок времени лежит в пределах одних суток. А если он в разных сутках? Вот функция вычисления разницы времени:

```
FUNCTION TOD_DIFF : TIME
    VAR_INPUT
        FromTime: TIME_OF_DAY; (* нижняя граница *)
        ToTime:   TIME_OF_DAY; (* верхняя граница *)
    END_VAR

    IF ToTime < FromTime THEN
        TOD_DIFF := T#24h - TOD_TO_TIME(FromTime) +
            TOD_TO_TIME(ToTime);
    ELSE
        TOD_DIFF := ToTime - FromTime;
    END_IF
END_FUNCTION
```

В данном случае более надежным было бы вычисление с использованием даты DT. Тогда, чтобы получить разницу времени, нужно просто вычесть одно время из другого.

```
TResult := DT_To - DT_From;
```

Пример на определение дня недели:

Вычислить день недели довольно просто, так как количество дней в неделе не менялось с начала мироздания и не смещалось. Все, что нам нужно, это разделить количество дней с известной нам даты, которая, как говорилось ранее, 1 января 1970 года, на 7 дней в неделю. Получаем количество полных недель с этой даты, а остаток от деления как раз и будет наш день недели.

```
FUNCTION DAY_OF_WEEK : USINT
    VAR_INPUT
        dDate : DATE;
    END_VAR
    VAR
        NumOfDays : UINT;
    END_VAR

    NumOfDay := DATE_TO_DWORD(dDate) / 86400;
    NumOfDay := NumOfDay + 3;
    DAY_OF_WEEK := DWORD_TO_UINT(NumOfDay MOD

END_FUNCTION
```

Этот расчет можно было сделать в одной строке, но я разделил его на несколько строк для наглядности примера.

В первой строке преобразуем дату в количество секунд с 1 января 1970 года DATE_TO_DWORD(dDate) и делим на 86400 (количество секунд в сутках). Вычисляем, сколько суток прошло с 1 января 1970 года.

Пока опустим вторую строку, она станет более понятной позже. Перейдём к разбору третей строки.

В третьей строке получаем остаток от деления DWORD_TO_UINT(NumOfDay MOD 7). Известно, что 1 января 1970-го был четверг. Если наше число ровно делится на 7 без остатка, значит вычисляемая дата - тоже четверг. Если остаток 1, то пятница, 2 - суббота и т.д. Четверг становится базовым 0 для отсчета дней недели.

Такой счет с четверга, как 0, не очень удобен. Обычно счет идет с 0 до 6, где 0 это воскресенье или понедельник. В большинстве стран мира принято считать первым днем недели воскресенье, но в некоторых странах, как и в нашей, это понедельник. В мире КТСИ общепринятой практикой считается использование стандарта ISO-8601.

ISO 8601 — международный стандарт, выпущенный организацией ISO (International Organization for Standardization), который описывает форматы дат и времени, и даёт рекомендации для его использования в международном контексте.

В этом стандарте первым днем недели считается понедельник, поэтому будет иметь смысл вернуть число от 0 до 6, где 0 это понедельник:

- 0 - Понедельник
- 1 - Вторник
- 2 - Среда
- 3 - Четверг
- 4 - Пятница
- 5 - Суббота
- 6 - Воскресенье

В этом счете четверг - третий день. Значит, чтобы изменить базовый 0 четверга на 3, нужно прибавить 3. Таким образом, мы сдвигаем счет. Прибавив 3, мы делаем так, что остаток от деления будет не 0, а 3. А это как раз четверг, если взять за 0 понедельник. Значит, если мы хотим вести счет от воскресенья, нужно прибавить 4, чтобы вернулось 4. Если 0 это воскресенье, то четверг будет четвертым днем.

Пример на определение високосного года:

Високосным считается любой год, который делится на 400, и годы, которые делятся на 4, но не делятся на 100. Вводным параметром в функцию будет год.

```
FUNCTION IS_LEAP_YEAR : BOOL
    VAR_INPUT
        uiYear : UINT;
    END_VAR
    IS_LEAP_YEAR := uiYear MOD 400 = 0 OR
        uiYear MOD 4 = 0 AND uiYear MOD 100 <> 0;
END_FUNCTION
```

```
FUNCTION DAYS_IN_YEAR : UINT
    VAR_INPUT
        uiYear: UINT;
    END_VAR

    IF IS_LEAP_YEAR(uiYear) THEN
        DAYS_IN_YEAR := 366;
    ELSE
        DAYS_IN_YEAR := 365;
    END_IF;
END_FUNCTION
```

Пример на вычисление порядкового дня в году:

```
FUNCTION DAY_OF_YEAR : UINT
    VAR_INPUT
        month: UINT; (* Текущий месяц *)
        day:   UINT; (* Текущий день *)
    END_VAR

    DAY_OF_YEAR := DAYS_TILL_MONTH(month) + day;
END_FUNCTION
```

У нас уже есть функция DAYS_TILL_MONTH, которая вычисляет количество дней в году до указанного месяца. Остается только добавить количество дней.

Пример на вычисления порядкового номера недели в году

```
FUNCTION WEEK_OF_YEAR : UINT
    VAR_INPUT
        cur_date: DATE; (* Текущая дата *);
    END_VAR

    WEEK_OF_YEAR := ((DAY_OF_YEAR(cur_date) + 6) / 7);
    IF
        DAY_OF_WEEK(cur_date) <
        DAY_OF_WEEK(YEAR_STARTS(cur_date))
    THEN
        WEEK_OF_YEAR := WEEK_OF_YEAR + 1;
    END_IF;
END_FUNCTION
```

Для вычисления порядкового номера недели необходимо знать, с какого дня недели начинается год. Для этого нужно из текущей даты извлечь текущий год и получить день недели на 01 января текущего года. Для простоты кода я создал отдельную функцию YEAR_STARTS, которая строковым способом извлекает дату на 1 января текущего года.

```
FUNCTION YEAR_STARTS : DATE
    VAR_INPUT
        cur_date: DATE; (* Текущая дата *);
    END_VAR
    VAR_TEMP
        str: STRING[25]; (* Временная строка *)
    END_VAR

    str := DATE_TO_STRING(cur_date); (* получаем строку `D#2000-12-30` *)
    YEAR_STARTS := STRING_TO_DATE(
        CONCAT(CONCAT('D#', MID(str, 3, 4)), '-01-01')
    );
END_FUNCTION
```

Пример на определение времени восхода и заката

Время захода и восхода солнца очень часто используется для организации освещения или, например, динамической подсветки зданий.

Первая функция, которая нам понадобится, вернет время солнцестояния на заданной широте в заданное время:

```
FUNCTION SUN_MIDDAY : TOD
VAR_INPUT
    LON : REAL;
    UTC : DATE;
END_VAR
VAR
    T : REAL;
    OFFSET : REAL;
END_VAR

    T := INT_TO_REAL(DAY_OF_YEAR(utc));
    OFFSET := -0.1752 * SIN(0.033430 * T + 0.5474) - 0.1340 *
        SIN(0.018234 * T - 0.1939);

    SUN_MIDDAY := HOUR_TO_TOD(12.0 - OFFSET - lon * 0.0666666666666);
END_FUNCTION
```

Еще нам понадобиться функция RAD которая конвертирует градусы в радианы.

```
FUNCTION RAD : REAL
    VAR_INPUT
        DEG : REAL;
    END_VAR

    RAD := MODR(0.0174532925199433 * DEG, math.PI2);
END_FUNCTION
```

Теперь сама функция расчета времени заката и восхода:

```
FUNCTION_BLOCK SUN_TIME
    VAR_INPUT
        LATITUDE : REAL;            (* Широта *)
        LONGITUDE : REAL;           (* Долгота  *)
        UTC : DATE;                 (* Текущее время в UTC *)
        (* Градус над горизонтом, при котором уже считается
        наступление заката или восхода *)
        H : REAL := -0.83333333333;
    END_VAR
    VAR_OUTPUT
        MIDDAY:          TOD;  (* Время дня солнцестояния *)
        SUN_RISE:        TOD;  (* Время восхода *)
        SUN_SET:         TOD;  (* Время заката *)
        (* Угол наклона солнца в солнцестояние в градусах *)
        SUN_DECLINATION: REAL;
    END_VAR
    VAR
        dk:    REAL; (* Угол наклона солнца в полдень в радианах *)
        delta: TIME; (* Дельта от солнцестояния до восхода или заката *)
        b:     REAL;
    END_VAR

    MIDDAY := SUN_MIDDAY(longitude, utc);
    b  := latitude * 0.0174532925199433;
    dk := 0.40954 * SIN(0.0172 * (INT_TO_REAL(DAY_OF_YEAR(utc)) - 79.35));

    SUN_DECLINATION := DEG(DK);

     IF SUN_DECLINATION > 180.0 THEN
        SUN_DECLINATION := SUN_DECLINATION - 360.0;
     END_IF;
     SUN_DECLINATION := 90.0 - LATITUDE + SUN_DECLINATION;

     delta := HOUR_TO_TIME(
        ACOS((SIN(RAD(H)) - SIN(B) * SIN(DK)) / (COS(B) * COS(DK))) *
        3.819718632
     );

     SUN_RISE := MIDDAY - delta;
     SUN_SET  := MIDDAY + delta;
END_FUNCTION_BLOCK
```

Другие функции, которые здесь используются, такие как DEG(), MODR(), вы найдете в математических примерах типов переменных REAL.

2.8 Методика поиска отказов

Исправное функционирование модулей КТСИ обозначается постоянным свечением зеленым цветом светодиодов «Питание», «CAN».

2.9 Устранение отказов

При возникновении отказов следует обратиться в службу технической поддержки изготовителя.

КТСИ поддерживает функцию «горячей замены» модулей ввода/вывода (кроме процессорного модуля) при возникновении отказов.

При обнаружении аппаратных неисправностей следует руководствоваться положениями, изложенными в руководстве по эксплуатации.

2.10 Техническая поддержка

Для обращения в службу технической поддержки Пользователю необходимо сформировать запрос на сайте технической поддержки: https://uzola.ru, либо отправить письмо по электронной почте: sekr@uzola.ru.

Обращение обязательно должно содержать следующие сведения:

1) подробное описание проблемы;
2) наименование объекта и его месторасположение;
3) наименование системы автоматизации;
4) серийный номер КТСИ;
5) версия IDE CODESYS;
6) файл экспорта сетевых настроек контроллера;
7) архив с лог-файлами, включающими в себя период времени, когда произошел отказ;
8) дата и время возникновения отказа;
9) а также периодичность и устойчивость повторения подобных отказов в случае, если такая информация имеется.

Желательно прислать проект для CODESYS, т.к. это может значительно упростить и ускорить процесс поиска причины отказа.

Лог-файлы, скопированные на компьютер, желательно поместить в архив. Объем заархивированных текстовых файлов сокращается примерно в 10 раз.

Для того, чтобы узнать, как получить необходимую информацию (сведения о версии Codesys, версии СПО и т.д.), следует ознакомиться с содержимым документа «Системное руководство».

2.11 Информация об Изготовителе

Изготовитель: ГК «Узола», Россия, г. Нижний Новгород, ул. Ларина, д.7а

Телефон: 8-800-7-759-759.

Сайт: https://uzola.ru .

Email: sekr@uzola.ru

Лист регистрации изменений

Изм.	Номера листов (страниц)				Всего листов (страниц) в документе	Номер документа	Входящий номер сопроводительного документа и дата	Подпись	Дата
	изме-нённых	заме-нённых	новых	аннули-рованных					

Подп. и дата | Инв. № дубл. | Взам. инв. № | Подп. и дата | Инв. № подл.

Изм.	Лист	№ докум.	Подп.	Дата	МПВР.421457.000И3	Лист 136