



CODESYS V3.5

Реализация нестандартных протоколов



Руководство пользователя

01.12.2018

версия 2.0

Оглавление

Глоссарий.....	2
1 Цель и структура документа.....	2
2 Основные сведения об обмене данными по последовательным интерфейсам.....	3
2.1 Общие принципы организации обмена	3
2.2 Работа с COM-портом	3
2.3 Типы протоколов обмена	3
2.4 Обработка ошибок обмена	5
3 Библиотека CAA SerialCom.....	6
3.1 Добавление библиотеки в проект CODESYS.....	6
3.2 Структура COM.PARAMETER.....	7
3.3 Список глобальных констант CAA_Parameter_Constants.....	7
3.4 ФБ COM.Open	8
3.5 ФБ COM.Write	9
3.6 ФБ COM.Read	10
3.7 ФБ COM.Close.....	11
4 Пример опроса модуля MB110-8A по протоколу DCON.....	12
4.1 Формулировка задачи	12
4.2 Описание протокола.....	12
4.3 Алгоритмизация задачи	14
4.4 ФБ управления портом (COM_CONTROL).....	14
4.4.1 Инициализация порта (шаг INITIALIZE)	17
4.4.2 Ожидание управляющего сигнала (шаг WAITING_FOR_SIGNAL)	17
4.4.3 Открытие порта (шаг OPEN_PORT)	18
4.4.4 Закрытие порта (шаг CLOSE_PORT)	21
4.5 ФБ опроса модуля (MV110_8A_DCON).....	23
4.5.1 Подготовка запроса (шаг CREATE_REQUEST).....	25
4.5.2 Отправка запроса (шаг SEND_REQUEST)	29
4.5.3 Получение ответа (шаг RECEIVE_RESPONSE).....	30
4.5.4 Организация задержки (шаг RESPONSE_DELAY).....	34
4.5.5 Завершение цикла опроса (шаг POLLING_CYCLE_ENDS).....	35
4.6 Программа опроса (PLC_PRG)	37
5 Пример опроса счетчика СЭТ-4ТМ.03М	39
5.1 Формулировка задачи	39
5.2 Описание протокола.....	40

5.3	Алгоритмизация задачи	42
5.4	5.4 ФБ управления портом (COM_CONTROL).....	42
5.5	ФБ опроса счетчика (SET_4TM).....	43
5.5.1	Подготовка запроса на открытие канала (шаг CREATE_CHANNEL)	45
5.5.2	Отправка запроса на открытие канала (шаг OPEN_CHANNEL)	49
5.5.3	Получение ответа на запрос открытия канала (шаг RECEIVE_CHANNEL)	50
5.5.4	Организация задержки (шаг RESPONSE_DELAY_CHANNEL)	52
5.5.5	Подготовка запроса на чтение данных (шаг CREATE_REQUEST)	52
5.5.6	Отправка запроса на чтение данных (шаг SEND_REQUEST)	53
5.5.7	Получение ответа (шаг RECEIVE_RESPONSE)	54
5.5.8	Организация задержки (шаг RESPONSE_DELAY)	57
5.5.9	Завершение цикла опроса (шаг POLLING_CYCLE_ENDS).....	57
5.6	Программа опроса (PLC_PRG).....	59
6	Рекомендации и замечания	61
	Приложение А. Листинг примера из п. 4.....	62
A.1	ФБ COM_CONTROL.....	63
A.2	ФБ MV110_8A_DCON	65
A.3	ФБ ANALYZE_DATA.....	69
A.4	Функция BYTE_TO_STRH	70
A.5	Функция _BUFFER_CLEAR	70
A.6	Перечисление COM_STATE	71
A.7	Перечисление DCON_STATE	71
	Приложение Б. Листинг примера из п. 5.....	72
B.1	ФБ COM_CONTROL.....	73
B.2	ФБ SET_4TM	75
B.3	Функция CRC_MG_GEN	81
B.4	Функция _BUFFER_CLEAR	81
B.5	Перечисление COM_STATE	82
B.6	Перечисление SET_4TM_STATE	82

Глоссарий

ПЛК – программируемый логический контроллер.

ФБ – функциональный блок.

1 Цель и структура документа

Одной из ключевых задач контроллеров является организация обмена данными с другими устройствами по последовательным интерфейсам RS-232/485.

В значительном количестве случаев обмен осуществляется по стандартным промышленным протоколам (например, **Modbus RTU**) с помощью средств конфигурирования **CODESYS** или библиотек ОВЕН, но иногда возникает необходимость организовать обмен с устройством, которое поддерживает только свой собственный специфичный протокол (в качестве примеров таких устройств можно привести различные тепло- и электросчетчики, весовые индикаторы, модули ввода-вывода и т. д.). В данном случае пользователю следует реализовать поддержку этого протокола в контроллере с помощью системных библиотек, которые позволяют работать с последовательным портом напрямую. Настоящее руководство описывает решение этой задачи для контроллеров ОВЕН, программируемых в **CODESYS V3.5**

Структура руководства:

В [п. 2](#) приведена основная информация о работе с последовательным портом и реализации протоколов обмена.

В [п. 3](#) приведено описание библиотеки **CAA SerialCom**, которая используется для работы с последовательным портом.

В [п. 4](#) рассмотрен пример опроса модуля **MB110-8A** по протоколу **DCON**.

В [п. 5](#) рассмотрен пример опроса счетчика **СЭТ-4ТМ.03М**.

В [п. 6](#) приведены рекомендации по реализации нестандартных протоколов обмена.

В [приложении](#) приведены листинги программ из примеров, рассмотренных в документе.



ПРИМЕЧАНИЕ

Работа с нестандартными протоколами подразумевает высокую квалификацию программиста, хорошее знание среды **CODESYS V3.5** и языка ST. Документ рекомендуется читать строго последовательно.

2 Основные сведения об обмене данными по последовательным интерфейсам

2.1 Общие принципы организации обмена

В рамках настоящего руководства рассмотрены протоколы, работающие поверх последовательного интерфейса RS-232/485 и основанные на архитектуре **Master – Slave** (ведущий – ведомый).

Данная архитектура подразумевает наличие в сети единственного master-устройства (обычно таким устройством является контроллер), которое последовательно опрашивает slave-устройства (ими могут быть модули ввода-вывода, панели оператора, частотные преобразователи и т. д.). Slave-устройство не может являться инициатором обмена (т. е. оно может только отвечать на полученные запросы).

Реализация нестандартного протокола обмена подразумевает решение двух смежных задач:

1. Организация работы с COM-портом (открытие порта, запись в порт, чтение из порта, закрытие порта).
2. Организация работы с данными (формирование запросов и анализ ответов согласно спецификации протокола).

В простейшем случае обмен происходит непрерывно: сразу после старта master-устройство начинает циклически опрашивать slave-устройства. Тогда в предельно упрощенном виде процедура опроса с точки зрения master-устройства может быть представлена следующим образом:

1. Открыть COM-порт с заданными настройками.
2. Отправить запрос первому slave-устройству.
3. Получить ответ от первого slave-устройства.
4. ...
5. Отправить запрос последнему slave-устройству.
6. Получить ответ от последнего slave-устройства.
7. Перейти к шагу 2.

2.2 Работа с COM-портом

Работа с COM-портом организуется с помощью одной из системных библиотек:

1. **SysCom** (создана на базе библиотеки **SysLibCom** из **CoDeSys 2.3**);
2. **CAA Serial** (разработана под **CODESYS V3**).

В рамках данного документа рассматривается библиотека [CAA Serial](#). Описание библиотеки приведено в [п. 3](#).

При работе с COM-портом выполняются следующие операции:

1. Открытие порта с заданными настройками. Обычно порт однократно открывается при старте контроллера и в дальнейшем не закрывается. Результатом успешного открытия порта является получение дескриптора (handle). Все последующие операции с портом производятся с указанием его дескриптора.
2. Запись в порт. Производится в случае необходимости отправить запрос slave-устройству.
3. Чтение из порта. Производится в случае необходимости получить ответ от slave-устройства.
4. Закрытие порта. В некоторых случаях при возникновении ошибок обмена на стороне master-устройства может потребоваться закрыть порт и открыть его заново.

2.3 Типы протоколов обмена

2. Основные сведения об обмене данными по последовательным интерфейсам

С точки зрения представления передаваемых данных можно выделить два типа протоколов обмена:

1. Бинарные (двоичные) протоколы.

В этом случае каждый передаваемый байт может принимать любые значения (от 00 до FF). Представление каждого типа данных должно соответствовать какому-либо формату. Благодаря этому для каждого запроса заранее известен размер ответа в байтах.

Одним из известных стандартных бинарных протоколов является **Modbus RTU**.

Число с плавающей точкой **111.222** при передаче с помощью бинарного протокола согласно стандарту [IEEE 754](#) будет выглядеть так:

Байт 3	Байт 2	Байт 1	Байт 0
0x43	0xDE	0x71	0xAA

2. Строковые (текстовые) протоколы.

В данном случае каждый передаваемый байт содержит код символа из [таблицы ASCII](#). Размер ответа в байтах на один и тот же запрос может отличаться в зависимости от передаваемых значений – например, значение «5» займет один байт, а «55» – два.

Одним из известных стандартных строковых протоколов является **Modbus ASCII**.

Число с плавающей точкой **111.222** при передаче с помощью строкового протокола будет выглядеть так:

Байт 6	Байт 5	Байт 4	Байт 3	Байт 2	Байт 1	Байт 0
0x32	0x32	0x32	0x2E	0x31	0x31	0x31
«2»	«2»	«2»	«.»	«1»	«1»	«1»

2.4 Обработка ошибок обмена

В простейшем случае обмен происходит непрерывно: сразу после старта master-устройство начинает циклически опрашивать slave-устройства. В предельно упрощенном виде процедура опроса с точки зрения master-устройства может быть представлена следующим образом:

1. Открыть COM-порт с заданными настройками.
2. Сформировать запрос для первого slave-устройства.
3. Отправить запрос первому slave-устройству.
4. Получить ответ от первого slave-устройства.
5. Разобрать ответ от первого slave-устройства.
6. ...
7. Сформировать запрос для последнего slave-устройства.
8. Отправить запрос последнему slave-устройству.
9. Получить ответ от последнего slave-устройства.
10. Разобрать ответ от последнего slave-устройства.
11. Перейти к шагу 2.

В данном случае подразумевается, что все slave-устройства всегда корректно отвечают на запросы. На практике могут возникать следующие ситуации:

1. Ответ не пришел (например, slave-устройство выключено или перестало работать).
Если в программе не предусмотрена обработка такой ситуации, то она будет продолжать ждать ответа, в результате чего обмен с другими устройствами прекратится. Поэтому обычно имеет смысл ожидать ответ в течение некоторого промежутка времени (который зависит от скорости обмена, количества передаваемых данных и индивидуальных особенностей устройства) и по его истечению переходить к опросу следующего регистра данного slave-устройства или следующего slave-устройства.
2. Ответ пришел в виде нескольких фрагментов (в большинстве случаев это происходит на низких скоростях обмена и при значительном количестве получаемых данных).
Соответственно, следует контролировать целостность ответа (в бинарных протоколах – по размеру ответа, в строковых – по наличию в ответе символов начала и конца). Если условие целостности не выполняется, то разбирать ответ не имеет смысла – необходимо сделать еще один запрос (сразу или в следующем цикле опроса).
3. Ответ был поврежден во время передачи (например, из-за действия помех).
Большинство протоколов используют контрольную сумму для проверки корректности ответа.
4. Некоторые slave-устройства в течение определенного интервала времени после ответа не могут принять запрос.
В таких случаях необходимо делать в программе паузу между получением ответа на текущий запрос и отправки следующего запроса.

3 Библиотека CAA SerialCom

3.1 Добавление библиотеки в проект CODESYS

Библиотека **CAA SerialCom** используется для работы с последовательным портом.

Для добавления библиотеки в проект **CODESYS** в **Менеджере библиотек** следует нажать кнопку **Добавить** и выбрать библиотеку **CAA SerialCom**, расположенную в папке **Intern/CAA/System**.

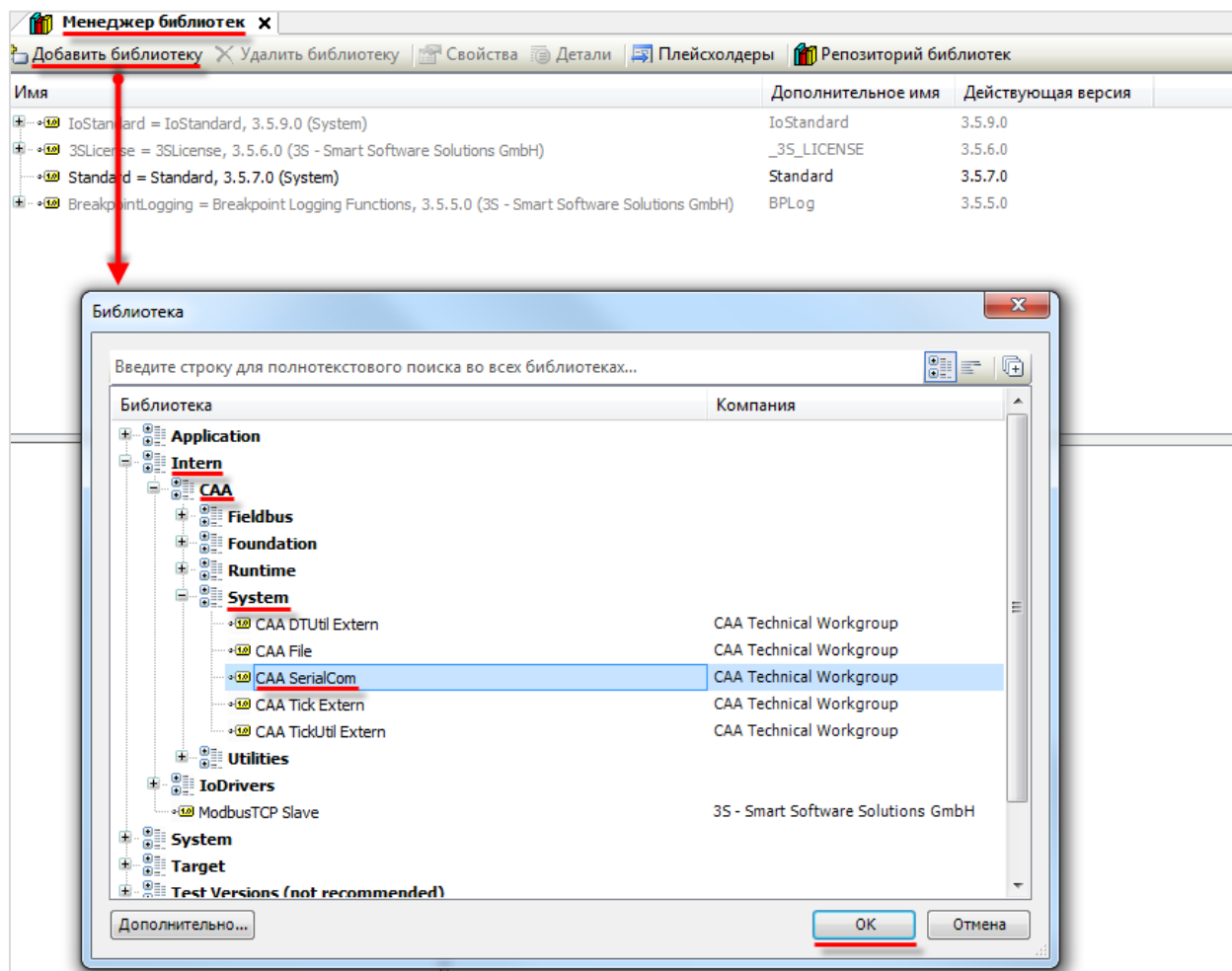


Рисунок 3.1 – Добавление библиотеки CAA SerialCom в проект CODESYS

При объявлении экземпляров ФБ библиотеки следует перед их названием указывать префикс «COM» (пример: **COM.Open**).

3.2 Структура COM.PARAMETER

Структура **COM.PARAMETER** содержит имя параметра COM-порта и его значения. Для задания настроек COM-порта формируется массив структур типа **COM.PARAMETER**, указатель на который передается на вход блока [COM.Open](#).

Таблица 3.1 – Описание переменных структуры COM.PARAMETER

Название	Тип данных	Описание
udiParameterID	UDINT	ID или имя параметра COM-порта. Параметры определены в списке глобальных констант CAA_Parameter_Constants
udiValue	UDINT	Значение параметра

3.3 Список глобальных констант CAA_Parameter_Constants

Список глобальных констант **CAA_Parameter_Constant** содержит ID и имена параметров COM-порта, которые присваиваются переменным **udiParameterID** структуры **COM.PARAMETER**.

Таблица 3.2 – Описание констант списка CAA_Parameter_Constant, поддерживаемых в контроллерах OVEN

Название	ID	Тип данных	Описание
Основные параметры			
udiPort	16#1	UDINT	Номер COM-порта. Номер COM-порта в CODESYS может не совпадать с номером на корпусе прибора. Подробнее см. в руководстве CODESYS 3.5. FAQ
udiStopBits	16#2	UDINT	Число стоп бит. Возможные значения определены в перечислении COM.STOPBIT : ONESTOPBIT – один стоп бит; ONE5STOPBITS – 1.5 стоп бита; TWOSTOPBITS – 2 стоп бита
udiParity	16#3	UDINT	Режим контроля четности. Возможные значения определены в перечислении COM.PARITY : EVEN – четный; ODD – нечетный; NONE – отсутствует
udiBaudrate	16#4	UDINT	Скорость обмена. Возможные значения: 1200/2400/4800/9600/19200/38400/57600/115200
udiByteSize	16#7	UDINT	Количество информационных бит в передаваемых/принимаемых байтах

3.4 ФБ COM.Open

Функциональный блок **COM.Open** используется для открытия COM-порта с заданными настройками.



ПРИМЕЧАНИЕ

Запрещается пытаться открыть уже открытый порт.

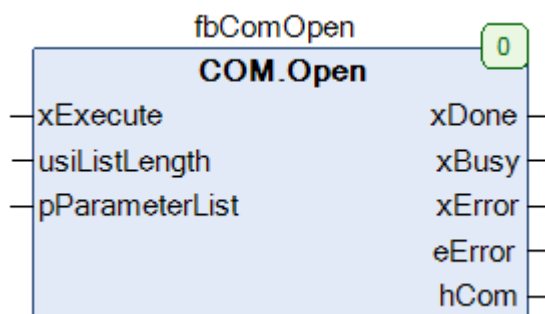


Рисунок 3.2 – Внешний вид ФБ COM.Open на языке CFC

Таблица 3.3 – Описание входов и выходов ФБ COM.Open

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной
usiListLength	USINT	Количество используемых параметров порта
pParameterList	CAA.PVOID	Указатель на массив параметров порта (COM.PARAMETER)
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение TRUE в случае возникновения ошибки
eError	COM.ERROR	Статус работы ФБ (или имя ошибки)
hCom	CAA.HANDLE	Дескриптор (handle) открытого порта

3.5 ФБ COM.Write

Функциональный блок **COM.Write** используется для записи данных в COM-порт.

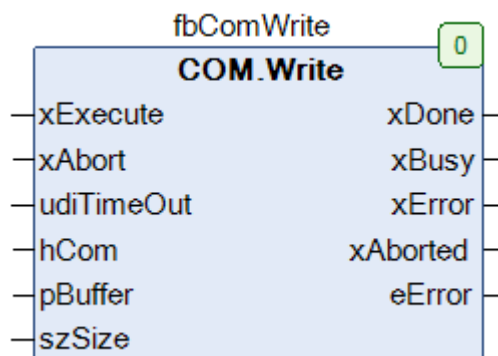


Рисунок 3.3 – Внешний вид ФБ COM.Write на языке CFC

Таблица 3.4 – Описание входов и выходов ФБ COM.Write

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной
xAbort	BOOL	Переменная прекращения работы блока. Если она принимает значение TRUE , то блок немедленно прекращает работу, а выходные переменные сбрасываются к начальным значениям
udiTimeOut	UDINT	Время в мс, через которое блок принудительно завершает свою работу. При значении 0 эта функция отключается
hCom	CAA.HANDLE	Дескриптор (handle) порта, в который происходит запись
pBuffer	CAA.PVOID	Указатель на записываемые данные
szSize	CAA.SIZE	Размер записываемых данных (в байтах)
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение TRUE в случае возникновения ошибки
xAborted	BOOL	Флаг «работа ФБ была прервана»
eError	COM.ERROR	Статус работы ФБ (или имя ошибки)

3.6 ФБ COM.Read

Функциональный блок **COM.Read** используется для чтения данных из COM-порта.

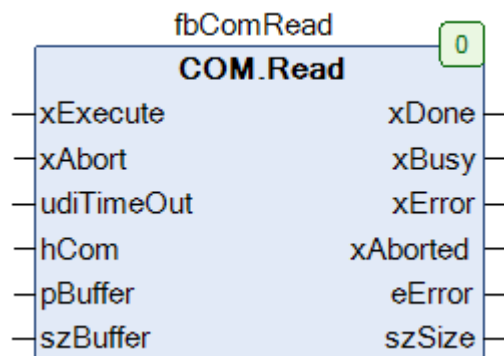


Рисунок 3.4 – Внешний вид ФБ COM.Read на языке CFC

Таблица 3.5 – Описание входов и выходов ФБ COM.Read

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной
xAbort	BOOL	Переменная прекращения работы блока. Если она принимает значение TRUE , то блок немедленно прекращает работу и выходные переменные сбрасываются к начальным значениям
udiTimeOut	UDINT	Время в мс, через которое блок принудительно завершает свою работу. При значении 0 эта функция отключается
hCom	CAA.HANDLE	Дескриптор (handle) порта, из которого происходит чтение
pBuffer	CAA.PVOID	Указатель, по которому будут записаны считанные данные
szBuffer	CAA.SIZE	Размер считываемых данных (в байтах)
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение TRUE в случае возникновения ошибки
xAborted	BOOL	Флаг «работа ФБ была прервана»
eError	COM.ERROR	Статус работы ФБ (или имя ошибки)
szSize	CAA.SIZE	Размер считанных данных (в байтах)

3.7 ФБ COM.Close

Функциональный блок **COM.Close** используется для закрытия COM-порта.



ПРИМЕЧАНИЕ

Запрещается пытаться закрыть уже закрытый порт.

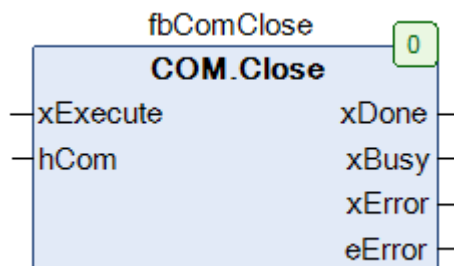


Рисунок 3.5 – Внешний вид ФБ COM.Close на языке CFC

Таблица 3.6 – Описание входов и выходов ФБ COM.Close

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной
hCom	CAA.HANDLE	Дескриптор (handle) закрываемого порта
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение TRUE в случае возникновения ошибки
eError	COM.ERROR	Статус работы ФБ (или имя ошибки)

4 Пример опроса модуля MB110-8A по протоколу DCON

4.1 Формулировка задачи

В качестве примера будет рассмотрен опрос модуля [MB110-8A](#) по протоколу **DCON** с помощью контроллера **СПК1xx [M01]**. Модуль подключен к порту COM3, его сетевые настройки приведены в таблице ниже.

Таблица 4.1 – Сетевые настройки модуля MB110-8A

Параметр	Значение
COM-порт контроллера	COM3
Адрес модуля	1
Скорость обмена	115200
Количество бит данных	8
Контроль четности	Отсутствует
Количество стоп-бит	1

Пример создан в среде **CODESYS 3.5 SP11 Patch 5** и подразумевает запуск на **СПК1xx [M01]** с таргет-файлом **3.5.11.x**. В случае необходимости запуска проекта на другом устройстве следует изменить таргет-файл в проекте (**ПКМ** на узел **Device** – **Обновить устройство**).

Пример доступен для скачивания: [Example MV110 8A DCON.zip](#)

Листинг ROU примера приведен в [приложении А](#).

4.2 Описание протокола

Протокол [DCON](#) является одним из простейших строковых протоколов обмена. Он основан на архитектуре «Master-Slave» и реализуется поверх физического интерфейса RS-485. На примере протокола DCON будут описаны принципы разработки, с помощью которых можно реализовать любой строковый протокол обмена.

Структура кадра в общем виде выглядит следующим образом:

Символ начала сообщения	Адрес	Команда	Данные	Контрольная сумма	Символ конца сообщения
1 байт	2 байта	1...5 байт	1...256 байт	2 байта	1 байт

В зависимости от конкретного устройства количество используемых полей кадра может быть различным.

В руководстве по эксплуатации на модуль MB110-8A приведена информация по реализации протокола для данного устройства. Модуль поддерживает всего две команды: единичный и групповой запрос результатов измерений каналов модуля. В рамках данного примера будет реализован групповой запрос.

Соответствующая команда выглядит следующим образом:

#AA[CHK](cr)

где # – символ начала сообщения;

AA – адрес модуля (в HEX);

[CHK] – контрольная сумма (в HEX);

(cr) – символ конца сообщения (\$R).

Ответ от модуля будет иметь следующий вид:

>(data)[CHK](cr)

где **>** – символ начала сообщения;

(data) – записанные подряд без пробелов результаты всех 8 измерений в десятичном представлении. Длина каждой записи об одном измерении равна семи символам (знак, десятичная точка и 5 цифр). Положение десятичной точки модуль выбирает автоматически в зависимости от измеренного значения. В случае возникновения в измерительном канале исключительной ситуации возвращается значение **-99999** или **+99999**. Диагностики для определения типа исключительной ситуации не производится.

[CHK] – контрольная сумма (в HEX);

(cr) – символ конца сообщения (\$R).

В случае получения синтаксически неверного запроса или несоответствии контрольной суммы модуль не отвечает.

Контрольная сумма представляет собой сумму значений кодов всех ASCII символов команды, исключая символы самой контрольной суммы. Если ее значение превышает **16#FF**, то используется только младший байт.

Пример запроса к модулю с адресом **1**:

#01[CHK]\$R

Адрес модуля записывается в виде двух HEX символов. Для модуля с адресом **74** запрос бы выглядел как **#4A[CHK]\$R**.

Контрольная сумма для модуля с адресом 1 представляет собой сумму ASCII кодов символов запроса, расположенных до контрольной суммы: #, 0 и 1. Узнав коды символов (непосредственно из [таблицы ASCII](#) или, например, воспользовавшись одним из [онлайн-конвертеров](#)), можно вычислить контрольную сумму. В данном случае, символ «#» имеет код 23h, символ «0» – 30h, символ «1» – 31h, и, соответственно, $CHK = 23h + 30h + 31h = 84h$.

Таким образом, запрос к модулю с адресом 1 с рассчитанной контрольной суммой будет иметь вид:

#0184\$R

Пример ответа от модуля выглядит следующим образом:

>+100.23+34.050+124.56+07.331-101.45+1038.9-50.501+05.880[CHK]\$R

4.2 Алгоритмизация задачи

Процесс обмена данными через последовательный порт можно представить в виде циклически выполняемого алгоритма (см. п. 2.1). На языке ST для реализации подобных алгоритмов в большинстве случаев используется оператор условного выбора CASE.

Алгоритм решаемой в примере задачи можно представить следующим образом:

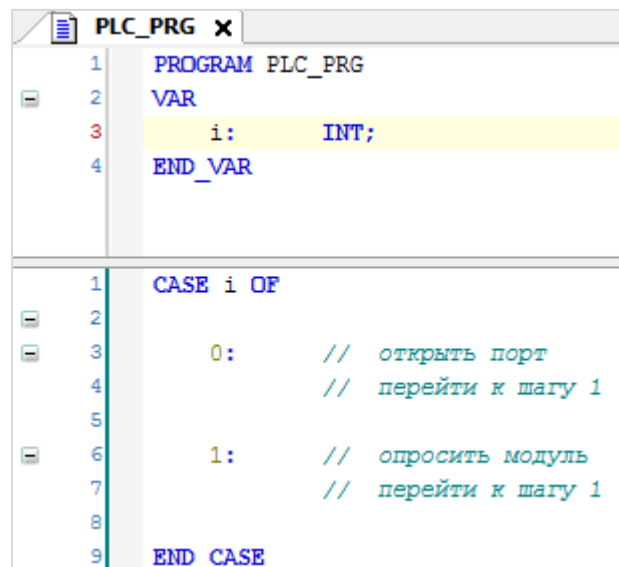


Рисунок 4.1 – Алгоритмизация задачи опроса модуля

Во время запуска программы будет однократно произведено открытие порта, после чего начнется циклический опрос модуля.

Соответственно, задача сводится к написанию кода, который будет выполняться в шагах 0 и 1. Рекомендуется упаковать его в два функциональных блока:

1. ФБ управления портом, который будет вызываться на шаге 0.
2. ФБ опроса модуля, который будет вызываться на шаге 1.

Предварительно следует добавить в проект библиотеку [CAA SerialCom](#), так как ее функции и ФБ потребуются при реализации протокола.

4.3 ФБ управления портом (COM_CONTROL)

Функциональный блок управления COM-портом с названием **COM_CONTROL** используется для открытия порта (для начала обмена) и его закрытия в случае возникновения ошибок или необходимости остановить обмен.

В качестве примера рассмотрим типичный функционал блока, чтобы определиться с набором входных и выходных переменных:

1. ФБ должен уметь открывать порт с заданными настройками по логическому сигналу.
2. ФБ должен уметь закрывать порт по логическому сигналу.
3. ФБ должен содержать выход, который сигнализирует о том, что порт успешно открыт.
4. ФБ должен содержать выход, который получает номер дескриптора (handle) открытого порта.
5. ФБ должен содержать выходы, сигнализирующие о возникновении ошибок при открытии и закрытии порта.

Получается следующий список переменных ФБ:

```

1  FUNCTION_BLOCK COM_CONTROL
2
3  VAR_IN_OUT
4      xOpen:          BOOL;          // сигнал открытия порта
5      xClose:         BOOL;          // сигнал закрытия порта
6  END_VAR
7
8  VAR_INPUT
9      uiPortNumber:    UINT;          // номер порта
10     udiBaudrate:      UDINT;         // скорость передачи данных
11     uiParity:         COM.PARITY;    // четность
12     uiByteSize:      UINT;          // кол-во бит данных в байте
13     uiStopBits:      COM.STOPBIT;   // кол-во стоп бит
14 END_VAR
15
16 VAR_OUTPUT
17     xDone:           BOOL;          // флаг успешного открытия порта
18     hCom:            COM.CAA.HANDLE; // дескриптор порта
19     xOpenError:      BOOL;          // флаг ошибки открытия порта
20     xCloseError:     BOOL;          // флаг ошибки закрытия порта
21 END_VAR
22
23 VAR
24     aComParams:      ARRAY [1..5] OF COM.PARAMETER; // структура настроек порта
25     fb_COMopen:      COM.Open;       // ФБ открытия порта
26     fb_COMclose:     COM.Close;      // ФБ закрытия порта
27     eState:          COM_STATE;      // текущий шаг работы с портом
28 END_VAR
29

```

Рисунок 4.2 – Объявление переменных ФБ COM_CONTROL



ПРИМЕЧАНИЕ

Переменные **xOpen** и **xClose** объявлены как **VAR_IN_OUT** – это позволяет менять их значения из ФБ. Таким образом, можно автоматически сбрасывать входные сигналы открытия и закрытия порта, чтобы предотвратить циклическое выполнение данных операций.

К локальным переменным ФБ относится структура параметров COM-порта и экземпляры ФБ [COM.Open](#) и [COM.Close](#), входящие в состав библиотеки [CAA SerialCom](#).

Как и сам алгоритм опроса, работу с COM-портом рекомендуется реализовывать через оператор CASE:

```

1  CASE eState OF
2
3
4      0:      // шаг инициализации блока
5
6      1:      // шаг ожидания управляющего сигнала
7
8      2:      // шаг открытия порта
9
10     3:      // шаг закрытия порта
11
12
13 END_CASE

```

Рисунок 4.3 – Алгоритм ФБ управления COM-портом

4. Пример опроса модуля MB110-8A по протоколу DCON

На шаге 0 будет происходить инициализация ФБ (завершение операций открытия/закрытия порта, завершенных при предыдущем вызове), на шаге 1 – ожидание управляющего сигнала. В зависимости от типа сигнала (открытие или закрытие порта), будет осуществлен переход к шагу 2 или 3. После завершения операции должно произойти возвращение к шагу 0.

На рисунке 4.3 для обозначения шагов используются порядковые номера. Данный подход лишен наглядности и затрудняет понимание алгоритма. Поэтому следует создать **перечисление COM_STATE**, которое позволит использовать имена для обозначения шагов (**Application – Добавление объекта – DUT – Перечисление**):

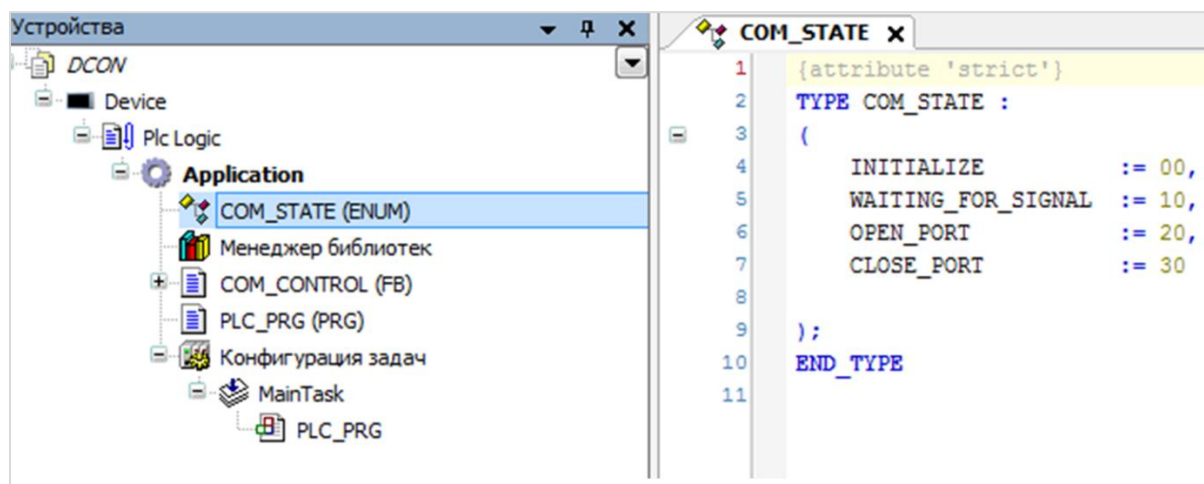


Рисунок 4.4 – Объявление перечисления COM_STATE

Для работы с перечислением следует в переменных программы объявить его экземпляр. В примере экземпляр уже был объявлен (см. переменную **eState** на рисунке 4.2). Разрывы в номерах шагов позволяют в случае необходимости добавлять промежуточные шаги.

Теперь в операторе CASE можно использовать имена шагов, что явно повышает читабельность кода. Подобный прием будет использоваться и в процессе разработки ФБ опроса модуля.

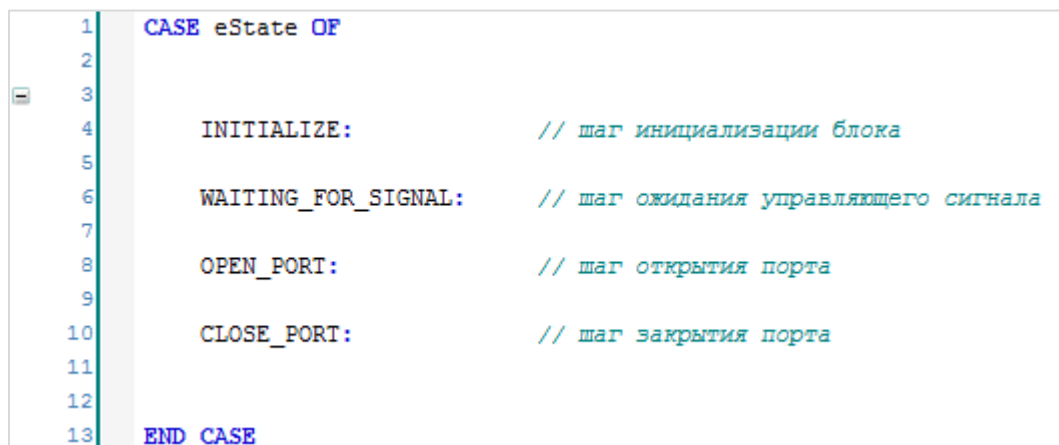


Рисунок 4.5 – Алгоритм ФБ управления COM-портом с использованием перечисления

4.3.1 Инициализация порта (шаг INITIALIZE)

На шаге **INITIALIZE** происходит сброс ФБ открытия и закрытия порта, после чего осуществляется переход к шагу ожидания управляющего сигнала ([WAITING_FOR_SIGNAL](#)):

```

4      INITIALIZE:                // шаг инициализации блока
5
6      fb_COMopen (xExecute:=FALSE); // сброс ФБ управления портом
7      fb_COMclose (xExecute:=FALSE);
8
9      eState:=WAITING_FOR_SIGNAL; // переход к шагу ожидания управляющего сигнала

```

Рисунок 4.6 – Код шага INITIALIZE

4.3.2 Ожидание управляющего сигнала (шаг WAITING_FOR_SIGNAL)

На шаге **WAITING_FOR_SIGNAL** будет происходить ожидание управляющего сигнала. В случае детектирования сигнала происходит переход к соответствующему шагу ([OPEN_PORT](#) или [CLOSE_PORT](#)):

```

12     WAITING_FOR_SIGNAL:        // шаг ожидания управляющего сигнала
13
14     // если получен сигнал открытия порта...
15     IF xOpen THEN
16         IF hCom=0 OR hCom=16#FFFFFF THEN // если порт еще не открыт (0) или (16#FFFFFF)...
17             eState := OPEN_PORT;          // ...то переходим к шагу открытия порта
18         ELSE                             // в противном случае считаем, что порт уже открыт, и тогда...
19             xOpen := FALSE;                // сбрасываем сигнал открытия порта
20             xDone := TRUE;                 // выставляем флаг успешного открытия порта
21             eState := INITIALIZE;          // ...переходим к шагу инициализации блока
22         END_IF
23     END_IF
24
25     // если получен сигнал закрытия порта...
26     IF xClose THEN
27         IF hCom>0 AND hCom<16#FFFFFF THEN // если порт уже открыт...
28             eState := CLOSE_PORT;          // ...то переходим к шагу закрытия порта
29         ELSE                             // в противном случае считаем, что порт уже закрыт, и тогда...
30             xClose := FALSE;               // сбрасываем сигнал закрытия порта
31             xDone := FALSE;                // сбрасываем флаг открытого порта
32             eState := INITIALIZE;          // ...переходим к шагу инициализации порта
33         END_IF
34     END_IF
35

```

Рисунок 4.7 – Код шага WAITING_FOR_SIGNAL

При получении управляющего сигнала производится проверка дескриптора (handle) **hCom**. Значение **0** соответствует закрытому порту, значение **16#FFFFFF** – ошибке открытия порта. Значения между **0** и **16#FFFFFF** соответствуют открытому порту. Соответственно, не имеет смысла открывать уже открытый порт и закрывать закрытый – в данном случае следует сбросить управляющий сигнал, обновить значение выхода **xDone** и перейти на шаг инициализации ФБ.

Если порт закрыт и получен сигнал открытия порта, то происходит переход к шагу [OPEN_PORT](#). Если порт открыт и получен сигнал закрытия порта – к шагу [CLOSE_PORT](#).

4. Пример опроса модуля MB110-8A по протоколу DCON

4.3.3 Открытие порта (шаг OPEN_PORT)

На шаге **OPEN_PORT** следует открыть порт с заданными параметрами (параметры являются входными переменными ФБ **COM_CONTROL**).

```
37
38 OPEN_PORT:           // шаг открытия порта
39
40 SETTINGS();          // задаем настройки порта
41
42 OPEN();              // открываем порт
43
44 IF fb_COMopen.xDone AND fb_COMopen.xExecute THEN // если блок открытия порта завершил работу...
45     xOpen := FALSE; // сбрасываем сигнал открытия порта
46     fb_COMopen(xExecute:=FALSE); // сбрасываем фб открытия порта
47     xDone := TRUE; // выставляем флаг успешного открытия порта
48     eState := INITIALIZE; // переходим к шагу инициализации порта
49 END_IF
50
51 IF fb_COMopen.xError AND fb_COMopen.xExecute THEN // если во время открытия порта возникли ошибки...
52     xOpenError := TRUE; // выставляем флаг ошибки
53     fb_COMopen(xExecute:=FALSE); // сбрасываем фб открытия порта
54     xDone := FALSE; // сбрасываем флаг открытия порта
55     eState := INITIALIZE; // переходим к шагу инициализации порта
56 END_IF
```

Рисунок 4.8 – Код шага OPEN_PORT

На этом шаге используется еще один прием структурирования программы – выделение законченных фрагментов кода в **действия**. Действия являются вложенными **POU**, для создания которых необходимо нажать **ПКМ** на основной **POU** и выбрать команду **Добавление объекта – Действие**. В примере для ФБ **COM_CONTROL** будут созданы три действия – **SETTINGS** (задание настроек порта), **OPEN** (открытие порта) и **CLOSE** (закрытие порта).

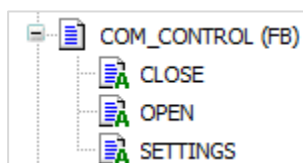


Рисунок 4.9 – Действия ФБ COM_CONTROL

Код действия **SETTINGS** выглядит следующим образом:

```
COM_CONTROL.SETTINGS x
1
2 // настройки COM-порта
3
4 aComParams[1].udiParameterId := COM.CAA_Parameter_Constants.udiPort; // номер
5 aComParams[1].udiValue := uiPortNumber;
6 aComParams[2].udiParameterId := COM.CAA_Parameter_Constants.udiBaudrate; // скорость
7 aComParams[2].udiValue := udiBaudrate;
8 aComParams[3].udiParameterId := COM.CAA_Parameter_Constants.udiParity; // четность
9 aComParams[3].udiValue := ANY_TO_UDINT(uiParity);
10 aComParams[4].udiParameterId := COM.CAA_Parameter_Constants.udiByteSize; // кол-во бит данных в байте
11 aComParams[4].udiValue := uiByteSize;
12 aComParams[5].udiParameterId := COM.CAA_Parameter_Constants.udiStopBits; // кол-во стоп бит
13 aComParams[5].udiValue := ANY_TO_UDINT(uiStopBits);
```

Рисунок 4.10 – Код действия SETTINGS

Массив **aComParams** содержит настройки COM-порта. Каждый его элемент представляет собой структуру типа [COM.PARAMETER](#), содержащую две переменных – имя параметра (**udiParameterId**) и его значение (**udiValue**). Имена параметров определены в списке глобальных констант **CAA_Parameter_Constant**. Их значения задаются на входе ФБ **COM_CONTROL**.

Параметры **uiParity** и **uiStopBits** представляют собой перечисления (**COM.Parity** и **COM.STOPBIT** соответственно). Чтобы избежать предупреждений компилятора, их следует преобразовать к нужному типу (т. е. к типу переменной **udiValue** – **UDINT**). Узнать фактический тип перечислений можно, посмотрев их описание в библиотеке, но проще будет воспользоваться функцией **ANY_TO_UDINT**, которая подходит для конверсии любого типа.

Код действия **OPEN** выглядит следующим образом:

```

1 // открываем порт
2
3 fb_COMopen.usiListLength := UINT_TO_USINT(SIZEOF(aComParams) / SIZEOF(COM.PARAMETER));
4 fb_COMopen.pParameterList := ADR(aComParams);
5 fb_COMopen.xExecute := TRUE;
6
7 fb_COMopen();
8
9 // получаем дескриптор
10 hCom := fb_COMopen.hCom;

```

Рисунок 4.11 – Код действия OPEN

В данном действии производится работа с экземпляром ФБ [COM.Open](#).

Сначала задаются входные параметры блока.

Вход **usiListLength** содержит число используемых параметров COM-порта. Разумеется, их можно задать и в явном виде (в рамках нашего примера **usiListLength := 5**), но тогда в случае необходимости изменить число параметров придется менять и данное значение. Этого можно избежать с помощью оператора **SIZEOF**, который возвращает размер переменной в байтах. В результате деления размера массива параметров на размер одного параметра будет получено текущее число параметров. При изменении размерности массива новое значение будет рассчитано автоматически.

Вход **pParameterList** содержит адрес структуры параметров COM-порта.

Вход **xExecute** используется для управления блоком.

После установки входных параметров необходимо вызвать ФБ, что приведет к открытию COM-порта с заданными настройками.

4. Пример опроса модуля MB110-8A по протоколу DCON

Затем следует получить значение дескриптора (handle), присвоив его выходной переменной **hCom** ФБ **COM_CONTROL**.

```
37
38 OPEN_PORT:           // шаг открытия порта
39
40 SETTINGS();          // задаем настройки порта
41
42 OPEN();              // открываем порт
43
44 IF fb_COMopen.xDone AND fb_COMopen.xExecute THEN // если блок открытия порта завершил работу...
45     xOpen := FALSE; // сбрасываем сигнал открытия порта
46     fb_COMopen(xExecute:=FALSE); // сбрасываем фБ открытия порта
47     xDone := TRUE; // выставляем флаг успешного открытия порта
48     eState := INITIALIZE; // переходим к шагу инициализации порта
49 END_IF
50
51 IF fb_COMopen.xError AND fb_COMopen.xExecute THEN // если во время открытия порта возникли ошибки...
52     xOpenError := TRUE; // выставляем флаг ошибки
53     fb_COMopen(xExecute:=FALSE); // сбрасываем фБ открытия порта
54     xDone := FALSE; // сбрасываем флаг открытия порта
55     eState := INITIALIZE; // переходим к шагу инициализации порта
56 END_IF
```

Рисунок 4.12 – Код шага OPEN_PORT

После выполнения действия **OPEN** следует проанализировать его результат. Если порт успешно открыт (у **fb_COMopen** выход **xDone = TRUE**), то происходит сброс управляющего сигнала, остановка работы **fb_COMopen**, взведение флага успешного открытия порта (выход **xDone** ФБ **COM_CONTROL**) и переход на шаг инициализации порта (**INITIALIZE**).

Если при открытии порта произошла ошибка (у ФБ **fb_COMopen** выход **xError = TRUE**), то происходит взведение флага ошибки открытия порта, остановка работы **fb_COMopen**, сброс флага успешного открытия порта (выход **xDone** ФБ **COM_CONTROL**) и переход на шаг инициализации порта (**INITIALIZE**).

4.3.4 Заккрытие порта (шаг CLOSE_PORT)

В случае перехода на шаг **CLOSE_PORT** следует закрыть порт. Заккрытие порта может потребоваться в случае необходимости прекращения обмена или обработке ошибок обмена.

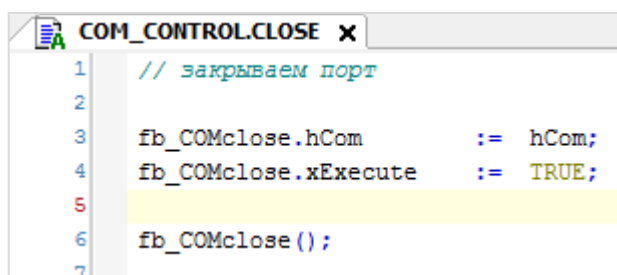
```

58
59      CLOSE_PORT:           // шаг закрытия порта
60
61      CLOSE();             // закрываем порт
62
63      IF fb_COMclose.xDone AND fb_COMclose.xExecute THEN // если блок закрытия порта завершил работу...
64          xClose := FALSE; // сбрасываем сигнал закрытия порта
65          xDone := FALSE;  // сбрасываем флаг открытия порта
66          hCom := 0;       // обнуляем хэндл
67          fb_COMclose(xExecute:=FALSE); // сбрасываем ФБ закрытия порта
68          eState := INITIALIZE; // переходим к шагу инициализации порта
69      END_IF
70
71      IF fb_COMclose.xError AND fb_COMclose.xExecute THEN // если во время закрытия порта возникли ошибки...
72          xCloseError := TRUE; // выставляем флаг ошибки
73          fb_COMclose(xExecute:=FALSE); // сбрасываем ФБ закрытия порта
74          // остаемся на шаге закрытия порта
75      END_IF

```

Рисунок 4.13 – Код шага CLOSE_PORT

Как и при открытии порта, код закрытия следует поместить в действие.



```

1 // закрываем порт
2
3 fb_COMclose.hCom := hCom;
4 fb_COMclose.xExecute := TRUE;
5
6 fb_COMclose();
7

```

Рисунок 4.14 – Код действия CLOSE

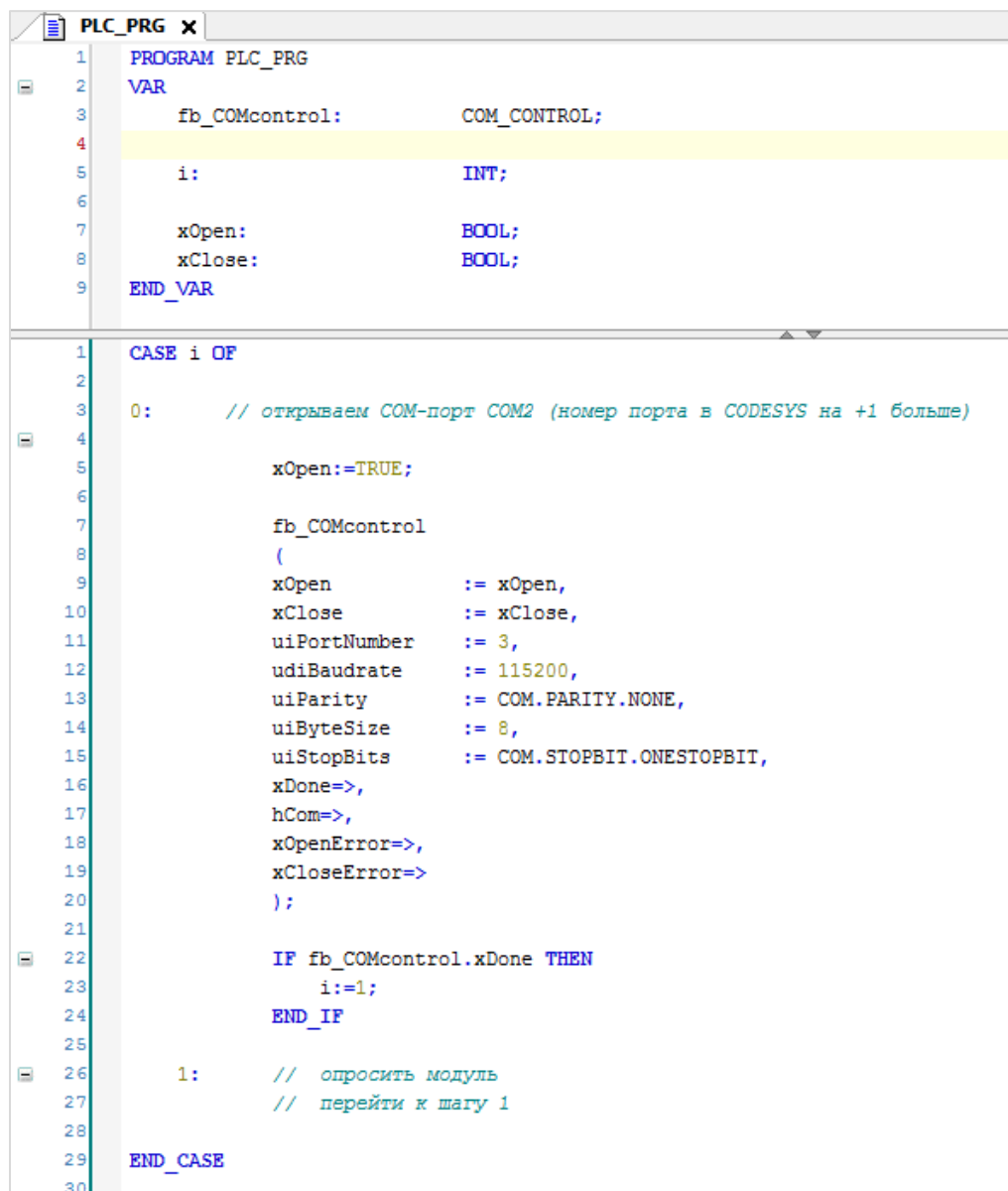
Для закрытия порта достаточно указать его дескриптор (handle) **hCom** и вызвать **fb_COMclose** с **xExecute = TRUE**.

После выполнения действия **CLOSE** следует проанализировать его результат. Если порт успешно открыт (у **fb_COMclose** выход **xDone = TRUE**), то происходит сброс управляющего сигнала, сброс флага успешного открытия порта, обнуление значение дескриптора (handle) на выходе ФБ **COM_CONTROL**, остановка работы **fb_COMclose** и переход на шаг инициализации порта.

Если при закрытии порта произошла ошибка (у ФБ **fb_COMclose** выход **xError = TRUE**), то происходит взведение флага ошибки закрытия порта и остановка работы **fb_COMclose**. Программа остается на том же шаге выполнения, т. е. в следующем цикле опять будет произведена попытка закрытия порта и так до тех пор, пока не произойдет успешное закрытие порта.

4. Пример опроса модуля MB110-8A по протоколу DCON

Теперь следует вернуться к основной программе [PLC_PRG](#), объявить экземпляр ФБ **COM_CONTROL** и написать код для шага 0:



```
1 PROGRAM PLC_PRG
2 VAR
3     fb_COMcontrol:      COM_CONTROL;
4
5     i:                  INT;
6
7     xOpen:              BOOL;
8     xClose:             BOOL;
9 END_VAR
10
11 CASE i OF
12 0: // открываем COM-порт COM2 (номер порта в CODESYS на +1 больше)
13
14     xOpen:=TRUE;
15
16     fb_COMcontrol
17     (
18         xOpen      := xOpen,
19         xClose     := xClose,
20         uiPortNumber := 3,
21         udiBaudrate := 115200,
22         uiParity    := COM.PARITY.NONE,
23         uiByteSize  := 8,
24         uiStopBits  := COM.STOPBIT.ONESTOPBIT,
25         xDone=>,
26         hCom=>,
27         xOpenError=>,
28         xCloseError=>
29     );
30
31     IF fb_COMcontrol.xDone THEN
32         i:=1;
33     END_IF
34
35 1: // опросить модуль
36    // перейти к шагу 1
37
38 END_CASE
```

Рисунок 4.15 – Алгоритмизация задачи опроса модуля с кодом шага 0

В рамках примера не будет рассматриваться закрытие порта, поэтому следует в явном виде присвоить переменной **xOpen** значение **TRUE**. Во время запуска программы шаг 0 будет выполнен однократно, поскольку возвращение на него не происходит.

Порт с заданными настройками открыт – теперь можно отправлять в него данные и считывать ответы. Теперь следует приступить к реализации протокола DCON.

Листинг ФБ **COM_CONTROL** приведен в [приложении А.1](#).

4.4 ФБ опроса модуля (MV110_8A_DCON)

Для опроса модуля **MB110-8A** по протоколу **DCON** следует создать ФБ с названием **MV110_8A_DCON**. В рамках примера его входными переменными будут являться:

1. Вход управления блоком (старт/прекращение работы).
2. Адрес опрашиваемого устройства.
3. Дескриптор (handle) COM-порта.

К выходным переменным будут относиться:

1. Измеренные значения каналов модуля.
2. Флаг окончания цикла опроса.
3. Флаги ошибок обмена.

Ниже также приводится список локальных переменных ФБ. Необходимость их объявления будет поясняться по мере описания принципов работы блока.

```

1  FUNCTION_BLOCK MV110_8A_DCON
2
3  VAR_INPUT
4      xEnable:          BOOL;           // сигнал опроса модуля
5      hCom:             COM.CAA.HANDLE; // дескриптор COM-порта
6      byAddress:        BYTE;          // адрес опрашиваемого модуля
7  END_VAR
8
9  VAR_OUTPUT
10     xDone:             BOOL;           // флаг окончания текущего цикла опроса модуля
11     xWriteError:       BOOL;           // флаг ошибки отправки запроса
12     xReadError:        BOOL;           // флаг ошибки получения ответа
13     xTimeout:          BOOL;           // флаг отсутствия ответа по истечению таймута опроса
14     xWrongCRC:         BOOL;           // флаг получения ответа с неправильной контрольной суммой
15
16     rMV110_8A_output1: REAL;           // считанные значения каналов модуля
17     rMV110_8A_output2: REAL;
18     rMV110_8A_output3: REAL;
19     rMV110_8A_output4: REAL;
20     rMV110_8A_output5: REAL;
21     rMV110_8A_output6: REAL;
22     rMV110_8A_output7: REAL;
23     rMV110_8A_output8: REAL;
24 END_VAR
25
26 VAR
27     eState:            DCON_state;     // текущий шаг опроса модуля
28
29     fb_COMwrite:        COM.Write;      // ФБ отправки запроса
30     fb_COMread:         COM.Read;       // ФБ получения ответа
31
32     sWriteData:         STRING(255);    // запрос к модулю в виде строки
33     sReadData:          STRING(255);    // полный ответ модуля в виде строки
34     sReadBuff:          STRING(255);    // часть ответа модуля
35
36     sAddress:           STRING(2);      // адрес опрашиваемого модуля в виде строки
37     byCRC:              BYTE;           // CRC
38     sCRC:               STRING(2);      // CRC в виде строки
39     xCorrectAnswer:     BOOL;           // флаг корректного (с верной CRC) ответа от модуля
40     i:                  INT;            // счетчик для цикла FOR
41
42     fb_TON:             TON;            // таймер
43     fb_ANALYZE_DATA:    ANALYZE_DATA;   // ФБ анализа ответа модуля
44 END_VAR
45
46 VAR CONSTANT
47     c_timeout:          TIME:=T#1S;     // таймат опроса модуля (время ожидания ответа)
48     c_Delay:            TIME:=T#50MS;    // задержка перед отправкой следующего запроса
49 END_VAR
50

```

Рисунок 4.16 – Объявление переменных ФБ MV110_8A_DCON

Для реализации блока будет использоваться пошаговый алгоритм с оператором CASE и **перечислением**, которое позволит использовать имена в качестве названий шагов. Экземпляр перечисления имеет название **eState**.

4. Пример опроса модуля MB110-8A по протоколу DCON

```
DCON_STATE x
1 {attribute 'strict'}
2 TYPE DCON_STATE :
3 (
4     CREATE_REQUEST      := 00,
5     SEND_REQUEST        := 10,
6     RECEIVE_RESPONSE    := 20,
7     RESPONSE_DELAY      := 30,
8     POLLING_CYCLE_ENDS  := 40
9 );
10 END_TYPE
```

Рисунок 4.17 – Объявление перечисления DCON_STATE

Алгоритм опроса модуля в общем виде:

```
1
2 IF xEnable THEN           // если получен сигнал опроса модуля...
3
4     CASE eState OF        // ...то запускаем циклический опрос
5
6         CREATE_REQUEST:   // шаг подготовки запроса
7
8         SEND_REQUEST:     // шаг отправки запроса к модулю
9
10        RECEIVE_RESPONSE: // шаг получения ответа от модуля
11
12        RESPONSE_DELAY:   // шаг задержки перед следующим циклом опроса
13
14        POLLING_CYCLE_ENDS: // шаг окончания цикла опроса
15
16    END_CASE
17
18 END_IF
```

Рисунок 4.18 – Алгоритм опроса модуля

На шаге **CREATE_REQUEST** будет происходить формирование запроса к модулю.

На шаге **SEND_REQUEST** запрос будет отправлен в COM-порт.

На шаге **RECEIVE_RESPONSE** будет происходить получение и анализ ответа от модуля.

Шаг **RESPONSE_DELAY** будет использоваться для создания задержки между получением ответа от модуля и отправки следующего запроса.

На шаге **POLLING_CYCLE_ENDS** будет происходить формирование значений выходов ФБ, после чего следует переход к начальному шагу.



ПРИМЕЧАНИЕ

Необходимость наличия шага **RESPONSE_DELAY** зависит от особенностей конкретного опрашиваемого устройства. Некоторые устройства удерживают линию связи фиксированное время после ответа, в связи с чем следует организовывать задержку перед отправкой следующего запроса. В случае опроса модуля MB110-8A такая задержка не обязательна, но рассматривается в настоящем руководстве в качестве примера.

Законченные фрагменты кода ФБ будут оформлены в виде **действий**:

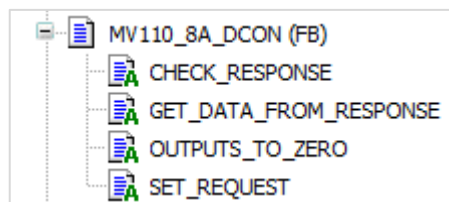


Рисунок 4.19 – Действия ФБ MV110_8A_DCON

4.4.1 Подготовка запроса (шаг CREATE_REQUEST)

На шаге **CREATE_REQUEST** происходит очистка буферов данных и подготовка запроса.

```

5  CREATE_REQUEST: // шаг подготовки запроса
6
7
8
9  _BUFFER_CLEAR(PT:=ADR(sReadBuff), SIZE:=SIZEOF(sReadBuff)); // очищаем буфер приема
10 _BUFFER_CLEAR(PT:=ADR(sReadData), SIZE:=SIZEOF(sReadData)); // очищаем буфер данных
11 _BUFFER_CLEAR(PT:=ADR(sWriteData), SIZE:=SIZEOF(sWriteData)); // очищаем буфер запроса
12
13 SET_REQUEST(); // подготовка запроса
14
15 xDone := FALSE; // сброс флага окончания предыдущего цикла опроса
16
17 eState := SEND_REQUEST; // переходим к шагу отправки запроса
18

```

Рисунок 4.20 – Код шага CREATE_REQUEST

Под буфером понимается переменная, используемая для временного хранения данных. В данном случае такими переменными являются **sReadBuff** (буфер фрагмента ответа), **sReadData** (буфер полного ответа) и **sWriteData** (буфер запроса). В рамках примера модуля каждый раз отправляется один и тот же запрос, поэтому, в целом, нет реальной необходимости каждый раз очищать буфер запроса и формировать запрос заново – но в других ситуациях это может понадобиться (например, когда необходимо с помощью одного ФБ считывать несколько параметров устройства с помощью разных запросов).

Для очистки буфера используется функция **_BUFFER_CLEAR** из библиотеки **OSCAT**. Библиотека доступна для скачивания на сайте oscat.de и [сайте компании Овен](#) в разделе **CODESYS V3/Библиотеки**. Библиотека OSCAT имеет открытые исходные коды, поэтому во многих случаях удобнее копировать ее функции и ФБ в пользовательский проект (вместо добавления через **Менеджер библиотек**).

Функция **_BUFFER_CLEAR** имеет два входных параметра: адрес буфера и его размер. Вместо задания размера буфера с помощью числа удобнее использовать оператор **SIZEOF**, возвращающий размер переменной в байтах. Возвращаемое значение имеет тип **BOOL** и принимает значение **TRUE** в случае успешного завершения очистки буфера.

4. Пример опроса модуля MB110-8A по протоколу DCON

```
_BUFFER_CLEAR x
1  FUNCTION _BUFFER_CLEAR : BOOL
2  VAR_INPUT
3      PT : POINTER TO BYTE;
4      SIZE : UINT;
5  END_VAR
6  VAR
7      ptw : POINTER TO DWORD;
8      temp: DWORD;
9      end, end32 : DWORD;
10 END_VAR

1  (* this routine uses 32 bit access to gain speed *)
2  (* first access bytes till pointer is aligned for 32 bit access *)
3  temp := pt;
4  end := temp + UINT_TO_DWORD(size);
5  end32 := end - 3;
6  WHILE (pt < end) AND ((temp AND 16#00000003) > 0) DO
7      pt^ := 0;
8      pt := pt + 1;
9      temp := temp + 1;
10 END_WHILE;
11 (* pointer is aligned, now copy 32 bits at a time *)
12 ptw := pt;
13 WHILE ptw < end32 DO (*)
14     ptw^ := 0;
15     ptw := ptw + 4;
16 END_WHILE;
17 (* copy the remaining bytes in byte mode *)
18 pt := ptw;
19 WHILE pt < end DO
20     pt^ := 0;
21     pt := pt + 1;
22 END_WHILE;
23
24 _BUFFER_CLEAR := TRUE;
```

Рисунок 4.21 – Код функции **_BUFFER_CLEAR** из библиотеки **OSCAT**

Код функции скопирован из библиотеки **OSCAT** без каких-либо изменений.

Действие **SET_REQUEST** используется для формирования запроса к модулю. Структура запроса известна из описания протокола **DCON**.



ПРИМЕЧАНИЕ

DCON является строковым протоколом – т. е. все данные в нем передаются в виде ASCII-кодов соответствующих символов.

```

1 // формирование запроса к модулю MB110-8A
2
3
4 sWriteData[0]:=16#23; // ASCII: #
5
6 sAdress:=BYTE_TO_STRH(byAdress); // ASCII: адрес модуля (два HEX символа)
7
8 sWriteData[1]:=sAdress[0]; //
9 sWriteData[2]:=sAdress[1]; //
10
11 byCRC:=sWriteData[0]+sWriteData[1]+sWriteData[2]; // расчет CRC
12 sCRC:=BYTE_TO_STRH(byCRC); // ASCII: CRC (два HEX символа)
13
14 sWriteData[3]:=sCRC[0]; //
15 sWriteData[4]:=sCRC[1]; //
16
17 sWriteData[5]:=16#D; // ASCII: CR (перевод каретки)
18

```

Рисунок 4.22 – Код действия SET_REQUEST

В **CODESYS V3.5** со строковыми переменными можно работать как с массивом байт, что является очень удобным при наполнении буфера.

Структура запроса к модулю выглядит следующим образом:

Таблица 4.2 – Структура запроса по протоколу DCON

	Старт символ	Адрес модуля	Контрольная сумма	Стоп символ
Размер	1 байт	2 байта	2 байта	1 байт
Значение	#	Переменная	Переменная	CR
Код ASCII	16#23	Переменная	Переменная	16#D

Входная переменная **byAdress** имеет тип **BYTE** – поэтому ее следует конвертировать в два HEX символа с помощью **BYTE_TO_STRH** из библиотеки **OSCAT** (см. рисунок 4.23). Код функции скопирован из библиотеки **OSCAT** без каких-либо изменений.

Методика расчета контрольной суммы известна из описания протокола – она, действительно, представляет собой сумму всех элементов буфера, расположенных до контрольной суммы (т. е. старт-символ и адрес модуля). В данном случае контрольная сумма вычисляется в явном виде, но в случае необходимости (например, в случае опроса нескольких различных модулей) ее расчет можно вынести в отдельную функцию.

4. Пример опроса модуля MB110-8A по протоколу DCON

```
1 FUNCTION BYTE_TO_STRH : STRING(2)
2   VAR_INPUT
3     IN : BYTE;
4   END_VAR
5   VAR
6     temp : BYTE;
7     PT : POINTER TO BYTE;
8   END_VAR
9
10  (*
11  version 1.3 29 mar. 2008
12  programmer hugo
13  tested by tobias
14  *)
15
16  (* read pointer to output string *)
17  PT := ADR(BYTE_TO_STRH);
18  (* calculate high order hex value *)
19  temp := SHR(in,4);
20  (* convert value to hex character *)
21  IF temp <= 9 THEN temp := temp + 48; ELSE temp := temp + 55; END_IF;
22  (* write first character to output string *)
23  PT^ := temp;
24  (* calculate low order hex value *)
25  temp := in AND 2#00001111;
26  IF temp <= 9 THEN temp := temp + 48; ELSE temp := temp + 55; END_IF;
27  (* increment pointer and write low order character *)
28  pt := pt + 1;
29  pt^ := temp;
30  (* set pointer at the end of the output string and close the string with 0 *)
31  pt := pt + 1;
32  pt^ := 0;
```

Рисунок 4.23 – Код функции BYTE_TO_STRH из библиотеки OSCAT

Теперь следует вернуться к коду шага **CREATE_REQUEST**.

Кроме очистки буферов и формирования запроса на данном шаге происходит сброс флага окончания предыдущего цикла, после чего осуществляется переход к шагу [SEND_REQUEST](#).

```
5
6   CREATE_REQUEST: // шаг подготовки запроса
7
8
9   _BUFFER_CLEAR(PT:=ADR(sReadBuff), SIZE:=SIZEOF(sReadBuff)); // очищаем буфер приема
10  _BUFFER_CLEAR(PT:=ADR(sReadData), SIZE:=SIZEOF(sReadData)); // очищаем буфер данных
11  _BUFFER_CLEAR(PT:=ADR(sWriteData), SIZE:=SIZEOF(sWriteData)); // очищаем буфер запроса
12
13  SET_REQUEST(); // подготовка запроса
14
15  xDone := FALSE; // сброс флага окончания предыдущего цикла опроса
16
17  eState := SEND_REQUEST; // переходим к шагу отправки запроса
18
```

Рисунок 4.24 – Код шага CREATE_REQUEST

4.4.2 Отправка запроса (шаг SEND_REQUEST)

На шаге **SEND_REQUEST** происходит отправка запроса, сформированного на предыдущем шаге.

```

19
20      SEND_REQUEST: // шаг отправки запроса к модулю
21
22
23      fb_COMwrite                                     // запускаем ФБ записи в порт
24      (
25      xExecute := TRUE,
26      hCom     := hCom,
27      pBuffer  := ADR(sWriteData),
28      szSize   := INT_TO_UINT(LEN(sWriteData))
29      );
30
31
32      IF fb_COMwrite.xError THEN                      // если при отправке запроса произошла ошибка...
33      xWriteError := TRUE;                          // выставляем флаг ошибки отправки запроса
34      eState      := POLLING_CYCLE_ENDS;            // переходим к шагу окончания опроса
35      END_IF
36
37
38      IF fb_COMwrite.xDone THEN                      // если отправка запроса успешно завершена...
39      fb_COMwrite(xExecute:=FALSE);                // сбрасываем ФБ записи в порт
40
41      xWriteError := FALSE;                        // сбрасываем флаг ошибки отправки запроса
42
43      fb_TON(IN:=FALSE, PT:=c_timeout);            // сбрасываем таймер
44      fb_TON(IN:=TRUE,  PT:=c_timeout);            // запускаем таймер таймаута
45
46      eState := RECEIVE_RESPONSE;                  // переходим к шагу получения ответа от модуля
47      END_IF
48

```

Рисунок 4.25 – Код шага SEND_REQUEST

В этом шаге производится работа с экземпляром ФБ [COM.Write](#).

Входными переменным блока являются сигнал его запуска (**xExecute**), дескриптор (handle) COM-порта (**hCom**), адрес буфера запроса (**pBuffer**) и размер буфера (**szSize**). В строковом протоколе размер запроса определяется числом его символов – поэтому следует использовать функцию **LEN**, которая возвращает длину строки.

После вызова блока отправки запроса следует проанализировать его результат. Если во время отправки произошла ошибка (у ФБ **fb_COMwrite** выход **xError = TRUE**), то следует взвести флаг ошибки и перейти к шагу завершения опроса.

Если запрос успешно отправлен (у **fb_COMwrite** выход **xDone = TRUE**), то следует завершить работу блока, сбросить флаг ошибки, запустить таймер таймаута (он потребуется на следующем шаге) и перейти к шагу [RECEIVE_RESPONSE](#).

4. Пример опроса модуля MB110-8A по протоколу DCON

4.4.3 Получение ответа (шаг RECEIVE_RESPONSE)

На шаге **RECEIVE_RESPONSE** происходит получение ответа от модуля.

```
50 RECEIVE_RESPONSE: // шаг получения ответа от модуля
51
52
53
54 fb_COMread // запускаем ФБ чтения из порта
55 (
56   xExecute := TRUE,
57   hCom     := hCom,
58   pBuffer  := ADDR(sReadBuff),
59   szBuffer := SIZEOF(sReadBuff),
60 );
61
62 fb_TON(); // вызываем таймер таймаута
63
64 IF fb_COMread.xError THEN // если при получении ответа произошла ошибка...
65   xReadError := TRUE; // выставляем флаг ошибки получения ответа
66   eState     := POLLING_CYCLE_ENDS; // переходим к шагу окончания опроса
67 END_IF
68
69 IF fb_COMread.xDone THEN // если ФБ чтения из порта завершил работу...
70   xTimeout:=FALSE; // сбрасываем флаг таймаута
71   CHECK_RESPONSE(); // ищем символы начала и конца ответа
72   fb_COMread(xExecute:=FALSE); // завершаем работу блока чтения из порта
73
74   IF xCorrectAnswer THEN // если в ответе есть и символ начала, и символ конца...
75     GET_DATA_FROM_RESPONSE(); // выделяем данные из ответа
76     eState := RESPONSE_DELAY; // переходим к шагу задержки перед следующим циклом опроса
77   END_IF
78 END_IF
79
80 IF fb_TON.Q THEN // если сработал таймер таймаута...
81   xTimeout := TRUE; // взводим флаг ошибки по таймауту
82   OUTPUTS_TO_ZERO(); // сбрасываем значения выходов блока
83   eState := RESPONSE_DELAY; // переходим к шагу задержки перед следующим циклом опроса
84 END_IF
85
```

Рисунок 4.26 – Код шага RECEIVE_RESPONSE

На данном шаге производится работа с экземпляром ФБ [COM.Read](#).

Входными переменными блока являются сигнал его запуска (**xExecute**), дескриптор (handle) COM-порта (**hCom**), адрес буфера фрагмента ответа (**pBuffer**) и размер буфера (**szSize**). Вместо задания размера буфера с помощью числа рекомендуется использовать оператор **SIZEOF**, возвращающий размер переменной в байтах. В данном случае функция **LEN** не требуется, т. к. размер буфера приема в большинстве случаев является статической величиной, зависящей только от размера соответствующей переменной.

В ряде случаев подчиненное устройство не ответит на запрос (например, в случае некорректного запроса или неисправности устройства). Если не предусмотреть эту ситуацию, то программа может «застрять» на шаге получения ответа. Для предотвращения данной ситуации следует использовать контроль таймаута опроса – по истечению заданного времени программа выходит из текущего шага вне зависимости от факта получения ответа. Таймер таймаута был запущен в конце предыдущего шага, поэтому на данном шаге достаточно вызвать его с теми же аргументами. Время таймаута определяется с помощью константы **c_tTimeout**.

После вызова блока получения ответа следует проанализировать его результат. Если при получении произошла ошибка (у **fb_COMread** выход **xError = TRUE**), то следует взвести флаг ошибки и перейти к шагу завершения опроса.

Если ответ успешно получен (у **fb_COMread** выход **xDone = TRUE**), то следует сбросить флаг ошибки таймаута, проверить корректность ответа и завершить работу блока.

Проверка корректности ответа вынесена в действие **CHECK_RESPONSE**:

4. Пример опроса модуля MB110-8A по протоколу DCON

```

MV110_8A_DCON.CHECK_RESPONSE x
1 IF fb_COMread.szSize>0 THEN // если из порта была считана порция данных...
2   sReadData := CONCAT(sReadData,sReadBuff); // ...то приклеиваем их к предыдущей порции...
3   // ...и остаемся на этом же шаге, чтобы считать следующую порцию данных
4
5   // ищем символы начала и конца ответа, вырезаем ответ
6
7   IF FIND(sReadData, '>') <> 0 AND FIND(sReadData, '$R') <>0 AND (FIND(sReadData, '$R') > FIND(sReadData, '>')) THEN
8     sReadData:=MID(sReadData, FIND(sReadData, '$R') - FIND(sReadData, '>') + 1, FIND(sReadData, '>'));
9     xCorrectAnswer:=TRUE; // взводим флаг получения корректного ответа
10  END_IF
11
12 END_IF

```

Рисунок 4.27 – Код действия CHECK_RESPONSE

В некоторых случаях (в частности, при чтении большого количества данных), ответ от подчиненного устройства может прийти в виде нескольких фрагментов. Корректный способ обработки данной ситуации: пока не сработал таймер таймаута, каждый полученный фрагмент ответа следует «приклеивать» к предыдущему фрагменту с помощью функции **CONCAT** в буфере полного ответа (**sReadData**), после чего искать в буфере старт-символ и стоп-символ с помощью функции **FIND**. Если оба символа обнаружены, и позиция стоп символа больше позиции старт-символа, то можно сделать вывод, что в буфере находится ответ от подчиненного устройства. В ряде специфических случаев ответ может находиться не в начале буфера, поэтому следует вырезать его, ориентируясь на позиции старт- и стоп-символа. Вырезанный ответ помещается в тот же самый буфер, после чего взводится флаг получения корректного ответа (**xCorrectAnswer**).

Вырезание из буфера полученного ответа (от старт-символа до стоп-символа)

...	Старт-символ	...	...	...	Стоп-символ	...
Старт- символ	...	...	...	Стоп-символ	...	...

Если один из символов не найден (например, пришел первый фрагмент ответа, в котором содержится старт-символ, но стоп-символа еще нет), то программа остается на данном шаге до истечения таймаута, что позволяет получить следующие фрагменты.

Затем следует вернуться к шагу **RECEIVE_RESPONSE** (см. [рисунок 4.26](#)). После выполнения кода действия **CHECK_RESPONSE** следует завершить работу **ФБ COM.Read**. Если взведен флаг корректного ответа (**xCorrectAnswer**), то следует перейти к действию **GET_DATA_FROM_RESPONSE**, в котором будет выполняться разбор полученного ответа.

4. Пример опроса модуля MB110-8A по протоколу DCON

```
MV110_8A_DCON.GET_DATA_FROM_RESPONSE x
1
2
3 // обнуляем CRC запроса
4 byCRC:=0;
5
6 // подсчитываем CRC ответа
7 FOR i:=0 TO FIND(sReadData, '$R') - 4 DO // -4 - потому что в расчете CRC не участвуют три последних...
8     byCRC:=byCRC+sReadData[i]; // ...символа ответа (два символа CRC и символ конца ответа)
9 // еще один символ - смещение (элементы массива - 0-59, а...
10 END_FOR // ...их позиции в строке - 1-60)
11
12 // переводим CRC в ASCII
13 sCRC:=BYTE_TO_STRH(byCRC);
14
15 // если рассчитанная CRC соответствует CRC ответа...
16 IF sCRC=MID(sReadData, 2, FIND(sReadData, '$R') - 2) THEN
17
18     // сбрасываем флаг неправильной CRC
19     xWrongCRC:=FALSE;
20
21
22
23 // разбираем ответ
24 fb_ANALYZE_DATA(sData:=sReadData);
25
26 // присваиваем значения на выход блока
27 rMV110_8A_output1:=fb_ANALYZE_DATA.arValue[1];
28 rMV110_8A_output2:=fb_ANALYZE_DATA.arValue[2];
29 rMV110_8A_output3:=fb_ANALYZE_DATA.arValue[3];
30 rMV110_8A_output4:=fb_ANALYZE_DATA.arValue[4];
31 rMV110_8A_output5:=fb_ANALYZE_DATA.arValue[5];
32 rMV110_8A_output6:=fb_ANALYZE_DATA.arValue[6];
33 rMV110_8A_output7:=fb_ANALYZE_DATA.arValue[7];
34 rMV110_8A_output8:=fb_ANALYZE_DATA.arValue[8];
35
36 ELSE // если CRC запроса и ответа не совпадают...
37     xWrongCRC:=TRUE; // ...выставляем флаг некорректной CRC
38 END_IF
```

Рисунок 4.28 – Код действия GET_DATA_FROM_RESPONSE

Сначала следует проверить контрольную сумму. Контрольная сумма представляет собой сумму ASCII кодов всех символов, расположенных перед контрольной суммой. Следует сравнить ее с контрольной суммой ответа, предварительно преобразовав в строковый вид с помощью функции **BYTE_TO_STRH** из библиотеки **Oscat** (из описания протокола известно, что контрольная сумма находится в двух байтах перед стоп-символом).

Если рассчитанная контрольная сумма совпадает с полученной, то можно переходить к разбору ответа. Предварительно следует сбросить флаг ошибки контрольной суммы.

Разбор ответа производится с помощью экземпляра ФБ **ANALYZE_DATA**. Входной переменной для блока является ответ модуля в виде строки, выходной – массив из значений типа **REAL**.

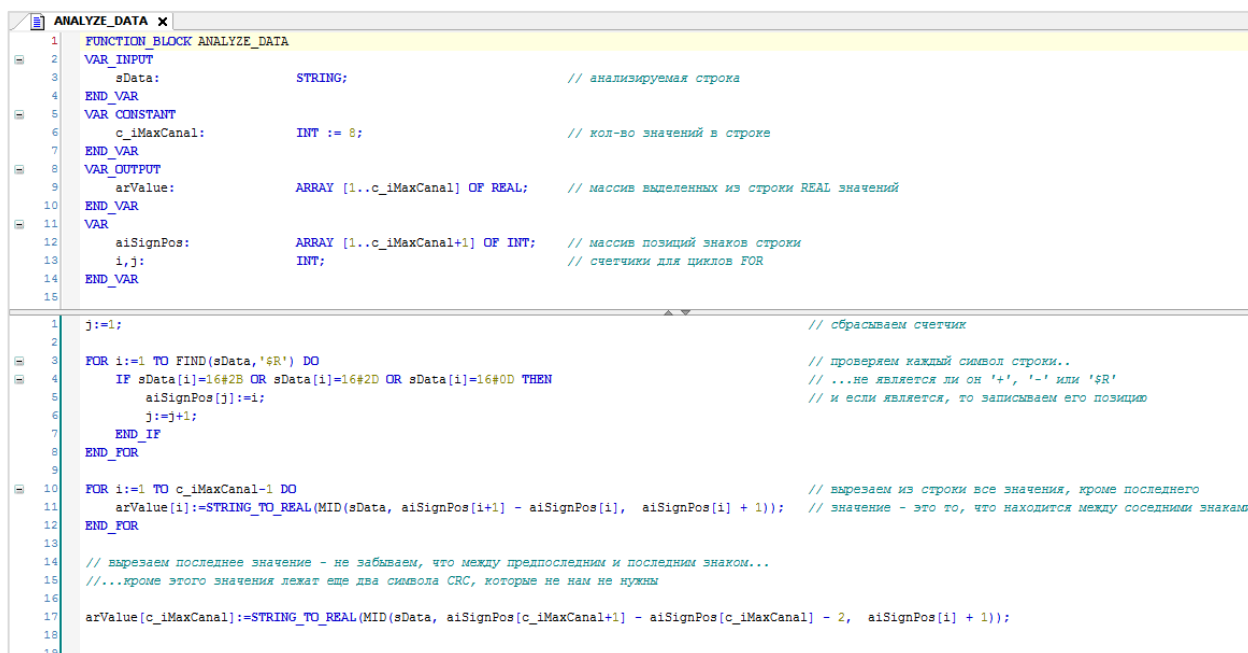


Рисунок 4.29 – Код ФБ ANALYZE_DATA

Как преобразовать строку со значениями в переменные типа **REAL**, при условии, что количество символов для каждого значения нам заранее неизвестно? (например, оно может быть как таким: **11.22**, так и таким: **333.444**)

Для преобразования строки со значениями в переменные типа **REAL**, при условии, что количество символов для каждого значения заранее неизвестно, следует ориентироваться на специальные символы-разделители. В контексте чисел такими знаками являются «+» и «-». Можно сказать, что значение – это то, что расположено между двумя любыми знаками, включая первый знак.

Сначала следует найти в строке данные знаки и запомнить их позиции. Также требуется найти стоп-символ для определения последнего значения в ответе.

В качестве примера приведем ответ, в котором содержатся 3 значения (1.2, -33.44 и 555.666). В реальном ответе от модуля будет 8 значений фиксированного размера, но принцип останется неизменным.

Затем следует найти позиции знаков и стоп-символа (они соответственно, равны 2, 6, 12, 22). Таким образом, первое значение занимает символы 2–5, второе – 6–11, а третье 12–19 (известно, что между последним значением и стоп-символом также находятся два символа CRC). Затем следует вырезать из строки соответствующие фрагменты и преобразовать их функцией **STRING_TO_REAL**. Ввиду упомянутых особенностей, последнее значение придется обрабатывать отдельно.

Символ ответа	>	+	1	.	2	-	3	3	.	4	4	+	5	5	5	.	6	6	6	CRC1	CRC2	\$R
Позиция	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Затем следует вернуться на уровень выше, в действие **GET_DATA_FROM_RESPONSE** (см. [рисунки 4.28](#)). После анализа ответа получается массив из 8-и значений типа **REAL**. Значения массива следует присвоить выходным переменным ФБ **MV110_8A_DCON**.

Если рассчитанная контрольная сумма не совпала с контрольной суммой ответа, то нет смысла анализировать его – достаточно взвести флаг некорректной контрольной суммы (**xWrongCRC**).

Теперь следует перейти на уровень выше и вернуться в код шага **RECEIVE_RESPONSE** (см. [рисунки 4.27](#)).

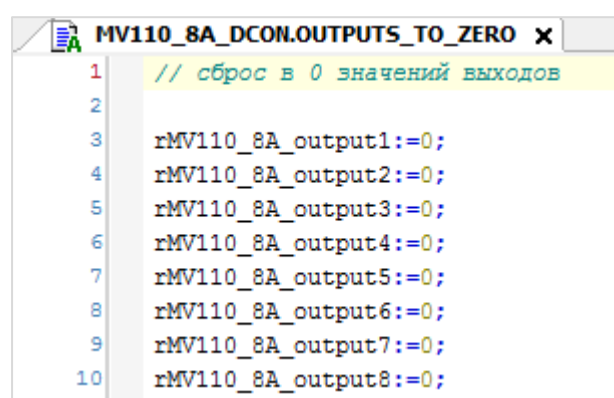
4. Пример опроса модуля MB110-8A по протоколу DCON

```
75      GET_DATA_FROM_RESPONSE();           // выделяем данные из ответа
76      eState := RESPONSE_DELAY;           // переходим к шагу задержки перед следующим циклом опроса
77      END_IF
78  END_IF
79
80      IF fb_TON.Q THEN                     // если сработал таймер таймаута...
81          xTimeout := TRUE;               // взводим флаг ошибки по таймауту
82          OUTPUTS_TO_ZERO();              // сбрасываем значения выходов блока
83          eState := RESPONSE_DELAY;       // переходим к шагу задержки перед следующим циклом опроса
84      END_IF
85
```

Рисунок 4.30 – Фрагмент кода шага RECCEIVE_RESPONSE

После анализа ответа следует перейти к шагу задержки перед следующим циклом опроса.

Если сработал таймер таймаута (основная причина – отсутствие ответа от модуля в течение заданного интервала времени), то следует выставить флаг ошибки по таймауту, выполнить действие **OUTPUTS_TO_ZERO** (сброс выходных переменных ФБ **MV110_8A_DCON** со считанными с модуля значениями) и перейти к шагу **RESPONSE_DELAY**.



```
MV110_8A_DCON.OUTPUTS_TO_ZERO x
1 // сброс в 0 значений выходов
2
3 rMV110_8A_output1:=0;
4 rMV110_8A_output2:=0;
5 rMV110_8A_output3:=0;
6 rMV110_8A_output4:=0;
7 rMV110_8A_output5:=0;
8 rMV110_8A_output6:=0;
9 rMV110_8A_output7:=0;
10 rMV110_8A_output8:=0;
```

Рисунок 4.31 – Код действия OUTPUTS_TO_ZERO

4.4.4 Организация задержки (шаг RESPONSE_DELAY)

На шаге **RESPONSE_DELAY** происходит запуск таймера задержки перед следующим циклом опроса. Задержка может понадобиться при опросе конкретных устройств, которые после отправки ответа блокируют линию связи на определенное время. Время задержки определяется с помощью константы **c_tDelay**.

После запуска таймера происходит переход к шагу **POLLING_CYCLE_ENDS**.

```
87      RESPONSE_DELAY: // шаг задержки перед следующим циклом опроса
88
89
90
91      fb_TON(IN:=FALSE, PT:=c_tDelay);     // сбрасываем таймер
92      fb_TON(IN:=TRUE, PT:=c_tDelay);      // запускаем таймер задержки
93
94      eState := POLLING_CYCLE_ENDS;        // переходим к шагу окончания опроса
```

Рисунок 4.32 – Код шага RESPONSE_DELAY

4.4.5 Завершение цикла опроса (шаг POLLING_CYCLE_ENDS)

На шаге **POLLING_CYCLE_ENDS** следует вызвать таймер задержки перед следующим циклом опроса. После его срабатывания происходит завершение работы экземпляров ФБ чтения и записи [COM.Read](#) и [COM.Write](#), сброс флага получения корректного ответа (в следующем цикле он должен быть просчитан заново), установка флага успешного завершения текущего цикла опроса и переход к шагу **CREATE_REQUEST**.

```

96      POLLING_CYCLE_ENDS: // шаг окончания цикла опроса
97
98      Fb_TON();           // вызываем таймер задержки
99
100     IF Fb_TON.Q THEN    // если сработал таймер задержки...
101         Fb_COMread (xExecute:=FALSE); // сбрасываем ФБ чтение из порта
102         Fb_COMwrite(xExecute:=FALSE); // сбрасываем ФБ записи в порт
103
104         xCorrectAnswer := FALSE;      // сбрасываем флаг корректного ответа
105
106         xDone           := TRUE;      // выставляем флаг окончания текущего цикла опроса модуля
107
108         eState := CREATE_REQUEST;     // переходим к шагу подготовки запроса
109     END_IF

```

Рисунок 4.33 – Код шага POLLING_CYCLE_ENDS

После написания кода для всех шагов опроса модуля следует вернуться к записи алгоритма работы ФБ **MV110_8A_DCON** и вспомнить, что весь написанный код начинает выполняться только при условии **xEnable=TRUE**. В процессе работы программы может возникнуть ситуация, когда обмен следует остановить. Соответственно, в данном случае потребуется также выполнить ряд действий (в операторе **ELSE**) – сбросить флаги завершения опроса и получения корректного ответа, обнулить значения выходных переменных типа **REAL** (см. рисунок 4.31) и перейти на шаг **CREATE_REQUEST** (чтобы при следующем запуске ФБ начать его выполнение с первого шага).

4. Пример опроса модуля MB110-8A по протоколу DCON

```
1
2 IF xEnable THEN           // если получен сигнал опроса модуля...
3
4     CASE eState OF        // ...то запускаем циклический опрос
5
6         CREATE_REQUEST: // шаг подготовки запроса
7
8         [11 lines]
9
10        SEND_REQUEST: // шаг отправки запроса к модулю
11
12        [27 lines]
13
14        RECEIVE_RESPONSE: // шаг получения ответа от модуля
15
16        [33 lines]
17
18        RESPONSE_DELAY: // шаг задержки перед следующим циклом опроса
19
20        [5 lines]
21
22        POLLING_CYCLE_ENDS: // шаг окончания цикла опроса
23
24        [13 lines]
25
26        END_CASE
27
28 ELSE                       // если сигнал опроса модуля пропал...
29     xDone := FALSE;        // сбрасываем флаг окончания текущего цикла опроса модуля
30
31     xCorrectAnswer := FALSE; // сбрасываем флаг корректного ответа
32
33     OUTPUTS_TO_ZERO();     // сбрасываем значения выходов блока
34
35     eState := CREATE_REQUEST; // переходим к шагу подготовки запроса
36
37 END_IF
```

Рисунок 4.34 – Код ФБ MV110_8A_DCON (код отдельных шагов скрыт)

Функциональный блок опроса модуля **MB110-8A** по протоколу **DCON** готов. Его листинг приведен в [приложении А.2](#).

4.5 Программа опроса (PLC_PRG)

Теперь следует вызвать созданные ранее ФБ в программе **PLC_PRG**. Код программы выглядит следующим образом:

```

1  CASE i OF
2
3
4      0:      // открываем COM-порт COM3
5
6      xOpen:=TRUE;
7
8      fb_COMcontrol
9      (
10     xOpen      := xOpen,
11     xClose     := xClose,
12     uiPortNumber := 3,
13     udiBaudrate := 115200,
14     uiParity    := COM.PARITY.NONE,
15     uiByteSize  := 8,
16     uiStopBits  := COM.STOPBIT.ONESTOPBIT
17     );
18
19     IF fb_COMcontrol.xDone THEN
20         i:=1;
21     END_IF
22
23     1:      // опрашиваем модуль MB110-8A по протоколу DCON
24
25     fb_MV110_8A_DCON
26     (
27     xEnable     := fb_COMcontrol.xDone,
28     hCom        := fb_COMcontrol.hCom,
29     byAddress    := 1
30     );
31
32     IF fb_MV110_8A_DCON.xDone THEN
33         i:=1;
34     END_IF
35
36 END_CASE
37

```

Рисунок 4.35 – Код программы PLC_PRG

При запуске программы однократно выполняется код шага 0, что приводит к открытию COM-порта контроллера с заданными настройками с помощью ФБ **COM_CONTROL**. После успешного открытия порта происходит переход на шаг 1, в котором с помощью ФБ **MV110_8A_DCON** организуется опрос модуля с заданным адресом. После окончания опроса модуля можно переходить к опросу следующего устройства. В примере других устройств нет, поэтому программа остается на текущем шаге и продолжает циклически опрашивать модуль.

Программа **PLC_PRG** привязана к задаче с временем цикла **10 мс**. Рекомендуется привязывать программы обмена к задачам с наименьшим временем цикла (не более 10–20 мс).

На рисунке 4.36 приведен скриншот программы в процессе работы. Значение второго канала равно **99999** – что соответствует возникновению в канале исключительной ситуации (например, обрыв датчика). Это является особенностью модуля, а не протокола **DCON**.

4. Пример опроса модуля MB110-8A по протоколу DCON

Device.Application.PLC_PRG		
Выражение	Тип	Значение
fb_COMcontrol	COM_CONTROL	
fb_MV110_8A_DCON	MV110_8A_DCON	
xEnable	BOOL	TRUE
hCom	DWORD	21
byAdress	BYTE	1
xDone	BOOL	FALSE
xWriteError	BOOL	FALSE
xReadError	BOOL	FALSE
xTimeout	BOOL	FALSE
xWrongCRC	BOOL	FALSE
rMV110_8A_output1	REAL	24.836
rMV110_8A_output2	REAL	99999
rMV110_8A_output3	REAL	28.8879986
rMV110_8A_output4	REAL	28.8539982
rMV110_8A_output5	REAL	26.3009987
rMV110_8A_output6	REAL	26.267
rMV110_8A_output7	REAL	26.267
rMV110_8A_output8	REAL	26.267
eState	DCON_STATE	POLLING_CYCLE_ENDS

Рисунок 4.36 – Считанные значения каналов модуля

5 Пример опроса счетчика СЭТ-4ТМ.03М

5.1 Формулировка задачи

В качестве примера будет рассмотрен опрос счетчика [СЭТ-4ТМ.03М](#) по нестандартному бинарному протоколу с помощью контроллера **СПК1хх [M01]**. В качестве опрашиваемого параметра используем текущее напряжение по фазе А (**Ua**). Счетчик подключен к порту **COM3**, его сетевые настройки приведены в таблице ниже.

Таблица 5.1 – Сетевые настройки счетчика СЭТ-4ТМ

Параметр	Значение
<i>COM-порт контроллера</i>	COM3
Адрес модуля	200
Скорость обмена	9600
Количество бит данных	8
Контроль четности	Нечетный
Количество стоп-бит	1

Пример создан в среде **CODESYS 3.5 SP11 Patch 5** и подразумевает запуск на **СПК1хх [M01]** с таргет-файлом **3.5.11.x**. В случае необходимости запуска проекта на другом устройстве следует изменить таргет-файл в проекте (**ПКМ** на узел **Device** – **Обновить устройство**).

Пример доступен для скачивания: [Example SET 4TM.zip](#)

Листинг ROU приведен в [приложении Б](#).

5.2 Описание протокола

Счетчики **СЭТ-4ТМ.03М** поддерживают нестандартный Modbus-подобный двоичный протокол. Его полное описание может быть получено по запросу к [производителю](#). Ниже приводятся ключевые моменты, на которые стоит обращать внимание при реализации обмена. Следует отметить, что в рамках примера будет использована не вся приведенная информация.

1. Критерием окончания ответа от счетчика является пауза длиной в 6–8 мс.
2. Максимальный размер буфера приема/передачи – 96 байт.
3. Структура запроса к счетчику:

Сетевой адрес	Код запроса	Код параметра	Параметры	Контрольная сумма
1 байт/5 байт	1 байт/5 байт	1 байт, может отсутствовать	До 91 байта, может отсутствовать	2 байта

Адрес и код запроса могут быть как обычными (занимать 1 байт), так и расширенными (занимать 5 байт). В рамках примера рассматривается работа только с обычными адресами и запросами.

4. Структура ответа счетчика:

Сетевой адрес	Поле данных ответа	Контрольная сумма
1 байт/5 байт	до 93 байт	2 байта

5. Расчет контрольной суммы осуществляется аналогично протоколу Modbus (в спецификации протокола приведен алгоритм расчета).
6. Перед опросом счетчика следует открыть канал связи с помощью специального запроса следующего формата:

Сетевой адрес	Код запроса	Пароль	Контрольная сумма
1 байт/5 байт	16#01	6 байт	2 байта

Пароль второго уровня (уровень «хозяин») доступа по умолчанию – **000000** в ASCII-кодах.

Если в течение 20 секунд счетчик не получает запросов, то канал связи закрывается.

7. Запрос для чтения текущего значения напряжения фазы А (U_a) выглядит следующим образом:
Код запроса – **16#08** (чтение параметров и данных);
Код параметра – **16#1B** (чтение данных в формате с плавающей точкой);
Параметры – один байт кода массива данных и один байт **RWRI** (регистр вспомогательных режимов измерения).
Код массива данных – **16#00**;

Таблица 2-45 – Запрашиваемые массивы данных

Значение байта «Массив данных»	Наименование запрашиваемых данных
<u>00h</u>	Данные вспомогательных режимов измерения (аналогично <u>08h\11h</u>)
01h	Зафиксированные данные вспомогательных режимов измерения (аналогично <u>08h\14h</u>)
02h	Групповые данные вспомогательных режимов измерений (аналогично <u>08h\16h</u>)
03h	Групповые зафиксированные данные вспомогательных режимов измерения (аналогично <u>08h\16h</u>)

Рисунок 5.1 – Таблица кодов массивов данных из описания протокола

Значение RWRI – 0001 00 01 = **16#11**.

Таблица 2-38 – Структура и возможные значения RWRI

RWRI								СЭТ-4ТМ.02	СЭТ-1М.01	СЭТ-4ТМ.03	ПСЧ-4ТМ.05, ПСЧ-3ТМ.05	СЭБ-1ТМ.01, СЭБ-1ТМ.02(1М)
7	6	5	4	3	2	1	0					
Номер вспомогательного режима измерения				Номер параметра		Номер фазы						
0h – мощность: – активная P; – реактивная Q; – полная S				0 – P 1 – Q 2 – S		0 – по сумме фаз 1 – по фазе 1 2 – по фазе 2 3 – по фазе 3		+	+	+	+	+
<u>1h – напряжение:</u> – фазное Uф				<u>0 – Uф</u>		<u>0 – по фазе 1</u> <u>1 – по фазе 1</u> 2 – по фазе 2 3 – по фазе 3		+	+	+	+	+
– межфазное Umф				1 – Umф		0 – межфазное 12 1 – межфазное 12 2 – межфазное 23 3 – межфазное 31		+	*	-	+	-
– прямой последовательности U1(1)				2 – U1(1)		любое		+	*	-	+	-
– встроенной батареи Uб				3 – Uб		любое		-	-	-	-	+

Рисунок 5.2 – Фрагмент таблицы RWRI из описания протокола

Зная структуру запроса, структуру ответа и методику расчета контрольной суммы можно начать реализацию протокола обмена в **CODESYS**.

5.3 Алгоритмизация задачи

Процесс обмена данными через последовательный порт можно представить в виде циклически выполняемого алгоритма. На языке ST для реализации подобных алгоритмов рекомендуется использовать оператор условного выбора CASE.

Алгоритм решаемой в примере задачи можно представить следующим образом:

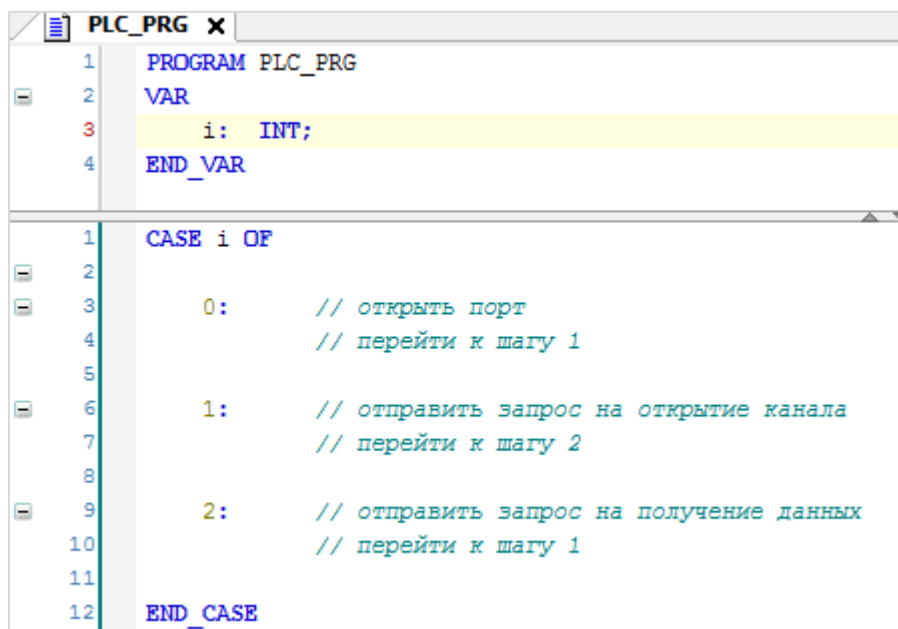


Рисунок 5.3 – Алгоритмизация задачи опроса устройства

Во время запуска программы будет однократно произведено открытие порта, после чего будет происходить циклический опрос счетчика. С целью упрощения примера, запрос на открытие канала к счетчику отправляется в каждом цикле.

Соответственно, задача сводится к написанию кода, который будет выполняться в шагах 0 и 1. Рекомендуется упаковать его в два функциональных блока:

1. ФБ управления портом, который будет вызываться на шаге 0.
2. ФБ опроса модуля, который будет вызываться на шаге 1.

Предварительно следует добавить в проект библиотеку [CAA SerialCom](#), так как ее функции и ФБ потребуются при реализации протокола.

5.4 ФБ управления портом (COM_CONTROL)

Процесс разработки ФБ управления портом описан в [п. 4.4](#). После прочтения этого пункта следует перейти к [п. 5.5](#).

5.5 ФБ опроса счетчика (SET_4ТМ)

В первую очередь следует создать функциональный блок опроса счетчика **СЭТ-4ТМ.03М** по нестандартному протоколу с названием **SET_4ТМ**. Для начала требуется определиться с набором входных и выходных переменных.

В рамках примера его входными переменными будут являться:

1. Вход управления блоком (старт/прекращение работы).
2. Адрес опрашиваемого устройства.
3. Дескриптор (handle) COM-порта.

К выходным переменным будут относиться:

1. Измеренное значение напряжение по фазе А (Ua).
2. Флаг окончания цикла опроса.
3. Флаги ошибок обмена.

Ниже также приводится список локальных переменных ФБ. Необходимость их объявления будет поясняться по мере описания принципов работы блока.

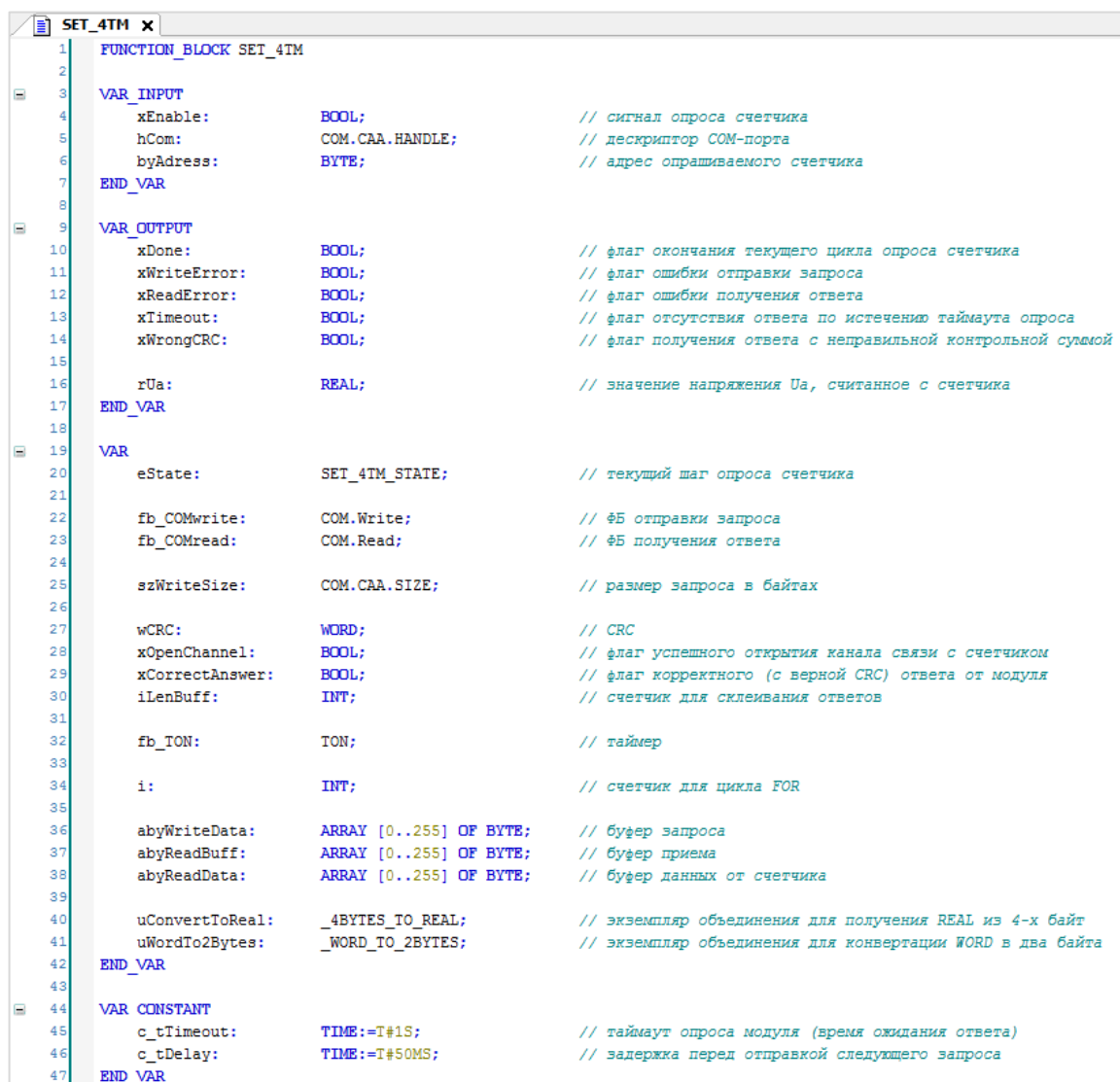
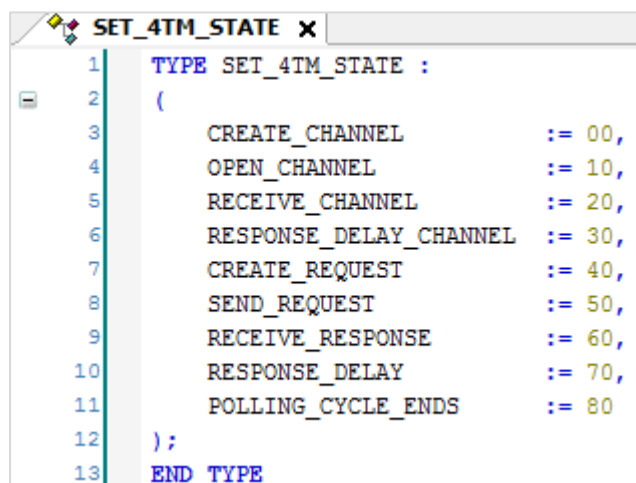


Рисунок 5.4 – Объявление переменных ФБ SET_4ТМ

5. Пример опроса счетчика СЭТ-4ТМ.03М

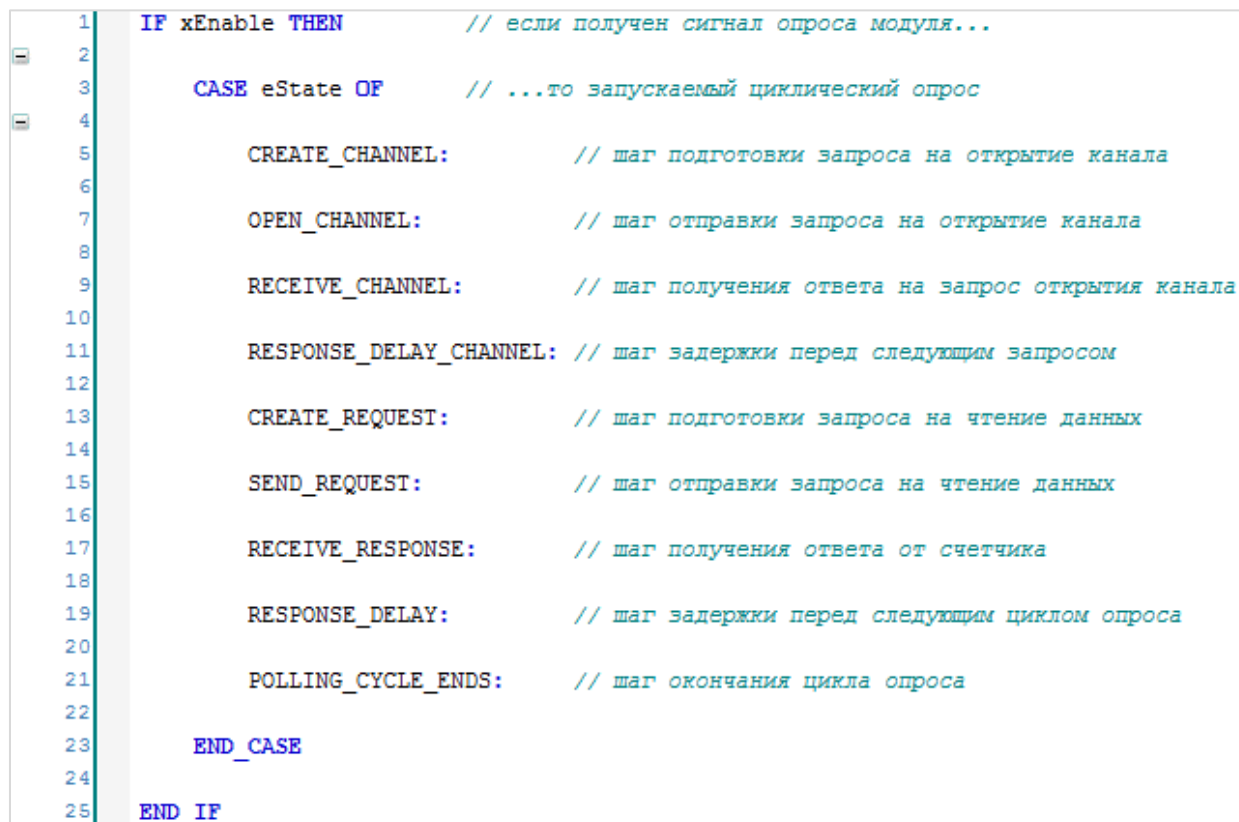
Для реализации блока будет использоваться пошаговый алгоритм с оператором CASE и **перечислением**, которое позволит использовать имена в качестве названий шагов. Экземпляр перечисления имеет название **eState**.



```
1  TYPE SET_4TM_STATE :  
2  (  
3      CREATE_CHANNEL           := 00,  
4      OPEN_CHANNEL             := 10,  
5      RECEIVE_CHANNEL          := 20,  
6      RESPONSE_DELAY_CHANNEL   := 30,  
7      CREATE_REQUEST           := 40,  
8      SEND_REQUEST             := 50,  
9      RECEIVE_RESPONSE         := 60,  
10     RESPONSE_DELAY           := 70,  
11     POLLING_CYCLE_ENDS       := 80  
12 );  
13 END_TYPE
```

Рисунок 5.5 – Объявление перечисления SET_4TM_STATE

Алгоритм опроса модуля в общем виде:



```
1  IF xEnable THEN           // если получен сигнал опроса модуля...  
2  
3      CASE eState OF        // ...то запускаемый циклический опрос  
4  
5          CREATE_CHANNEL:    // шаг подготовки запроса на открытие канала  
6  
7          OPEN_CHANNEL:      // шаг отправки запроса на открытие канала  
8  
9          RECEIVE_CHANNEL:   // шаг получения ответа на запрос открытия канала  
10  
11         RESPONSE_DELAY_CHANNEL: // шаг задержки перед следующим запросом  
12  
13         CREATE_REQUEST:    // шаг подготовки запроса на чтение данных  
14  
15         SEND_REQUEST:      // шаг отправки запроса на чтение данных  
16  
17         RECEIVE_RESPONSE:  // шаг получения ответа от счетчика  
18  
19         RESPONSE_DELAY:    // шаг задержки перед следующим циклом опроса  
20  
21         POLLING_CYCLE_ENDS: // шаг окончания цикла опроса  
22  
23     END_CASE  
24  
25 END_IF
```

Рисунок 5.6 – Алгоритм опроса счетчика

На шаге **CREATE_CHANNEL** будет происходить формирование запроса на открытие канала связи.

На шаге **OPEN_CHANNEL** запрос будет отправлен в COM-порт.

На шаге **RECEIVE_CHANNEL** будет происходить получение и анализ ответа от счетчика.

Шаг **RESPONSE_DELAY_CHANNEL** будет использоваться для создания задержки между получением ответа от счетчика на запрос открытия канала и отправки следующего запроса.

На шаге **CREATE_REQUEST** будет происходить формирование запроса к счетчику.

На шаге **SEND_REQUEST** этот запрос будет отправлен в COM-порт.

На шаге **RECEIVE_RESPONSE** будет происходить получение и анализ ответа от счетчика.

Шаг **RESPONSE_DELAY** будет использоваться для создания задержки между получением ответа от счетчика и отправки следующего запроса.

На шаге **POLLING_CYCLE_ENDS** будет происходить обновление выходов ФБ, после чего следует переход к начальному шагу.



ПРИМЕЧАНИЕ

Необходимость наличия шага **RESPONSE_DELAY** зависит от особенностей конкретного опрашиваемого устройства. Некоторые устройства удерживают линию связи фиксированное время после ответа, в связи с чем следует организовывать задержку перед отправкой следующего запроса. В случае опроса счетчика **СЭТ-4ТМ.03М** такая задержка не обязательна, но рассматривается в настоящем руководстве в качестве примера.

Законченные фрагменты кода ФБ будут оформлены в виде **действий**:

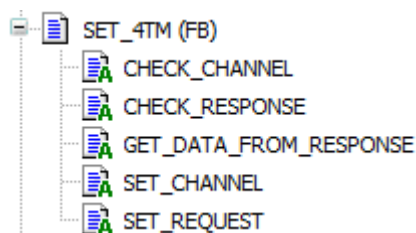


Рисунок 5.7 – Действия ФБ SET_4ТМ

5.5.1 Подготовка запроса на открытие канала (шаг CREATE_CHANNEL)

На шаге **CREATE_CHANNEL** происходит очистка буферов данных и подготовка запроса на открытие канала.

```

6      CREATE_CHANNEL: // шаг подготовки запроса на открытие канала связи
7
8
9
10     _BUFFER_CLEAR(PT:=ADR(abyReadBuff), SIZE:=SIZEOF(abyReadBuff)); // очищаем буфер приема
11     _BUFFER_CLEAR(PT:=ADR(abyReadData), SIZE:=SIZEOF(abyReadData)); // очищаем буфер данных
12     _BUFFER_CLEAR(PT:=ADR(abyWriteData), SIZE:=SIZEOF(abyWriteData)); // очищаем буфер запроса
13
14     xDone := FALSE; // сброс флага окончания предыдущего цикла опроса
15
16     SET_CHANNEL(); // подготовка запроса на открытие канала связи
17
18     eState := OPEN_CHANNEL; // переходим к шагу отправки запроса на открытие канала связи

```

Рисунок 5.8 – Код шага CREATE_CHANNEL

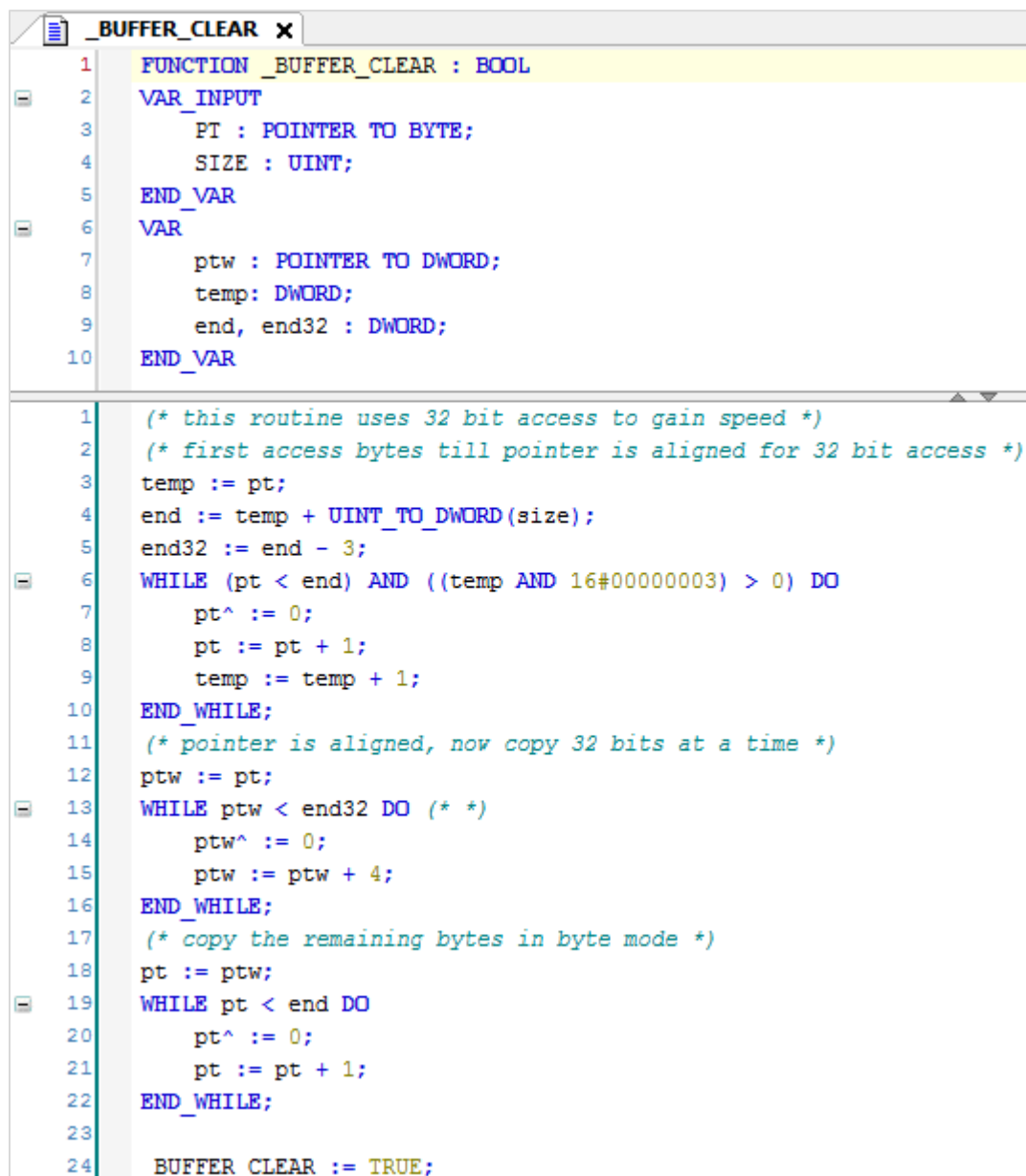
Под буфером понимается переменная, используемая для временного хранения данных. В данном случае такими переменными являются **abyReadBuff** (буфер фрагмента ответа), **abyReadData** (буфер полного ответа) и **abyWriteData** (буфер запроса). В рамках примера счетчику каждый раз отправляются одни и те же запросы, поэтому нет необходимости каждый раз очищать буфер запроса и формировать запрос заново – но в других ситуациях это может понадобиться (например, когда необходимо с помощью одного ФБ считывать несколько параметров устройства с помощью разных запросов).

Для очистки буфера используется функция **_BUFFER_CLEAR** из библиотеки **OSCAT**. Библиотека доступна для скачивания на сайте oscat.de и [сайте компании Овен](http://сайт_компании_Овен) в разделе **CODESYS V3/Библиотеки**. Библиотека OSCAT имеет открытые исходные коды, поэтому во многих случаях удобнее копировать ее функции и ФБ в пользовательский проект (вместо добавления через **Менеджер библиотек**).

Функция **_BUFFER_CLEAR** имеет два входных параметра: адрес буфера и его размер. Вместо задания размера буфера с помощью числа рекомендуется использовать оператор **SIZEOF**, возвращающий

5. Пример опроса счетчика СЭТ-4ТМ.03М

размер переменной в байтах. Возвращаемое значение имеет тип **BOOL** и принимает значение **TRUE** в случае успешного завершения очистки буфера.



```
1 FUNCTION _BUFFER_CLEAR : BOOL
2 VAR_INPUT
3     PT : POINTER TO BYTE;
4     SIZE : UINT;
5 END_VAR
6 VAR
7     ptw : POINTER TO DWORD;
8     temp: DWORD;
9     end, end32 : DWORD;
10 END_VAR
11
12 (* this routine uses 32 bit access to gain speed *)
13 (* first access bytes till pointer is aligned for 32 bit access *)
14 temp := pt;
15 end := temp + UINT_TO_DWORD(size);
16 end32 := end - 3;
17 WHILE (pt < end) AND ((temp AND 16#00000003) > 0) DO
18     pt^ := 0;
19     pt := pt + 1;
20     temp := temp + 1;
21 END_WHILE;
22 (* pointer is aligned, now copy 32 bits at a time *)
23 ptw := pt;
24 WHILE ptw < end32 DO (* *)
25     ptw^ := 0;
26     ptw := ptw + 4;
27 END_WHILE;
28 (* copy the remaining bytes in byte mode *)
29 pt := ptw;
30 WHILE pt < end DO
31     pt^ := 0;
32     pt := pt + 1;
33 END_WHILE;
34 _BUFFER_CLEAR := TRUE;
```

Рисунок 5.9 – Код функции **_BUFFER_CLEAR** из библиотеки **OSCAT**

Код функции скопирован из библиотеки **OSCAT** без каких-либо изменений.

Действие **SET_CHANNEL** используется для формирования запроса на открытие канала. Структура запроса известна из описания протокола счетчика. Следует обратить внимание, что протокол обмена является бинарным, но пароль, определяющий уровень пользователя функции, передается в виде кодов ASCII-символов.

```

1  // подготавливаем запрос на открытие канала
2
3  abyWriteData[0]:=byAddress;           // адрес счетчика
4  abyWriteData[1]:=16#01;               // код функции
5  abyWriteData[2]:=16#30;               // байты 2-7 - пароль "000000" в ASCII
6  abyWriteData[3]:=16#30;
7  abyWriteData[4]:=16#30;
8  abyWriteData[5]:=16#30;
9  abyWriteData[6]:=16#30;
10 abyWriteData[7]:=16#30;
11 //abyWriteData[8]:=16#03;
12 //abyWriteData[9]:=16#D2;
13
14 wCRC:=CRC_MB_GEN(pData:=ADR(abyWriteData), Size:=8); // расчет CRC запроса (8 - число байт запроса без учета CRC)
15
16 uWordTo2Bytes.wValue:=wCRC;           // преобразование CRC в два байта
17
18 abyWriteData[8]:=uWordTo2Bytes.abvWord[0]; // CRC запроса
19 abyWriteData[9]:=uWordTo2Bytes.abvWord[1]; //
20
21 szWriteSize:=10;                       // число байт запроса с учетом CRC
22

```

Рисунок 5.10 – Код действия SET_CHANNEL

В протоколе счетчика используется та же методика расчета контрольной суммы, что и в протоколе Modbus – **CRC16** (циклический избыточный код). Описание алгоритма расчета можно найти в [Википедии](#), спецификации протокола Modbus, спецификации протокола счетчика и т. д. Следует создать ФБ **CRC_MB_GEN**, который будет выполнять эту задачу.

```

1  FUNCTION CRC_MB_GEN : WORD
2
3  VAR_INPUT
4      pData:  POINTER TO BYTE;      (* указатель на блок данных *)
5      Size:   WORD;                 (* размер блока данных *)
6  END_VAR
7
8  VAR
9      Cnt:    BYTE;                 (* счетчик битов *)
10 END_VAR
11
12 (* расчет CRC для MODBUS RTU*)
13
14 CRC_MB_GEN := 16#FFFF;
15 WHILE Size > 0 DO
16     CRC_MB_GEN := CRC_MB_GEN XOR pData^;
17     FOR Cnt := 0 TO 7 DO
18         IF CRC_MB_GEN.0 = 0 THEN
19             CRC_MB_GEN := SHR(CRC_MB_GEN, 1);
20         ELSE
21             CRC_MB_GEN := SHR(CRC_MB_GEN, 1) XOR 16#A001;
22         END_IF
23     END_FOR;
24     pData := pData + 1;
25     Size := Size - 1;
26 END_WHILE
27

```

Рисунок 5.11 – Код ФБ CRC_MB_GEN

**ПРИМЕЧАНИЕ**

У внимательного читателя, вероятно, возникнет вопрос – почему бы не воспользоваться готовым блоком расчета CRC из библиотеки **OSCAT**?

Да, действительно, в OSCAT есть ФБ **CRC_GEN**, который выполняет данную задачу – но из-за особенностей реализации он не возвращает корректное значение для пакетов данных размером меньше, чем 4 байта. В нашем случае ответ от счетчика на запрос открытия канала связи занимает всего 2 байта – так что придется воспользоваться своим ФБ.

5. Пример опроса счетчика СЭТ-4ТМ.03М

Рассчитанная контрольная сумма представляет собой переменную типа **WORD** – но чтобы записать ее в массив, следует представить ее в виде двух байт. Для этого создается объединение **_WORD_TO_2BYTES** (его экземпляр **uWordTo2Bytes** мы уже объявили в переменных **ФБ SET_4ТМ**, см. рисунок 5.4).

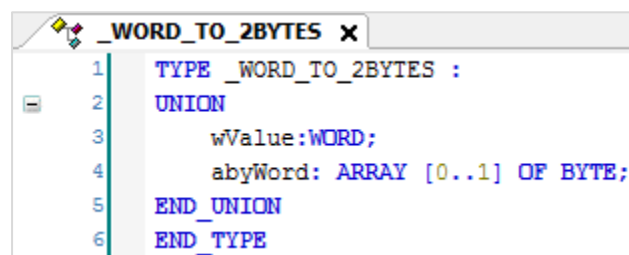


Рисунок 5.12 – Объявление объединения **_WORD_TO_2BYTES**

Последняя операция в действии **SET_CHANNEL** – указание числа байт запроса (переменная **szWriteSize**).

Затем следует вернуться к коду шага **CREATE_CHANNEL**.

Кроме очистки буферов и формирования запроса на данном шаге происходит сброс флага окончания предыдущего цикла, после чего осуществляется переход к шагу [OPEN_CHANNEL](#).

```
6  CREATE_CHANNEL: // шаг подготовки запроса на открытие канала связи  
7  
8  
9      _BUFFER_CLEAR(PT:=ADR(abyReadBuff), SIZE:=SIZEOF(abyReadBuff)); // очищаем буфер приема  
10     _BUFFER_CLEAR(PT:=ADR(abyReadData), SIZE:=SIZEOF(abyReadData)); // очищаем буфер данных  
11     _BUFFER_CLEAR(PT:=ADR(abyWriteData), SIZE:=SIZEOF(abyWriteData)); // очищаем буфер запроса  
12  
13     xDone := FALSE; // сброс флага окончания предыдущего цикла опроса  
14  
15     SET_CHANNEL(); // подготовка запроса на открытие канала связи  
16  
17     eState := OPEN_CHANNEL; // переходим к шагу отправки запроса на открытие канала связи  
18
```

Рисунок 5.13 – Код шага **CREATE_CHANNEL**

5.5.2 Отправка запроса на открытие канала (шаг OPEN_CHANNEL)

На шаге **OPEN_CHANNEL** происходит отправка запроса, сформированного в предыдущем шаге.

```

20 OPEN_CHANNEL: // шаг отправки запроса на открытие канала связи
21
22
23     fb_COMwrite                                // запускаем ФБ записи в порт
24     (
25         xExecute := TRUE,
26         hCom      := hCom,
27         pBuffer   := ADR(abyWriteData),
28         szSize    := szWriteSize
29     );
30
31 IF fb_COMwrite.xError THEN                      // если при отправке запроса произошла ошибка...
32     xWriteError := TRUE;                       // выставляем флаг ошибки отправки запроса
33     eState      := POLLING_CYCLE_ENDS;         // переходим к шагу окончания опроса
34 END_IF
35
36
37 IF fb_COMwrite.xDone THEN                      // если отправка запроса успешно завершена...
38     fb_COMwrite(xExecute:=FALSE);             // сбрасываем ФБ записи в порт
39
40     xWriteError := FALSE;                     // сбрасываем флаг ошибки отправки запроса
41
42     fb_TON(IN:=FALSE, PT:=c_tTimeout);        // сбрасываем таймер
43     fb_TON(IN:=TRUE, PT:=c_tTimeout);         // запускаем таймер таймаута
44
45     eState := RECEIVE_CHANNEL;                // переходим к шагу получения ответа от счетчика
46 END_IF
47

```

Рисунок 5.14 – Код шага SEND_REQUEST

В данном шаге производится работа с экземпляром ФБ [COM.Write](#).

Входными переменным блока являются сигнал его запуска (**xExecute**), дескриптор COM-порта (**hCom**), адрес буфера запроса (**pBuffer**) и размер буфера (**szWriteSize**). Размер буфера указывается в явном виде в коде действия **SET_CHANNEL** (см. [рисунок 5.10](#)).

После вызова блока отправки запроса следует проанализировать его результат. Если во время отправки произошла ошибка (у ФБ **fb_COMwrite** выход **xError = TRUE**), то следует взвести флаг ошибки и перейти к шагу завершения опроса.

Если запрос успешно отправлен (у **fb_COMwrite** выход **xDone = TRUE**), то следует завершить работу блока, сбросить флаг ошибки, запустить таймер таймаута (он потребуется на следующем шаге) и перейти к шагу [RECEIVE_CHANNEL](#).

5. Пример опроса счетчика СЭТ-4ТМ.03М

5.5.3 Получение ответа на запрос открытия канала (шаг RECEIVE_CHANNEL)

На шаге **RECEIVE_CHANNEL** происходит получение ответа от счетчика на запрос открытия канала.

```
49 RECEIVE_CHANNEL: // шаг получения ответа от счетчика
50
51
52
53     fb_COMread                                     // запускаем ФБ чтения в порт
54     (
55         xExecute := TRUE,
56         hCom      := hCom,
57         pBuffer   := ADR(abyReadBuff),
58         szBuffer  := SIZEOF(abyReadBuff)
59     );
60
61     fb_TON();                                     // вызываем таймер таймаута
62
63     IF fb_COMread.xError THEN                     // если при получении ответа произошла ошибка...
64         xReadError := TRUE;                       // выставляем флаг ошибки получения ответа
65         eState      := POLLING_CYCLE_ENDS;        // переходим к шагу окончания опроса
66     END_IF
67
68
69     IF fb_COMread.xDone THEN                     // если ФБ чтения из порта завершил работу...
70         xTimeout:=FALSE;                         // сбрасываем флаг таймаута
71         CHECK_CHANNEL();                         // проверяем корректность ответа
72         fb_COMread(xExecute := FALSE);          // завершаем работу блока чтения из порта
73
74         IF xOpenChannel THEN                     // если канал успешно открыт...
75             eState:=RESPONSE_DELAY_CHANNEL;    // ...то переходим к шагу задержки перед следующим запросом
76         END_IF
77     END_IF
78
79
80     IF fb_TON.Q THEN                             // если сработал таймер таймаута...
81         xTimeout := TRUE;                       // взводим флаг ошибки по таймауту
82         rUa      := 0;                         // сбрасываем значения выходов блока
83         eState    := RESPONSE_DELAY;           // переходим к шагу задержки перед следующим циклом опроса
84     END_IF
```

Рисунок 5.15 – Код шага RECEIVE_CHANNEL

В данном шаге производится работа с экземпляром ФБ [COM.Read](#).

Входными переменными блока являются сигнал его запуска (**xExecute**), дескриптор (handle) COM-порта (**hCom**), адрес буфера фрагмента ответа (**pBuffer**) и размер буфера (**szSize**). Вместо задания размера буфера с помощью рекомендуется использовать оператор **SIZEOF**, возвращающий размер переменной в байтах.

Следует понимать, что в ряде случаев подчиненное устройство не ответит на запрос (например, в случае некорректного запроса или неисправности устройства). Если не предусмотреть эту ситуацию, то программа можно «застыть» на шаге получения ответа. Для предотвращения данной ситуации используется контроль таймаута опроса – по истечению заданного времени осуществляется выход из текущего шага вне зависимости от факта получения ответа. Таймер таймаута был запущен в конце предыдущего шага, поэтому на данном шаге достаточно вызвать его с теми же аргументами. Время таймаута определяется с помощью константы **c_tTimeout**.

После вызова блока получения ответа следует проанализировать его результат. Если при получении произошла ошибка (у **fb_COMread** выход **xError = TRUE**), то следует взвести флаг ошибки и перейти к шагу завершения опроса.

Если ответ успешно получен (у **fb_COMread** выход **xDone = TRUE**), то следует сбросить флаг ошибки таймаута, проверить корректность ответа и завершить работу блока.

Проверка корректности ответа вынесена в действие **CHECK_CHANNEL**:

```

1  SET_4TM.CHECK_CHANNEL x
2
3  IF fb_COMread.szSize>0 THEN // если из порта были считана порция данных...
4
5      FOR i:=0 TO ANY_TO_INT(fb_COMread.szSize)-1 DO
6          abyReadData[iLenBuff + i]:=abyReadBuff[i]; // ...то приклеиваем их к предыдущей порции
7      END_FOR // ...и остаемся на этом же шаге, чтобы считать следующую порцию данных
8
9      iLenBuff:=iLenBuff + ANY_TO_INT(fb_COMread.szSize); // следующая порция должна быть записана после предыдущей
10
11      wCRC:=CRC_MB_GEN(pData:=ADR(abyReadData), Size:=2); // расчет CRC ответа (2 - число байт корректного ответа без учета CRC)
12
13      uWordTo2Bytes.wValue:=wCRC; // преобразование CRC в два байта
14
15      // если первый байт ответа совпадает с адресом счетчика, а рассчитанная CRC соответствует CRC ответа...
16      IF abyReadData[0]=byAddress AND abyReadData[1]=0 AND abyReadData[2]=uWordTo2Bytes.abvWord[0] AND abyReadData[3]=uWordTo2Bytes.abvWord[1] THEN
17          xWrongCRC := FALSE; // ...то сбрасываем флаг неправильной CRC
18          xOpenChannel := TRUE; // взводим флаг получения корректного ответа
19      ELSE // в противном случае...
20          xWrongCRC := TRUE; // ...взводим флаг неправильной CRC
21      END_IF
22  END IF

```

Рисунок 5.16 – Код действия CHECK_CHANNEL

В некоторых случаях (в частности, при чтении большого количества данных), ответ от подчиненного устройства может прийти в виде нескольких фрагментов. Чтобы корректно обработать данную ситуацию следует каждый полученный фрагмент ответа «приклеивать» к предыдущему фрагменту в буфере полного ответа (**abyReadData**), управляя позицией записи в буфер с помощью переменной **iLenBuff**.

Затем следует проверить корректность ответа. Из описания протокола известно, что в случае успешного открытия канала, счетчик отвечает пакетом со следующей структурой:

Адрес счетчика	16#00	CRC
1 байт	1 байт	2 байта

Соответственно, следует проверить, что нулевой байт ответа содержит адрес счетчика, значение первого байта – 0, а во втором и третьем и байтах содержится корректная контрольная сумма. Для этого используется ФБ **CRC_MB_GEN** и объединение **_WORD_TO_2BYTES** (см. рисунок 5.16).

Если все условия выполняются, то канал связи успешно открыт. Следует сбросить флаг некорректной CRC и взвести флаг открытия канала (**xOpenChannel**). В противном случае следует взвести флаг некорректной контрольной суммы (для упрощения не рассматривается ситуация, когда ответ содержит код ошибки и корректную контрольную сумму).

Затем следует вернуться к шагу **RECEIVE_CHANNEL** (см. рисунок 5.17). После выполнения кода действия **CHECK_CHANNEL** следует завершить работу ФБ **COM.Read**. Если взведен флаг корректного ответа (**xOpenChannel**), то следует к шагу [RESPONSE_DELAY_CHANNEL](#).

Если сработал таймер таймаута (основной причиной является отсутствие ответа от модуля в течение заданного интервала времени), то следует взвести флаг ошибки по таймауту, сбросить значение счетчика в 0 и перейти к шагу [RESPONSE_DELAY](#).

5. Пример опроса счетчика СЭТ-4ТМ.03М

```
IF fb_COMread.xDone THEN           // если фБ чтения из порта завершил работу...
    xTimeout:=FALSE;               // сбрасываем флаг таймута
    CHECK_CHANNEL();               // проверяем корректность ответа
    fb_COMread(xExecute := FALSE); // завершаем работу блока чтения из порта

    IF xOpenChannel THEN           // если канал успешно открыт...
        eState:=RESPONSE_DELAY_CHANNEL; // ...то переходим к шагу задержки перед следующим запросом
    END_IF
END_IF

IF fb_TON.Q THEN                   // если сработал таймер таймута...
    xTimeout := TRUE;              // вводим флаг ошибки по таймауту
    rUa := 0;                     // сбрасываем значения выходов блока
    eState := RESPONSE_DELAY;      // переходим к шагу задержки перед следующим циклом опроса
END_IF
```

Рисунок 5.17 – Фрагмент кода шага RECEIVE_CHANNEL

5.5.4 Организация задержки (шаг RESPONSE_DELAY_CHANNEL)

На шаге **RESPONSE_DELAY_CHANNEL** происходит запуск таймера задержки перед следующим запросом. Задержка может понадобиться в случае опроса конкретных устройств, которые после отправки ответа блокируют линию связи на определенное время. Время задержки определяется с помощью константы **c_tDelay**.

После запуска таймера происходит переход к шагу [CREATE_REQUEST](#).

```
87      RESPONSE_DELAY_CHANNEL: // шаг задержки перед следующим запросом
88
89
90      fb_TON(IN:=FALSE, PT:=c_tDelay);           // сбрасываем таймер
91      fb_TON(IN:=TRUE, PT:=c_tDelay);            // запускаем таймер задержки
92
93      eState := CREATE_REQUEST;                  // переходим к шагу подготовки запроса данных
```

Рисунок 5.18 – Код шага RESPONSE_DELAY_CHANNEL

5.5.5 Подготовка запроса на чтение данных (шаг CREATE_REQUEST)

На шаге **CREATE_REQUEST** происходит подготовка запроса на чтение данных счетчика.

```
96      CREATE_REQUEST: // шаг подготовки запроса данных
97
98      fb_TON();           // вызываем таймер задержки
99
100
101      SET_REQUEST();      // подготовка запроса
102
103      xDone := FALSE;     // сброс флага окончания предыдущего цикла опроса
104
105      IF fb_TON.Q THEN    // если сработал таймер задержки...
106          eState := SEND_REQUEST; // переходим к шагу отправки запроса
107      END_IF
108
```

Рисунок 5.19 – Код шага CREATE_REQUEST

Для формирования запроса используется действие **SET_REQUEST**. Структура запроса известна из описания протокола счетчика (см. [п. 5.2](#)). Расчет контрольной суммы происходит по аналогии с шагом [CREATE_CHANNEL](#).

```

1  // подготавливаем запрос на получение данных
2
3  abyWriteData[0]:=byAddress;           // адрес счетчика
4  abyWriteData[1]:=16#00;               // код функции
5  abyWriteData[2]:=16#1B;               // код параметра
6  abyWriteData[3]:=16#00;               // код массива данных
7  abyWriteData[4]:=16#11;               // RWRI
8  //abyWriteData[5]:=16#77;
9  //abyWriteData[6]:=16#BB;
10
11 wCRC:=CRC_MB_GEN(pData:=ADR(abyWriteData), Size:=5); // расчет CRC запроса (5 - число байт запроса без учета CRC)
12
13 uWordTo2Bytes.wValue:=wCRC;           // преобразование CRC в два байта
14
15 abyWriteData[5]:=uWordTo2Bytes.abvWord[0]; // CRC запроса
16 abyWriteData[6]:=uWordTo2Bytes.abvWord[1]; //
17
18 szWriteSize:=7;                       // число байт запроса с учетом CRC

```

Рисунок 5.20 – Код действия SET_REQUEST

Затем следует вернуться к коду шага **CREATE_REQUEST**. Помимо формирования запроса, следует вызвать таймер задержки перед следующим запросом, запущенный на шаге **RESPONSE_DELAY_CHANNEL**. После его срабатывания следует перейти к шагу [SEND_REQUEST](#).

5.5.6 Отправка запроса на чтение данных (шаг SEND_REQUEST)

На шаге **SEND_REQUEST** происходит отправка запроса, сформированного на предыдущем шаге.

```

110  SEND_REQUEST: // шаг отправки запроса на получение данных
111
112  fb_COMwrite                                     // запускаем ФБ записи в порт
113  (
114    xExecute := TRUE,
115    hCom     := hCom,
116    pBuffer  := ADR(abyWriteData),
117    szSize   := szWriteSize
118  );
119
120
121
122  IF fb_COMwrite.xError THEN                      // если при отправке запроса произошла ошибка...
123    xWriteError := TRUE;                         // выставляем флаг ошибки отправки запроса
124    eState      := POLLING_CYCLE_ENDS;          // переходим к шагу окончания опроса
125  END_IF
126
127
128  IF fb_COMwrite.xDone THEN                      // если отправка запроса успешно завершена...
129    fb_COMwrite(xExecute:=FALSE);               // сбрасываем ФБ записи в порт
130
131    xWriteError := FALSE;                       // сбрасываем флаг ошибки отправки запроса
132
133    fb_TON(IN:=FALSE, PT:=c_tTimeout);          // сбрасываем таймер таймаута
134    fb_TON(IN:=TRUE, PT:=c_tTimeout);           // запускаем таймер таймаута
135
136    eState := RECEIVE_RESPONSE;                // переходим к шагу получения ответа от модуля
137  END_IF
138
139  iLenBuff := 0;                               // обнуляем счетчик для склеивания ответов
140
141  _BUFFER_CLEAR(PT:=ADR(abyReadBuff), SIZE:=SIZEOF(abyReadBuff)); // очищаем буфер приема
142  _BUFFER_CLEAR(PT:=ADR(abyReadData), SIZE:=SIZEOF(abyReadData)); // очищаем буфер данных

```

Рисунок 5.21 – Код шага SEND_REQUEST

В данном шаге производится работа с экземпляром ФБ [COM.Write](#).

5. Пример опроса счетчика СЭТ-4ТМ.03М

Входными переменным блока являются сигнал его запуска (**xExecute**), дескриптор (handle) COM-порта (**hCom**), адрес буфера запроса (**pBuffer**) и размер буфера (**szWriteSize**). Размер буфера указывается в явном виде в коде действия **SET_REQUEST** (см. рисунок 5.20).

После вызова блока отправки запроса необходимо проанализировать его результат. Если при отправке произошла ошибка (у ФБ **fb_COMwrite** выход **xError = TRUE**), то следует взвести флаг ошибки и перейти к шагу завершения опроса.

Если запрос успешно отправлен (у **fb_COMwrite** выход **xDone = TRUE**), то следует завершить работу блока, сбросить флаг ошибки, запустить таймер таймаута (он потребуется на следующем шаге) и перейти к шагу [RECEIVE_RESPONSE](#).

На следующем шаге ожидается получение ответа от счетчика, поэтому следует обнулить переменную **iLenBuff** (она используется для склеивания фрагментов ответа от счетчика) и очистить буфер приема и буфер полученных данных (это уже производилось на шаге **CREATE_CHANNEL** – но с того момента в буфер попал ответ счетчика на запрос открытия канала, поэтому следует очистить его повторно).

5.5.7 Получение ответа (шаг RECEIVE_RESPONSE)

На шаге **RECEIVE_RESPONSE** происходит получение ответа от счетчика.

```
145 RECEIVE_RESPONSE: // шаг получения ответа от модуля
146
147 fb_COMread(xExecute := TRUE, hCom := hCom, pBuffer := ADR(abyReadBuff), szBuffer := SIZEOF(abyReadBuff) );
148
149
150 fb_TON(); // вызываем таймер таймаута
151
152 IF fb_COMread.xError THEN // если при получении ответа произошла ошибка...
153     xReadError := TRUE; // выставляем флаг ошибки получения ответа
154     eState := POLLING_CYCLE_ENDS; // переходим к шагу окончания опроса
155 END_IF
156
157 IF fb_COMread.xDone THEN // если ФБ чтения из порта завершил работу...
158     xTimeout := FALSE; // сбрасываем флаг таймаута
159     CHECK_RESPONSE(); // проверяем корректность ответа
160     fb_COMread(xExecute := FALSE); // сбрасываем ФБ чтения из порта
161
162     IF xCorrectAnswer THEN // если ответ корректен...
163         GET_DATA_FROM_RESPONSE(); // выделяем из него данные
164         eState := RESPONSE_DELAY; // переходим к шагу задержки перед следующим циклом опроса
165     END_IF
166 END_IF
167
168
169 IF fb_TON.Q THEN // если сработал таймер таймаута...
170     xTimeout := TRUE; // взводим флаг ошибки по таймауту
171     rUa := 0; // сбрасываем значения выходов блока
172     eState := RESPONSE_DELAY; // переходим к шагу задержки перед следующим циклом опроса
173 END_IF
```

Рисунок 5.22 – Код шага RECEIVE_RESPONSE

В данном шаге производится работа с экземпляром ФБ [COM.Read](#).

Входными переменным блока являются сигнал его запуска (**xExecute**), дескриптор (handle) COM-порта (**hCom**), адрес буфера фрагмента ответа (**pBuffer**) и размер буфера (**szSize**). Вместо задания размера буфера с помощью числа рекомендуется использовать оператор **SIZEOF**, возвращающий размер переменной в байтах.

В ряде случаев подчиненное устройство не ответит на запрос (например, в случае некорректного запроса или неисправности устройства). Если не предусмотреть эту ситуацию, то программа может «застыть» на шаге получения ответа. Для предотвращения данной ситуации используется контроль таймаута опроса – по истечению заданного времени осуществляется выход из текущего шага вне зависимости от факта получения ответа. Таймер таймаута был запущен в конце предыдущего шага, поэтому на данном шаге достаточно вызвать его с теми же аргументами. Время таймаута определяется с помощью константы **c_timeout**.

После вызова блока получения ответа следует проанализировать его результат. Если при получении ответа произошла ошибка (у **fb_COMread** выход **xError = TRUE**), то следует взвести флаг ошибки и перейти к шагу **RESPONSE_DELAY**.

Если ответ успешно получен (у **fb_COMread** выход **xDone = TRUE**), то следует сбросить флаг ошибки таймута, проверить корректность ответа и завершить работу блока.

Проверка корректности ответа вынесена в действие **CHECK_RESPONSE**:

```

1  IF fb_COMread.szSize>0 THEN                                // если из порта была считана порция данных...
2
3
4  FOR i:=0 TO ANY_TO_INT(fb_COMread.szSize)-1 DO
5      abyReadData[iLenBuff + i]:=abyReadBuff[i];            // ...то приклеиваем их к предыдущей порции...
6  END_FOR                                                    // ...и остаемся на этом же шаге, чтобы считать следующую порцию данных
7
8      iLenBuff:=iLenBuff + ANY_TO_INT(fb_COMread.szSize);    // следующая порция должна быть записана после предыдущей
9
10     wCRC:=CRC_MB_GEN(pData:=ADR(abyReadData), Size:=5);    // расчет CRC ответа (5 - число байт корректного ответа без учета CRC)
11
12     uWordTo2Bytes.wValue:=wCRC;                             // преобразование CRC в два байта
13
14     // если первый байт ответа совпадает с адресом счетчика, а рассчитанная CRC соответствует CRC ответа...
15     IF abyReadData[0]=byAdress AND abyReadData[5]=uWordTo2Bytes.abvWord[0] AND abyReadData[6]=uWordTo2Bytes.abvWord[1] THEN
16         xWrongCRC      := FALSE;                            // ...то сбрасываем флаг неправильной CRC
17         xCorrectAnswer  := TRUE;                             // взводим флаг получения корректного ответа
18     ELSE
19         xWrongCRC      := TRUE;                              // в противном случае...
20     END_IF
21 END_IF

```

Рисунок 5.23 – Код действия CHECK_RESPONSE

В некоторых случаях (в частности, при чтении большого количества данных), ответ от подчиненного устройства может прийти в виде нескольких фрагментов. Чтобы корректно обработать эту ситуацию, следует каждый полученный фрагмент ответа «приклеивать» к предыдущему фрагменту в буфере полного ответа (**abyReadData**), управляя позицией записи в буфер с помощью переменной **iLenBuff**.

Затем следует проверить корректность ответа. Из описания протокола известно, что в случае успешного открытия канал, счетчик отвечает пакетом со следующей структурой:

Адрес счетчика	Значение Ua	CRC
1 байт	4 байта	2 байта

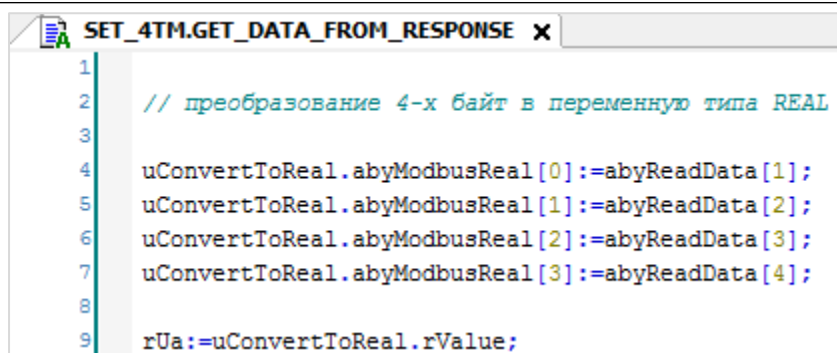
Соответственно, следует проверить, что нулевой байт ответа содержит адрес счетчика, а в пятом и шестом байтах содержится корректная CRC. Для этого используется ФБ **CRC_MB_GEN** и объединение **_WORD_TO_2BYTES**.

Если все условия выполняются, то от счетчика получен ответ с корректной CRC. Следует сбросить флаг некорректной CRC и взвести флаг получения корректного ответа (**xCorrectAnswer**). В противном случае, следует взвести флаг некорректной CRC (для упрощения не рассматривается ситуация, когда ответ содержит код ошибки и корректную CRC).

Затем следует вернуться к коду шага **RECEIVE_RESPONSE** (см. рисунок 5.22).

Если полученный от счетчика ответ корректен (**xCorrectAnswer=TRUE**), то можно приступить к его анализу. Анализ вынесен в действие **GET_DATA_FROM_RESPONSE**.

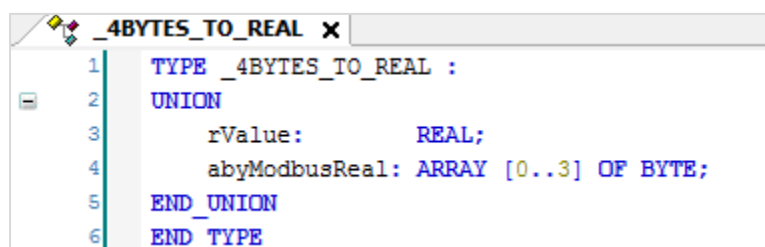
5. Пример опроса счетчика СЭТ-4ТМ.03М



```
1
2 // преобразование 4-х байт в переменную типа REAL
3
4 uConvertToReal.abvModbusReal[0]:=abyReadData[1];
5 uConvertToReal.abvModbusReal[1]:=abyReadData[2];
6 uConvertToReal.abvModbusReal[2]:=abyReadData[3];
7 uConvertToReal.abvModbusReal[3]:=abyReadData[4];
8
9 rUa:=uConvertToReal.rValue;
```

Рисунок 5.24 – Код действия GET_DATA_FROM_RESPONSE

Ответ счетчика содержит измеренное напряжение по фазе А (Ua) в виде [значения с плавающей точкой в формате IEEE 754](#), которое занимает 4 байта. Но в контроллере требуется отображать это значение в виде переменной типа **REAL**. Соответственно, следует преобразовать его с помощью **объединения**. Ранее объединение **_WORD_TO_2BYTES** использовалось при расчете CRC. Создадим еще одно объединение с названием **_4BYTES_TO_REAL**:

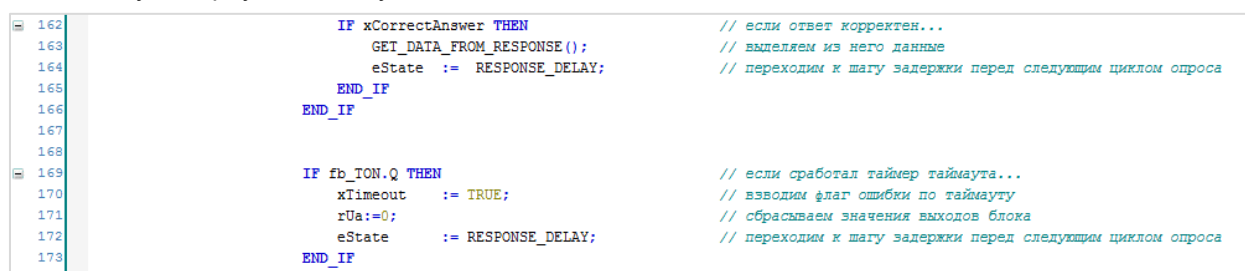


```
1 TYPE _4BYTES_TO_REAL :
2 UNION
3     rValue:      REAL;
4     abvModbusReal: ARRAY [0..3] OF BYTE;
5 END_UNION
6 END_TYPE
```

Рисунок 5.25 – Объявление объединения _4BYTES_TO_REAL

Экземпляр **uConvertToReal** этого объединения уже объявлен в переменных ФБ **SET_4TM**. Следует записать в него байты считанного значения из буфера приема (в нулевом байте содержится адрес счетчика, поэтому он не используется) и забрать из объединения значение типа **REAL**. Таким образом, в переменную **rUa** попадет измеренное значение напряжение по фазе А.

Затем следует вернуться к коду шага **RECEIVE_RESPONSE**:



```
162 IF xCorrectAnswer THEN // если ответ корректен...
163     GET_DATA_FROM_RESPONSE(); // выделяем из него данные
164     eState := RESPONSE_DELAY; // переходим к шагу задержки перед следующим циклом опроса
165 END_IF
166 END_IF
167
168
169 IF fb_TON.Q THEN // если сработал таймер таймаута...
170     xTimeout := TRUE; // взводим флаг ошибки по таймауту
171     rUa:=0; // сбрасываем значения выходов блока
172     eState := RESPONSE_DELAY; // переходим к шагу задержки перед следующим циклом опроса
173 END_IF
```

Рисунок 5.26 – Фрагмент кода шага RECEIVE_RESPONSE

После анализа ответа следует перейти к шагу [RESPONSE_DELAY](#).

Если сработал таймер таймаута (основной причиной этого является отсутствие ответа от модуля в течение заданного интервала времени), то следует выставить флаг ошибки по таймауту, обнулить значение переменной **rUa** и перейти к шагу [RESPONSE_DELAY](#).

5.5.8 Организация задержки (шаг RESPONSE_DELAY)

На шаге **RESPONSE_DELAY** происходит запуск таймера задержки перед следующим циклом опроса. Такая задержка может понадобиться для опроса конкретных устройств, которые после отправки ответа блокируют линию связи на определенное время. Время задержки определяется с помощью константы **c_tDelay**.

После запуска таймера происходит переход к шагу [POLLING_CYCLE_ENDS](#).

```

176      RESPONSE_DELAY: // шаг задержки перед следующим циклом опроса
177
178
179
180      fb_TON(IN:=FALSE, PT:=c_tDelay);           // сбрасываем таймер задержки
181      fb_TON(IN:=TRUE, PT:=c_tDelay);           // запускаем таймер задержки
182
183      eState := POLLING_CYCLE_ENDS;             // переходим к шагу окончания опроса
184

```

Рисунок 5.27 – Код шага RESPONSE_DELAY

5.5.9 Завершение цикла опроса (шаг POLLING_CYCLE_ENDS)

На шаге **POLLING_CYCLE_ENDS** следует вызвать таймер задержки перед следующим циклом опроса. После его срабатывания происходит завершение работы экземпляров ФБ **COM.Read** и **COM.Write**, обнуление переменной **iLenBuff**, сброс флага открытого канала, сброс флага получения корректного ответа (в следующем цикле они должны быть просчитаны заново), установка флага успешного завершения текущего цикла опроса и переход к шагу подготовки запроса на открытие канала.

```

185      POLLING_CYCLE_ENDS: // шаг окончания цикла опроса
186
187
188      fb_COMread (xExecute:=FALSE);             // сбрасываем ФБ чтение из порта
189      fb_COMwrite(xExecute:=FALSE);            // сбрасываем ФБ записи в порт
190
191      fb_TON();                                 // вызываем таймер задержки
192
193      IF fb_TON.Q THEN                          // если сработал таймер задержки...
194
195          iLenBuff      := 0;                   // обнуляем счетчик для склеивания ответов
196
197          xOpenChannel  := FALSE;               // сбрасываем флаг открытого канала
198          xCorrectAnswer := FALSE;             // сбрасываем флаг корректного ответа
199          xDone         := TRUE;               // выставляем флаг окончания текущего цикла опроса счетчика
200
201          eState := CREATE_CHANNEL;            // переходим к шагу подготовки запроса на открытие канала
202      END_IF

```

Рисунок 5.28 – Код шага POLLING_CYCLE_ENDS

После написания кода для всех шагов опроса модуля следует вернуться к записи алгоритма работы ФБ **SET_4TM** и вспомнить, что весь написанный код начинает выполняться только при условии **xEnable=TRUE**. Но в процессе работы программы может возникнуть ситуация, когда обмен необходимо остановить. Соответственно, в данном случае потребуется также выполнить ряд действий (в операторе **ELSE**) – сбросить флаги завершения опроса, открытого канала и получения корректного ответа, обнулить значение выходных переменных ФБ и перейти на шаг [CREATE_CHANNEL](#) (чтобы при следующем запуске ФБ начать его выполнение с первого шага).

5. Пример опроса счетчика СЭТ-4ТМ.03М

```
2  IF xEnable THEN           // если получен сигнал опроса счетчика...
3
4  CASE eState OF           // ...то запускаем циклический опрос
5
6      CREATE_CHANNEL: // шаг подготовки запроса на открытие канала связи
7
8      [11 lines]
9
10     OPEN_CHANNEL: // шаг отправки запроса на открытие канала связи
11
12     [26 lines]
13
14     RECEIVE_CHANNEL: // шаг получения ответа от счетчика
15
16     [34 lines]
17
18     RESPONSE_DELAY_CHANNEL: // шаг задержки перед следующим запросом
19
20     [6 lines]
21
22     CREATE_REQUEST: // шаг подготовки запроса данных
23
24     [12 lines]
25
26     SEND_REQUEST: // шаг отправки запроса на получение данных
27
28     [32 lines]
29
30     RECEIVE_RESPONSE: // шаг получения ответа от модуля
31
32     [29 lines]
33
34     RESPONSE_DELAY: // шаг задержки перед следующим циклом опроса
35
36     [5 lines]
37
38     POLLING_CYCLE_ENDS: // шаг окончания цикла опроса
39
40     [16 lines]
41
42     END_CASE
43
44 ELSE
45     xDone := FALSE;           // если сигнал опроса модуля пропал...
46                                 // сбрасываем флаг окончания текущего цикла опроса счетчика
47
48     xOpenChannel := FALSE;     // сбрасываем флаг открытого канала
49
50     xCorrectAnswer := FALSE;   // сбрасываем флаг открытого канала
51
52     rUa := 0;                 // сбрасываем значения выходов блока
53
54     eState := CREATE_CHANNEL; // переходим к шагу подготовки запроса на открытие канала
55
56 END_IF
```

Рисунок 5.29 – Код ФБ SET_4ТМ (без кода отдельных шагов)

Функциональный блок считывания измеренного напряжения фазы А с счетчика **СЭТ-4ТМ.03М** готов. Его листинг приведен в [приложении Б.2](#).

5.6 Программа опроса (PLC_PRG)

Код программы **PLC_PRG** выглядит следующим образом:

```

1  CASE i OF
2
3      0:      // открываем COM-порт COM3
4
5      xOpen:=TRUE;
6
7      fb_COMcontrol
8      (
9      xOpen      := xOpen,
10     xClose     := xClose,
11     uiPortNumber := 3,
12     udiBaudrate := 9600,
13     uiParity    := COM.PARITY.ODD,
14     uiByteSize  := 8,
15     uiStopBits  := COM.STOPBIT.ONESTOPBIT
16     );
17
18     IF fb_COMcontrol.xDone THEN
19         i:=1;
20     END_IF
21
22
23     1:      // опрашиваем счетчик СЭТ-4ТМ.03
24
25     fb_SET_4TM
26     (
27     xEnable     := fb_COMcontrol.xDone,
28     hCom        := fb_COMcontrol.hCom,
29     byAddress    := 200
30     );
31
32     IF fb_SET_4TM.xDone THEN
33         i:=1;
34     END_IF
35
36 END_CASE
37

```

Рисунок 5.30 – Код программы PLC_PRG

Во время запуска программы однократно выполняется код шага 0, что приводит к открытию COM-порта контроллера с заданными настройками при помощи ФБ **COM_CONTROL**. После успешного открытия порта происходит переход на шаг 1, в котором с помощью ФБ **СЭТ-4ТМ.03М** организуется опрос модуля с заданным адресом. После окончания цикла опроса происходит переход к началу шага 1, что запускает новый цикл опроса.

5. Пример опроса счетчика СЭТ-4ТМ.03М

Программа **PLC_PRG** привязана к задаче с временем цикла 10 мс. Рекомендуется привязывать программы обмена к задачам с наименьшим временем цикла.

На рисунке ниже приведен скриншот программы в процессе работы.

Device.Application.PLC_PRG		
Выражение	Тип	Значение
fb_COMcontrol	COM_CONTROL	
fb_SET_4TM	SET_4TM	
xEnable	BOOL	TRUE
hCom	DWORD	23
byAdress	BYTE	200
xDone	BOOL	FALSE
xWriteError	BOOL	FALSE
xReadError	BOOL	FALSE
xTimeout	BOOL	FALSE
xWrongCRC	BOOL	FALSE
rUa	REAL	225.053024
eState	SET_4TM_STATE	RESPONSE_DELAY_CHANNEL

Рисунок 5.31 – Считанное значение напряжения Ua

6 Рекомендации и замечания

Ниже кратко перечислены основные тезисы и рекомендации по реализации нестандартных протоколов, использованные в примерах данного документа:

1. ФБ и программы, участвующие в обмене, разбиваются на шаги, которые выполняются через оператор CASE.
2. Чтобы сделать прозрачным переходы между шагами, можно использовать перечисления.
3. Чтобы упростить отладку и повысить читабельность кода, можно выделять его законченные фрагменты в действия.
4. Переход к следующему шагу должен происходить только после окончания предыдущего. Контроль окончания шага, в частности, может осуществляться с помощью переменных **xDone** соответствующих ФБ и таймеров.
5. Если возможна ситуация «застывания» на одном из шагов, необходимо использовать таймер таймаута, который позволяет выйти из шага после превышения заданного интервала времени.
6. Некоторые подчиненные устройства удерживают линию определенное время после ответа – следует предусмотреть задержку перед каждым следующим запросом.
7. После завершения цикла обмена, работа всех его ФБ должна быть завершена (обычно под этим понимается их вызов с параметром **xExecute=FALSE**).
8. В текстовых протоколах для разбора полученных данных используются функции работы со строками, в бинарных – объединения и указатели.
9. Операции чтения и записи должны выполняться в разных циклах. Утверждение справедливо и для операций открытия и закрытия порта. Реализовать разнесение операций по циклам помогает оператор CASE.

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

1. В случае необходимости опрашивать несколько устройств, следует увеличить число шагов в программе PLC_PRG. Т. е. после опроса первого подчиненного устройства (шаг 1) должен происходить переход к опросу второго (шаг 2), потом третьего (шаг 3) и т. д. После опроса последнего устройства следует начать новый цикл опроса (с первого устройства).
2. В общем случае ФБ опроса должны быть как можно более универсальными и предоставлять пользователю возможность задавать адрес модуля, адрес и количество считываемых/записываемых регистров, используемую функцию и т. д.
3. Если описание протокола содержит объемный список кодов ошибок, то вместо битовых флагов (как в примерах) рекомендуется использовать одну WORD переменную, которая будет содержать код ошибки и перечисление с расшифровкой кодов.
4. В некоторых случаях требуется тщательная обработка ошибок обмена. Например, в случае отсутствия ответа от модуля можно сделать еще несколько попыток опроса перед переходом к опросу следующего устройства. В случае, когда ошибки обмена связаны с работой контроллера, рекомендуется закрыть COM-порт, очистить буферные переменные, сделать задержку в 100 мс и открыть порт снова.
5. Для представления пакетов данных хорошо подходят структуры – их использование увеличивает прозрачность работы программы.
6. Указатели являются эффективным средством для работы с большими объемами данных, но при некорректном использовании могут привести к утечкам памяти и зависанию программы – поэтому работа с ними подразумевает высокую квалификацию программиста и полное понимание производимых операций.

Приложение А. Листинг примера из п. 4

```

PROGRAM PLC_PRG
VAR
    fb_COMcontrol:          COM_CONTROL;
    fb_MV110_8A_DCON:      MV110_8A_DCON;

    i:                      INT;

    xOpen:                  BOOL;
    xClose:                 BOOL;
END_VAR

CASE i OF

    0:                      // открываем COM-порт COM3

        xOpen:=TRUE;

        fb_COMcontrol
        (
            xOpen              := xOpen,
            xClose             := xClose,
            uiPortNumber       := 3,
            udiBaudrate        := 115200,
            uiParity           := COM.PARITY.NONE,
            uiByteSize         := 8,
            uiStopBits         := COM.STOPBIT.ONESTOPBIT
        );

        IF fb_COMcontrol.xDone THEN
            i:=1;
        END_IF

    1:                      // опрашиваем модуль MB110-8A по протоколу DCON

        fb_MV110_8A_DCON
        (
            xEnable             := fb_COMcontrol.xDone,
            hCom                := fb_COMcontrol.hCom,
            byAdress            := 1
        );

        IF fb_MV110_8A_DCON.xDone THEN
            i:=1;
        END_IF

END_CASE

```

A.1 ФБ COM_CONTROL

```

FUNCTION_BLOCK COM_CONTROL

VAR_IN_OUT
    xOpen:          BOOL;
    xClose:         BOOL;
END_VAR

VAR_INPUT
    uiPortNumber:   UINT;
    udiBaudrate:    DINT;
    uiParity:       COM.PARITY;
    uiByteSize:     UINT;
    uiStopBits:     COM.STOPBIT;
END_VAR

VAR_OUTPUT
    xDone:          BOOL;
    hCom:           COM.CAA.HANDLE;
    xOpenError:     BOOL;
    xCloseError:    BOOL;
END_VAR

VAR
    aComParams:     ARRAY [1..5] OF COM.PARAMETER;
    fb_COMopen:     COM.Open;
    fb_COMclose:    COM.Close;
    eState:         COM.STATE;
END_VAR

CASE eState OF

    INITIALIZE:

        fb_COMopen (xExecute:=FALSE);
        fb_COMclose (xExecute:=FALSE);

        eState:=WAITING_FOR_SIGNAL;

    WAITING_FOR_SIGNAL:

        IF xOpen THEN
            IF hCom=0 OR hCom=16#FFFFFFFF THEN
                eState := OPEN_PORT;
            ELSE
                xOpen := FALSE;
                xDone := TRUE;
                eState := INITIALIZE;
            END_IF
        END_IF

        IF xClose THEN
            IF hCom>0 AND hCom<16#FFFFFFFF THEN
                eState := CLOSE_PORT;
            ELSE
                xClose := FALSE;
                xDone := FALSE;
                eState := INITIALIZE;
            END_IF
        END_IF

```

```

OPEN_PORT:

    SETTINGS();

    OPEN();

    IF fb_COMopen.xDone AND fb_COMopen.xExecute THEN
        xOpen := FALSE;
        fb_COMopen(xExecute:=FALSE);
        xDone := TRUE;
        eState := INITIALIZE;
    END_IF

    IF fb_COMopen.xError AND fb_COMopen.xExecute THEN
        xOpenError := TRUE;
        fb_COMopen(xExecute:=FALSE);
        xDone := FALSE;
        eState := INITIALIZE;
    END_IF

CLOSE_PORT:

    CLOSE();

    IF fb_COMclose.xDone AND fb_COMclose.xExecute THEN
        xClose := FALSE;
        xDone := FALSE;
        hCom := 0;
        fb_COMclose(xExecute:=FALSE);
        eState := INITIALIZE;
    END_IF

    IF fb_COMclose.xError AND fb_COMclose.xExecute THEN
        xCloseError := TRUE;
        fb_COMclose(xExecute:=FALSE);
    END_IF

END_CASE

```

A.1.1. Действие SETTINGS

```

aComParams[1].udiParameterId := COM.CAA_Parameter_Constants.udiPort;
aComParams[1].udiValue       := uiPortNumber;
aComParams[2].udiParameterId := COM.CAA_Parameter_Constants.udiBaudrate;
aComParams[2].udiValue       := udiBaudrate;
aComParams[3].udiParameterId := COM.CAA_Parameter_Constants.udiParity;
aComParams[3].udiValue       := ANY_TO_UDINT(uiParity);
aComParams[4].udiParameterId := COM.CAA_Parameter_Constants.udiByteSize;
aComParams[4].udiValue       := uiByteSize;
aComParams[5].udiParameterId := COM.CAA_Parameter_Constants.udiStopBits;
aComParams[5].udiValue       := ANY_TO_UDINT(uiStopBits);

```

A.1.2. Действие OPEN

```

fb_COMopen.usiListLength := UINT_TO_USINT(SIZEOF(aComParams) / SIZEOF(COM.PARAMETER));
fb_COMopen.pParameterList := ADR(aComParams);
fb_COMopen.xExecute      := TRUE;

fb_COMopen();

hCom := fb_COMopen.hCom;

```

A.1.3. Действие CLOSE

```

fb_COMclose.hCom      := hCom;
fb_COMclose.xExecute  := TRUE;

fb_COMclose();

```

A.2 ФБ MV110_8A_DCON

```

FUNCTION_BLOCK MV110_8A_DCON
VAR_INPUT
    xEnable:          BOOL;           // сигнал опроса модуля
    hCom:             COM.CAA.HANDLE; // дескриптор COM-порта
    byAddress:        BYTE;           // адрес опрашиваемого модуля
END_VAR

VAR_OUTPUT
    xDone:            BOOL;           // флаг окончания текущего цикла опроса модуля
    xWriteError:      BOOL;           // флаг ошибки отправки запроса
    xReadError:       BOOL;           // флаг ошибки получения ответа
    xTimeout:         BOOL;           // флаг отсутствия ответа по истечению таймута опроса
    xWrongCRC:        BOOL;           // флаг получения ответа с неправильной контрольной суммой

    rMV110_8A_output1: REAL;          // считанные значения каналов модуля
    rMV110_8A_output2: REAL;
    rMV110_8A_output3: REAL;
    rMV110_8A_output4: REAL;
    rMV110_8A_output5: REAL;
    rMV110_8A_output6: REAL;
    rMV110_8A_output7: REAL;
    rMV110_8A_output8: REAL;
END_VAR

VAR
    eState:          DCON_state;      // текущий шаг опроса модуля

    fb_COMwrite:      COM.Write;       // ФБ отправки запроса
    fb_COMread:       COM.Read;        // ФБ получения ответа

    sWriteData:       STRING(255);     // запрос к модулю в виде строки
    sReadData:        STRING(255);     // полный ответ модуля в виде строки
    sReadBuff:        STRING(255);     // часть ответа модуля

    sAddress:         STRING(2);       // адрес опрашиваемого модуля в виде строки
    byCRC:            BYTE;            // CRC
    sCRC:             STRING(2);       // CRC в виде строки
    xCorrectAnswer:   BOOL;            // флаг корректного (с верной CRC) ответа от модуля
    i:               INT;              // счетчик для цикла FOR

    fb_TON:           TON;             // таймер
    fb_ANALYZE_DATA:  ANALYZE_DATA;    // ФБ анализа ответа модуля
END_VAR

VAR CONSTANT
    c_tTimeout:      TIME:=T#1S;      // таймаут опроса модуля (время ожидания ответа)
    c_tDelay:        TIME:=T#50MS;    // задержка перед отправкой следующего запроса
END_VAR

IF xEnable THEN

```

CASE eState OF

CREATE_REQUEST:

```
_BUFFER_CLEAR(PT:=ADR(sReadBuff), SIZE:=SIZEOF(sReadBuff));
_BUFFER_CLEAR(PT:=ADR(sReadData), SIZE:=SIZEOF(sReadData));
_BUFFER_CLEAR(PT:=ADR(sWriteData), SIZE:=SIZEOF(sWriteData));

SET_REQUEST();

xDone := FALSE;

eState := SEND_REQUEST;
```

SEND_REQUEST:

```
fb_COMwrite
(
  xExecute := TRUE,
  hCom     := hCom,
  pBuffer  := ADR(sWriteData),
  szSize   := INT_TO_UINT(LEN(sWriteData))
);

IF fb_COMwrite.xError THEN
  xWriteError := TRUE;
  eState      := POLLING_CYCLE_ENDS;
END_IF

IF fb_COMwrite.xDone THEN
  fb_COMwrite(xExecute:=FALSE);

  xWriteError := FALSE;

  fb_TON(IN:=FALSE, PT:=c_tTimeout);
  fb_TON(IN:=TRUE, PT:=c_tTimeout);

  eState := RECEIVE_RESPONSE;
END_IF
```

RECEIVE_RESPONSE:

```
fb_COMread

(
  xExecute := TRUE,
  hCom     := hCom,
  pBuffer  := ADR(sReadBuff),
  szBuffer := SIZEOF(sReadBuff),
);

fb_TON();

IF fb_COMread.xError THEN
  xReadError := TRUE;
  eState      := POLLING_CYCLE_ENDS;
END_IF

IF fb_COMread.xDone THEN
  xTimeout:=FALSE;
  CHECK_RESPONSE();
  fb_COMread(xExecute:=FALSE);

  IF xCorrectAnswer THEN
    GET_DATA_FROM_RESPONSE();
    eState := RESPONSE_DELAY;
  END_IF
END_IF

IF fb_TON.Q THEN
  xTimeout := TRUE;
  OUTPUTS_TO_ZERO();
  eState   := RESPONSE_DELAY;
END_IF
```

```

RESPONSE_DELAY:

    fb_TON(IN:=FALSE, PT:=c_tDelay);
    fb_TON(IN:=TRUE, PT:=c_tDelay);

    eState := POLLING_CYCLE_ENDS;

POLLING_CYCLE_ENDS:

    fb_TON();

    IF fb_TON.Q THEN
        fb_COMread (xExecute:=FALSE);
        fb_COMwrite(xExecute:=FALSE);

        xCorrectAnswer := FALSE;

        xDone           := TRUE;

        eState := CREATE_REQUEST;
    END_IF

END_CASE

ELSE

    xDone := FALSE;

    xCorrectAnswer := FALSE;

    OUTPUTS_TO_ZERO();

    eState := CREATE_REQUEST;
END_IF

```

A.2.1 Действие SET_REQUEST

```

sWriteData[0]:=16#23;

    sAdress:=BYTE_TO_STRH(byAdress);

sWriteData[1]:=sAdress[0];
sWriteData[2]:=sAdress[1];

    byCRC:=sWriteData[0]+sWriteData[1]+sWriteData[2];
    sCRC:=BYTE_TO_STRH(byCRC);

sWriteData[3]:=sCRC[0];
sWriteData[4]:=sCRC[1];

sWriteData[5]:=16#D;

```

A.2.2 Действие CHECK_RESPONSE

```

IF fb_COMread.szSize>0 THEN
    sReadData := CONCAT(sReadData,sReadBuff);

    IF FIND(sReadData, '>') <> 0 AND FIND(sReadData, '$R') <>0 AND
        (FIND(sReadData, '$R') > FIND(sReadData, '>')) THEN

        sReadData:=MID(sReadData, FIND(sReadData, '$R') - FIND(sReadData, '>') + 1,
            FIND(sReadData, '>'));
        xCorrectAnswer:=TRUE;
    END_IF
END_IF

```

A.2.3 Действие GET_DATA_FROM_RESPONSE

```
byCRC:=0;

FOR i:=0 TO FIND(sReadData, '$R') - 4 DO
    byCRC:=byCRC+sReadData[i];
END_FOR

sCRC:=BYTE_TO_STRH(byCRC);

IF sCRC=MID(sReadData, 2, FIND(sReadData, '$R') - 2) THEN

    xWrongCRC:=FALSE;

    fb_ANALYZE_DATA(sData:=sReadData);

    rMV110_8A_output1:=fb_ANALYZE_DATA.arValue[1];
    rMV110_8A_output2:=fb_ANALYZE_DATA.arValue[2];
    rMV110_8A_output3:=fb_ANALYZE_DATA.arValue[3];
    rMV110_8A_output4:=fb_ANALYZE_DATA.arValue[4];
    rMV110_8A_output5:=fb_ANALYZE_DATA.arValue[5];
    rMV110_8A_output6:=fb_ANALYZE_DATA.arValue[6];
    rMV110_8A_output7:=fb_ANALYZE_DATA.arValue[7];
    rMV110_8A_output8:=fb_ANALYZE_DATA.arValue[8];

ELSE
    xWrongCRC:=TRUE;
END_IF
```

A.2.4 Действие OUTPUTS_TO_ZERO

```
rMV110_8A_output1:=0;
rMV110_8A_output2:=0;
rMV110_8A_output3:=0;
rMV110_8A_output4:=0;
rMV110_8A_output5:=0;
rMV110_8A_output6:=0;
rMV110_8A_output7:=0;
rMV110_8A_output8:=0;
```

A.3 ФБ ANALYZE_DATA

```
FUNCTION_BLOCK ANALYZE_DATA

VAR_INPUT
    sData:          STRING;
END_VAR

VAR_CONSTANT
    c_iMaxCanal:    INT := 8;
END_VAR

VAR_OUTPUT
    arValue:        ARRAY [1..c_iMaxCanal] OF REAL;
END_VAR

VAR
    aiSignPos:      ARRAY [1..c_iMaxCanal+1] OF INT;
    i,j:            INT;
END_VAR

j:=1;

FOR i:=1 TO FIND(sData,'$R') DO

    IF sData[i]=16#2B OR sData[i]=16#2D OR sData[i]=16#0D THEN

        aiSignPos[j]:=i;

        j:=j+1;

    END_IF
END_FOR

FOR i:=1 TO c_iMaxCanal-1 DO
    arValue[i]:= STRING_TO_REAL(MID(sData, aiSignPos[i+1] - aiSignPos[i], aiSignPos[i] + 1));
END_FOR

arValue[c_iMaxCanal]:=STRING_TO_REAL(
    MID(sData, aiSignPos[c_iMaxCanal+1] - aiSignPos[c_iMaxCanal] - 2,
    aiSignPos[i] + 1));
```

А.4 Функция BYTE_TO_STRH

```

FUNCTION BYTE_TO_STRH : STRING(2)

VAR_INPUT
    IN : BYTE;
END_VAR

VAR
    temp : BYTE;
    PT : POINTER TO BYTE;
END_VAR

PT := ADR(BYTE_TO_STRH);

temp := SHR(in,4);

IF temp <= 9 THEN temp := temp + 48; ELSE temp := temp + 55; END_IF;

PT^ := temp;

temp := in AND 2#00001111;

IF temp <= 9 THEN temp := temp + 48; ELSE temp := temp + 55; END_IF;

pt := pt + 1;

pt^ := temp;

pt := pt + 1;

pt^:= 0;

```

А.5 Функция _BUFFER_CLEAR

```

FUNCTION _BUFFER_CLEAR : BOOL

VAR_INPUT
    PT : POINTER TO BYTE;
    SIZE : UINT;
END_VAR

VAR
    ptw : POINTER TO DWORD;
    temp: DWORD;
    end, end32 : DWORD;
END_VAR

temp := pt;
end := temp + UINT_TO_DWORD(size);
end32 := end - 3;
WHILE (pt < end) AND ((temp AND 16#00000003) > 0) DO
    pt^ := 0;
    pt := pt + 1;
    temp := temp + 1;
END_WHILE;

ptw := pt;
WHILE ptw < end32 DO
    ptw^ := 0;
    ptw := ptw + 4;
END_WHILE;

pt := ptw;
WHILE pt < end DO
    pt^ := 0;
    pt := pt + 1;
END_WHILE;

_BUFFER_CLEAR := TRUE;

```

А.6 Перечисление COM_STATE

```
TYPE COM_STATE :  
(  
    INITIALIZE           := 00,  
    WAITING_FOR_SIGNAL  := 10,  
    OPEN_PORT           := 20,  
    CLOSE_PORT          := 30  
);  
END_TYPE
```

А.7 Перечисление DCON_STATE

```
TYPE DCON_STATE :  
(  
    CREATE_REQUEST      := 00,  
    SEND_REQUEST        := 10,  
    RECEIVE_RESPONSE    := 20,  
    RESPONSE_DELAY      := 30,  
    POLLING_CYCLE_ENDS  := 40  
);  
END_TYPE
```

Приложение Б. Листинг примера из п. 5

```

PROGRAM PLC_PRG
VAR
    fb_COMcontrol:          COM_CONTROL;
    fb_SET_4TM:             SET_4TM;

    i:                      INT;

    xOpen:                  BOOL;
    xClose:                 BOOL;
END_VAR

CASE i OF

    0:                      // открываем COM-порт COM3

        xOpen:=TRUE;

        fb_COMcontrol
        (
            xOpen            := xOpen,
            xClose           := xClose,
            uiPortNumber      := 3,
            udiBaudrate       := 9600,
            uiParity          := COM.PARITY.ODD,
            uiByteSize        := 8,
            uiStopBits        := COM.STOPBIT.ONESTOPBIT
        );

        IF fb_COMcontrol.xDone THEN
            i:=1;
        END_IF

    1:                      // опрашиваем счетчик СЭТ-4ТМ.03

        fb_SET_4TM
        (
            xEnable          := fb_COMcontrol.xDone,
            hCom              := fb_COMcontrol.hCom,
            byAdress         := 200
        );

        IF fb_SET_4TM.xDone THEN
            i:=1;
        END_IF

END_CASE

```

Б.1 ФБ COM_CONTROL

```

FUNCTION_BLOCK COM_CONTROL

VAR_IN_OUT
    xOpen:          BOOL;          // сигнал открытия порта
    xClose:         BOOL;          // сигнал закрытия порта
END_VAR

VAR_INPUT
    uiPortNumber:   UINT;          // номер порта
    udiBaudrate:    DINT;          // скорость передачи данных
    uiParity:       COM.PARITY;    // четность
    uiByteSize:     UINT;          // кол-во бит данных в байте
    uiStopBits:     COM.STOPBIT;   // кол-во стоп бит
END_VAR

VAR_OUTPUT
    xDone:          BOOL;          // флаг успешного открытия порта
    hCom:           COM.CAA.HANDLE; // дескриптор порта
    xOpenError:     BOOL;          // флаг ошибки открытия порта
    xCloseError:    BOOL;          // флаг ошибки закрытия порта
END_VAR

VAR
    aComParams:     ARRAY [1..5] OF COM.PARAMETER; // структура настроек порта
    fb_COMopen:     COM.Open;          // ФБ открытия порта
    fb_COMclose:    COM.Close;         // ФБ закрытия порта
    eState:         COM.STATE;         // текущий шаг работы с портом
END_VAR

CASE eState OF

    INITIALIZE:

        fb_COMopen (xExecute:=FALSE);
        fb_COMclose (xExecute:=FALSE);

        eState:=WAITING_FOR_SIGNAL;

    WAITING_FOR_SIGNAL:

        IF xOpen THEN
            IF hCom=0 OR hCom=16#FFFFFFFF THEN
                eState := OPEN_PORT;
            ELSE
                xOpen := FALSE;
                xDone := TRUE;
                eState := INITIALIZE;
            END_IF
        END_IF

        IF xClose THEN
            IF hCom>0 AND hCom<16#FFFFFFFF THEN
                eState := CLOSE_PORT;
            ELSE
                xClose := FALSE;
                xDone := FALSE;
                eState := INITIALIZE;
            END_IF
        END_IF

```

```

OPEN_PORT:

    SETTINGS();

    OPEN();

    IF fb_COMopen.xDone AND fb_COMopen.xExecute THEN
        xOpen := FALSE;
        fb_COMopen(xExecute:=FALSE);
        xDone := TRUE;
        eState := INITIALIZE;
    END_IF

    IF fb_COMopen.xError AND fb_COMopen.xExecute THEN
        xOpenError := TRUE;
        fb_COMopen(xExecute:=FALSE);
        xDone := FALSE;
        eState := INITIALIZE;
    END_IF

CLOSE_PORT:

    CLOSE();

    IF fb_COMclose.xDone AND fb_COMclose.xExecute THEN
        xClose := FALSE;
        xDone := FALSE;
        hCom := 0;
        fb_COMclose(xExecute:=FALSE);
        eState := INITIALIZE;
    END_IF

    IF fb_COMclose.xError AND fb_COMclose.xExecute THEN
        xCloseError := TRUE;
        fb_COMclose(xExecute:=FALSE);
    END_IF

END_CASE

```

Б.1.1 Действие SETTINGS

```

aComParams[1].udiParameterId := COM.CAA_Parameter_Constants.udiPort;
aComParams[1].udiValue       := uiPortNumber;
aComParams[2].udiParameterId := COM.CAA_Parameter_Constants.udiBaudrate;
aComParams[2].udiValue       := udiBaudrate;
aComParams[3].udiParameterId := COM.CAA_Parameter_Constants.udiParity;
aComParams[3].udiValue       := ANY_TO_UDINT(uiParity);
aComParams[4].udiParameterId := COM.CAA_Parameter_Constants.udiByteSize;
aComParams[4].udiValue       := uiByteSize;
aComParams[5].udiParameterId := COM.CAA_Parameter_Constants.udiStopBits;
aComParams[5].udiValue       := ANY_TO_UDINT(uiStopBits);

```

Б.1.2 Действие OPEN

```

fb_COMopen.usiListLength := UINT_TO_USINT(SIZEOF(aComParams) / SIZEOF(COM.PARAMETER));
fb_COMopen.pParameterList := ADR(aComParams);
fb_COMopen.xExecute      := TRUE;

fb_COMopen();

hCom := fb_COMopen.hCom;

```

Б.1.3 Действие CLOSE

```

fb_COMclose.hCom      := hCom;
fb_COMclose.xExecute  := TRUE;

fb_COMclose();

```

Б.2 ФБ SET_4TM

```

FUNCTION_BLOCK SET_4TM

VAR_INPUT
    xEnable:          BOOL;
    hCom:              COM.CAA.HANDLE;
    byAdress:          BYTE;
END_VAR

VAR_OUTPUT
    xDone:             BOOL;
    xWriteError:        BOOL;
    xReadError:         BOOL;
    xTimeout:           BOOL;
    xWrongCRC:          BOOL;
    rUa:               REAL;
END_VAR

VAR
    eState:            SET_4TM_STATE;

    fb_COMwrite:        COM.Write;
    fb_COMread:         COM.Read;

    szWriteSize:        COM.CAA.SIZE;

    wCRC:               WORD;
    xOpenChannel:        BOOL;
    xCorrectAnswer:      BOOL;
    iLenBuff:           INT;

    fb_TON:             TON;

    i:                 INT;

    abyWriteData:        ARRAY [0..255] OF BYTE;
    abyReadBuff:         ARRAY [0..255] OF BYTE;
    abyReadData:         ARRAY [0..255] OF BYTE;

    uConvertToReal:      _4BYTES_TO_REAL;
    uWordTo2Bytes:       _WORD_TO_2BYTES;
END_VAR

VAR_CONSTANT
    c_tTimeout:          TIME:=T#1S;
    c_tDelay:            TIME:=T#50MS;
END_VAR

```

```

IF xEnable THEN
    CASE eState OF

        CREATE_CHANNEL:

            _BUFFER_CLEAR(PT:=ADR(abyReadBuff), SIZE:=SIZEOF((abyReadBuff)));
            _BUFFER_CLEAR(PT:=ADR(abyReadData), SIZE:=SIZEOF((abyReadData)));
            _BUFFER_CLEAR(PT:=ADR(abyWriteData), SIZE:=SIZEOF((abyWriteData)));

            xDone := FALSE;

            SET_CHANNEL();

            eState := OPEN_CHANNEL;

        OPEN_CHANNEL: // шаг отправки запроса на открытие канала связи

            fb_COMwrite
            (
                xExecute := TRUE,
                hCom      := hCom,
                pBuffer   := ADR(abyWriteData),
                szSize    := szWriteSize
            );

            IF fb_COMwrite.xError THEN
                xWriteError := TRUE;
                eState      := POLLING_CYCLE_ENDS;
            END_IF

            IF fb_COMwrite.xDone THEN
                fb_COMwrite(xExecute:=FALSE);

                xWriteError := FALSE;

                fb_TON(IN:=FALSE, PT:=c_tTimeout);
                fb_TON(IN:=TRUE,  PT:=c_tTimeout);
                eState := RECEIVE_CHANNEL;
            END_IF

        RECEIVE_CHANNEL:

            fb_COMread
            (
                xExecute := TRUE,
                hCom      := hCom,
                pBuffer   := ADR(abyReadBuff),
                szBuffer  := SIZEOF(abyReadBuff)
            );

            fb_TON();

            IF fb_COMread.xError THEN
                xReadError := TRUE;
                eState      := POLLING_CYCLE_ENDS;
            END_IF

            IF fb_COMread.xDone THEN
                xTimeout:=FALSE;
                CHECK_CHANNEL();
                fb_COMread(xExecute := FALSE);

                IF xOpenChannel THEN
                    eState:=RESPONSE_DELAY_CHANNEL;
                END_IF
            END_IF
    
```

```

IF fb_TON.Q THEN
    xTimeout      := TRUE;
    rUa           := 0;
    eState        := RESPONSE_DELAY;
END_IF

RESPONSE_DELAY_CHANNEL:

fb_TON(IN:=FALSE, PT:=c_tDelay);
fb_TON(IN:=TRUE,  PT:=c_tDelay);

eState := CREATE_REQUEST;

CREATE_REQUEST:

fb_TON();

SET_REQUEST();

xDone := FALSE;

IF fb_TON.Q THEN
    eState := SEND_REQUEST;
END_IF

SEND_REQUEST:

fb_COMwrite
(
    xExecute := TRUE,
    hCom     := hCom,
    pBuffer  := ADR(abyWriteData),
    szSize   := szWriteSize
);

IF fb_COMwrite.xError THEN
    xWriteError := TRUE;
    eState      := POLLING_CYCLE_ENDS;
END_IF

IF fb_COMwrite.xDone THEN
    fb_COMwrite(xExecute:=FALSE);

    xWriteError := FALSE;

    fb_TON(IN:=FALSE, PT:=c_tTimeout);
    fb_TON(IN:=TRUE,  PT:=c_tTimeout);

    eState := RECEIVE_RESPONSE;
END_IF

iLenBuff := 0;

_BUFFER_CLEAR(PT:=ADR(abyReadBuff), SIZE:=SIZEOF(abyReadBuff));
_BUFFER_CLEAR(PT:=ADR(abyReadData), SIZE:=SIZEOF(abyReadData));

```

```

RECEIVE_RESPONSE: // шаг получения ответа от модуля

    fb_COMread
    (
        xExecute := TRUE,
        hCom      := hCom,
        pBuffer   := ADR(abyReadBuff),
        szBuffer  := SIZEOF(abyReadBuff)
    );

    fb_TON();

    IF fb_COMread.xError THEN
        xReadError := TRUE;
        eState     := POLLING_CYCLE_ENDS;
    END_IF

    IF fb_COMread.xDone THEN
        xTimeout:=FALSE;
        CHECK_RESPONSE();
        fb_COMread(xExecute:=FALSE);

        IF xCorrectAnswer THEN
            GET_DATA_FROM_RESPONSE();
            eState := RESPONSE_DELAY;
        END_IF
    END_IF

    IF fb_TON.Q THEN
        xTimeout      := TRUE;
        rUa:=0;
        eState        := RESPONSE_DELAY;
    END_IF

RESPONSE_DELAY:

    fb_TON(IN:=FALSE, PT:=c_tDelay);
    fb_TON(IN:=TRUE,  PT:=c_tDelay);

    eState := POLLING_CYCLE_ENDS;

POLLING_CYCLE_ENDS:

    fb_COMread (xExecute:=FALSE);
    fb_COMwrite(xExecute:=FALSE);

    fb_TON();

    IF fb_TON.Q THEN

        iLenBuff      :=      0;

        xOpenChannel  :=      FALSE;
        xCorrectAnswer:=      FALSE;
        xDone          :=      TRUE;

        eState := CREATE_CHANNEL;
    END_IF

END_CASE

ELSE
    xDone          := FALSE;
    xOpenChannel   := FALSE;
    xCorrectAnswer := FALSE;
    rUa            := 0;
    eState         := CREATE_CHANNEL;
END_IF

```

Б.2.1 Действие SET_CHANNEL

```

abyWriteData[0]:=byAdress;
    abyWriteData[1]:=16#01;
abyWriteData[2]:=16#30;
    abyWriteData[3]:=16#30;
abyWriteData[4]:=16#30;
abyWriteData[5]:=16#30;
abyWriteData[6]:=16#30;
abyWriteData[7]:=16#30;

wCRC:=CRC_MB_GEN(pData:=ADR(abyWriteData), Size:=8);
uWordTo2Bytes.wValue:=wCRC;

abyWriteData[8]:=uWordTo2Bytes.abvWord[0];
abyWriteData[9]:=uWordTo2Bytes.abvWord[1];

szWriteSize:=10;

```

Б.2.2 Действие CHECK_CHANNEL

```

IF fb_COMread.szSize>0 THEN

    FOR i:=0 TO ANY_TO_INT(fb_COMread.szSize)-1 DO
        abyReadData[iLenBuff + i]:=abyReadBuff[i];
    END_FOR

    iLenBuff:=iLenBuff + ANY_TO_INT(fb_COMread.szSize);

    wCRC:=CRC_MB_GEN(pData:=ADR(abyReadData), Size:=2);
    uWordTo2Bytes.wValue:=wCRC;

    IF abyReadData[0]=byAdress AND abyReadData[1]=0 AND abyReadData[2]=uWordTo2Bytes.abvWord[0]
    AND abyReadData[3]=uWordTo2Bytes.abvWord[1] THEN
        xWrongCR      := FALSE;
        xOpenChannel  := TRUE;
    ELSE
        xWrongCRC      := TRUE;
    END_IF
END_IF

```

Б.2.3 Действие SET_REQUEST

```

abyWriteData[0]:=byAdress;
abyWriteData[1]:=16#08;
abyWriteData[2]:=16#1B;
abyWriteData[3]:=16#00;
abyWriteData[4]:=16#11;

wCRC:=CRC_MB_GEN(pData:=ADR(abyWriteData), Size:=5);
uWordTo2Bytes.wValue:=wCRC;

abyWriteData[5]:=uWordTo2Bytes.abvWord[0];
abyWriteData[6]:=uWordTo2Bytes.abvWord[1];

szWriteSize:=7;

```

Б.2.4 Действие CHECK_RESPONSE

```

IF fb_COMread.szSize>0 THEN

    FOR i:=0 TO ANY_TO_INT(fb_COMread.szSize)-1 DO
        abyReadData[iLenBuff + i]:=abyReadBuff[i];
    END_FOR

    iLenBuff:=iLenBuff + ANY_TO_INT(fb_COMread.szSize);
    wCRC:=CRC_MB_GEN(pData:=ADR(abyReadData), Size:=5);
    uWordTo2Bytes.wValue:=wCRC;

    IF abyReadData[0]=byAdress AND abyReadData[5]=uWordTo2Bytes.abyWord[0] AND
    abyReadData[6]=uWordTo2Bytes.abyWord[1] THEN

        xWrongCRC           := FALSE;
        xCorrectAnswer       := TRUE;
    ELSE
        xWrongCRC           := TRUE;
    END_IF
END_IF

```

Б.2.5 Действие GET_DATA_FROM_RESPONSE

```

uConvertToReal.abyModbusReal[0]:=abyReadData[1];
uConvertToReal.abyModbusReal[1]:=abyReadData[2];
uConvertToReal.abyModbusReal[2]:=abyReadData[3];
uConvertToReal.abyModbusReal[3]:=abyReadData[4];

rUa:=uConvertToReal.rValue;

```

Б.3 Функция CRC_MB_GEN

```

FUNCTION CRC_MB_GEN : WORD

VAR_INPUT
    pData:  POINTER TO BYTE;
    Size:   WORD;
END_VAR

VAR
    Cnt:    BYTE;
END_VAR

CRC_MB_GEN := 16#FFFF;
WHILE Size > 0 DO
    CRC_MB_GEN := CRC_MB_GEN XOR pData^;
    FOR Cnt := 0 TO 7 DO
        IF CRC_MB_GEN.0 = 0 THEN
            CRC_MB_GEN := SHR(CRC_MB_GEN, 1);
        ELSE
            CRC_MB_GEN := SHR(CRC_MB_GEN, 1) XOR 16#A001;
        END_IF
    END_FOR;
    pData := pData + 1;
    Size := Size - 1;
END_WHILE

```

Б.4 Функция _BUFFER_CLEAR

```

FUNCTION _BUFFER_CLEAR : BOOL

VAR_INPUT
    PT : POINTER TO BYTE;
    SIZE : UINT;
END_VAR

VAR
    ptw : POINTER TO DWORD;
    temp: DWORD;
    end, end32 : DWORD;
END_VAR

temp := pt;
end := temp + UINT_TO_DWORD(size);
end32 := end - 3;
WHILE (pt < end) AND ((temp AND 16#00000003) > 0) DO
    pt^ := 0;
    pt := pt + 1;
    temp := temp + 1;
END_WHILE;

ptw := pt;
WHILE ptw < end32 DO
    ptw^ := 0;
    ptw := ptw + 4;
END_WHILE;

pt := ptw;
WHILE pt < end DO
    pt^ := 0;
    pt := pt + 1;
END_WHILE;

_BUFFER_CLEAR := TRUE;

```

Б.5 Перечисление COM_STATE

```
TYPE COM_STATE :  
(  
    INITIALIZE           := 00,  
    WAITING_FOR_SIGNAL   := 10,  
    OPEN_PORT            := 20,  
    CLOSE_PORT           := 30  
);  
END_TYPE
```

Б.6 Перечисление SET_4TM_STATE

```
TYPE SET_4TM_STATE :  
(  
    CREATE_CHANNEL       := 00,  
    OPEN_CHANNEL         := 10,  
    RECEIVE_CHANNEL      := 20,  
    RESPONSE_DELAY_CHANNEL := 30,  
    CREATE_REQUEST       := 40,  
    SEND_REQUEST         := 50,  
    RECEIVE_RESPONSE     := 60,  
    RESPONSE_DELAY       := 70,  
    POLLING_CYCLE_ENDS   := 80  
);  
END_TYPE
```