

2017

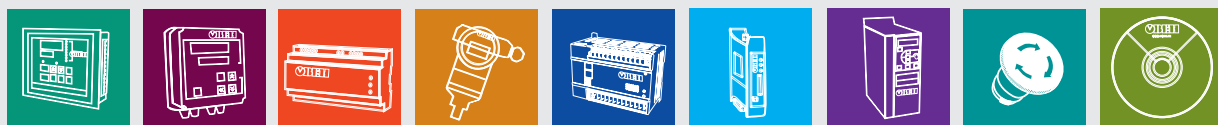


# СПК

## Реализация нестандартных протоколов обмена

Руководство для продвинутых пользователей

Версия: 1.1  
Дата: 15.08.2017



## Оглавление

1. Цель и структура документа .....	4
2. Основные сведения об обмене данными по последовательным интерфейсам .....	5
2.1. Общие принципы организации обмена .....	5
2.2. Работа с COM-портом.....	6
2.3. Типы протоколов обмена .....	7
2.4. Обработка ошибок обмена .....	8
3. Библиотека CAA SerialCom.....	10
3.1. Добавление библиотеки в проект CODESYS.....	10
3.2. Структура COM.PARAMETER.....	11
3.3. Список глобальных констант CAA_Parameter_Constants .....	11
3.4. ФБ COM.Open.....	12
3.5. ФБ COM.Write.....	13
3.6. ФБ COM.Read.....	14
3.7. ФБ COM.Close .....	15
4. Пример опроса модуля MB110-8A по протоколу DCON .....	16
4.1. Формулировка задачи.....	16
4.2. Описание протокола .....	17
4.3. Алгоритмизация задачи.....	19
4.4. ФБ управления портом (COM_CONTROL) .....	20
4.4.1. Инициализация порта (шаг INITIALIZE) .....	22
4.4.2. Ожидание управляющего сигнала (шаг WAITING_FOR_SIGNAL).....	23
4.4.3. Открытие порта (шаг OPEN_PORT) .....	24
4.4.4. Закрытие порта (шаг CLOSE_PORT).....	28
4.5. ФБ опроса модуля (MV110_8A_DCON) .....	30
4.5.1. Подготовка запроса (шаг CREATE_REQUEST).....	32
4.5.2. Отправка запроса (шаг SEND_REQUEST) .....	36
4.5.3. Получение ответа (шаг RECEIVE_RESPONSE).....	37
4.5.4. Организация задержки (шаг RESPONSE_DELAY) .....	42
4.5.5. Завершение цикла опроса (шаг POLLING_CYCLE_ENDS).....	42
4.6. Программа опроса (PLC_PRG).....	44
5. Пример опроса счетчика СЭТ-4ТМ.03М .....	46
5.1. Формулировка задачи.....	46

5.2. Описание протокола .....	47
5.3. Алгоритмизация задачи .....	49
5.4. ФБ управления портом (COM_CONTROL) .....	50
5.5. ФБ опроса счетчика (SET_4TM).....	50
5.5.1. Подготовка запроса на открытие канала (шаг CREATE_CHANNEL).....	53
5.5.2. Отправка запроса на открытие канала (шаг OPEN_CHANNEL).....	57
5.5.3. Получение ответа на запрос открытия канала (шаг RECEIVE_CHANNEL) .....	58
5.5.4. Организация задержки (шаг RESPONSE_DELAY_CHANNEL).....	60
5.5.5. Подготовка запроса на чтение данных (шаг CREATE_REQUEST).....	61
5.5.6. Отправка запроса на чтение данных (шаг SEND_REQUEST) .....	62
5.5.7. Получение ответа (шаг RECEIVE_RESPONSE).....	63
5.5.8. Организация задержки (шаг RESPONSE_DELAY) .....	66
5.5.9. Завершение цикла опроса (шаг POLLING_CYCLE_ENDS).....	67
5.6. Программа опроса (PLC_PRG).....	69
6. Рекомендации и замечания .....	71
Приложение.....	73
А. Листинг программы из п. 4.....	73
А.1. ФБ COM_CONTROL.....	74
А.2. ФБ MV110_8A_DCON.....	76
А.3. ФБ ANALYZE_DATA .....	80
А.4. Функция BYTE_TO_STRH .....	81
А.5. Функция _BUFFER_CLEAR .....	81
А.6. Перечисление COM_STATE .....	82
А.7. Перечисление DCON_STATE .....	82
Б. Листинг программы из п. 5.....	83
Б.1. ФБ COM_CONTROL.....	84
Б.2. ФБ SET_4TM.....	86
Б.3. Функция CRC_MG_GEN.....	92
Б.4. Функция _BUFFER_CLEAR .....	92
Б.5. Перечисление COM_STATE .....	93
Б.6. Перечисление SET_4TM_STATE .....	93

## 1. Цель и структура документа

Одной из ключевых задач сенсорных панельных контроллеров СПК является организация обмена данными с другими устройствами по последовательным интерфейсам RS232/485.

В значительном количестве случаев обмен осуществляется по стандартным промышленным протоколам (например, **Modbus RTU**, **CANopen**) 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 этом случае пользователю необходимо реализовать поддержку этого протокола на СПК с помощью системных библиотек, которые позволяют работать с последовательным портом напрямую. Решению подобных задач и посвящен данный документ.

**Необходимо отметить**, что работа с нестандартными протоколами подразумевает высокую квалификацию программиста, а также хорошее знание среды **CODESYS 3.5** и языка ST. Документ рекомендуется читать строго последовательно.

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.

## 2. Основные сведения об обмене данными по последовательным интерфейсам

### 2.1. Общие принципы организации обмена

В рамках данного документа рассмотрены протоколы, работающие поверх последовательного интерфейса RS232/485 и основанные на архитектуре **Master – Slave** (ведущий – ведомый).

Эта 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.). При этом 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 при необходимости отправить запрос slave-устройству;
3. Чтение из порта. Производится при 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

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), а 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 этом случае каждый передаваемый байт содержит код символа из [таблицы ASCII](#). В данном случае 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-устройства оно 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. 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 же 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, необходимо 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 (либо сразу, либо в следующем цикле опроса).

**3.** Ответ был поврежден при передаче (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** нажмите кнопку **Добавить** и выберите библиотеку **CAA SerialCom**, расположенную в папке **Intern/CAA/System**.

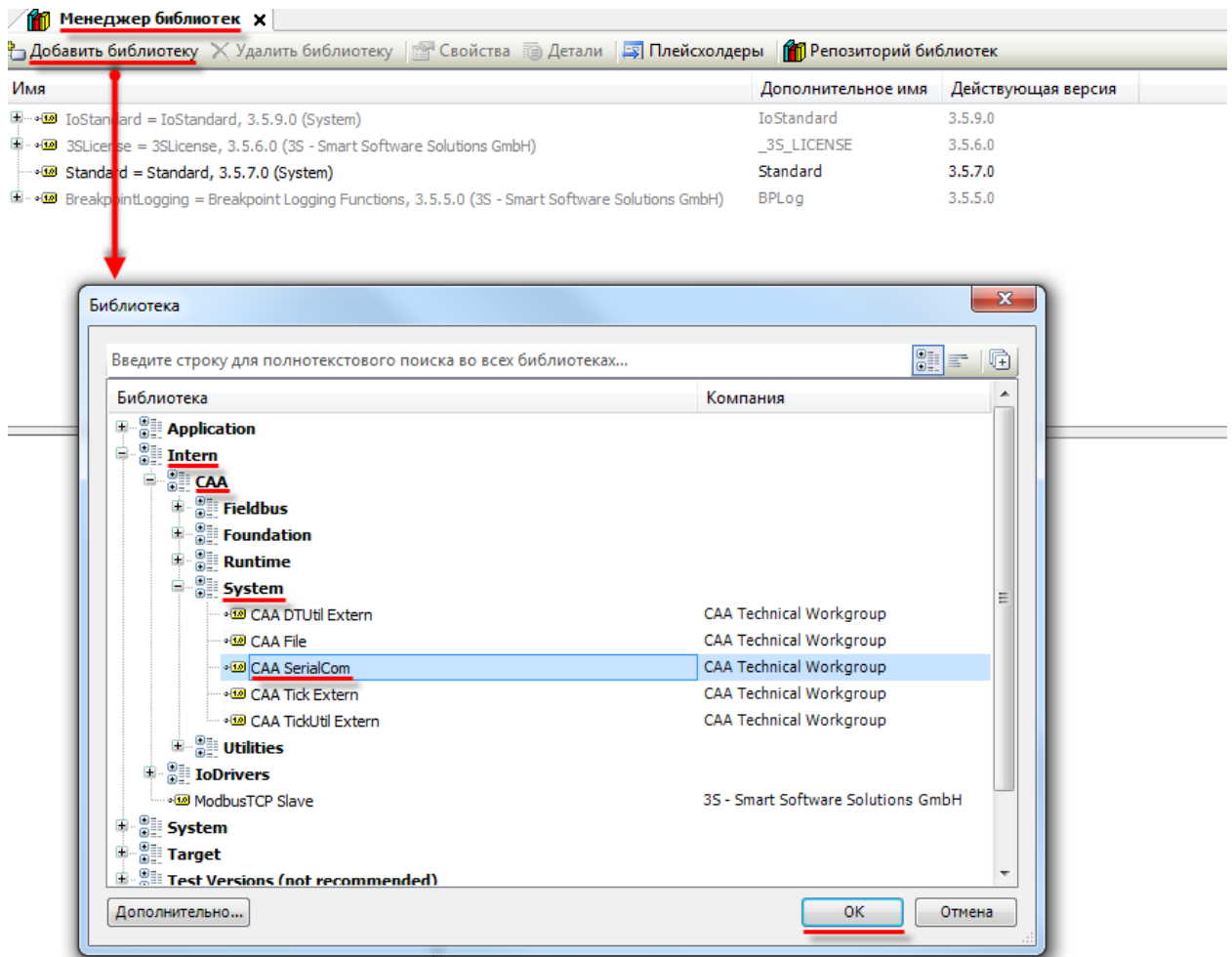


Рис. 3.1. Добавление библиотеки **CAA SerialCom** в проект **CODESYS**

**Обратите внимание**, что при объявлении экземпляров ФБ библиотеки необходимо 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](#).

Название	Тип данных	Описание
udiParameterID	UDINT	ID или имя параметра COM-порта. Параметры определены в списке глобальных констант <b>CAA_Parameter_Constants</b> .
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**.

Название	ID	Тип данных	Описание
<i>Основные параметры</i>			
udiPort	16#1	UDINT	Номер COM-порта. <b>Обратите внимание</b> , что номер COM-порта в CODESYS может не совпадать с номером на корпусе прибора. Подробнее см. в документе <b>СПК. Modbus</b> .
udiStopBits	16#2	UDINT	Число стоп бит. Возможные значения определены в перечислении <b>COM.STOPBIT</b> : <b>ONESTOPBIT</b> – один стоп бит; <b>ONE5STOPBITS</b> – 1.5 стоп бита; <b>TWOSTOPBITS</b> – 2 стоп бита.
udiParity	16#3	UDINT	Режим контроля четности. Возможные значения определены в перечислении <b>COM.PARITY</b> : EVEN – четный; ODD – нечетный; NONE – отсутствует.
udiBaudrate	16#4	UDINT	Скорость обмена. Возможные значения для СПК: 4800/9600/19200/38400/57600/115200;
udiTimeout	16#5	UDINT	Аппаратный таймаут COM-порта. <u>Не может быть изменен для СПК.</u>
udiBufferSize	16#6	UDINT	Размер буфера COM-порта. <u>Не может быть изменен для СПК.</u>
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. **Обратите внимание**, запрещается пытаться открыть уже открытый порт.

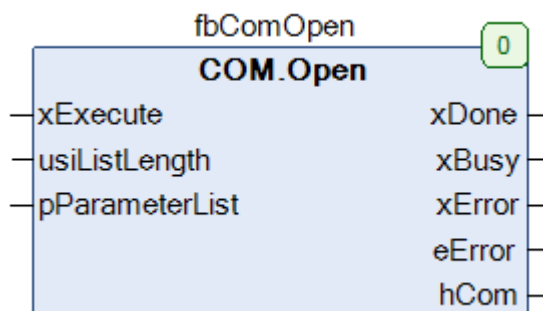


Рис. 3.2. Внешний вид ФБ **COM.Open** на языке CFC

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

### 3.5. ФБ COM.Write

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

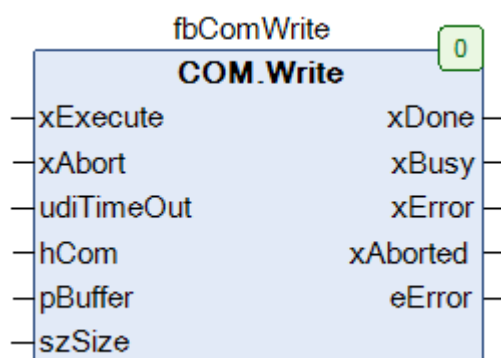


Рис. 3.3. Внешний вид ФБ **COM.Write** на языке CFC

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

### 3.6. ФБ COM.Read

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

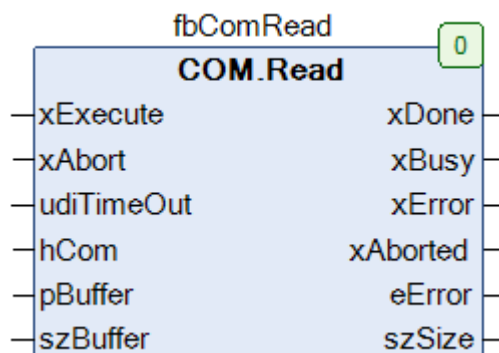


Рис. 3.4. Внешний вид ФБ **COM.Read** на языке **CFC**

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

### 3.7. ФБ COM.Close

Функциональный блок **COM.Close** используется для закрытия COM-порта. **Обратите внимание**, запрещается пытаться закрыть уже закрытый порт.

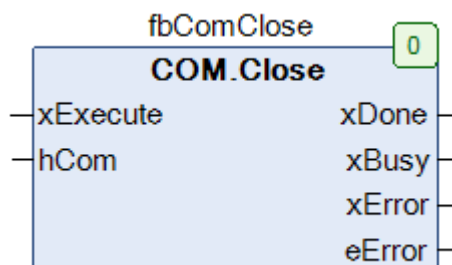


Рис. 3.5. Внешний вид ФБ **COM.Close** на языке **CFC**

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

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

### 4.1. Формулировка задачи

Рассмотрим пример опроса модуля [MB110-8A](#) по протоколу **DCON** с помощью контроллера СПК207.03. Модуль подключен к порту COM2, его сетевые настройки приведены в табл. 4.1.

Табл. 4.1. Сетевые настройки модуля **MB110-8A**

Параметр	Значение
<i>COM-порт СПК, к которому подключен модуль</i>	COM2
Адрес модуля	1
Скорость обмена	115200
Количество бит данных	8
Контроль четности	Отсутствует
Количество стоп-бит	1

Пример создан в среде **CODESYS 3.5 SP7 Patch4** и подразумевает запуск на **СПК207.03.CS(-WEB)** с таргет-файлом **3.5.4.20 (023)**.

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 RS485. На примере протокола DCON мы постараемся описать 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

При этом 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 мы будем реализовывать 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;

**(данные)** – 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. При возникновении 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).

При синтаксически неверном запросе 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.

Рассмотрим пример запроса к модулю с адресом **1**:

**#01[CHK]\$R**

Отметим, что адрес модуля записывается в виде двух HEX символов. Соответственно, для модуля с адресом **74** запрос бы выглядел как **#4A[CHK]\$R**.

Рассчитаем контрольную сумму для запроса к модулю с адресом **1**. Он будет представлять 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:

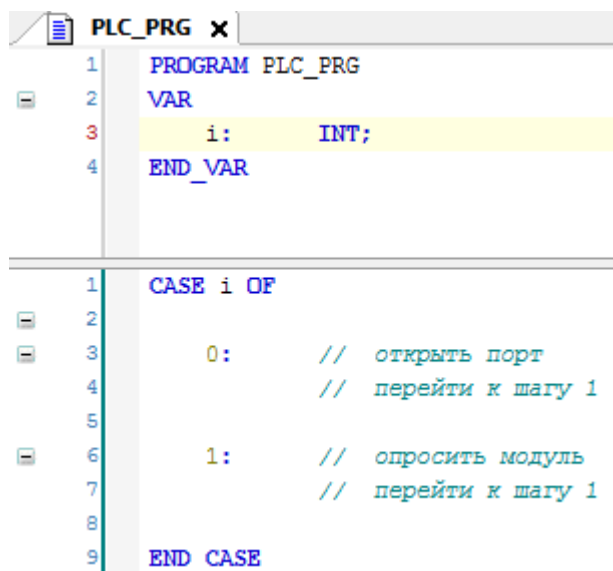
**>+100.23+34.050+124.56+07.331-101.45+1038.9-50.501+05.880[CHK]\$R**

Итак, мы знаем структуру запроса, структуру ответа и методику расчета контрольной суммы. Этой информации достаточно, чтобы начать реализацию протокола обмена в **CODESYS**.

### 4.3. Алгоритмизация задачи

Как уже упоминалось в [п. 2.1](#), процесс обмена данными через последовательный порт можно представить в виде циклически выполняемого алгоритма. На языке ST для реализации подобных алгоритмов в большинстве случаев используется оператор условного выбора CASE.

Алгоритм решаемой в примере задачи можно представить следующим образом:



```
PLC_PRG x
1 PROGRAM PLC_PRG
2 VAR
3   i: INT;
4 END_VAR
5
6 CASE i OF
7   0: // открыть порт
8     // перейти к шагу 1
9
10  1: // опросить модуль
11     // перейти к шагу 1
12
13 END_CASE
```

Рис. 4.1. Алгоритмизация задачи опроса модуля

Таким образом, при запуске программы будет однократно произведено открытие порта, после чего начнется циклический опрос модуля.

Соответственно, наша задача сводится к написанию кода, который будет выполняться в шагах 0 и 1. Удобно будет 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 необходимо добавить в проект библиотеку [CAA SerialCom](#), поскольку ее функции и ФБ потребуются при реализации протокола.

#### 4.4. ФБ управления портом (COM\_CONTROL)

Создадим функциональный блок управления COM-портом с названием **COM\_CONTROL**. Он будет использоваться для открытия порта (для начала обмена) и его закрытия в случае возникновения ошибок или необходимости остановить обмен.

Попробуем более подробно сформулировать требования к его функционалу – это поможет 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.

Проанализировав требования, получим следующий список переменных ФБ:

```
COM_CONTROL x
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
```

Рис. 4.2. Объявление переменных ФБ **COM\_CONTROL**

**Обратите внимание**, что 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, а также экземпляры ФБ `COM.Open` и `COM.Close`, входящие в состав библиотеки `CAA SerialCom`.

Как и сам алгоритм опроса, работу с COM-портом удобно реализовать 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
```

Рис. 4.3. Алгоритм ФБ управления COM-портом

На шаге 0 будет происходить инициализация ФБ (под этим подразумевается 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 рис. 4.3 для обозначения шагов мы используем порядковые номера; такой подход лишен наглядности и затрудняет понимание алгоритма. Поэтому создадим **перечисление COM\_STATE**, которое позволит использовать имена для обозначения шагов (**Application – Добавление объекта – DUT – Перечисление**):

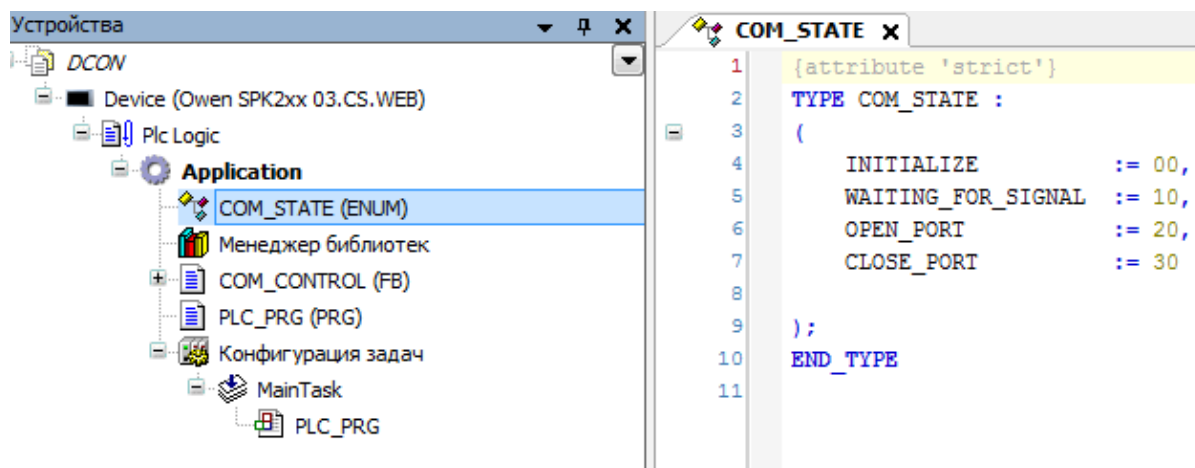


Рис. 4.4. Объявление перечисления `COM_STATE`

Для работы с перечислением нужно в переменных программы объявить его экземпляр. На самом деле, мы это уже сделали – см. переменную **eState** на рис. 4.2. Разрывы в номерах шагов позволяют при необходимости добавлять промежуточные шаги.

Теперь в операторе CASE можно использовать имена шагов, что явно повышает читаемость кода. Подобным приемом мы будем пользоваться и при разработке ФБ опроса модуля.

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

Рис. 4.5. Алгоритм ФБ управления COM-портом с использованием перечисления

Осталось написать код для каждого шага.

#### 4.4.1. 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
```

Рис. 4.6. Код шага **INITIALIZE**

#### 4.4.2. Ожидание управляющего сигнала (шаг WAITING\_FOR\_SIGNAL)

На шаге **WAITING\_FOR\_SIGNAL** будет происходить ожидание управляющего сигнала. При детектировании 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#FFFFFFFF THEN      // если порт еще не открыт (0) или (16#FFFFFFFF)...
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
26     // если получен сигнал закрытия порта...
27     IF xClose THEN
28         IF hCom>0 AND hCom<16#FFFFFFFF THEN     // если порт уже открыт...
29             eState := CLOSE_PORT;                // ...то переходим к шагу закрытия порта
30         ELSE                                     // в противном случае считаем, что порт уже закрыт, и тогда...
31             xClose := FALSE;                     // сбрасываем сигнал закрытия порта
32             xDone := FALSE;                       // сбрасываем флаг открытого порта
33             eState := INITIALIZE;                 // ...переходим к шагу инициализации порта
34         END_IF
35     END_IF
```

Рис. 4.7. Код шага **WAITING\_FOR\_SIGNAL**

При получении управляющего сигнала производится проверка дескриптора (handle) **hCom**. Значение **0** соответствует закрытому порту, значение **16#FFFFFFFF** – ошибке открытия порта. Значения между **0** и **16#FFFFFFFF** соответствуют открытому порту. Соответственно, не имеет смысла открывать уже открытый порт и закрывать закрытый – в этом случае необходимо 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](#).

### 4.4.3. Открытие порта (шаг OPEN\_PORT)

На шаге **OPEN\_PORT** необходимо открыть порт с заданными параметрами (напомним, эти 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
```

Рис. 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**. Создадим для ФБ **COM\_CONTROL** три действия – **SETTINGS** (задание настроек порта), **OPEN** (открытие порта) и **CLOSE** (закрытие порта).

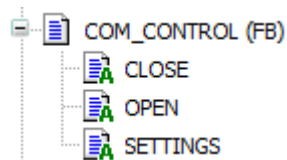


Рис. 4.9. Действия ФБ **COM\_CONTROL**



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 := uiBaudrate;
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);
```

Рис. 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, преобразуем их 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:

```
COM_CONTROL.OPEN x
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;
```

Рис. 4.11. Код действия **OPEN**

В этом 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. Поделив размер массива параметров на размер одного параметра, получим текущее число параметров. При этом 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. Это приведет к открытию COM-порта с заданными настройками.

После этого необходимо получить значение дескриптора (handle), присвоив его выходной переменной **hCom** ФБ **COM\_CONTROL**.

Мы рассмотрели код действий **SETTINGS** и **OPEN**, которые выполняются на шаге **OPEN\_PORT**. Теперь разберем остальной код, который выполняется на этом шаге.

```

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

```

Рис. 4.12. Код шага **OPEN\_PORT**

После выполнения действия **OPEN** необходимо проанализировать его результат. Если порт успешно открыт (у **fb\_COMopen** выход **xDone = TRUE**), то сбрасываем управляющий сигнал, останавливаем работу **fb\_COMopen**, взводим флаг успешного открытия порта (выход **xDone** ФБ **COM\_CONTROL**) и переходим на шаг инициализации порта ([INITIALIZE](#)).

Если при открытии порта произошла ошибка (у ФБ **fb\_COMopen** выход **xError = TRUE**), то взводим флаг ошибки открытия порта, останавливаем работу **fb\_COMopen**, сбрасываем флаг успешного открытия порта (выход **xDone** ФБ **COM\_CONTROL**) и переходим на шаг инициализации порта ([INITIALIZE](#)).

Осталось рассмотреть код шага закрытия порта.

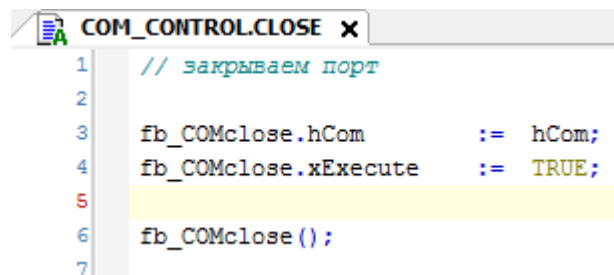
#### 4.4.4. Закрытие порта (шаг CLOSE\_PORT)

При попадании на шаг **CLOSE\_PORT** необходимо закрыть порт. Это может потребоваться при необходимости прекращения обмена, а также 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
```

Рис. 4.13. Код шага **CLOSE\_PORT**

Как и при открытии порта, сам код закрытия поместим в действие.



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

Рис. 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** необходимо проанализировать его результат. Если порт успешно открыт (у **fb\_COMclose** выход **xDone = TRUE**), то сбрасываем управляющий сигнал, сбрасываем флаг успешного открытия порта, обнуляем значение дескриптора (handle) на выходе ФБ **COM\_CONTROL**, останавливаем работу **fb\_COMclose** и переходим на шаг инициализации порта.

Если при закрытии порта произошла ошибка (у ФБ **fb\_COMclose** выход **xError = TRUE**), то вводим флаг ошибки закрытия порта и останавливаем работу **fb\_COMclose**. При этом мы остаемся на том же шаге, т.е. в следующем цикле опять будет произведена попытка закрытия порта и так до тех пор, пока не произойдет успешное закрытие порта.

Итак, мы создали ФБ, с помощью которого сможем открывать порт с заданными параметрами и, при необходимости, закрывать его.

Теперь мы можем вернуться к нашей основной программе [PLC\\_PRG](#), объявить экземпляр ФБ **COM\_CONTROL** и написать код для шага 0:

```
PLC_PRG x
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

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

Рис. 4.15. Алгоритмизация задачи опроса модуля с кодом шага 0

В рамках примера мы не будем рассматривать закрытие порта, поэтому в явном виде присвоим переменной **xOpen** значение **TRUE**. При старте программы шаг 0 будет выполнен однократно, поскольку возвращение на него не происходит.

Что ж, порт с заданными настройками открыт – теперь мы можем отправлять в него данные и считывать ответы. Пора переходить непосредственно к реализации протокола DCON.

Листинг ФБ **COM\_CONTROL** приведен в [приложении А.1](#).

## 4.5. ФБ опроса модуля (MV110\_8A\_DCON)

Создадим функциональный блок опроса модуля **MB110-8A** по протоколу **DCON** с названием **MV110\_8A\_DCON**. Для начала требуется определиться с набором входных и выходных переменных.

Вполне очевидно, что нам будет нужен вход для управления блоком, с помощью которого мы будем начинать и прекращать его работу. Поскольку в общем случае к контроллеру может быть подключено несколько модулей, нам также понадобится вход для задания адреса опрашиваемого модуля. Еще один вход займет дескриптор (handle) COM-порта.

К выходным переменным будут относиться измеренные значения каналов модуля, флаг окончания цикла опроса и флаги ошибок обмена.

Мы сразу приведем список локальных переменных ФБ. Необходимость их объявления будет поясняться по мере описания принципов работы блока.

```
MV110_8A_DCON x
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;     // таймаут опроса модуля (время ожидания ответа)
48   c_tDelay:        TIME:=T#50MS;   // задержка перед отправкой следующего запроса
49 END_VAR
50
```

Рис. 4.16. Объявление переменных ФБ MV110\_8A\_DCON

Как и в предыдущем пункте, мы будем реализовывать пошаговый алгоритм через оператор CASE – и для этого опять воспользуемся **перечислением**, которое позволит использовать имена в качестве названий шагов. Экземпляр перечисления назовем **eState**.

```
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
```

Рис. 4.17. Объявление перечисления **DCON\_STATE**

Запишем 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
```

Рис. 4.18. Алгоритм опроса модуля

На шаге **CREATE\_REQUEST** будет происходить формирование запроса к модулю.

На шаге **SEND\_REQUEST** этот 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.

Следует отметить, что необходимость наличия шага **RESPONSE\_DELAY** зависит от особенностей конкретного опрашиваемого устройства. Некоторые устройства удерживают линию связи фиксированное время после ответа, в связи с чем необходимо организовывать задержку перед отправкой следующего запроса. В случае опроса модуля MB110-8A такая задержка не обязательна, но поскольку целью примера является демонстрация общих принципов организации обмена, то необходимо рассмотреть такую ситуацию.

Теперь можно начать писать код для каждого шага опроса. Как и в предыдущем пункте, законченные фрагменты кода мы будем оформлять в виде **действий**:

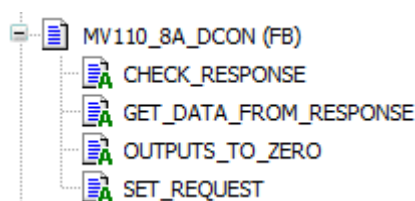


Рис. 4.19. Действия ФБ **MV110\_8A\_DCON**

#### 4.5.1. Подготовка запроса (шаг **CREATE\_REQUEST**)

На шаге **CREATE\_REQUEST** происходит очистка буферов данных и подготовка запроса.

```

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

```

Рис. 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 модулю 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](http://oscat.de), а также на [сайте компании Овен](#) в разделе **CODESYS V3/Библиотеки**. Библиотека OSCAT имеет открытые исходные коды, поэтому во многих случаях удобнее копировать ее функции и ФБ в пользовательский проект (вместо добавления через **Менеджер библиотек**).



Функция **\_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. 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

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;

```

Рис. 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**. Напомним, что 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

```

Рис. 4.22. Код действия **SET\_REQUEST**

Напомним, что со строковыми переменными можно работать как с массивом байт – это является очень удобным при наполнении буфера.

Структура запроса к модулю выглядит следующим образом:

	Старт символ	Адрес модуля	Контрольная сумма	Стоп символ
<b>Размер</b>	1 байт	2 байта	2 байта	1 байт
<b>Значение</b>	#	переменная	переменная	CR
<b>Код ASCII</b>	16#23	переменная	переменная	16#D

Входная переменная **byAdress** имеет тип **BYTE** – поэтому необходимо конвертировать ее в два HEX символа. Для этого воспользуемся функцией **BYTE\_TO\_STRH** из библиотеки **OSCAT** (см. рис. 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 мы вычисляем контрольную сумму в явном виде, но при необходимости (например, при опросе нескольких различных модулей) ее расчет можно вынести в отдельную функцию.

```

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
19 (* calculate high order hex value *)
20 temp := SHR(in,4);
21
22 (* convert value to hex character *)
23 IF temp <= 9 THEN temp := temp + 48; ELSE temp := temp + 55; END_IF;
24
25 (* write first character to output string *)
26 PT^ := temp;
27
28 (* calculate low order hex value *)
29 temp := in AND 2#00001111;
30
31 IF temp <= 9 THEN temp := temp + 48; ELSE temp := temp + 55; END_IF;
32
33 (* increment pointer and write low order character *)
34 pt := pt + 1;
35 pt^ := temp;
36
37 (* set pointer at the end of the output string and close the string with 0 *)
38 pt := pt + 1;
39 pt^ := 0;

```

Рис. 4.23. Код функции **BYTE\_TO\_STRH** из библиотеки **OSCAT**

Еще раз вернемся к коду шага **CREATE\_REQUEST**.

Мы рассмотрели процесс очистки буферов и формирования запроса. Кроме этого, 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
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
```

## 4.5.2. 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_ION(IN:=FALSE, PT:=c_timeout);         // сбрасываем таймер
44         fb_ION(IN:=TRUE,  PT:=c_timeout);        // запускаем таймер таймаута
45
46         eState := RECEIVE_RESPONSE;              // переходим к шагу получения ответа от модуля
47     END_IF
48
```

Рис. 4.25. Код шага **SEND\_REQUEST**

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

Входными переменными 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 воспользуемся функцией **LEN**, которая возвращает длину строки.

После вызова блока отправки запроса необходимо проанализировать его результат. Если при отправке произошла ошибка (у ФБ **fb\_COMwrite** выход **xError = TRUE**), то взводим флаг ошибки и переходим к шагу завершения опроса.

Если запрос успешно отправлен (у **fb\_COMwrite** выход **xDone = TRUE**), то завершаем работу блока, сбрасываем флаг ошибки, запускаем таймер таймаута (он потребуется нам на следующем шаге) и переходим к шагу [RECEIVE\\_RESPONSE](#).

### 4.5.3. 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 := ADR(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
```

Рис. 4.26. Код шага **RECEIVE\_RESPONSE**

В этом 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 удобнее использовать оператор **SIZEOF**, возвращающий размер переменной. В данном случае функция **LEN** уже не нужна, т.к. размер буфера приема в большинстве случаев является статической величиной, зависящей только от размера соответствующей переменной.

Необходимо 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 может «застрять» на шаге получения ответа, чего происходить не должно. Для этого используется контроль таймаута опроса – по истечению заданного времени мы выходим из текущего шага в независимости от факта получения ответа. Таймер таймаута был запущен нами в конце предыдущего шага, поэтому на данном шаге достаточно просто вызвать его с теми же аргументами. Время таймаута определяется с помощью константы **c\_tTimeout**.

После вызова блока получения ответа необходимо проанализировать его результат. Если при получении произошла ошибка (у **fb\_COMread** выход **xError = TRUE**), то взводим флаг ошибки и переходим к шагу завершения опроса.

Если ответ успешно получен (у **fb\_COMread** выход **xDone = TRUE**), то сбрасываем флаг ошибки таймаута, проверяем корректность ответа и завершаем 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   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

```

Рис. 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. Чтобы корректно обработать эту ситуацию, мы будем делать следующее: пока не сработал таймер таймаута, каждый полученный фрагмент ответа будем «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 при этом позиция стоп символа больше позиции старт символа, то можно сделать вывод что в буфере находится ответ от подчиненного устройства. При этом в ряде специфических случаев он может находиться не в начале буфера, поэтому следует вырезать его, ориентируясь на позиции старт- и стоп символа. Вырезанный ответ помещаем в тот же самый буфер и взводим флаг получения корректного ответа (**xCorrectAnswer**).

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

*вырезаем из буфера полученный ответ (от старт символа до стоп символа)*

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

Вернемся к шагу **RECEIVE\_RESPONSE** (см. рис. 4.26). После выполнения кода действия **CHECK\_RESPONSE** необходимо завершить работу **ФБ COM.Read**. Если взведен флаг корректного ответа (**xCorrectAnswer**), то переходим к действию **GET\_DATA\_FROM\_RESPONSE**, в котором будем разбирать полученный ответ.

```

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 // разбираем ответ
23 fb_ANALYZE_DATA(sData:=sReadData);
24
25 // присваиваем значения на выход блока
26 rMV110_8A_output1:=fb_ANALYZE_DATA.arValue[1];
27 rMV110_8A_output2:=fb_ANALYZE_DATA.arValue[2];
28 rMV110_8A_output3:=fb_ANALYZE_DATA.arValue[3];
29 rMV110_8A_output4:=fb_ANALYZE_DATA.arValue[4];
30 rMV110_8A_output5:=fb_ANALYZE_DATA.arValue[5];
31 rMV110_8A_output6:=fb_ANALYZE_DATA.arValue[6];
32 rMV110_8A_output7:=fb_ANALYZE_DATA.arValue[7];
33 rMV110_8A_output8:=fb_ANALYZE_DATA.arValue[8];
34
35
36 ELSE // если CRC запроса и ответа не совпадают...
37   xWrongCRC:=TRUE; // ...выставляем флаг некорректной CRC
38 END_IF

```

Рис. 4.28. Код действия **GET\_DATA\_FROM\_RESPONSE**

В первую очередь необходимо проверить контрольную сумму. Рассчитаем ее по известному нам алгоритму (просуммируем ASCII коды всех символов, расположенных перед контрольной суммой). Сравним ее с контрольной суммой ответа, предварительно преобразовав в строковый вид с помощью функции **BYTE\_TO\_STRH** из библиотеки **Oscat** (из описания протокола мы знаем, что контрольная сумма находится в двух байтах перед стоп символом).

Если рассчитанная контрольная сумма совпадает с полученной, то можно переходить к разбору ответа. Предварительно необходимо 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**.

```

1 FUNCTION_BLOCK ANALYZE_DATA
2 VAR_INPUT
3   sData:          STRING;          // анализируемая строка
4 END_VAR
5 VAR_CONSTANT
6   c_iMaxCanal:   INT := 8;        // кол-во значений в строке
7 END_VAR
8 VAR_OUTPUT
9   arValue:       ARRAY [1..c_iMaxCanal] OF REAL; // массив выделенных из строки REAL значений
10 END_VAR
11 VAR
12   aiSignPos:     ARRAY [1..c_iMaxCanal+1] OF INT; // массив позиций знаков строки
13   i, j:          INT;             // счетчики для циклов FOR
14 END_VAR
15
16 j:=1; // сбрасываем счетчик
17
18 FOR i:=1 TO FIND(sData, '$R') DO // проверяем каждый символ строки..
19   IF sData[i]=16#2B OR sData[i]=16#2D OR sData[i]=16#0D THEN // ...не является ли он '+', '-' или '$R'
20     aiSignPos[j]:=i; // и если является, то записываем его позицию
21     j:=j+1;
22   END_IF
23 END_FOR
24
25 FOR i:=1 TO c_iMaxCanal-1 DO // вырезаем из строки все значения, кроме последнего
26   arValue[i]:=STRING_TO_REAL(MID(sData, aiSignPos[i+1] - aiSignPos[i], aiSignPos[i] + 1)); // значение - это то, что находится между соседними знаками
27 END_FOR
28
29 // вырезаем последнее значение - не забываем, что между предпоследним и последним знаком...
30 //...кроме этого значения лежат еще два символа CRC, которые не нам не нужны
31 arValue[c_iMaxCanal]:=STRING_TO_REAL(MID(sData, aiSignPos[c_iMaxCanal+1] - aiSignPos[c_iMaxCanal] - 2, aiSignPos[i] + 1));
32
33

```

Рис. 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)

В этом случае надо ориентироваться на специальные символы-разделители. В контексте чисел такими знаками являются «+» и «-». Мы можем сказать, что значение – это то, что расположено между двумя любыми знаками, включая первый знак.

Поэтому сначала необходимо найти в строке эти знаки и запомнить их позиции. Помимо этого, потребуется найти стоп символ – он понадобится нам для определения последнего значения в ответе.

Рассмотрим это на примере, в котором в ответе содержится 3 значения (1.2, -33.44 и 555.666). В реальном ответе от модуля будет 8 значений фиксированного размера, но принцип останется неизменным.

Найдем позиции знаков и стоп символа (они соответственно, равны 2, 6, 12, 22). Теперь мы знаем, что первое значение занимает символы 2-5, второе – 6-11, а третье 12-19 (мы знаем, что между последним значением и стоп символом также находятся два символа CRC). Осталось только 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



Вернемся на уровень выше, в действие **GET\_DATA\_FROM\_RESPONSE** (см. рис. 4.28). Итак, мы проанализировали ответ и получили массив из 8-и значений типа **REAL**. Теперь присвоим их выходным переменным ФБ **MV110\_8A\_DCON**.

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

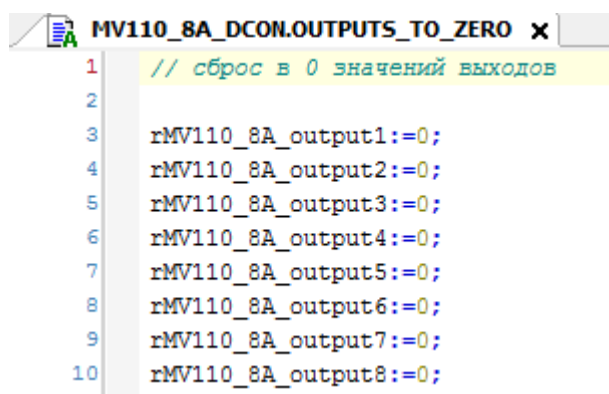
Еще раз перейдем на уровень выше и вернемся в код шага **RECEIVE\_RESPONSE** (см. рис. 4.27).

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

Рис. 4.30. Фрагмент кода шага **RECEIVE\_RESPONSE**

После анализа ответа переходим к шагу задержки перед следующим циклом опроса.

Если же сработал таймер таймаута (основной причиной этого является отсутствие ответа от модуля в течение заданного интервала времени), то выставляем флаг ошибки по таймауту, выполняем действие **OUTPUTS\_TO\_ZERO** (сброс выходных переменных ФБ **MV110\_8A\_DCON** со считанными с модуля значениями) и переходим 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;
```

Рис. 4.31. Код действия **OUTPUTS\_TO\_ZERO**

#### 4.5.4. Организация задержки (шаг RESPONSE\_DELAY)

На шаге **RESPONSE\_DELAY** происходит запуск таймера задержки перед следующим циклом опроса. Как уже упоминалось, эта 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
```

Рис. 4.32. Код шага **RESPONSE\_DELAY**

#### 4.5.5. Завершение цикла опроса (шаг POLLING\_CYCLE\_ENDS)

На шаге **POLLING\_CYCLE\_ENDS** необходимо 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
```

Рис. 4.33. Код шага **POLLING\_CYCLE\_ENDS**

Итак, мы написали код для всех шагов опроса модуля. Теперь вернемся к тому моменту, когда мы записывали алгоритм работы ФБ **MV110\_8A\_DCON**, и вспомним, 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 этом случае потребуется также выполнить ряд действий (в операторе **ELSE**) – сбросить флаги завершения опроса и получения корректного ответа, обнулить значения выходных переменных типа **REAL** (см. рис. 4.31) и перейти на шаг **CREATE\_REQUEST** (чтобы при следующем запуске ФБ начать его выполнение с первого шага).

```

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

```

Рис. 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](#).

## 4.6. Программа опроса (PLC\_PRG)

Итак, мы создали ФБ управления COM-портом и ФБ опроса модуля **MB110-8A** по протоколу **DCON**. Теперь осталось вызвать их в программе **PLC\_PRG**. Как уже упоминалось при алгоритмизации задачи, сам процесс опроса тоже удобнее разбить на шаги.

```
PLC_PRG x
1 PROGRAM PLC_PRG
2 VAR
3     fb_COMcontrol:      COM_CONTROL;
4     fb_MV110_8A_DCON:  MV110_8A_DCON;
5
6     i:                  INT;
7
8     xOpen:              BOOL;
9     xClose:             BOOL;
10 END_VAR
11
12 CASE i OF
13
14     0: // открываем COM-порт COM2 (номер порта в CODESYS на +1 больше)
15
16         xOpen:=TRUE;
17
18         fb_COMcontrol
19         (
20             xOpen      := xOpen,
21             xClose     := xClose,
22             uiPortNumber := 3,
23             udiBaudrate := 115200,
24             uiParity    := COM.PARITY.NONE,
25             uiByteSize  := 8,
26             uiStopBits  := COM.STOPBIT.ONESTOPBIT
27         );
28
29         IF fb_COMcontrol.xDone THEN
30             i:=1;
31         END_IF
32
33     1: // опрашиваем модуль MB110-8A по протоколу DCON
34
35         fb_MV110_8A_DCON
36         (
37             xEnable     := fb_COMcontrol.xDone,
38             hCom        := fb_COMcontrol.hCom,
39             byAddress    := 1
40         );
41
42         IF fb_MV110_8A_DCON.xDone THEN
43             i:=1;
44         END_IF
45
46 END_CASE
```

Рис. 4.35. Код программы **PLC\_PRG**

При запуске программы однократно выполняется код шага 0, что приводит к открытию COM-порта СПК с заданными настройками при помощи 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 нашем случае других устройств нет, поэтому мы остаемся на текущем шаге и продолжаем 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 рис. 4.36 приведен скриншот программы в процессе работы. Как можно заметить, значение второго канала равно **99999** – что соответствует возникновению в канале исключительной ситуации (например, обрыву датчика). Это является особенностью модуля, а не протокола **DCON**.

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

Рис. 4.36. Считанные значения каналов модуля

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

### 5.1. Формулировка задачи

Рассмотрим пример опроса счетчика [СЭТ-4ТМ.03М](#) по нестандартному протоколу с помощью контроллера **СПК207.04.WEB**. В качестве опрашиваемого параметра используем текущее напряжение по фазе А (**Ua**). Счетчик подключен к порту **COM3**, его сетевые настройки приведены в табл. 5.1.

Табл. 5.1. Сетевые настройки счетчика **СЭТ-4ТМ**

Параметр	Значение
<i>COM-порт СПК, к которому подключен модуль</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 SP7 Patch4** и подразумевает запуск на **СПК207.04.CS(-WEB)**.

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 описание представляет собой 200-страничный документ; ниже мы приведем 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 данном примере мы рассматриваем работу 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 необходимо 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 2-го 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<sub>a</sub>) 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> )

Рис. 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(ДМ)
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		-	-	-	-	+

Рис. 5.2. Фрагмент таблицы **RWRI** из описания протокола

Итак, мы знаем структуру запроса, структуру ответа и методику расчета контрольной суммы. Этой информации достаточно, чтобы начать реализацию протокола обмена в **CODESYS**.



### 5.3. Алгоритмизация задачи

Как уже упоминалось в [п. 2.1](#), процесс обмена данными через последовательный порт можно представить в виде циклически выполняемого алгоритма. На языке ST для реализации подобных алгоритмов в большинстве случаев используется оператор условного выбора CASE.

Алгоритм решаемой в примере задачи можно представить следующим образом:

```
PLC_PRG x
1 PROGRAM PLC_PRG
2 VAR
3   i: INT;
4 END_VAR

1 CASE i OF
2
3   0: // открыть порт
4     // перейти к шагу 1
5
6   1: // отправить запрос на открытие канала
7     // перейти к шагу 2
8
9   2: // отправить запрос на получение данных
10    // перейти к шагу 1
11
12 END_CASE
```

Рис. 5.3. Алгоритмизация задачи опроса устройства

Таким образом, при запуске 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, наша задача сводится к написанию кода, который будет выполняться в шагах 0 и 1. Удобно будет 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 необходимо добавить в проект библиотеку [CAA SerialCom](#), поскольку 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, перейдите к 5.5.

#### 5.5. ФБ опроса счетчика (SET\_4TM)

Создадим функциональный блок опроса счетчика **СЭТ-4ТМ.03М** по нестандартному протоколу с названием **SET\_4TM**. Для начала требуется определиться с набором входных и выходных переменных.

Вполне очевидно, что нам потребуется вход для управления блоком, с помощью которого мы будем начинать и прекращать его работу. Поскольку в общем случае к контроллеру может быть подключено несколько счетчиков, нам также понадобится вход для задания адреса опрашиваемого счетчика. Еще один вход займет дескриптор (handle) COM-порта.

К выходным переменным будут относиться измеренные значение напряжение по фазе А ( $U_a$ ), флаг окончания опроса модуля и флаги ошибок обмена.

Мы сразу приведем список локальных переменных ФБ. Необходимость их объявления будет поясняться по мере описания принципов работы блока.

```

1  FUNCTION_BLOCK SET_4TM
2
3  VAR_INPUT
4      xEnable:          BOOL;           // сигнал опроса счетчика
5      hCom:             COM.CAA.HANDLE; // дескриптор COM-порта
6      byAddress:        BYTE;          // адрес опрашиваемого счетчика
7  END_VAR
8
9  VAR_OUTPUT
10     xDone:             BOOL;          // флаг окончания текущего цикла опроса счетчика
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     rUa:               REAL;          // значение напряжения Ua, считанное с счетчика
17 END_VAR
18
19 VAR
20     eState:            SET_4TM_STATE; // текущий шаг опроса счетчика
21
22     fb_COMwrite:       COM.Write;     // ФБ отправки запроса
23     fb_COMread:        COM.Read;      // ФБ получения ответа
24
25     szWriteSize:       COM.CAA.SIZE;  // размер запроса в байтах
26
27     wCRC:              WORD;          // CRC
28     xOpenChannel:      BOOL;          // флаг успешного открытия канала связи с счетчиком
29     xCorrectAnswer:    BOOL;          // флаг корректного (с верной CRC) ответа от модуля
30     iLenBuff:          INT;           // счетчик для склеивания ответов
31
32     fb_TON:            TON;           // таймер
33
34     i:                 INT;           // счетчик для цикла FOR
35
36     abyWriteData:      ARRAY [0..255] OF BYTE; // буфер запроса
37     abyReadBuff:       ARRAY [0..255] OF BYTE; // буфер приема
38     abyReadData:       ARRAY [0..255] OF BYTE; // буфер данных от счетчика
39
40     uConvertToReal:    _4BYTES_TO_REAL; // экземпляр объединения для получения REAL из 4-х байт
41     uWordTo2Bytes:     _WORD_TO_2BYTES; // экземпляр объединения для конвертации WORD в два байта
42 END_VAR
43
44 VAR CONSTANT
45     c_tTimeout:        TIME:=T#1S;    // таймат опроса модуля (время ожидания ответа)
46     c_tDelay:          TIME:=T#50MS;   // задержка перед отправкой следующего запроса
47 END_VAR

```

Рис. 5.4. Объявление переменных ФБ SET\_4TM

Как и в предыдущем пункте, мы будем реализовывать пошаговый алгоритм через оператор CASE – и для этого опять воспользуемся перечислением, которое позволит использовать имена в качестве названий шагов. Экземпляр перечисления назовем **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

```

Рис. 5.5. Объявление перечисления SET\_4TM\_STATE

Запишем алгоритм опроса счетчика в общем виде:

```
1  IF xEnable THEN           // если получен сигнал опроса модуля...
2
3  CASE eState OF           // ...то запускаем 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
```

Рис. 5.6. Алгоритм опроса счетчика

На шаге **CREATE\_CHANNEL** будет происходить формирование запроса на открытие канала связи.

На шаге **OPEN\_CHANNEL** этот 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.

Следует отметить, что необходимость наличия шага **RESPONSE\_DELAY** зависит от особенностей конкретного опрашиваемого устройства. Некоторые устройства удерживают линию связи фиксированное время после ответа, в связи с чем необходимо организовывать задержку перед отправкой следующего запроса. В случае опроса счетчика СЭТ-4ТМ.03М такая задержка не обязательна, но поскольку целью примера является демонстрация общих принципов организации обмена, то необходимо рассмотреть такую ситуацию.

Теперь можно начать писать код для каждого шага опроса. Как и в предыдущем пункте, законченные фрагменты кода мы будем оформлять в виде действий:

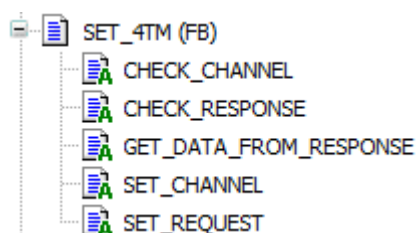


Рис. 5.7. Действия ФБ SET\_4ТМ

### 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      _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
  
```

Рис. 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, 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](http://oscat.de), а также на [сайте компании Овен](#) в разделе **CODESYS V3/Библиотеки**. Библиотека OSCAT имеет открытые исходные коды, поэтому во многих

случаях удобнее копировать ее функции и ФБ в пользовательский проект (вместо добавления через **Менеджер библиотек**).

Функция **\_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. 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

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;
```

Рис. 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 хотя протокол является бинарным, 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

```

Рис. 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. Создадим ФБ **CRC\_MB\_GEN**, который будет выполнять эту задачу.

```

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

```

Рис. 5.11. Код ФБ **CRC\_MB\_GEN**

У внимательного читателя, вероятно, возникнет вопрос – почему бы не воспользоваться готовым блоком расчета CRC из библиотеки **OSCAT**? Да, действительно, в OSCAT'е 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.

Рассчитанная контрольная сумма представляет собой переменную типа **WORD** – но чтобы записать ее в массив, необходимо представить ее в виде двух байт. Для этого создадим объединение **\_WORD\_TO\_2BYTES** (его экземпляр **uWordTo2Bytes** мы уже объявили в переменных ФБ **SET\_4TM**, см. рис. 5.4).

```

1  TYPE _WORD_TO_2BYTES :
2  UNION
3      wValue:WORD;
4      abyWord: ARRAY [0..1] OF BYTE;
5  END_UNION
6  END_TYPE

```

Рис. 5.12. Объявление объединения **\_WORD\_TO\_2BYTES**

Последнее, что необходимо сделать в действии **SET\_CHANNEL** – указать число байт запроса, присвоив соответствующее значение переменной **szWriteSize**.

Вернемся к коду шага **CREATE\_CHANNEL**.

Мы рассмотрели процесс очистки буферов и формирования запроса. Кроме этого, 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
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

```

Рис. 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 на 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
```

Рис. 5.14. Код шага **SEND\_REQUEST**

В этом 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**). Напомним, размер буфера мы указали в явном виде в коде действия **SET\_CHANNEL** (см. рис. 5.10).

После вызова блока отправки запроса необходимо проанализировать его результат. Если при отправке произошла ошибка (у ФБ **fb\_COMwrite** выход **xError = TRUE**), то взводим флаг ошибки и переходим к шагу завершения опроса.

Если запрос успешно отправлен (у **fb\_COMwrite** выход **xDone = TRUE**), то завершаем работу блока, сбрасываем флаг ошибки, запускаем таймер таймаута (он потребуется нам на следующем шаге) и переходим к шагу [RECEIVE\\_CHANNEL](#).

### 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
```

Рис. 5.15. Код шага **RECEIVE\_CHANNEL**

В этом 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 удобнее использовать оператор **SIZEOF**, возвращающий размер переменной.

Необходимо 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 «застрять» на шаге получения ответа, чего происходить не должно. Для этого используется контроль таймаута опроса – по истечению заданного времени мы выходим из текущего шага в независимости от факта получения ответа. Таймер таймаута был запущен нами в конце предыдущего шага, поэтому на данном шаге достаточно просто вызвать его с теми же аргументами. Время таймаута определяется с помощью константы **c\_tTimeout**.

После вызова блока получения ответа необходимо проанализировать его результат. Если при получении произошла ошибка (у **fb\_COMread** выход **xError = TRUE**), то взводим флаг ошибки и переходим к шагу завершения опроса.

Если ответ успешно получен (у **fb\_COMread** выход **xDone = TRUE**), то сбрасываем флаг ошибки таймаута, проверяем корректность ответа и завершаем работу блока.

Проверка корректности ответа вынесена в действие **CHECK\_CHANNEL**:

```

1  IF fb_COMread.szSize>0 THEN                                // если из порта были считана порция данных...
2
3
4  FOR i:=0 TO ANY_TO_INT(fb_COMread.szSize)-1 DO
5      abyReadData[iLenBuff + 1]:=abyReadBuff[1];           // ...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:=2); // расчет CRC ответа (2 - 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]=byAddress AND abyReadData[1]=0 AND abyReadData[2]=uWordTo2Bytes.abyWord[0] AND abyReadData[3]=uWordTo2Bytes.abyWord[1] THEN
16         xWrongCRC := FALSE; // ...то сбрасываем флаг неправильной CRC
17         xOpenChannel := TRUE; // взводим флаг получения корректного ответа
18     ELSE // в противном случае...
19         xWrongCRC := TRUE; // ...взводим флаг неправильной CRC
20     END_IF
21 END_IF

```

Рис. 5.16. Код действия **CHECK\_CHANNEL**

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

После этого необходимо проверить корректность ответа. Из описания протокола мы знаем, что в случае успешного открытия канала, счетчик отвечает пакетом со следующей структурой:

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

Соответственно, нам необходимо 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 воспользуемся ФБ **CRC\_MB\_GEN** и объединением **\_WORD\_TO\_2BYTES** (см. рис. 5.16).

Если все условия выполняются, то мы делаем вывод, что канал связи успешно открыт, сбрасываем флаг некорректной CRC и взводим флаг открытия канал (**xOpenChannel**). В противном случае, взводим флаг некорректной контрольной суммы (для упрощения мы не рассматриваем ситуацию, когда ответ содержит код ошибки и корректную контрольную сумму).

Вернемся к шагу **RECEIVE\_CHANNEL** (см. рис. 5.17). После выполнения кода действия **CHECK\_CHANNEL** необходимо завершить работу ФБ **COM.Read**. Если взведен флаг корректного ответа (**xOpenChannel**), то переходим к шагу [RESPONSE\\_DELAY\\_CHANNEL](#).

Если же сработал таймер таймаута (основной причиной этого является отсутствие ответа от модуля в течение заданного интервала времени), то выставляем флаг ошибки по таймауту, сбрасываем значение счетчика в 0 и переходим к шагу [RESPONSE\\_DELAY](#).

```

IF fb_COMread.xDone THEN // если фБ чтения из порта завершил работу...
  xTimeout:=FALSE; // сбрасываем флаг таймаута
  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 таймаута...
  xTimeout := TRUE; // взводим флаг ошибки по таймауту
  rUa := 0; // сбрасываем значения выходов блока
  eState := RESPONSE_DELAY; // переходим к шагу задержки перед следующим циклом опроса
END_IF

```

Рис. 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. Как уже упоминалось, эта 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 [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

```

Рис. 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
```

Рис. 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](#).

```
SET_ATM.SET_REQUEST x
1 // подготавливаем запрос на получение данных
2
3 abyWriteData[0]:=byAddress; // адрес счетчика
4 abyWriteData[1]:=16#08; // 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.aboWord[0]; // CRC запроса
16 abyWriteData[6]:=uWordTo2Bytes.aboWord[1]; //
17
18 szWriteSize:=7; // число байт запроса с учетом CRC
```

Рис. 5.20. Код действия **SET\_REQUEST**

Вернемся к коду шага **CREATE\_REQUEST**. Помимо формирования запроса, необходимо вызвать таймер задержки перед следующим запросом, запущенный на шаге **RESPONSE\_DELAY\_CHANNEL**. После его срабатывания перейдем 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
113     fb_COMwrite // запускаем ФБ записи в порт
114     (
115     xExecute := TRUE,
116     hCom := hCom,
117     pBuffer := ADR(abyWriteData),
118     szSize := szWriteSize
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_ION(IN:=FALSE, PT:=c_tTimeout); // сбрасываем таймер таймаута
134     fb_ION(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
```

Рис. 5.21. Код шага **SEND\_REQUEST**

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

Входными переменными блока являются сигнал его запуска (**xExecute**), дескриптор (handle) COM-порта (**hCom**), адрес буфера запроса (**pBuffer**) и размер буфера (**szWriteSize**). Напомним, размер буфера мы указали в явном виде в коде действия **SET\_REQUEST** (см. рис. 5.20).

После вызова блока отправки запроса необходимо проанализировать его результат. Если при отправке произошла ошибка (у ФБ **fb\_COMwrite** выход **xError = TRUE**), то взводим флаг ошибки и переходим к шагу завершения опроса.

Если запрос успешно отправлен (у **fb\_COMwrite** выход **xDone = TRUE**), то завершаем работу блока, сбрасываем флаг ошибки, запускаем таймер таймаута (он потребуется нам на следующем шаге) и переходим к шагу [RECEIVE\\_RESPONSE](#).

На следующем шаге мы ожидаем получить ответ от счетчика, поэтому нам необходимо обнулить переменную **iLenBuff** (она используется для склеивания фрагментов ответа от счетчика) и очистить буфер приема и буфер полученных данных (напомним, мы уже делали это на шаге **CREATE\_CHANNEL** – но с того момента в буфер попал ответ счетчика на запрос открытие канала, поэтому необходимо 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
```

Рис. 5.22. Код шага **RECEIVE\_RESPONSE**

В этом 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 удобнее использовать оператор **SIZEOF**, возвращающий размер переменной.

Необходимо 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, чего происходить не должно. Для этого используется контроль таймаута опроса – по истечению заданного времени мы выходим из текущего шага в независимости от факта получения ответа. Таймер таймаута был запущен нами в конце предыдущего шага, поэтому на данном шаге достаточно просто вызвать его с теми же аргументами. Время таймаута определяется с помощью константы **c\_timeout**.

После вызова блока получения ответа необходимо проанализировать его результат. Если при получении ответа произошла ошибка (у **fb\_COMread** выход **xError = TRUE**), то взводим флаг ошибки и переходим к шагу **RESPONSE\_DELAY**.

Если ответ успешно получен (у **fb\_COMread** выход **xDone = TRUE**), то сбрасываем флаг ошибки таймаута, проверяем корректность ответа и завершаем работу блока.

## Проверка корректности ответа вынесена в действие **CHECK\_RESPONSE**:

```

1  SET_4TM.CHECK_RESPONSE x
2  IF fb_COMread.szSize>0 THEN           // если из порта была считана порция данных...
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]=byAddress AND abyReadData[5]=uWordTo2Bytes.abyWord[0] AND abyReadData[6]=uWordTo2Bytes.abyWord[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         xWrongCRC      := TRUE;                  // ...взводим флаг неправильной CRC
21     END_IF
22 END_IF

```

Рис. 5.23. Код действия **CHECK\_RESPONSE**

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

После этого необходимо проверить корректность ответа. Из описания протокола мы знаем, что в случае успешного открытия канал, счетчик отвечает пакетом со следующей структурой:

Адрес счетчика	Значение Ua	CRC
1 байт	4 байта	2 байта

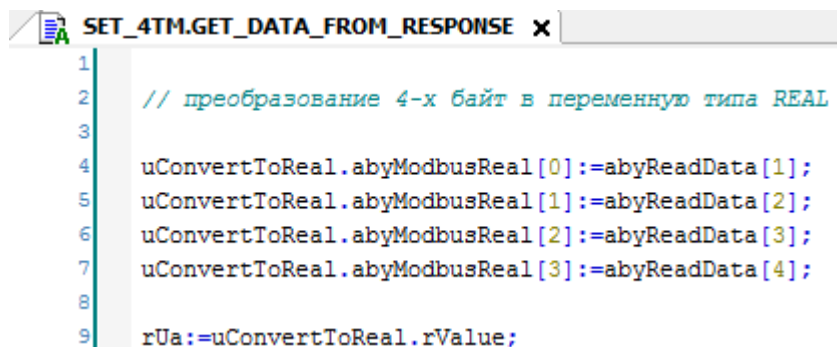
Соответственно, нам необходимо проверить, что нулевой байт ответа содержит адрес счетчика, а пятом и шестом и байтах содержится корректная CRC. Для этого снова воспользуемся ФБ **CRC\_MB\_GEN** и объединением **\_WORD\_TO\_2BYTES**.

Если все условия выполняются, то мы делаем вывод, что от счетчика получен ответ с корректной CRC, сбрасываем флаг некорректной CRC и взводим флаг открытия канал (**xCorrectAnswer**). В противном случае, взводим флаг некорректной CRC (для упрощения мы не рассматриваем ситуацию, когда ответ содержит код ошибки и корректную CRC).



Вернемся к коду шага **RECEIVE\_RESPONSE** (см. рис. 5.22).

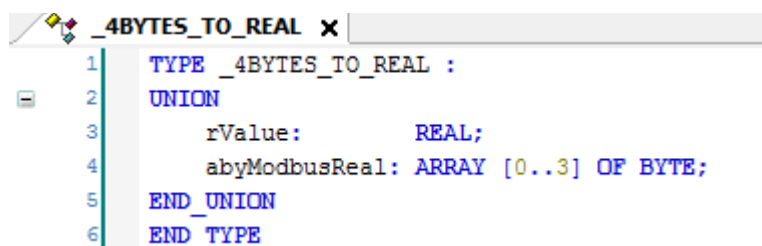
Если полученный от счетчика ответ корректен (**xCorrectAnswer=TRUE**), то можно приступить к его анализу. Анализ вынесен в действие **GET\_DATA\_FROM\_RESPONSE**.



```
1 // преобразование 4-х байт в переменную типа REAL
2
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;
```

Рис. 5.24. Код действия **GET\_DATA\_FROM\_RESPONSE**

Мы получаем от счетчика измеренное напряжение по фазе А ( $U_a$ ) в виде [значения с плавающей точкой в формате IEE 754](#), которое занимает 4 байта. Но в контроллере мы по очевидным причинам хотим видеть это значение как переменную типа **REAL**. Соответственно, необходимо преобразовать его. Проще всего это сделать с помощью **объединения**. Мы уже использовали объединение **\_WORD\_TO\_2BYTES** 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
```

Рис. 5.25. Объявление объединения **\_4BYTES\_TO\_REAL**

Экземпляр **uConvertToReal** этого объединения мы уже объявили в переменных ФБ **SET\_4TM**. Запишем в него байты считанного значения из буфера приема (заметим, что в нулевом байте содержится адрес счетчика, поэтому его не используем) и заберем из объединения значение типа **REAL**. Таким образом, в переменную **rUa** попадет измеренное значение напряжение по фазе А.

Вернемся 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
174
```

Рис. 5.26. Фрагмент кода шага **RECEIVE\_RESPONSE**

После анализа ответа переходим к шагу [RESPONSE\\_DELAY](#).

Если же сработал таймер таймаута (основной причиной этого является отсутствие ответа от модуля в течение заданного интервала времени), то выставляем флаг ошибки по таймауту, обнуляем значение переменной **rUa** и переходим к шагу [RESPONSE\\_DELAY](#).

### 5.5.8. Организация задержки (шаг **RESPONSE\_DELAY**)

На шаге **RESPONSE\_DELAY** происходит запуск таймера задержки перед следующим циклом опроса. Как уже упоминалось, эта 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](#).

```
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
```

Рис. 5.27. Код шага **RESPONSE\_DELAY**

### 5.5.9. Завершение цикла опроса (шаг POLLING\_CYCLE\_ENDS)

На шаге **POLLING\_CYCLE\_ENDS** необходимо 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
```

Рис. 5.28. Код шага **POLLING\_CYCLE\_ENDS**

Итак, мы написали код для всех шагов опроса модуля. Теперь вернемся к тому моменту, когда мы записывали алгоритм работы ФБ **SET\_4TM** и вспомним, 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 этом 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).

```

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
46         xDone := FALSE; // если сигнал опроса модуля пропал...
47                        // сбрасываем флаг окончания текущего цикла опроса счетчика
48
49         xOpenChannel := FALSE; // сбрасываем флаг открытого канала
50
51         xCorrectAnswer := FALSE; // сбрасываем флаг открытого канала
52
53         rUa := 0; // сбрасываем значения выходов блока
54
55         eState := CREATE_CHANNEL; // переходим к шагу подготовки запроса на открытие канала
56     END_IF

```

Рис. 5.29. Код ФБ SET\_4TM (без кода отдельных шагов)

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

## 5.6. Программа опроса (PLC\_PRG)

Итак, мы создали ФБ управления COM-портом и ФБ опроса счетчика **СЭТ-4ТМ.03М**. Теперь осталось вызвать их в программе **PLC\_PRG**. Как уже упоминалось при алгоритмизации задачи, сам процесс опроса тоже удобнее разбить на шаги.

```
PLC_PRG x
1 PROGRAM PLC_PRG
2 VAR
3     fb_COMcontrol:      COM_CONTROL;
4     fb_SET_4TM:        SET_4TM;
5
6     i:                  INT;
7
8     xOpen:              BOOL;
9     xClose:             BOOL;
10 END_VAR
11
12 CASE i OF
13
14     0: // открываем COM-порт COM3 (номер порта в CODESYS на +1 больше)
15
16         xOpen:=TRUE;
17
18         fb_COMcontrol
19         (
20             xOpen      := xOpen,
21             xClose     := xClose,
22             uiPortNumber := 4,
23             udiBaudrate := 9600,
24             uiParity    := COM.PARITY.ODD,
25             uiByteSize  := 8,
26             uiStopBits  := COM.STOPBIT.ONESTOPBIT
27         );
28
29         IF fb_COMcontrol.xDone THEN
30             i:=1;
31         END_IF
32
33     1: // опрашиваем счетчик СЭТ-4ТМ.03
34
35         fb_SET_4TM
36         (
37             xEnable     := fb_COMcontrol.xDone,
38             hCom        := fb_COMcontrol.hCom,
39             byAddress   := 200
40         );
41
42         IF fb_SET_4TM.xDone THEN
43             i:=1;
44         END_IF
45
46 END_CASE
```

Рис. 5.30. Код программы **PLC\_PRG**

При запуске программы однократно выполняется код шага 0, что приводит к открытию COM-порта СПК 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, в свою очередь, запускает новый цикл опроса.

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

На рис. 5.31 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

Рис. 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. Для того 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 – в этом случае необходимо 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; это же утверждение справедливо 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. При 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 необходимо 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) удобнее использовать одну WORD переменную, которая будет содержать код ошибки и перечисление с расшифровкой кодов.
4. В некоторых случаях требуется тщательная обработка ошибок обмена. Например, при отсутствии 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 программы 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 COM2 (номер порта в CODESYS на +1 больше)

        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

END_CASE
```

```

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 таймаута 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  
  
  sAdress:          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_timeout:        TIME:=T#1S;     // таймаут опроса модуля (время ожидания ответа)  
  c_delay:          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
    END_CASE

```

```

        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));
```



#### A.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;
```

#### A.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;
```

## A.6. Перечисление COM\_STATE

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

## A.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
```

## Б. Листинг программы 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 (номер порта в CODESYS на +1 больше)

        xOpen:=TRUE;

        fb_COMcontrol
        (
            xOpen              := xOpen,
            xClose             := xClose,
            uiPortNumber       := 4,
            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

END_CASE
```

```

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_timeout);
        fb_TON (IN:=TRUE, PT:=c_timeout);
        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
  END_CASE

```

```

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.abyWord[0];
abyWriteData[9]:=uWordTo2Bytes.abyWord[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.abyWord[0] AND abyReadData[3]=uWordTo2Bytes.abyWord[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.abyWord[0];
abyWriteData[6]:=uWordTo2Bytes.abyWord[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\_MG\_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
```