2016

СПК

Реализация нестандартных протоколов обмена

Руководство для продвинутых пользователей

Версия: 1.0
Дата: 06.12.2016

Оглавление

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 иди 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 протоколов – 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**.

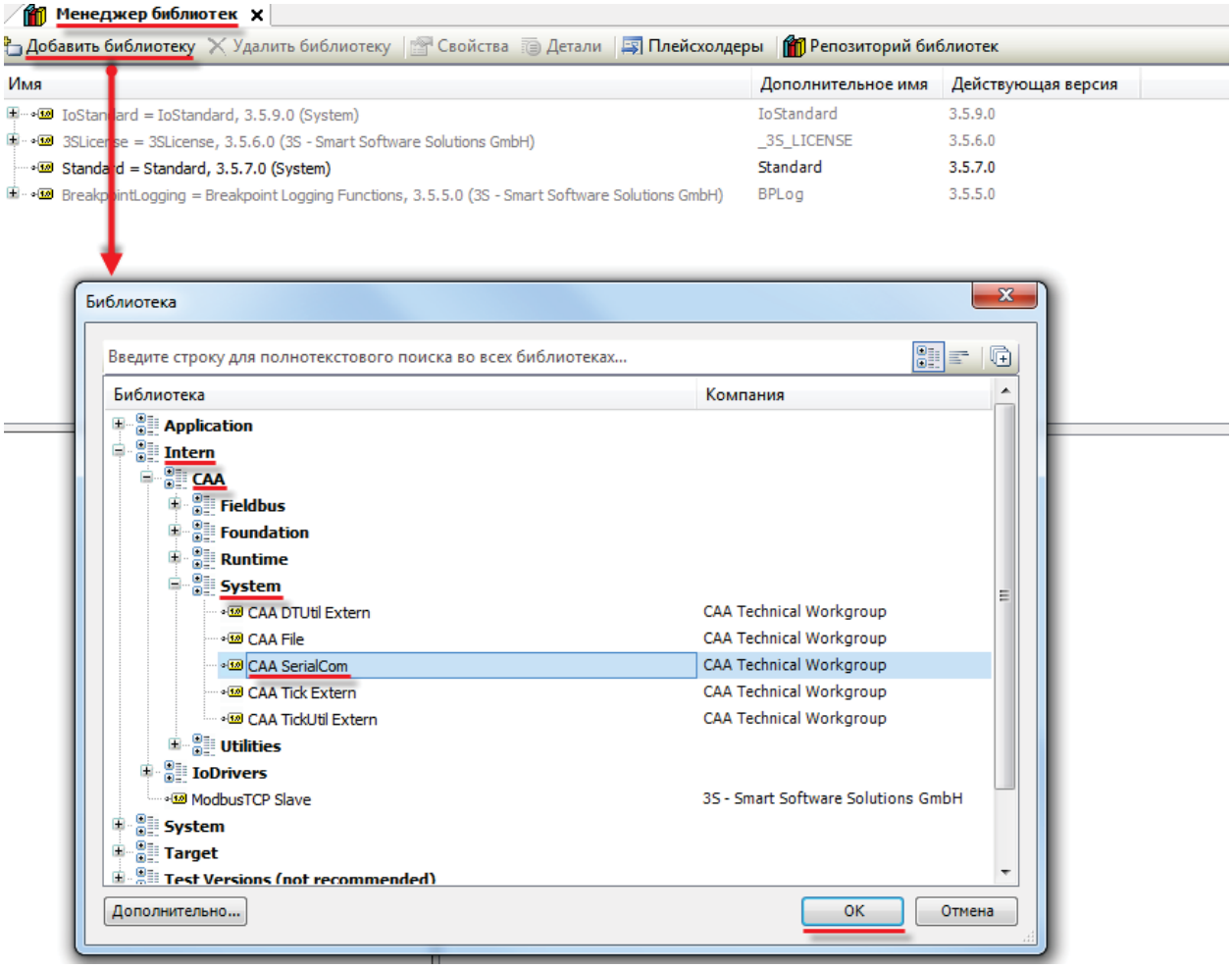

Рис. 3.1. Добавление библиотеки **CAA SerialCom** в проект **CODESYS**

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 **CAA_Parameter_Constants**.
udiValue	UDINT	Значение параметра.

3.3. Список глобальных констант CAA_Parameter_Constants

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

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

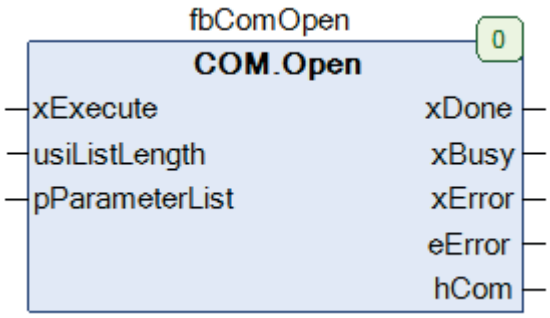

Рис. 3.2. Внешний вид ФБ **COM.Open** на языке **CFC**

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

3.5. ФБ COM.Write

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

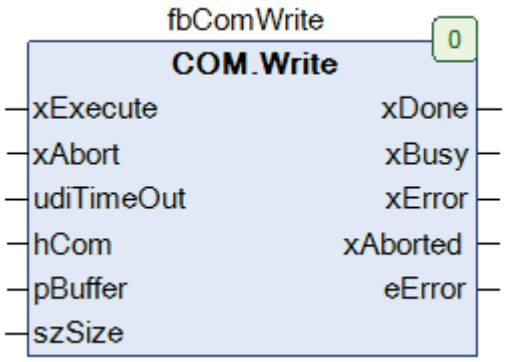

Рис. 3.3. Внешний вид ФБ **COM.Write** на языке **CFC**

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

3.6. ФБ COM.Read

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

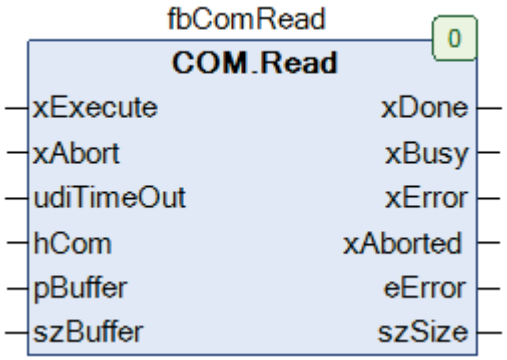

Рис. 3.4. Внешний вид ФБ **COM.Read** на языке **CFC**

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

3.7. ФБ COM.Close

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

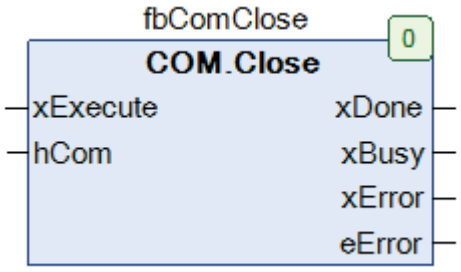

Рис. 3.5. Внешний вид ФБ **COM.Close** на языке **CFC**

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной.
hCom	CAA.HANDLE	Дескриптор (хэндл) закрываемого порта.
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** при возникновении ошибки.
eError	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**
COM-порт СПК, к которому подключен модуль	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 POU 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, исключающую 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

```
PROGRAM PLC_PRG
VAR
    i:      INT;
END_VAR
```

```
CASE i OF

    0:      //  открыть порт
            //  перейти к шагу 1

    1:      //  опросить модуль
            //  перейти к шагу 1

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

```
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:

```
CASE eState OF

    0:      // шаг инициализации блока

    1:      // шаг ожидания управляющего сигнала

    2:      // шаг открытия порта

    3:      // шаг закрытия порта


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**):

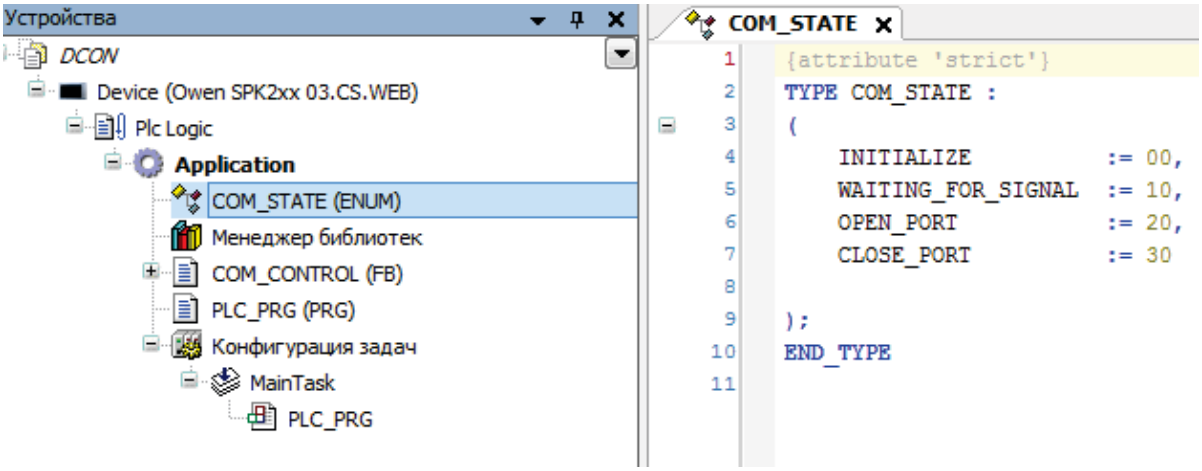

Рис. 4.4. Объявление перечисления **COM_STATE**

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 читабельность 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 входным 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).

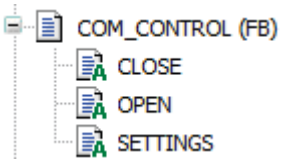

Рис. 4.9. Действия ФБ **COM_CONTROL**

Код действия **SETTINGS** выглядит следующим образом:

COM_CONTROL.SETTINGS

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

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

```
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 **pParamterList** 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

```
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 взводим 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

```
PROGRAM PLC_PRG
VAR
    fb_COMcontrol:          COM_CONTROL;

    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,
            xDone=>,
            hCom=>,
            xOpenError=>,
            xCloseError=>
            );

            IF fb_COMcontrol.xDone THEN
                i:=1;
            END_IF

    1:      //  опросить модуль
            //  перейти к шагу 1

END_CASE
```

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

```
FUNCTION_BLOCK MV110_8A_DCON

VAR_INPUT
    xEnable:            BOOL;                   // сигнал опроса модуля
    hCom:               COM.CAA.HANDLE;         // дескриптор COM-порта
    byAdress:           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_tTimeout:         TIME:=T#1S;             // таймаут опроса модуля (время ожидания ответа)
    c_tDelay:           TIME:=T#50MS;           // задержка перед отправкой следующего запроса
END_VAR
```

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**.

DCON_STATE

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

Рис. 4.17. Объявление перечисления **DCON_STATE**

Запишем алгоритм опроса модуля в общем виде:

```
IF xEnable THEN         // если получен сигнал опроса модуля...

    CASE eState OF          // ...то запускаем циклический опрос

        CREATE_REQUEST:     // шаг подготовки запроса

        SEND_REQUEST:       // шаг отправки запроса к модулю

        RECEIVE_RESPONSE:   // шаг получения ответа от модуля

        RESPONSE_DELAY:     // шаг задержки перед следующим циклом опроса

        POLLING_CYCLE_ENDS: // шаг окончания цикла опроса

    END_CASE

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**:

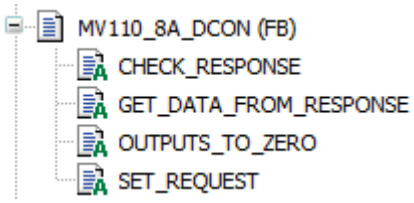

Рис. 4.19. Действия ФБ **MV110_8A_DCON**

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 , 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.

_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
```

```
(* this routine uses 32 bit access to gain speed *)
(* first access bytes till pointer is aligned for 32 bit access *)
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;
(* pointer is aligned, now copy 32 bits at a time *)
ptw := pt;
WHILE ptw < end32 DO (* *)
    ptw^ := 0;
    ptw := ptw + 4;
END_WHILE;
(* copy the remaining bytes in byte mode *)
pt := ptw;
WHILE pt < end DO
    pt^ := 0;
    pt := pt + 1;
END_WHILE;

_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.

MV110_8A_DCON.SET_REQUEST

```
// формирование запроса к модулю MB110-8A


sWriteData[0]:=16#23;                                   // ASCII: #

    sAdress:=BYTE_TO_STRH(byAdress);                    // ASCII: адрес модуля (два HEX символа)

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

    byCRC:=sWriteData[0]+sWriteData[1]+sWriteData[2];   //  расчет CRC
    sCRC:=BYTE_TO_STRH(byCRC);                          //  ASCII: CRC (два HEX символа)

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

sWriteData[5]:=16#D;                                    // ASCII: CR (перевод каретки)

```

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**
Размер	1 байт	2 байта	2 байта	1 байт
Значение	#	переменная	переменная	CR
Код ASCII	16#23	переменная	переменная	16#D

Входная переменная **byAdress** имеет тип **BYTE** – поэтому 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.

BYTE_TO_STRH

```
FUNCTION BYTE_TO_STRH : STRING(2)
VAR_INPUT
    IN : BYTE;
END_VAR
VAR
    temp : BYTE;
    PT : POINTER TO BYTE;
END_VAR

(*
version 1.3 29 mar. 2008
programmer  hugo
tested by       tobias
```

```
(* read pointer to output string *)
PT := ADR(BYTE_TO_STRH);
(* calculate high order hex value *)
temp := SHR(in,4);
(* convert value to hex character *)
IF temp <= 9 THEN temp := temp + 48; ELSE temp := temp + 55; END_IF;
(* write friat character to output string *)
PT^ := temp;
(* calculate low order hex value *)
temp := in AND 2#00001111;
IF temp <= 9 THEN temp := temp + 48; ELSE temp := temp + 55; END_IF;
(* increment pointer and wirte low order character *)
pt := pt + 1;
pt^ := temp;
(* set pointer at the end of the output string and close the string with 0 *)
pt := pt + 1;
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.

```
CREATE_REQUEST: // шаг подготовки запроса

                _BUFFER_CLEAR(PT:=ADR(sReadBuff),  SIZE:=SIZEOF((sReadBuff)));  // очищаем буфер приема
                _BUFFER_CLEAR(PT:=ADR(sReadData),  SIZE:=SIZEOF((sReadData)));  // очищаем буфер данных
                _BUFFER_CLEAR(PT:=ADR(sWriteData), SIZE:=SIZEOF((sWriteData))); // очищаем буфер запроса

                SET_REQUEST();                              // подготовка запроса

                xDone  := FALSE;                            // сброс флага окончания предыдущего цикла опроса

                eState := SEND_REQUEST;                     // переходим к шагу отправки запроса
```

Рис. 4.24. Код шага **CREATE_REQUEST**

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_TON(IN:=FALSE, PT:=c_tTimeout);        // сбрасываем таймер
44                        fb_TON(IN:=TRUE,  PT:=c_tTimeout);        // 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 переменным блока являются сигнал его запуска (**xExecute**), дескриптор (handle) COM-порта (**hCom**), адрес буфера запроса (**pBuffer**) и размер буфера (**szSize**). В строковом протоколе размер запроса определяется числом его символов – поэтому 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.

```
RECEIVE_RESPONSE: // шаг получения ответа от модуля



                fb_COMread                                  // запускаем ФБ чтения из порта
                (
                xExecute    :=  TRUE,
                hCom        :=  hCom,
                pBuffer     :=  ADR(sReadBuff),
                szBuffer    :=  SIZEOF(sReadBuff),
                );

                fb_TON();                                   // вызываем таймер таймаута

                IF fb_COMread.xError THEN                   // если при получении ответа произошла ошибка...
                    xReadError  := TRUE;                    // выставляем флаг ошибки получения ответа
                    eState      := POLLING_CYCLE_ENDS;      // переходим к шагу окончания опроса
                END_IF

                IF fb_COMread.xDone THEN                    // если ФБ чтения из порта завершил работу...
                    xTimeout:=FALSE;                        // сбрасываем флаг таймаута
                    CHECK_RESPONSE();                       // ищем символы начала и конца ответа
                    fb_COMread(xExecute:=FALSE);            // завершаем работу блока чтения из порта

                    IF xCorrectAnswer THEN                  // если в ответе есть и символ начала, и символ конца...
                        GET_DATA_FROM_RESPONSE();           // выделяем данные из ответа
                        eState := RESPONSE_DELAY;           // переходим к шагу задержки перед следующим циклом опроса
                    END_IF
                END_IF

                IF fb_TON.Q THEN                            // если сработал таймер таймаута...
                    xTimeout    := TRUE;                    // взводим флаг ошибки по таймауту
                    OUTPUTS_TO_ZERO();                      // сбрасываем значения выходов блока
                    eState      := RESPONSE_DELAY;          // переходим к шагу задержки перед следующим циклом опроса
                END_IF

```

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**:

MV110_8A_DCON.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.

MV110_8A_DCON.GET_DATA_FROM_RESPONSE

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

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**.

ANALYZE_DATA

```
FUNCTION_BLOCK ANALYZE_DATA
VAR_INPUT
    sData:              STRING;                     // анализируемая строка
END_VAR
VAR CONSTANT
    c_iMaxCanal:        INT := 8;                   // кол-во значений в строке
END_VAR
VAR_OUTPUT
    arValue:            ARRAY [1..c_iMaxCanal] OF REAL;     // массив выделенных из строки REAL значений
END_VAR
VAR
    aiSignPos:          ARRAY [1..c_iMaxCanal+1] OF INT;    // массив позиций знаков строки
    i,j:                INT;                        // счетчики для циклов FOR
END_VAR

j:=1;                                                                   // сбрасываем счетчик

FOR i:=1 TO FIND(sData,'$R') DO                                         // проверяем каждый символ строки..
    IF sData[i]=16#2B OR sData[i]=16#2D OR sData[i]=16#0D THEN          // ...не является ли он '+', '-' или '$R'
        aiSignPos[j]:=i;                                                // и если является, то записываем его позицию
        j:=j+1;
    END_IF
END_FOR

FOR i:=1 TO c_iMaxCanal-1 DO                                            // вырезаем из строки все значения, кроме последнего
    arValue[i]:=STRING_TO_REAL(MID(sData, aiSignPos[i+1] - aiSignPos[i],  aiSignPos[i] + 1));   // значение - это то, что находится между соседними знаками
END_FOR

// вырезаем последнее значение - не забываем, что между предпоследним и последним знаком...
//...кроме этого значения лежат еще два символа CRC, которые не нам не нужны

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

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              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. 4.30. Фрагмент кода шага **RECCEIVE_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

```
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).

```
IF xEnable THEN          // если получен сигнал опроса модуля...

    CASE eState OF       // ...то запускаем циклический опрос

        CREATE_REQUEST: // шаг подготовки запроса

         [11 lines]
        SEND_REQUEST: // шаг отправки запроса к модулю

                        [27 lines]
        RECEIVE_RESPONSE: // шаг получения ответа от модуля


                [33 lines]
        RESPONSE_DELAY: // шаг задержки перед следующим циклом опроса


         [5 lines]
        POLLING_CYCLE_ENDS: // шаг окончания цикла опроса
         [13 lines]
    END_CASE


ELSE                            // если сигнал опроса модуля пропал...
     xDone  := FALSE;           // сбрасываем флаг окончания текущего цикла опроса модуля

     xCorrectAnswer := FALSE;   // сбрасываем флаг корректного ответа

     OUTPUTS_TO_ZERO();         // сбрасываем значения выходов блока

     eState := CREATE_REQUEST;  // переходим к шагу подготовки запроса
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

```
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
```

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.**

PLC_PRG

Device.Application.PLC_PRG

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

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**
COM-порт СПК, к которому подключен модуль	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 POU 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 (Ua) 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
00h	Данные вспомогательных режимов измерения (аналогично 08h\11h)
01h	Зафиксированные данные вспомогательных режимов измерения (аналогично 08h/14h)
02h	Групповые данные вспомогательных режимов измерений (аналогично 08h\16h)
03h	Групповые зафиксированные данные вспомогательных режимов измерения (аналогично 08h\16h)

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		+	+	+	+	+
1h – напряжение: – фазное Uф				0 – Uф		0 – по фазе 1 1 – по фазе 1 2 – по фазе 2 3 – по фазе 3		+	+	+	+	+
– межфазное Uмф				1– Uмф		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

```
PROGRAM PLC_PRG
VAR
    i:  INT;
END_VAR
```

```
CASE i OF

    0:      // открыть порт
            // перейти к шагу 1

    1:      // отправить запрос на открытие канала
            // перейти к шагу 2

    2:      // отправить запрос на получение данных
            // перейти к шагу 1

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 (Ua), 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.

SET_4TM

```
FUNCTION_BLOCK SET_4TM

VAR_INPUT
    xEnable:            BOOL;                   // сигнал опроса счетчика
    hCom:               COM.CAA.HANDLE;         // дескриптор COM-порта
    byAdress:           BYTE;                   // адрес опрашиваемого счетчика
END_VAR

VAR_OUTPUT
    xDone:              BOOL;                   // флаг окончания текущего цикла опроса счетчика
    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

    rUa:                REAL;                   // значение напряжения Ua, считанное с счетчика
END_VAR

VAR
    eState:             SET_4TM_STATE;          // текущий шаг опроса счетчика

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

    szWriteSize:        COM.CAA.SIZE;           // размер запроса в байтах

    wCRC:               WORD;                   // CRC
    xOpenChannel:       BOOL;                   // флаг успешного открытия канала связи с счетчиком
    xCorrectAnswer:     BOOL;                   // флаг корректного (с верной CRC) ответа от модуля
    iLenBuff:           INT;                    // счетчик для склеивания ответов

    fb_TON:             TON;                    // таймер

    i:                  INT;                    // счетчик для цикла FOR

    abyWriteData:       ARRAY [0..255] OF BYTE; // буфер запроса
    abyReadBuff:        ARRAY [0..255] OF BYTE; // буфер приема
    abyReadData:        ARRAY [0..255] OF BYTE; // буфер данных от счетчика

    uConvertToReal:     _4BYTES_TO_REAL;        // экземпляр объединения для получения REAL из 4-х байт
    uWordTo2Bytes:      _WORD_TO_2BYTES;        // экземпляр объединения для конвертации WORD в два байта
END_VAR

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

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**.

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
```

Рис. 5.5. Объявление перечисления **SET_4TM_STATE**

Запишем алгоритм опроса счетчика в общем виде:

```
IF xEnable THEN          // если получен сигнал опроса модуля...

    CASE eState OF       // ...то запускаемый циклический опрос

        CREATE_CHANNEL:          // шаг подготовки запроса на открытие канала

        OPEN_CHANNEL:            // шаг отправки запроса на открытие канала

        RECEIVE_CHANNEL:         // шаг получения ответа на запрос открытия канала

        RESPONSE_DELAY_CHANNEL: // шаг задержки перед следующим запросом

        CREATE_REQUEST:          // шаг подготовки запроса на чтение данных

        SEND_REQUEST:            // шаг отправки запроса на чтение данных

        RECEIVE_RESPONSE:        // шаг получения ответа от счетчика

        RESPONSE_DELAY:          // шаг задержки перед следующим циклом опроса

        POLLING_CYCLE_ENDS:      // шаг окончания цикла опроса

    END_CASE

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:

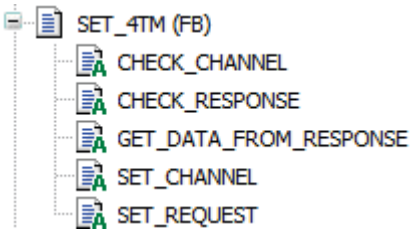

Рис. 5.7. Действия ФБ **SET_4TM**

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 , 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.

_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
```

```
(* this routine uses 32 bit access to gain speed *)
(* first access bytes till pointer is aligned for 32 bit access *)
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;
(* pointer is aligned, now copy 32 bits at a time *)
ptw := pt;
WHILE ptw < end32 DO (* *)
    ptw^ := 0;
    ptw := ptw + 4;
END_WHILE;
(* copy the remaining bytes in byte mode *)
pt := ptw;
WHILE pt < end DO
    pt^ := 0;
    pt := pt + 1;
END_WHILE;

_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.

SET_4TM.SET_CHANNEL

```
// подготавливаем запрос на открытие канала

abyWriteData[0]:=byAdress;                          // адрес счетчика
abyWriteData[1]:=16#01;                             // код функции
abyWriteData[2]:=16#30;                             // байты 2-7 - пароль "000000" в ASCII
abyWriteData[3]:=16#30;
abyWriteData[4]:=16#30;
abyWriteData[5]:=16#30;
abyWriteData[6]:=16#30;
abyWriteData[7]:=16#30;
//abyWriteData[8]:=16#03;
//abyWriteData[9]:=16#D2;

wCRC:=CRC_MB_GEN(pData:=ADR(abyWriteData), Size:=8);    // расчет CRC запроса (8 - число байт запроса без учета CRC)

uWordTo2Bytes.wValue:=wCRC;                         // преобразование CRC в два байта

abyWriteData[8]:=uWordTo2Bytes.abyWord[0];          // CRC запроса
abyWriteData[9]:=uWordTo2Bytes.abyWord[1];          //

szWriteSize:=10;                                    // число байт запроса с учетом CRC

```

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.

CRC_MB_GEN

```
FUNCTION CRC_MB_GEN : WORD
VAR_INPUT
    pData:  POINTER TO BYTE;        (* указатель на блок данных *)
    Size:   WORD;                   (* размер блока  данных *)
END_VAR
VAR
    Cnt:    BYTE;                   (* счетчик битов *)
END_VAR
```

```
(* расчет CRC для MODBUS RTU*)

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
```

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).

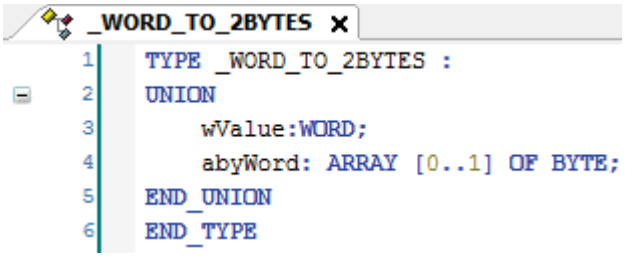

Рис. 5.12. Объявление объединения **_WORD_TO_2BYTES**

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**.**

```
CREATE_CHANNEL: // шаг подготовки запроса на открытие канала связи


                _BUFFER_CLEAR(PT:=ADR(abyReadBuff),  SIZE:=SIZEOF((abyReadBuff)));   // очищаем буфер приема
                _BUFFER_CLEAR(PT:=ADR(abyReadData),  SIZE:=SIZEOF((abyReadData)));   // очищаем буфер данных
                _BUFFER_CLEAR(PT:=ADR(abyWriteData), SIZE:=SIZEOF((abyWriteData)));  // очищаем буфер запроса

                xDone  := FALSE;                                // сброс флага окончания предыдущего цикла опроса

                SET_CHANNEL();                                  // подготовка запроса на открытие канала связи

                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.

```
OPEN_CHANNEL: // шаг отправки запроса на открытие канала связи


                fb_COMwrite                                 // запускаем ФБ записи в порт
                (
                xExecute :=  TRUE,
                hCom     :=  hCom,
                pBuffer  :=  ADR(abyWriteData),
                szSize   :=  szWriteSize
                );

                IF fb_COMwrite.xError THEN                  // если при отправке запроса произошла ошибка...
                    xWriteError := TRUE;                    // выставляем флаг ошибки отправки запроса
                    eState      := POLLING_CYCLE_ENDS;      // переходим к шагу окончания опроса
                END_IF


                IF fb_COMwrite.xDone THEN                   // если отправка запроса успешно завершена...
                    fb_COMwrite(xExecute:=FALSE);           // сбрасываем ФБ записи в порт

                    xWriteError := FALSE;                   // сбрасываем флаг ошибки отправки запроса

                    fb_TON(IN:=FALSE, PT:=c_tTimeout);      // сбрасываем таймер
                    fb_TON(IN:=TRUE,  PT:=c_tTimeout);      // запускаем таймер таймаута

                    eState  :=  RECEIVE_CHANNEL;            // переходим к шагу получения ответа от счетчика
                END_IF
```

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.

```
RECEIVE_CHANNEL: // шаг получения ответа от счетчика



                fb_COMread                                  // запускаем ФБ чтения в порт
                (
                xExecute :=  TRUE,
                hCom     :=  hCom,
                pBuffer  :=  ADR(abyReadBuff),
                szBuffer :=  SIZEOF(abyReadBuff)
                );

                fb_TON();                                   // вызываем таймер таймаута

                IF fb_COMread.xError THEN                   // если при получении ответа произошла ошибка...
                    xReadError  := TRUE;                    // выставляем флаг ошибки получения ответа
                    eState      := POLLING_CYCLE_ENDS;      // переходим к шагу окончания опроса
                END_IF


                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. 5.15. 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**:

SET_4TM.CHECK_CHANNEL

```
IF fb_COMread.szSize>0 THEN                              // если из порта были считана порция данных...

    FOR i:=0 TO ANY_TO_INT(fb_COMread.szSize)-1 DO
            abyReadData[iLenBuff + i]:=abyReadBuff[i];   // ...то приклеиваем их к предыдущей порции
        END_FOR                                          // ...и остаемся на этом же шаге, чтобы считать следующую порцию данных

        iLenBuff:=iLenBuff + ANY_TO_INT(fb_COMread.szSize); // следующая порция должна быть записана после предущей

        wCRC:=CRC_MB_GEN(pData:=ADR(abyReadData), Size:=2); // расчет CRC ответа (2 - число байт корректного ответа без учета CRC)

        uWordTo2Bytes.wValue:=wCRC;                      // преобразование CRC в два байта

        // если первый байт ответа совпадает с адресом счетчика, а расчитанная CRC соответствует CRC ответа...
        IF abyReadData[0]=byAdress AND abyReadData[1]=0 AND abyReadData[2]=uWordTo2Bytes.abyWord[0] AND abyReadData[3]=uWordTo2Bytes.abyWord[1] THEN
            xWrongCRC    :=  FALSE;  // ...то сбрасываем флаг неправильной CRC
            xOpenChannel :=  TRUE;   // взводим флаг получения корректного ответа
        ELSE                         // в противном случае...
            xWrongCRC    :=  TRUE;   // ...взводим флаг неправильной CRC
        END_IF
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_4TM.SET_REQUEST

```
1   // подготавливаем запрос на получение данных
2
3   abyWriteData[0]:=byAdress;                          // 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.abyWord[0];          // CRC запроса
16  abyWriteData[6]:=uWordTo2Bytes.abyWord[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.

```
SEND_REQUEST: // шаг отправки запроса на получение данных


                fb_COMwrite                                     // запускаем ФБ записи в порт
                (
                xExecute :=  TRUE,
                hCom     :=  hCom,
                pBuffer  :=  ADR(abyWriteData),
                szSize   :=  szWriteSize
                );


                IF fb_COMwrite.xError THEN                      // если при отправке запроса произошла ошибка...
                    xWriteError := TRUE;                        // выставляем флаг ошибки отправки запроса
                    eState      := POLLING_CYCLE_ENDS;          // переходим к шагу окончания опроса
                END_IF


                IF fb_COMwrite.xDone THEN                       // если отправка запроса успешено завершена...
                    fb_COMwrite(xExecute:=FALSE);               // сбрасываем ФБ записи в порт

                    xWriteError := FALSE;                       // сбрасываем флаг ошибки отправки запроса

                    fb_TON(IN:=FALSE, PT:=c_tTimeout);          // сбрасываем таймер таймаута
                    fb_TON(IN:=TRUE,  PT:=c_tTimeout);          // запускаем таймер таймаута

                    eState  :=  RECEIVE_RESPONSE;               // переходим к шагу получения ответа от модуля
                END_IF

                iLenBuff := 0;                                  // обнуляем счетчик для склеивания ответов

                _BUFFER_CLEAR(PT:=ADR(abyReadBuff), SIZE:=SIZEOF((abyReadBuff)));   // очищаем буфер приема
                _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 переменным 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.

```
RECEIVE_RESPONSE: // шаг получения ответа от модуля

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


                  fb_TON();                                   // вызываем таймер таймаута

                  IF fb_COMread.xError THEN                   // если при получении ответа произошла ошибка...
                      xReadError  := TRUE;                    // выставляем флаг ошибки получения ответа
                      eState      := POLLING_CYCLE_ENDS;      // переходим к шагу окончания опроса
                  END_IF

                  IF fb_COMread.xDone THEN                    // если ФБ чтения из порта завершил работу...
                      xTimeout:=FALSE;                        // сбрасываем флаг таймаута
                      CHECK_RESPONSE();                       // проверяем корректность ответа
                      fb_COMread(xExecute:=FALSE);            // сбрасываем ФБ чтения из порта

                      IF xCorrectAnswer THEN                  // если ответ корректен...
                          GET_DATA_FROM_RESPONSE();           // выделяем из него данные
                          eState  :=  RESPONSE_DELAY;         // переходим к шагу задержки перед следующим циклом опроса
                      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. 5.22. 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 «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 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**:

SET_4TM.CHECK_RESPONSE

```
IF fb_COMread.szSize>0 THEN                                 // если из порта была считана порция данных...

    FOR i:=0 TO ANY_TO_INT(fb_COMread.szSize)-1 DO
        abyReadData[iLenBuff + i]:=abyReadBuff[i];          // ...то приклеиваем их к предыдущей порции...
    END_FOR                                                 // ...и остаемся на этом же шаге, чтобы считать следующую порцию данных

        iLenBuff:=iLenBuff + ANY_TO_INT(fb_COMread.szSize); // следующая порция должна быть записана после предущей

        wCRC:=CRC_MB_GEN(pData:=ADR(abyReadData), Size:=5); // расчет CRC ответа (5 - число байт корректного ответа без учета CRC)

        uWordTo2Bytes.wValue:=wCRC;                         // преобразование CRC в два байта

        // если первый байт ответа совпадает с адресом счетчика, а расчитанная CRC соответствует CRC ответа...
        IF abyReadData[0]=byAdress AND abyReadData[5]=uWordTo2Bytes.abyWord[0] AND abyReadData[6]=uWordTo2Bytes.abyWord[1] THEN
            xWrongCRC       :=  FALSE;      // ...то сбрасываем флаг неправильной CRC
            xCorrectAnswer  :=  TRUE;       // взводим флаг получения корректного ответа
        ELSE                                // в противном случае...
            xWrongCRC       :=  TRUE;       // ...взводим флаг неправильной CRC
        END_IF
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 приступать к его анализу. Анализ вынесен в действие **GET_DATA_FROM_RESPONSE**.

SET_4TM.GET_DATA_FROM_RESPONSE

```
1
2   // преобразование 4-х байт в переменную типа REAL
3
4   uConvertToReal.abyModbusReal[0]:=abyReadData[1];
5   uConvertToReal.abyModbusReal[1]:=abyReadData[2];
6   uConvertToReal.abyModbusReal[2]:=abyReadData[3];
7   uConvertToReal.abyModbusReal[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 (Ua) в виде значения с плавающей точной 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**:

_4BYTES_TO_REAL

```
1   TYPE _4BYTES_TO_REAL :
2   UNION
3       rValue:         REAL;
4       abyModbusReal: 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
```

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 переменной ФБ и перейти на шаг CREATE_CHANNEL (чтобы при следующем запуске ФБ начать его выполнение с первого шага).

```
IF xEnable THEN         // если получен сигнал опроса счетчика...

    CASE eState OF      // ...то запускаем циклический опрос

        CREATE_CHANNEL: // шаг подготовки запроса на открытие канала связи

[11 lines]
        OPEN_CHANNEL: // шаг отправки запроса на открытие канала связи

                        [26 lines]
        RECEIVE_CHANNEL: // шаг получения ответа от счетчика


                        [34 lines]
        RESPONSE_DELAY_CHANNEL: // шаг задержки перед следующим запросом

                           [6 lines]
        CREATE_REQUEST: // шаг подготовки запроса данных
[12 lines]
        SEND_REQUEST: // шаг отправки запроса на получение данных

         [32 lines]
        RECEIVE_RESPONSE: // шаг получения ответа от модуля
         [29 lines]
        RESPONSE_DELAY: // шаг задержки перед следующим циклом опроса


         [5 lines]
        POLLING_CYCLE_ENDS: // шаг окончания цикла опроса

         [16 lines]
    END_CASE

ELSE                                    // если сигнал опроса модуля пропал...
    xDone          :=  FALSE;           // сбрасываем флаг окончания текущего цикла опроса счетчика

    xOpenChannel   :=  FALSE;           // сбрасываем флаг открытого канала

    xCorrectAnswer :=  FALSE;           // сбрасываем флаг открытого канала

    rUa            :=  0;               // сбрасываем значения выходов блока

    eState         :=  CREATE_CHANNEL; // переходим к шагу подготовки запроса  на открытие канала
END_IF
```

Рис. 5.29. Код ФБ **SET_4TM** (без кода отдельных шагов)

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

5.6. Программа опроса (PLC_PRG)

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

```
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
```

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** 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
```

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

```
IF xEnable THEN

	CASE eState OF

		CREATE_REQUEST:


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

			SET_REQUEST();

			xDone  := FALSE;

			eState := SEND_REQUEST;


		SEND_REQUEST:


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


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


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

				xWriteError := FALSE;

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

				eState      := RECEIVE_RESPONSE;
			END_IF


		RECEIVE_RESPONSE:


			fb_COMread

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

			fb_TON();

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

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

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

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


        RESPONSE_DELAY:



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

                eState := POLLING_CYCLE_ENDS;


        POLLING_CYCLE_ENDS:

                fb_TON();

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

                        xCorrectAnswer  := FALSE;

                        xDone           := TRUE;

                        eState := CREATE_REQUEST;
                END_IF

        END_CASE


ELSE
         xDone  := FALSE;

         xCorrectAnswer := FALSE;

         OUTPUTS_TO_ZERO();

         eState := CREATE_REQUEST;
END_IF
```

A.2.1. Действие SET_REQUEST

```
sWriteData[0]:=16#23;

        sAdress:=BYTE_TO_STRH(byAdress);

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

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

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

sWriteData[5]:=16#D;
```

A.2.2. Действие CHECK_RESPONSE

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


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

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

END_IF
```

A.2.3. Действие GET_DATA_FROM_RESPONSE

```
byCRC:=0;

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


sCRC:=BYTE_TO_STRH(byCRC);

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

		xWrongCRC:=FALSE;

		fb_ANALYZE_DATA(sData:=sReadData);

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

ELSE
	xWrongCRC:=TRUE;
END_IF
```

A.2.4. Действие OUTPUTS_TO_ZERO

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

А.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
```

```
        OPEN_PORT:

                SETTINGS();

                OPEN();

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

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


        CLOSE_PORT:

                        CLOSE();

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

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

END_CASE
```

Б.1.1. Действие SETTINGS

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

Б.1.2. Действие OPEN

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

fb_COMopen();

hCom := fb_COMopen.hCom;
```

Б.1.3. Действие CLOSE

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

fb_COMclose();
```

Б.2. ФБ SET_4TM

```
FUNCTION_BLOCK SET_4TM

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

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

VAR
	eState:				SET_4TM_STATE;

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

	szWriteSize:			COM.CAA.SIZE;

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

	fb_TON:				TON;

	i:				INT;

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

	uConvertToReal:			_4BYTES_TO_REAL;
	uWordTo2Bytes:			_WORD_TO_2BYTES;
END_VAR

VAR CONSTANT
	c_tTimeout:			TIME:=T#1S;
	c_tDelay:			TIME:=T#50MS;
END_VAR
```

```
IF xEnable THEN
	CASE eState OF

		CREATE_CHANNEL:


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


			xDone  :=  FALSE;

			SET_CHANNEL();


			eState :=  OPEN_CHANNEL;


		OPEN_CHANNEL: // шаг отправки запроса на открытие канала связи


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

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


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

				xWriteError := FALSE;

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


		RECEIVE_CHANNEL:

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

			fb_TON();


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


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

				IF xOpenChannel THEN
					eState:=RESPONSE_DELAY_CHANNEL;
				END_IF

			END_IF
```

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


RESPONSE_DELAY_CHANNEL:


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

        eState := CREATE_REQUEST;


CREATE_REQUEST:

        fb_TON();

        SET_REQUEST();

        xDone  := FALSE;

        IF fb_TON.Q THEN
                eState := SEND_REQUEST;
        END_IF


SEND_REQUEST:


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


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


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

                xWriteError := FALSE;

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

                eState      :=  RECEIVE_RESPONSE;
        END_IF

        iLenBuff := 0;

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

```
                RECEIVE_RESPONSE: // шаг получения ответа от модуля

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


                        fb_TON();

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

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

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


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


                RESPONSE_DELAY:


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

                        eState := POLLING_CYCLE_ENDS;


                POLLING_CYCLE_ENDS:


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

                        fb_TON();


                        IF fb_TON.Q THEN

                                iLenBuff       :=     0;

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

                                eState := CREATE_CHANNEL;
                        END_IF

        END_CASE

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

Б.2.1. Действие SET_CHANNEL

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

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

abyWriteData[8]:=uWordTo2Bytes.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
```