



CODESYS V3.5

Реализация нестандартных протоколов



Руководство пользователя

24.05.2022

версия 3.0

Оглавление

Оглавление.....	3
1 Цель и структура документа	6
2 Основные сведения об обмене данными по последовательным интерфейсам	7
2.1 Общие принципы организации обмена	7
2.2 Работа с COM-портом	8
2.3 Типы протоколов обмена	9
2.4 Обработка ошибок обмена	10
3 Описание библиотеки OwenCommunication	11
3.1 Установка библиотеки	11
3.2 Добавление библиотеки в проект CODESYS	12
3.3 Перечисления	13
3.3.1 Перечисление ERROR.....	13
3.3.2 Перечисление COM_PARITY.....	13
3.3.3 Перечисление COM_STOPBIT	14
3.4. ФБ для реализации нестандартных протоколов (RS-485/RS-232).....	15
3.4.1 ФБ COM_Control.....	15
3.4.2 ФБ UNM_SerialRequest	16
3.5 Функции и ФБ преобразования данных	18
3.5.1 ФБ DWORD_TO_WORD2.....	18
3.5.2 ФБ REAL_TO_WORD2	19
3.5.3 Функция WORD2_TO_DWORD	20
3.5.4 Функция WORD2_TO_REAL	21
3.5.5 Функция SWAP_DATA.....	22
4 Описание библиотеки CAA SerialCom	23
4.1 Добавление библиотеки в проект CODESYS	23
4.2 Типы данных и константы.....	24
4.2.1 Перечисление ERROR.....	24
4.2.2 Перечисление PARITY.....	24
4.2.3 Перечисление STOPBIT	24
4.2.4 Перечисление DTR_CONTROL	25
4.2.5 Перечисление RTS_CONTROL	25
4.2.6 Структура COM.PARAMETER.....	26
4.2.7 Список глобальных констант CAA_Parameter_Constants	26
4.3 Функциональные блоки	27
4.3.1 ФБ Open	27

4.3.2 ФБ Write.....	28
4.3.3 ФБ Read.....	29
4.3.4 ФБ Close	30
5 Основные сведения о примерах документа	31
5.1 Общая информация	31
5.2 Используемые библиотеки.....	32
5.3 Информация об асинхронном выполнении и модели поведения ФБ	33
6 Пример реализации протокола DCON.....	34
6.1 Общая информация	34
6.2 Описание протокола.....	34
6.3 Формулировка задачи	36
6.4 Реализация протокола DCON с помощью библиотеки OwenCommunication	37
6.4.1 Перечисление DCON_ERROR_OCL	38
6.4.2 Перечисление DCON_PARAMS	39
6.4.3 ФБ DCON_MASTER_OCL.....	40
6.4.4 Метод prvInitParams.....	47
6.4.5 Метод prvCreateRequest	49
6.4.6 Метод prvParseResponse.....	50
6.4.7 Метод prvConvertGroupAiToReal	53
6.4.8 Программа DCON_PRG_OCL.....	55
6.5 Реализация протокола DCON с помощью библиотеки CAA SerialCom	58
6.5.1 Перечисление DCON_ERROR_CAA	59
6.5.2 Перечисление COM_STATE.....	60
6.5.3 ФБ USR_COM_Control	61
6.5.4 ФБ USR_SerialRequest.....	64
6.5.5 ФБ DCON_MASTER_CAA	69
6.5.6 Программа DCON_PRG_CAA.....	70
7 Пример реализации протокола Пульсар-М	71
7.1 Общая информация	71
7.2 Описание протокола.....	71
7.3 Формулировка задачи	75
7.4 Реализация протокола Пульсар-М с помощью библиотеки OwenCommunication.....	76
7.4.1 Перечисление PULSAR_ERROR_OCL.....	77
7.4.2 Перечисление PULSAR_PARAMS.....	79
7.4.3 ФБ PULSAR_MASTER_OCL	80
7.4.4 Метод prvInitParams.....	87
7.4.5 Метод prvCreateRequest	88
7.4.6 Метод prvParseResponse	90

7.4.7 Метод prvGetDateAndTime.....	94
7.4.8 Программа PULSAR_PRG_OCL	95
7.5 Реализация протокола Пульсар-М с помощью библиотеки CAA SerialCom.....	98
7.5.1 Перечисление PULSAR_ERROR_CAA.....	98
7.5.2 ФБ PULSAR_MASTER_CAA	99
7.5.3 Программа PULSAR_PRG_CAA	100
8 Дополнительные вопросы.....	101
8.1 Нумерация COM-портов в CODESYS	101
8.2 Ограничения, связанные с библиотекой CAA AsyncManager	101
8.3 Реализация обмена по нестандартному протоколу в режиме Slave	103
8.4 Отладка обмена по нестандартному протоколу без контроллера или опрашиваемого устройства	104

1 Цель и структура документа

Одной из важных задач программируемых контроллеров является обмен данными с другими устройствами по последовательным интерфейсам RS-485/RS-232.

В значительном числе случаев обмен осуществляется по стандартным промышленным протоколам (например, **Modbus RTU**) с помощью средств конфигурирования **CODESYS** или библиотек **OBEH**, 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, расходомеры, газоанализаторы т. д.). Ряд таких протоколов для распространенных устройств (например, электросчетчиков Меркурий, Энергомера и т. д.) реализован в составе библиотеке **OwenVendorProtocols**, доступной для загрузки на сайте [OBEH](#) в разделе [CODESYS V3](#). Но устройств с нестандартными протоколами очень много, и поддерживать их все – попросту невозможно. Поэтому в случае необходимости поддержки протоколов устройств, не включенных в библиотеку, пользователю следует реализовать самостоятельно с помощью системных библиотек, которые позволяют работать с последовательным портом «напрямую». Настоящее руководство описывает решение этой задачи для контроллеров **OBEH**, программируемых в среде **CODESYS V3.5**

Структура руководства:

- в [п. 2](#) описываются общие принципы реализации протоколов обмена;
- в [п. 3](#) приведено описание некоторых функций, функциональных блоков и типов данных библиотеки **OwenCommunication**, которые упрощают поддержку протоколов;
- в [п. 4](#) приведено описание библиотеки **CAA SerialCom**, которая содержит низкоуровневые функциональные блоки для работы с COM-портом «напрямую»;
- в [п. 5](#) приведена общая информация о примерах документа;
- в [п. 6](#) приведен пример поддержки протокола DCON с использованием библиотек **OwenCommunication** и **CAA SerialCom** (включая листинги кода и комментарии к реализации);
- в [п. 7](#) приведен пример поддержки протокола Пульсар с использованием библиотек **OwenCommunication** и **CAA SerialCom** (включая листинги кода и комментарии к реализации);
- в [п. 8](#) обзорно рассмотрены специфические вопросы, касающиеся реализации нестандартных протоколов.



ПРИМЕЧАНИЕ

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



ПРИМЕЧАНИЕ

Реализация нестандартных протоколов во многом связана с работой с памятью (конвертацией данных, отображению буферов на структуры данных и т. д.). Информация о работе с памятью в **CODESYS V3.5** приведена в [этой статье](#).

2 Основные сведения об обмене данными по последовательным интерфейсам

2.1 Общие принципы организации обмена

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

Такая архитектура подразумевает наличие в сети единственного master-устройства (обычно таким устройством является контроллер), которое последовательно опрашивает slave-устройства (ими могут быть тепло- и электросчетчики, расходомеры, модули ввода-вывода и т. д.). Slave-устройство не может являться инициатором обмена – оно лишь только отвечает на полученные от master-устройства запросы.

Реализация нестандартного протокола обмена подразумевает решение двух смежных задач:

1. Организация работы с COM-портом (открытие порта, отправка данных в порт, чтение данных из буфера порта).
2. Организация работы с данными (формирование запросов и разбор 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 первый запрос первому slave-устройству.
3. Получить ответ от первого slave-устройства.
4. ... (*последовательная отправка остальных запросов первому slave-устройству*)
5. Отправить последний запрос первому slave-устройству.
6. Получить ответ на последний запрос от первого slave-устройства.
7. ... (*последовательная отправка запросов остальным slave-устройствам*)
8. Отправить последний запрос последнему slave-устройству.
9. Получить ответ на последний запрос от последнего slave-устройства.
10. Перейти к шагу 2.

2.2 Работа с COM-портом

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

1. [OwenCommunication](#) – наиболее функциональный и простой в использовании вариант, поддерживаемый только для контроллеров OWEN (библиотека также может использоваться на виртуальном контроллере **CODESYS Control Win V3** для отладки обмена в случае отсутствия «реального» контроллера). Способ обмена – [асинхронный](#) (неблокирующий режим);
2. [CAA SerialCom](#) – менее удобный по сравнению с **OwenCommunication** вариант (разработчику потребуется написать свои «обертки» над библиотечными блоками для комфортной работы), но поддерживается для большинства контроллеров с системой исполнений CODESYS V3.5. Способ обмена – [асинхронный](#) (неблокирующий режим);
3. **SysCom** – наиболее сложный и наименее удобный вариант для настройки обмена. Способ обмена – синхронный (блокирующий режим).

В рамках данного документа рассматриваются библиотеки **OwenCommunication** и **CAA SerialCom**. Использование библиотеки **SysCom** оправданно только в тех случаях, когда основной задачей контроллера является опрос одного (*или нескольких – но крайне ограниченного количества*) устройства с максимально возможной частотой (при этом время выполнения остального пользовательского кода, не связанного с обменом и обработкой его результатов, не принципиально).

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 данным портом производятся с указанием его дескриптора.
2. Отправка данных в порт. Производится в случае необходимости отправить запрос slave-устройству.
3. Чтение данных из буфера порта. Производится в случае необходимости получения ответа 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 (в диапазоне 0x00...0xFF). Представление каждого типа данных должно соответствовать какому-либо формату. Благодаря этому для каждого запроса обычно 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 0	Байт 1	Байт 2	Байт 3
0xAA	0x71	0xDE	0x42

2. Строковые (текстовые) протоколы

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

Одним из известных строковых протоколов является [DCON](#).

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

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

2.4 Обработка ошибок обмена

В идеальном случае все опрашиваемые slave-устройства всегда корректно отвечают на запросы. Но на практике могут возникать следующие ситуации:

1. Ответ не пришел (например, slave-устройство выключено или перестало функционировать). Если в программе не предусмотрена обработка такой ситуации, то она будет продолжать ожидать ответа, в результате чего обмен с другими устройствами прекратится. Поэтому обычно имеет смысл ожидать ответ в течение некоторого промежутка времени, называемого таймаутом (значение таймаута зависит от скорости обмена, количества передаваемых данных и индивидуальных особенностей устройства), и по его истечению переходить к отправке следующего запроса данному slave-устройству или следующему slave-устройству.
2. Ответ пришел в виде нескольких фрагментов (в большинстве случаев это происходит на низких скоростях обмена и при значительном количестве передаваемых данных). Соответственно, следует контролировать целостность ответа (в бинарных протоколах – по размеру ответа, в строковых – по наличию в ответе стартового и стопового символов). Если условие целостности не выполняется, то разбирать ответ не имеет смысла – необходимо повторно отправить запрос.
3. Ответ был поврежден во время передачи (например, из-за действия помех). Большинство протоколов используют контрольную сумму для проверки корректности ответа.
4. Ответ не соответствует запросу (например, в ответе содержится код команды, не совпадающий с кодом команды в отправленном запросе). В этом случае данный ответ должен расцениваться как некорректный.
5. Некоторые slave-устройства в течение определенного интервала времени после ответа не могут принять новый запрос. В таких случаях необходимо делать в программе паузу между получением ответа на текущий запрос и отправкой следующего запроса.

В рамках примеров документа рассматривается обработка некоторых упомянутых выше ошибок.

3 Описание библиотеки OwenCommunication



ПРИМЕЧАНИЕ

Работа библиотеки поддерживается только на контроллерах ОВЕН и виртуальном контроллере CODESYS Control Win V3.



ПРИМЕЧАНИЕ

Блоки библиотеки, рассмотренные в документе, построены на блоках библиотеки [CAA SerialCom](#).



ПРИМЕЧАНИЕ

В рамках данного документа описаны только те типы данных и POU библиотеки, которые необходимы для реализации нестандартных протоколов обмена по интерфейсам RS-485/RS-232. Описание остальных элементов библиотеки приведено в документе **CODESYS V3.5. Протокол Modbus**.

3.1 Установка библиотеки

Библиотека **OwenCommunication** доступна на сайте компании [ОВЕН](#) в разделе [CODESYS V3/Библиотеки и компоненты](#). Для установки библиотеки в **CODESYS** в меню **Инструменты** следует выбрать пункт **Репозиторий библиотек**, после чего нажать кнопку **Установить** и указать путь к файлу библиотеки:

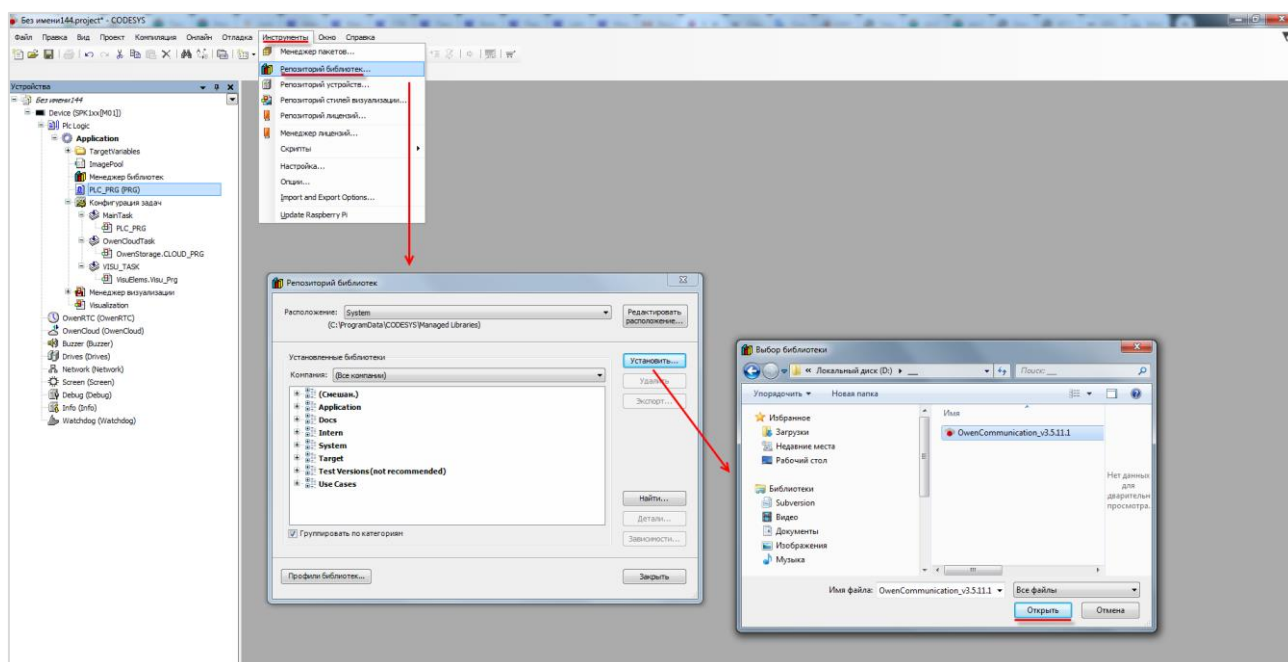


Рисунок 3.1.1 – Установка библиотеки

3.2 Добавление библиотеки в проект CODESYS

Для добавления библиотеки **OwenCommunication** в проект **CODESYS** в **Менеджере библиотек** следует нажать кнопку **Добавить библиотеку**, в появившемся списке выбрать библиотеку **OwenCommunication** и нажать **OK**.

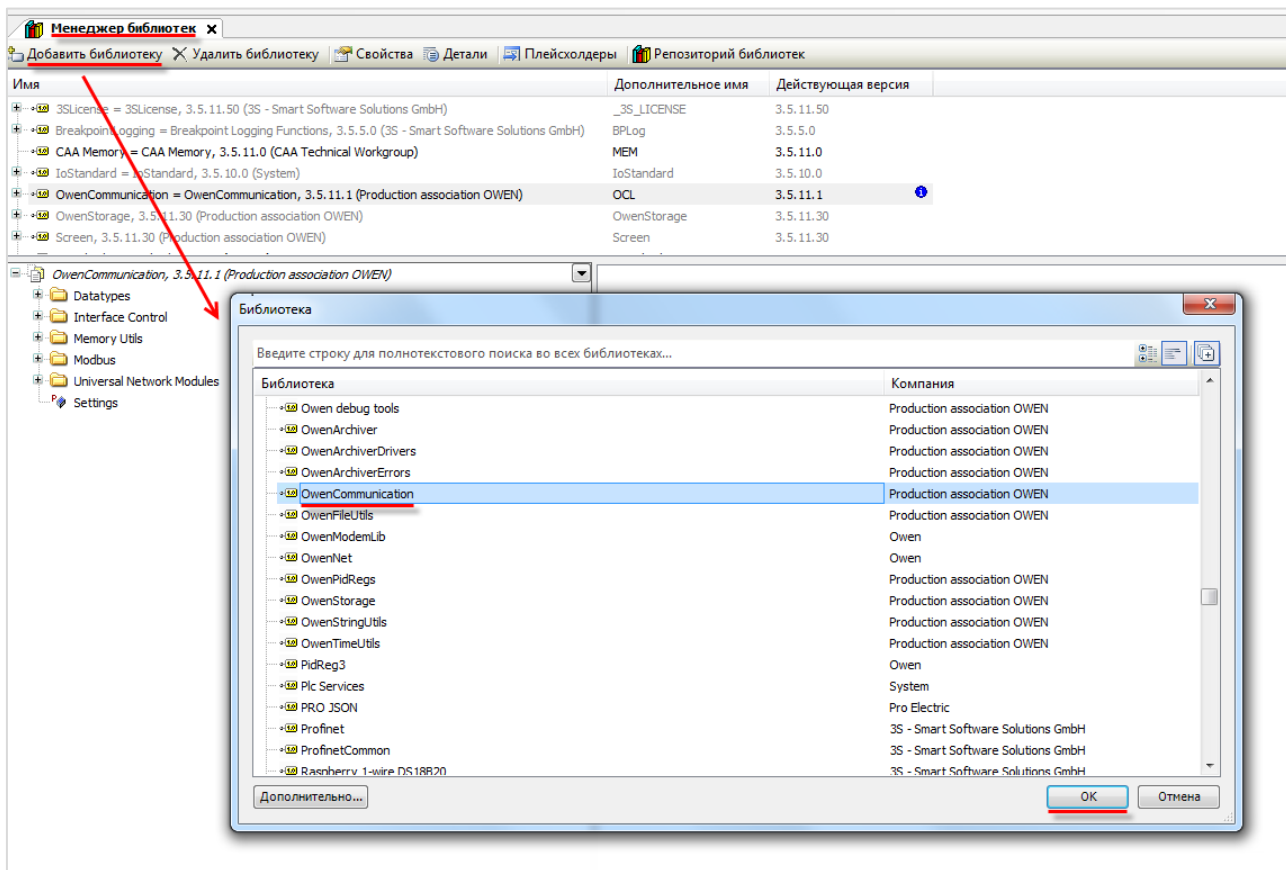


Рисунок 3.2.1 – Добавление библиотеки OwenCommunication

После добавления библиотека появится в списке **Менеджера библиотек**:

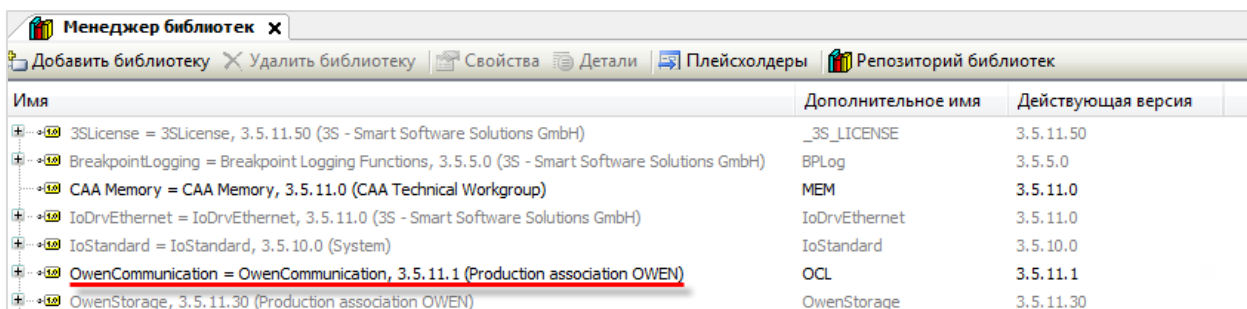


Рисунок 3.2.2 – Список библиотек проекта



ПРИМЕЧАНИЕ

При обращении к объектам библиотеки следует перед их названием указывать префикс **OCL** (пример: **OCL.COM_Control**).

3.3 Перечисления

3.3.1 Перечисление ERROR

Перечисление **ERROR** содержит ошибки, которые могут возникнуть при вызове ФБ библиотеки.

Таблица 3.3.1 – Описание элементов перечисления ERROR

Название	Значение	Описание
Общие ошибки¹		
NO_ERROR	0	Нет ошибок
TIME_OUT	1	Ошибка таймаута
HANDLE_INVALID	10	Некорректное значение дескриптора интерфейса
ERROR_UNKNOWN	11	Неизвестная ошибка (зарезервировано для будущих версий)
WRONG_PARAMETER	12	Как минимум один из аргументов ФБ имеет некорректное значение
WRITE_INCOMPLETE	13	Отправка сообщения не была завершена
INVALID_DATAPOINTER	20	Некорректный указатель на буфер данных
INVALID_DATASIZE	21	Некорректный размер буфера данных
INVALID_ADDR	22	Некорректное значение в поле IP-адреса
Ошибки UDP/TCP		
UDP_RECEIVE_ERROR	30	Ошибка получения UDP-запроса
UDP_SEND_ERROR	31	Ошибка отправки UDP-запроса
UDP_SEND_NOT_COMPLETE	32	Отправка UDP-запроса не была завершена (зарезервировано для будущих версий)
UDP_OPEN_ERROR	33	Ошибка создания UDP-сокета
UDP_CLOSE_ERROR	34	Ошибка закрытия UDP-сокета
TCP_SEND_ERROR	40	Ошибка отправки TCP-запроса
TCP_RECEIVE_ERROR	41	Ошибка получения TCP-запроса
TCP_OPEN_ERROR	42	Ошибка создания TCP-сокета
TCP_CONNECT_ERROR	43	Ошибка при установке TCP-соединения
TCP_CLOSE_ERROR	44	Ошибка при закрытии TCP-соединения
TCP_SERVER_ERROR	45	Ошибка TCP-сервера (зарезервировано для будущих версий)
TCP_NO_CONNECTION	46	TCP-соединение отсутствует
IOCTL_ERROR	47	Внутренняя ошибка при использовании системных вызовов
Ошибки Modbus		
ILLEGAL_FUNCTION	50	Данная функция Modbus не поддерживается slave-устройством
ILLEGAL_DATA_ADDRESS	51	Как минимум один из регистров, указанных в запросе, отсутствует в slave-устройстве
ILLEGAL_DATA_VALUE	52	Некорректное значение в поле данных
SLAVE_DEVICE_FAILURE	53	Slave-устройство не может обработать данный запрос
Специфические ошибки		
RESPONSE_CRC_FAIL	60	Рассчитанная CRC не соответствует CRC посылки
NOT_OWEN_DEVICE	61	Данное устройство не является контроллером OWEN

3.3.2 Перечисление COM_PARITY

Перечисление **COM_PARITY** описывает режим контроля четности COM-порта.

Таблица 3.3.2 – Описание элементов перечисления COM_PARITY

Название	Значение	Описание
----------	----------	----------

¹ При использовании библиотеки для реализации нестандартных протоколов поверх интерфейсов RS-485/RS-232 в рамках перечисления **ERROR** используются только общие ошибки (другие ошибки возвращаются блоками библиотеки, не рассматриваемыми в данном документе).

EVEN	0	Проверка на четность
ODD	1	Проверка на нечетность
NONE	2	Проверка отсутствует

3.3.3 Перечисление COM_STOPBIT

Перечисление **COM_STOPBIT** описывает возможное число стоп-битов при обмене через COM-порт.

Таблица 3.3.3 – Описание элементов перечисления COM_STOPBIT

Название	Значение	Описание
ONE	0	Один стоп-бит
ONE_HALF	1	Полтора стоп-бита (<i>не поддерживается для Linux</i>)
TWO	2	Два стоп-бита

3.4. ФБ для реализации нестандартных протоколов (RS-485/RS-232)

3.4.1 ФБ COM_Control

Функциональный блок **COM_Control** используется для открытия COM-порта с заданными настройками, а также его закрытия.



ПРИМЕЧАНИЕ

Одновременно (в пределах одного цикла контроллера) могут вызываться не более 20 экземпляров асинхронно выполняемых блоков (см. [п. 8.2](#)).

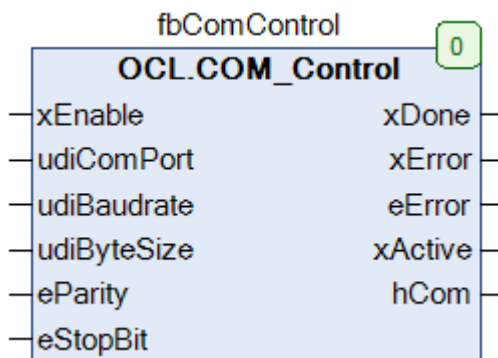


Рисунок 3.4.1 – Внешний вид ФБ COM_Control на языке CFC

Таблица 3.4.1 – Описание входов и выходов ФБ COM_Control

Название	Тип	Описание
Входы		
xEnable	BOOL	По переднему фронту происходит открытие COM-порта, по заднему – закрытие
udiComPort	UDINT	Номер COM-порта (см. п. 8.1)
udiBaudrate	UDINT	Скорость обмена в бодах. Стандартные возможные значения: 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200
udiByteSize	UDINT(7..8)	Число бит данных (7 или 8)
eParity	COM_PARITY	Режим контроля четности
eStopBit	COM_STOPBIT	Число стоп-бит
Выходы		
xDone	BOOL	Принимает TRUE на один цикл ПЛК при успешном открытии порта
xError	BOOL	Принимает значение TRUE в случае возникновения ошибки
eError	ERROR	Статус работы ФБ (или код ошибки)
xActive	BOOL	Пока порт открыт, данный выход имеет значение TRUE
hCom	CAA.HANDLE	Дескриптор COM-порта

3.4.2 ФБ UNM_SerialRequest

Функциональный блок **UNM_SerialRequest** используется для реализации нестандартного протокола при обмене через COM-порт. По переднему фронту на входе **xExecute** происходит отправка содержимого буфера запроса, расположенного по указателю **pRequest** и размером **szRequest** байт через COM-порт, определяемый дескриптором **hCom**, полученным от ФБ [COM_Control](#). Ответ от slave-устройства ожидается в течение времени **tTimeout**. При получении ответа происходит его проверка на основании значений входов **szExpectedSize** и **wStopChar**:

- если **szExpectedSize** $\neq 0$, то ответ считается корректным, если его размер в байтах = **szExpectedSize**;
- если **szExpectedSize** = 0 и **wStopChar** $\neq 16\#0000$, то последние один (при **wStopChar** = **16\#00xx**) или два (при **wStopChar** = **16\#xxxx**) байта ответа (где x – произвольное значение) проверяются на равенство младшему или обоим байтам **wStopChar**. Это может использоваться при реализации строковых протоколов, в которых заранее известен стоп-символ;
- если **szExpectedSize** = 0 и **wStopChar** = **16\#0000**, то любой полученный ответ считается корректным.

В случае получения корректного ответа выход **xDone** принимает значение **TRUE**, выход **eError** = **NO_ERROR**, а на выходе **uiResponseSize** отображается размер ответа в байтах. Полученные данные помещаются в буфер, расположенный по указателю **pResponse** и имеющий размер **szResponse** байт.

В случае отсутствия ответа ФБ повторяет запрос. Число повторений определяется входом **usiRetry** (значение 0 соответствует отсутствию повторений). Если ни на один из запросов не был получен ответ, то выход **xError** принимает значение **TRUE**, а выход **eError** = **TIME_OUT**.

Для отправки нового запроса следует создать передний фронт на входе **xExecute**.



ПРИМЕЧАНИЕ

Одновременно (в пределах одного цикла контроллера) могут вызываться не более 20 экземпляров асинхронно выполняемых блоков (см. [п. 8.2](#)).



ПРИМЕЧАНИЕ

В случае отправки запросов, для которых не подразумевается получение ответа, рекомендуется для входа **tTimeout** установить значение **T#1ms**.

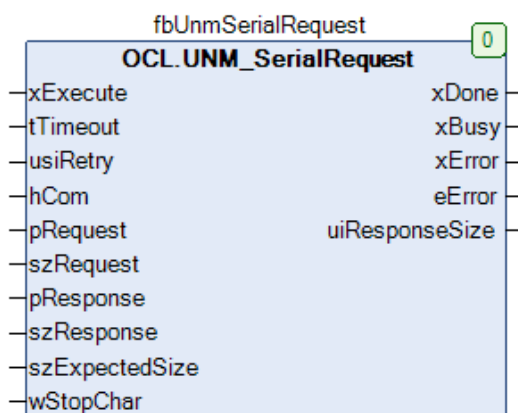


Рисунок 3.4.2 – Внешний вид ФБ UNM_SerialRequest на языке CFC

Таблица 3.4.2 – Описание входов и выходов ФБ UNM_SerialRequest

Название	Тип	Описание
Входы		
xExecute	BOOL	По переднему фронту происходит однократная (с возможностью переповторов в случае отсутствия ответа) отправка запроса
tTimeout	TIME	Таймаут ожидания ответа от slave-устройства (T#0ms – время ожидания не ограничено)
usiRetry	USINT	Число переповторов в случае отсутствия ответа
hCom	CAA.HANDLE	Дескриптор COM-порта, полученный от ФБ COM_Control
pRequest	CAA.PVOID	Указатель на буфер запроса
szRequest	CAA.SIZE	Размер буфера запроса в байтах
pResponse	CAA.PVOID	Указатель на буфер ответа
szResponse	CAA.SIZE	Размер буфера ответа в байтах
szExpectedSize	CAA.SIZE	Ожидаемый размер ответа в байтах (0 – размер неизвестен)
wStopChar	WORD	Стоп-символы протокола. Для протокола с двумя стоп-символами оба байта переменной должны быть отличны от нуля. Для протокола с одним стоп-символом старший байт должен быть равен нулю, а младший быть отличным от нуля. Если в протоколе отсутствуют стоп-символы, то следует установить значение 0
Выходы		
xDone	BOOL	TRUE – получен корректный ответ от slave-устройства
xBusy	BOOL	TRUE – ФБ находится в работе
xError	BOOL	Принимает значение TRUE в случае возникновения ошибки
eError	ERROR	Статус работы ФБ (или код ошибки)
uiResponseSize	UINT	Размер полученного ответа в байтах

3.5 Функции и ФБ преобразования данных

3.5.1 ФБ DWORD_TO_WORD2

Функциональный блок **DWORD_TO_WORD2** используется для преобразования переменной типа **DWORD** в две переменные типа **WORD**.

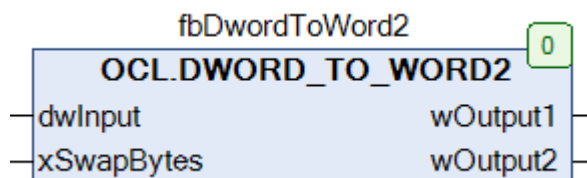


Рисунок 3.5.1 – Внешний вид ФБ DWORD_TO_WORD2 на языке CFC

Таблица 3.5.1 – Описание входов и выходов ФБ DWORD_TO_WORD2

Название	Тип	Описание
Входы		
dwInput	DWORD	Исходная переменная
xSwapBytes	BOOL	TRUE – выполнить перестановку байт (A1 B2 C3 D4 ---> B2 A1 D4 C3)
Выходы		
wOutput1	WORD	Старшее слово исходной переменной
wOutput2	WORD	Младшее слово исходной переменной

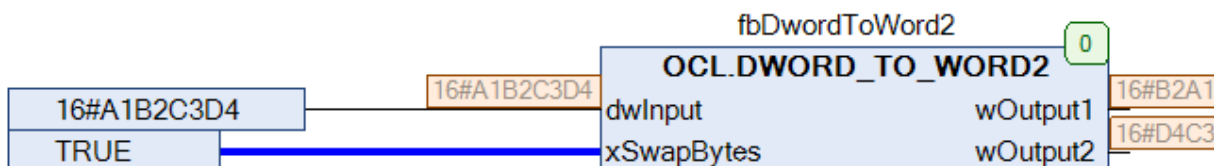


Рисунок 3.5.1 – Пример использования ФБ DWORD_TO_WORD2 на языке CFC

3.5.2 ФБ REAL_TO_WORD2

Функциональный блок **REAL_TO_WORD2** используется для преобразования переменной типа **REAL** в две переменные типа **WORD**.

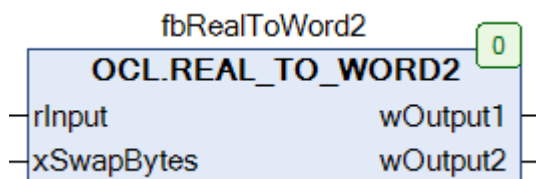


Рисунок 3.5.2 – Внешний вид ФБ REAL_TO_WORD2 на языке CFC

Таблица 3.5.2 – Описание входов и выходов ФБ REAL_TO_WORD2

Название	Тип	Описание
Входы		
rInput	REAL	Исходная переменная
xSwapBytes	BOOL	TRUE – выполнить перестановку байт (A1 B2 C3 D4 ---> B2 A1 D4 C3)
Выходы		
wOutput1	WORD	Старшее слово исходной переменной
wOutput2	WORD	Младшее слово исходной переменной

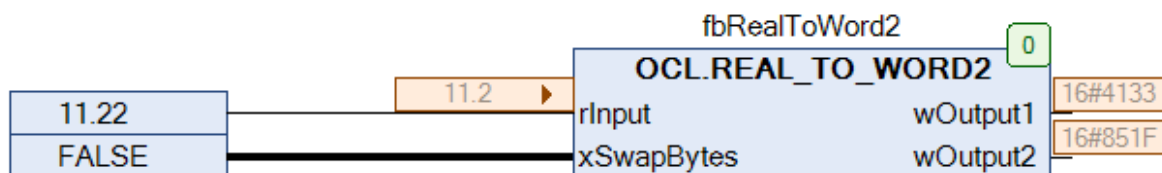


Рисунок 3.5.2 – Пример использования ФБ REAL_TO_WORD2 на языке CFC (см. [онлайн-конвертер](#) для проверки)

3.5.3 Функция WORD2_TO_DWORD

Функция **WORD2_TO_DWORD** используется для преобразования двух переменных типа **WORD** в переменную типа **DWORD**.

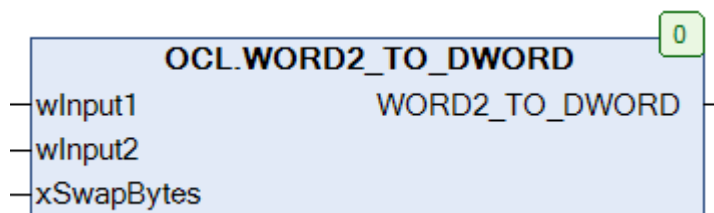


Рисунок 3.5.3 – Внешний вид функции WORD2_TO_DWORD на языке CFC

Таблица 3.5.3 – Описание входов и выходов функции WORD2_TO_DWORD

Название	Тип	Описание
Входы		
wInput1	WORD	Исходная переменная 1
wInput2	WORD	Исходная переменная 2
xSwapBytes	BOOL	TRUE – выполнить перестановку байт (A1 B2 C3 D4 ---> B2 A1 D4 C3)
Выходы		
WORD2_TO_DWORD	DWORD	Новая переменная типа DWORD

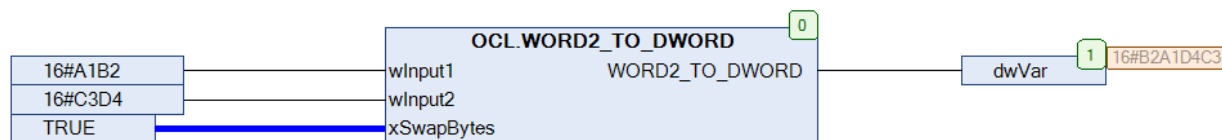


Рисунок 3.5.3 – Пример использования функции WORD2_TO_DWORD на языке CFC

3.5.4 Функция WORD2_TO_REAL

Функция **WORD2_TO_REAL** используется для преобразования двух переменных типа **WORD** в переменную типа **REAL**.

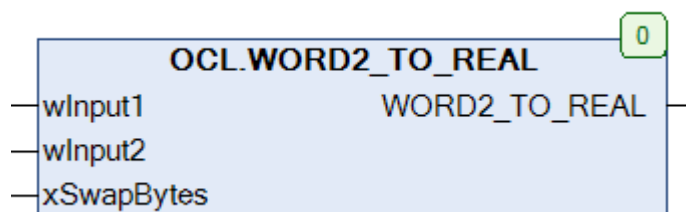


Рисунок 3.5.4 – Внешний вид функции WORD2_TO_REAL на языке CFC

Таблица 3.5.4 – Описание входов и выходов функции WORD2_TO_REAL

Название	Тип	Описание
Входы		
wInput1	WORD	Исходная переменная 1
wInput2	WORD	Исходная переменная 2
xSwapBytes	BOOL	TRUE – выполнить перестановку байт (A1 B2 C3 D4 ---> B2 A1 D4 C3)
Выходы		
WORD2_TO_REAL	REAL	Новая переменная типа REAL

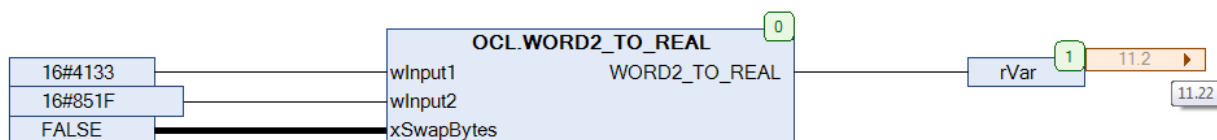


Рисунок 3.5.4 – Пример использования функции WORD2_TO_REAL на языке CFC (см. [онлайн-конвертер](#) для проверки)

3.5.5 Функция SWAP_DATA

Функция **SWAP_DATA** используется для копирования данных из одного буфера в другой с перестановкой байт и регистров.

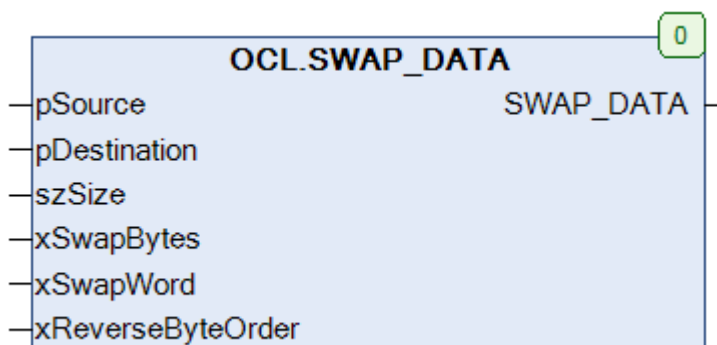


Рисунок 3.5.5 – Внешний вид функции SWAP_DATA на языке CFC

Таблица 3.5.5 – Описание входов и выходов функции SWAP_DATA

Название	Тип	Описание
Входы		
pSource	CAA.PVOID	Указатель на буфер исходных данных
pDestination	CAA.PVOID	Указатель на буфер, в который будут скопированы данные. Может совпадать с pSource – тогда после изменения данные будут помещены в тот же буфер
szSize	CAA.SIZE	Размер копируемых данных в байтах (должен не превышать размеры буферов)
xSwapBytes	BOOL	TRUE – выполнить перестановку байт (A1 B2 C3 D4 ---> B2 A1 D4 C3)
xSwapWord	BOOL	TRUE – выполнить перестановку регистров (A1 B2 C3 D4 ---> C3 D4 A1 B2)
xReverseByteOrder	BOOL	TRUE – изменить порядок байт на противоположный (A1 B2 C3 D4 ---> D4 C3 B2 A1). Если данный вход имеет значение TRUE , то входы xSwapBytes и xSwapWord не обрабатываются
Выходы		
SWAP_DATA	BOOL	TRUE – операция выполнена

4 Описание библиотеки CAA SerialCom

4.1 Добавление библиотеки в проект CODESYS

Библиотека **CAA SerialCom** используется для работы с последовательным портом. Библиотека входит в дистрибутив CODESYS и не требует отдельной установки.

Для добавления библиотеки в проект **CODESYS** в **Менеджере библиотек** следует нажать кнопку **Добавить библиотеку**, в появившемся списке выбрать библиотеку **CAA SerialCom** и нажать **OK**.

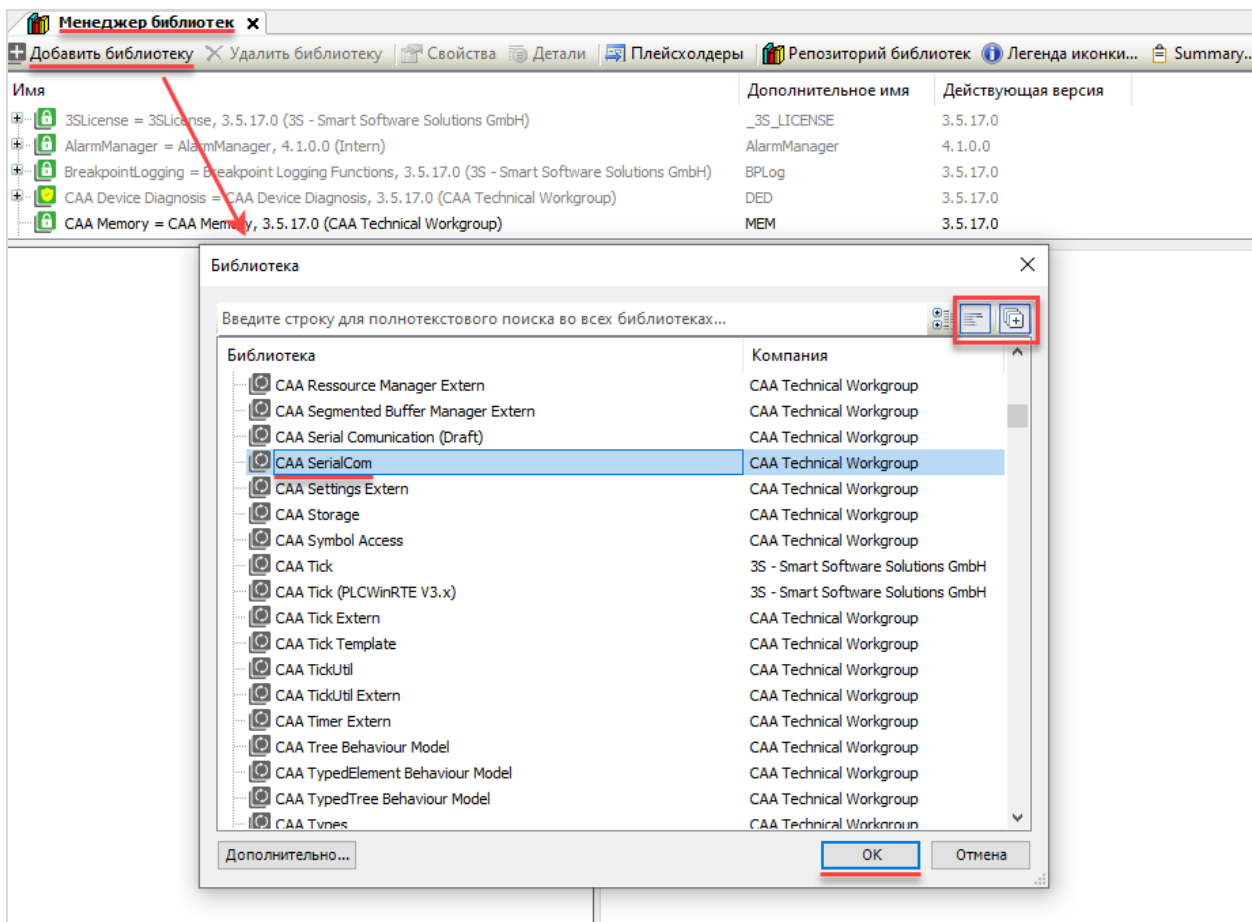


Рисунок 4.1 – Добавление библиотеки CAA SerialCom в проект CODESYS



ПРИМЕЧАНИЕ

При обращении к объектам библиотеки следует перед их названием указывать префикс **COM** (пример: **COM.Open**).

4.2 Типы данных и константы

4.2.1 Перечисление ERROR

Перечисление **ERROR** содержит ошибки, которые могут возникнуть при вызове ФБ библиотеки.

Таблица 4.2.1 – Описание элементов перечисления ERROR

Название	Значение	Описание
NO_ERROR	0	Нет ошибок
FIRST_ERROR	5000	«Маркер» ² – первая ошибка библиотеки
TIME_OUT	5001	Ошибка таймаута выполнения операции (см. вход udiTimeOut)
ABORT	5002	Выполнение ФБ было прервано с помощью входа xAbort
HANDLE_INVALID	5003	Некорректное значение дескриптора интерфейса
ERROR_UNKNOWN	5004	Неизвестная ошибка
WRONG_PARAMETER	5005	Как минимум один из аргументов ФБ имеет некорректное значение
WRITE_INCOMPLETE	5006	Отправка сообщения не была завершена
FIRST_MF	5050	«Маркер» – первая ошибка, определенная производителем контроллера ³
LAST_ERROR	5099	«Маркер» – последняя ошибка библиотеки

4.2.2 Перечисление PARITY

Перечисление **PARITY** описывает режим контроля четности COM-порта.

Таблица 4.2.2 – Описание элементов перечисления PARITY

Название	Значение	Описание
EVEN	0	Проверка на четность
ODD	1	Проверка на нечетность
NONE	2	Проверка отсутствует

4.2.3 Перечисление STOPBIT

² Маркеры носят информационный характер. В данном случае с помощью маркеров поясняется, что коды ошибок библиотеки принадлежат диапазону 5000...5099, при этом коды 5050...5099 могут использоваться производителями контроллеров для возвращения информации о специфичных для них ошибках.

³ Для контроллеров OVEN не поддерживаются специфических кодов ошибок.

Перечисление **STOPBIT** описывает возможное число стоп-битов при обмене через COM-порт.

Таблица 4.2.3 – Описание элементов перечисления STOPBIT

Название	Значение	Описание
ONE	0	Один стоп-бит
ONE_HALF	1	Полтора стоп-бита (<i>не поддерживается для Linux</i>)
TWO	2	Два стоп-бита

4.2.4 Перечисление DTR_CONTROL⁴

Перечисление **DTR_CONTROL** описывает режим управления сигналом [DTR](#) для интерфейса RS-232.

Таблица 4.2.4 – Описание элементов перечисления DTR_CONTROL

Название	Значение	Описание
DTR_CONTROL_DISABLE	0	Сигнал DTR всегда отключен
DTR_CONTROL_ENABLE	1	Сигнал DTR всегда включен
DTR_CONTROL_HANDSHAKE	2	Состояние сигнала DTR определяется подключенным устройством

4.2.5 Перечисление RTS_CONTROL⁵

Перечисление **RTS_CONTROL** описывает режим управления сигналом [RTS](#) для интерфейса RS-232.

Таблица 4.2.5 – Описание элементов перечисления RTS_CONTROL

Название	Значение	Описание
RTS_CONTROL_DISABLE	0	Сигнал RTS всегда отключен
RTS_CONTROL_ENABLE	1	Сигнал RTS всегда включен
RTS_CONTROL_HANDSHAKE	2	Состояние сигнала RTS определяется подключенным устройством
RTS_CONTROL_TOGGLE	3	Сигнал RTS будет включен, если в буфере COM-порта есть байты для передачи. После отправки всех байтов сигнал RTS будет отключен

⁴ COM-порты контроллеров OBEH, программируемых в среде CODESYS V3.5, не имеют сигнала DTR.

⁵ COM-порты контроллеров OBEH, программируемых в среде CODESYS V3.5, не имеют сигнала RTS.

4.2.6 Структура COM.PARAMETER

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

Таблица 4.2.6 – Описание переменных структуры COM.PARAMETER

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

4.2.7 Список глобальных констант CAA_Parameter_Constants

Список глобальных констант **CAA_Parameter_Constant** содержит параметры COM-порта. Ниже описаны только параметры, поддерживаемые контроллерами ОВЕН. Полный список параметров приведен в описании библиотеки в менеджере библиотек.

Таблица 4.2.7 – Описание констант списка CAA_Parameter_Constant, поддерживаемых в контроллерах ОВЕН

Название	Значение	Описание
udiPort	16#1	Номер COM-порта (см. п. 8.1)
udiStopBits	16#2	Число стоп-бит. См. перечисление STOPBIT
udiParity	16#3	Режим контроля четности. См. перечисление PARITY
udiBaudrate	16#4	Скорость обмена в бодах. Возможные значения: 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200
udiByteSize	16#7	Количество информационных бит в передаваемых/принимаемых байтах (7 или 8)

4.3 Функциональные блоки

4.3.1 ФБ Open

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



ПРИМЕЧАНИЕ

Одновременно (в пределах одного цикла контроллера) могут вызываться не более 20 экземпляров асинхронно выполняемых блоков (см. [п. 8.2](#)).

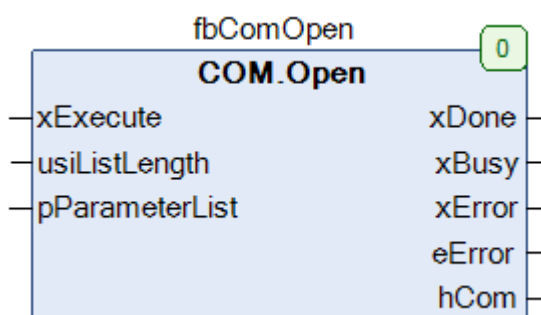


Рисунок 4.3.1 – Внешний вид ФБ Open на языке CFC

Таблица 4.3.1 – Описание входов и выходов ФБ Open

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	По переднему фронту выполняется открытие COM-порта
usiListLength	USINT	Количество используемых параметров порта
pParameterList	CAA.PVOID	Указатель на массив параметров порта (см. PARAMETER)
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки
eError	ERROR	Статус работы ФБ (или код ошибки)
hCom	CAA.HANDLE	Дескриптор (handle) открытого порта

4.3.2 ФБ Write

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



ПРИМЕЧАНИЕ

Одновременно (в пределах одного цикла контроллера) могут вызываться не более 20 экземпляров асинхронно выполняемых блоков (см. [п. 8.2](#)).

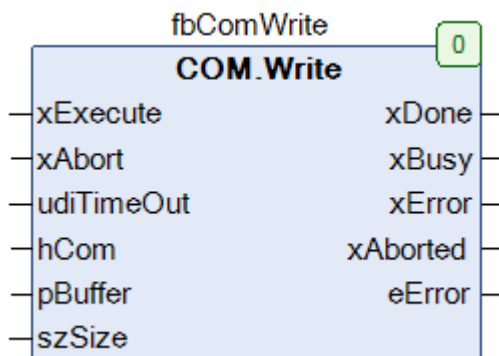


Рисунок 4.3.2 – Внешний вид ФБ Write на языке CFC

Таблица 4.3.2 – Описание входов и выходов ФБ Write

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	По переднему фронту выполняется отправка данных в COM-порт
xAbort	BOOL	Переменная прекращения работы блока. Если она принимает значение TRUE , то блок немедленно прекращает работу, а выходные переменные сбрасываются к начальным значениям
udiTimeOut	UDINT	Время в мс, через которое блок принудительно завершает свою работу (0 – время не ограничено)
hCom	CAA.HANDLE	Дескриптор (handle) порта, через который выполняется отправка данных. См. выход hCom ФБ COM.Open
pBuffer	CAA.PVOID	Указатель на отправляемые данные
szSize	CAA.SIZE	Размер отправляемых данных (в байтах)
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение TRUE в случае возникновения ошибки
xAborted	BOOL	Флаг «работа ФБ была прервана»
eError	ERROR	Статус работы ФБ (или код ошибки)

4.3.3 ФБ Read

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



ПРИМЕЧАНИЕ

Одновременно (в пределах одного цикла контроллера) могут вызываться не более 20 экземпляров асинхронно выполняемых блоков (см. [п. 8.2](#)).

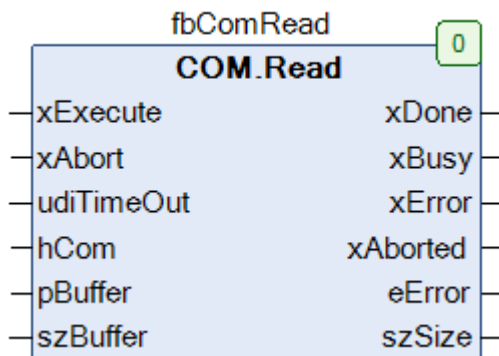


Рисунок 4.3.3 – Внешний вид ФБ Read на языке CFC

Таблица 4.3.3 – Описание входов и выходов ФБ Read

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	По переднему фронту выполняется чтение данных из <u>буфера</u> COM-порта
xAbort	BOOL	Переменная прекращения работы блока. Если она принимает значение TRUE , то блок немедленно прекращает работу и выходные переменные сбрасываются к начальным значениям
udiTimeOut	UDINT	Время в мс, через которое блок принудительно завершает свою работу (0 – время не ограничено)
hCom	CAA.HANDLE	Дескриптор (handle) порта, из буфера которого происходит чтение. См. выход hCom ФБ COM.Open
pBuffer	CAA.PVOID	Указатель, по которому будут размещены считанные данные
szBuffer	CAA.SIZE	Размер буфера считываемых данных (в байтах)
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока (<i>при этом в буфере могут отсутствовать данные – это корректная ситуация, которая не считается ошибкой</i>)
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение TRUE в случае возникновения ошибки
xAborted	BOOL	Флаг «работа ФБ была прервана»
eError	ERROR	Статус работы ФБ (или код ошибки)
szSize	CAA.SIZE	Размер считанных данных (в байтах)

4.3.4 ФБ Close

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



ПРИМЕЧАНИЕ

Одновременно (в пределах одного цикла контроллера) могут вызываться не более 20 экземпляров асинхронно выполняемых блоков (см. [п. 8.2](#)).

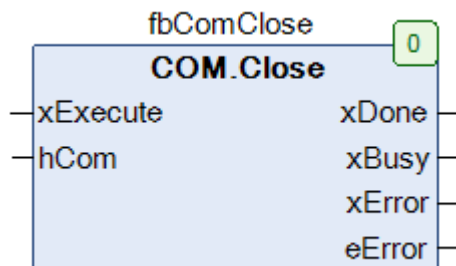


Рисунок 4.3.4 – Внешний вид ФБ Close на языке CFC

Таблица 4.3.4 – Описание входов и выходов ФБ Close

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	По переднему фронту выполняется закрытие COM-порта
hCom	CAA.HANDLE	Дескриптор (handle) закрываемого порта. См. выход hCom ФБ COM.Open
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение TRUE в случае возникновения ошибки
eError	ERROR	Статус работы ФБ (или код ошибки)

5 Основные сведения о примерах документа

5.1 Общая информация

В рамках примеров документа рассматривается частичная реализация протоколов [DCON](#) и [Пульсар-М](#). Для каждого протокола приведены две реализации:

- с помощью библиотеки [OwenCommunication](#) – эта реализация является более простой (включает в себя меньше строк кода), но поддерживается только для контроллеров OBEH (библиотека также может использоваться на виртуальном контроллере **CODESYS Control Win V3** для отладки обмена в случае отсутствия «реального» контроллера);
- с помощью библиотеки [CAA SerialCom](#) – эта реализация несколько сложнее, но является универсальной (может использоваться на любом контроллере с **CODESYS V3.5**, который поддерживает эту библиотеку).

Реализации для обеих библиотек гармонизированы и в максимально возможной степени схожи друг с другом.

При разработке примеров делался акцент на балансе между их расширяемостью (возможностью добавления в примеры дополнительных команд протоколов), наглядностью и функциональностью. В описании примеров будут поясняться конкретные фрагменты кода, но их построчное описание не приводится, так как является избыточным. Кроме того, исходный код примера сопровождается подробными комментариями.

Обратите внимание, что приведенные примеры созданы не «в один заход», а в процессе нескольких последовательных итераций, каждая из которых сопровождалась тестированием добавленного функционала. Рекомендуется придерживаться такого подхода и при разработке пользовательских блоков нестандартных протоколов.

Основной программой примера является программа **MAIN_PRG**, привязанная к задаче MainTask. В этой программе выполняется вызов одной из 4-х программ примера:

- **DCON_PRG_OCL** – опрос устройства по протоколу **DCON** с помощью блоков библиотеки **OwenCommunication**;
- **DCON_PRG_CAA** – опрос устройства по протоколу **DCON** с помощью блоков библиотеки **CAA SerialCom**;
- **PULSAR_PRG_OCL** – опрос устройства по протоколу **Пульсар-М** с помощью блоков библиотеки **OwenCommunication**;
- **PULSAR_PRG_CAA** – опрос устройства по протоколу **Пульсар-М** с помощью блоков библиотеки **CAA SerialCom**.

Так как все примеры рассчитаны на использование одного и того же COM-порта – то в каждый момент времени должна быть раскомментирована только одна из программ, а остальные три должны быть закомментированы. В примере используется COM-порт с ID = 1 и настройками 115200-8-N-1. В случае необходимости пользователь может изменить эти настройки (они задаются в вызове ФБ открытия COM-порта в первых строках программы).

5.2 Используемые библиотеки

Помимо **OwenCommunication** и **CAA SerialCom** в примерах используются следующие библиотеки (если вы повторяете примеры «с нуля», то они должны быть добавлены в менеджере библиотек вашего проекта):

Таблица 5.2.1 – Описание библиотек примера

Библиотека	Пространство имен	Объекты библиотеки, используемые в примере
CAA Types	CAA	Псевдонимы (ALIAS) типов данных, используемых другими библиотеками (CAA.HANDLE, CAA.PVOID и т. д.)
CAA Memory	MEM	Функции для работы с блоками памяти (копирование, заполнение и т. д.).
Standard	Standard	Детекторы фронтов, таймеры, базовые функции для работы со строками
Util	Util	Функции для конверсии формата DEC в BCD и «сборки» системного времени из отдельных разрядов
OwenStringUtils	OSU	Расширенные функции для работы со строками. Библиотека не входит в дистрибутив CODESYS и должна быть загружена с сайта OBEH (раздел CODESYS V3)

5.3 Информация об асинхронном выполнении и модели поведения ФБ

Блоки библиотек **OwenCommunication** и **CAA SerialCom** являются асинхронными. Это означает, что выполнение блоком своей операции может занимать несколько циклов задачи контроллера, в которой он вызывается. Выполнение начинается в момент генерации переднего фронта входе **xExecute** – при этом необязательно удерживать **xExecute** в **TRUE** в процессе работы блока, хотя такой подход тоже является легитимным и активно используется в примерах документа. В этом же цикле выход **xBusy** принимает значение **TRUE**, что является признаком того, что блок приступил к работе. В случае успешного завершения работы выход **xDone** принимает значение **TRUE**. Если же в процессе выполнения произошла какая-либо ошибка, то выход **xError** принимает значение **TRUE**, а выход **eError** содержит код возникшей ошибки (выход **xDone** при этом имеет значение **FALSE**). В том цикле, в котором выход **xDone** или **xError** получает значение **TRUE**, выполняется сброс выхода **xBusy** в **FALSE**, так как блок к этому моменту уже завершил работу.

Сброс выходов **xDone**, **xError**, **eError** и ряда специфических выходов каждого ФБ (например, выхода **uiResponseSize** ФБ [UNM_SerialRequest](#)) может происходить по одному из двух сценариев:

- если к моменту завершения работы блока (успешно или с ошибкой) вход **xExecute** имеет значение **FALSE** (например, на вход был подан единичный импульс), то выходы блока устанавливаются на один цикл, после чего сбрасываются в значения по умолчанию;
- если к моменту завершения работы блока (успешно или с ошибкой) вход **xExecute** имеет значение **TRUE**, то выходы блока будут иметь актуальные значения до тех пор, пока вход **xExecute** не примет значение **FALSE**.

Таким образом, признаком успешного завершения работы блока является значение **TRUE** на выходе **xDone** – только в этом случае имеет смысл обрабатывать специфические выходы ФБ и данные, которые блок возвращает по указателю.

Описанное поведение являются частью модели поведения **CODESYS Common Behaviour Model** (также известной как **PLCopen Behaviour Model**). Более подробная информация об этой модели поведения приведена в [онлайн-справке CODESYS](#) и [этой статье](#).



ПРИМЕЧАНИЕ

Одновременно (в пределах одного цикла контроллера) могут вызываться не более 20 экземпляров асинхронно выполняемых блоков (см. [п. 8.2](#)).

6 Пример реализации протокола DCON

6.1 Общая информация

Протокол [DCON](#) является одним из простейших строковых протоколов обмена. На примере протокола DCON будут описаны принципы, с помощью которых можно реализовать любой подобный строковый протокол.

Протокол DCON используется в модулях ввода-вывода таких производителей, как OBEH, Advantech, ICP DAS, RealLab и др. Протокол не имеет единой спецификации – в руководстве на конкретное устройство обычно перечислены детали реализации протокола и поддерживаемые команды.

В рамках примера будут поддерживаться команды для работы с модулями аналогового ввода OBEH MB110-xA(C).

Ссылка на готовый пример: [Example_UserProtocols.projectarchive](#)

Пример создан в среде **CODESYS 3.5 SP17 Patch 3** и подразумевает запуск на **СПК1xx [M01]** с таргет-файлом **3.5.17.31**. В случае необходимости запуска проекта на другом устройстве следует изменить таргет-файл в проекте (**ПКМ** на узел **Device** – **Обновить устройство**).

6.2 Описание протокола

Структура запроса протокола DCON в общем виде выглядит следующим образом:

Символ начала сообщения	Адрес	Команда	Данные	Контрольная сумма	Символ конца сообщения
1 байт	2 байта	N байт	N байт	2 байта	1 байт

Ответ имеет аналогичную структуру, но поле **Команда** в нем отсутствует.

Символ начала сообщения (стартовый символ) – отличается для разных устройств, команд, а также типа кадра (запрос или ответ). Примеры стартового символа запросов: \$, #, %, @, *. Примеры стартового символа ответов: ~, !, ?, >.

Адрес – адрес конкретного прибора в диапазоне 0...255. Присутствует практически⁶ во всех запросах, но может отсутствовать в ответе на определенные запросы.

Команда – используемая команда. Может отсутствовать в запросе и всегда отсутствует в ответе.

Данные – данные, передаваемые в запросе или полученные в ответе. Могут отсутствовать как в запросе, так и в ответе.

Контрольная сумма – сумма ASCII-кодов всех предшествующих контрольной сумме символов. В случае переполнения – старшие разряды отбрасываются (контрольная сумма всегда занимает 2

⁶ Некоторые приборы поддерживают широковебательные команды (например, #** и ~**). В рамках примера они не рассматриваются – подразумевается, что поле адреса всегда присутствует в запросе.

байта). Обычно приборы с поддержкой протокола DCON позволяют отключить использование контрольной суммы – тогда ее поле будет отсутствовать в запросах и ответах.

Символ конца сообщения – символ перевода строки (ASCII-код: 0x0D).

Все данные передаются в символьном виде – в виде ASCII-кодов HEX-символов верхнего регистра (см. примеры ниже).

Модули ОВЕН МВ110-хА(С) поддерживают две команды – команду группового чтения всех аналоговых входов и команду считывания значения конкретного аналогового входа (*описание команд приведено в руководстве по эксплуатации на модуль*). Ниже приведены примеры этих команд и ответы на них для модуля с адресом **11** ([CHK] и (cr) – обозначения контрольной суммы и стопового символа соответственно).

Групповое чтение аналоговых входов

Запрос: #0B[CHK](cr)

Ответ на корректный запрос:

>+499.98+33.758+49.998+33.880+50.000+49.998+34.601+34.652[CHK](ск)

Ответ на некорректный запрос: отсутствует (запрос не может быть некорректным)

Чтение значения 3-го аналогового входа

Запрос: #0B2[CHK](cr)

В реализации протокола в модуле нумерация входов ведется с 0, поэтому для запроса 3-го входа необходимо передать значение 2.

Ответ на корректный запрос:

>+49.998[CHK](cr)

Ответ на запрос с некорректным номером входа: ?AA[CHK](cr)

6.3 Формулировка задачи

После ознакомления с [п. 6.1](#) и [п. 6.2](#) можно сформулировать требования к функциональному блоку, который будет реализовывать обмен по протоколу DCON:

- блок должен поддерживать команды модулей ОВЕН МВ110-хА(С);
- блок должен позволять разработчику с минимумом усилий добавлять поддержку новых команд;
- блок должен позволять отправить произвольный запрос – это полезно в тех случаях, когда блок будет распространяться в виде скомпилированной библиотеке и у конечного пользователя не будет возможности его доработать;
- блок должен поддерживать использование контрольной суммы, но также и позволять отключить ее проверку (так как отключение контрольной суммы поддерживается большинство модулей с поддержкой DCON – и в случае интеграции контроллера в систему с уже настроенными таким образом модулями у пользователя должна быть возможность их опросить);
- блок должен обеспечивать обработку ошибок.

К обрабатываемым ошибкам будут относиться:

- получение ответа с некорректным стартовым символом (в том числе, со стартовым символом, который является признаком ошибки – см. пример «Ответ на запрос с некорректным номером входа» в [п. 6.3](#);
- получение ответа с некорректной контрольной суммой (т. е. контрольная сумма ответа не соответствует рассчитанной контрольной сумме для этого ответа);
- получение ответа с некорректным стоповым символом (отличающимся от символа с ASCII-кодом 0x0D).

Команды и коды ошибок протокола удобно представить в виде перечислений.

6.4 Реализация протокола DCON с помощью библиотеки OwenCommunication

В следующих пунктах описана реализация протокола DCON с помощью функциональных блоков из библиотеки **OwenCommunication**. В рамках реализации созданы следующие объекты:

- перечисление команд протокола [DCON_PARAMS](#);
- перечисление кодов ошибок [DCON_ERROR_OCL](#);
- функциональный блок [DCON_MASTER_OCL](#), реализующий обмен по протоколу DCON и включающий в себя ряд методов.

Перечисления позволяют задать соответствие между числовыми значениями и символьными именами – это повышает читабельность кода. Например, вместо того, чтобы помнить, что код ошибки 12 означает «некорректные входные параметры ФБ» вы можете использовать в коде его символьное имя, объявленное в перечислении – **DCON_OCL_ERROR.WRONG_PARAMETER**.

Для создания перечисления следует нажать **ПКМ** на узел **Application**, использовать команду **Добавление объекта – DUT** и указать тип объекта **Перечисления**.

Методы позволяют упростить код функционального блока или программы, разделив его на отдельные фрагменты. Кроме того, методы могут иметь входные, выходные и локальные переменные – что позволяет их использовать в качестве «встраиваемых» функций.

Для создания метода следует нажать **ПКМ** на объект, к которому должен быть добавлен метод, и использовать команду **Добавление объекта – Метод**.

Папки используются для структуризации проекта.

Для создания папки следует нажать **ПКМ** на свободное пространство в дереве проекта и использовать команду **Добавить папку**.

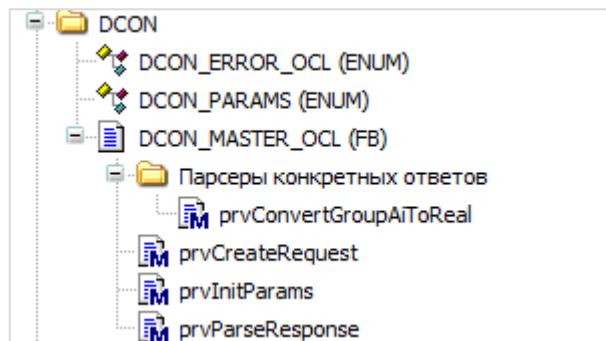


Рисунок 6.4.1 – Объекты примера

6.4.1 Перечисление DCON_ERROR_OCL

Перечисление **DCON_ERROR_OCL** содержит ошибки, которые могут возникнуть при вызове ФБ [DCON_MASTER_OCL](#). Перечисление включает в себя коды ошибок из перечисления [ERROR](#) библиотеки [OwenCommunication](#) (только те, которые могут возникнуть при обмене по интерфейсам RS-485/RS-232) и специфические коды ошибок, определенные в п. 6.3. Числовой тип перечисления (INT) выбран в соответствии с числовым типом перечисления [OCL.ERROR](#), чтобы упростить конвертацию одного перечисления в другое.

Таблица 6.4.1 – Описание элементов перечисления DCON_ERROR_OCL

Название	Значение	Описание
Общие ошибки		
NO_ERROR	0	Нет ошибок
TIME_OUT	1	Ошибка таймаута
HANDLE_INVALID	10	Некорректное значение дескриптора интерфейса
ERROR_UNKNOWN	11	Неизвестная ошибка (зарезервировано для будущих версий)
WRONG_PARAMETER	12	Как минимум один из аргументов ФБ имеет некорректное значение
WRITE_INCOMPLETE	13	Отправка сообщения не была завершена
INVALID_DATAPOINTER	20	Некорректный указатель на буфер данных
INVALID_DATASIZE	21	Некорректный размер буфера данных
Специфические ошибки протокола DCON		
NOT_EXPECTED_START_CHAR	30	Стартовый символ в ответе не соответствует ожидаемому
INVALID_CRC ⁷	31	Несовпадение контрольной суммы ответа
INVALID_STOP_CHAR	32	Некорректный стоповый символ в ответе

```

{attribute 'qualified_only'}
{attribute 'strict'}
// Ошибки протокола DCON (при реализации с помощью библиотеки OwenCommunication)
TYPE DCON_ERROR_OCL :
(
    // Стандартные коды ошибок для ФБ COM\_Control и UNM\_SerialRequest из OCL.ERROR
    NO_ERROR                := 0,
    TIME_OUT                := 1,
    HANDLE_INVALID          := 10,
    ERROR_UNKNOWN           := 11,
    WRONG_PARAMETER         := 12,
    WRITE_INCOMPLETE        := 13,
    INVALID_DATAPOINTER     := 20,
    INVALID_DATASIZE        := 21,
    // Специфические ошибки протокола DCON
    //
    // Стартовый символ в ответе не соответствует ожидаемому
    NOT_EXPECTED_START_CHAR := 30,
    // Несовпадение контрольной суммы ответа
    INVALID_CRC              := 31,
    // Некорректный стоповый символ в ответе
    INVALID_STOP_CHAR        := 32
) INT; // для гарантированного совпадения по размеру с OCL.ERROR
END_TYPE

```

⁷ Контрольная сумма протокола DCON не является циклической и поэтому формально некорректно обозначать ее как CRC. Но поскольку аббревиатура CRC является лаконичной и понятной, то далее в документе она используется для обозначения контрольных сумм независимо от их типа.

6.4.2 Перечисление DCON_PARAMS

Перечисление **DCON_PARAMS** содержит коды команд протокола DCON, поддерживаемых в рамках примера. В зависимости от конкретной команды под указателями входов **pRequestData** и **pResponseData** ФБ [DCON_MASTER_OCL](#) размещаются переменные различного типа.

Таблица 6.4.2 – Описание элементов перечисления DCON_PARAMS

Название	Значение	Описание	Тип переменной по указателю pRequestData	Тип переменной по указателю pResponseData
USER_DEFINED	0	Произвольный запрос. Параметры запроса задаются пользователем при вызове ФБ	STRING (см. спецификацию протокола для конкретного устройства)	STRING(255) (см. спецификацию протокола для конкретного устройства)
OWEN_MV110_xA_READ_ALL	1	ОВЕН MB110-xA(C): групповое чтение всех аналоговых входов	-	ARRAY [1..x] OF REAL
OWEN_MV110_xA_READ_CHANNEL	2	ОВЕН MB110-xA(C): запрос конкретного аналогового входа	UINT	REAL

Для запроса **USER_DEFINED** в данных запроса передается строка, содержащая команду и ее данные. Содержимое строки полностью определяется пользователем – это дает ему возможность отправить произвольный запрос, не поддерживаемый разработчиком блока. Поскольку в разных запросах могут использоваться разные стартовые символы – то в данном случае стартовый символ задается с помощью входа **sUserDefinedStartChar**. Ответ предоставляется в виде строки без какого-либо разбора (со стартовым символом, контрольной суммой (если она включена) и стоповым символом).

Запрос **OWEN_MV110_xA_READ_ALL** (групповое чтение всех аналоговых входов) не требует никаких аргументов. Ответ предоставляется в виде массива переменных типа **REAL** (размерность массива зависит от количества аналоговых входов модуля).

Для запроса **OWEN_MV110_xA_READ_CHANNEL** в данных запроса передается номер считываемого входа. Ответ предоставляется в виде переменной типа **REAL**.

```
{attribute 'qualified_only'}
{attribute 'strict'}
// Команды протокола DCON
TYPE DCON_PARAMS :
(
  // Пользовательский запрос.
  // pRequestData - STRING (см. документацию на прибор), pResponseData - STRING(255)
  // (см. документацию на прибор)
  USER_DEFINED := 0,
  // ОВЕН MB110-xA(C): групповой запрос всех аналоговых входов
  // pRequestData - не используется, pResponseData - указатель на ARRAY [1..x] OF
  // REAL
  OWEN_MV110_xA_READ_ALL := 1,
  // ОВЕН MB110-xA(C): запрос конкретного аналогового входа
  // pRequestData - указатель на UINT, pResponseData - указатель на REAL
  OWEN_MV110_xA_READ_CHANNEL := 2
);
END_TYPE
```

6.4.3 ФБ DCON_MASTER_OCL

Функциональный блок **DCON_MASTER_OCL** используется для опроса устройств по протоколу DCON.

Описание работы блока:

По переднему фронту на входе **xExecute** происходит отправка запроса устройству с адресом **usiDeviceAddr** через COM-порт, определяемый дескриптором **hCom**, полученным от ФБ [COM_Control](#). Выбор команды осуществляется с помощью входа **eParam**, представляющего собой перечисление типа [DCON_PARAMS](#). Для отправки произвольного запроса вход **eParam** должен иметь значение **DCON_PARAMS.USER_DEFINED**. В этом случае стартовый символ запроса определяется значением входа **sUserDefinedStartChar**. Если в запросе должны быть переданы код команды и записываемые данные – они размещаются по указателю **pRequestData**, а вход **szRequestData** определяет размер буфера записываемых данных в байтах. Описание параметров конкретного запроса можно найти в спецификации конкретного устройства.

В случае получения корректного ответа выход **xDone** принимает значение **TRUE**. Полученные данные помещаются в буфер, расположенный по указателю **pResponseData** и имеющий размер **szResponseData** байт. В описании перечисления [DCON_PARAMS](#) приведены типы данных переменных, которые должны быть размещены под указателем для конкретного запроса.

Если ответ на запрос не был получен за время **tTimeout**, то выход **xError** принимает значение **TRUE**, а выход **eError** = **TIME_OUT**. В случае получения некорректного ответа выход **xError** принимает значение **TRUE**, а на выходе **eError** отображается код ошибки из перечисления [DCON_ERROR_OCL](#).

Если вход **xUseCrc** имеет значение **TRUE**, то при формировании запроса и разборе ответа используется контрольная сумма; если вход имеет значение **FALSE**, то контрольная сумма не используется.

Для отправки нового запроса следует создать передний фронт на входе **xExecute**.

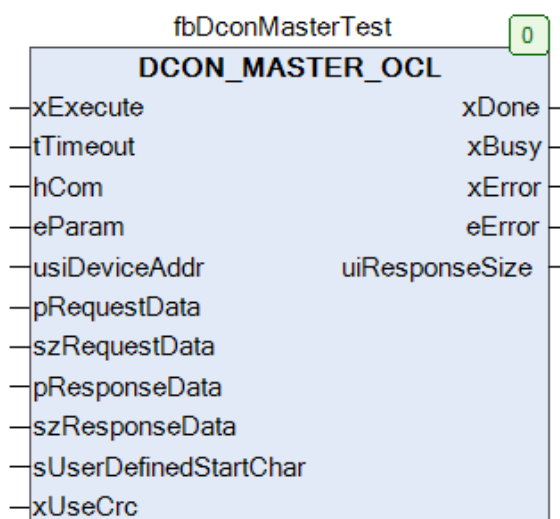


Рисунок 6.4.2 – Внешний вид ФБ DCON_MASTER_OCL на языке CFC

Ниже приведено описание входов/выходов/локальных переменных/констант блока и его листинг с комментариями к реализации.

Таблица 6.4.3 – Описание переменных ФБ DCON_MASTER_OCL

Название	Тип	Описание
Входы		
xExecute	BOOL	По переднему фронту происходит однократная отправка запроса
tTimeout	TIME	Таймаут ожидания ответа от slave-устройства (T#0ms – время ожидания не ограничено)
hCom	CAA.HANDLE	Дескриптор COM-порта, полученный от ФБ COM_Control
eParam	DCON_PARAMS	Команда протокола DCON, используемая в запросе
usiDeviceAddr	USINT	Адрес устройства
pRequestData	CAA.PVOID	Указатель на буфер запроса
szRequestData	CAA.SIZE	Размер буфера запроса в байтах
pResponseData	CAA.PVOID	Указатель на буфер ответа
szResponseData	CAA.SIZE	Размер буфера ответа в байтах
sUserDefinedStartChar	STRING(1)	Стартовый символ для произвольного запроса (DCON_PARAMS.USER_DEFINED)
xUseCrc	BOOL	TRUE – использовать контрольную сумму для запросов и ответов, FALSE – не использовать
Выходы		
xDone	BOOL	TRUE – получен корректный ответ
xBusy	BOOL	TRUE – ФБ находится в работе
xError	BOOL	Принимает значение TRUE в случае возникновения ошибки
eError	DCON_ERROR_OCL	Статус работы ФБ (или код ошибки)
uiResponseSize	UINT	Размер полученного ответа в байтах
Локальные переменные		
fbUnmSerialRequest	OCL.UNM_SerialRequest	Коммуникационный ФБ из библиотеки OwenCommunication . Реализует отработку произвольного набора байт и получения ответа
fbExecute	R_TRIG	Детектор запуска блока
xNeedReset	BOOL	TRUE – нужно произвести переинициализацию блока
sStartChar	STRING(c_usiStartCharLen)	Стартовый символ запроса
sExpectedStartChar	STRING(c_usiStartCharLen)	Ожидаемый стартовый символ ответа
xlInitParams	BOOL	TRUE – инициализация параметров проведена успешно, FALSE – ошибка инициализации параметров
xWrongParameter	BOOL	Флаг ошибки инициализации параметров, удерживается до заднего фронта на xExecute . Используется для запрета вызова fbUnmSerialRequest при некорректных параметрах запроса
sCommand	STRING	Команда запроса (вместе с аргументами)

sRequest	STRING ⁸	Буфер запроса
xlsValidPointerForRequest	BOOL	TRUE – выделен буфер запроса (входы pRequestData и szRequestData имеют корректные значения)
xlsValidPointerForResponse	BOOL	TRUE – выделен буфер ответа (входы pResponseData и szResponseData имеют корректные значения)
sDeviceAddr	STRING(c_usiAddrLen)	Адрес устройства в виде HEX-строки
sResponse	STRING(255) ⁸	Буфер ответа

Таблица 6.4.4 – Описание констант ФБ DCON_MASTER_OCL

Название	Тип	Значение	Описание
c_usiStartCharLen	USINT	1	Длина (в символах) стартовой последовательности символов
c_usiAddrLen	USINT	2	Длина адреса (в символах)
c_usiCrcLen	USINT	2	Длина контрольной суммы (в символах)
c_usiStopCharLen	USINT	1	Длина (в символах) стоповой последовательности символов
c_sStopChar	STRING (c_usiStopCharLen)	'\$0D'	Стоповый символ протокола DCON
c_usiMaxAiCount	USINT	32	Максимальное число входов аналогового модуля (выбрано эмпирически)

Константы позволяют избежать в коде использования «[магических чисел](#)», заменив их более понятными символьными именами. Первые 4 константы определяют размеры полей запроса/ответа (стартовый символ, контрольная сумма и т. д.). Константа **c_sStopChar** представляет собой стоповый символ протокола DCON. Константа **c_usiMaxAiCount** не имеет явного отношения к протоколу – она используется для определения размера буферов данных, используемых при разборе ответа на групповой запрос чтения аналоговых входов. Поскольку разные модули могут иметь различное число аналоговых входов, а стандарт МЭК 61131-3 декларирует статическую типизацию и статическое выделение памяти – то следует выбрать некоторое максимальное значение. В рамках примера используется число 32 (*сложно представить, что пользователю встретится модуль с поддержкой протокола DCON, который имеет большее количество аналоговых входов*).

```
{attribute 'qualified_only'}
// ФБ для опроса устройств по протоколу DCON
FUNCTION_BLOCK DCON_MASTER_OCL
VAR_INPUT
    // По переднему фронту происходит однократная отправка запроса
```

⁸ Так как у протокола DCON отсутствует единая спецификация, то максимальный размер запроса и ответа зависят от конкретного устройства. В рамках примера для буфера запроса выбран тип STRING, а для буфера ответа – STRING(255). Таким образом, подразумевается, что запрос может включать в себя до 80 символов (*это значение автоматически применяется для типа STRING, если в скобках после него не указано конкретное ограничение*), а ответ – до 255 символов. Для подавляющего большинства приборов с поддержкой протокола DCON эти ограничения будут выполняться.

```

xExecute:                BOOL;
// Таймаут ожидания ответа (0 - время не ограничено)
tTimeout:                TIME;
// Дескриптор COM-порта, полученный от ФБ OCL.COM CONTROL
hCom:                    CAA.HANDLE;
// Команда протокола DCON
eParam:                   DCON\_PARAMS;
// Адрес устройства
usiDeviceAddr:            USINT;
// Указатель на буфер данных запроса
// (см. описание нужных типов данных в DCON_PARAMS)
pRequestData:             CAA.PVOID;
// Размер буфера данных запроса в байтах
szRequestData:            CAA.SIZE;
// Указатель на буфер данных ответа
// (см. описание нужных типов данных в DCON_PARAMS)
pResponseData:            CAA.PVOID;
// Размер буфера данных ответа в байтах
szResponseData:           CAA.SIZE;
// Стартовый символ пользовательского запроса (USER_DEFINED)
sUserDefinedStartChar:    STRING(c_usiStartCharLen);
// TRUE - использовать контрольную сумму для запросов и ответов,
// FALSE - не использовать
xUseCrc:                   BOOL := TRUE;
END_VAR
VAR_OUTPUT
// TRUE - получен корректный ответ
xDone:                     BOOL;
// TRUE - блок находится в работе
xBusy:                     BOOL;
// Флаг ошибки
xError:                    BOOL;
// Код ошибки
eError:                     DCON\_ERROR\_OCL;
// Размер ответа в байтах
uiResponseSize:            UINT;
END_VAR
VAR
// Коммуникационный ФБ из библиотеки OwenCommunication.
// Реализует отправку произвольного набора байт и получения ответа
fbUnmSerialRequest:        OCL.UNM SerialRequest;

// Детектор запуска блока
fbExecute:                 R_TRIG;
// TRUE - нужно произвести переинициализацию блока
xNeedReset:                BOOL;
// Стартовый символ запроса
sStartChar:                STRING(c_usiStartCharLen);
// Ожидаемый стартовый символ ответа
sExpectedStartChar:        STRING(c_usiStartCharLen);
// TRUE - инициализация параметров проведена успешно,
// FALSE - ошибки инициализации параметров
xIsInitParams:             BOOL;
// Флаг "ошибки инициализации параметров", удерживается до заднего фронта
// на xExecute.
// Используется для запрета вызова fbUnmSerialRequest при некорректных параметрах
// запроса
xWrongParameter:           BOOL;
// Команда запроса (вместе с аргументами)
sCommand:                  STRING;
// Буфер запроса
sRequest:                  STRING;

// TRUE - выделен буфер запроса
// (входы pRequestData и szRequestData имеют корректные значения)
xIsValidPointerForRequest:  BOOL;
// TRUE - выделен буфер ответа
// (входы pResponseData и szResponseData имеют корректные значения)
xIsValidPointerForResponse:  BOOL;

```

```

// Адрес устройства в виде HEX-строки
sDeviceAddr:          STRING(c_usiAddrLen);
// Буфер ответа
sResponse:            STRING(255);
END_VAR
VAR CONSTANT
// Длина (в символах) стартовой последовательности символов
c_usiStartCharLen:    USINT := 1;
// Длина адреса (в символах)
c_usiAddrLen:         USINT := 2;
// Длина контрольной суммы (в символах)
c_usiCrcLen:          USINT := 2;
// Длина (в символах) стоповой последовательности символов
c_usiStopCharLen:     USINT := 1;
// Стоповый символ протокола DCON
c_sStopChar:          STRING(c_usiStopCharLen) := '$0D';

// Максимальное число каналов аналогового модуля (выбрано эмпирически)
c_usiMaxAiCount:      USINT := 32;
END_VAR

//
// код ФБ
//

// [1]
// переинициализация блока перед новым вызовом
IF NOT(xExecute) AND xNeedReset THEN

    // сбрасываем выходы блока
    xDone := FALSE;
    xBusy := FALSE;
    xError := FALSE;
    eError := DCON_ERROR_OCL.NO_ERROR;
    uiResponseSize := 0;

    // очищаем буферы ответа и команды
    // причины:
    // так как fbUnmSerialRequest возвращает ответ по указателю
    // - то текущий ответ может "склеиться" с концом предыдущего ответа
    // если новый ответ короче прошлого
    MEM.MemFill(ADR(sResponse), SIZEOF(sResponse), 0);
    // так как есть запросы, в которых команда отсутствует
    // (например, запрос OWEN_MV110_xA_READ_ALL включает в себя только адрес модуля)
    // кроме того, в USER_DEFINED команда передается по указателю,
    // и может "склеиться" с концом предыдущей команды
    // если новая команда короче предыдущей
    MEM.MemFill(ADR(sCommand), SIZEOF(sCommand), 0);

    // очищать sRequest нет смысла, так как он формируется только строковыми функциями
    // без использования указателей
    // поэтому "склеиться" там ничего не может

    // сбрасываем флаг некорректной инициализации
    xWrongParameter := FALSE;

    // сбрасываем коммуникационный блок для возможности его следующего вызова
    fbUnmSerialRequest(xExecute := FALSE);

    // переинициализация завершена
    xNeedReset := FALSE;

END_IF
// [2]
// детектор запуска блока
fbExecute(CLK := xExecute);

IF fbExecute.Q THEN

```

```

// попытка инициализации параметров запроса
xIsInitParams := prvInitParams();

// инициализация прошла успешно - поэтому формируем запрос
IF xIsInitParams THEN
    prvCreateRequest();
    // в процессе инициализации произошла ошибка - блок не будет вызван
ELSE
    xError := TRUE;
    eError := DCON_ERROR_OCL.WRONG_PARAMETER;
    xWrongParameter := TRUE;
    xNeedReset := TRUE;
END_IF

END_IF

// [3]
// вызов коммуникационного ФБ
fbUnmSerialRequest
(
    xExecute      := xExecute AND NOT(xWrongParameter),
    tTimeout      := tTimeout,
    hCom          := hCom,
    pRequest      := ADR(sRequest),
    szRequest     := TO_UDINT(LEN(sRequest) ),
    pResponse     := ADR(sResponse),
    szResponse    := SIZEOF(sResponse),
    xBusy         => xBusy
);

// [4]
// получен ответ - надо его обработать
IF fbUnmSerialRequest.xDone THEN

    uiResponseSize := fbUnmSerialRequest.uiResponseSize;

    prvParseResponse();
    // сброс блока будет выполнен после того, как xExecute примет значение FALSE
    // (см. начало кода ФБ)
    xNeedReset := TRUE;

// произошла ошибка (ошибка таймаута или получен некорректный)
ELSIF fbUnmSerialRequest.xError THEN

    xError := TRUE;
    // мы не можем "напрямую" присвоить значение из OCL.ERROR в DCON_ERROR
    // можно было бы написать метод или функцию для этого
    // а можно и просто "переложить" содержимое экземпляра одного перечисления
    // в экземпляр другого
    // но важно убедиться, что размер обоих перечислений совпадает
    MEM.MemMove(ADR(fbUnmSerialRequest.eError), ADR(eError), SIZEOF(eError) );
    // сброс блока будет выполнен после того, как xExecute примет значение FALSE
    // (см. начало кода ФБ)
    xNeedReset := TRUE;

END_IF

```

В коде ФБ можно выделить 4 фрагмента – в листинге они помечены маркерами [1], [2], [3] и [4]. Ниже приведено их описание.

В фрагменте [1] выполняется переинициализация блока – сброс выходов и некоторых локальных переменных, а также очистка буферов. Переинициализация производится после завершения работы блока (в этот момент переменной **xNeedReset** присваивается значение **TRUE**) и только в том случае, если к этому моменту вход **xExecute** уже имеет значение **FALSE** (это необходимо для соответствия [модели поведения](#)).

Обратите внимание, что хотя буферы и являются строковыми переменными, их очистка осуществляется не с помощью присваивания пустой строки (в стиле `sResponse := ""`), а с помощью функции **MemFill** из библиотеки [CAA Memory](#). Это связано с тем, что операция присваивания пустой строки лишь записывает в первый символ строки нуль-символ (символ окончания строки). Содержимое остальных символов при этом не изменяется. Так как работа с буферами производится с помощью указателей, то при перезаписи начального фрагмента строки, включающего нуль-символ, эти оставшиеся символы становятся частью новой строки – и, таким образом, строка включает как «новые», так и часть «старых» символов. Поэтому для очистки строкового буфера следует использовать функцию **MemFill**, которая обнуляет все байты строки.

В фрагменте [2] производится подготовка к отправке нового запроса. За счет детектора переднего фронта весь код подготовки выполняется только один раз (а не в каждом цикле ПЛК) – это позволяет не нагружать контроллер лишними операциями. Подготовка включает в себя два действия:

- инициализация параметров запроса с помощью метода **prvInitParams**. Код метода будет рассмотрен в [п. 6.4.4](#);
- формирование строки запроса с помощью метода **prvCreateRequest**. Код метода будет рассмотрен в [п. 6.4.5](#).

Инициализация параметров включает в себя определение стартового символа для конкретного запроса, команды, данных команды и т. д. Инициализация может завершиться неудачей – например, если при использовании команды чтения конкретного входа по указателю **pRequestData** не размещается номер этого входа. В этом случае формирование запроса даже не начинается – происходит генерация ошибки **WRONG_PARAMETER**.

В фрагменте [3] происходит вызов экземпляра коммуникационного ФБ [UNM_SerialRequest](#) из библиотеки [OwenCommunication](#). Вызов производится только в том случае, если в прошлом фрагменте был успешно сформирован запрос – иначе блок не будет вызван, так как переменная **xWrongParameter** будет иметь значение **TRUE**.

В фрагменте [4] происходит обработка результатов вызова экземпляра коммуникационного ФБ – либо на запрос получен ответ (выход **xDone** принял значение **TRUE**) и тогда происходит его разбор с помощью метода **prvParseResponse** (код метода будет рассмотрен в [п. 6.4.6](#)), либо в процессе выполнения произошла ошибка (выход **xError** принял значение **TRUE**) – тогда происходит генерация ошибки на выходе **eError** блока **DCON_MASTER_OCL**.

При этом выходы **eError** блоков [UNM_SerialRequest](#) и **DCON_MASTER_OCL** имеют разные типы (**OCL.ERROR** и [DCON_ERROR_OCL](#)), но все ошибки перечисления **OCL.ERROR**, которые могут возникнуть в процессе работы ФБ [UNM_SerialRequest](#), присутствуют и в [DCON_ERROR_OCL](#). Тем не менее, присвоить значение экземпляра одного перечисления экземпляру другого нельзя.

```
// этот код вызовет ошибку компиляции, потому что переменные являются экземплярами...
// ...разных перечислений
eError := fbUnmSerialRequest.eError;
```

Поэтому выполняется копирование содержимого одной переменной в другую с помощью функции **MemMove** из библиотеки [CAA Memory](#).

6.4.4 Метод prvInitParams

Метод **prvInitParams** проверяет корректность выделения памяти буферов данных запроса и ответа и в зависимости от выбранной пользователем команды формирует значения следующих локальных переменных ФБ [DCON_MASTER_OCL](#):

- **sStartChar** – стартовый символ запроса;
- **sCommand** – код команды и ее аргументы (если они требуются);
- **sExpectedStartChar** – стартовый символ ожидаемого корректного ответа (поскольку он зависит от кода команды – то проще определить его заранее на этом этапе, чтобы избежать дублирования кода в методе разбора ответа).

Проверка выделения памяти связана с тем, что аргументы запроса передаются в ФБ [DCON_MASTER_OCL](#) по указателю **pRequestData**. Данные, выделенные из ответа, размещаются по указателю **pResponseData**. Соответственно, если эти указатели не будут проинициализированы при вызове экземпляра блока – то в процессе его работы может произойти [ошибка сегментации памяти](#). Поэтому перед использованием указателей необходимо проверить их корректность. При этом для разных запросов критерии корректности могут быть разными – например, запрос на групповое чтение аналоговых входов не включает в себя команду, и поэтому вход **pRequestData** в данном случае может не использоваться, а запрос на чтение конкретного входа в качестве аргумента принимает номер этого входа, так что в этом случае указатель **pRequestData** должен быть проинициализирован. В рамках примера проверка указателя осуществляется путем сравнения его значения с константой **CAA.Constants.gc_pNULL** из библиотеки [CAA Types](#) (фактически – выполняется проверка на неравенство нулю).

Входы **szRequestData** и **szResponseData**, определяющие размеры буферов, также проверяются на неравенство нулю с помощью сравнения с константой **CAA.Constants.gc_szZERO**. В рамках примера других проверок размера буфера не производится, но в реальном проекте они могут быть добавлены для каждого запроса – например, если известно, что в ответ на запрос придет информация, которая после обработки занимает 60 байт данных, то размер буфера **szResponseData** не должен быть менее 60 байт.

В случае успешных проверок и инициализации параметров метод возвращает значение **TRUE**, в случае ошибок – **FALSE**.

```
// метод инициализации параметров запросов
// TRUE - инициализация завершена успешно, FALSE - возникли ошибки
// (например, передан нулевой указатель в запросе, где это неприемлемо)
METHOD prvInitParams :          BOOL
VAR
    // промежуточные переменные для конкретных запросов

    // число каналов для запроса OWEN_MV110_xA_READ_CHANNEL
    m_usiAiChannelNumber: USINT(1..c_usiMaxAiCount);
END_VAR

// проверяем указатели и размеры буферов функционального блока на валидность
xIsValidPointerForRequest := (pRequestData <> CAA.Constants.gc_pNULL) AND
    (szRequestData > CAA.Constants.gc_szZERO);
xIsValidPointerForResponse := (pResponseData <> CAA.Constants.gc_pNULL) AND
    (szResponseData > CAA.Constants.gc_szZERO);
```

```

CASE eParam OF

    // пользовательский запрос
    DCON_PARAMS.USER_DEFINED:
        // в данном случае стартовый символ определяется пользователем
        sStartChar := sUserDefinedStartChar;

        IF xIsValidPointerForRequest THEN
            MEM.MemMove(pRequestData, ADR(sCommand), SIZEOF(sCommand) );
        END_IF

        // так как запрос может не включать команды и не подразумевать ответа -
        // то инициализация всегда считается корректной
        prvInitParams := TRUE;

    // ОВЕН MB110-xA: групповой запрос всех аналоговых каналов
    DCON_PARAMS.OWEN_MV110_xA_READ_ALL:

        sStartChar := '#';
        sCommand := '';
        sExpectedStartChar := '>';

        // на этот запрос придет ответ - так что под него должен быть выделен буфер
        IF xIsValidPointerForResponse THEN
            prvInitParams := TRUE;
        END_IF

    // ОВЕН MB110-xA: запрос конкретного аналогового канала
    DCON_PARAMS.OWEN_MV110_xA_READ_CHANNEL:

        sStartChar := '#';
        sExpectedStartChar := '>';

        // в этом запросе есть аргумент (номер канала) и на него придет ответ -
        // так что нужно выделить буферы под данные запросы и ответа
        IF xIsValidPointerForRequest AND xIsValidPointerForResponse THEN
            // копируем номер канала в промежуточную переменную
            MEM.MemMove(pRequestData, ADR(m_usiAiChannelNumber),
                SIZEOF(m_usiAiChannelNumber) );
            // нумерация каналов на корпусе модуля - 1..8,
            // а в реализации протокола - 0..7
            // поэтому делаем -1
            sCommand := TO_STRING(m_usiAiChannelNumber - 1);

            prvInitParams := TRUE;
        END_IF

ELSE
    ; // здесь можно реализовать обработку ошибки типа INVALID_COMMAND
    // она может произойти, например, если из-за ошибок при работе с памятью
    // в переменную eParam попало значение, выходящее за диапазон перечисления
END_CASE

```

6.4.5 Метод prvCreateRequest

Метод **prvCreateRequest** формирует запрос протокола DCON на основании отдельных полей. Поля стартового символа и команды (а также аргументов команды) определяются в методе [prvInitParams](#). Стоповый символ определяется константой **c_sStopChar** ФБ [DCON_MASTER_OCL](#). Поэтому данный метод выполняет следующие операции:

- конвертацию адреса (вход **usiDeviceAddr** ФБ [DCON_MASTER_OCL](#)) из числового вида в HEX-строку (это обусловлено требованиями протокола);
- расчет контрольной суммы запроса;
- объединение всех полей запроса в одну строку.

Для конвертации адреса в HEX-строку используется функция **WORD_TO_HEX_STR** из библиотеки [OwenStringUtils](#). Поскольку адрес всегда занимает два символа, то может потребоваться дополнить его ведущим нулем. Для этого используется функция **ADD_CHAR** из той же библиотеки.

Алгоритм расчета контрольной суммы описан в [п. 6.2](#). Контрольная сумма представляет собой сумму всех ASCII-кодов запроса, предшествующих стоповому символу. Поскольку, как уже упоминалось, многие устройства с поддержкой DCON позволяют отключить проверку контрольной суммы – то она формируется только в том случае, если вход **xUseCrc** ФБ [DCON_MASTER_OCL](#) имеет значение **TRUE**. Для конвертации контрольной суммы в HEX-строку так же, как и для адреса используются функции **WORD_TO_HEX_STR** и **ADD_CHAR**.

Отдельные поля запроса «склеиваются» в одну строку **sRequest** с помощью функций **CONCAT4** (библиотека [OwenStringUtils](#)) и **CONCAT** (библиотека [Standard](#)). Их отличие заключается в том, что **CONCAT4** позволяет объединить за один вызов четыре строки, а **CONCAT** – только две.

```
// метод формирования запроса
METHOD prvCreateRequest
VAR
    // контрольная сумма запроса в численном виде
    m_sCrc:          STRING(c_usiCrcLen);
    // контрольная сумма запроса в виде HEX-строки
    m_byCrc:         BYTE;
    // счетчик цикла
    i:               INT;
END_VAR

// конвертируем адрес в HEX-строку в верхнем регистре
// так как адрес всегда занимает c_usiAddrLen символов -
// может потребоваться дополнить его ведущими нулями
sDeviceAddr := OSU.WORD_TO_HEX_STR(wInput := usiDeviceAddr,
    xUpperCase := TRUE, sPrefix := '');
sDeviceAddr := OSU.ADD_CHAR(sInputString := sDeviceAddr,
    usiTargetLen := c_usiAddrLen, sAddChar := '0', xRight := FALSE);

sRequest := OSU.CONCAT4(sStartChar, sDeviceAddr, sCommand);

// если включена обработка CRC - добавим ее в ответ
IF xUseCrc THEN

    m_byCrc := 0;

    FOR i := 0 TO LEN(sRequest) - 1 DO
        m_byCrc := m_byCrc + sRequest[i];
    END_FOR

    // конвертируем CRC в HEX-строку в верхнем регистре
    m_sCrc := OSU.WORD_TO_HEX_STR(wInput := m_byCrc, xUpperCase := TRUE, sPrefix := '');
END_IF
```

```
// так как CRC всегда занимает c_usiCrcLen символов -
// может потребоваться дополнить ее ведущими нулями
m_sCrc := OSU.ADD_CHAR(sInputString := m_sCrc,
    usiTargetLen := c_usiCrcLen, sAddChar := '0', xRight := FALSE);

sRequest := OSU.CONCAT4(sRequest, m_sCrc, c_sStopChar);

ELSE
    // если CRC не обрабатывается - сразу добавляем стоп-символ
    sRequest := CONCAT(sRequest, c_sStopChar);

END_IF
```

6.4.6 Метод prvParseResponse

Метод **prvParseResponse** используется для проверки корректности ответа, выделения из него данных и их преобразования к нужному пользователю виду (например, числа с плавающей точкой удобно предоставлять пользователю в виде переменных типа **REAL**).

Проверка ответа начинается с расчета контрольной суммы (алгоритм расчета был описан в предыдущем пункте). Контрольная сумма, размещенная в самом ответе, копируется в переменную **m_sResponseCrc**.

Кроме проверки контрольной суммы выполняется проверка стартового и стопового символа ответа. Стартовый символ ответа должен соответствовать ожидаемому (заданному для конкретного запроса в методе [prvInitParams](#)), а стоповый – протоколу DCON (константе **c_sStopChar**). Если хотя бы одна из проверок не выполняется – то генерируется ошибка и происходит выход из метода с помощью оператора **RETURN**, потому что разбирать некорректный ответ не имеет смысла.

Если же ошибок не возникло, то начинается разбор ответа. Для каждого запроса эта процедура происходит по-разному.

Для пользовательского запроса (**USER_DEFINED**) в рамках примера вся строка ответа возвращается по указателю **pResponseData** без какой-либо обработки. В рамках реального проекта реализация могла бы быть иной – например, можно было бы возвращать структуру, содержащую стартовый символ и данные ответа (без контрольной суммы и стопового символа).

Для группового запроса всех аналоговых входов (**OWEN_MV110_xA_READ_ALL**) разбор ответа выполняется с помощью метода [prvConvertGroupAiToReal](#). Разумно выделять код обработки результата каждого сложного запроса в отдельный метод, чтобы не перегружать код метода **prvParseResponse**. Метод [prvConvertGroupAiToReal](#) всегда возвращает массив из 32 значений (это связано со значением константы **c_usiMaxAiCount** – на этапе проектирования было сформулировано, что модулей с поддержкой протокола DCON и большим числом аналоговых входов, вероятно, не существует). Но пользователю по указателю **pResponseData** возвращается ровно столько значений, сколько он ожидает – за счет того, что в вызове функции **MemMove** число копируемых байт определяется значением **szResponseData**, которое задает сам пользователь на входе ФБ [DCON_MASTER_OCL](#).

Для запроса чтения конкретного аналогового входа (**OWEN_MV110_xA_READ_CHANNEL**) разбор ответа осуществляется следующим образом: известно, что значение в ответе начинается сразу после стартового символа. Хотя для модулей ОВЕН жестко определено, что длина значения всегда составляет 7 символов (знак, 5 цифр и разделитель-точка) – в целях универсальности разбор ответа

производится без этого предположения: значение «вырезается» с помощью функции **MID** (из библиотеки [Standard](#)) между стартовым символом и контрольной суммой (если включена обработка контрольной суммы, т. е. вход **xUseCrc** имеет значение **TRUE**) или между стартовым и стоповым символом (если обработка контрольной суммы отключена). После этого значение конвертируется из строки (**STRING**) в число с плавающей точкой (**REAL**) и возвращается по указателю **pResponseData**.

После завершения разбора ответа выходу **xDone** присваивается значение **TRUE**, чтобы проинформировать пользователя об успешном завершении работы блока.

```
// метод разбора ответа
METHOD prvParseResponse
VAR
    // рассчитанная контрольная сумма ответа в численном виде
    m_byCalcCrc:      BYTE;
    // рассчитанная контрольная сумма ответа в виде HEX-строки
    m_sCalcCrc:       STRING(c_usiCrcLen);
    // контрольная сумма, извлеченная из ответа, в виде HEX-строки
    m_sResponseCrc:   STRING(c_usiCrcLen);
    // длина ответа в символах
    m_iResponseLen:   INT;
    // счетчик цикла
    i:                INT;
    // буфер для манипуляций данными
    m_sBuffer:        STRING(255);
    // длина значения из ответа в символах
    m_iValueLen:      INT;

    // переменные для конкретных запросов

    // значение одного аналогового канала
    m_rAiValue:       REAL;
    // группа значений аналоговых каналов
    m_arAiValues:     ARRAY [1..c_usiMaxAiCount] OF REAL;
END_VAR

m_iResponseLen := LEN(sResponse);

// если надо - рассчитаем контрольную сумму
IF xUseCrc THEN

    m_byCalcCrc := 0;

    // CRC рассчитывается по всем символам ответа, кроме самой CRC и стоп символа
    // -1 нужен, так как байты строки нумеруются с 0, а символы - с 1
    FOR i := 0 TO m_iResponseLen - c_usiCrcLen - c_usiStopCharLen - 1 DO
        m_byCalcCrc := m_byCalcCrc + sResponse[i];
    END_FOR

    // конвертируем CRC в HEX-строку в верхнем регистре
    m_sCalcCrc := OSU.WORD_TO_HEX_STR(wInput := m_byCalcCrc,
        xUpperCase := TRUE, sPrefix := '');
    // так как CRC всегда занимает c_usiCrcLen символов
    // - может потребоваться дополнить ее ведущими нулями
    m_sCalcCrc := OSU.ADD_CHAR(sInputString := m_sCalcCrc,
        usiTargetLen := c_usiCrcLen, sAddChar := '0', xRight := FALSE);

    // позиция начала CRC вычисляется без учета стоп-символа, и это корректно:
    // например, длина ответа = 10, длина CRC = 2, длина стоп-символа = 1
    // тогда 10-2 = 8, т.е. CRC расположена в символах 8 и 9,
    // а стоп-символ имеет позицию 10
    m_sResponseCrc := MID(sResponse, c_usiCrcLen, m_iResponseLen - c_usiCrcLen);

END_IF
```

```

// проверка стартового символа
IF eParam <> DCON_PARAMS.USER_DEFINED AND sResponse[0] <> sExpectedStartChar[0] THEN
    xError := TRUE;
    eError := DCON_ERROR_OCL.NOT_EXPECTED_START_CHAR;
// проверка стопового символа
ELSIF xUseCrc AND (m_sCalcCrc <> m_sResponseCrc) THEN
    xError := TRUE;
    eError := DCON_ERROR_OCL.INVALID_CRC;
// проверка контрольной суммы
// -1 нужен, так как байты строки нумеруются с 0, а символы - с 1
ELSIF sResponse[m_iResponseLen - 1] <> c_sStopChar[0] THEN
    xError := TRUE;
    eError := DCON_ERROR_OCL.INVALID_STOP_CHAR;
END_IF

// если в ответе есть ошибка - то нет смысла его парсить
IF xError THEN
    RETURN;
END_IF

CASE eParam OF

    // пользовательский запрос
    DCON_PARAMS.USER_DEFINED:

        // возвращаем ответ без каких-либо обработок
        MEM.MemMove(ADR(sResponse), pResponseData, TO_UINT(szResponseData) );

        // ОВЕН MB110-xA: групповой запрос всех аналоговых каналов
        // пример ответа (с CRC):
        // >+499.98+33.758+49.998+33.880+50.000+49.998+34.601+34.65252$0D
        DCON_PARAMS.OWEN_MV110_xA_READ_ALL:

            m_arAiValues := prvConvertGroupAiToReal\(\);
            MEM.MemMove(ADR(m_arAiValues), pResponseData, TO_UINT(szResponseData) );

        // ОВЕН MB110-xA: запрос конкретного аналогового канала
        // пример ответа (с CRC): >+49.998AE$0D
        DCON_PARAMS.OWEN_MV110_xA_READ_CHANNEL:

            // учитываем факт использования контрольной суммы
            // при определении длины значения канала
            IF xUseCrc THEN
                m_iValueLen := m_iResponseLen - c_usiStartCharLen - c_usiCrcLen -
                    c_usiStopCharLen;
            ELSE
                m_iValueLen := m_iResponseLen - c_usiStartCharLen - c_usiStopCharLen;
            END_IF

            // значение расположено сразу после стартового символа
            m_sBuffer := MID(sResponse, m_iValueLen, c_usiStartCharLen + 1);
            m_rAiValue := TO_REAL(m_sBuffer);
            MEM.MemMove(ADR(m_rAiValue), pResponseData, TO_UINT(szResponseData) );

END_CASE

xDone := TRUE;

```

6.4.7 Метод prvConvertGroupAiToReal

Метод **prvConvertGroupAiToReal** используется для выделения данных из ответа на групповой запрос чтения всех аналоговых входов модуля. Поскольку разные модули могут иметь различное число аналоговых входов, а стандарт МЭК 61131-3 декларирует статическую типизацию и статическое выделение памяти – то следует задаться выбрать некоторое максимальное значение. В рамках примера используется константа **c_usiMaxAiCount** со значением 32 (сложно представить, что пользователю встретится модуль с поддержкой протокола DCON, который имеет большее количество аналоговых входов). Поэтому метод возвращает массив из 32 элементов типа **REAL**, но в зависимости от числа входов модуля часть массива может быть заполнена нулями. Пользователю при этом возвращается столько элементов, сколько ему нужно – см. описание метода [prvParseResponse](#).

Строка ответа для модуля с 8 аналоговыми входами выглядит следующим образом:

```
>+499.98+33.758+49.998+33.880+50.000+49.998+34.601+34.65252$0D
```

Хотя для модулей ОВЕН жестко определено, что длина значения всегда составляет 7 символов (знак, 5 цифр и разделитель-точка) – в целях универсальности разбор ответа производится без этого предположения.

Метод включает в себя два цикла **FOR**.

В первом цикле в ответе определяются позиции символов-разделителей («+» и «-»).

Во втором цикле на основании этой информации происходит «вырезание» значений (значение расположено между соседними символами-разделителями, включая первый разделитель). При этом последнее значение обрабатывается особым образом (так как после него разделителя уже нет) – оно «вырезается» с помощью функции **MID** (из библиотеки [Standard](#)) между последним разделителем и контрольной суммой (если включена обработка контрольной суммы, т. е. вход **xUseCrc** имеет значение **TRUE**) или между последним разделителем и стоповым символом (если обработка контрольной суммы отключена). После этого значения конвертируются из строк (**STRING**) в числа с плавающей точкой (**REAL**).

```
// парсинг группового запроса аналоговых каналов
// максимально допустимое число каналов определяется константой блока c_usiMaxAiCount
METHOD prvConvertGroupAiToReal : ARRAY [1..c_usiMaxAiCount] OF REAL;
VAR
    // массив позиций разделителей
    m_aiSignPos:          ARRAY [1..c_usiMaxAiCount] OF INT;
    // длина значения в символах
    m_iLenOfValue:        INT;
    // счетчики циклов
    i, j:                 INT;
END_VAR
VAR CONSTANT
    // разделителями аналоговых значений являются их знаки
    c_m_sPlusSign:        STRING(1) := '+';
    c_m_sMinusSign:       STRING(1) := '-';
END_VAR

// пример ответа (с CRC):
>+499.98+33.758+49.998+33.880+50.000+49.998+34.601+34.65252$0D
```

```

// определяем позиции разделителей
j := 1;

FOR i := 0 TO LEN(sResponse) - 1 DO

    IF sResponse[i] = c_m_sPlusSign[0] OR sResponse[i] = c_m_sMinusSign[0] THEN
        // +1 нужен, так как байты строки нумеруются с 0, а символы - с 1
        m_aiSignPos[j] := i + 1;
        j := j + 1;
    END_IF

END_FOR

// вырезаем из ответа отдельные значения
// (j - 1) - потому что после цикла выше производится еще один дополнительный
инкремент j
// вычитая единицу, мы получаем найденное число разделителей (и, соответственно,
значений в ответе)
FOR i := 1 TO j - 1 DO

    // длина последнего значения в массиве определяется особым образом:
    // последнее значение размещено между последним разделителем и началом контрольной
суммы (если используется контрольная сумма) или стоповым символом
    IF i = (j - 1) THEN

        // +1 нужен, так как разделитель (знак) является частью значения
        IF xUseCrc THEN
            m_iLenOfValue := LEN(sResponse) - c_usiStopCharLen - c_usiCrcLen -
                m_aiSignPos[i] + 1;
        ELSE
            m_iLenOfValue := LEN(sResponse) - c_usiStopCharLen - m_aiSignPos[i] + 1;
        END_IF

    ELSE

        // а длину всех остальных значений можно определить
        // как разность позиций соседних разделителей
        m_iLenOfValue := m_aiSignPos[i + 1] - m_aiSignPos[i];

    END_IF

    // зная позицию начала значения и его длину - вырезаем его из ответа
    prvConvertGroupAiToReal[i] := TO_REAL(MID(sResponse, m_iLenOfValue,
        m_aiSignPos[i] ) );

END_FOR

```

6.4.8 Программа DCON_PRG_OCL

В программе **DCON_PRG_OCL** реализован опрос модуля MB110-8A(C) с помощью ФБ [COM_Control](#) из библиотеки [OwenCommunication](#) и [DCON_MASTER_OCL](#), основанном на ФБ [UNM_SerialRequest](#) из библиотеки [OwenCommunication](#).

В рамках примера производится отправка всех трех поддерживаемых блоком запросов. Обмен реализован с помощью единственного экземпляра ФБ [DCON_MASTER_OCL](#), который последовательно вызывается с различными аргументами через оператор **CASE**. В качестве переменной-селектора используется экземпляр перечисления [DCON_PARAMS](#).

В реальном проекте пользователю может потребоваться опрашивать несколько модулей с разными адресами. В этом случае ему требуется создать свое перечисление и добавить в оператор **CASE** нужное число шагов по аналогии с примером.

Задержка между запросами определяется константой **c_tPollDelay**. Переключение между шагами оператора **CASE** осуществляется с помощью метода **prvDelayAndSwitchToNextRequest**, в котором запускается таймер задержки.

```
PROGRAM DCON_PRG_OCL
VAR
  // ФБ открытия COM-порта
  fbComControl:      OCL.COM\_Control;
  // ФБ для опроса устройств по протоколу DCON
  fbDconMaster:      DCON\_MASTER\_OCL;
  // номер опрашиваемого канала для запроса OWEN_MV110_8A_READ_CHANNEL
  uiAiChannelNumber:  UINT := 2;
  // значение канала
  rAiChannelValue:    REAL;
  // массив значений всех каналов модуля
  arAiChannelsValues: ARRAY [1..8] OF REAL;
  // ответ на запрос USER_DEFINED (без какой-либо обработки)
  sUserDefinedResponse: STRING;

  // таймер задержки между запросами
  fbPollDelay:        TON;
  // текущий обрабатываемый запрос
  eDconParam:         DCON\_PARAMS := DCON_PARAMS.OWEN_MV110_xA_READ_ALL;
END_VAR
VAR CONSTANT
  // задержка между запросами
  c_tPollDelay:        TIME := T#1S;
END_VAR

// открытие COM-порта
fbComControl
(
  xEnable      := TRUE,
  udiComPort    := 1,
  udiBaudrate   := 115200,
  udiByteSize   := 8,
  eParity       := OCL.COM_PARITY.NONE,
  eStopBit      := OCL.COM_STOPBIT.ONE
);
```

```

// код опроса
IF fbComControl.xActive THEN

    CASE eDconParam OF

        // групповой опрос всех каналов
        DCON_PARAMS.OWEN_MV110_xA_READ_ALL:

            fbDconMaster
            (
                xExecute           := TRUE,
                // можно задать таймаут через константу (как уставку таймера задержки)
                tTimeout           := T#500MS,
                hCom               := fbComControl.hCom,
                eParam             := eDconParam,
                usiDeviceAddr      := 1,
                pResponseData      := ADR(arAiChannelsValues),
                szResponseData     := SIZEOF(arAiChannelsValues)
            );

            // задержка, сброс блока и переход к следующему запросу

            prvDelayAndSwitchToNextRequest(DCON_PARAMS.OWEN_MV110_xA_READ_CHANNEL);

            // опрос конкретного канала по его номеру
            DCON_PARAMS.OWEN_MV110_xA_READ_CHANNEL:

                fbDconMaster
                (
                    xExecute       := TRUE,
                    tTimeout       := T#500MS,
                    hCom           := fbComControl.hCom,
                    eParam         := eDconParam,
                    usiDeviceAddr  := 1,
                    // номер канала
                    pRequestData   := ADR(uiAiChannelNumber),
                    szRequestData  := SIZEOF(uiAiChannelNumber),
                    //
                    pResponseData  := ADR(rAiChannelValue),
                    szResponseData := SIZEOF(rAiChannelValue)
                );

                // задержка, сброс блока и переход к следующему запросу
                prvDelayAndSwitchToNextRequest(DCON_PARAMS.USER_DEFINED);

            // пользовательский запрос (в рамках примера - групповой опрос всех каналов)
            DCON_PARAMS.USER_DEFINED:

                fbDconMaster
                (
                    xExecute           := TRUE,
                    tTimeout           := T#500MS,
                    hCom               := fbComControl.hCom,
                    eParam             := eDconParam,
                    usiDeviceAddr      := 1,
                    // для используемой в примере команды нет никаких аргументов -
                    // но для других запросов они могут потребоваться
                    pRequestData       := ,
                    szRequestData      := ,
                    //
                    pResponseData      := ADR(sUserDefinedResponse),
                    szResponseData     := SIZEOF(sUserDefinedResponse),
                    // для пользовательского запроса необходимо указать стартовый символ команды
                    sUserDefinedStartChar := '#'
                );

```

```

        // задержка, сброс блока и переход к первому запросу
        prvDelayAndSwitchToNextRequest(DCON_PARAMS.OWEN_MV110_xA_READ_ALL);

    END_CASE

END_IF

```

```

// метод перехода к следующему запросу
METHOD prvDelayAndSwitchToNextRequest
VAR_INPUT
    // следующий запрос
    m_eDconParam: DCON\_PARAMS;
END_VAR

// запуск таймера задержки между запросами
fbPollDelay(IN := fbDconMaster.xDone OR fbDconMaster.xError, PT := c_tPollDelay);

IF fbPollDelay.Q THEN

    // сброс блока опроса для возможности его следующего вызова
    fbDconMaster
    (
        xExecute      := FALSE,
        // если этого не сделать, то на шаге USER_DEFINED (где это входы не
        // используются) -
        // они получают свои значения из предыдущего шага, что приведет к формированию
        // некорректной команды
        pRequestData  := CAA.Constants.gc_hINVALID,
        szRequestData := CAA.Constants.gc_szZERO
    );

    // сброс таймера для возможности его следующего вызова
    fbPollDelay(IN := FALSE);

    // переход к следующему запросу
    eDconParam := m_eDconParam;

END_IF

```

Device.Application.DCON_PRG_OCL		
Выражение	Тип	Значение
fbComControl	OCL.COM_Control	
fbDconMaster	DCON_MASTER_OCL	
uiAiChannelNumber	UINT	2
rAiChannelValue	REAL	32.195
arAiChannelsValues	ARRAY [1..8] OF REAL	
arAiChannelsValues[1]	REAL	499.97998
arAiChannelsValues[2]	REAL	32.195
arAiChannelsValues[3]	REAL	49.998
arAiChannelsValues[4]	REAL	32.3159981
arAiChannelsValues[5]	REAL	50
arAiChannelsValues[6]	REAL	49.998
arAiChannelsValues[7]	REAL	31.849
arAiChannelsValues[8]	REAL	31.9499989
sUserDefinedResponse	STRING	'>+499.98+32.195+49.998+32.316+50.000+49.998+31.849+31.9504E\$R'
fbPollDelay	TON	
eDconParam	DCON_PARAMS	OWEN_MV110_xA_READ_ALL
c_tPollDelay	TIME	T#1s

Рисунок 6.4.3 – Результаты работы программы

6.5 Реализация протокола DCON с помощью библиотеки CAA SerialCom

Пример, описанный в [п. 6.4](#), использует блоки из библиотеки [OwenCommunication](#). Эта библиотека может использоваться только в проектах, запускаемых на контроллерах OBEH и на виртуальном контроллере CODESYS Control Win V3. Но в ряде случаев требуется, чтобы проект был полностью платформо-независимым. Для этого пользователю необходимо создать аналоги блоков [COM_Control](#) и [UNM_SerialRequest](#) из библиотеки [OwenCommunication](#). Для работы с COM-портом в данном случае будет использоваться стандартная библиотека из дистрибутива CODESYS – [CAA SerialCom](#).

В рамках примера созданы следующие объекты:

- перечисление кодов ошибок [DCON_ERROR_CAA](#);
- функциональный блок [USR_COM_Control](#) (аналог [COM_Control](#) из библиотеки [OwenCommunication](#));
- функциональный блок [USR_SerialRequest](#) (аналог [UNM_SerialRequest](#) из библиотеки [OwenCommunication](#)) и перечисление шагов его машины состояний [COM_STATE](#).

6.5.1 Перечисление DCON_ERROR_CAA

Перечисление **DCON_ERROR_CAA** содержит ошибки, которые могут возникнуть при вызове ФБ [DCON_MASTER_CAA](#). Перечисление включает в себя коды ошибок из перечисления [ERROR](#) библиотеки [CAA SerialCom](#) и специфические коды ошибок, определенные в [п. 6.3](#). Числовой тип перечисления (INT) выбран в соответствии с числовым типом перечисления [COM.ERROR](#), чтобы упростить конвертацию одного перечисления в другое.

Таблица 6.5.1 – Описание элементов перечисления DCON_ERROR_CAA

Название	Значение	Описание
Общие ошибки		
NO_ERROR	0	Нет ошибок
TIME_OUT	5001	Ошибка таймаута
HANDLE_INVALID	5003	Некорректное значение дескриптора интерфейса
ERROR_UNKNOWN	5004	Неизвестная ошибка (зарезервировано для будущих версий)
WRONG_PARAMETER	5005	Как минимум один из аргументов ФБ имеет некорректное значение
WRITE_INCOMPLETE	5006	Отправка сообщения не была завершена
Специфические ошибки протокола DCON		
NOT_EXPECTED_START_CHAR	30	Стартовый символ в ответе не соответствует ожидаемому
INVALID_CRC	31	Несовпадение контрольной суммы ответа
INVALID_STOP_CHAR	32	Некорректный стоповый символ в ответе

```
{attribute 'qualified_only'}
{attribute 'strict'}
// Ошибки протокола DCON (при реализации с помощью библиотеки CAA SerialCom)
TYPE DCON_ERROR_CAA :
(
    // Стандартные коды ошибок для ФБ COM.Read и COM.Write
    NO_ERROR           := 0,
    TIME_OUT           := 5001,
    HANDLE_INVALID     := 5003,
    ERROR_UNKNOWN      := 5004,
    WRONG_PARAMETER    := 5005,
    WRITE_INCOMPLETE   := 5006,
    //
    // Специфические ошибки протокола DCON
    //
    // Стартовый символ в ответе не соответствует ожидаемому
    NOT_EXPECTED_START_CHAR := 30,
    // Несовпадение контрольной суммы ответа
    INVALID_CRC           := 31,
    // Некорректный стоповый символ в ответе
    INVALID_STOP_CHAR     := 32
) INT; // для гарантированного совпадения по размеру с COM.ERROR
END_TYPE
```

6.5.2 Перечисление COM_STATE

Перечисление **COM_STATE** описывает шаги машины состояний блока [USR_SerialRequest](#).

Таблица 6.5.2 – Описание элементов перечисления COM_STATE

Название	Значение	Описание
IDLE	0	Ожидание команды
WRITE	1	Запись в COM-порт
READ	2	Чтение из буфера COM-порта

```
{attribute 'qualified_only'}
{attribute 'strict'}
// Шаги машины состояний ФБ USR\_SerialRequest
TYPE COM_STATE :
(
    // Ожидание команды
    IDLE      := 0,
    // Запись в COM-порт
    WRITE     := 1,
    // Чтение из буфера COM-порта
    READ      := 2
);
END_TYPE
```

6.5.3 ФБ USR_COM_Control

Функциональный блок **USR_COM_Control** используется для открытия COM-порта с заданными настройками, а также его закрытия. Интерфейс и поведение блока полностью идентичны блоку [COM_Control](#) из библиотеки [OwenCommunication](#).



ПРИМЕЧАНИЕ

Одновременно (в пределах одного цикла контроллера) могут вызываться не более 20 экземпляров асинхронно выполняемых блоков (см. [п. 8.2](#)).

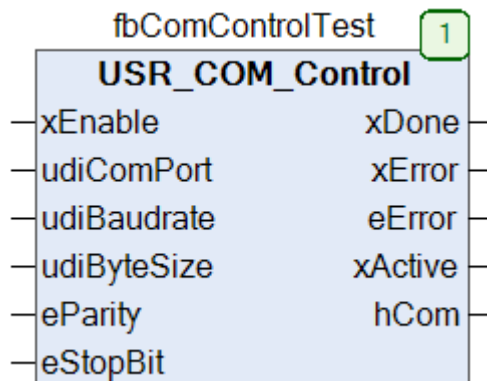


Рисунок 6.5.1 – Внешний вид ФБ USR_COM_Control на языке CFC

Таблица 6.5.3 – Описание входов и выходов ФБ USR_COM_Control

Название	Тип	Описание
Входы		
xEnable	BOOL	По переднему фронту происходит открытие COM-порта, по заднему – закрытие
udiComPort	UDINT	Номер COM-порта (см. п. 8.1)
udiBaudrate	UDINT	Скорость обмена в бодах. Стандартные возможные значения: 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200
udiByteSize	UDINT(7..8)	Число бит данных (7 или 8)
eParity	COM_PARITY	Режим контроля четности
eStopBit	COM_STOPBIT	Число стоп-бит
Выходы		
xDone	BOOL	Принимает TRUE на один цикл ПЛК при успешном открытии порта
xError	BOOL	Принимает значение TRUE в случае возникновения ошибки
eError	ERROR	Статус работы ФБ (или код ошибки)
xActive	BOOL	Пока порт открыт, данный выход имеет значение TRUE
hCom	CAA.HANDLE	Дескриптор COM-порта
Локальные переменные		
fbEnableTrig	R_TRIG	Детектор команды открытия COM-порта
fbDisableTrig	F_TRIG	Детектор команды закрытия COM-порта

fbComOpen	COM.Open	ФБ открытия COM-порта
fbComClose	COM.Close	ФБ закрытия COM-порта
astComParams	ARRAY [1..5] OF COM.PARAMETER	Массив параметров COM-порта

Блок работает следующим образом:

- по переднему фронту на входе **xEnable** осуществляется запись входных параметров (номера COM-порта, скорости обмена и т. д.) в массив структур **astComParams**, после чего происходит вызов экземпляра блока открытия COM-порта [COM.Open](#), на вход которому передается указатель на этот массив;
- по заднему фронту на входе **xEnable** происходит закрытие COM-порта с помощью экземпляра блока закрытия COM-порта [COM.Close](#) и обнуление дескриптора COM-порта на выходе блока.

```
// ФБ управления COM-портом
FUNCTION_BLOCK USR_COM_Control
VAR_INPUT
    // Пока вход имеет значение TRUE, блок находится в работе.
    // По заднему фронту происходит закрытие COM-порта
    xEnable:          BOOL;
    // Номер COM-порта (см. номера портов в узле Device - Информация)
    udiComPort:       UDINT;
    // Скорость обмена в бодах
    // (стандартные значения: 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200)
    udiBaudrate:      UDINT;
    // Количество бит данных
    udiByteSize:      UDINT;
    // Режим контроля четности
    eParity:           COM.PARITY;
    // Количество стоп-бит
    eStopBit:         COM.STOPBIT;
END_VAR
VAR_OUTPUT
    // Принимает TRUE на один цикл при успешном открытии порта
    xDone:             BOOL;
    // Флаг "Произошла ошибка"
    xError:            BOOL;
    // Код ошибки
    eError:            COM.ERROR;
    // Флаг "Порт успешно открыт"
    xActive:           BOOL;
    // Дескриптор открытого порта
    hCom:              COM.CAA_HANDLE;
END_VAR
VAR
    // Детектор команды открытия COM-порта
    fbEnableTrig:      R_TRIG;
    // Детектор команды закрытия COM-порта
    fbDisableTrig:     F_TRIG;
    // ФБ открытия COM-порта
    fbComOpen:         COM.Open;
    // ФБ закрытия COM-порта
    fbComClose:        COM.Close;

    // Структура настроек COM-порта
    astComParams:      ARRAY [1..5] OF COM.PARAMETER :=
    [
        (udiParameterId := COM.CAA_Parameter_Constants.udiPort),
        (udiParameterId := COM.CAA_Parameter_Constants.udiBaudrate),
```

```

        (udiParameterId := COM.CAA_Parameter_Constants.udiByteSize),
        (udiParameterId := COM.CAA_Parameter_Constants.udiParity),
        (udiParameterId := COM.CAA_Parameter_Constants.udiStopBits)
    ];
END_VAR

// детекторы команд
fbEnableTrig(CLK := xEnable);
fbDisableTrig(CLK := xEnable);

// по команде открытия COM-порта - формируем структуру настроек на основании входов
// блока
IF fbEnableTrig.Q THEN
    // порядок элементов должен соответствовать порядку в astComParams
    astComParams[1].udiValue := udiComPort;
    astComParams[2].udiValue := udiBaudrate;
    astComParams[3].udiValue := udiByteSize;
    astComParams[4].udiValue := TO_UDINT(eParity);
    astComParams[5].udiValue := TO_UDINT(eStopBit);
END_IF

// открываем COM-порт
fbComOpen
(
    xExecute      := fbEnableTrig.Q,
    // число элементов структуры определяется автоматически
    usiListLength := SIZEOF(astComParams) / SIZEOF(COM.PARAMETER),
    pParameterList := ADR(astComParams),
    xDone          => xDone,
    xError         => xError,
    eError         => eError,
);

IF fbComOpen.xDone THEN
    xActive := TRUE;
    hCom    := fbComOpen.hCom;
END_IF

// закрываем COM-порт по заднему фронту на входе xEnable
fbComClose
(
    xExecute := fbDisableTrig.Q,
    hCom     := hCom,
    xError   => xError,
    eError   => eError,
);

// обнуляем выходы блока после закрытия COM-порта
IF fbComClose.xDone THEN
    hCom := COM.CAA.Constants.gc_hINVALID;
    xActive := FALSE;
END_IF

```

6.5.4 ФБ USR_SerialRequest

Функциональный блок **USR_SerialRequest** используется для реализации нестандартного протокола при обмене через COM-порт. Блок является аналогом ФБ [UNM_SerialRequest](#) из библиотеки [OwenCommunication](#) со следующими отличиями:

- не реализована отправка повторных запросов в случае отсутствия ответа (вход **usiRetry** в ФБ [UNM_SerialRequest](#));
- не реализовано определение конца ответа по его размеру или стоповым символам (входы **szExpectedSize** и **wStopChar** в ФБ [UNM_SerialRequest](#)). Ответом считается вся информация, которая находится в буфере COM-порта после истечения времени **tTimeout**.



ПРИМЕЧАНИЕ

Одновременно (в пределах одного цикла контроллера) могут вызываться не более 20 экземпляров асинхронно выполняемых блоков (см. [п. 8.2](#)).



ПРИМЕЧАНИЕ

В случае отправки запросов, для которых не подразумевается получение ответа, рекомендуется для входа **tTimeout** установить значение **T#1ms**.

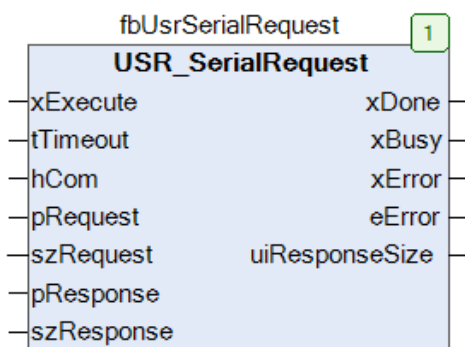


Рисунок 6.5.2 – Внешний вид ФБ UNM_SerialRequest на языке CFC

Таблица 6.5.4 – Описание входов и выходов ФБ UNM_SerialRequest

Название	Тип	Описание
Входы		
xExecute	BOOL	По переднему фронту происходит однократная отправка запроса
tTimeout	TIME	Таймаут ожидания ответа от slave-устройства (T#0ms – время ожидания не ограничено)
usiRetry	USINT	Число переповторов в случае отсутствия ответа
hCom	CAA.HANDLE	Дескриптор COM-порта, полученный от ФБ USR_COM_Control
pRequest	CAA.PVOID	Указатель на буфер запроса
szRequest	CAA.SIZE	Размер буфера запроса в байтах
pResponse	CAA.PVOID	Указатель на буфер ответа
szResponse	CAA.SIZE	Размер буфера ответа в байтах
Выходы		
xDone	BOOL	TRUE – получен корректный ответ от slave-устройства
xBusy	BOOL	TRUE – ФБ находится в работе
xError	BOOL	Принимает значение TRUE в случае возникновения ошибки
eError	ERROR	Статус работы ФБ (или код ошибки)
uiResponseSize	UINT	Размер полученного ответа в байтах
Локальные переменные		
fbExecute	R TRIG	Детектор запуска блока
fbComWrite	COM.Write	ФБ записи данных в COM-порт
fbComRead	COM.Read	ФБ чтения данных из буфера COM-порта
eState	COM_STATE	Текущий шаг машины состояний блока
fbTimeout	TON	Таймер таймаута
udiCurrentResponseSize	UDINT	Размер ответа для <u>текущего</u> вызова ФБ COM.Read
abyResponseBuffer	ARRAY [1..c_uiResponseBufferSize] OF BYTE	Буфер ответа
xNeedReset	BOOL	TRUE – нужно произвести переинициализацию блока
Константы		
c_uiResponseBufferSize	UINT	Размер буфера ответа в байтах

Машина состояний блока включает в себя 3 шага: IDLE, WRITE и READ. Ниже приведен листинг блока с комментариями к реализации.

```

// ФБ отправки в COM-порт произвольного набора байт и получения ответа
FUNCTION_BLOCK USR_SerialRequest
VAR_INPUT
    // По переднему фронту происходит однократная отправка запроса
    xExecute:          BOOL;
    // Таймаут ожидания ответа (0 - время не ограничено)
    tTimeout:          TIME;
    // Дескриптор COM-порта, полученный от ФБ USR\_COM\_CONTROL
    hCom:              COM.CAA.HANDLE;
    // Указатель на буфер данных запроса
    pRequest:          COM.CAA.PVOID;
    // Размер буфера данных запроса в байтах
    szRequest:         COM.CAA.SIZE;
    // Указатель на буфер данных ответа
    pResponse:         COM.CAA.PVOID;
    // Размер буфера данных ответа в байтах
    szResponse:        COM.CAA.SIZE;
END_VAR
VAR_OUTPUT
    // TRUE - получен корректный ответ
    xDone:             BOOL;
    // TRUE - блок находится в работе
    xBusy:             BOOL;
    // Флаг ошибки
    xError:            BOOL;
    // Код ошибки
    eError:            COM.ERROR;
    // Размер полученного ответа в байтах
    uiResponseSize:    UINT;
END_VAR
VAR
    // Детектор запуска блока
    fbExecute:         R_TRIG;
    // ФБ записи данных в COM-порт
    fbComWrite:        COM.Write;
    // ФБ чтения данных из буфера COM-порта
    fbComRead:         COM.Read;
    // Текущий шаг машины состояний блока
    eState:            COM.STATE;
    // Таймер таймаута
    fbTimeout:         TON;
    // Размер ответа для текущего вызова ФБ COM.Read
    udiCurrentResponseSize: UDINT;
    // Буфер ответа
    abyResponseBuffer: ARRAY [0..c_uiResponseBufferSize - 1] OF BYTE;
    // TRUE - нужно произвести переинициализацию блока
    xNeedReset:        BOOL;
END_VAR
VAR_CONSTANT
    // Размер буфера ответа в байтах
    c_uiResponseBufferSize:  UINT := 1024;
END_VAR

```

```

CASE eState OF

  // ожидание новой команды
  COM_STATE.IDLE:

    fbExecute(CLK := xExecute);

    IF fbExecute.Q THEN
      eState := COM_STATE.WRITE;
    END_IF

    // переинициализация блока перед новым вызовом
    IF NOT(xExecute) AND xNeedReset THEN

      uiResponseSize := 0;
      udiCurrentResponseSize := 0;

      // очищаем буфер ответа
      Mem.MemFill(pResponse, TO_UINT(szResponse), 0);

      fbComWrite(xExecute := FALSE);
      fbComRead(xExecute := FALSE);
      fbTimeout(IN := FALSE);

      xDone := FALSE;
      xError := FALSE;
      eError := COM.ERROR.NO_ERROR;

      xNeedReset := FALSE;

    END_IF

    // шаг записи данных в COM-порт
    COM_STATE.WRITE:

      xBusy := TRUE;

      fbComWrite
      (
        xExecute := TRUE,
        hCom := hCom,
        pBuffer := pRequest,
        szSize := szRequest,
        xError => xError,
        eError => eError
      );

      IF fbComWrite.xDone THEN
        fbComWrite(xExecute := FALSE);
        eState := COM_STATE.READ;
      ELSIF fbComWrite.xError THEN
        xBusy := FALSE;
        xNeedReset := TRUE;
        eState := COM_STATE.IDLE;
      END_IF

```

```

// шаг чтения данных из буфера COM-порта
COM_STATE.READ:

    fbTimeout(IN := TRUE, PT := tTimeout);

    fbComRead
    (
        xExecute := TRUE,
        hCom      := hCom,
        // используется промежуточный буфер, так как ответ
        // может прийти в виде нескольких отдельных фрагментов
        // и надо склеить их в один, перед тем, как отдать пользователю
        pBuffer   := ADR(abyResponseBuffer),
        szBuffer  := SIZEOF(abyResponseBuffer),
        xError    => xError,
        eError    => eError
    );

    IF fbComRead.xError THEN
        xBusy := FALSE;
        xNeedReset := TRUE;
        eState := COM_STATE.IDLE;
    END_IF

    // если при чтении буфер окажется пустым, то xDone всё равно примет значение TRUE
    IF fbComRead.xDone THEN

        // поэтому проверяем, что в буфере что-то есть
        IF fbComRead.szSize <> 0 THEN

            // и приклеиваем к предыдущему фрагменту ответа, если в буфере еще есть место
            IF (udiCurrentResponseSize + fbComRead.szSize) <= szResponse THEN
                Mem.MemMove(ADR(abyResponseBuffer), pResponse + udiCurrentResponseSize,
                    TO_UINT(fbComRead.szSize));
                udiCurrentResponseSize := udiCurrentResponseSize + fbComRead.szSize;
            END_IF

        END_IF

        // сбрасываем блок для возможности его последующего вызова
        fbComRead(xExecute := FALSE);

    END_IF

    // сработал таймер таймаута
    IF fbTimeout.Q THEN

        // если в буфере что-то есть - информируем об этом пользователя
        IF udiCurrentResponseSize <> 0 THEN
            xDone := TRUE;
            uiResponseSize := TO_UINT(udiCurrentResponseSize);
        // а если нет - генерируем ошибку
        ELSE
            xError := TRUE;
            eError := COM.ERROR.TIME_OUT;
        END_IF

        xBusy := FALSE;
        xNeedReset := TRUE;
        eState := COM_STATE.IDLE;

    END_IF

END_CASE

```

На шаге **IDLE** происходит ожидание новой команды (переднего фронта на входе **xExecute**). После получения команды происходит переход на шаг **WRITE**. Кроме того, выполняется переинициализация блока – сброс выходов и некоторых локальных переменных, а также очистка буфера ответа. Переинициализация производится после завершения работы блока (в этот момент переменной **xNeedReset** присваивается значение **TRUE**) и только в том случае, если к этому моменту вход **xExecute** уже имеет значение **FALSE** (это необходимо для соответствия [модели поведения](#)).

На шаге **WRITE** выполняется отправка содержимого буфера запроса (вход блока **pRequest**) в COM-порт с помощью экземпляра ФБ [COM.Write](#). Если отправка прошла успешно (выход **xDone** имеет значение **TRUE**) – то происходит переход на шаг **READ**, в случае возникновения ошибки (выход **xError** имеет значение **TRUE**) – возвращение на шаг **IDLE** (так как если запрос не был отправлен, нет смысла пытаться вычитать данные из буфера COM-порта).

На шаге **READ** выполняется чтение из буфера COM-порта с помощью экземпляра ФБ [COM.Read](#). Однократного чтения недостаточно – потому что задержка между отправкой запроса и получением ответа часто является непредсказуемой величиной и, кроме того, ответ может прийти не одним пакетом, а в виде последовательности фрагментов. Поэтому чтение из буфера COM-порта выполняется на протяжении всего времени таймута (**tTimeout**). При получении новой порции данных – они добавляются в конец буфера ответа. После истечения таймута возможны два варианта:

- в буфере есть какие-то данные – в этом случае выход **xDone** блока **USR_SerialRequest** принимает значение **TRUE** – это является для пользователя сигналом о том, что можно скопировать содержимое буфера и приступить к его обработке;
- в буфере нет данных – в этом случае генерируется ошибка **TIME_OUT**.

Независимо от варианта происходит возвращение на шаг **IDLE** для ожидания следующей команды.

6.5.5 ФБ DCON_MASTER_CAA

Функциональный блок **DCON_MASTER_CAA** отличается от блока [DCON_MASTER_OCL](#) лишь следующим:

- выход **eError** имеет тип [DCON_ERROR_CAA](#) вместо [DCON_ERROR_OCL](#) (потому что коды ошибок блоков [COM.Read](#) / [COM.Write](#) и [UNM_SerialRequest](#) отличаются);
- в коде ФБ **DCON_MASTER_CAA** и его метода **prvParseResponse** при обращении к выходу **eError** должно использоваться перечисление [DCON_ERROR_CAA](#) вместо [DCON_ERROR_OCL](#);
- переменная **fbUnmSerialRequest** имеет тип [USR_SerialRequest](#) вместо [UNM_SerialRequest](#).

Других отличий нет.

6.5.6 Программа DCON_PRG_CAA

Программа **DCON_PRG_CAA** отличается от программы [DCON_PRG_OCL](#) лишь следующим:

- переменная **fbComControl** имеет тип [USR_COM_Control](#) вместо [OCL.COM_Control](#);
- в вызове **fbComControl** используются перечисления [COM.PARITY](#) и [COM.STOPBIT](#) вместо [OCL.COM_PARITY](#) и [OCL.COM_STOPBIT](#);
- переменная **fbDconMaster** имеет тип [DCON_MASTER_CAA](#) вместо [DCON_MASTER_OCL](#).

Других отличий нет.

7 Пример реализации протокола Пульсар-М

7.1 Общая информация

Протокол «Пульсар-М» используется в счетчиках и других приборах линейки «Пульсар», разработанной [ООО НПП «ТЕПЛОВОДОХРАН»](#). Спецификация протокола доступна [по ссылке](#) (см. также [старый вариант спецификации](#) – в нем присутствуют примеры конкретных пакетов). В отличие от строкового протокола [DCON](#) – протокол Пульсар-М является бинарным (данные передаются в двоичном виде).

В рамках примера будут поддерживаны команды чтения текущих значений измерительных каналов и системного времени прибора.

Ссылка на готовый пример: [Example_UserProtocols.projectarchive](#)

Пример создан в среде **CODESYS 3.5 SP17 Patch 3** и подразумевает запуск на **СПК1xx [M01]** с таргет-файлом **3.5.17.31**. В случае необходимости запуска проекта на другом устройстве следует изменить таргет-файл в проекте (ПКМ на узел **Device** – **Обновить устройство**).

7.2 Описание протокола

Структура пакета протокола Пульсар-М выглядит следующим образом (*запросы и ответы имеют идентичную структуру*):

Адрес	Функция (код команды)	Длина пакета	Данные	Идентификатор пакета	Контрольная сумма
4 байта	1 байт	1 байт	до 245 байт	2 байта	2 байта

Адрес – адрес конкретного прибора в [BCD](#)-формате (т. е. для прибора с адресом 12345678 в запросе должен быть указан адрес 0x12345678), который совпадает с его серийным номером. В запросе может быть указан широковещательный адрес 0x00000000 – прибор ответит на этот запрос, при этом поле «адрес» в ответе будет иметь значение «реального» адрес прибора. Поскольку на запрос по широковещательному адресу отвечают все приборы на шине, то в случае интерфейса RS-485 использование широковещательного адреса возможно только в том случае, если к шине подключен один прибор.

Функция – используемая команда (см. список команд в спецификации протокола).

Длина пакета – общий размер пакета в байтах. Диапазон возможных значений: 10...255 байт.

Данные – данные, которые передаются в запросе или ответе. Формат данных и их размер зависят от используемой функции.

Идентификатор пакета – псевдослучайное число, которое генерируется master-устройством. Ответ slave-устройства должен включать такой же идентификатор.

Контрольная сумма – контрольная сумма пакета, рассчитанная по алгоритму CRC-16/MODBUS. В отличие от протокола Modbus – в протоколе Пульсар-М контрольная сумма передается старшим байтом вперед.

Пример запроса на считывание текущих значений каналов счетчика импульсов-регистратора Пульсар (функция 0x01)

Запрос: 0x12 0x34 0x56 0x78 0x01 0x0E 0x02 0x00 0x00 0x00 0x5E 0xA4 0x41 0x63

Расшифровка:

- адрес прибора – 12345678 (передается в BCD-формате);
- используемая функция – 0x01 (чтение данных по каналам);
- длина запроса – 14 байт (0x0E);
- данные запроса – битовая маска считываемых каналов, младшим байтом вперед. В запросе считывается канал с номером 2 (потому что 0x02 = 2#00000010). Для чтения первого и второго каналов нужно было бы указать битовую маску 0x03 00 00 00, так как 0x03 = 2#00000011);
- идентификатор запроса – 0x5EA4;
- контрольная сумма – 0x4163.

Ответ: 0x12 0x34 0x56 0x78 0x01 0x12 0x00 0x00 0x40 0x70 0x3D 0x0A 0x01 0x40 0x5E 0xA4 0x82 0x37

Расшифровка:

- адрес прибора – 12345678 (передается в BCD-формате);
- используемая функция – 0x01 (чтение данных по каналам);
- длина запроса – 18 байт (0x12);
- данные ответа – для счетчиков импульсов-регистраторов Пульсар значения передаются в виде чисел двойной точности (в спецификации протокола и некоторых языках программирования этот тип данных называется **Double**, в CODESYS V3.5 ему соответствует тип **LREAL**). Данные возвращаются младшим байтом вперед. В данном случае значение канала приближенно равно 2.13 (см. рисунок ниже);
- идентификатор запроса – 0x5EA4;
- контрольная сумма – 0x8207.

IEEE-754 Floating-Point Conversion
From 64-bit Hexadecimal Representation
To Decimal Floating-Point
Along with the Equivalent 32-bit Hexadecimal and Binary Patterns

Enter the 64-bit hexadecimal representation of a floating-point number here, then click either the **Rounded** or the **Not Rounded** button.

Hexadecimal Representation:

Rounding from 64-bit to 32-bit representation uses the IEEE-754 round-to-nearest-value mode.

Results:

Decimal Value Entered:

Рисунок 7.2.1 – Онлайн-конвертер HEX-представления типа Double в числовое (обратите внимание – в конвертере данные нужно вводить старшим байтом вперед)

При получении ответа на запрос master должен валидировать его следующим образом:

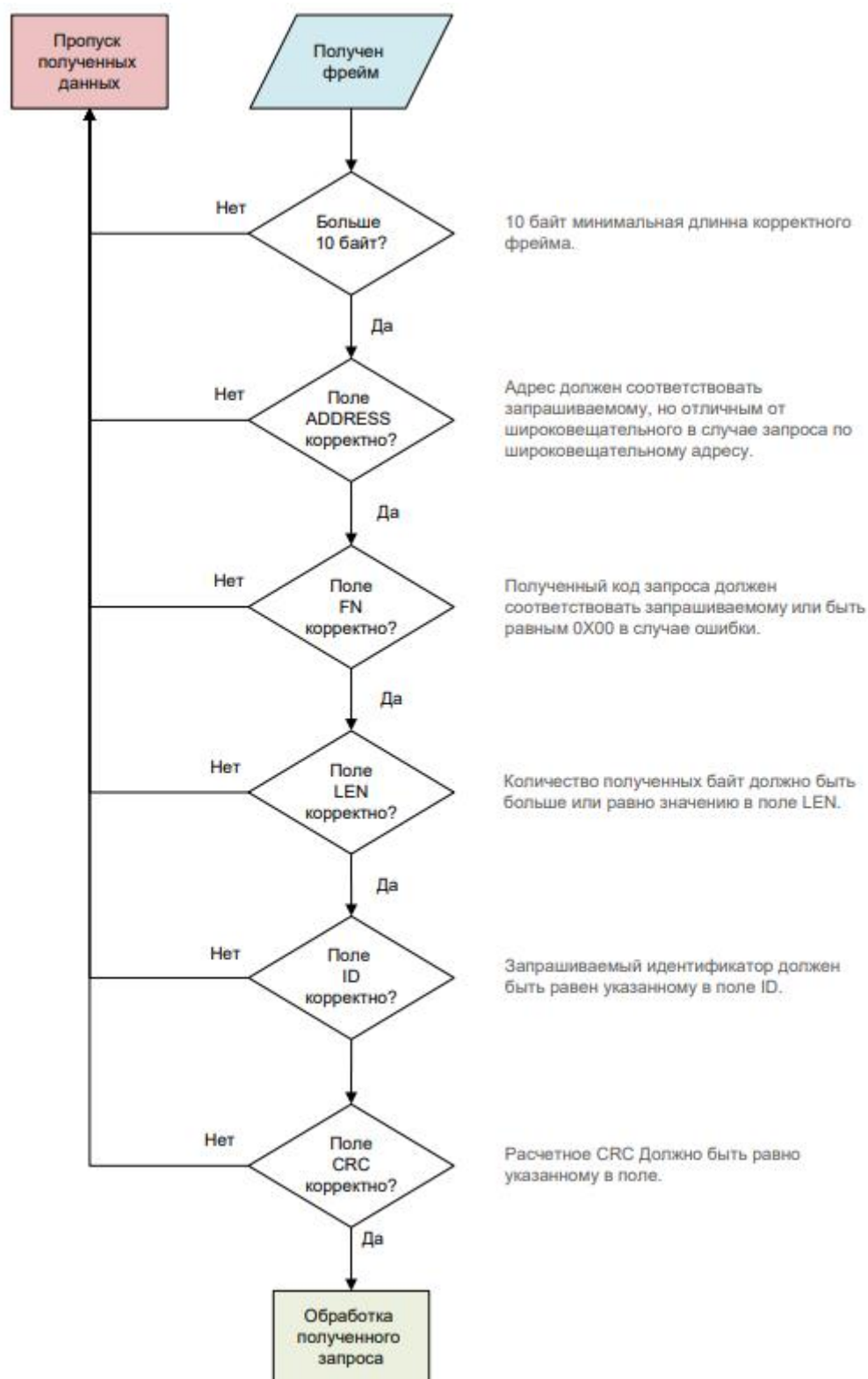


Рисунок 7.2.2 – Алгоритм проверки ответа на корректность

Таким образом, при проверке ответа могут быть обнаружены следующие ошибки:

- некорректная длина пакета (менее 10 байт или реальная длина ответа не совпадает со значением поля длины ответа);
- некорректный адрес в ответе (не совпадает с адресом в запросе при условии, что адрес в запросе – не широковещательный);
- некорректный код команды в ответе – не совпадает с кодом команды в запросе и при этом не равен 0. Код команды 0 в ответе означает, что slave-устройство обнаружило ошибку в запросе master-устройства. Код ошибки размещен в поле данных и занимает 1 байт. Список кодов ошибок приведен ниже;
- некорректный идентификатор пакета в ответе (не совпадает с идентификатором в запросе);
- некорректная контрольная сумма в ответе (CRC в ответе не совпадает с рассчитанной CRC для данного ответа).

Список кодов ошибок slave-устройства (для кода команды = 0):

- 0x01 – отсутствует запрашиваемый код функции;
- 0x02 – ошибка в битовой маске запроса;
- 0x03 – ошибочная длина запроса;
- 0x04 – отсутствует параметр;
- 0x05 – запись заблокирована, требуется авторизация;
- 0x06 – записываемое значение (параметр) находится вне заданного диапазона;
- 0x07 – отсутствует запрашиваемый тип архива;
- 0x08 – превышение максимального количества запрашиваемых архивных значений за один пакет.

7.3 Формулировка задачи

После ознакомления с [п. 7.1](#) и [п. 7.2](#) можно сформулировать требования к функциональному блоку, который будет реализовывать обмен по протоколу Пульсар-М:

- блок должен поддерживать команды протокола Пульсар-М – в рамках примера будет реализована поддержка команды считывания значений каналов (код 0x01) и чтения системного времени (код 0x04);
- блок должен позволять разработчику с минимумом усилий добавлять поддержку новых команд;
- блок должен позволять отправить произвольный запрос – это полезно в тех случаях, когда блок будет распространяться в виде скомпилированной библиотеке и у конечного пользователя не будет возможности его доработать;
- блок должен обеспечивать обработку ошибок в соответствии с информацией из [п. 7.2](#).

Команды и коды ошибок протокола удобно представить в виде перечислений.

7.4 Реализация протокола Пульсар-М с помощью библиотеки OwenCommunication

В следующих пунктах описана реализация протокола Пульсар-М с помощью функциональных блоков из библиотеки [OwenCommunication](#). Пример в максимально возможной степени соответствует [примеру реализации протокола DCON](#). Если вы ознакомитесь с [п. 6.4](#), то увидите, что отличия между двумя примерами касаются только особенностей конкретного протокола.

В рамках реализации созданы следующие объекты:

- перечисление команд протокола [PULSAR_PARAMS](#);
- перечисление кодов ошибок [PULSAR_ERROR_OCL](#);
- функциональный блок [PULSAR_MASTER_OCL](#), реализующий обмен по протоколу Пульсар-М и включающий в себя ряд методов.

Перечисления позволяют задать соответствие между числовыми значениями и символьными именами – это повышает читабельность кода. Например, вместо того чтобы помнить, что код ошибки 12 означает «некорректные входные параметры ФБ» вы можете использовать в коде его символьное имя, объявленное в перечислении – **PULSAR_OCL_ERROR.WRONG_PARAMETER**.

Для создания перечисления следует нажать **ПКМ** на узел **Application**, использовать команду **Добавление объекта – DUT** и указать тип объекта **Перечисления**.

Методы позволяют упростить код функционального блока или программы, разделив его на отдельные фрагменты. Кроме того, методы могут иметь входные, выходные и локальные переменные – что позволяет их использовать в качестве «встраиваемых» функций.

Для создания метода следует нажать **ПКМ** на объект, к которому должен быть добавлен метод, и использовать команду **Добавление объекта – Метод**.

Папки используются для структуризации проекта.

Для создания папки следует нажать **ПКМ** на свободное пространство в дереве проекта и использовать команду **Добавить папку**.

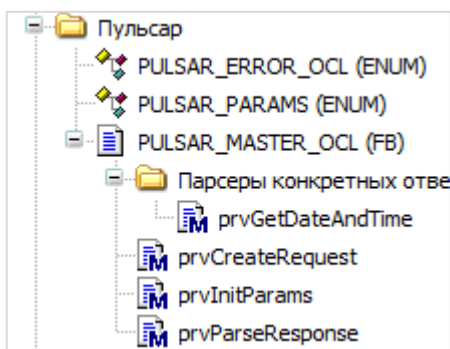


Рисунок 7.4.1 – Объекты примера

7.4.1 Перечисление PULSAR_ERROR_OCL

Перечисление **PULSAR_ERROR_OCL** содержит ошибки, которые могут возникнуть при вызове ФБ **PULSAR_MASTER_OCL**. Перечисление включает в себя коды ошибок из перечисления **ERROR** библиотеки **OwenCommunication** (только те, которые могут возникнуть при обмене по интерфейсам RS-485/RS-232) и специфические коды ошибок, определенных в п. 7.2. Числовой тип перечисления (INT) выбран в соответствии с числовым типом перечисления **OCL.ERROR**, чтобы упростить конвертацию одного перечисления в другое.

Обратите внимание, что выбранные значения для специфических ошибок (INVALID_FUNCTION и т. д.) отличаются от кодов ошибок из спецификации протокола на +100 (см. п. 7.2). Использовать значения из спецификации не представляется возможным, так как они уже заняты кодами «общих» ошибок. Это смещение учтено в методе **prvParseResponse** блока **PULSAR_MASTER_OCL** с помощью константы **mc_byPulsarErrorCodeEnumOffset**.

Таблица 7.4.1 – Описание элементов перечисления PULSAR_ERROR_OCL

Название	Значение	Описание
Общие ошибки		
NO_ERROR	0	Нет ошибок
TIME_OUT	1	Ошибка таймаута
HANDLE_INVALID	10	Некорректное значение дескриптора интерфейса
ERROR_UNKNOWN	11	Неизвестная ошибка (зарезервировано для будущих версий)
WRONG_PARAMETER	12	Как минимум один из аргументов ФБ имеет некорректное значение
WRITE_INCOMPLETE	13	Отправка сообщения не была завершена
INVALID_DATAPOINTER	20	Некорректный указатель на буфер данных
INVALID_DATASIZE	21	Некорректный размер буфера данных
Специфические ошибки протокола Пульсар-М		
INVALID_FUNCTION	101	Отсутствует запрашиваемый код функции
INVALID_MASK	102	Ошибка в битовой маске запроса
INVALID_REQUEST_LEN	103	Ошибочная длина запроса
INVALID_PARAM	104	Отсутствует параметр
AUTHORIZATION_REQUIRED	105	Запись заблокирована, требуется авторизация
PARAM_OUT_OF_RANGE	106	Записываемое значение (параметр) находится вне заданного диапазона
NO_ARCHIVE_TYPE	107	Отсутствует запрашиваемый тип архива
TOO_MUCH_ARCHIVE_DATA	108	Превышение максимального количества считываемых архивных значений за один пакет
INVALID_RESPONSE_LEN	120	Некорректная длина ответа (<10 байт или значение поля LEN не соответствует фактической длине ответа)
INVALID_ADDR	121	Некорректный адрес устройства в ответе
INVALID_TRANSACTION_ID	122	Некорректный идентификатор пакета в ответе
INVALID_CRC	123	Некорректная CRC в ответе

```

{attribute 'qualified_only'}
{attribute 'strict'}
// Ошибки протокола Пульсар (при реализации с помощью библиотеки OwenCommunication)
TYPE PULSAR_ERROR_OCL :
(
    // Стандартные коды ошибок для ФБ COM_Control и UNM_SerialRequest из OCL.ERROR
    NO_ERROR := 0,
    TIME_OUT := 1,
    HANDLE_INVALID := 10,
    ERROR_UNKNOWN := 11,
    WRONG_PARAMETER := 12,
    WRITE_INCOMPLETE := 13,
    INVALID_DATAPOINTER := 20,
    INVALID_DATASIZE := 21,
    //
    // Специфические ошибки протокола Пульсар-M
    //
    // Отсутствует запрашиваемый код функции
    INVALID_FUNCTION := 101,
    // Ошибка в битовой маске запроса
    INVALID_MASK := 102,
    // Ошибочная длина запроса
    INVALID_REQUEST_LEN := 103,
    // Отсутствует параметр
    INVALID_PARAM := 104,
    // Запись заблокирована, требуется авторизация
    AUTHORIZATION_REQUIRED := 105,
    // Записываемое значение (параметр) находится вне заданного диапазона
    PARAM_OUT_OF_RANGE := 106,
    // Отсутствует запрашиваемый тип архива
    NO_ARCHIVE_TYPE := 107,
    // Превышение максимального количества считываемых архивных значений за один пакет
    TOO_MUCH_ARCHIVE_DATA := 108,
    // Некорректная длина ответа
    // (<10 байт или значение поля LEN не соответствует фактической длине ответа)
    INVALID_RESPONSE_LEN := 120,
    // Некорректный адрес устройства в ответе
    INVALID_ADDR := 121,
    // Некорректный идентификатор пакета в ответе
    INVALID_TRANSACTION_ID := 122,
    // Некорректная CRC в ответе
    INVALID_CRC := 123
) INT; // для гарантированного совпадения по размеру с OCL.ERROR
END_TYPE

```

7.4.2 Перечисление PULSAR_PARAMS

Перечисление **PULSAR_PARAMS** содержит коды команд протокола Пульсар-М, поддерживаемых в рамках примера. В зависимости от конкретной команды под указателями входов **pRequestData** и **pResponseData** ФБ [PULSAR_MASTER_OCL](#) размещаются переменные различного типа.

Таблица 7.4.2 – Описание элементов перечисления PULSAR_PARAMS

Название	Значение ⁹	Описание	Тип переменной по указателю pRequestData	Тип переменной по указателю pResponseData
USER_DEFINED	0	Произвольный запрос. Параметры запроса задаются пользователем при вызове ФБ	см. спецификацию протокола	см. спецификацию протокола)
GET_CHANNELS_DATA	1	Чтение значений каналов прибора	DWORD	ARRAY [1..x] OF INT/UINT/REAL/LREAL
GET_RTC	4	Чтение системного времени прибора	-	DT

Для запроса **USER_DEFINED** данные запроса передаются по указателю **pRequestData** (описание данных для конкретной команды приведено в спецификации протокола). Поле данных ответа размещается по указателю **pResponseData** без какой-либо обработки.

Для запроса **GET_CHANNELS_DATA** в данных запроса передается битовая маска считываемых каналов (в виде переменной типа **DWORD**). Ответ предоставляется без какой-либо обработки – в зависимости от конкретного прибора это может быть массив типа **INT/UINT/REAL/LREAL**. Число элементов массива определяется заданной пользователем битовой маской.

Для запроса **GET_RTC** поле данных отсутствует. Ответ предоставляется в виде переменной типа **DT** после «сборки» из отдельных разрядов времени с помощью метода [prvGetDateAndTime](#).

```
{attribute 'qualified_only'}
{attribute 'strict'}
// Команды протокола Пульсар-М
TYPE PULSAR_PARAMS :
(
  // Пользовательский запрос.
  // pRequestData - см. спецификацию протокола,
  // pResponseData - см. спецификацию протокола
  USER_DEFINED := 0,
  // Чтение данных каналов счетчика
  // pRequestData - указатель на DWORD (битовая маска считываемых каналов)
  // pResponseData - указатель на ARRAY [1..x] OF REAL/LREAL/INT/UINT
  // (см. спецификацию протокола для конкретного прибора)
  GET_CHANNELS_DATA := 1,
  // Чтение системного времени
  // pRequestData - не используется, pResponseData - указатель на DT
  GET_RTC := 4
) BYTE;
END_TYPE
```

⁹ Значения элементов перечисления соответствуют кодам команд протокола.

7.4.3 ФБ PULSAR_MASTER_OCL

Функциональный блок **PULSAR_MASTER_OCL** используется для опроса устройств по протоколу Пульсар-М.

Описание работы блока:

По переднему фронту на входе **xExecute** происходит отправка запроса устройству с адресом **udiDeviceAddr** через COM-порт, определяемый дескриптором **hCom**, полученным от ФБ [COM_Control](#). Выбор команды осуществляется с помощью входа **eParam**, представляющего собой перечисление типа [PULSAR_PARAMS](#). Для отправки произвольного запроса вход **eParam** должен иметь значение **PULSAR_PARAMS.USER_DEFINED**; код команды в этом случае определяется значением входа **byUserDefinedFunction**.

Если в запросе должны быть переданы записываемые данные – они размещаются по указателю **pRequestData**, а вход **szRequestData** определяет размер буфера записываемых данных в байтах. Описание параметров конкретного запроса можно найти в спецификации конкретного устройства.

В случае получения корректного ответа выход **xDone** принимает значение **TRUE**. Полученные данные помещаются в буфер, расположенный по указателю **pResponseData** и имеющий размер **szResponseData** байт. В описании перечисления [PULSAR_PARAMS](#) приведены типы данных переменных, которые должны быть размещены под указателем для конкретного запроса.

Если ответ на запрос не был получен за время **tTimeout**, то выход **xError** принимает значение **TRUE**, а выход **eError** = **TIME_OUT**. В случае получения некорректного ответа выход **xError** принимает значение **TRUE**, а на выходе **eError** отображается код ошибки из перечисления [PULSAR_ERROR_OCL](#).

Для отправки нового запроса следует создать передний фронт на входе **xExecute**.

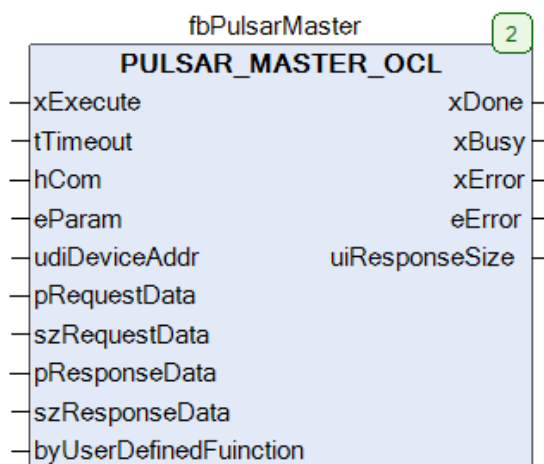


Рисунок 7.4.2 – Внешний вид ФБ PULSAR_MASTER_OCL на языке CFC

Ниже приведено описание входов/выходов/локальных переменных/констант блока и его листинг с комментариями к реализации.

Таблица 7.4.3 – Описание переменных ФБ PULSAR_MASTER_OCL

Название	Тип	Описание
Входы		
xExecute	BOOL	По переднему фронту происходит однократная отправка запроса
tTimeout	TIME	Таймаут ожидания ответа от slave-устройства (T#0ms – время ожидания не ограничено)
hCom	CAA.HANDLE	Дескриптор COM-порта, полученный от ФБ COM_Control
eParam	PULSAR_PARAMS	Команда протокола Пульсар-М, используемая в запросе
udiDeviceAddr	UDINT	Адрес устройства (совпадает с серийным номером)
pRequestData	CAA.PVOID	Указатель на буфер запроса
szRequestData	CAA.SIZE	Размер буфера запроса в байтах
pResponseData	CAA.PVOID	Указатель на буфер ответа
szResponseData	CAA.SIZE	Размер буфера ответа в байтах
byUserDefinedFunction	BYTE	Код команды для произвольного запроса (PULSAR_PARAMS.USER_DEFINED)
Выходы		
xDone	BOOL	TRUE – получен корректный ответ
xBusy	BOOL	TRUE – ФБ находится в работе
xError	BOOL	Принимает значение TRUE в случае возникновения ошибки
eError	PULSAR_ERROR_OCL	Статус работы ФБ (или код ошибки)
uiResponseSize	UINT	Размер полученного ответа в байтах
Локальные переменные		
fbUnmSerialRequest	OCL.UNM_SerialRequest	Коммуникационный ФБ из библиотеки OwenCommunication . Реализует отставку произвольного набора байт и получения ответа
fbExecute	R_TRIG	Детектор запуска блока
xNeedReset	BOOL	TRUE – нужно произвести переинициализацию блока
xlslnitParams	BOOL	TRUE – инициализация параметров проведена успешно, FALSE - ошибки инициализации параметров
xWrongParameter	BOOL	Флаг ошибки инициализации параметров, удерживается до заднего фронта на xExecute . Используется для запрета вызова fbUnmSerialRequest при некорректных параметрах запроса
abyRequest	ARRAY [0..255] ¹⁰ OF BYTE	Буфер запроса

¹⁰ Согласно спецификации протокола (см. п. 7.2) – максимальный размер пакета составляет 255 байт. В рамках примера для буфера выделяется массив размером [0..255], который занимает 256 байт. Это решение основано не на практических соображениях – просто в большинстве случаев удобно определять буферы с четным размером (особенно если он является степенью двойки) – например, это упрощает отображение буферов на структуры с выравниваем памяти и т. д.

xlValidPointerForRequest	BOOL	TRUE – выделен буфер запроса (входы pRequestData и szRequestData имеют корректные значения)
xlValidPointerForResponse	BOOL	TRUE – выделен буфер ответа (входы pResponseData и szResponseData имеют корректные значения)
dwRequestBcdAddr	DWORD	Адрес устройства в запросе в BCD -формате
dwResponseBcdAddr	DWORD	Адрес устройства в ответе BCD -формате
eResponseParam	PULSAR_PARAMS	Код команды в ответе
usiRequestLen	USINT	Длина запроса в байтах
usiResponseLen	USINT	Длина ответа в байтах
uiRequestTransactionId	UINT	Идентификатор пакета в запросе
uiResponseTransactionId	UINT	Идентификатор пакета в ответе
wRequestCrc	WORD	CRC в запросе
wResponseCrc	WORD	CRC в ответе
abyResponse	ARRAY [0..255] OF BYTE	Буфер ответа

Таблица 7.4.4 – Описание констант ФБ PULSAR_MASTER_OCL

Название	Тип	Значение	Описание
c_usiAddrSize	USINT	4	Размер адреса в байтах
c_usiFunctionSize	USINT	1	Размер кода команды в байтах
c_usiLenSize	USINT	1	Размер длины пакета в байтах
c_usiTransactionIdSize	USINT	2	Размер идентификатора пакета в байтах
c_usiCrcSize	USINT	2	Размер CRC в байтах
c_usiAddrBufferOffset	USINT	0	Позиция в буфере, начиная с которой размещается адрес
c_usiFunctionBufferOffset	USINT	4	Позиция в буфере, начиная с которой размещается код команды
c_usiLenBufferOffset	USINT	5	Позиция в буфере, начиная с которой размещается длина пакета
c_usiDataBufferOffset	USINT	6	Позиция в буфере, начиная с которой размещаются данные
c_usiMinValidResponseLen	USINT	10	Минимально возможная длина корректного ответа
c_byErrorFunctionCode	BYTE	0	Код команды в ответе в случае присутствия в ответе кода ошибки

Константы позволяют избежать в коде использования «[магических чисел](#)», заменив их более понятными символьными именами. Константы с постфиксом **_Size** определяют размеры полей запроса/ответа (адрес, код команды и т. д.). Константы с постфиксом **_Offset** определяют позиции в буферах запроса/ответа для ряда полей (адреса, кода команды и т. д.). Константа **c_usiMinValidResponseLen** определяет минимально возможную длину корректного ответа – согласно спецификации протокола она составляет 10 байт (см. [п. 7.2](#)). Константа **c_byErrorFunctionCode** определяет код команды, который возвращает в ответе slave-устройство в том случае, если ответ содержит код ошибки.

```

// ФБ для опроса устройств по протоколу Пульсар-M
FUNCTION_BLOCK PULSAR_MASTER_OCL
VAR_INPUT
    // По переднему фронту происходит однократная отправка запроса
    xExecute:          BOOL;
    // Таймаут ожидания ответа (0 - время не ограничено)
    tTimeout:          TIME;
    // Дескриптор COM-порта, полученный от ФБ OCL.COM\_CONTROL
    hCom:              CAA.HANDLE;
    // Команда протокола Пульсар-M
    eParam:            PULSAR\_PARAMS;
    // Адрес устройства
    udiDeviceAddr:     UDINT;
    // Указатель на буфер данных запроса
    // (см. описание нужных типов данных в PULSAR_PARAMS)
    pRequestData:      CAA.PVOID;
    // Размер буфера данных запроса в байтах
    szRequestData:     CAA.SIZE;
    // Указатель на буфер данных ответа
    // (см. описание нужных типов данных в PULSAR_PARAMS)
    pResponseData:     CAA.PVOID;
    // Размер буфера данных ответа в байтах
    szResponseData:    CAA.SIZE;
    // Код команды пользовательского запроса (USER_DEFINED)
    byUserDefinedFunction: BYTE;
END_VAR
VAR_OUTPUT
    // TRUE - получен корректный ответ
    xDone:             BOOL;
    // TRUE - блок находится в работе
    xBusy:             BOOL;
    // Флаг ошибки
    xError:            BOOL;
    // Код ошибки
    eError:            PULSAR\_ERROR\_OCL;
    // Размер ответа в байтах
    uiResponseSize:    UINT;
END_VAR
VAR
    // Коммуникационный ФБ из библиотеки OwenCommunication.
    // Реализует отправку произвольного набора байт и получения ответа
    fbUnmSerialRequest: OCL.UNM\_SerialRequest;
    // Детектор запуска блока
    fbExecute:          R_TRIG;
    // TRUE - нужно произвести переинициализацию блока
    xNeedReset:         BOOL;
    // TRUE - инициализация параметров проведена успешно,
    // FALSE - ошибки инициализации параметров
    xIsInitParams:      BOOL;
    // Флаг "ошибки инициализации параметров",
    // удерживается до заднего фронта на xExecute.
    // Используется для запрета вызова fbUnmSerialRequest
    // при некорректных параметрах запроса
    xWrongParameter:    BOOL;
    // Буфер запроса
    abyRequest:         ARRAY [0..255] OF BYTE;
    // TRUE - выделен буфер запроса
    // (входы pRequestData и szRequestData имеют корректные значения)
    xIsValidPointerForRequest: BOOL;
    // TRUE - выделен буфер ответа
    // (входы pResponseData и szResponseData имеют корректные значения)
    xIsValidPointerForResponse: BOOL;
    // Адрес устройства в запросе в BCD-формате
    dwRequestBcdAddr:   DWORD;
    // Адрес устройства в ответе BCD-формате
    dwResponseBcdAddr:  DWORD;
    // Код команды в ответе
    eResponseParam:     PULSAR\_PARAMS;

```

```

// Длина запроса в байтах
usiRequestLen:          USINT;
// Длина ответа в байтах
usiResponseLen:         USINT;
// Идентификатор пакета в запросе
uiRequestTransactionId:  UINT := 16#8A78;
// Идентификатор пакета в ответе
uiResponseTransactionId:  UINT;
// CRC в запросе
wRequestCrc:            WORD;
// CRC в ответе
wResponseCrc:           WORD;
// Буфер ответа
abyResponse:            ARRAY [0..255] OF BYTE;
END_VAR
VAR CONSTANT
// Размер адреса в байтах
c_usiAddrSize:          USINT := 4;
// Размер кода команды в байтах
c_usiFunctionSize:      USINT := 1;
// Размер длины пакета в байтах
c_usiLenSize:           USINT := 1;
// Размер идентификатора пакета в байтах
c_usiTransactionIdSize:  USINT := 2;
// Размер CRC в байтах
c_usiCrcSize:           USINT := 2;
// Позиция в буфере, начиная с которой размещается адрес
c_usiAddrBufferOffset:  USINT := 0;
// Позиция в буфере, начиная с которой размещается код команды
c_usiFunctionBufferOffset:  USINT := 4;
// Позиция в буфере, начиная с которой размещается длина пакета
c_usiLenBufferOffset:    USINT := 5;
// Позиция в буфере, начиная с которой размещаются данные
c_usiDataBufferOffset:   USINT := 6;
// Минимально возможная длина корректного ответа
c_usiMinValidResponseLen:  UDINT := 10;
// Код команды в ответе в случае присутствия в ответе кода ошибки
c_byErrorFunctionCode:   BYTE := 0;
END_VAR
//
// код ФБ
//
// [1]
// переинициализация блока перед новым вызовом
IF NOT(xExecute) AND xNeedReset THEN

    // инкрементируем идентификатор пакета для каждого нового запроса
    uiRequestTransactionId := uiRequestTransactionId + 1;

    // сбрасываем выходы блока
    xDone := FALSE;
    xBusy := FALSE;
    xError := FALSE;
    eError := PULSAR_ERROR_OCL.NO_ERROR;
    uiResponseSize := 0;

    // очищаем буферы ответа и запроса
    // чтобы на этапе отладки не видеть в буферах куски предыдущих запросов и ответов
    MEM.MemFill(ADR(abyResponse), SIZEOF(abyResponse), 0);
    MEM.MemFill(ADR(abyRequest), SIZEOF(abyRequest), 0);

    // сбрасываем флаг некорректной инициализации
    xWrongParameter := FALSE;
    // сбрасываем коммуникационный блок для возможности его следующего вызова
    fbUnmSerialRequest(xExecute := FALSE);
    // переинициализация завершена
    xNeedReset := FALSE;
END_IF

```

```

// [2]
// детектор запуска блока
fbExecute(CLK := xExecute);

IF fbExecute.Q THEN

    // попытка инициализации параметров запроса
    xIsInitParams := prvInitParams();

    // инициализация прошла успешно - поэтому формируем запрос
    IF xIsInitParams THEN
        prvCreateRequest();
    // в процессе инициализации произошла ошибка - блок не будет вызван
    ELSE
        xError := TRUE;
        eError := PULSAR_ERROR_OCL.WRONG_PARAMETER;
        xWrongParameter := TRUE;
        xNeedReset := TRUE;
    END_IF

END_IF

// [3]
// вызов коммуникационного ФБ
fbUnmSerialRequest
(
    xExecute      := xExecute AND NOT(xWrongParameter),
    tTimeout      := tTimeout,
    hCom          := hCom,
    pRequest      := ADR(abyRequest),
    // длина запроса вычисляется в методе prvCreateRequest
    szRequest     := TO_UDINT(usiRequestLen),
    pResponse     := ADR(abyResponse),
    szResponse    := SIZEOF(abyResponse),
    xBusy         => xBusy
);

// [4]
// получен ответ - надо его обработать
IF fbUnmSerialRequest.xDone THEN

    uiResponseSize := fbUnmSerialRequest.uiResponseSize;

    prvParseResponse();
    // сброс блока будет выполнен после того, как xExecute примет значение FALSE
    // (см. начало кода ФБ)
    xNeedReset := TRUE;

    // произошла ошибка (ошибка таймаута или получен некорректный)
    ELSIF fbUnmSerialRequest.xError THEN

        xError := TRUE;
        // мы не можем "напрямую" присвоить значение из OCL.ERROR в PULSAR_ERROR
        // можно было бы написать метод или функцию для этого
        // а можно и просто "переложить" содержимое экземпляра одного перечисления
        // в экземпляр другого
        // но важно убедиться, что размер обоих перечислений совпадает
        MEM.MemMove(ADR(fbUnmSerialRequest.eError), ADR(eError), SIZEOF(eError));

        // сброс блока будет выполнен после того, как xExecute примет значение FALSE
        // (см. начало кода ФБ)
        xNeedReset := TRUE;

    END_IF

```

В коде ФБ можно выделить 4 фрагмента – в листинге они помечены маркерами [1], [2], [3] и [4]. Ниже приведено их описание.

В фрагменте [1] выполняется переинициализация блока – сброс выходов и некоторых локальных переменных, а также очистка буферов. Кроме того, на этом этапе производится инкрементирование идентификатора пакета (чтобы каждый отправленный пакет имел уникальный идентификатор). Переинициализация производится после завершения работы блока (в этот момент переменной **xNeedReset** присваивается значение **TRUE**) и только в том случае, если к этому моменту вход **xExecute** уже имеет значение **FALSE** (это необходимо для соответствия [модели поведения](#)).

В фрагменте [2] производится подготовка к отправке нового запроса. За счет детектора переднего фронта весь код подготовки выполняется только один раз (а не в каждом цикле ПЛК) – это позволяет не нагружать контроллер лишними операциями. Подготовка включает в себя два действия:

- проверка параметров запроса с помощью метода **prvInitParams**. Код метода будет рассмотрен в [п. 7.4.4](#);
- формирование запроса с помощью метода **prvCreateRequest**. Код метода будет рассмотрен в [п. 7.4.5](#).

Проверка параметров может завершиться неудачей – например, если при использовании команды чтения значений каналов по указателю **pRequestData** не размещается битовая маска запрашиваемых каналов. В этом случае формирование запроса даже не начинается – происходит генерация ошибки **WRONG_PARAMETER**.

В фрагменте [3] происходит вызов экземпляра коммуникационного ФБ [UNM_SerialRequest](#) из библиотеки [OwenCommunication](#). Вызов производится только в том случае, если в прошлом фрагменте был успешно сформирован запрос – иначе блок не будет вызван, так как переменная **xWrongParameter** будет иметь значение **TRUE**.

В фрагменте [4] происходит обработка результатов вызова экземпляра коммуникационного ФБ – либо на запрос получен ответ (выход **xDone** принял значение **TRUE**) и тогда происходит его разбор с помощью метода [prvParseResponse](#) (код метода будет рассмотрен в [п. 7.4.6](#)), либо в процессе выполнения произошла ошибка (выход **xError** принял значение **TRUE**) – тогда происходит генерация ошибки на выходе [PULSAR_MASTER_OCL](#).

При этом выходы **eError** блоков [UNM_SerialRequest](#) и [PULSAR_MASTER_OCL](#) имеют разные типы ([OCL.ERROR](#) и [PULSAR_ERROR_OCL](#)), но все ошибки перечисления [OCL.ERROR](#), которые могут возникнуть в процессе работы ФБ [UNM_SerialRequest](#), присутствуют и в [PULSAR_ERROR_OCL](#). Тем не менее, присвоить значение экземпляра одного перечисления экземпляру другого нельзя.

```
// этот код вызовет ошибку компиляции, потому что переменные являются экземплярами...
// ...разных перечислений
eError := fbUnmSerialRequest.eError;
```

Поэтому выполняется копирование содержимого одной переменной в другую с помощью функции **MemMove** из библиотеки [CAA Memory](#).

7.4.4 Метод prvInitParams

Метод **prvInitParams** проверяет корректность выделения памяти буферов данных запроса и ответа.

Проверка выделения памяти связана с тем, что данные запроса передаются в ФБ [PULSAR_MASTER_OCL](#) по указателю **pRequestData**. Данные, выделенные из ответа, размещаются по указателю **pResponseData**. Соответственно, если эти указатели не будут проинициализированы при вызове экземпляра блока – то в процессе его работы может произойти [ошибка сегментации памяти](#). Поэтому перед использованием указателей необходимо проверить их корректность. При этом для разных запросов критерии корректности могут быть разными – например, запрос на чтение системного времени не включает в себя никаких данных, и поэтому вход **pRequestData** в данном случае может не использоваться, а запрос на чтение значений каналов в качестве аргумента использует битовую маску считываемых каналов, так что в этом случае указатель **pRequestData** должен быть проинициализирован. В рамках примера проверка указателя осуществляется путем сравнения его значения с константой **CAA.Constants.gc_pNULL** из библиотеки [CAA_Types](#) (фактически – выполняется проверка на неравенство нулю).

Входы **szRequestData** и **szResponseData**, определяющие размеры буферов, также проверяются на неравенство нулю с помощью сравнения с константой **CAA.Constants.gc_szZERO**. В рамках примера других проверок размера буфера не производится, но в реальном проекте они могут быть добавлены для каждого запроса – например, если известно, что в ответ на запрос придет информация, которая после обработки занимает 60 байт данных, то размер буфера **szResponseData** не должен быть менее 60 байт.

В случае успешных проверок и инициализации параметров метод возвращает значение **TRUE**, в случае ошибок – **FALSE**.

```
// метод инициализации параметров запросов
// TRUE - инициализация завершена успешно, FALSE - возникли ошибки
// (например, передан нулевой указатель в запросе, где это неприемлемо)
METHOD prvInitParams :          BOOL
VAR
END_VAR

// проверяем указатели и размеры буферов функционального блока на валидность
xIsValidPointerForRequest := (pRequestData <> CAA.Constants.gc_pNULL) AND
    (szRequestData > CAA.Constants.gc_szZERO);
xIsValidPointerForResponse := (pResponseData <> CAA.Constants.gc_pNULL) AND
    (szResponseData > CAA.Constants.gc_szZERO);

CASE eParam OF

    // пользовательский запрос
    PULSAR_PARAMS.USER_DEFINED:

        // так как запрос может не включать команды и не подразумевать ответа -
        // то инициализация всегда считается корректной
        prvInitParams := TRUE;

    // чтение данных каналов счетчика
    PULSAR_PARAMS.GET_CHANNELS_DATA:

        // в этом запросе есть аргумент (битовая маска каналов) и на него придет ответ -
        // так что нужно выделить буферы под данные запросы и ответа
        IF xIsValidPointerForRequest AND xIsValidPointerForResponse THEN
            prvInitParams := TRUE;
        END_IF
END_CASE
```

```

// чтение системного времени
PULSAR_PARAMS.GET_RTC:

    // на этот запрос придет ответ – так что под него должен быть выделен буфер
    IF xIsValidPointerForResponse THEN
        prvInitParams := TRUE;
    END_IF

ELSE
    ; // здесь можно реализовать обработку ошибки типа INVALID_COMMAND
    // она может произойти, например, если из-за ошибок при работе с памятью
    // в переменную eParam попало значение, выходящее за диапазон перечисления
END_CASE

```

7.4.5 Метод prvCreateRequest

Метод **prvCreateRequest** формирует запрос протокола Пульсар-М на основании значений входов ФБ [PULSAR_MASTER_OCL](#). Данный метод выполняет следующие операции:

- конвертацию адреса в [BCD](#)-формат;
- вычисление длины запроса;
- вычисление контрольной суммы (CRC);
- размещение в буфере всех полей запроса: адреса, кода команды, длины запроса, данных (если они присутствуют в запросе), идентификатора пакета и CRC.

Для определения позиций в буфере для размещения конкретных полей запроса используются константы блока [PULSAR_MASTER_OCL](#) и размер данных запроса (вход **szRequestData**).

При копировании в буфер значения адреса в нем меняется порядок байт – потому что спецификация протокола определяет, что адрес передается с «прямым» порядком (т. е. адрес 12345678 в запросе представляется как 0x12 0x34 0x56 0x78), но функция **DWORD_TO_BCD** (из библиотеки **Util**) возвращает значение с «обратным» порядком байт (0x78 0x56 0x34 0x12).

Для пользовательского запроса (**USER_DEFINED**) код команды определяется не входом **eParam**, а специальным входом **byUserDefinedFunction**.

При копировании CRC в буфер запроса порядок байт в ней не меняется – хотя в протоколе Пульсар-М используется та же контрольная сумма, что и в протоколе Modbus, но в отличие от Modbus она передается старшим байтом вперед.

```

// метод формирования запроса
METHOD prvCreateRequest
VAR
END_VAR

// конвертируем адрес из DEC в BCD и меняем порядок байт на противоположный
// после этого - копируем в буфер
dwRequestBcdAddr := UTIL.DWORD_TO_BCD(udiDeviceAddr);
OCL.SWAP_DATA(ADR(dwRequestBcdAddr), ADR(abyRequest[c_usiAddrBufferOffset]),
    sizeof(dwRequestBcdAddr), FALSE, FALSE, TRUE);

// для пользовательского запроса - код команды определяется
// входом byUserDefinedFunction
// для "известного" запроса - входом eParam
// копируем код команды в буфер
IF eParam = PULSAR_PARAMS.USER_DEFINED THEN
    abyRequest[c_usiFunctionBufferOffset] := TO_BYTE(byUserDefinedFunction);
ELSE
    abyRequest[c_usiFunctionBufferOffset] := TO_BYTE(eParam);
END_IF

// вычисляем длину запроса и копируем в буфер
usiRequestLen := c_usiAddrSize + c_usiFunctionSize + c_usiLenSize +
    TO_BYTE(szRequestData) + c_usiTransactionIdSize + c_usiCrcSize;
abyRequest[c_usiLenBufferOffset] := usiRequestLen;

// копируем в буфер данные запроса (если они есть)
IF szRequestData > CAA.Constants.gc_szZERO THEN
    MEM.MemMove(pRequestData, ADR(abyRequest[c_usiDataBufferOffset]),
        TO_UINT(szRequestData) );
END_IF

// копируем в буфер идентификатор пакета (он инкрементируется при сбросе блока, см.
// первые строки ФБ)
MEM.MemMove(ADR(uiRequestTransactionId),
    ADR(abyRequest[c_usiDataBufferOffset + szRequestData]),
    sizeof(uiRequestTransactionId) );

// рассчитываем и копируем в буфер CRC
// нюанс - в отличие от Modbus менять порядок байт не надо
wRequestCrc := MEM.CRC16_Modbus(ADR(abyRequest),
    abyRequest[c_usiLenBufferOffset] - c_usiCrcSize);
MEM.MemMove(ADR(wRequestCrc),
    ADR(abyRequest[c_usiDataBufferOffset + szRequestData + c_usiTransactionIdSize]),
    sizeof(wRequestCrc) );

```

7.4.6 Метод `prvParseResponse`

Метод **`prvParseResponse`** используется для проверки корректности ответа, выделения из него данных и их преобразования к нужному пользователю виду (например, системное время удобно предоставлять пользователю в виде переменной типа **DT**).

Алгоритм проверки ответа приведен в [п. 7.2](#).

В первую очередь выполняется проверка корректности длины ответа. Это принципиально важно, так как для следующих проверок выполняется обращение к массиву буфера ответа, и индексы массива вычисляются в т. ч. по длине ответа. Если **`uiResponseSize`** будет меньше 10 (например, из-за помехи на линии в COM-порт придет несколько байт «мусора») – то при безусловном выполнении этих операций из-за переполнения значения индекса массива произошла бы [ошибка сегментации памяти](#), поэтому эти операции вынесены в тело оператора **IF**.

Далее выполняется проверка ответа на ошибки, описанные в [п. 7.2](#):

- проверка адреса (совпадает ли он с адресом в запросе при условии, что адрес в запросе – не широковещательный);
- проверка кода команды – совпадает ли он с кодом команды в запросе и не равен ли при этом 0. Код команды 0 в ответе означает, что slave-устройство обнаружило ошибку в запросе master-устройства – этот случай проверяется отдельно;
- проверка длины пакета (совпадает ли значение поля длины с реальной длиной ответа);
- проверка идентификатора пакета (совпадает ли он с идентификатором в запросе);
- проверка контрольной суммы (совпадает ли CRC в ответе с рассчитанной CRC для данного ответа).

Если хотя бы одна из проверок не выполняется – то генерируется ошибка и происходит выход из метода с помощью оператора **RETURN**, потому что выделять данные из некорректного ответа не имеет смысла.

Если же ошибок не возникло, то начинается разбор ответа. Для каждого запроса эта процедура происходит по-разному.

Для пользовательского запроса (**USER_DEFINED**) поле данных ответа возвращается без какой-либо обработки. Пользователь должен реализовать обработку данных самостоятельно на основании информации из спецификации протокола.

Для запроса значений измерительных каналов поле данных ответа также возвращается без какой-либо обработки. Пользователь должен ознакомиться с описанием конкретного прибора и разместить под указателем **`pResponseData`** массив нужного размера и типа данных (размер массива зависит от битовой маски считываемых каналов, которую пользователь размещает под указателем **`pRequestData`**, тип данных – от особенностей конкретного прибора).

Ответ на запрос чтения системного времени представляет собой набор из 6 байт, в котором каждый байт соответствует одному разряду времени. Этот набор байт конвертируется в значение типа **DT** с помощью метода **`prvGetDateAndTime`**, после чего возвращается пользователю.

После завершения разбора ответа выходу **`xDone`** присваивается значение **TRUE**, чтобы проинформировать пользователя об успешном завершении работы блока.

```

// метод разбора ответа
METHOD prvParseResponse
VAR
    // рассчитанная CRC ответа
    m_wResponseCalcCrc:          WORD;
    // число байт данных в ответе
    m_usiResponseDataSize:       USINT;
    //
    m_bySlaveErrorCode:          BYTE;
    //
    // переменные для конкретных запросов
    //
    // считанные дата и время
    m_dtGetDateAndTime:          DT;
END_VAR
VAR CONSTANT
    mc_byPulsarErrorCodeEnumOffset:  BYTE := 100;
END_VAR

// проверяем, что длина ответа корректна
// это принципиально важно, так как дальше идет обращение к массиву буфера, и индексы
// массива вычисляются в т.ч. по длине ответа
// если uiResponseSize будет меньше 10 (например, из-за помехи на линии в COM-порт
// придет несколько байт "мусора") -
// то при безусловном выполнении этих операций из-за переполнения значения индекса
// массива произойдет бы ошибка сегментации памяти
// поэтому эти операции вынесены под условие
IF uiResponseSize >= c_usiMinValidResponseLen THEN

    // рассчитываем CRC ответа
    m_wResponseCalcCrc := MEM.CRC16_Modbus(ADR(abyResponse),
        uiResponseSize - c_usiCrcSize);
    // копируем CRC из ответа
    OCL.SWAP_DATA(ADR(abyResponse[uiResponseSize - c_usiCrcSize]),
        ADR(wResponseCrc), SIZEOF(wResponseCrc), FALSE, FALSE, FALSE);

    // копируем из ответа адрес устройства
    OCL.SWAP_DATA(ADR(abyResponse[c_usiAddrBufferOffset]), ADR(dwResponseBcdAddr),
        SIZEOF(dwResponseBcdAddr), FALSE, FALSE, TRUE);

    // копируем из ответа код функции
    MEM.MemMove(ADR(abyResponse[c_usiFunctionBufferOffset]), ADR(eResponseParam),
        SIZEOF(eResponseParam) );
    // копируем из ответа длину ответа
    MEM.MemMove(ADR(abyResponse[c_usiLenBufferOffset]), ADR(usiResponseLen),
        SIZEOF(usiResponseLen) );
    // копируем из ответа идентификатор пакета
    MEM.MemMove(ADR(abyResponse[uiResponseSize - c_usiCrcSize - c_usiTransactionIdSize]),
        ADR(uiResponseTransactionId), SIZEOF(uiResponseTransactionId) );

    // рассчитываем число байт данных в ответе
    m_usiResponseDataSize := TO_USINT(uiResponseSize) - c_usiCrcSize -
        c_usiTransactionIdSize - c_usiLenSize - c_usiFunctionSize - c_usiAddrSize;
ELSE
    xError := TRUE;
    eError := PULSAR_ERROR_OCL.INVALID_RESPONSE_LEN;
END_IF

```

```

// проверка совпадения адреса в ответе и запросе
// в ответе на широкопереданный запрос (на адрес 0) приходит "реальный" адрес прибора
// этот случай также надо проверить
IF (dwRequestBcdAddr <> dwResponseBcdAddr) AND (dwRequestBcdAddr <> 0) THEN
    xError := TRUE;
    eError := PULSAR_ERROR_OCL.INVALID_ADDR;
// проверка совпадения кода команды в ответе и запросе
// (для "известного" запроса и USER_DEFINED)
// если код команды = c_byErrorFunctionCode, то в ответе пришел код ошибки
// этот случай обрабатывается отдельно
ELSIF (eParam <> eResponseParam) AND (eResponseParam <> byUserDefinedFunction)
    AND (eResponseParam <> c_byErrorFunctionCode) THEN

    xError := TRUE;
    eError := PULSAR_ERROR_OCL.INVALID_FUNCTION;
// проверка совпадения значения поля длины ответа и реальной длины ответа
ELSIF usiResponseLen <> uiResponseSize THEN
    xError := TRUE;
    eError := PULSAR_ERROR_OCL.INVALID_RESPONSE_LEN;
ELSIF eResponseParam = c_byErrorFunctionCode THEN
    xError := TRUE;
    // код ошибки записан в поле данных - копируем его на выход ФБ eError
    // прибавляем смещение, так как коды ошибок в перечислении PULSAR_ERROR
    // отличаются от кодов ошибок протокола на mc_byPulsarErrorCodeEnumOffset
    m_bySlaveErrorCode := abyResponse[c_usiDataBufferOffset] +
        mc_byPulsarErrorCodeEnumOffset;
    MEM.MemMove(ADR(m_bySlaveErrorCode), ADR(eError), 1);
// проверка совпадения идентификатора пакета в запросе и ответе
ELSIF uiRequestTransactionId <> uiResponseTransactionId THEN
    xError := TRUE;
    eError := PULSAR_ERROR_OCL.INVALID_TRANSACTION_ID;
// проверка CRC
ELSIF m_wResponseCalcCrc <> wResponseCrc THEN
    xError := TRUE;
    eError := PULSAR_ERROR_OCL.INVALID_CRC;
END_IF

// если в ответе есть ошибка - то нет смысла его парсить
IF xError THEN
    RETURN;
END_IF

CASE eParam OF

    // пользовательский запрос
    PULSAR_PARAMS.USER_DEFINED:

        // возвращаем ответ без каких-либо обработок, но в пределах выделенного буфера

        IF m_usiResponseDataSize <= szResponseData THEN
            MEM.MemMove(ADR(abyResponse[c_usiDataBufferOffset]), pResponseData,
                m_usiResponseDataSize);
        ELSE
            MEM.MemMove(ADR(abyResponse[c_usiDataBufferOffset]), pResponseData,
                TO_UINT(szResponseData));
        END_IF

```

```

// чтение данных каналов счетчика
// пример ответа для запроса одного канала (HEX):
// 12 34 56 78 01 12 00 00 40 70 3D 0A 01 40 5E A4 82 37
// значение в ответе: 2.13 (LREAL)
PULSAR_PARAMS.GET_CHANNELS_DATA:

    // возвращаем ответ без каких-либо обработок, но в пределах выделенного буфера
    IF m_usiResponseDataSize <= szResponseData THEN
        MEM.MemMove(ADR(abyResponse[c_usiDataBufferOffset]), pResponseData,
            m_usiResponseDataSize);
    ELSE
        MEM.MemMove(ADR(abyResponse[c_usiDataBufferOffset]), pResponseData,
            TO_UINT(szResponseData) );
    END_IF

// чтение системного времени
// пример ответа (HEX): 12 34 56 78 04 10 0C 07 17 09 1F 1A 78 8A 1E 1C
// значение в ответе: DT#2012-7-23-9:31:26
PULSAR_PARAMS.GET_RTC:

    // обрабатываем ответ - конвертируем набор отдельных разрядов времени в DT
    m_dtGetDateAndTime := prvGetDateAndTime\(\);

    IF SIZEOF(dtGetDateAndTime) <= szResponseData THEN
        MEM.MemMove(ADR(dtGetDateAndTime), pResponseData, SIZEOF(dtGetDateAndTime) );
    ELSE
        ; // если значение не помещается в буфер - то не копируем его
        // для этого запроса размер ответа заранее известен -
        // можно организовать в методе prvInitParams...
        // ...проверку на то, что пользователь выделил буфер достаточного размера
    END_IF

END_CASE

xDone := TRUE;

```

7.4.7 Метод prvGetDateAndTime

Метод **prvGetDateAndTime** используется для конвертации 6 отдельных разрядов времени в переменную типа **DT**. Для этого применяется функция **JoinDateTime** из библиотеки [Util](#). Следует обратить внимание на то, что:

- функция возвращает метку времени в виде числа миллисекунд. Поскольку размерность типа **DT** – секунды, то результат функции делится на константу **mc_uiMillisecondsPerSecond** для приведения к секундам;
- последним аргументом функции является число миллисекунд. Так как в протоколе Пульсар-М системное время не содержит миллисекунд, то на этот вход функции при вызове передается 0;
- год в протоколе Пульсар-М кодируется двумя последними цифрами, поэтому при передаче его в функцию используется смещение **mc_uiYearValueOffset** для представления его в «полном» виде.

```
// парсинг данных запроса чтения системного времени
METHOD prvGetDateAndTime :      DT;
VAR
    // значение метки времени в секундах
    m_uliGetDateAndTime:      ULINT;
END_VAR
VAR CONSTANT
    // в протоколе Пульсар-М год передается в виде двух последних цифр
    // т.е. 12 соответствует 2012
    mc_uiYearValueOffset:      UINT := 2000;
    // число миллисекунд в секунде
    mc_uiMillisecondsPerSecond:  UINT := 1000;
END_VAR

// собираем из отдельных разрядов метку времени в секундах
// порядок разрядов в ответе:
// год (2 последние цифры года) - месяц - день - часы - минуты - секунды
// последним аргументом в функцию передается число миллисекунд (0 - так как
// в протоколе Пульсар-М наименьший разряд времени - секунды)
m_uliGetDateAndTime := UTIL.JoinDateTime(abyResponse[c_usiDataBufferOffset] +
    mc_uiYearValueOffset, abyResponse[c_usiDataBufferOffset + 1],
    abyResponse[c_usiDataBufferOffset + 2], abyResponse[c_usiDataBufferOffset + 3],
    abyResponse[c_usiDataBufferOffset + 4],
    abyResponse[c_usiDataBufferOffset + 5], 0) / mc_uiMillisecondsPerSecond;

prvGetDateAndTime := TO_DT(m_uliGetDateAndTime);
```

7.4.8 Программа PULSAR_PRG_OCL

В программе **PULSAR_PRG_OCL** реализован опрос счетчика импульсов-регистратора Пульсар с помощью ФБ [COM Control](#) из библиотеки [OwenCommunication](#) и [PULSAR MASTER OCL](#), основанном на ФБ [UNM SerialRequest](#) из библиотеки [OwenCommunication](#).

В рамках примера производится отправка всех трех поддерживаемых блоком запросов. Обмен реализован с помощью единственного экземпляра ФБ [PULSAR MASTER OCL](#), который последовательно вызывается с различными аргументами через оператор **CASE**. В качестве переменной-селектора используется экземпляр перечисления [PULSAR_PARAMS](#).

В реальном проекте пользователю может потребоваться опрашивать несколько модулей с разными адресами. В этом случае ему требуется создать свое перечисление и добавить в оператор **CASE** нужное число шагов по аналогии с примером.

Задержка между запросами определяется константой **c_tPollDelay**. Переключение между шагами оператора **CASE** осуществляется с помощью метода **prvDelayAndSwitchToNextRequest**, в котором запускается таймер задержки.

```
PROGRAM PULSAR_PRG_OCL
VAR
    // ФБ открытия COM-порта
    fbComControl:      OCL.COM Control;
    // ФБ для опроса устройств по протоколу Пульсар-М
    fbPulsarMaster:    PULSAR MASTER OCL;
    // битовая маска опрашиваемых каналов
    // в рамках примера - считывается канал 2, поэтому маска имеет значение...
    // 2#0000_0000_0000_0000_0000_0000_0000_0010 = 10#2
    dwChannelMask:     DWORD := 2;
    // значение счетчика
    lrCounterChannel:   LREAL;
    // системное время прибора
    dtGetDateAndTime:   DT;
    // вес импульса
    rPulseWeight:       REAL;

    // таймер задержки между запросами
    fbPollDelay:        TON;
    // текущий обрабатываемый запрос
    ePulsarParams:      PULSAR\_PARAMS := PULSAR_PARAMS.GET_CHANNELS_DATA;
END_VAR
VAR CONSTANT
    // задержка между запросами
    c_tPollDelay:        TIME := T#1S;
END_VAR

// открытие COM-порта
fbComControl
(
    xEnable      := TRUE,
    udiComPort    := 5,
    udiBaudrate   := 115200,
    udiByteSize   := 8,
    eParity       := OCL.COM_PARITY.NONE,
    eStopBit      := OCL.COM_STOPBIT.ONE
);
```

```

// код опроса
IF fbComControl.xActive THEN

    CASE ePulsarParams OF

        // чтение данных каналов счетчика
        PULSAR_PARAMS.GET_CHANNELS_DATA:

            fbPulsarParamsMaster
            (
                xExecute      := TRUE,
                // можно задать таймаут через константу (как уставку таймера задержки)
                tTimeout      := T#500MS,
                hCom          := fbComControl.hCom,
                eParam        := ePulsarParams,
                udiDeviceAddr := 12345678,
                // битовая маска считываемых каналов
                pRequestData  := ADR(dwChannelMask),
                szRequestData := SIZEOF(dwChannelMask),
                pResponseData := ADR(lrCounterChannel),
                szResponseData := SIZEOF(lrCounterChannel)
            );

            // задержка, сброс блока и переход к следующему запросу
            prvDelayAndSwitchToNextRequest(PULSAR_PARAMS.GET_RTC);

        // чтение системного времени прибора
        PULSAR_PARAMS.GET_RTC:

            fbPulsarParamsMaster
            (
                xExecute      := TRUE,
                // можно задать таймаут через константу (как уставку таймера задержки)
                tTimeout      := T#500MS,
                hCom          := fbComControl.hCom,
                eParam        := ePulsarParams,
                udiDeviceAddr := 12345678,
                // в этом запросе не нужны аргументы
                pRequestData  :=,
                szRequestData :=,
                pResponseData := ADR(dtGetDateAndTime),
                szResponseData := SIZEOF(dtGetDateAndTime)
            );

            // задержка, сброс блока и переход к следующему запросу
            prvDelayAndSwitchToNextRequest(PULSAR_PARAMS.USER_DEFINED);

        // пользовательский запрос (в рамках примера - чтение весов импульсов по каналам)
        PULSAR_PARAMS.USER_DEFINED:

            fbPulsarParamsMaster
            (
                xExecute      := TRUE,
                // можно задать таймаут через константу (как уставку таймера задержки)
                tTimeout      := T#500MS,
                hCom          := fbComControl.hCom,
                eParam        := ePulsarParams,
                udiDeviceAddr := 12345678,
                // битовая маска считываемых каналов
                pRequestData  := ADR(dwChannelMask),
                szRequestData := SIZEOF(dwChannelMask),
                pResponseData := ADR(rPulseWeight),
                szResponseData := SIZEOF(rPulseWeight),
                byUserDefinedFunction := 7
            );

```

```

        // задержка, сброс блока и переход к первому запросу
        prvDelayAndSwitchToNextRequest (PULSAR_PARAMS.GET_CHANNELS_DATA);

    END_CASE

END_IF

```

```

// метод перехода к следующему запросу
METHOD prvDelayAndSwitchToNextRequest
VAR_INPUT
    // следующий запрос
    m_ePulsarParams: PULSAR\_PARAMS;
END_VAR

// запуск таймера задержки между запросами
fbPollDelay(IN := fbPulsarParams.xDone OR fbPulsarParams.xError, PT := c_tPollDelay);

IF fbPollDelay.Q THEN

    // сброс блока опроса для возможности его следующего вызова
    fbPulsarParams
    (
        xExecute      := FALSE,
        // если этого не сделать, то на шаге GET_RTC (где это входы не используются) -
        // они получают свои значения из предыдущего шага,
        // что приведет к формированию некорректной команды
        pRequestData  := CAA.Constants.gc_hINVALID,
        szRequestData := CAA.Constants.gc_szZERO
    );

    // сброс таймера для возможности его следующего вызова
    fbPollDelay(IN := FALSE);

    // переход к следующему запросу
    ePulsarParams := m_ePulsarParams;

END_IF

```

7.5 Реализация протокола Пульсар-М с помощью библиотеки CAA SerialCom

Пример, описанный в [п. 7.4](#), использует блоки из библиотеки [OwenCommunication](#). Эта библиотека может использоваться только в проектах, запускаемых на контроллерах ОВЕН и на виртуальном контроллере **CODESYS Control Win V3**. Но в ряде случаев требуется, чтобы проект был полностью платформо-независимым. Для этого пользователю необходимо создать аналоги блоков [COM_Control](#) и [UNM_SerialRequest](#) из библиотеки [OwenCommunication](#). Для работы с COM-портом в данном случае будет использоваться стандартная библиотека из дистрибутива CODESYS – [CAA SerialCom](#).

Описание этих аналогов приведено в [п. 6.5.3](#) (описание ФБ [USR_COM_Control](#) – аналога [COM_CONTROL](#)) и [п. 6.5.4](#) (описание ФБ [USR_SerialRequest](#) – аналога [UNM_SerialRequest](#)).

7.5.1 Перечисление PULSAR_ERROR_CAA

Перечисление **PULSAR_ERROR_CAA** содержит ошибки, которые могут возникнуть при вызове ФБ [PULSAR_MASTER_CAA](#). Перечисление включает в себя коды ошибок из перечисления [ERROR](#) библиотеки [CAA SerialCom](#) и специфические коды ошибок, определенные в [п. 7.4.1](#). Числовой тип перечисления (INT) выбран в соответствии с числовым типом перечисления [COM.ERROR](#), чтобы упростить конвертацию одного перечисления в другое.

Таблица 7.5.1 – Описание элементов перечисления PULSAR_ERROR_CAA

Название	Значение	Описание
Общие ошибки		
NO_ERROR	0	Нет ошибок
TIME_OUT	5001	Ошибка таймаута
HANDLE_INVALID	5003	Некорректное значение дескриптора интерфейса
ERROR_UNKNOWN	5004	Неизвестная ошибка (зарезервировано для будущих версий)
WRONG_PARAMETER	5005	Как минимум один из аргументов ФБ имеет некорректное значение
WRITE_INCOMPLETE	5006	Отправка сообщения не была завершена
Специфические ошибки протокола Пульсар-М		
INVALID_FUNCTION	101	Отсутствует запрашиваемый код функции
INVALID_MASK	102	Ошибка в битовой маске запроса
INVALID_REQUEST_LEN	103	Ошибочная длина запроса
INVALID_PARAM	104	Отсутствует параметр
AUTHORIZATION_REQUIRED	105	Запись заблокирована, требуется авторизация
PARAM_OUT_OF_RANGE	106	Записываемое значение (параметр) находится вне заданного диапазона
NO_ARCHIVE_TYPE	107	Отсутствует запрашиваемый тип архива
TOO_MUCH_ARCHIVE_DATA	108	Превышение максимального количества считываемых архивных значений за один пакет
INVALID_RESPONSE_LEN	120	Некорректная длина ответа (<10 байт или значение поля LEN не соответствует фактической длине ответа)
INVALID_ADDR	121	Некорректный адрес устройства в ответе
INVALID_TRANSACTION_ID	122	Некорректный идентификатор пакета в ответе
INVALID_CRC	123	Некорректная CRC в ответе

```

{attribute 'qualified_only'}
{attribute 'strict'}
// Ошибки протокола Пульсар-М (при реализации с помощью библиотеки CAA SerialCom)
TYPE PULSAR_ERROR_CAA :
(
    // Стандартные коды ошибок для ФБ COM.Read и COM.Write
    NO_ERROR := 0,
    TIME_OUT := 5001,
    HANDLE_INVALID := 5003,
    ERROR_UNKNOWN := 5004,
    WRONG_PARAMETER := 5005,
    WRITE_INCOMPLETE := 5006,
    //
    // Специфические ошибки протокола Пульсар-М
    //
    // Отсутствует запрашиваемый код функции
    INVALID_FUNCTION := 101,
    // Ошибка в битовой маске запроса
    INVALID_MASK := 102,
    // Ошибочная длина запроса
    INVALID_REQUEST_LEN := 103,
    // Отсутствует параметр
    INVALID_PARAM := 104,
    // Запись заблокирована, требуется авторизация
    AUTHORIZATION_REQUIRED := 105,
    // Записываемое значение (параметр) находится вне заданного диапазона
    PARAM_OUT_OF_RANGE := 106,
    // Отсутствует запрашиваемый тип архива
    NO_ARCHIVE_TYPE := 107,
    // Превышение максимального количества считываемых архивных значений за один пакет
    TOO_MUCH_ARCHIVE_DATA := 108,
    // Некорректная длина ответа (<10 байт или значение поля LEN не соответствует
    // фактической длине ответа)
    INVALID_RESPONSE_LEN := 120,
    // Некорректный адрес устройства в ответе (не совпадает с адресом в запросе)
    INVALID_ADDR := 121,
    // Некорректный идентификатор пакета в ответе (не совпадает с идентификатором
    // в запросе)
    INVALID_TRANSACTION_ID := 122,
    // Некорректная CRC в ответе
    INVALID_CRC := 123
) INT; // для гарантированного совпадения по размеру с COM.ERROR
END_TYPE

```

7.5.2 ФБ PULSAR_MASTER_CAA

Функциональный блок **PULSAR_MASTER_CAA** отличается от блока [PULSAR_MASTER_OCL](#) лишь следующим:

- выход **eError** имеет тип [PULSAR_ERROR_CAA](#) вместо [PULSAR_ERROR_OCL](#) (потому что коды ошибок блоков [COM.Read](#) / [COM.Write](#) и [UNM_SerialRequest](#) отличаются);
- в коде ФБ **PULSAR_MASTER_CAA** и его метода **prvParseResponse** при обращении к выходу **eError** должно использоваться перечисление [PULSAR_ERROR_CAA](#) вместо [PULSAR_ERROR_OCL](#);
- переменная **fbUnmSerialRequest** имеет тип [USR_SerialRequest](#) вместо [UNM_SerialRequest](#).

Других отличий нет.

7.5.3 Программа PULSAR_PRG_CAA

Программа **PULSAR_PRG_CAA** отличается от программы [PULSAR_PRG_OCL](#) лишь следующим:

- переменная **fbComControl** имеет тип [USR_COM_Control](#) вместо [OCL.COM_Control](#);
- в вызове **fbComControl** используются перечисления [COM.PARITY](#) и [COM.STOPBIT](#) вместо [OCL.COM_PARITY](#) и [OCL.COM_STOPBIT](#);
- переменная **fbPulsarMaster** имеет тип [PULSAR_MASTER_CAA](#) вместо [PULSAR_MASTER_OCL](#).

Других отличий нет.

8 Дополнительные вопросы

8.1 Нумерация COM-портов в CODESYS

Во время настройки интерфейсов RS-485/RS-232 следует указывать номера портов. Для контроллеров OWEN эти номера приведены в таргет-файле устройства в узле **Device** на вкладке **Информация**:

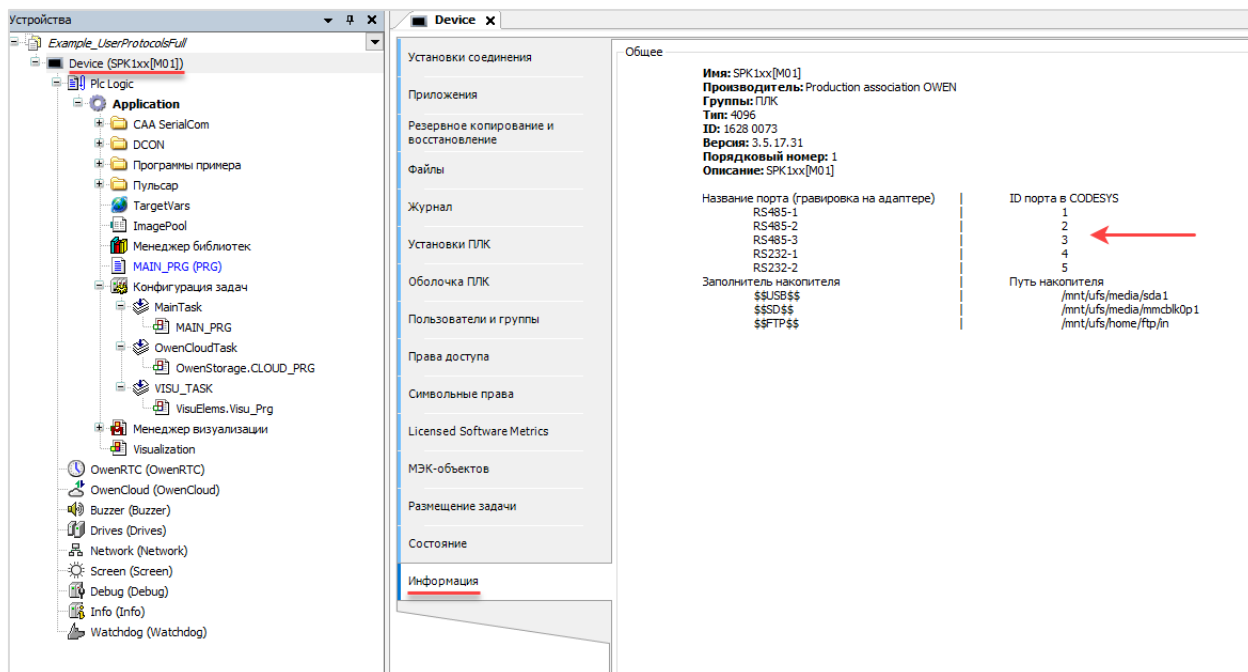


Рисунок 8.1 – Информация о нумерации COM-портов в таргет-файле

8.2 Ограничения, связанные с библиотекой CAA AsyncManager

Библиотеки, включающие в себя [асинхронно выполняемые функциональные блоки](#) (к таким библиотекам, например, относятся **CAA SerialCom**, **CAA NetBaseServices**, **CAA File** и др.), используют в своей реализации библиотеку **CAA AsyncManager**.

Данная библиотека имеет следующее ограничение: **в каждый момент времени (в пределах одного цикла контроллера) в программе может вызываться не более 20 экземпляров асинхронно выполняемых функциональных блоков.**

Под «вызовом» подразумевается исполнение кода, в котором экземпляру асинхронно выполняемого блока передается на вход **xExecute** значение **TRUE** или в котором выход **xBusy** имеет значение **TRUE**.

Библиотека [OwenCommunication](#) построена на блоках библиотеки **CAA SerialCom**, и также подвержена этим ограничениям, причем ФБ [COM_Control](#) включает в себя один экземпляр асинхронно выполняемого блока ([COM.Open](#) или [COM.Close](#); в каждый момент времени выполняется только один из них), а ФБ [UNM SerialRequest](#) – два ([COM.Write](#) и [COM.Read](#); при этом на момент вызова [COM.Read](#) вход **xExecute** блока [COM.Write](#) все еще имеет значение **TRUE**). На практике это означает следующее: если одновременно (в пределах одного цикла контроллера),

вызывать 3 экземпляра ФБ [COM_Control](#), 3 экземпляра ФБ [UNM_SerialRequest](#) и, например, 11 экземпляров ФБ [FILE.Write](#) из библиотеки **CAA File**, то лимит для асинхронно выполняемых блоков окажется исчерпанными – при попытке вызова в том же цикле ПЛК еще одного экземпляра он вернет код ошибки (для разных библиотек этот код будет разным, например, для [OwenCommunicaton](#) – **WRONG_PARAMETER**, а для CAA File – [5802](#)).

На практике это ограничение обычно не создает существенных проблем при программировании на языке ST – общепринятый подход подразумевает «сброс» (вызов со значением **FALSE** на входе **xExecute**) блока после его выполнения; этот подход используется и в примерах документа.

Но в языке CFC цепочка блоков на холсте обрабатывается более специфическим образом – даже при визуальном значении **FALSE** на входе **xExecute** какого-либо блока его «внутренние» экземпляры асинхронно выполняемых блоков могут сохранять значение **TRUE** на своих входах **xExecute** до следующего «прохода» данного ФБ на холсте и учитываться в упомянутом выше ограничении. Реализовать «сброс» каждого экземпляра на холсте после его выполнения в графическом языке программирования достаточно трудоемко; поэтому рекомендуется для решения сложных задач (реализация протоколов обмена, работа с файлами и т. д.) использовать исключительно язык ST.

Ниже приведен пример проявления ограничения на число одновременно вызываемых асинхронно выполняемых функциональных блоков в языке CFC: на холсте расположены экземпляр [COM_Control](#) (включает 1 экземпляр асинхронно выполняемого блока) и 10 экземпляров **MB_SerialRequest** (как и [UNM_Serial_Request](#) – включает 2 экземпляра асинхронно выполняемых блоков). Из-за особенностей языка CFC «сброс» блоков происходит только перед началом нового «прохода» холста, и поэтому 10-й экземпляр **MB_SerialRequest** возвращает ошибку **WRONG_PARAMETER** (она возникает на этапе вызова в нем экземпляра ФБ [COM.Read](#), потому что он является 21-м экземпляром асинхронно выполняемых блоков на холсте).

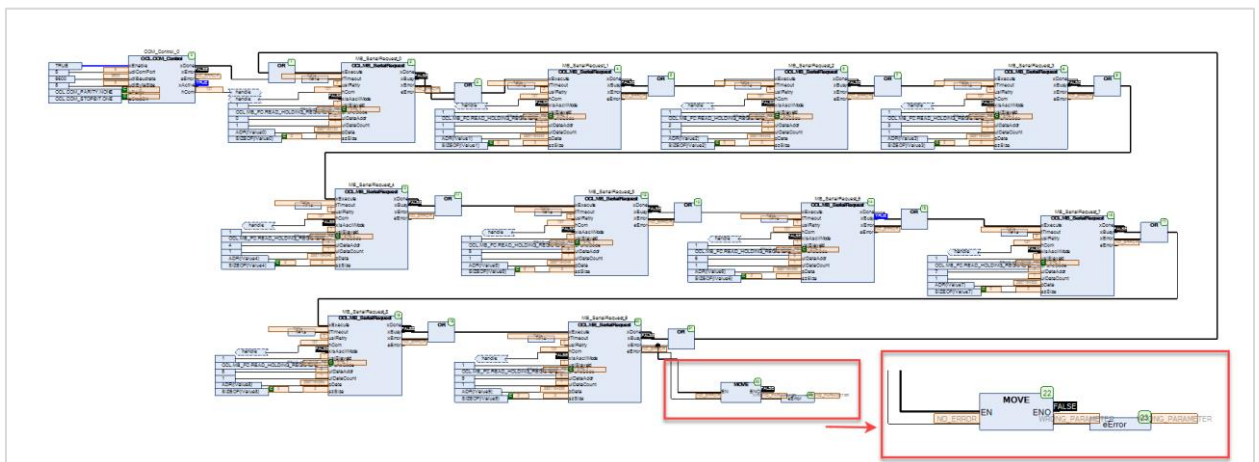


Рисунок 8.2 – Пример проявления ограничения на число одновременно вызываемых асинхронно выполняемых функциональных блоков в языке CFC

8.3 Реализация обмена по нестандартному протоколу в режиме Slave

В ряде специфических случаев оборудование с поддержкой нестандартного протокола работает в режиме master – например, сканер штрихкодов отправляет в порт строку символов после очередной операции сканирования, джойстик циклически присылает информацию о своих текущих координатах и т. д.

См. пример решения подобной задачи: Example_UnmSerialSlave_3514v1.projectarchive

Пример создан в среде **CODESYS 3.5 SP14 Patch 3** и подразумевает запуск на **СПК1xx [M01]** с таргет-файлом **3.5.14.34**. В случае необходимости запуска проекта на другом устройстве следует изменить таргет-файл в проекте (**ПКМ** на узел **Device** – **Обновить устройство**).

В примере контроллер ожидает команды **TEST** (именно в верхнем регистре) и отвечает на нее сообщением **OK**. При получении любой другой команды контроллер ничего не отвечает (если переменная **xUseExceptionCode** имеет значение **FALSE**) или отправляет в ответ сообщение **ILLEGAL_COMMAND** (если переменная **xUseExceptionCode** имеет значение **TRUE**).

На базе примера можно реализовать свой протокол обмена, используя контроллер в режиме slave.

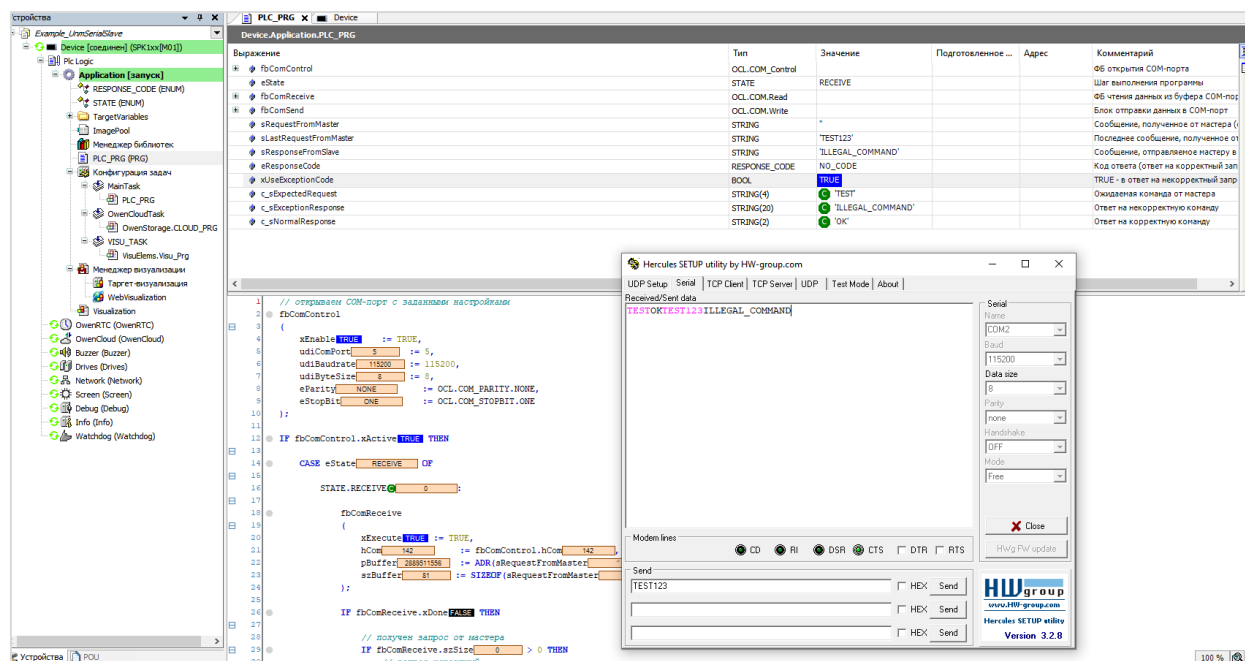


Рисунок 8.3 – Проверка работы примера с помощью утилиты [Hercules SETUP Utility](http://www.hw-group.com)

8.4 Отладка обмена по нестандартному протоколу без контроллера или опрашиваемого устройства

Рассмотренные в документе примеры могут быть запущены на виртуальном контроллере **CODESYS Control Win V3**. В качестве номера COM-порта в данном случае следует использовать номер виртуального COM-порта ПК. Более подробная информация о запуске виртуального контроллера приведена в документе **CODESYS V3.5. FAQ** и [этом видео](#).

Если опрашиваемое устройство отсутствует, то можно подключить контроллер к ПК с помощью конвертера RS-485/USB или RS-232/USB, и эмулировать команды устройства с помощью какой-нибудь терминальной утилиты – например, [Hercules SETUP Utility](#).

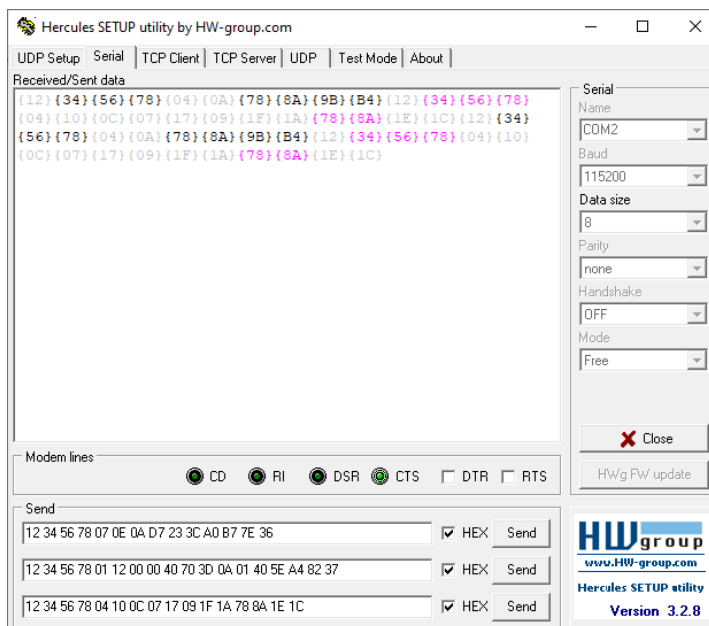


Рисунок 8.4 – Эмуляция ответов устройства с помощью утилиты Hercules SETUP Utility