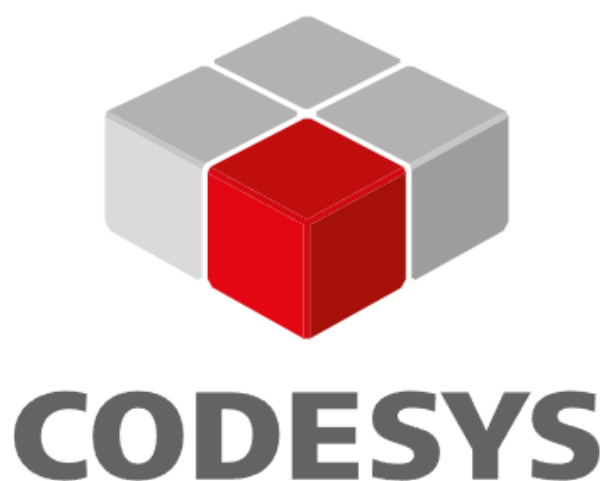


# Работа с памятью в CODESYS V3



22.02.2022  
версия 1.0

## Оглавление

Оглавление.....	2
Введение .....	3
1. Инструменты, полезные при конвертации данных.....	4
1.1. Структуры .....	4
1.1.1. Общая информация.....	4
1.1.2. Наследование структур .....	6
1.1.3. Использование структур для хранения битовых масок (тип BIT) .....	8
1.1.4. Выравнивание памяти в структурах.....	11
1.2. Перечисления .....	14
1.2.1. Общая информация.....	14
1.2.2. Атрибуты перечислений .....	18
1.2.3. Использование перечислений со встроенными списками текстов .....	21
1.2.4. Локальные перечисления .....	23
1.3. Объединения .....	24
1.4. Библиотека CAA Memory .....	26
1.5. Библиотека OwenCommunication.....	28
1.6. Функции для работы с BCD .....	30
2. Примеры конвертации данных .....	31
2.1. «2 WORD в REAL» и т.д. ....	31
2.1.1. Функции библиотеки OwenCommunication .....	33
2.1.2. Функции библиотеки CAA Memory .....	34
2.1.3. Объединения .....	35
2.1.4. Не только ведь REAL .....	37
2.2. Паспортные данные электросчетчика Меркурий.....	39
2.3. Текущие измерения счетчика Питерфлоу .....	46
2.4. Счетчики весового преобразователя ТВ-011 (протокол «Тензо-М»).....	51
3. Особенности конвертации строк .....	53

## Введение

Одной из типичных задач программируемых логических контроллеров является обмен данными с другими устройствами для мониторинга и управления. В большинстве современных промышленных протоколов (EtherCAT, Profinet, EtherNet/IP и т.д.) для настройки обмена используются файлы описания. Пользователь импортирует эти файлы в свою среду разработки и видит в ней доступные параметры устройства с именами, типами и комментариями. Ему остается только связать эти параметры с переменными ПЛК.

С другой стороны, некоторые протоколы не поддерживают ни стандартизированные файлы описания, ни даже типизацию данных. К числу таких протоколов относится, например, Modbus (который по состоянию на 2021 [занимает 10% рынка промышленных протоколов](#)), различные проприетарные протоколы тепло- и электросчетчиков и т.д.

В этом случае данные, получаемые от устройства (и отправляемые на него), представляют собой набор байт, который разработчику требуется в программе контроллера преобразовать к нужному набору переменных. В данной статье приведен ряд рекомендаций по решению этой задачи в среде программирования **CODESYS V3.5**.

В [п. 1](#) перечислены типы данных, функции и библиотеки, которые могут оказаться полезными при конвертации данных. Если вы уже имеете представление обо всех этих средствах – то можете сразу переходить к [п. 2](#), в котором приведены конкретные примеры конвертации данных.

В [п. 3](#) рассмотрены некоторые аспекты по работе со строками и преобразованию кодировок.

Подразумевается, что читатель имеет базовые знания о стандарте МЭК 61131-3 и опыт работы в среде CODESYS.

**Автор:** Евгений Кислов

## 1. Инструменты, полезные при конвертации данных

### 1.1. Структуры

#### 1.1.1. Общая информация

Структура – это пользовательский тип данных (DUT), включающий в себя набор значений одного или различных (обычно – различных) типов. Использование структур повышает модульность кода и в ряде случаев делает его более простым.

Для создания структуры в **CODESYS V3.5** необходимо нажать **ПКМ** на узел **Application** и выбрать команду **Добавление объекта – DUT**.

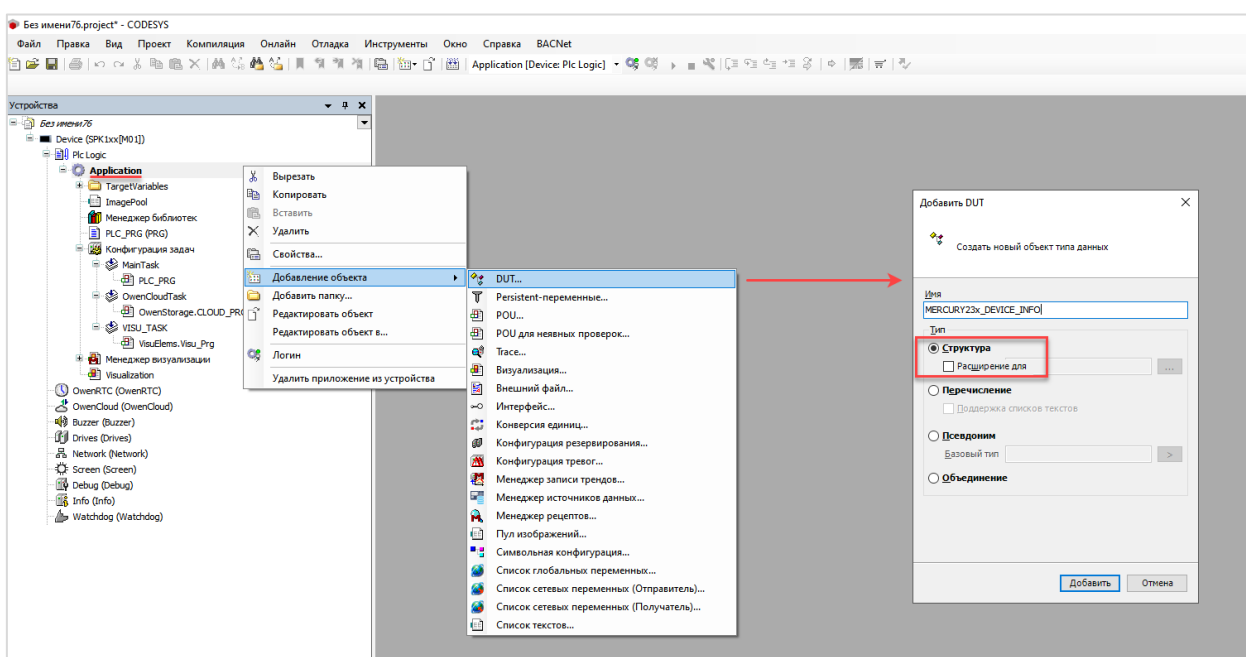


Рис. 1.1.1. Создание структуры в CODESYS V3.5

В случае установки галочки **Расширение для** можно указать для создаваемой структуры родительскую структуру. В этом случае в созданную структуру будут неявно (без отображения в списке) добавлены элементы родительской структуры (см. [п. 1.1.2](#)).

Рассмотрим пример использования структуры.

В [спецификации протокола электросчетчиков Меркурий 23х](#) описан запрос **Серийный номер счетчика и дата выпуска** (п. 4.4.2):

*«В ответ на запрос счетчик возвращает 7 байт в поле данных ответа. Первые 4 байта – серийный номер в двоичном позиционном коде, следующие 3 байта – дата выпуска в двоичном позиционном коде (каждый байт интерпретируется отдельно) в последовательности: число, месяц, год.»* (пример разбора ответа на этот запрос приведен в [п. 2.2](#)).

**Пример:** Прочитать серийный номер и дату выпуска счетчика с сетевым адресом 128 (0x80).

Запрос: 80 08 00 (CRC).

Ответ: 80 29 5A 40 43 16 06 14 (CRC), где:

- «29 5A 40 43» – серийный номер 41906467;
- «16 06 14» – дата выпуска 22.06.2020.

Данные, которые считываются этим запросом, можно представить в виде подобной структуры:

```

1  TYPE MERCURY23x_DEVICE_INFO :
2  STRUCT
3      // Серийный номер
4      udiSerialNumber:    UDINT;
5      // Дата выпуска прошивки
6      dManufactureDate:  DATE;
7  END_STRUCT
8  END_TYPE
9

```

Рис. 1.1.2. Элементы структуры **MERCURY23x\_DEVICE\_INFO**

Отдельно следует обсудить выбор типов данных.

Серийный номер счетчика содержит только цифры (это известно из руководства по эксплуатации), количество которых не превышает восьми (т.е. «максимально возможный» номер счетчика – 99999999) – поэтому для него выбран тип **UDINT** (беззнаковый целый с диапазоном 0...4294967295).

Дату выпуска прошивки удобно представить типом **DATE**, который как раз предназначен для хранения дат.

Для работы со структурой в коде программы требуется объявить ее экземпляр, который будет использоваться для хранения конкретного набора значений. Для обращения в коде к элементам структуры требуется ввести имя экземпляра, поставить точку и указать имя нужного элемента (среда разработки сама поможет сделать это с помощью выпадающего списка). Ниже приведен пример кода с доступом к элементу структуры.

```

1  PROGRAM PLC_PRG
2  VAR
3      stDeviceInfo:    MERCURY23x_DEVICE_INFO;
4      xSoOldDevice:    BOOL;
5  END_VAR
6
7      xSoOldDevice := (stDeviceInfo.dManufactureDate < DATE#2014-1-1);
8
9

```

Рис. 1.1.3. Объявление экземпляра структуры и доступ к его элементу в коде

### 1.1.2. Наследование структур

Рассмотрим еще один запрос из [спецификации протокола Меркурий 23x – Паспортные данные](#) (п. 4.4.3):

**Пример:** Прочитать параметры прибора с сетевым адресом 66 (0x42).

Запрос: 42 08 01 (CRC).

Ответ: 42 20 57 2F 42 1A 06 12 09 00 00 B4 E3 C2 97 DF 58 (CRC), где:

- «20 57 2F 42» – серийный номер 32874766;
- «1A 06 12» – дата выпуска 26.06.2018;
- «09 00 00» – версия ПО 9.0.0;
- «B4 E3 C2 97 DF 58» – вариант исполнения (см. п. 4.4.16).

Как можно заметить – этот ответ включает в себя два элемента из ответа на запрос **Серийный номер счетчика и дата выпуска**. Соответственно, мы можем создать новую структуру, наследуя ее от **MERCURY23x\_DEVICE\_INFO**:

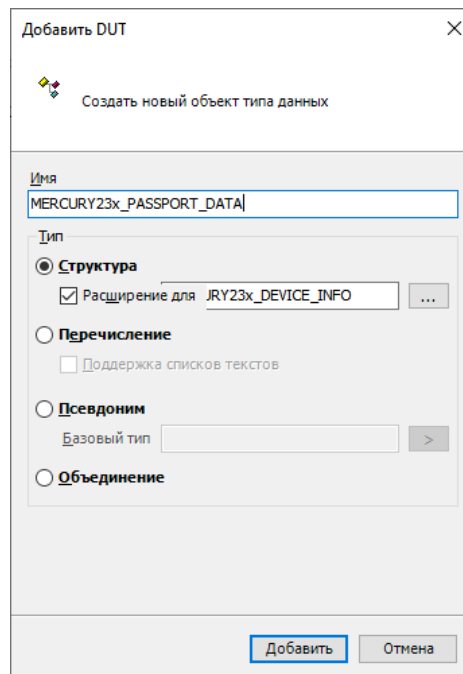


Рис. 1.1.4. Создание структуры с наследованием от другой структуры (на скриншоте виден баг русскоязычной локализации CODESYS – перевод закрывает часть поля ввода имени родительской структуры)

Объявление такой структуры будет включать в себя ключевое слово **EXTENDS**, которое указывает на имя родительской структуры. CODESYS не поддерживает множественное наследование структур – у каждой структуры может быть только одна родительская структура. **Обратите внимание**, что в объявлении структуры-наследника не отображаются элементы родительской структуры **MERCURY23x\_DEVICE\_INFO** – они добавлены в нее неявно. Увидеть их можно при выборе элемента экземпляра структуры в коде программы (см. рис. 1.1.6).

Объявим в созданной структуре две дополнительные переменные – версию прошивки и вариант исполнения.

Версию прошивки удобно представить в виде строковой переменной (потому что она представляет собой три числа, разделенных точками – и для такого отображения подойдет только строка). Мы знаем, что каждый разряд версии прошивки занимает один байт – т.е. «максимальная» версия прошивки будет выглядеть как **255.255.255**. Соответственно, мы можем ограничить длину строки 11 символами, чтобы сэкономить используемую память.

Вариант исполнения представляет собой 6 байт, в которых зашифровано несколько десятков параметров счетчика (на хранение каждого параметра отводится от 1 до 4 бит). В данный момент мы не будем акцентироваться на этой особенности и представим вариант исполнения в виде массива байт; как преобразовать такой набор данных в удобный для человека вид – рассмотрим в [п. 2.2](#).

```

MERCURY23x_PASSPORT_DATA x
1  TYPE MERCURY23x_PASSPORT_DATA EXTENDS MERCURY23x_DEVICE_INFO :
2  STRUCT
3      // Версия прошивки
4      sFirmwareVersion:  STRING(11);
5      // Вариант исполнения
6      abyModelInfo:      ARRAY [0..5] OF BYTE;
7  END_STRUCT
8  END_TYPE

```

Рис. 1.1.5. Добавление новых элементов в структуру-наследник

```

PLC_PRG x
1  PROGRAM PLC_PRG
2  VAR
3      stDeviceInfo:  MERCURY23x_DEVICE_INFO;
4      stPassportData: MERCURY23x_PASSPORT_DATA;
5      xSoOldDevice:  BOOL;
6  END_VAR

1
2  xSoOldDevice := (stDeviceInfo.dManufactureDate < DATE#2014-1-1);
3
4  stPassportData.
5
6  abyModelInfo
7  dManufactureDate
8  sFirmwareVersion
9  udiSerialNumber
10

```

Рис. 1.1.6. Структура-наследник неявно включает в себя элементы родительской структуры – в данном случае, **udiSerialNumber** и **dManufactureDate** из структуры **MERCURY23x\_DEVICE\_INFO**

### 1.1.3. Использование структур для хранения битовых масок (тип BIT)

Достаточно часто логические параметры прибора хранятся в виде битовой маски. Битовая маска представляет собой переменную, каждый бит которой характеризует один из параметров прибора.

Приведем пример из [документации на расходомер Питерфлоу](#):

**4.4 Текущие измерения**

Название	Адрес	Тип	Доступ	Примечание
Часы реального времени	10500	date time	R/O	
Время наработки (мин.)	10503	unsigned long	R/O	
Интеграл V+ (м³)	10505	double	R/O	Интеграл объема в прямом направлении
Интеграл V- (м³)	10509	double	R/O	Интеграл объема в обратном направлении
Флаги событий	10513		R/O	Набор битовых флагов в соответствии с <a href="#">Приложением 7</a>
Время наработки с ошибкой (мин.)	10515	unsigned long	R/O	
Расход (м³/ч)	10517	float	R/O	
Код АЦП	10519	float	R/O	
Напряжение питания (В)	10521	float	R/O	
Температура индуктора (°C)	10523	float	R/O	
Остаточная емкость батареи (А*ч)	10525	float	R/O	
Сопротивление среды (кОм)	10527	float		
Флаги состояния	10529	unsigned long		Набор битовых флагов в соответствии с <a href="#">Приложением 8</a>
Ток индуктора (мА)	10531	float		

**Приложение 8. Флаги состояния аппаратуры**

Флаги состояния прибора представлены битовой маской:

- бит 0 - отсутствие батарейки;
- бит 1 - питание от 5В;
- бит 2 - питание от 12В;
- бит 3 - питание от батареи.

Значения остальных битов могут быть представлены в шестнадцатеричном виде.

Рис. 1.1.7. Описание параметров расходомера Питерфлоу

Параметр **Флаги состояния** имеет тип unsigned long (32-битное беззнаковое целое) и при этом, как указано в примечании, представляет собой набор битовых флагов. Фактически используются только 4 младших бита – остальные, вероятно, оставлены как резерв для будущих версий ПО или других приборов на этой же аппаратной платформе. Смысл фразы «значения остальных битов могут быть представлены в шестнадцатеричном виде», честно говоря, для нас остался непонятным – возможно, имеется в виду, что остальные биты параметра являются незначимыми и никак не интерпретируются.

В CODESYS для представления 32-битных беззнаковых чисел используется тип **UDINT**. Соответственно, мы могли бы объявить переменную такого типа и в коде обращаться к ее отдельным битам:

```

1  PROGRAM PLC_PRG
2  VAR
3      udiStateFlags: UDINT;
4  END_VAR

1  // если отсутствует батареяка
2  IF udiStateFlags.0 THEN
3      // выполняем какой-то код
4  END_IF
5

```

Рис. 1.1.8. Доступ к битам целочисленной переменной

Недостатком этого кода является то, что без комментария он является совершенно нечитаемым – даже разработчик, который решал эту задачу, рано или поздно забудет о том, какой бит какому параметру соответствует. Если потребуется внести какие-то изменения в программу – придется открывать описание протокола и вспоминать назначение каждого бита.

Поэтому можно скопировать биты в отдельные логические переменные с понятными именами:

```

1  PROGRAM PLC_PRG
2  VAR
3      dwStateFlags: DWORD;
4
5      xIsNoBattery: BOOL;
6      xIsPowerFrom5V: BOOL;
7      xIsPowerFrom12V: BOOL;
8      xIsPowerFromBattery: BOOL;
9  END_VAR
10
1  xIsNoBattery := dwStateFlags.0;
2  xIsPowerFrom5V := dwStateFlags.1;
3  xIsPowerFrom12V := dwStateFlags.2;
4  xIsPowerFromBattery := dwStateFlags.3;
5
6  // этот код понятен без комментариев
7  IF xIsNoBattery THEN
8      // выполняем какой-то код
9  END_IF
10

```

Рис. 1.1.9. Копирование битов переменной в отдельные BOOL-переменные

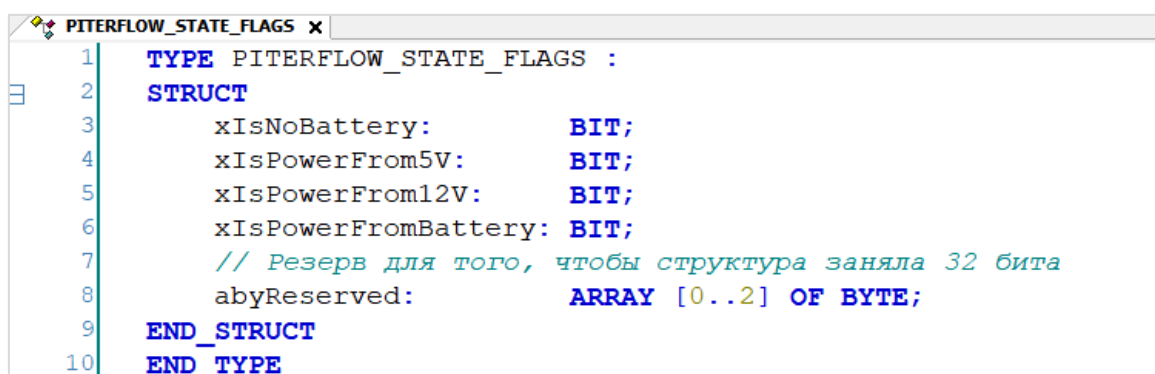
Этот вариант кода читабельнее предыдущего – все флаги состояний представлены переменными с понятными именами. Правда, для хранения каждой такой переменной потребовалась дополнительная память (переменная типа **BOOL** занимает в CODESYS 8 бит памяти)

и дополнительные ресурсы процессора на копирование переменных, но в значительном числе (*хотя и не во всех*) случаях разработчику, использующему ПЛК с CODESYS V3.5, вполне допустимо не задумываться об этом – сам факт поддержки CODESYS V3.5 на ПЛК говорит о достаточно серьезных аппаратных ресурсах.

Отметим еще один момент – при изменении кода мы также поменяли тип переменной с **UDINT** на **DWORD**. Давайте зададим себе интересный вопрос – в чем вообще разница между этими типами?

В соответствии со стандартом МЭК 61131-3 типы **USINT/UINT/UDINT/ULINT** относятся к категории «целые беззнаковые числа», а **BYTE/WORD/DWORD/LWORD** – к категории «наборы бит» (bit strings). Для целых чисел определены арифметические операции. Для наборов бит определен побитовый доступ к данным. CODESYS V3.5 позволяет производить арифметические операции и обеспечивает побитовый доступ к любому из этих типов данных – так что фактически отличия отсутствуют. Поддержка всех типов сохранена для соответствия стандарту и совместимости с другим ПО. В целом, можно порекомендовать использовать **UDINT** (и остальные «числовые» типы) для представления исчисляемых величин (например, количества подсчитанных импульсов, числа произведенных продуктов и т.д.), а наборы бит - для представления бинарных данных (например, битовой маски состояния входов/выходов, кода ошибки и т.д.).

Итак, вариант с отдельными логическими переменными позволил нам сделать код читабельнее. Но мы можем сделать его еще более простым, воспользовавшись особенностью<sup>1</sup> структур – поддержкой переменных типа **BIT**.



```

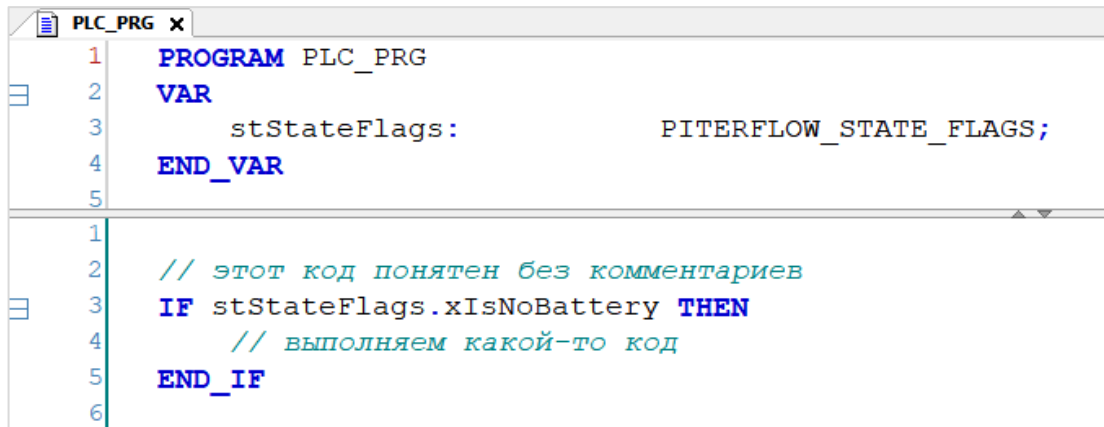
1  TYPE PITERFLOW_STATE_FLAGS :
2  STRUCT
3      xIsNoBattery:          BIT;
4      xIsPowerFrom5V:       BIT;
5      xIsPowerFrom12V:      BIT;
6      xIsPowerFromBattery: BIT;
7      // Резерв для того, чтобы структура заняла 32 бита
8      abyReserved:         ARRAY [0..2] OF BYTE;
9  END_STRUCT
10 END_TYPE

```

Рис. 1.1.10. Объявление в структуре элементов типа **BIT**

Каждый элемент типа **BIT** занимает один бит в памяти контроллера (*в отличие от переменных типа **BOOL** – как мы уже упоминали, они занимают по 8 бит*). При этом память под хранение таких переменных выделяется порциями по 8 бит – экземпляр структуры с одной **BIT**-переменной займет 8 бит памяти, а экземпляр с 9-ю **BIT**-переменными – 16. В нашей ситуации исходный параметр **Флаги состояния** занимает 32 бита; чтобы экземпляр нашей структуры занял столько же места (это может потребоваться при «прямом» копировании области памяти массива байт приемного буфера в структуру наших данных – мы рассмотрим это в [п. 2.3](#)) мы добавим в него массив из трех байт (каждый байт занимает 8 бит и еще 8 бит было выделено при объявлении первого элемента типа **BIT**).

<sup>1</sup> Переменные типа **BIT** могут быть объявлены только в структурах, [объединениях](#) и функциональных блоках.



```

1  PROGRAM PLC_PRG
2  VAR
3      stStateFlags:          PITERFLOW_STATE_FLAGS;
4  END_VAR
5
6  // этот код понятен без комментариев
7  IF stStateFlags.xIsNoBattery THEN
8      // выполняем какой-то код
9  END_IF

```

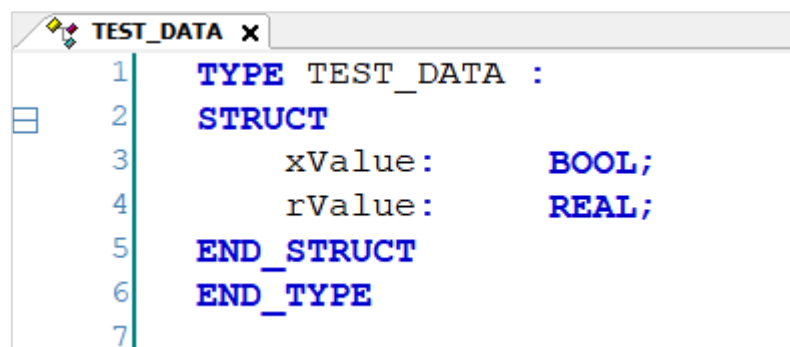
Рис. 1.1.11. Работа с BIT-переменными структуры в коде программы

Наш код не утратил читабельности по сравнению с прошлым вариантом, но стал более компактным. Отметим, что справка CODESYS [указывает](#) на то, что доступ к элементам типа **BIT** является более ресурсоемким, чем к элементам типа **BOOL**; впрочем, доступ к отдельным битам переменной типа **DWORD** (как в предыдущем варианте) является сопоставимым по ресурсоемкости – так что нельзя сказать, что вариант с BIT-переменными как-то негативно повлиял на производительность приложения.

Напоследок еще раз отметим, что объявление переменных типа **BIT** возможно только в структурах, объединениях и функциональных блоках.

#### 1.1.4. Выравнивание памяти в структурах

Самый важный вопрос использования структур связан с концепцией выравнивания памяти. Проще всего пояснить влияние выравнивания на простом примере. Пусть у нас есть структура, включающая в себя элементы типа **BOOL** и **REAL**.



```

1  TYPE TEST_DATA :
2  STRUCT
3      xValue:          BOOL;
4      rValue:          REAL;
5  END_STRUCT
6  END_TYPE
7

```

Рис. 1.1.12. Пример структуры для изучения выравнивания памяти

Мы знаем, что переменная типа **BOOL** занимает 1 байт памяти, а переменная типа **REAL** – 4 байта. Соответственно, интуитивно можно предположить, что экземпляр структуры займет 5 байт в памяти ПЛК. Но проверив эту гипотезу с помощью оператора **SIZEOF** на нашем конкретном ПЛК – мы получили 8. Для других ПЛК результат мог бы быть иным. Наблюдаемый эффект связан с *выравниванием памяти*.

В [статье на википедии](#) приведен необходимый минимум для понимания этой концепции (а в англоязычной версии статьи есть конкретные примеры):

*«Центральные процессоры в качестве основной единицы при работе с памятью используют машинное слово, размер которого может быть различным. Однако размер слова всегда равен нескольким байтам (байт является наименьшей единицей, в которой отсчитываются адреса). Как правило, машинное слово равно  $2^k$  байтам, то есть состоит из одного, двух, четырёх, восьми и т. д. байтов.*

*При сохранении какого-то объекта в памяти может случиться, что некое поле, состоящее из нескольких байтов, пересечёт «естественную границу» слов в памяти. Некоторые модели процессоров не могут обращаться к данным в памяти, нарушающим границы машинных слов. Некоторые могут обращаться к невыровненным данным дольше, нежели к данным, находящимся внутри целого «машинного слова» в памяти.»*

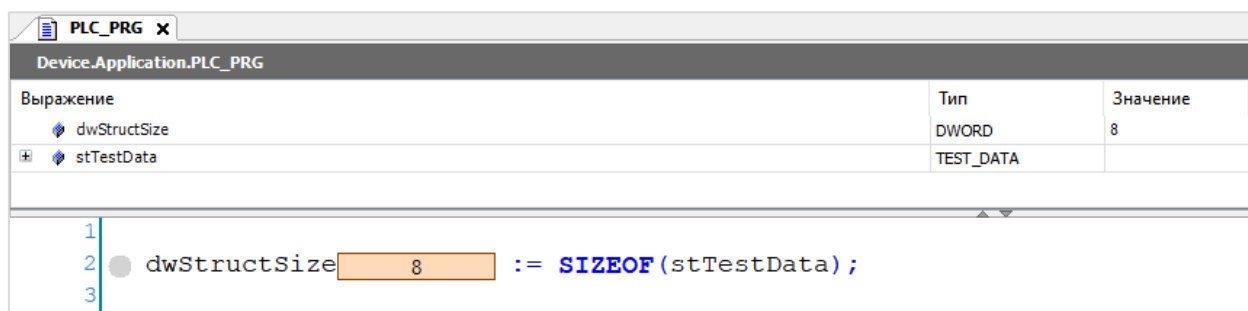


Рис. 1.1.13. Определение размера структуры позволяет увидеть влияние выравнивания

В некоторых случаях наличие выравнивания может создавать неудобства – например, вы считываете с устройства 5 байт данных и знаете, что один из них представляет собой переменную типа **BOOL**, а еще 4 – типа **REAL**. Соответственно, вам бы хотелось просто скопировать содержимое массива приемного буфера в память, выделенную под хранение экземпляра вашей структуры – но наличие выравнивания как будто бы делает это невозможным.

В таких случаях очень полезным является наличие в CODESYS V3.5 специального атрибута (прагмы) **pack\_mode** для управления выравниванием в структурах. Атрибут указывается в окне объявления структуры перед ее именем. Возможные значения атрибута – **0/1/2/4/8**. Значение характеризует кратность адресов памяти, по которым будут размещены переменные. Например, значение «1» означает, что кратность адресов памяти равна 1 – и, соответственно, все элементы структуры будут размещены в памяти последовательно без каких-либо разрывов. Более детальная информация об атрибуте с конкретными примерами доступна в [соответствующей статье](#) в справке CODESYS.

```

1 {attribute 'pack_mode' := '1' }
2 TYPE TEST_DATA :
3 STRUCT
4     xValue:      BOOL;
5     rValue:      REAL;
6 END_STRUCT
7 END_TYPE

```

Рис. 1.1.14. Установка атрибута **pack\_mode**

Device.Application.PLC_PRG				
Выражение	Тип	Значение	Подготовленное ...	Адрес
dwStructSize	DWORD	5		
stTestData	TEST_DATA			

```

1
2 dwStructSize 5 := SIZEOF(stTestData);
3

```

Рис. 1.1.15. Влияние атрибута **pack\_mode** на размер структуры

В справке CODESYS указано, что доступ к элементам экземпляра структуры с атрибутом **pack\_mode** является более ресурсоемким по сравнению с аналогичной структурой, не имеющей данного атрибута. Тем не менее, в ряде случаев использование атрибута позволяет упростить код приложения и повысить его производительность за счет того, что программисту не придется добавлять код преобразования исходного массива данных в элементы невыровненной структуры. В каждом конкретном случае решение об использовании атрибута должно приниматься разработчиком с учетом особенностей и требований решаемой задачи.

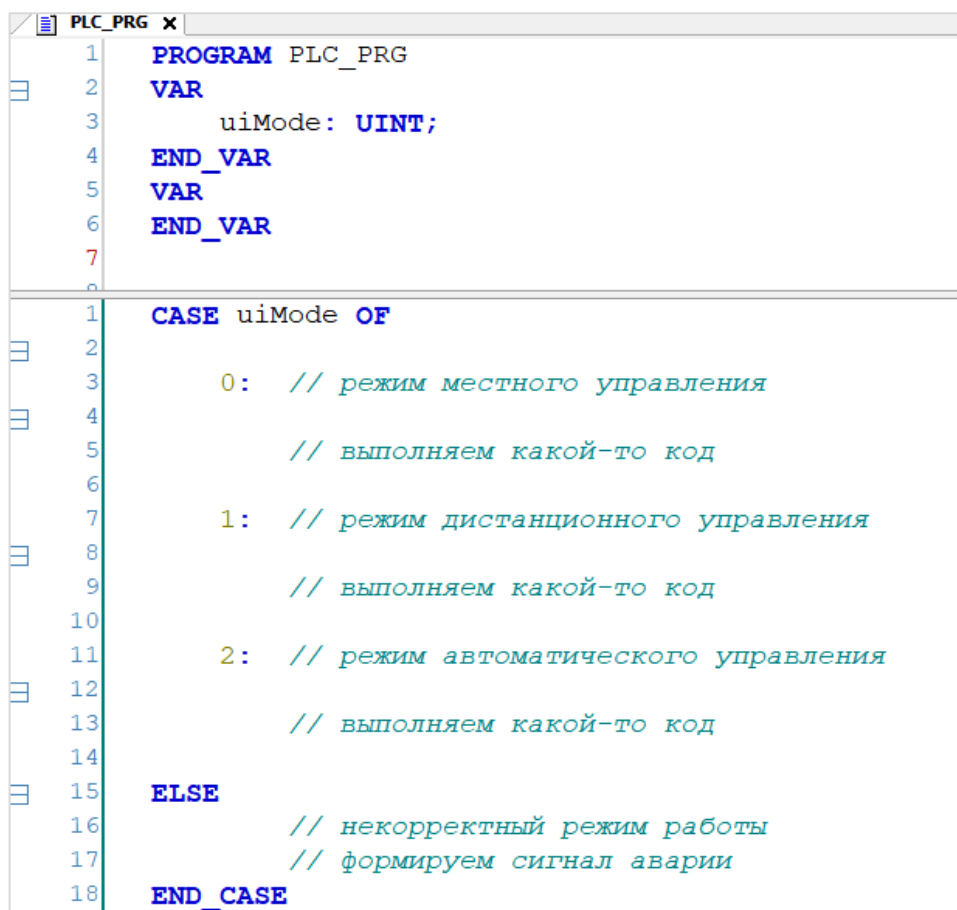
## 1.2. Перечисления

### 1.2.1. Общая информация

Перечисление – это пользовательский тип данных (DUT), который позволяет указать для конкретных значений целочисленной переменной символьные описания. Это повышает читабельность кода и позволяет избежать ошибок, связанных с присвоением переменным некорректных значений.

Рассмотрим их использование на конкретном примере.

Пусть управление технологическим процессом может производиться в трех режимах – в режиме местного управления (например – через визуализацию ПЛК), режиме дистанционного управления (через SCADA или другую систему верхнего уровня) и автоматическом режиме (согласно заложенным в ПЛК алгоритмам). Выбор режима производится с помощью изменения значения переменной ПЛК. Тогда в общем виде код приложения может выглядеть следующим образом:



```
PLC_PRG x
1  PROGRAM PLC_PRG
2  VAR
3      uiMode: UINT;
4  END_VAR
5  VAR
6  END_VAR
7
8
9
10
11  CASE uiMode OF
12
13      0: // режим местного управления
14          // выполняем какой-то код
15
16      1: // режим дистанционного управления
17          // выполняем какой-то код
18
19      2: // режим автоматического управления
20          // выполняем какой-то код
21
22  ELSE
23      // некорректный режим работы
24      // формируем сигнал аварии
25  END_CASE
```

Рис. 1.2.1. Пример кода обработки режима работы

Этот код не является читабельным – без комментариев невозможно понять, какое значение какому режиму работы соответствует. И даже наличие комментариев не является панацеей – во время разработки проекта соответствие значений и режимов может поменяться (например, в

системе верхнего уровня будут выбраны другие значения для режимов, и в проект ПЛК потребуется внести изменения – есть шанс, что комментарии при этом поправятся забудут, и они перестанут соответствовать действительности).

Поэтому более удобным вариантом является использование перечислений.

Создание перечислений происходит тем же образом, что и структур – необходимо нажать ПКМ на узел **Application** и выбрать команду **Добавление объекта – DUT**. В случае установки галочки **Поддержка списков текстов** вместе с перечислением будет создан список текстов – это упростит использование перечислений в визуализации (далее мы более подробно рассмотрим эту настройку).

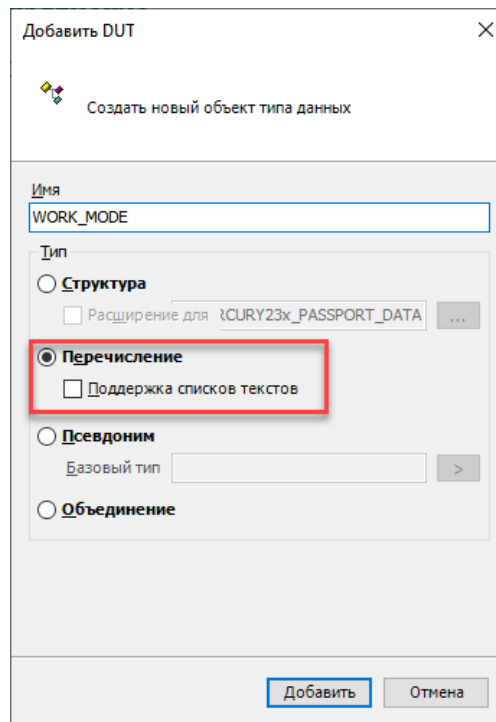


Рис. 1.2.2. Создание перечисления

Опишем в перечислении режимы работы:

```

1  {attribute 'qualified_only'}
2  {attribute 'strict'}
3  TYPE WORK_MODE :
4  (
5      // Местное управление
6      LOCAL := 0,
7      // Дистанционное управление
8      REMOTE := 1,
9      // Автоматическое управление
10     AUTO := 2
11 );
12 END_TYPE

```

Рис. 1.2.3. Объявление элементов перечисления

Теперь можно объявить в программе экземпляр перечисления и переписать наш код с рис. 1.2.1 следующим образом:

```
PLC_PRG x
1  PROGRAM PLC_PRG
2  VAR
3     eWorkMode: WORK_MODE;
4  END_VAR
5  VAR
6  END_VAR
7
8
9
10
11
12
13
14
15
16
17
18
1  CASE eWorkMode OF
2
3     WORK_MODE.LOCAL:
4
5         // выполняем какой-то код
6
7     WORK_MODE.REMOTE:
8
9         // выполняем какой-то код
10
11     WORK_MODE.AUTO:
12
13         // выполняем какой-то код
14
15 ELSE
16     // некорректный режим работы
17     // формируем сигнал аварии
18 END_CASE
```

Рис. 1.2.4. Код обработки режима работы с использованием перечисления

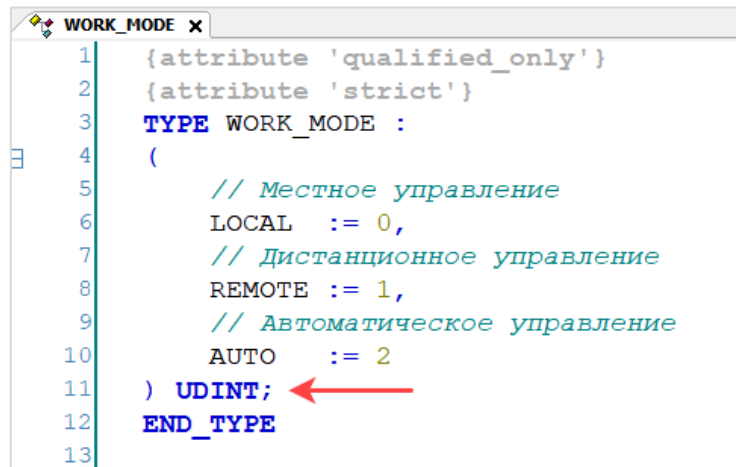
Использование перечисления сделало код более читабельным; кроме того, если значения, соответствующие режимам работы, потребуется изменить – то достаточно будет сделать это в самом перечислении (в исходном варианте нам бы пришлось искать в коде все места, где используется переменная `uiMode`, и редактировать их).

Типичные случаи использования перечислений:

- описание кодов ошибок;
- описание режимов работы оборудования;
- описание шагов машины состояний функциональных блоков (INIT, READ, WRITE, RESET и т.д.).

Собственно, перечисления подходят для описания любого конечного множества заранее известных целочисленных значений.

По умолчанию под хранение экземпляра перечисления выделяется 16 бит памяти. В случае необходимости явного определения размера – пользователь может при создании перечисления указать соответствующий ему тип:



```

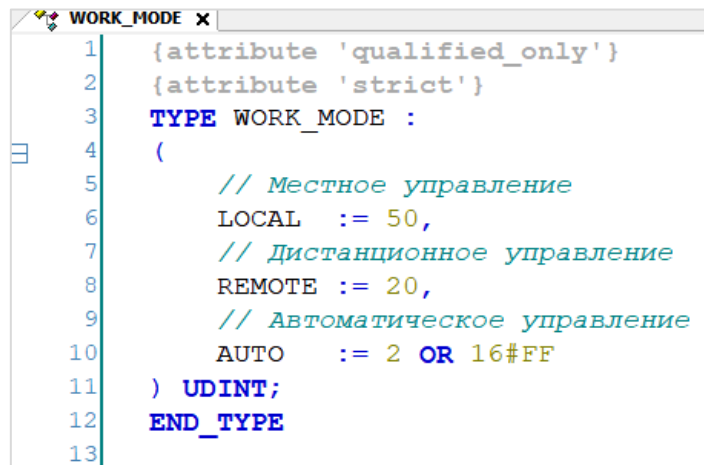
1  {attribute 'qualified_only'}
2  {attribute 'strict'}
3  TYPE WORK_MODE :
4  (
5      // Местное управление
6      LOCAL := 0,
7      // Дистанционное управление
8      REMOTE := 1,
9      // Автоматическое управление
10     AUTO := 2
11 ) UDINT;
12 END_TYPE
13

```

Рис. 1.2.5. Указание типа данных для перечисления

Как можно увидеть на рис. 1.2.4 – для использования элемента перечисления в качестве символьной константы требуется указывать имя перечисления<sup>2</sup>, поставить точку и выбрать из списка нужный элемент перечисления (т.е. название перечисления является пространством имен для элементов перечисления). Это, в частности, позволят создавать перечисления, включающие в себя элементы с одинаковыми именам (например: **ERROR\_RS.TIME\_OUT** и **ERROR\_TCP.TIME\_OUT**).

Числовые значения элементов перечисления могут задаваться произвольным образом и в произвольном порядке. Единственное очевидное ограничение – каждому элементу должно соответствовать уникальное в рамках перечисления числовое значение.



```

1  {attribute 'qualified_only'}
2  {attribute 'strict'}
3  TYPE WORK_MODE :
4  (
5      // Местное управление
6      LOCAL := 50,
7      // Дистанционное управление
8      REMOTE := 20,
9      // Автоматическое управление
10     AUTO := 2 OR 16#FF
11 ) UDINT;
12 END_TYPE
13

```

Рис. 1.2.6. Примеры значений элементов перечисления

Наследование перечислений не поддерживается. Вместо этого справка CODESYS рекомендует при необходимости выполнять «[маппирование](#)» (присвоение значений одному перечислению на основании значений другого) перечислений в коде программы.

<sup>2</sup> Это требование вызвано наличием атрибута **qualified\_only** – см. информацию в [п. 1.2.2](#).

Если для какого-либо из элементов числовое значение не задано в явном виде, то оно автоматически выбирается на единицу большим по сравнению с предыдущим заданным:

```

1  {attribute 'qualified_only'}
2  {attribute 'strict'}
3  TYPE WORK_MODE :
4  (
5      // Местное управление
6      LOCAL,      0
7      // Дистанционное управление
8      REMOTE := 10,
9      // Автоматическое управление
10     AUTO      11
11 ) UDINT;
12 END_TYPE
13

```

Рис. 1.2.7. Автоматическое распределение числовых значений для элементов перечисления

### 1.2.2. Атрибуты перечислений

Начиная с версии **CODESYS V3.5 SP7** при создании перечисления у него автоматически указано два атрибута – **qualified\_only** и **strict** (см, например, рис. 1.2.7).

Атрибут **qualified\_only** запрещает доступ к элементу перечисления без использования имени самого перечисления в качестве пространства имен. Поясним это на примере:

```

1  {attribute 'qualified_only'}
2  {attribute 'strict'}
3  TYPE WORK_MODE :
4  (
5      // Местное управление
6      LOCAL := 0,
7      // Дистанционное управление
8      REMOTE := 1,
9      // Автоматическое управление
10     AUTO  := 2
11 ) UDINT;
12 END_TYPE
13

```

```

1  eWorkMode := LOCAL;
2
3
4
5

```

Идентификатор 'LOCAL' не задан

Рис. 1.2.8. Влияние атрибута **qualified\_only**

```

1  WORK_MODE
2  {attribute 'strict'}
3  TYPE WORK_MODE :
4  (
5      // Местное управление
6      LOCAL := 0,
7      // Дистанционное управление
8      REMOTE := 1,
9      // Автоматическое управление
10     AUTO := 2
11 ) UDINT;
12 END_TYPE
13
1  eWorkMode := LOCAL;
2

```

Рис. 1.2.9. Без атрибута **qualified\_only** необязательно указывать имя перечисления для обращения к его элементам

Использование имен перечислений для доступа к их элементам повышает читаемость кода и позволяет использовать в разных перечислениях одинаковые имена элементов. Поэтому не рекомендуется из перечисления удалять атрибут **qualified\_only**.

Атрибут **strict** запрещает арифметические операции для работы с перечислениями. Если вы считаете, что в рамках вашей конкретной задачи такие операции имеют смысл – то атрибут можно удалить.

```

1  udiVar := 1 + WORK_MODE.REMOTE;
2
3

```

Арифметические действия недопустимы для строгих перечислений 'WORK\_MODE'

Рис. 1.2.10. Влияние атрибута **strict** (1)

Атрибут **strict** не запрещает присваивание экземпляру перечисления числовых значений, но добавляет ограничения на такие значения – они должны принадлежать набору значений, указанных в перечислении.

```

3  eWorkMode := 2;
4
3  eWorkMode := 5;
4
5

```

'5' - неподходящее значение для типа ENUM 'WORK\_MODE'

Рис. 1.2.11. Влияние атрибута **strict** (2)

В некоторых ситуациях удобно иметь возможность присвоить экземпляру перечисления произвольное число. Например, вы разрабатываете функциональный блок опроса некоего устройства и описываете в перечислении его коды команд (READ\_POWER := 16#F1, READ\_TIME := 16#F2 и т.д.), но допускаете, что производитель может добавить в следующих версиях прошивки новые команды, о которых вам в данный момент ничего не известно. Соответственно, в этом случае

было бы удобно, если бы пользователь вашего блока мог ввести вместо имени команды из перечисления произвольный код.

Для обеспечения такой возможности атрибут `strict` следует удалить.

```

1 {attribute 'qualified only'}
2
3 TYPE WORK_MODE :
4 (
5     // Местное управление
6     LOCAL := 0,
7     // Дистанционное управление
8     REMOTE := 1,
9     // Автоматическое управление
10    AUTO := 2
11 ) UDINT;
12 END_TYPE
13
14 eWorkMode := 5;

```

Рис. 1.2.12. Без атрибута `strict` можно присвоить экземпляру перечисления произвольное целочисленное значение

Начиная с версии **CODESYS V3.5 SP11** перечисления поддерживают атрибут `to_string`. Этот атрибут влияет на то, как работает оператор `TO_STRING` при преобразовании значения перечисления в строку. Без использования атрибута – строка будет содержать строковое представление числового значения перечисления. С использованием атрибута – строка будет содержать символьное обозначение элемента перечисления.

```

1 {attribute 'qualified_only'}
2 {attribute 'strict'}
3
4 TYPE WORK_MODE :
5 (
6     // Местное управление
7     LOCAL := 0,
8     // Дистанционное управление
9     REMOTE := 1,
10    // Автоматическое управление
11    AUTO := 2
12 ) UDINT;
13 END_TYPE
14
15 eWorkMode := WORK_MODE.REMOTE;
16 sWorkMode := TO_STRING(eWorkMode);

```

Рис. 1.2.13. Конвертация перечисления в строку без атрибута `to_string`

```

1 {attribute 'qualified_only'}
2 {attribute 'strict'}
3 {attribute 'to_string'}
4 TYPE WORK_MODE :
5 (
6     // Местное управление
7     LOCAL := 0,
8     // Дистанционное управление
9     REMOTE := 1,
10    // Автоматическое управление
11    AUTO := 2
12 ) UDINT;
13 END_TYPE
14
15 eWorkMode := WORK_MODE.REMOTE;
16 sWorkMode := TO_STRING(eWorkMode);

```

Рис. 1.2.14. Конвертация перечисления в строку с атрибутом `to_string`

### 1.2.3. Использование перечислений со встроенными списками текстов

В конце прошлого пункта мы рассмотрели, как получить в программе строковое представление символьного обозначения элемента перечисления – например, для отображения в визуализации. Но такие символьные обозначения могут содержать только символы латиницы – что, соответственно, делает неудобным их использование, если интерфейс визуализации разрабатывается на другом языке.

Поэтому для облегчения этой ситуации в **CODESYS V3.5 SP9** появилась поддержка списков текстов для перечислений. Приведем пример их использования для работы с элементом визуализации **Выпадающий список** (в русской локализации он назван **Комбинированное окно – Целочисленный**).

1. Добавим в проект перечисление **COLOR** с поддержкой списка текстов (**ПКМ** на узел **Application** – **Добавить объект** – **DUT**, выберем тип **Перечисление**, установим галочку **Поддержка списка текстов**).

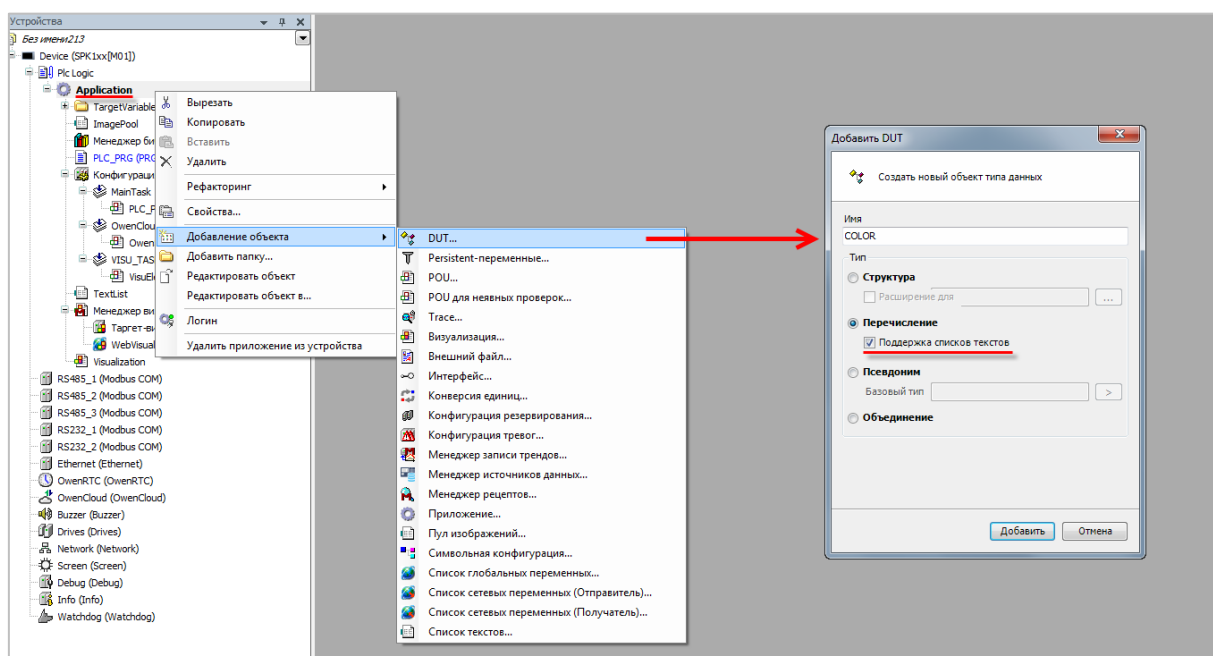


Рис. 1.2.15. Добавление в проект перечисления с поддержкой списка текстов

```

1  {attribute 'qualified_only'}
2  {attribute 'strict'}
3  TYPE COLOR :
4  (
5      RED    := 0,
6      GREEN  := 1,
7      BLUE   := 2
8  );
9  END_TYPE
10

```

Рис. 1.2.16. Содержимое перечисления **COLOR**

2. Нажмем кнопку **Локализация**, чтобы переключить перечисление в режим списка текстов. Добавим язык 'ru' и зададим тексты для элементов перечисления.

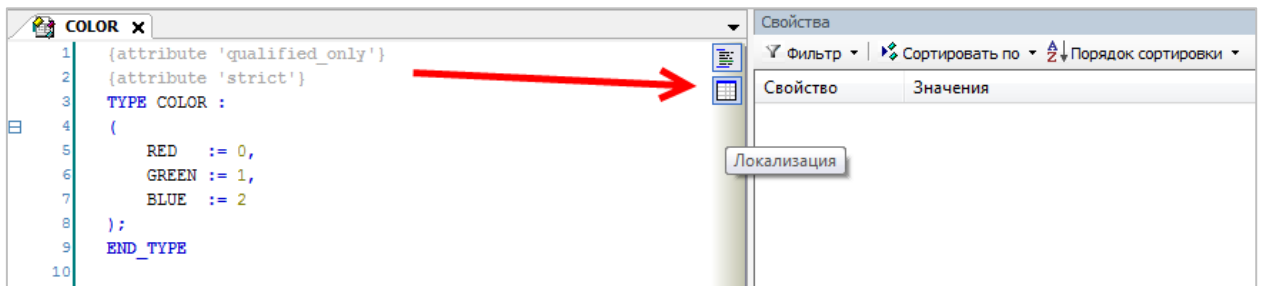


Рис. 1.2.17 Переключение перечисления в режим списка текстов

Member	Value	ru
RED	0	Красный
GREEN	1	Зеленый
BLUE	2	Синий

Рис. 1.2.18. Содержимое списка текстов перечисления

3. Объявим в программе **PLC\_PRG** переменную **eColor** типа **COLOR**.

4. В менеджере визуализации выберем язык 'ru'.

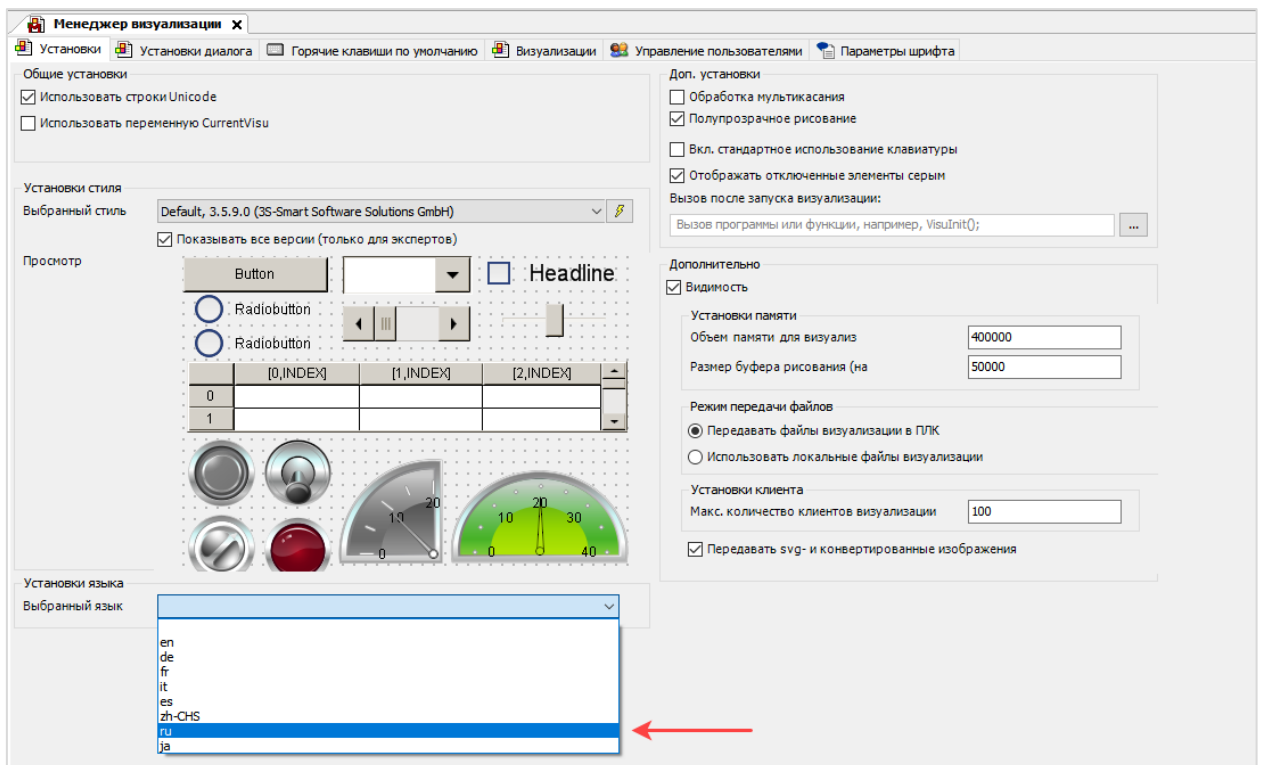


Рис. 1.2.19. Выбор языка в менеджере визуализации

5. В визуализации проекта добавим элемент **Комбинированное окно – Целочисленный** и привяжите к его параметру **Переменная** объявленную переменную. **Обратите внимание**, что после названия переменной автоматически отобразится название перечисления (точнее – название списка текстов, одноименного перечислению).

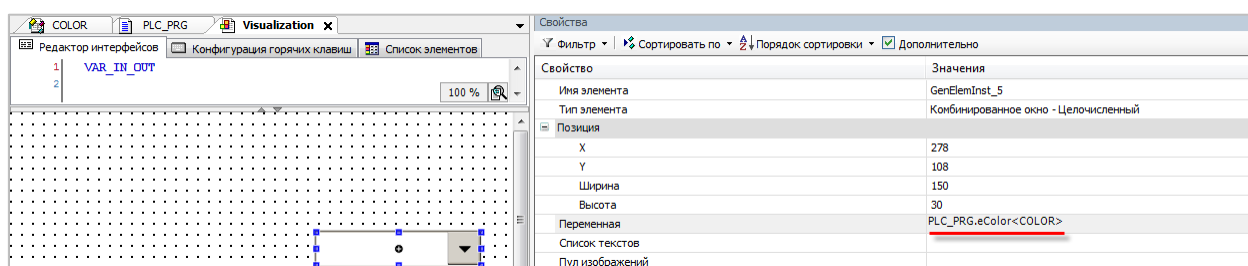


Рис. 1.2.20. Привязка экземпляра перечисления к элементу **Комбинированное окно – Целочисленный**

6. Загрузите проект в контроллер или запустите его в эмуляции. Выберите значение с помощью выпадающего списка.

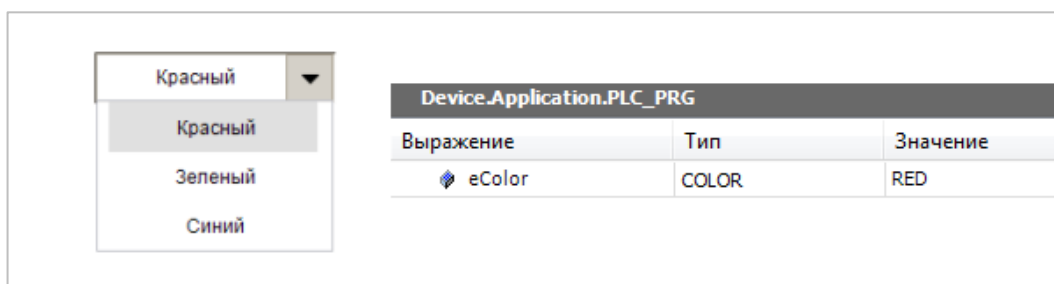


Рис. 1.2.21. Работа с элементом в визуализации



#### ПРИМЕЧАНИЕ

В версиях **CODESYS V3.5 SP15** и **SP16** данный функционал работает некорректно – в визуализации в выпадающем списке отображаются целочисленные значения, диапазон которых не ограничен диапазоном перечисления. Баг исправлен в версии **CODESYS V3.5 SP17**.

#### 1.2.4. Локальные перечисления

Интересной недокументированной (на момент версии **V3.5 SP17**) возможностью CODESYS является объявление локальных перечислений. Эти перечисления создаются не при добавлении в проект объекта **DUT** типа **Перечисление**, а прямо в области объявления переменных. Объявление локальных перечислений поддерживается в POU (функциях, функциональных блоках и программах) и списках локальных переменных, но не поддерживается в методах.

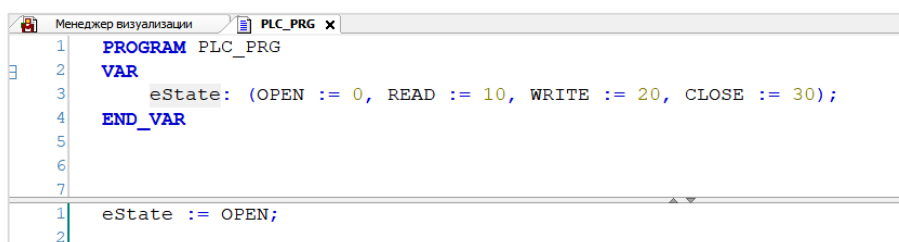


Рис. 1.2.22. Объявление локального перечисления

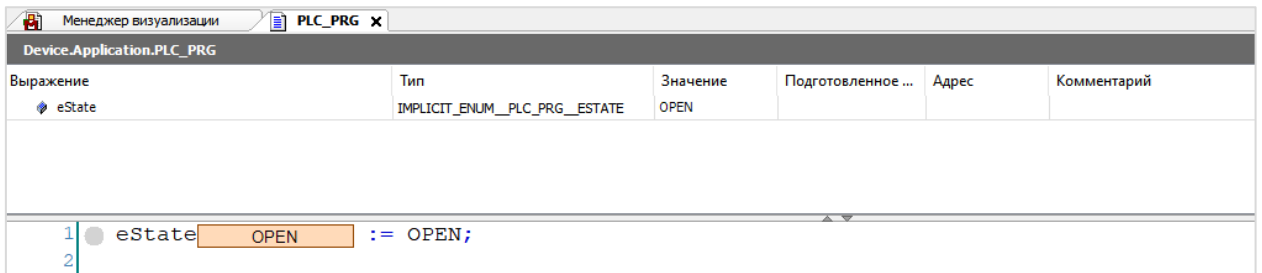


Рис. 1.2.23. Использование локального перечисления в коде программы

### 1.3. Объединения

Объединение – это вариант структуры, в которой все элементы размещены в одной и той же области памяти. Соответственно, изменение любого из элементов приводит к изменению всех остальных. Объединения удобны в тех случаях, когда нужно использовать в программе различные варианты представления одних и тех же данных. Типичный пример – работа со стандартным компонентом **Modbus Slave**, к каналам которого можно привязать только переменные типа **WORD**. Поэтому, например, для привязки к слэйву переменной типа **REAL** требуется представить ее в виде массива из двух WORD-переменных.

Как и [структуры](#), объединения поддерживают:

- объявление [переменных типа BIT](#);
- [наследование](#) (но ключевое слово **EXTENDS** в данном случае придется добавлять вручную, см. [пример](#));
- [атрибут pack\\_mode](#) для управления выравниванием.

Объединение является специфическим функционалом CODESYS, не описанном в стандарте МЭК 61131-3.

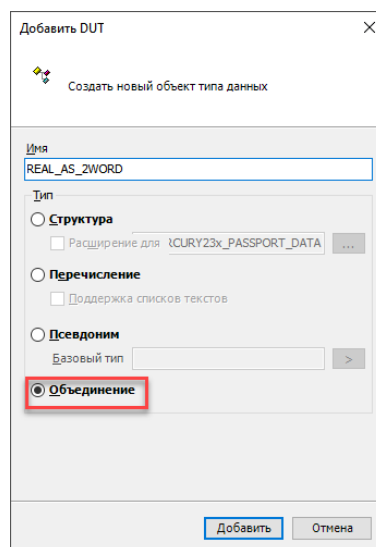


Рис. 1.3.1. Создание объединения



## 1.4. Библиотека CAA Memory

В прошлых пунктах мы рассмотрели типы переменных, которые позволяют представить обработанные данные в удобной пользователю форме. Вспомним типичную задачу, которую мы описали во [введении](#): преобразовать набор байт, полученных от некоего устройства, к нужному набору переменных. Иными словами – нам нужно «переложить» данные из области памяти нашего массива байт в область памяти экземпляра нашей структуры. Один из самых простых способов сделать это – использовать функцию **MemMove** из библиотеки **CAA Memory**, которая входит в дистрибутив CODESYS.

Функция производит копирование **uiNumberOfBytes** байт из области памяти, размещенной по указателю **pSource**, в область памяти, размещенную по указателю **pDestination**. Области памяти могут пересекаться.

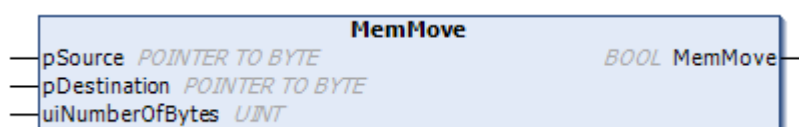


Рис. 1.4.1. Внешний вид функции **MemMove** на языке CFC

В значительном числе случаев при работе с функцией удобно использовать оператор **ADR** для получения адресов памяти и **SIZEOF** для определения числа копируемых байт (разумеется, это справедливо только для тех случаев, когда нужно скопировать объект целиком).

Пример вызова функции для копирования содержимого массива **abyData** в экземпляр структуры **stData**:

```

PLC_PRG x
1  PROGRAM PLC_PRG
2  VAR
3      abyData:      ARRAY [0..7] OF BYTE;
4      stData:      TEST_DATA;
5  END_VAR
6
1
2  MEM.MemMove (ADR (abyData), ADR (stData), SIZEOF (stData) );
3

```

Рис. 1.4.2. Пример вызова функции **MemMove** на языке ST

Подразумевается, что переменные **abyData** и **stData** имеют одинаковый размер, равный 8 байтам. На вход **uiNumberOfBytes** мы передаем размер переменной **stData**, потому что именно в эту переменную происходит копирование данных; если бы мы указали в качестве числа копируемых байт размер **abyData**, то в случае ошибки (например, при случайном объявлении массива с размером > 8 байт) при копировании данных была бы перезаписана не только память, в которой хранится переменная **stData**, но и следующие за ней байты памяти (в которых хранятся совершенно другие данные). Это типичный пример [переполнения буфера](#). В худшем случае подобная ситуация приведет к [ошибке сегментации памяти](#).

Если нам требуется скопировать только фрагмент исходного объекта (например, первые три байта массива) – то значение на входе **uiNumberOfBytes** можно указать в виде константы или литерала:

```
2 MEM.MemMove (ADR (abyData) , ADR (stData) , 3) ;
```

Рис. 1.4.3. Копирование части объекта

Еще одна крайне полезная функция из библиотеки CAA Memory – это **MemFill**. Она позволяет «заполнить» область памяти по указателю **pMemoryBlock** и размером **uLength** байтов значением **byFillValue**.

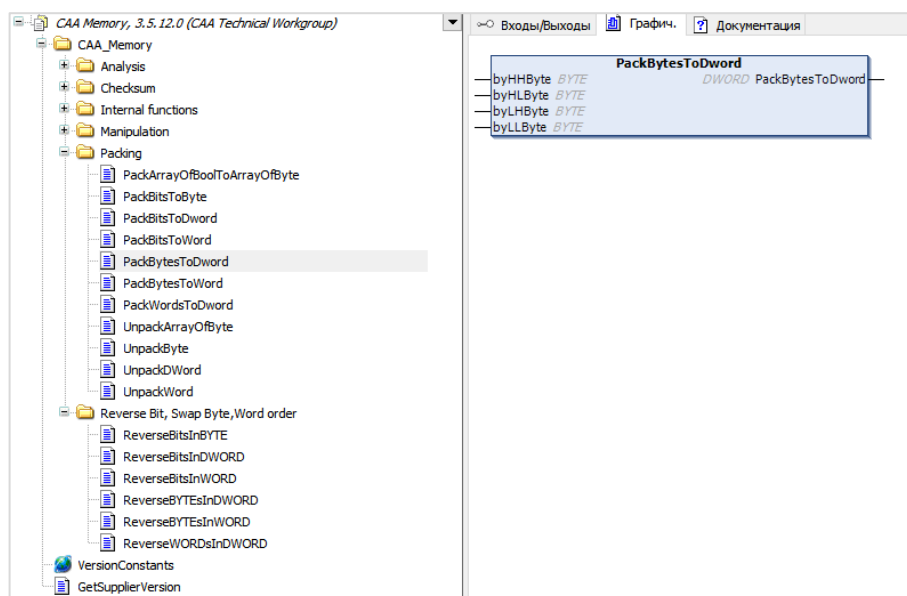
Рис. 1.4.5. Внешний вид функции **MemFill** на языке CFC

Типичный случай, в котором используется эта функция – очистка буфера.

```
2 // обнуляем все элементы массива
3 MEM.MemFill (ADR (abyData) , sizeof (abyData) , 0) ;
```

Рис. 1.4.6. Пример вызова функции **MemFill** на языке ST

Библиотека **CAA Memory** включает в себя и другие полезные функции – например, функции для «упаковки» и «распаковки» значений (они могут быть полезны, когда требуется «разобрать» переменную на байты или наоборот собрать ее из них) и функции изменения порядка бит/байт/WORD (поскольку, например, порядок байт на ПЛК, для которого вы пишете программу, и устройства, от которого вы получаете данные, может отличаться).

Рис. 1.4.7. Состав библиотеки **CAA Memory**

## 1.5. Библиотека OwenCommunication

Мы закончили прошлый пункт на фразе о том, что задача изменения порядка байт в принятых данных является довольно типичной. Возможно, вы подумали о том, что было бы удобно иметь возможность скопировать данные и изменить в них порядок байт с помощью одной функции. Такая функция есть в библиотеке **OwenCommunication**, которая доступна для загрузки [в разделе CODESYS V3 на сайте OVEN](#) – она называется **SWAP\_DATA**.

Функция **SWAP\_DATA** основана на функции **MemMove** и имеет 3 дополнительных входа:

- **xSwapBytes** – флаг перестановки байт;
- **xSwapWord** – флаг перестановки WORD (регистров);
- **xReverseByteOrder** – флаг инверсии порядка байт (если этот флаг установлен, то значения предыдущих двух входов не обрабатываются).

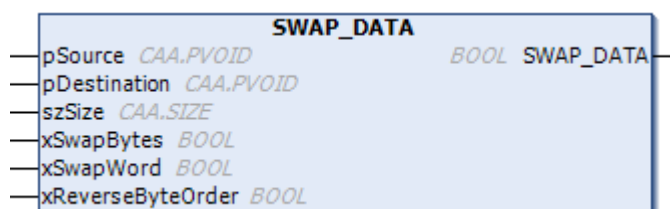


Рис. 1.5.1. Внешний вид функции **SWAP\_DATA** на языке CFC

Проще всего продемонстрировать принцип работы функции на примере конвертации одного и того же значения (рисунки хорошо масштабируются):

```

1  PROGRAM PLC_PRG
2  VAR
3      lwInput:    LWORD := 16#AABBCCDDEEFF0011;
4      lwOutput:   LWORD;
5  END_VAR

```

Рис. 1.5.2. Исходные данные

The screenshot shows the PLC\_PRG editor with the following code:

```

1  OCL.SWAP_DATA (ADR(lwInput 16#AABBCCDDEEFF0011), ADR(lwOutput 16#BBAADCCFFEE1100), SIZEOF(lwOutput 16#BBAADCCFFEE1100), TRUE, FALSE, FALSE);
2

```

A table below the code shows the values of the variables:

Выражение	Тип	Значение	Подготовленное ...	Адрес	Комментарий
lwInput	LWORD	16#AABBCCDDEEFF0011			
lwOutput	LWORD	16#BBAADCCFFEE1100			

Рис. 1.5.3. Копирование с перестановкой байт

The screenshot shows the PLC\_PRG editor with the following code:

```

1  OCL.SWAP_DATA (ADR(lwInput 16#AABBCCDDEEFF0011), ADR(lwOutput 16#CCDDAABB0011EEFF), SIZEOF(lwOutput 16#CCDDAABB0011EEFF), FALSE, TRUE, FALSE);
2

```

A table below the code shows the updated values:

Выражение	Тип	Значение	Подготовленное ...	Адрес	Комментарий
lwInput	LWORD	16#AABBCCDDEEFF0011			
lwOutput	LWORD	16#CCDDAABB0011EEFF			

Рис. 1.5.4. Копирование с перестановкой WORD

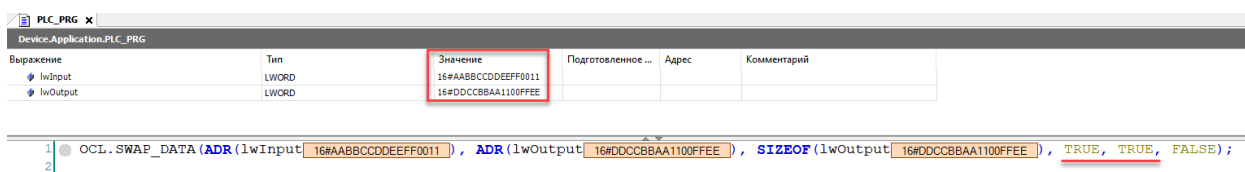


Рис. 1.5.5. Копирование с перестановкой байт и WORD

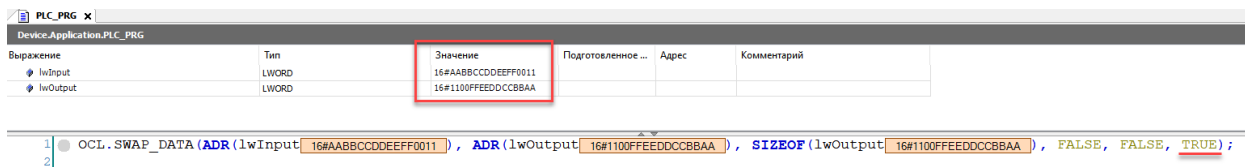
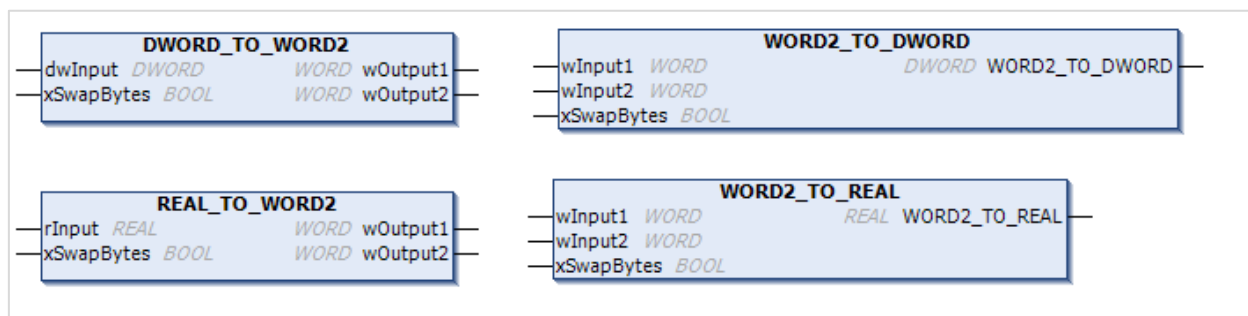


Рис. 1.5.6. Копирование с инверсией порядка байт

Результат вызовов функций с различными значениями аргументов в табличном виде:

Установленные флаги	Исходное значение	Преобразованное значение
xSwapBytes := TRUE	16#AABBCCDDEEFF0011	16#BBAA DDCC FFEE 1100
xSwapWord := TRUE	16#AABBCCDDEEFF0011	16#CCDDAABB0011EEFF
xSwapBytes := TRUE xSwapWord := TRUE	16#AABBCCDDEEFF0011	16#DDCCBBA1100FFEE
xReverseByteOrder := TRUE	16#AABBCCDDEEFF0011	16#1100FFEEDDCCBBA

Также библиотека включает в себя функции для «сборки» переменных типа **DWORD** и **REAL** из двух переменных типа **WORD** с возможностью изменения порядка байт и аналогичные функции «разборки». Функции полезны при работе с протоколом Modbus, спецификация которого не описывает никакие типы параметров – все данные передаются в виде наборов регистров (регистр представляет собой 16-битный блок памяти, которому в CODESYS соответствует переменная типа **WORD**).

Рис. 1.5.7. Внешний вид других функций библиотеки **OwenCommunication**

## 1.6. Функции для работы с BCD

В некоторых протоколах обмена используется [двоично-десятичный формат](#) (BCD) данных для представления параметров. Например, в [протоколе счетчиков Пульсар](#) адрес счетчика указывается именно в таком формате: для счетчика с адресом 12345678 первые 4 байта запроса, в которых и указывается его адрес, должны иметь значения **0x12 0x34 0x56 0x78**.

Для конвертации значений из десятичного формата в формат BCD и обратно можно воспользоваться функциями из библиотеки **Util**:

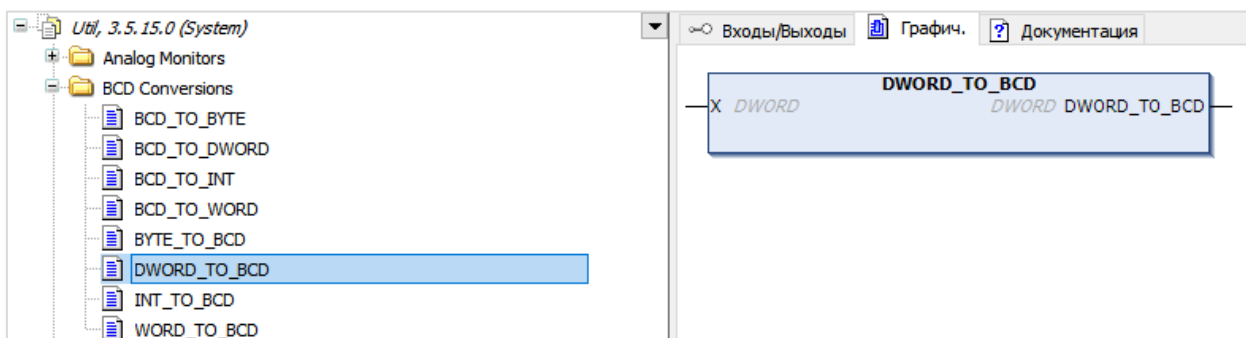


Рис. 1.6.1. Внешний вид функции **DWORD\_TO\_BCD** на языке CFC

```

1  dwPulsarBcdAddr 16#12345678 := UTIL.DWORD_TO_BCD(udiPulsarAddr 12345678 );
2

```

Рис. 1.6.2. Пример вызова функции **DWORD\_TO\_BCD** на языке ST

## 2. Примеры конвертации данных

### 2.1. «2 WORD в REAL» и т.д.

Мы уже упоминали, что спецификация протокола Modbus не описывает какие-либо типы данных – информация всегда передается в виде регистров (16-битных блоков памяти; в рамках стандарта МЭК 61131-3 регистру соответствует тип данных **WORD**). Если разработчик среды программирования не предусмотрел в своем коммуникационном драйвере возможность привязки к каналам опроса переменных произвольных типов – то пользователю придется самостоятельно произвести конвертацию массива **WORD** в переменную нужного типа (или наоборот преобразовать переменную в массив **WORD** для последующей передачи по Modbus).

Стандартный драйвер Modbus в среде CODESYS V3.5 поддерживает привязку к каналам опроса только переменных типа **WORD** (для регистров) и **BOOL** (для доступа к coils, discrete inputs и отдельным битам регистров). Соответственно, начинающий пользователь, который пытается считать или записать значение типа **REAL** (или другого типа, отличного от **BOOL** и **WORD**), сталкивается с проблемой.

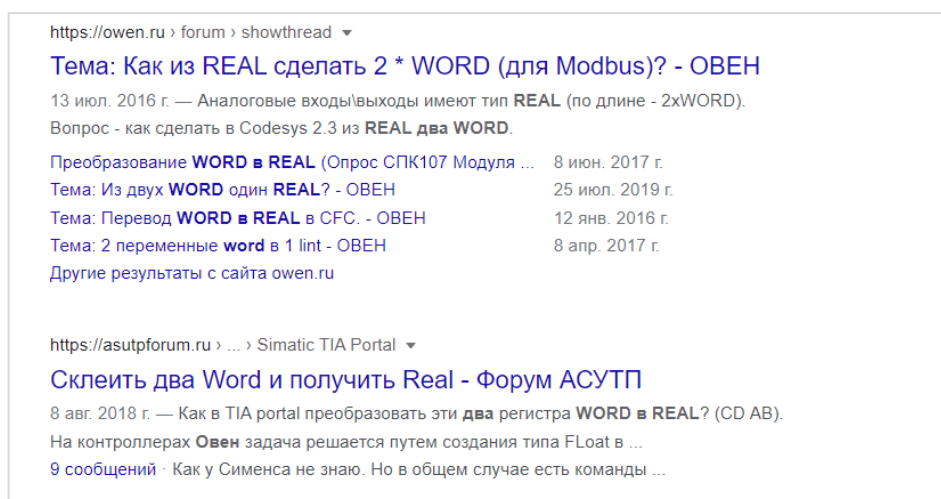


Рис. 2.1.1. Формулировки проблем пользователей

Надо отметить, что в баг-трекере CODESYS зафиксировано пожелание по доработке драйвера для возможности привязки к каналам опроса переменных любых типов, но будет ли оно реализовано – в данный момент неизвестно. Поэтому разберемся, как произвести подобную конвертацию данных своими силами.

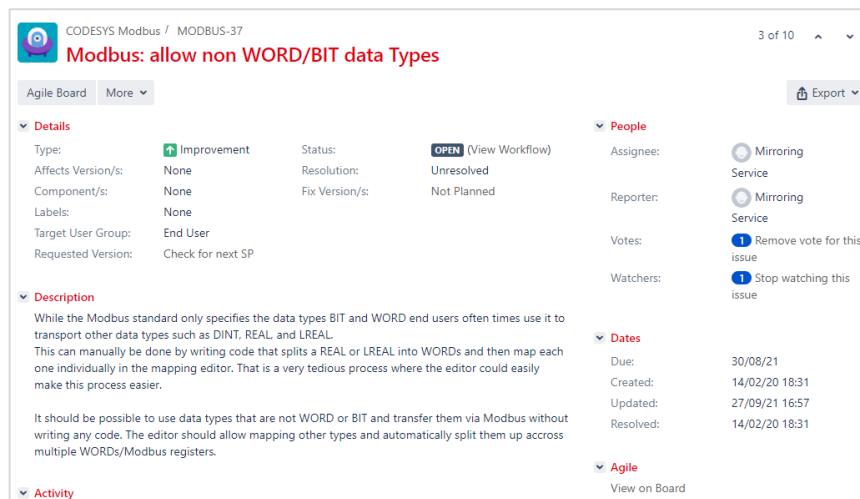
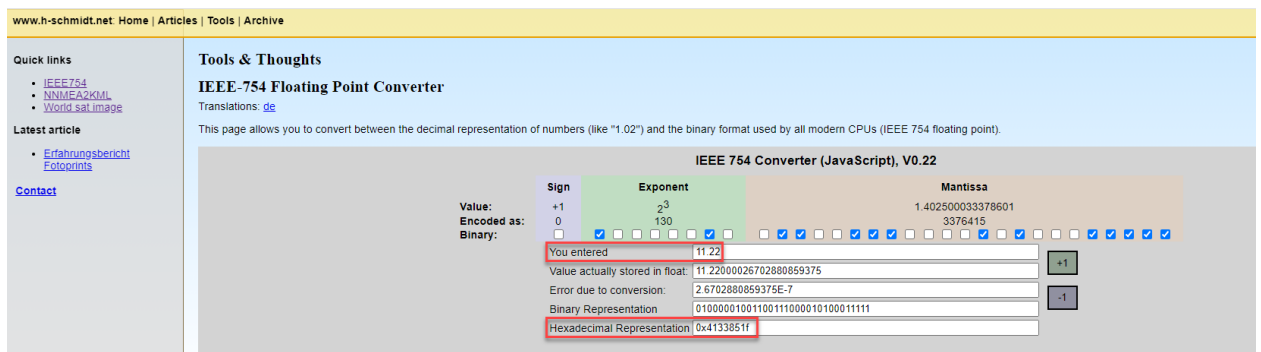


Рис. 2.1.2. Пожелание в баг-трекере CODESYS по доработке драйвера Modbus

Для начала выберем какое-то число с плавающей точкой (в рамках примера мы будем рассматривать число **11.22**) и с помощью [онлайн-конвертера](#) посмотрим, как оно выглядит в «сыром» виде:

Рис. 2.1.3. Значение 11.22 в бинарном виде (по стандарту [IEEE-754](#))

Объявим в программе массив из двух переменных типа **WORD** и присвоим ему это значение. Над порядком записи значений задумываться не будем; вообще, по сети данные могут передаваться с разным порядком байт и регистров – это зависит от особенностей устройства, которое их передает.

```

1 PROGRAM PLC_PRG
2 VAR
3     awData: ARRAY [0..1] OF WORD := [16#4133, 16#851F];
4     rData: REAL; // то, что хотим получить
5 END_VAR

```

Рис. 2.1.4. Объявление переменных

Теперь рассмотрим, как преобразовать этот массив в переменную типа **REAL**, с помощью средств, рассмотренных в [главе 2](#).

### 2.1.1. Функции библиотеки OwenCommunication

Самый простой способ решения нашей задачи – воспользоваться функциями конвертации из [библиотеки OwenCommunication](#) (ее, соответственно, потребуется добавить в наш проект). Подходящая нам функция называется **WORD2\_TO\_REAL**. На вход она принимает два значения типа **WORD**, а возвращает значение типа **REAL** – это именно то, что нам нужно. Кроме того, функция имеет дополнительный вход **xSwapBytes** – он нужен для тех случаев, когда при конвертации надо поменять порядок байт в обрабатываемых **WORD**. Мы пока не знаем, нужно ли это в нашем случае – поэтому присвоим входу значение **FALSE**.

```
rData := OCL.WORD2_TO_REAL(awData[0], awData[1], FALSE);
```

```
1 rData := OCL.WORD2_TO_REAL(awData[0], awData[1], FALSE);
2
```

Рис. 2.1.5. Вызов функции **WORD2\_TO\_REAL**

В нашем случае всё сразу получилось верно. Но что, если бы данные пришли по сети с другим порядком регистров? Это не проблема – мы бы просто изменили порядок передачи элементов массива на вход функции.

```
1 PROGRAM PLC_PRG
2 VAR
3   awData: ARRAY [0..1] OF WORD := [16#851F, 16#4133];
4   rData: REAL; // то, что хотим получить
5 END_VAR
6
7 rData := OCL.WORD2_TO_REAL(awData[1], awData[0], FALSE);
8
9 rData := OCL.WORD2_TO_REAL(awData[1], awData[0], FALSE);
10
```

Рис. 2.1.6. Вызов функции **WORD2\_TO\_REAL** с другим порядком передачи аргументов

А если бы кроме порядка регистров отличался бы и порядок байт – то тогда нам бы помог вход функции **xSwapBytes**, который мы упоминали выше:

```
1 PROGRAM PLC_PRG
2 VAR
3   awData: ARRAY [0..1] OF WORD := [16#1F85, 16#3341];
4   rData: REAL; // то, что хотим получить
5 END_VAR
6
7 rData := OCL.WORD2_TO_REAL(awData[1], awData[0], TRUE);
8
9 rData := OCL.WORD2_TO_REAL(awData[1], awData[0], TRUE);
10
```

Рис. 2.1.7. Вызов функции **WORD2\_TO\_REAL** с перестановкой байт

### 2.1.2. Функции библиотеки CAA Memory

Решим ту же самую задачу с помощью функции [MemMove из библиотеки CAA Memory](#) (которую надо предварительно добавить в проекте). Вернем наш массив к исходному виду (как на рис. 2.1.4) и вызовем нашу функцию:

```
MEM.MemMove (ADR (awData), ADR (rData), SIZEOF (rData) );
```

```
2 MEM.MemMove (ADR (awData), ADR (rData -7.49E-36 ), SIZEOF (rData -7.49E-36 ) );
3
```

Рис. 2.1.8. Вызов функции **MemMove**

Кажется, что-то пошло не так. Скорее всего, проблема в порядке регистров или байт – давайте попробуем их поменять. Начнем с регистров. Понятно, что мы можем просто изменить порядок значений при объявлении массива – но в реальной жизни вы не задаете эти значения вручную; они приходят по сети от другого устройства с тем порядком, который есть. Поэтому сделаем крайне простой и очевидный ход – объявим еще один массив и скопируем в него наш массив, переставив элементы местами:

```
1 PROGRAM PLC_PRG
2 VAR
3   awData:   ARRAY [0..1] OF WORD := [16#4133, 16#851F];
4   awBuffer: ARRAY [0..1] OF WORD;
5   rData:    REAL; // то, что хотим получить
6 END_VAR
```

Рис. 2.1.9. Объявление промежуточного массива для перестановки регистров

```
awBuffer[0] := awData[1];
awBuffer[1] := awData[0];

MEM.MemMove (ADR (awBuffer), ADR (rData), SIZEOF (rData) );
```

```
2 awBuffer[0] 34079 := awData[1] 34079;
3 awBuffer[1] 16691 := awData[0] 16691;
4
5 MEM.MemMove (ADR (awBuffer), ADR (rData 11.2 ), SIZEOF (rData 11.2 ) );
6 11.22
```

Рис. 2.1.10. Вызов функции **MemMove** после перестановки регистров

Мы получили именно то, что ожидали. Теперь, как и в прошлом примере, предположим, что порядок байт в исходном массиве у нас другой. Тогда при копировании значений в промежуточный массив мы можем поменять его с помощью функции **ReverseBYTESInWORD** (она тоже расположена в библиотеке **CAA Memory**).

```

1  PROGRAM PLC_PRG
2  VAR
3     awData:   ARRAY [0..1] OF WORD := [16#3341, 16#1F85];
4     awBuffer: ARRAY [0..1] OF WORD;
5     rData:   REAL; // то, что хотим получить
6  END_VAR
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

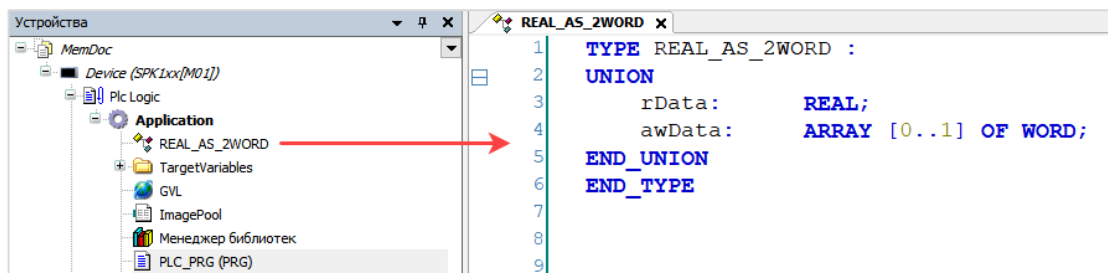
```

Рис. 2.1.11. Вызов функции **MemMove** после перестановки байт функцией **ReverseBYTESInWORD**

Как мы видим, решение нашей задачи с помощью функций библиотеки **CAA Memory** может потребовать большего количества строк кода по сравнению с [OwenCommunication](#). Кроме того, в состав **OwenCommunication** входит функция **SWAP\_DATA** – она представляет собой «обертку» над **MemMove** и позволяет при копировании сразу поменять порядок регистров и байт (мы рассказывали про эту функцию в [п. 1.5](#)). С другой стороны, библиотека **CAA Memory** входит в дистрибутив CODESYS – а **OwenCommunication** придется установить отдельно.

### 2.1.3. Объединения

Опять вернем наш массив к исходному виду (как на рис 2.1.4) и решим всё ту же задачу с помощью объединения.



```

1  TYPE REAL_AS_2WORD :
2  UNION
3     rData:   REAL;
4     awData:  ARRAY [0..1] OF WORD;
5  END_UNION
6  END_TYPE
7
8
9

```

Рис. 2.1.12. Создание объединения

В программе объявим экземпляр нашего объединения. В его элемент-массив скопируем наш исходный массив **awData**, а значение элемента **rData** присвоим одноименной переменной нашей программы.

```

1  PROGRAM PLC_PRG
2  VAR
3      awData:      ARRAY [0..1] OF WORD := [16#4133, 16#851F];
4
5      uData:      REAL_AS_2WORD;
6
7      rData:      REAL; // то, что хотим получить
8  END_VAR
9
10
1
2  uData.awData := awData;
3  rData      := uData.rData;
4
2  ● uData.awData := awData;
3  ● rData -7.49E-36 := uData.rData -7.49E-36 ;
4

```

Рис. 2.1.13. Использование объединения

Очень похоже на картину, которую мы наблюдали на рис. 2.1.8, не правда ли? Как мы уже знаем – проблема связана с порядком регистров. Способ ее решения нам тоже известен:

```

uData.awData[0] := awData[1];
uData.awData[1] := awData[0];

rData := uData.rData;

```

```

2  ● uData.awData[0] 16#851F := awData[1] 16#851F ;
3  ● uData.awData[1] 16#4133 := awData[0] 16#4133 ;
4
5  ● rData 11.2 := uData.rData 11.2 ;
6
11.22

```

Рис. 2.1.14. Перестановка регистров перед использованием объединения

Мы снова получили нужный нам результат. Причем если бы данные пришли с подходящим нам порядком регистров – то можно было бы вообще обойтись без кода и промежуточных переменных – мы могли бы, например, привязать переменную **uData.awData** к каналу Modbus, а **uData.rData** использовать в коде программы. В результате переменные **awData**, **rData** и связанный с ними код вообще были бы не нужны.

С другой стороны, если бы нам потребовалось поменять порядок байт – то пришлось бы использовать функции [библиотеки CAA Memory](#) или какие-то другие средства – но, так или иначе, код стал бы более громоздким.

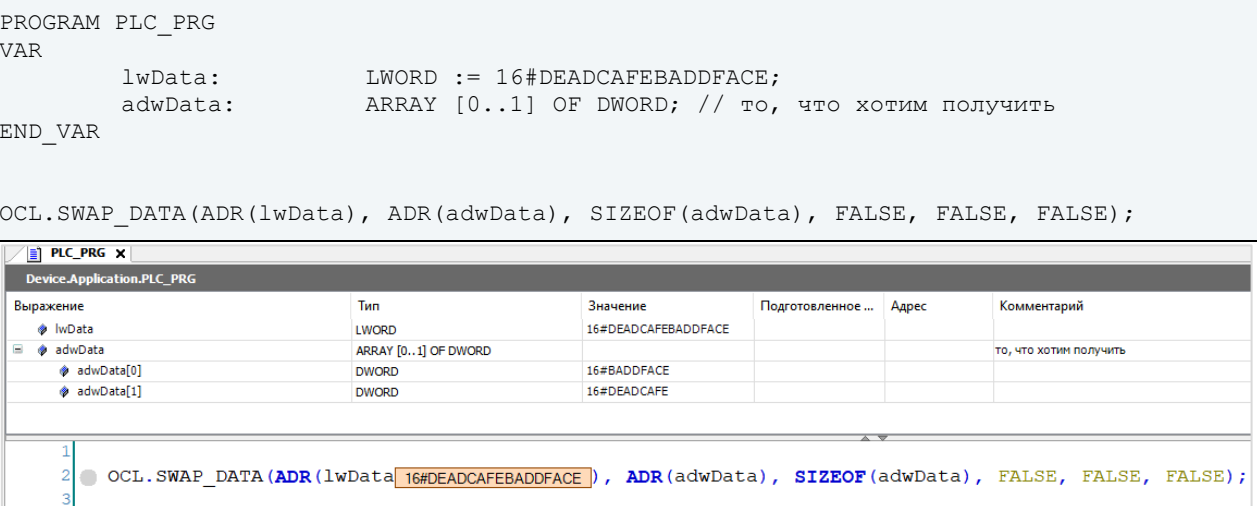
### 2.1.4. Не только ведь REAL

Понятно, что конвертация **двух WORD в REAL** – это конкретный частный случай. Но другие подобные задачи решаются совершенно аналогично. Например, пусть нам требуется конвертировать переменную типа **LWORD** в массив из двух **DWORD**. Рассмотрим ее решения теми же тремя вариантами – с помощью [библиотеки OwenCommunication](#), с помощью [библиотеки CAA Memory](#) и с помощью [объединений](#). Останавливаться на порядке регистров и байт в данном случае уже не будем.

1. В библиотеке **OwenCommunication** есть функция **SWAP\_DATA**, которая позволяет конвертировать данные из произвольного типа в произвольный.

```
PROGRAM PLC_PRG
VAR
    lwData:          LWORD := 16#DEADCAFEBAADDFACE;
    adwData:         ARRAY [0..1] OF DWORD; // то, что хотим получить
END_VAR

OCL.SWAP_DATA(ADR(lwData), ADR(adwData), SIZEOF(adwData), FALSE, FALSE, FALSE);
```



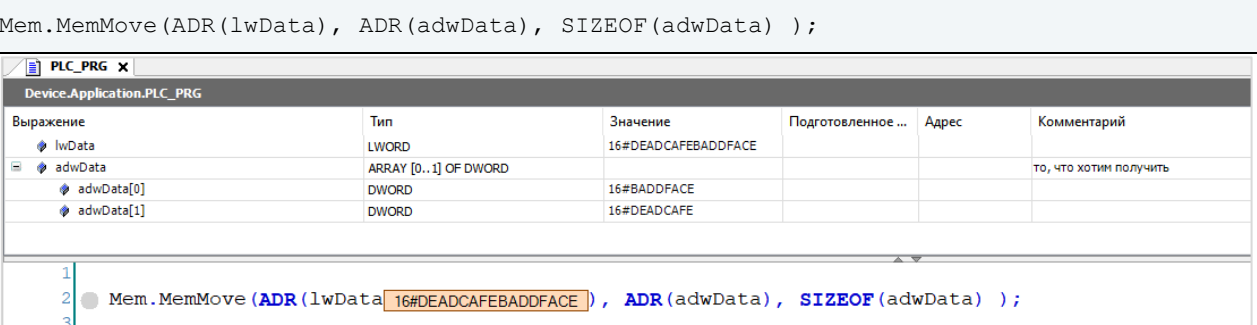
Выражение	Тип	Значение	Подготовленное ...	Адрес	Комментарий
lwData	LWORD	16#DEADCAFEBAADDFACE			
adwData	ARRAY [0..1] OF DWORD				то, что хотим получить
adwData[0]	DWORD	16#BAADDFACE			
adwData[1]	DWORD	16#DEADCAFE			

```
1
2 OCL.SWAP_DATA(ADR(lwData 16#DEADCAFEBAADDFACE), ADR(adwData), SIZEOF(adwData), FALSE, FALSE, FALSE);
3
```

Рис. 2.1.15. Использование функции **SWAP\_DATA**

2. В случае выбора библиотеки **CAA Memory** – используем уже хорошо известную нам функцию **MemMove**.

```
Mem.MemMove(ADR(lwData), ADR(adwData), SIZEOF(adwData) );
```



Выражение	Тип	Значение	Подготовленное ...	Адрес	Комментарий
lwData	LWORD	16#DEADCAFEBAADDFACE			
adwData	ARRAY [0..1] OF DWORD				то, что хотим получить
adwData[0]	DWORD	16#BAADDFACE			
adwData[1]	DWORD	16#DEADCAFE			

```
1
2 Mem.MemMove(ADR(lwData 16#DEADCAFEBAADDFACE), ADR(adwData), SIZEOF(adwData) );
3
```

Рис. 2.1.16. Использование функции **MemMove**

3. Вариант с объединением тоже не будет иметь принципиальных отличий от примера с **REAL**.

```

TYPE LWORD_AS_2DWORD :
UNION
    lwData:          LWORD;
    adwData:         ARRAY [0..1] OF DWORD;
END_UNION
END_TYPE

PROGRAM PLC_PRG
VAR
    lwData:          LWORD := 16#DEADCAFEBAFFFACE;
    adwData:         ARRAY [0..1] OF DWORD; // то, что хотим получить

    uData:           LWORD_AS_2DWORD;
END_VAR

uData.lwData := lwData;
adwData := uData.adwData;

```

The screenshot shows a software window titled 'PLC\_PRG x LWORD\_AS\_2DWORD'. Below the title bar, it displays 'Device.Application.PLC\_PRG'. A table lists variables and their values:

Выражение	Тип	Значение	Подготовленное ...	Адрес	Комментарий
lwData	LWORD	16#DEADCAFEBAFFFACE			
adwData	ARRAY [0..1] OF DWORD				то, что хотим получить
adwData[0]	DWORD	16#BAFFFACE			
adwData[1]	DWORD	16#DEADCAFE			
uData	LWORD_AS_2DWORD				

Below the table, a code editor shows the following lines of code:

```

1
2 uData.lwData := 16#DEADCAFEBAFFFACE := lwData 16#DEADCAFEBAFFFACE ;
3 adwData := uData.adwData;

```

Рис. 2.1.17. Использование объединения

## 2.2. Паспортные данные электросчетчика Меркурий

Давайте вернемся к примеру, который мы уже упоминали в [п. 1.1](#) – паспортным данным электросчетчика Меркурий.

**Пример:** Прочитать параметры прибора с сетевым адресом 66.

Запрос: 42 08 01 (CRC).

Ответ: 42 20 57 2F 42 1A 06 12 09 00 00 B4 E3 C2 97 DF 58 (CRC), где:

- «20 57 2F 42» – серийный номер 32874766;
- «1A 06 12» – дата выпуска 26.06.2018;
- «09 00 00» – версия ПО 9.0.0;
- «B4 E3 C2 97 DF 58» – вариант исполнения (см. п. 4.4.16 в [спецификации протокола](#)).

Итак, у нас есть массив байт **42 20 57 2F 42 1A 06 12 09 00 00 B4 E3 C2 97 DF 58** (CRC рассматривать не будем) и нам требуется преобразовать его вот в такую структуру (почему она будет именно такой – мы обсуждали в п. 1.1; наследование структур в данном примере мы использовать не будем):

```

1  TYPE MERCURY23x_PASSPORT_DATA :
2  STRUCT
3      udiSerialNumber:      UDINT;
4      dManufactureDate:    DATE;
5      sFirmwareVersion:    STRING(11);
6      abyModelInfo:       ARRAY [0..5] OF BYTE;
7  END_STRUCT
8  END_TYPE
9

```

Рис. 2.2.1. Создание структуры

Для начала объявим переменные для нашего массива и структуры, и организуем в программе условие для однократного выполнения кода конвертации массива (поскольку конвертировать данные имеет смысл только в момент их поступления – выполнять этот код циклически не требуется).

```

PROGRAM PLC_PRG
VAR
    abyResponse: ARRAY [0..16] OF BYTE := [16#42, 16#20, 16#57, 16#2F, 16#42,
        16#1A, 16#06, 16#12, 16#09, 16#00, 16#00, 16#B4, 16#E3, 16#C2, 16#97, 16#DF,
        16#58];
    stPassportData: MERCURY23x_PASSPORT_DATA;

    xUnpackResponse: BOOL := TRUE;

END_VAR

IF xUnpackResponse THEN

    // тут будет код конвертации
    xUnpackResponse := FALSE;

END_IF

```

Серийный номер счетчика хранится в виде 4 байт. В примере из документации он равен **20 57 2F 42** (HEX), что соответствует **32874766**. Видно, что серийный номер представляет собой последовательно записанные значения этих 4-х байтов в десятичной системе ( $16\#20 = 10\#32$ ,  $16\#57 = 10\#87$ ,  $16\#2F = 10\#47$ ,  $16\#42 = 10\#66$ ). Младший (четвертый) байт содержит единицы и десятки серийного номера, третий байт – сотни и тысячи и т.д.

Таким образом, выполнить конвертацию можно умножением байт на нужные коэффициенты и последующим сложением:

```
IF xUnpackResponse THEN

    stPassportData.udiSerialNumber := abyResponse[4] + abyResponse[3] * 100 +
        abyResponse[2] * 10000 + abyResponse[1] * 1000000;

    xUnpackResponse := FALSE;

END_IF
```

**Обратите внимание** – нулевой байт массива нам неинтересен, так как он содержит адрес счетчика. Данные ответа размещаются начиная с первого байта.

В результате выполнения этого кода произойдет следующее:

```
_____66
+_____4700
+_____870000
+320000000
=32874766
```

Мы получим именно то значение, которое ожидали.

Возможно, вы заметили, что по мере уменьшения индекса байта в массиве нам приходится умножать его на всё большее число и при этом есть зависимость между индексом массива и степенью коэффициента:

- `abyResponse[4]` – умножаем на 1 ( $10^0$ );
- `abyResponse[3]` – умножаем на 100 ( $10^2$ );
- `abyResponse[2]` – умножаем на 10000 ( $10^4$ );
- `abyResponse[1]` – умножаем на 1000000 ( $10^6$ ).

Это может подтолкнуть вас к идее оформить приведенный выше код в виде цикла:

```
IF xUnpackResponse THEN

    FOR i := 1 TO 4 DO
        stPassportData.udiSerialNumber := stPassportData.udiSerialNumber +
            abyResponse[i] * TO_UDINT(EXPT(10, 6 - 2*(i-1) ));
    END_FOR

    xUnpackResponse := FALSE;

END_IF
```

Но фактически вариант с циклом имеет лишь недостатки по сравнению с первым вариантом:

- для него требуется дополнительная переменная (счетчика цикла **i** типа **INT**);
- он занимает больше строк кода;
- он является более ресурсоемким из-за использования дополнительных операторов (**EXPT** и **TO\_UDINT**).

Поэтому использовать его не имеет смысла.

Эти же недостатки имеет и «строковый» вариант конвертации:

```
IF xUnpackResponse THEN
    FOR i := 1 TO 4 DO
        sSerialNumber := CONCAT(sSerialNumber, TO_STRING(abyResponse[i]) );
    END_FOR

    stPassportData.udiSerialNumber := TO_UDINT(sSerialNumber)

    xUnpackResponse := FALSE;
END_IF
```

Хотя все три варианта приведут нас к одинаковому результату – наиболее осмысленным решением является использование первого из них (без цикла).

Перейдем к дате выпуска счетчика. Она занимает три байта – **1A 06 12** (HEX), что соответствует дате «26.06.2018». Таким образом, значение каждого байта в десятичной системе соответствует одному из разрядов времени, и при этом байт года содержит только две последние цифры. Получается, нам надо «собрать» переменную типа **DATE** из трех отдельных целочисленных переменных. Проще всего это сделать с помощью готовой функции **DateConcat** из библиотеки **CAA DTUtil** (эту библиотеку потребуется добавить в проект – и, **обратите внимание**, она не будет работать в режиме эмуляции):

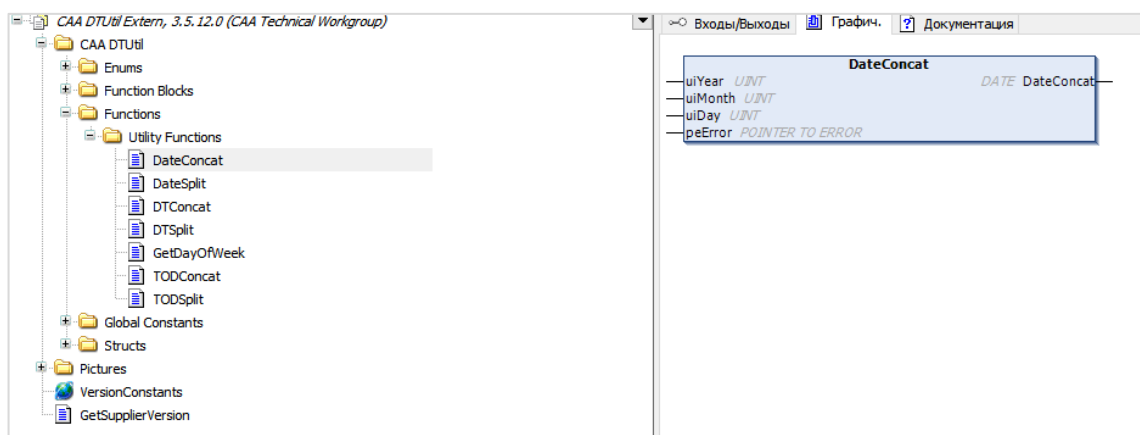


Рис. 2.2.2. Функция **DateConcat** из библиотеки **CAA DTUtil**

Одним из входов функции является указатель на переменную типа **ERROR**, в которую будет записан код ошибки в случае ее возникновения – нам потребуется объявить переменную такого типа.

```

PROGRAM PLC_PRG
VAR
    abyResponse:    ARRAY [0..16] OF BYTE := [16#42, 16#20, 16#57, 16#2F, 16#42,
        16#1A, 16#06, 16#12, 16#09, 16#00, 16#00, 16#B4, 16#E3, 16#C2,16#97,16#DF,
        16#58];
    stPassportData: MERCURY23x_PASSPORT_DATA;

    xUnpackResponse: BOOL := TRUE;

    eDateError:     DTU.ERROR;

END_VAR

IF xUnpackResponse THEN

    stPassportData.udiSerialNumber := abyResponse[4] + abyResponse[3] * 100 +
        abyResponse[2] * 10000 + abyResponse[1] * 1000000;

    stPassportData.dManufactureDate :=DTU.DateConcat(abyResponse[7] + 2000,
        abyResponse[6], abyResponse[5], ADR(eDateError) );

    xUnpackResponse := FALSE;

END_IF

```

Дальше на очереди версия прошивки. Как и дата, она представлена тремя байтами – **09 00 00** (HEX), что соответствует версии ПО **9.0.0**. Мы договорились, что будем отдавать пользователю версию прошивки в виде строки с точками-разделителями. Собрать строку из отдельных численных переменных можно с помощью функции **CONCAT** из библиотеки **Standard**. Но эта функция позволяет «склеить» только по 2 строки за раз – так что нам потребовалось бы вызвать ее несколько раз. Вместо этого воспользуемся функцией **CONCAT8** из библиотеки **OwenStringUtils** – она позволяет «склеить» до 8 строк за один вызов.

```

IF xUnpackResponse THEN

    stPassportData.udiSerialNumber := abyResponse[4] + abyResponse[3] * 100 +
        abyResponse[2] * 10000 + abyResponse[1] * 1000000;

    stPassportData.dManufactureDate :=DTU.DateConcat(abyResponse[7] + 2000,
        abyResponse[6], abyResponse[5], ADR(eDateError) );

    stPassportData.sFirmwareVersion := OSU.CONCAT8(TO_STRING(abyResponse[8]), '.',
        TO_STRING(abyResponse[9]), '.', TO_STRING(abyResponse[10]), '', '', '');

    xUnpackResponse := FALSE;

END_IF

```

Отметим, что в вызове функции **CONCAT8** нам пришлось указывать все 8 аргументов, хотя реально в данном случае требуется только 5 (три числа и две точки). Это связано с тем, что CODESYS требует при вызове функции обязательной передачи ей всех аргументов – поэтому три последних аргумента у нас являются пустыми строками. Но начиная с версии **CODESYS V3.5 SP16** появилась поддержка опциональных аргументов функций и методов (*если эти аргументы имеют какие-то начальные создания*) – так что если вы используете эту или более свежую версию среды, то лишние аргументы можно опустить.

Последнее, с чем нам осталось разобраться – это 6 байт варианта исполнения. В [п. 1.1](#) мы акцентировали внимание на том, что они из себя представляют, и просто решили объявить их как массив байт. Давайте сначала последуем этому решению и просто скопируем 6 байт из исходного массива ответа счетчика в массив, объявленный в нашей структуре, с помощью уже хорошо известной нам функции [MemMove](#):

```
IF xUnpackResponse THEN

  stPassportData.udiSerialNumber := abyResponse[4] + abyResponse[3] * 100 +
    abyResponse[2] * 10000 + abyResponse[1] * 1000000;

  stPassportData.dManufactureDate :=DTU.DateConcat(abyResponse[7] + 2000,
    abyResponse[6], abyResponse[5], ADR(eDateError) );

  stPassportData.sFirmwareVersion := OSU.CONCAT8(TO_STRING(abyResponse[8]), '.',
    TO_STRING(abyResponse[9]), '.', TO_STRING(abyResponse[10]), ',', ',', '');

  MEM.MemMove(ADR(abyResponse[11]), ADR(stPassportData.abyModelInfo),
    SIZEOF(stPassportData.abyModelInfo) );

  xUnpackResponse := FALSE;

END_IF
```

Если загрузить наш код в ПЛК, то можно убедиться, что он работает корректно – мы получили именно те значения, которые указаны в примере из спецификации протокола счетчика:

Рис. 2.2.3. Выполнение кода разбора массива ответа

В байтах варианта исполнения кодируется подробная информация о конкретной модели счетчика. Расшифровка байт приведена в [п. 4.4.16 спецификации протокола](#). Например, вот что означают первые два (**важно** – не младшие, а именно первые по порядку в ответе) байта:

№ байта ответа	7	6	5	4	3	2	1	0
1-й	CI A – класс точности по активной энергии: 0 – 0,2S; 1 – 0,5S; 2 – 1,0; 3 – 2,0		CI R – класс точности по реактивной энергии: 0 – 0,2; 1 – 0,5; 2 – 1,0; 3 – 2,0		Un – номинальное напряжение: 0 – 57,7 В; 1 – 230 В		In – номинальный ток: 0 – 5 А; 1 – 1 А; 2 – 10 А	
2-й	Число направлений: 0 – 2; 1 – 1	Температурный диапазон: 0 – -20°C; 1 – -40°C	Учет профиля средних мощностей: 0 – нет; 1 – да	Число фаз: 0 – 3; 1 – 1	Постоянная счетчика: 0 – 5000 имп/кВт·ч; 1 – 25000 имп/кВт·ч; 2 – 1250 имп/кВт·ч; 3 – 500 имп/кВт·ч; 4 – 1000 имп/кВт·ч; 5 – 250 имп/кВт·ч			

Рис. 2.2.4. Описание параметров варианта исполнения электросчетчика Меркурий 23х

Каждый байт описывает несколько параметров, и каждый параметр занимает от 2 до 4 бит. Каждый параметр счетчика характеризуется ограниченным числом возможных значений. Поэтому удобно представить эти параметры в виде [перечислений](#). Давайте добавим в наш пример разбор первого байта варианта исполнения – он содержит 4 параметра. Создадим для них три перечисления (так как классы точности для активной и реактивной энергии можно описать одним перечислением).

```

RATED_CURRENT
1 {attribute 'qualified_only'}
2 {attribute 'strict'}
3 TYPE RATED_CURRENT :
4 (
5     _5A := 0,
6     _1A := 1,
7     _10A := 2
8 );
9 END_TYPE

RATED_VOLTAGE
1 {attribute 'qualified_only'}
2 {attribute 'strict'}
3 TYPE RATED_VOLTAGE
4 :(
5     _57_7V := 0,
6     _230V := 1
7 );
8 END_TYPE

CI
1 {attribute 'qualified_only'}
2 {attribute 'strict'}
3 TYPE CI :
4 (
5     _0_2 := 0,
6     _0_5 := 1,
7     _1_0 := 2,
8     _2_0 := 3
9 );
10 END_TYPE

```

Рис. 2.2.5. Создание перечислений

Создадим структуру с экземплярами этих перечислений и добавим ее экземпляр в нашу исходную структуру **MERCURY23x\_PASSPORT\_DATA**:

```

MERCURY23x_MODEL_PARAMS
1 TYPE MERCURY23x_MODEL_PARAMS :
2 STRUCT
3     eRatedCurrent: RATED_CURRENT;
4     eRatedVoltage: RATED_VOLTAGE;
5     eCiA: CI;
6     eCiR: CI;
7 END_STRUCT
8 END_TYPE

MERCURY23x_PASSPORT_DATA
1 TYPE MERCURY23x_PASSPORT_DATA :
2 STRUCT
3     udiSerialNumber: UDINT;
4     dManufactureDate: DATE;
5     sFirmwareVersion: STRING(11);
6     abyModelInfo: ARRAY [0..5] OF BYTE;
7     stModelParams: MERCURY23x_MODEL_PARAMS;
8 END_STRUCT
9 END_TYPE

```

Рис. 2.2.6. Создание структуры параметров исполнения и объявление ее экземпляра в структуре паспортных данных счетчика

Чтобы преобразовать блоки бит байта варианта исполнения в перечисления – воспользуемся обычным побитовым копированием:

```
IF xUnpackResponse THEN

  stPassportData.udiSerialNumber := abyResponse[4] + abyResponse[3] * 100 +
    abyResponse[2] * 10000 + abyResponse[1] * 1000000;

  stPassportData.dManufactureDate :=DTU.DateConcat(abyResponse[7] + 2000,
    abyResponse[6], abyResponse[5], ADR(eDateError) );

  stPassportData.sFirmwareVersion := OSU.CONCAT8(TO_STRING(abyResponse[8]), '.',
    TO_STRING(abyResponse[9]), '.', TO_STRING(abyResponse[10]), '', '', '');

  MEM.MemMove(ADR(abyResponse[11]), ADR(stPassportData.abyModelInfo),
    SIZEOF(stPassportData.abyModelInfo) );

  stPassportData.stModelParams.eRatedCurrent.0 := abyResponse[11].0;
  stPassportData.stModelParams.eRatedCurrent.1 := abyResponse[11].1;

  stPassportData.stModelParams.eRatedVoltage.0 := abyResponse[11].2;
  stPassportData.stModelParams.eRatedVoltage.1 := abyResponse[11].3;

  stPassportData.stModelParams.eCiA.0 := abyResponse[11].4;
  stPassportData.stModelParams.eCiA.1 := abyResponse[11].5;

  stPassportData.stModelParams.eCiR.0 := abyResponse[11].6;
  stPassportData.stModelParams.eCiR.1 := abyResponse[11].7;

  xUnpackResponse := FALSE;

END_IF
```

По аналогии можно разобрать и остальные байты варианта исполнения счетчика.

### 2.3. Текущие измерения счетчика Питерфлоу

Представим, что мы опрашиваем расходомер-счетчик Питерфлоу по протоколу Modbus. Текущие измеренные значения счетчика занимают 33 регистра (это видно по их адресам: начальный адрес – 10500, конечный – 10532; так как параметр «ток индуктора» имеет тип Float, он занимает два регистра – 10531-10532). Их адреса расположены последовательно, так что их можно считать одним групповым запросом.

#### 4.4 Текущие измерения

Название	Адрес	Тип	Доступ	Примечание
Часы реального времени	10500	date time	R/O	
Время наработки (мин.)	10503	unsigned long	R/O	
Интеграл V+ (м³)	10505	double	R/O	Интеграл объема в прямом направлении
Интеграл V- (м³)	10509	double	R/O	Интеграл объема в обратном направлении
Флаги событий	10513		R/O	Набор битовых флагов в соответствии с <a href="#">Приложением 7</a>
Время наработки с ошибкой (мин.)	10515	unsigned long	R/O	
Расход (м³/ч)	10517	float	R/O	
Код АЦП	10519	float	R/O	
Напряжение питания (В)	10521	float	R/O	
Температура индуктора (°C)	10523	float	R/O	
Остаточная емкость батареи (А*ч)	10525	float	R/O	
Соппротивление среды (кОм)	10527	float		
Флаги состояния	10529	unsigned long		Набор битовых флагов в соответствии с <a href="#">Приложением 8</a>
Ток индуктора (мА)	10531	float		

Рис. 2.3.1. Фрагмент [карты регистров Modbus счетчика Питерфлоу](#)

Предположим, что мы уже считали эти данные и разместили их в массиве типа WORD (это довольно удобно, так как WORD соответствует одному регистру Modbus).

```
PROGRAM PLC_PRG
VAR
    awResponse: ARRAY [0..32] OF WORD;
END_VAR
```

Осталось преобразовать их в удобную для пользователя структуру. Чтобы понять, какого типа должны быть элементы этой структуры – необходимо изучить [документацию на протокол счетчика](#). Типы данных счетчика описаны в п. 3.1. Ознакомившись с этим пунктом мы поймем, что:

- тип **date\_time** соответствует **ARRAY [0..5] OF BYTE**, где каждый байт соответствует одному разряду времени (для года указываются только две последних цифры – как, кстати, и в дате изготовления счетчика Меркурий, который мы рассматривали [в прошлом пункте](#));
- тип **unsigned long** соответствует **UDINT**;
- тип **float** соответствует **REAL**;
- тип **double** соответствует **LREAL**.

То есть за исключением даты и времени – все остальные параметры можно описать типами стандарта МЭК 61131-3 и избежать каких-либо преобразований. Давайте оставим дату и время в виде массива байт и сформируем нашу структуру.

```

TYPE PITERFLOW_CURRENT_DATA :
STRUCT
    abyDateAndTime:          ARRAY [0..5] OF BYTE;
    udiOnTime:               UDINT;
    lrVolumeIntegralForward: LREAL;
    lrVolumeIntegralReverse: LREAL;
    udiEventFlags:          UDINT;
    udiOnTimeWithError:     UDINT;
    rConsumption:           REAL;
    rAdcCode:               REAL;
    rPowerVoltage:          REAL;
    rInductorTemp:          REAL;
    rRemainingBatteryCapacity: REAL;
    rEnvironmentResistance: REAL;
    udiStateFlags:          UDINT;
    rInductorCurrent:       REAL;
END_STRUCT
END_TYPE

```

Для нас принципиально важно, чтобы наша структура занимала в памяти контроллера 33 регистра (то есть 66 байт) – это позволит нам «наложить» на нее наш исходный массив WORD-ов, избежав каких-либо дополнительных конвертаций. Для начала проверим, сколько структура занимает сейчас:

```

2 | ● dwSize 72 := SIZEOF(PITERFLOW_CURRENT_DATA);

```

72 – это явно не 66. Поэтому добавим атрибут *pack\_mode*, чтобы «упаковать» данные структуры вплотную друг к другу и избежать разрывов между ними. Мы подробно рассказывали про этот атрибут в [п. 1.1.4](#).

```

{attribute 'pack_mode' := '1'}
TYPE PITERFLOW_CURRENT_DATA :
STRUCT
    abyDateAndTime:          ARRAY [0..5] OF BYTE;
    udiOnTime:               UDINT;
    lrVolumeIntegralForward: LREAL;
    lrVolumeIntegralReverse: LREAL;
    udiEventFlags:          UDINT;
    udiOnTimeWithError:     UDINT;
    rConsumption:           REAL;
    rAdcCode:               REAL;
    rPowerVoltage:          REAL;
    rInductorTemp:          REAL;
    rRemainingBatteryCapacity: REAL;
    rEnvironmentResistance: REAL;
    udiStateFlags:          UDINT;
    rInductorCurrent:       REAL;
END_STRUCT
END_TYPE

```

```

2 | ● dwSize 66 := SIZEOF(PITERFLOW_CURRENT_DATA);

```

Теперь обсудим переменные **udiEventFlags** и **udiStateFlags**. В [документации](#) (см. рис. 2.3.1) указано, что они представляют собой «наборы битовых флагов» и приведены ссылки на приложения 7 и 8 (см. рис. 2.3.2).

## Приложение 7. Система диагностики

Список флагов событий:

Бит	Описание
0	Загрязнение электродов
1	Запись в защищенный журнал невозможна по причине заполнения. Изменение параметров невозможно.
2	Нет воды, останов счёта.
3	Неисправность аппаратуры
4	Магнит, может быть отключен пользователем. Также можно остановить счёт
5	Короткое замыкание электродов
6	Переполение АЦП
7	Расчетная ошибка. Программная ошибка
8	Заданы неверные значения калибровки
9	Установлено аппаратное разрешение записи, необходимое при калибровке.
10	Ошибка стабилизатора тока.
11	Переполение частотного выхода
12	Расход больше максимума- Q3
13	RESERVED
14	Утечка - вода протекала через счетчик, не останавливаясь ни на один целый час в течении заданного времени. Можно задать порог срабатывания в процентах от максимума. Диапазон 0-100%, 0 отключает, 1% по умолчанию
15	Разрыв - расход воды имеет большое значение непрерывно в течении получаса, Можно задать порог срабатывания в процентах от максимума. Диапазон 0-100%, 0 отключает, 10% по умолчанию
16	Останов протока воды — расход меньше отсечки непрерывно в течение заданного времени. Диапазон 0-256 минут, 0 отключает
17	Большой шум электродов
18	Только для батарейных приборов. Low battery –сообщает за два месяца (уставка в % емкости) до момента, когда следует заменить батарейку или если напряжение питания падает ниже 3,0 вольт.
19	Только для батарейных приборов. Bad battery возникает при падении напряжения питания ниже 2,7 вольт и останавливает работу
20	Обрыв индуктора
21	КЗ индуктора
22	Ошибка тока
23	RESERVED
24	Счёт не вёлся, расход обнуляется (во время фатальных ошибок, перезапуска, не было воды)
25	Изменился защищённый журнал
26	Только для батарейных приборов. Внешнее питание
27	Перезапуск
28	Часы RTC (при наличии) не идут
29	RESERVED
30	RESERVED
31	RESERVED

## Приложение 8. Флаги состояния аппаратуры

Флаги состояния прибора представлены битовой маской:

- бит 0 - отсутствие батарейки;
- бит 1 - питание от 5В;
- бит 2 - питание от 12В;
- бит 3 - питание от батареи.

Значения остальных битов могут быть представлены в шестнадцатеричном виде.

Рис. 2.3.2. Описание отдельных бит наборов битовых флагов

То есть флаги хранятся в виде битовой маски – каждый бит переменной означает наличие (при значении **TRUE**) или отсутствие (при значении **FALSE**) какого-либо события. Давайте облегчим жизнь пользователю и предоставим ему эти переменные как [структуры с битовыми полями](#).

```

TYPE PITERFLOW_EVENT_FLAGS :
STRUCT
    xElectrodeContamination:      BIT;
    xWriteWasforbidden:           BIT;
    xNoWaterNoCount:              BIT;
    xHardwareFailure:             BIT;
    xMagnetCanBeDisabled:         BIT;
    xShortCircuitInElectrodes:    BIT;
    xAdcOverflow:                 BIT;
    xCalcError:                   BIT;
    xInvalidCalibrationValues:    BIT;
    xCalibrationIsPermitted:      BIT;
    xCurrentStabilizerError:      BIT;
    xFrequencyOutputOverflow:     BIT;
    xConsumptionIsOverMax:        BIT;
    xReserved13:                  BIT;
    xWaterLeak:                   BIT;
    xPipeRupture:                 BIT;
    xStopWaterFlow:              BIT;
    xLargeElectrodeNoise:         BIT;
    xLowBattery:                  BIT;
    xBadBattery:                  BIT;
    xOpenInductor:                BIT;
    xShortCircuitInductor:        BIT;
    xCurrentError:                BIT;
    xReserved23:                  BIT;
    xNoCountSoConsumptionIsReset: BIT;
    xWriteToProtectedLog:         BIT;
    xIsExternalPower:             BIT;
    xRestart:                     BIT;
    xRtcDoesNotWork:              BIT;
    xReserved29:                  BIT;
    xReserved30:                  BIT;
    xReserved31:                  BIT;
END_STRUCT
END_TYPE

```

```

TYPE PITERFLOW_STATE_FLAGS :
STRUCT
    xIsNoBattery:                 BIT;
    xIsPowerFrom5v:               BIT;
    xIsPowerFrom12v:              BIT;
    xIsPowerFromBattery:          BIT;
    // структура должна занять 32 бита – как исходный UDINT
    // так что добавляем «запасные» байты
    {attribute 'hide'}
    abyReserved:                  ARRAY [0..2] OF BYTE;
END_STRUCT
END_TYPE

```

В счетчике всего 4 флага событий, но под их хранение выделено 32 бита (тип – unsigned long). Поэтому наша структура **PITERFLOW\_STATE\_FLAGS** тоже должна занимать 32 бита – для этого мы добавим массив из трех байт – т.е. 24 бит (*поскольку память выделяется блоками по 8 бит и в момент объявления первого бита – **xIsNoBattery** – структура уже заняла 8 бит памяти*). Если вы объявляете такую структуру в библиотеке, то можно добавить для массива «запасных» байт [атрибут 'hide'](#), чтобы пользователь не видел его в менеджере библиотек – ведь этот массив все равно ему не нужен.

В итоге наша исходная структура будет выглядеть так:

```
{attribute 'pack_mode' := '1'}
TYPE PITERFLOW_CURRENT_DATA :
STRUCT
    abyDateAndTime:          ARRAY [0..5] OF BYTE;
    udiOnTime:               UDINT;
    lrVolumeIntegralForward: LREAL;
    lrVolumeIntegralReverse: LREAL;
    stEventFlags:          PITERFLOW_EVENT_FLAGS;
    udiOnTimeWithError:     UDINT;
    rConsumption:           REAL;
    rAdcCode:               REAL;
    rPowerVoltage:          REAL;
    rInductorTemp:          REAL;
    rRemainingBatteryCapacity: REAL;
    rEnvironmentResistance: REAL;
    stStateFlags:        PITERFLOW_STATE_FLAGS;
    rInductorCurrent:       REAL;
END_STRUCT
END_TYPE
```

Осталось только переложить в нее исходный массив WORD-ов. Например, с помощью функции **SWAP\_DATA** из библиотеки [OwenCommunication](#):

```
PROGRAM PLC_PRG
VAR
    awResponse:          ARRAY [0..32] OF WORD;
    stPiterflowCurrentData: PITERFLOW_CURRENT_DATA;
END_VAR

OCL.SWAP_DATA(ADR(awResponse), ADR(stPiterflowCurrentData),
    SIZEOF(stPiterflowCurrentData), FALSE, FALSE, FALSE);
```

Судя по [документации на счетчик](#) (п. 3.3) перестановка байт и регистров не потребуется. Впрочем, даже если бы это требовалось – нам было бы достаточно изменить значения соответствующих входов функции.

Последнее, что осталось обсудить – элемент **abyDateAndTime**. Он представляет собой метку времени и пользователю, вероятно, было бы удобно работать с ней как с переменной типа **DT** (или, по крайней мере, иметь такую возможность). «Собрать» переменную типа **DT** из отдельных целочисленных разрядов времени можно с помощью функции **DTConcat** из библиотеки **CAA DTUtil** (одну из функций этой библиотеки мы уже рассматривали в [п. 2.2](#)):

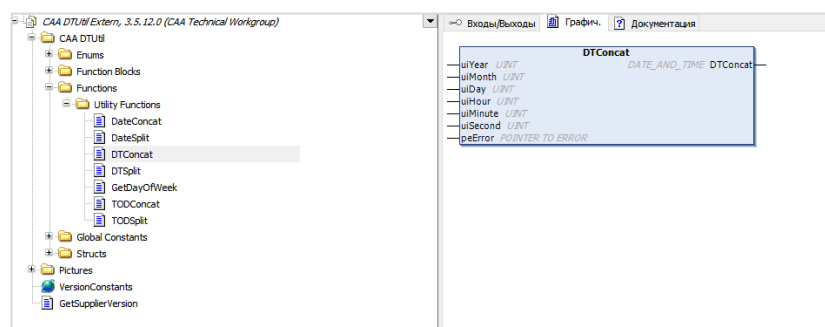


Рис. 2.3.3. Внешний вид функции **DTConcat**

## 2.4. Счетчики весового преобразователя ТВ-011 (протокол «Тензо-М»)

В последнем рассматриваемом примере мы немного поработаем с данными в [двоично-десятичном формате](#). Для этого откроем [документацию на протокол обмена](#) весового преобразователя ТВ-011 от компании Тензо-М (*название протокола совпадает с названием компании*) и посмотрим описание команды чтения счетчика (C8h).

ТВ-011, версия ПО "Pt-1.06" Протокол обмена данными по интерфейсу	«ТЕНЗО-М» Весоизмерительная компания
--	---

**2.12 C8h – передать счетчик (счетчики)**

**Запрос:** Adr, COP, NW, CRC (если включен при настройке);  
**Ответ:** Adr, COP, NW, W0, W1...Wn, CRC (если включен при настройке);  
 Код операции COP: **C8h**

Байт NW содержит номер запрашиваемого суммарного счетчика – от 0 до 9.  
 Если NW в старшем бите содержит лог. «1», то в ответной посылке передаются счетчики от 0 до указанного количества в младшей тетраде, но не более 9.

W0 ... Wn – соответствующий счетчик (по 5 байт), упакованный BCD – формат. Первые младшие байты.

0 – счетчик перезапусков преобразователя;  
 1 – суммарный вес продукта, перевешенного весами (в кг) с учетом установленного в параметрах количества знаков после запятой. Счетчик администратора – «С.»;  
 2 – значение заданной ограниченной дозы (в тоннах) с учетом установленного в параметрах количества знаков после запятой плюс один знак. Счетчик – «п.»;  
 3 – количество отвесов. Счетчик администратора – «С.п.»;  
 4 – суммарный вес продукта, перевешенного весами (в кг) с учетом установленного в параметрах количества знаков после запятой. Счетчик пользователя – «Е.»;  
 5 – количество отвесов. Счетчик пользователя – «Е.п.»;  
 6 – вес продукта, высыпаемого в последнем цикле (в кг) с учетом установленного в параметрах количества знаков после запятой. Счетчик – «d.»;  
 7 – значение заданной производительности (в тонн/час). Параметр SEL\_8-7;  
 8 – значение текущей производительности (в тонн/час). Счетчик – «Р.»4  
 9 – время цикла последнего отвеса (в десятых долях секунды с точностью 0,1 сек.). Счетчик – «t.»

**Пример:** FF, Adr, C8, 01, 00, 12, 05, 00, 00 CRC, FF, FF  
 соответствует 51200кг в счетчике «С.». Количество знаков после запятой определяется при передаче текущего веса.

Рис. 2.4.1. Фрагмент документации протокола Тензо-М для преобразователя ТВ-011

В ответ на запрос показаний счетчика преобразователь возвращает номер запрошенного счетчика и его значение в двоично-десятичном формате (BCD). Номер счетчика совпадает с тем, который указан в запросе – поэтому вряд ли он будет интересовать пользователя, который сам этот запрос и отправил. Нам нужно разобраться только с полученным значением. Из документации следует, что для некоторых счетчиков значение является целочисленным, а для некоторых – с плавающей запятой. Но фактически данные в ответе всегда являются целочисленными – просто для некоторых типов счетчиков пользователю потребуется разделить его на  $10^{\text{кол-во знаков после запятой}}$ , где

«кол-во знаков после запятой» задается в настройках прибора. Поэтому пользователю мы будем возвращать именно целочисленное значение, оставив обработку на его совести.

Давайте предположим, что мы уже извлекли из ответа нужные нам байты и разместили их в отдельном массиве. Заодно сразу объявим переменную, в которую мы хотим этот массив преобразовать:

```
PROGRAM PLC_PRG
VAR
    abyResponse:    ARRAY [0..4] OF BYTE := [16#00, 16#12, 16#05, 16#00, 16#00];
    lwCounterValue: LWORD; // должно получиться 51200
END_VAR
```

Тип нашей целочисленной переменной – **LWORD**. Это связано с тем, что исходное передаваемое значение представляет собой 5 байт в двоично-десятичном формате. Максимальное значение байта в таком формате – 16#99. Т.е. максимальное значение счетчика – это 999999999, а для хранения такого значения подойдут только три типа: **LINT**, **ULINT** и **LWORD**. Поскольку отрицательным значение счетчика быть не может – то нам на выбор остается только **ULINT** или **LWORD**. На самом деле, в CODESYS V3.5 между этими двумя типами нет никакой разницы; мы уже рассказывали об этом в [п. 1.1.3](#).

Если бы значение нашего счетчика занимало 4 байта – то мы могли бы переложить наш массив в переменную типа **DWORD** с противоположным порядком байт (*обратите внимание*, что в массиве значения младших разрядов счетчика расположены в первом байте, а старших – в последнем) и воспользоваться функцией **BCD\_TO\_DWORD** из библиотеки **Util**. Но функции **BCD\_TO\_LWORD** в этой библиотеке, к сожалению, нет. Поэтому мы сконвертируем каждый байт из BCD в десятичный формат с помощью функции **BCD\_TO\_BYTE**, а потом домножим их на нужные коэффициенты и сложим (мы уже использовали [этот принцип](#) при конвертации серийного номера электросчетчика Меркурий 23х, так что не будем повторно обсуждать его):

```
PROGRAM PLC_PRG
VAR
    abyResponse:    ARRAY [0..4] OF BYTE := [16#00, 16#12, 16#05, 16#00, 16#00];
    lwCounterValue: LWORD; // должно получиться 51200
END_VAR

lwCounterValue := UTIL.BCD_TO_BYTE(abyResponse[0]) +
    UTIL.BCD_TO_BYTE(abyResponse[1]) * 100 + UTIL.BCD_TO_BYTE(abyResponse[2]) * 10000 +
    UTIL.BCD_TO_BYTE(abyResponse[3]) * 1000000 +
    UTIL.BCD_TO_BYTE(abyResponse[4]) * 100000000;
```

Рис. 2.4.2. Преобразование значения счетчика из BCD в десятичный формат

Как можно заметить – мы получили именно тот результат, который ожидали.

### 3. Особенности конвертации строк

В [примерах](#) мы в основном рассматривали конвертацию числовых данных. На практике в некоторых случаях требуется также работать со строками – например, при реализации символьных протоколов (пример такого протокола – [DCON](#)), чтении и записи архивов и рецептов, использовании [REST API](#) для интеграции с web-сервисами и т.д. В данном разделе мы тезисно расскажем о реализации строк в CODESYS и особенностях работы с ними.

1. В CODESYS реализовано два типа данных для работы со строками – **STRING** и **WSTRING**. **STRING** используется для представления 8-битных [ASCII-based](#) кодировок. Конкретная используемая кодировка зависит от настроек операционной системы ПК, на котором вы работаете в CODESYS. Например, если вы работаете на ПК с русскоязычными региональными настройками, то при инициализации приведенной ниже строки в ее байты, вероятно<sup>3</sup>, будут записаны коды кодировки [Win1251](#):

```
PROGRAM PLC_PRG
VAR
    sTest: STRING := 'книги и коты';
END_VAR
```

Если вы работаете в ОС, где язык – чешский, то, вероятно, будут записаны коды кодировки [ISO 8859-2](#):

```
PROGRAM PLC_PRG
VAR
    sTest: STRING := 'láska žije tři roky';
END_VAR
```

**WSTRING** использует для представления символов кодировку **UCS-2** (которая [крайне близка](#) к [UTF-16](#)).

2. Литералы типа **STRING** записываются в одинарных кавычках, литералы типа **WSTRING** – в двойных:

```
PROGRAM PLC_PRG
VAR
    sTest: STRING := 'test';
    wsTest: WSTRING := "тест";
END_VAR
```

<sup>3</sup> Кодировка Win1251 является крайне популярной, но у конкретных пользователей может выбрана и другая кодировка – например, KOI8-R

3. Каждый символ **STRING** занимает 1 байт. Каждый символ **WSTRING** занимает 2 байта. Строки являются нужно-терминированными. Для каждой строки автоматически выделяется память для хранения терминирующего символа (его значение – 16#00 для STRING и 16#0000 для WSTRING). В большинстве случаев пользователю нет необходимости задумываться о терминирующих символах, но важно помнить об этом при расчете памяти, занимаемой строкой. Эта память выделяется статически при объявлении строки и зависит от максимально возможного числа символов, которое указано при ее объявлении. Если число символов не указано, то используется число **80**. Максимально возможная длина строки не имеет явных ограничений (но такие ограничения, например, могут возникнуть на уровне объема оперативной памяти контроллера).

```
PROGRAM PLC_PRG
VAR
// максимальная длина - 80 символов (по умолчанию)
// размер переменной - 81 байт (80 символов по 1 байту + 1 байт на терминатор)
sTest1: STRING;

// максимальная длина - 80 символов (по умолчанию)
// размер переменной - 162 байта (80 символов по 2 байта + 2 байта на терминатор)
wsTest1: WSTRING;

// максимальная длина - 300 символов
// размер переменной - 301 байт (300 символов по 1 байту + 1 байт на терминатор)
sTest2: STRING(300);

// максимальная длина - 300 символов
// размер переменной - 602 байт (300 символов по 2 байта + 2 байта на терминатор)
wsTest2: WSTRING(300);

END_VAR
```

4. Строки в CODESYS поддерживают индексный доступ – т.е. с переменной типа **STRING** можно работать как с массивом байт, а с переменной **WSTRING** – как с массивом WORD. Ниже приведен пример использования этой возможности для определения количества вхождений в строке нужного символа:

```
PROGRAM PLC_PRG
VAR
    sSource:          STRING(255) := 'текст, в котором, видимо, много запятых';
    i:                INT;
    iCount:           INT;
END_VAR
VAR CONSTANT
    c_sInterestingChar:  STRING(1) := ',';
END_VAR

iCount := 0;

FOR i := 0 TO LEN(sSource) DO
    IF sSource[i] = c_sInterestingChar[0] THEN
        iCount := iCount + 1;
    END_IF
END_FOR
```

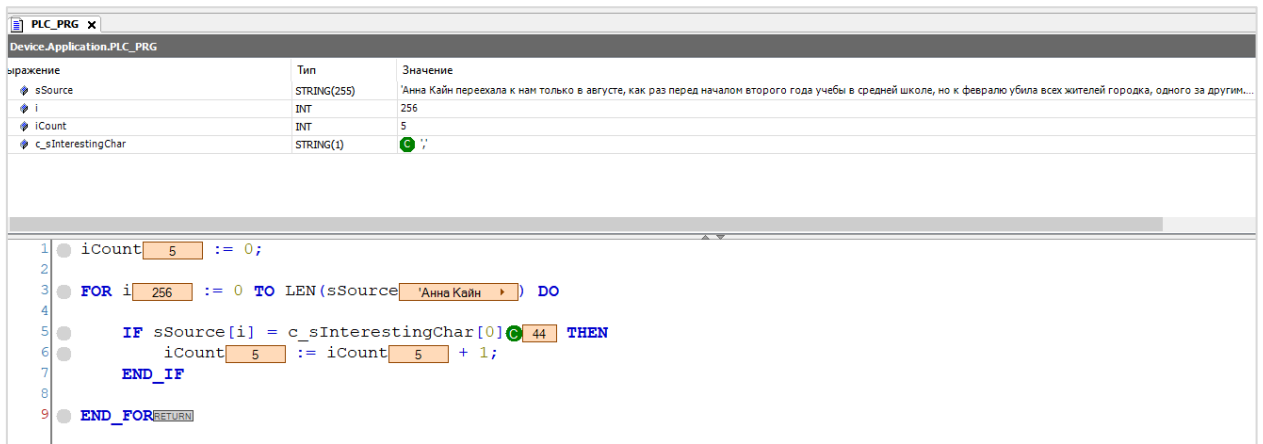


Рис. 3.1. Пример индексного доступа к строке

5. Операция присваивания пустой строки фактически только записывает в ее первый символ значение терминатора (см. пп. 3 на прошлой странице). Поэтому если вы работаете со строкой как с буфером и вам необходимо полностью ее очистить – используйте функцию **MemFill** из [библиотеки CAA Memory](#).

```
sTest := '1234';
sTest := ''; // фактически произошло sTest[0] := 0;
sTest[0] := 16#61; // теперь строка имеет значения 'a234'
// т.к. 16#61 - ASCII-код символа 'a'

// если надо «зачистить» строку
Mem.MemFill(ADR(sTest), SIZEOF(sTest), 0);
```

6. Строки в CODESYS поддерживают [управляющие последовательности](#).

Список доступных управляющих последовательностей для переменных типа **STRING/WSTRING**, которые используются во время работы с текстом (переход на новую строку, возврат каретки и т. д.), приведен в таблице:

Символ	Результат использования/Отображаемое значение
\$\$	\$ (символ доллара)
\$'	' (апостроф)
\$L	Перевод строки
\$N	Новая строка
\$R	Возврат каретки
\$P	Новая страница
\$T	Табуляция
\$xx (xx – код символа в HEX)	Символ <a href="#">таблицы ASCII</a> (только для <b>STRING</b> )

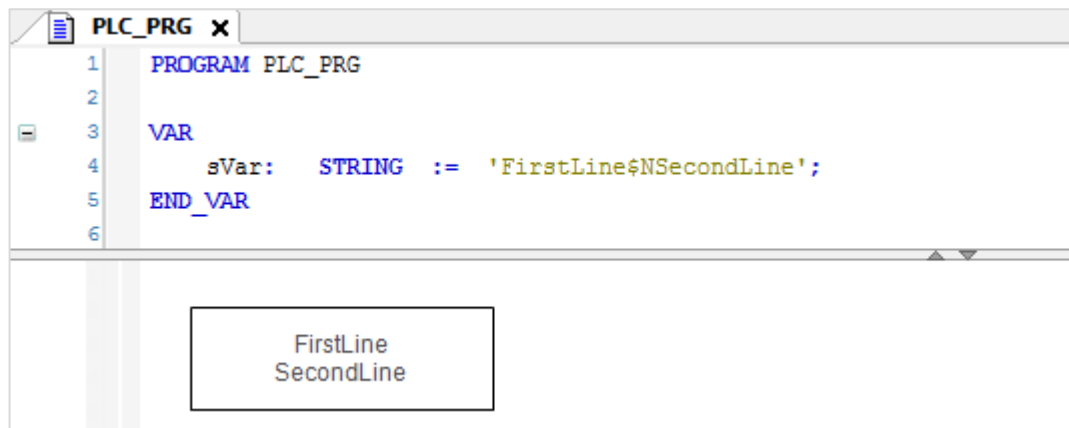


Рис. 3.2. Пример использования управляющей последовательности

7. Использование строковых переменных и литералов в визуализации может иметь различные особенности в зависимости от конкретного ПЛК. Например, некоторые панельные ПЛК могут не поддерживать кодировку Win1251, и для отображения в визуализации кириллического текста потребуется обязательно использовать тип WSTRING.

8. Основные библиотеки, используемые для работы со строками:

- [Standard](#) (базовые функции для работы со строками типа **STRING** – объединение, определение длины и т.д.);
- [Standard64](#) (базовые функции для работы со строками типа **WSTRING**, набор функций аналогичен **Standard**);
- [StringUtils](#) (набор расширенных функций для работы со строками, основанных на использовании указателей);
- [OwenStringUtils](#) (набор расширенных функций для работы со строками – конвертация кодировок, функции поиска, замены и т.д.);
- *В CODESYS V3.5 SP18 запланирован релиз библиотеки для эффективной работы с длинными строками (аналог StringBuilder из Java/C#).*

**Примечание** – функции библиотеки **Standard** и **Standard64** рассчитаны на работу со строками, длина которых не превышает **255** символов. Для работы с более длинными строками – используйте библиотеку **StringUtils** (см. [учебное видео 1](#) и [учебное видео 2](#)).

9. В библиотеке **StingUtils** есть функции для преобразования строк типа **WSTRING** в массивы байт кодировки [UTF-8](#) и обратно. *В CODESYS V3.5 SP18 запланирована нативная поддержка строк с кодировкой UTF-8.*

10. Существует [системный тип XSTRING](#), который при компиляции конвертируется в **STRING** или **WSTRING** в зависимости от настроек визуализации и директив компилятора.

11. Встроенные операторы **STRING\_TO\_WSTRING** и **WSTRING\_TO\_STRING** не выполняют конверсию кодировок.

```
1 wsTest "    " := STRING_TO_WSTRING('тест'); RETURN
```

Рис. 3.3. Пример работы функции **STRING\_TO\_WSTRING**

Фактически при вызове операторов происходит просто копирование данных. Например, в данном случае в младший байт каждого Unicode-символа был записан ASCII-код.

Для корректной конвертации кодировок Win1251/UCS-2 необходимо использовать функции из библиотеки **OwenStringUtils** – **CP1251\_TO\_UNICODE** и **UNICODE\_TO\_CP1251**.