Программирование ПЛК в соответствии с IEC 61 131-3 в среде «Mosaic»

десятый выпуск
ноябрь 2007г.
фирма оставляет за собой право проводить изменения

История изменений

Дата	Выпуск	Описание изменений
Август 2004г.	1	Первый вариант
Октябрь 2004г.	2	Дополнено описание стандартной библиотеки
Январь 2005г.	3	Проведены изменения для «Mosaic Help»
Февраль 2005г.	4	Изменение примера 3.6.2.5 – дополнено слово „DO“
Апрель 2005г.	5	Дополнена табл.3.3 Специальные знаки в последовательностях Исправление размеров типа SINT в гл.3.2.1 в Табл.3.5 Табл.3.18 дополнена образцами вызова функций над последовательностью знаков Дополнена гл.3.7.2 Библиотека функций над последовательностью знаков
Ноябрь 2005г.	6	Дополнено описание библиотеки конверсионных функций Исправлена Табл.3.20 Стандартные функции с типами даты и времени Дополнено описание типа данных PTR_TO
Февраль 2006г.	7	Расширено описание типов данных и переменных Добавлено описание библиотеки арифметических функций
Март 2006г.	8	Дополнены основные идеи программирования согласно норме Дополнено описание языка IL Описание библиотек перемещено в отдельный документ TXV 003 22
Ноябрь 2006г.	9	Добавлено описание директив
Ноябрь 2007г.	10	Добавлено описание графических языков LD и FBD

1 ВВЕДЕНИЕ

1.1 Норма IEC 61 131

Норма IEC 61 131 для программируемых систем управления состоит из пяти основных частей и представляет собой совокупность требований к современным системам управления. Не зависит от конкретной организации или фирмы и имеет широкую международную поддержку. Отдельные части нормы посвящены как техническому так и программному обеспечению данных систем.

В ЧР были приняты отдельные части данной нормы под следующими номерами и названиями:

ČSN EN 61 131-1 Программируемые блоки управления- Часть 1: Общая информация
ČSN EN 61 131-2 Программируемые блоки управления- Часть 2: Требования к оборудованию и испытания
ČSN EN 61 131-3 Программируемые блоки управления- Часть 3: Языки программирования
ČSN EN 61 131-4 Программируемые блоки управления- Часть 4: Поддержка пользователей
ČSN EN 61 131-5 Программируемые блоки управления- Часть 5: Связь
ČSN EN 61 131-7 Программируемые блоки управления- Часть 7: Программирование fuzzy управления

В Европейском Союзе эти нормы приняты под номером EN IEC 61 131.

Языки программирования определяет норма IEC 61 131-3, которая является частью семейства норм IEC 61 131 и представляет первую важную попытку стандартизации языков программирования в промышленной автоматизации.

Норму 61 131-3 можно рассматривать с разных точек зрения, напр. так, что это результат трудоемкого труда семи международных компаний, которые в разработку норм вложили свой десятилетний опыт на поле промышленной автоматизации, или же так, что в своей совокупностьи содержит около 200 страниц текста и где-то 60 таблиц. На ее создании работала группа, принадлежащая к рабочему коллективу SC65B WG7 международной организации по стандартизации IEC (International Electrotechnical Comission). Результатом является *спецификация синтакса и семантики унифицированного набора языков программирования, включая общую модель программного обеспечения и структурного языка*. Эта норма была принята в качестве инструкции большинством крупных изготовителей ПЛК.

1.2 Терминология

Комплекс норм для программируемых блоков управления хотя и был в ЧР принят, но не был переведен на чешский язык. По этой причине это руководство использует терминологию так, как это декламируется на ČVUT FSI Praha при обучении автоматизации. Одновременно во всем тексте приводится также английская терминология с целью однозначно причислить чешскую трактовку английскому оригиналу.

1.3 Основные принципы нормы IEC 61 131-3

Норма IEC 61 131-3 – третья часть всего семейства норм IEC 61 131. По существу разделяется на две основные части:

- Общие элементы
- Языки программирования

1.3.1 Общие элементы

Типы данных

В рамках общих элементов определены типы данных. Определение типов данных помогает предупредить ошибки в самом начале реализации проекта. Необходимо определить типы всех используемых параметров. Обычные типы данных **`BOOL`**, **`BYTE`**, **`WORD`**, **`INT`** (Integer), **`REAL`**, **`DATE`**, **`TIME`**, **`STRING`** и т.д. Из этих основных типов данных потом можно выводить собственные пользовательские типы данных, т. наз. производные типы данных. Таким способом можно напр. определить в качестве отдельного типа данных аналоговый входной канал и повторно его использовать под определенным названием.

Переменные

Переменные могут быть отнесены явно к технически обеспеченным адресам (напр. входам, выходам) только в конфигурациях, источниках или программах. Таким образом достигается высокой степени независимости технического обеспечения и возможности повторного использования программного обеспечения на различных платформах программного обеспечения.

Область действия переменных обычно ограничена только на ту программную организационную единицу, в которой были декларированы (переменные у нее локальные). Это означает, что их названия могут использоваться в других частях без ограничений. Этими мерами будет элиминирован ряд других ошибок. Пока переменные действуют глобально, напр. в рамках целого проекта, то онир должны декларироваться как глобальные (**`VAR_GLOBAL`**). Чтобы можно было правильно настроить исходное состояние процесса или машины, параметрам может быть причислена исходное значение при запуске или холодном повторном запуске.

Конфигурация, источники и задачи

На наивысшем уровне все решение программного обеспечения определенной проблеммы управления формулируется как т. наз. *конфигурация* (Configuration). Конфигурация зависит от конкретной стистемы управления, в том числе организация технического обеспечения, как например типы узлов процессора, области памяти, предназначенной для входных и выходных каналов и характеристики программного обеспечения системы (операционной системы).

В рамках конфигурации потом можем определить один или более т. наз. *источников* (Resource). Источник можем рассматривать как какое-либо оборудование, которое способно проводить IEC программы.

Внутри источника можем определить одну или более т. наз. *задач* (Task). Задачи руководят проведением файла программ и/или функциональных блоков. Эти единицы могут про-

водиться или же периодически или после возникновения специального пускового события, напр. изменение переменной.

Программы оформлены из ряда различных элементов программного обеспечения, которые записаны на одном из языков определенных в норме. Часто программа состоит из сети функций и функциональных блоков, которые способны обмениваться данными. Функции и функциональные блоки - основные строительные камни, которые содержат структуры данных и алгоритм.

Программные организационныея единицы

Функции, функциональные блоки и программы в рамках нормы IEC 61 131 совместно называются *программные организационные единицы* (Program Organisation Units, иногда для этого важного и часто используемого термина используется сокращение POUs).

Функции

IEC 61 131-3 определяет стандартные функции и определенные пользователем функции. Стандартные функции напр.**ADD** для считывания, **ABS** для абсолютной величины, **SQRT** для корня, **SIN** для синуса и **COS** для косинуса. Как только будут определены новые пользовательские функции, они могут использоваться повторно.

Функциональные блоки

Функциональные блоки можно рассматривать как интегрированные контуры, которые представляют решение технического обеспечения специализированной функции управления. Содержат алгоритмы и данные, могут также сохранять информацию о прошлом, (этим отличаются от функций). Они имеют точно определенный интерфейс и скрытые внутренние переменные, так же как интегрированный контур или черный ящик. Этим позволяют однозначно отделить различные уровни программаторов или обслуживающего персонала. Классическим примером функционального блока являются напр. контур регулирования для температуры или PID регулятор.

Как только один раз будет определен функциональный блок, он может использоваться повторно в данной программе, или в другой программе, или даже в другом проекте. То есть он универсален и может использоваться многократно. Функциональные блоки могут записываться на любом языке, определенном в норме. То есть может полностью определяться пользователем. Производные функциональные блоки основаны на стандартных функциональных блоках, но в рамках правил нормы можно создавать и совсем новые пользовательские функциональные блоки.

Интерфейс функций и функциональных блоков описан таким же способом: Между декларацией обозначается наименование блока и декларация для конца блока куказан список деклараций входных переменных, выходных переменных и собственный код в т. наз. корпусе блока.

Программы

На основании выше указанных определений можно сказать, что программа в сущности является сетью функций и функциональных блоков. Программа может быть записана на любом языке определенном в норме.

1.3.2 Языки программирования

В рамках стандарта определены четыре языка программирования. Их семантика и синтаксис точно определены и не имеется какое-либо пространство для неточного выражения. Освоением этих языков открывается путь к использованию широкой шкалы систем управления, которые основаны на данном стандарте.

Языки программирования разделяются на две основных категории:

Текстовые языки
IL - Instruction List - язык перечня инструкций
ST - Structured Text - язык структурированного текста

Графические языки
LD - Ladder Diagram - язык диаграммы перегородок (язык контактных схем)
FBD - Function Block Diagram - язык функциональной блок-схемы

Для первой сводки на Рис.1.1 та же логическая функция, а именно - произведение переменной **A** и отрицательной переменной **B** с результатом, записываемым в переменные **C**, выражен на всех четырех языках программирования.

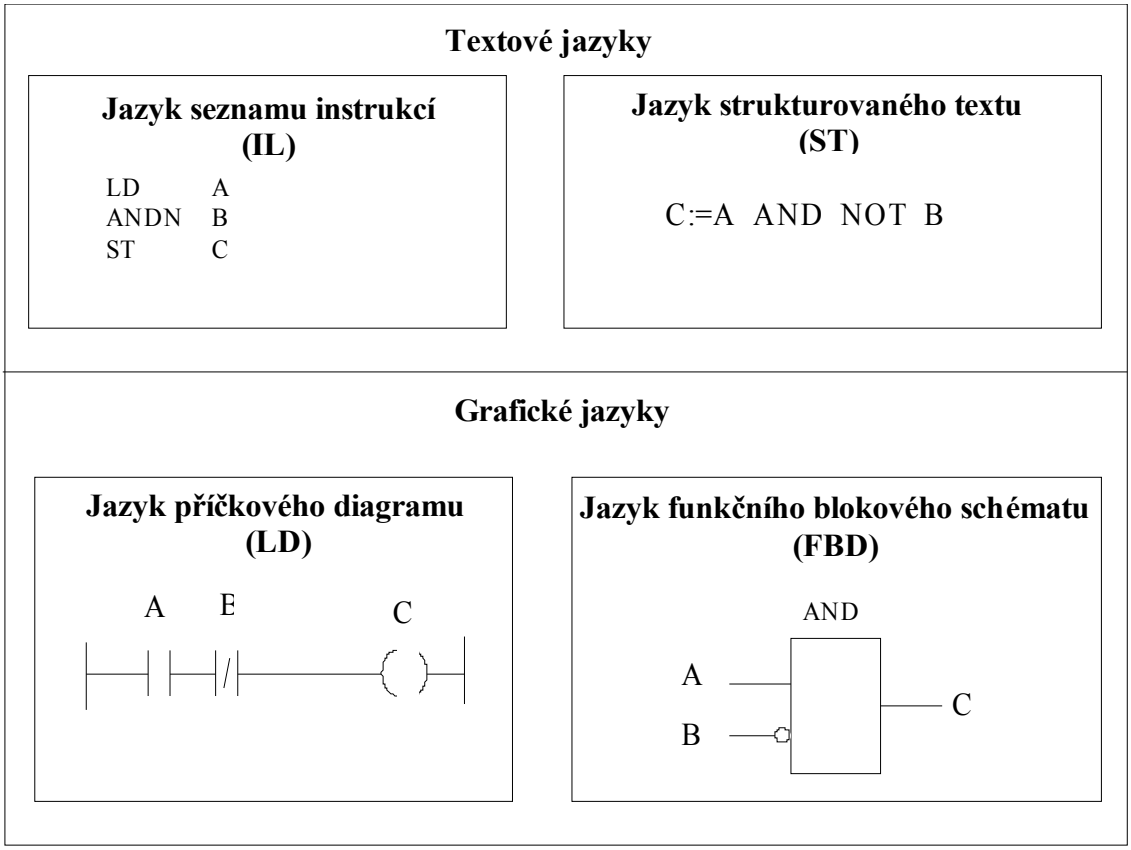


Рис. 1 Логическая функция ANDN на четырех основных языках

Выбор языка программирования зависит от опыта программатора, от типа решаемой проблеммы, от уровня описания проблеммы, от структуры системы управления и от ряда

других обстоятельств, как напр. тип отрасли промышленности, обычаев фирмы, которая внедряет систему управления, опыта сотрудников и группы и т.п.

Все четыре основные языки (IL, ST, LD и FBD) взаимосвязаны. Приложение, описанное в них образует определенный основной массив данных, к которому принадлежит большой объем технического опыта (“know-how”). Собственно говоря создают также основной инструмент коммуникации для договоренности специалистов различных отраслей и сфер.

LD - Ladder Diagram - язык перегородочной диаграммы
- происходит из США. Основан на графическом представлении релейной логики.

IL - Instruction List - язык перечня инструкций
- это его европейская противоположность. Как текстовой язык напоминает ассемблер.

FBD - Function Block Diagram - язык функциональной блочной схемы
- очень близок процессуальной промышленности. Выражает поведение функций, функциональных блоков и программ как массив взаимосвязанных графических блоков, так же как в электронных контурах диаграмм. Это определенная система элементов, которые перерабатывают сигналы.

ST - Structured Text - язык структурированного текста
- очень продуктивный язык программирования, корни которого имеются в известных языках Ada, Pascal и C. Содержит все важные элементы современного языка программирования, в том числе разветвление (**`IF-THEN-ELSE`** а **`CASE OF`**) и итеративные контуры (**`FOR`**, **`WHILE`** а **`REPEAT`**). Эти элементы могут быть погружены. Данный язык является отличным инструментом для определения комплексных функциональных блоков, которые могут использоваться в любом языке программирования.

Известно, что для систематического программирования существуют в сущности два подхода, а именно - сверху - *вниз* (Top-down) или *снизу - вверх* (Bottom-up).

Указанный стандарт поддерживает оба доступа развития программ. Специфицируем все приложение и разделим его на части (субсистемы), декларируем переменные и т.п. или начнем программировать приложение снизу, напр. через производные (пользовательские) функции и функциональные блоки. Независимо от того, какой путь мы выберем, опытная среда «Mosaic», которая отвечает требованиям стандарта IEC 11 131-3, будет ее поддерживать и помогать при создании целого приложения.

2 ОСНОВНЫЕ ПОНЯТИЯ

Эта глава коротко объясняет значение и использование основных понятий при программировании систем ПЛК согласно норме IEC 61 131-3. Эти понятия будут объяснены на простых примерах. Детальное описание объясняемых понятий читатель найдет в следующих главах.

2.1 Основные строительные блоки программы

Основным понятием при программировании согласно норме IEC 61 131-3 является термин ***Программируемый Организационный Элемент*** или сокращенно ***POU*** *(Program Organisation Unit)*. Как следует из названия, POU – это наименьшая независимая часть пользовательской программы. POU могут поставляться от производителя системы управления или ее может написать пользователь. Каждая POU может вызвать следующую POU и при этом вызовен может на выбор передавать выбранной POU один или более параметров.

Существует три основных типа POU :

- ***функция*** *(function, FUN)*
- ***функциональный блок*** *(function block, FB)*
- ***программа*** *(program, PROG)*

Самой простой POU является ***функция***, главной характеристикой которой является то, что если она вызвана такими же входными параметрами, она должна продуцировать такой же результат (рабочую величину). Функция может возвращать только один результат.

Следующим типом POU является ***функциональный блок***, который в отличие от функции, может помнить некоторые значения из предыдущего вызова (напр. информацию о состоянии). Они могут влиять на результат. Главным отличием между функцией и функциональным блоком является способность функционального блока владеть памятью для запоминания значений некоторых переменных. Эту способность функции не имеют и их результат однозначно определен входными параметрами при вызове функции. Функциональный блок может также (в отличие от функции) возвращать более чем один результат.

Последним типом POU является ***программа***, которая представляет максимальную программную единицу в пользовательской программе. Центральная единица ПЛК может перерабатывать более программ и язык программирования ST содержит средства для определения запуска программ (в какой период выполнять программу, с каким приоритетом, и т.п.).

Каждая POU состоит из двух основных частей : ***декларационной*** и ***исполнительной***, как это указано на Рис.2.1. В декларационной части POU определяются переменные необходимые для работы POU. Исполнительная часть содержит собственные команды для реализации требуемого алгоритма.

Определение POU на Рис.2.2 начинается ключевым словом **PROGRAM** и заканчивается ключевым словом **END_PROGRAM**. Эти ключевые слова ограничивают диапазон POU. За ключевым словом **PROGRAM** указано название POU. После этого последует Декларационная часть POU. Онаа содержит определение переменных указанное между ключевыми словами **VAR_INPUT** и **END_VAR** или же **VAR** и **END_VAR**. В заключение указана исполнительная часть POU содержащая команды языка ST для обработки переменных. Тексты указанные между знаками **(*** а ***)** – это примечания (коментарии).

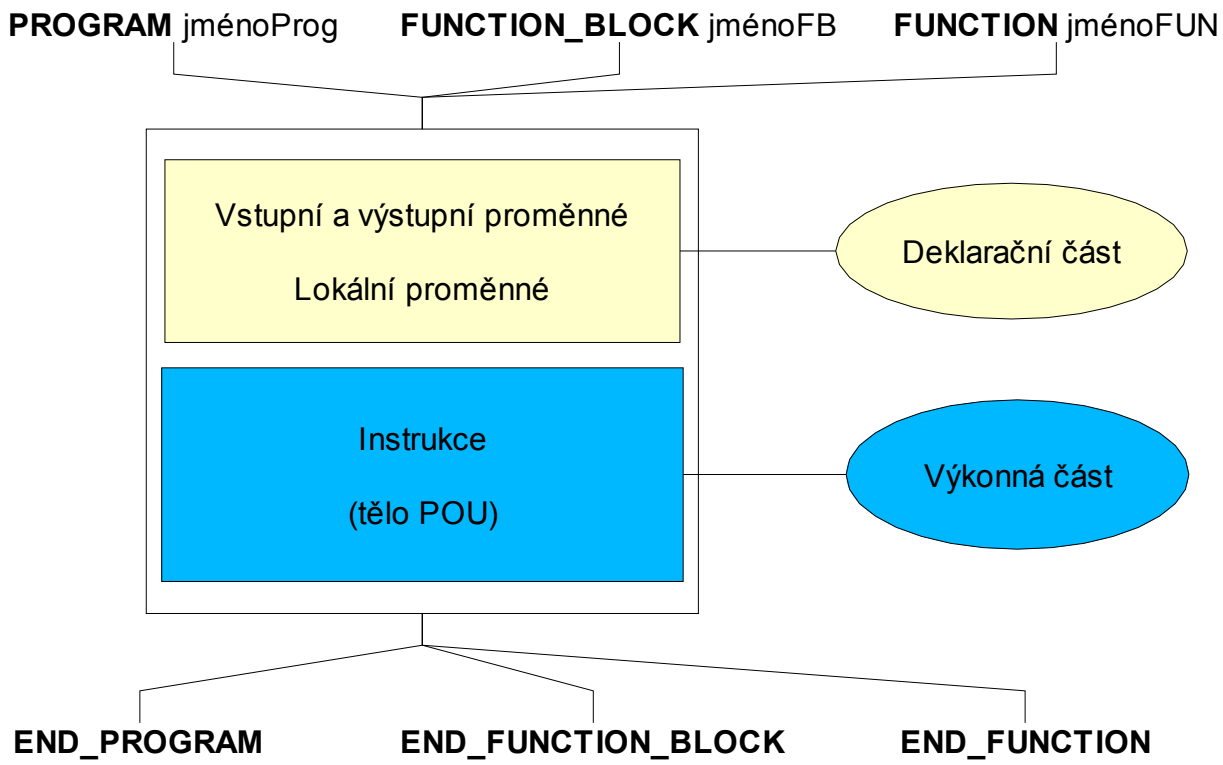


Рис. 2 Основная структура POU

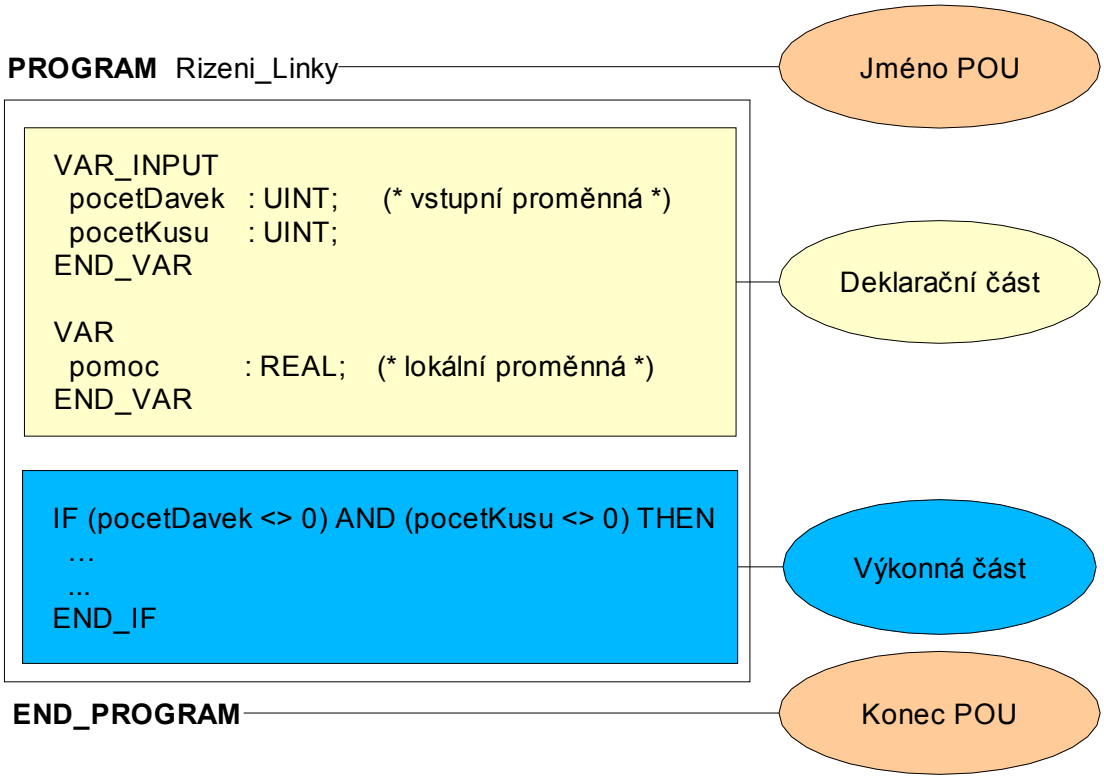


Рис.3 Основная структура POU PROGRAMA

2.2 Декларационная часть POU

Декларационная часть POU содержит определение переменных необходимых для работы POU. Переменные используются для записи и обработки информации. Каждая ***переменная*** определена ***названием переменной*** и ***типом данных***. Тип данных определяет величину переменной в памяти и одновременно в значительной степени определяет способ обработки переменных. Для определения переменных в распоряжении имеются стандартные типы данных (**BOOL**, **BYTE**, **INT**, …). Использование этих типов зависит от того, какая информация будет в переменных записана (напр. тип **BOOL** для информации типа ДА-НЕТ, тип **INT** для записи целых цифр со знаками т.п.). Пользователь само собой имеет возможность определить свои собственные типы данных. Размещение переменных в памяти системы ПЛК обеспечивается автоматически средой программирования. Если это необходимо, может размещение переменных в памяти определить также пользователь.

Переменные можем разделить согласно использованию на ***глобальные*** и ***локальные***. Глобальные переменные определены вне POU и могут использоваться в любой POU (видимы из любой POU). Локальные переменные определены внутри POU и в рамках этой POU могут использоваться (из остальных POU невидимы).

И наконец переменные также используются для передачи параметров при вызове POU. В данных случаях речь идет о ***входных*** или же ***выходных*** переменных.

Пример 2.1 Декларация переменных POU

```
FUNCTION_BLOCK ExampleDeclarationVar

  VAR_INPUT                       (* входные переменные *)
    logCondition      : BOOL;     (* бинарная величина*)
  END_VAR
  VAR_OUTPUT                      (* выходные переменные *)
    result            : INT;      (* целочисленная величина со знаком*)
  END_VAR
  VAR                             (* локальные переменные *)
    CheckSum          : UINT;     (* целочисленная величина*)
    tempResult        : REAL;     (* фактическая величина*)
  END_VAR

END_FUNCTION_BLOCK
```

В примере 2.1 указано определение входные переменные POU, переменная называется **logCondition** и имеет типа **BOOL**, что означает, что может содержать значение **TRUE** (логическая „1“) или **FALSE** (логическая „0“). Эта переменная предназначена в качестве входного параметра передаваемого при вызове POU.

Следующая определенная переменная - выходная, называется **Result** и имеет тип **INT** (integer), поэтому может содержать целочисленное значение в диапазоне от –32 768 до +32 767. В данной переменной передается значение в вышестоящую POU.

Переменные определенные между ключевыми словами **VAR** и **END_VAR** - локальные и их можно использовать только в рамках POU. Переменная **CheckSum** имеет тип **UINT** (unsigned integer) и может сохранять целые цифры в диапазоне от 0 до 65535. Переменная **tempResult** имеет тип **REAL** и предназначена для работы с реальными числами.

2.3 Исполнительная часть POU

Исполнительная часть POU последует за декларационной частью и содержит команды и инструкции, которые обработанные центральной единицей ПЛК. В исключительных случаях определение POU не обязательно должно содержать какую-либо Декларационную часть и потом исполнительная часть указана непосредственно за определением начала POU. В качестве примера может послужить POU, которая работает только с глобальными переменными, что хотя и не является идеальным решением, но может существовать.

Исполнительная часть POU может содержать вызов других POU. При вызове могут передаваться параметрами для вызванной функции или же функциональных блоков.

2.4 Демонстрация программы

Пример 2.2 Демонстрация программы

```
VAR_GLOBAL
  // входа
  sb1 AT %X0.0,
  sb2 AT %X0.1,
  sb3 AT %X0.2,
  sb4 AT %X0.3   : BOOL;

  // выхода
  km1 AT %Y0.0,
  km2 AT %Y0.1,
  km3 AT %Y0.2,
  km4 AT %Y0.3   : BOOL;
END_VAR


FUNCTION_BLOCK fbStartStop
//-----------------------------------------------------------------------------
  VAR_INPUT
    start        : BOOL R_EDGE;
    stop         : BOOL R_EDGE;
  END_VAR
  VAR_OUTPUT
    output       : BOOL;
  END_VAR

  output := ( output OR start) AND NOT stop;
END_FUNCTION_BLOCK



FUNCTION_BLOCK fbMotor
//-----------------------------------------------------------------------------
  VAR_INPUT
    motorStart    : BOOL;
    motorStop     : BOOL;
  END_VAR
  VAR
    StartStop     : fbStartStop;
    MotorIsRun    : BOOL;
    startingTime  : TON;
  END_VAR
  VAR_OUTPUT
    wye          : BOOL;       // звезда
    delta        : BOOL;       // тройугольник
  END_VAR

  StartStop( start := motorStart, stop := motorStop,
             output => MotorIsRun);
  startingTime( IN := MotorIsRun, PT := TIME#12s, Q => delta);
END_FUNCTION_BLOCK
```

```
PROGRAM Test
//------------------------------------------------------------------------------
  VAR
    motor1       : fbMotor;
    motor2       : fbMotor;
  END_VAR

  motor1( motorStart := sb1, motorStop := sb2,
          wye => km1, delta => km2);
  motor2( motorStart := sb3, motorStop := sb4,
          wye => km3, delta => km4);
END_PROGRAM


CONFIGURATION exampleProgramST
  RESOURCE CPM
    TASK FreeWheeling(Number := 0);
    PROGRAM prg WITH FreeWheeling : Test ();
  END_RESOURCE
END_CONFIGURATION
```

3 ОБЩИЕ ЭЛЕМЕНТЫ

Данная глава описывает синтаксис и семантику основных общих элементов языков программирования для ПЛК системы согласно стандарту IEC 61 131-3.

Синтаксис описывает элементы, которые имеются в распоряжении для программирования ПЛК и способ, которым они могут взаимно коббинироваться.

Семантика выражает их значение.

3.1 Основные элементы

Каждая программа для ПЛК состоит из основных ***простых элементов***, определенных минимальных элементов, из которых образуются декларации и команды. Эти простые элементы можем разделить на :

- *разделители* (Delimiters),
- *идентификаторы* (Identifiers)
- *литеральные константы* (Literals)
- *ключевые слова* (Keywords)
- *коментарии* (Comments)

Для большей обзорности текста ключевые слова написанные жирным шрифтом, лучше выразить структуру деклараций и команд. В среде «Mosaic» кроме того различаются по цвету.

Разделители - это специальные знаки (напр.(,), =, :, пробел, и т.п.) с различным значением.

Идентификаторы - это альфацифровые последовательности знаков, которые предназначены для выразить название пользовательских функций, меток или программных организационных элементов (напр. **`Temp_N1`**, **`Switch_On`**, **`Step4`**, **`Move_right`** и т.п.).

Литеральные константы предназначены для прямого представления значений переменных (напр.*0,1*; *84*; *3,79*; *TRUE* ; *зеленая* и т.п.).

Ключевые слова - это стандартные идентификаторы (напр. **`FUNCTION`**, **`REAL`**, **`VAR_OUTPUT`**, и т.п.). Их точная форма и значение отвечает норме IEC 61 131-3. Ключевые слова не должны использоваться для создания каких-либо имен пользователей. Для записи ключевых слов могут использоваться как большие так и маленькие буквы или же их произвольная комбинация.

К забронированным ключевым словам принадлежат :

- *названия элементарных типов данных*
- *названия стандартных функций*
- *названия стандартных функциональных блоков*
- *названия входных параметров стандартных функций*
- *названия входных и выходных параметров стандартных функциональных блоков*
- *элементы языка IL и ST*

Все забронированные ключевые слова – указаны в приложении H нормы IEC 61 131-3.

Коментарии не имеют синтактическое или семантическое значение, но они являются важной частью документации программы. Коментарий можно записать в программе там, где можно записать знак пробела. При переводе эти последовательности игнорируются, поэтому они могут содержать также знаку национальной азбуки. Программа перевода различает два вида коментариев :

- *общие коментарии*
- *построчные коментарии*

Общие коментарии - это последовательности знаков начинающиеся парой знаков **(*** и законченные парой знаков ***)**. Это позволяет проводить запись всех необходимых типов коментариев, как это указано на примере.

Построчные коментарии - это последовательности знаков начинающиеся парой знаков **//** и заканчиваются в конце строчки. Преимуществом построчных коментариев является возможность их погружения в общие коментарии (см. строчки с определенными переменными **Aux1** и **Aux2** в последующем примере, которые будут считаться коментарием и не будут переводиться программой перевода).

Пример 3.1 Коментарии

```
(*************************************************
   это демонстрация
   многострочного коментария
*************************************************)

VAR_GLOBAL
  Start,              (* общий коментарий, напр.: кнопка СТАРТ *)
  Stop      : BOOL;   (* кнопка СТОП *)
  Aux       : INT;    // построчный коментарий

(*
  Aux1    : INT;     // погруженный построчный коментарий
  Aux2    : INT;
*)

END_VAR
```

3.1.1 Идентификаторы

Идентификатор - это последовательность букв (маленьких или больших), цифр и подчеркивающих знаков, которая используется для названия следующих элементов :

- *названия постоянных величин*
- *названия переменных*
- *названия производных типа данных*
- *названия функций, функциональных блоков и программ*
- *названия задач*

Идентификатор должен начинаться буквой или подчеркивающим знаком и не должен содержать пробелы. Знаки национальной азбуки (буквы с диактрическими знаками) неразрешается применять в идентификаторах. Расположение подчеркивающего знака важно, напр. „BF_LM“ и „BFL_M“ - это два различных идентификатора. Больше подчеркивающих знаков идущих друг за другом не разрешается. Величина букв в идентификаторе не имеет значение. Например запись „motor_off“ эквивалентна записи „MOTOR_OFF“ или же „Motor_Off“. Если „motor_off“ будет переменной, то все указанные записи будут обозначать ту же переменную.

Максимальная длина идентификатора составляет 64 знаков.

Табл.1 Примеры действительных и недействительных идентификаторов

Действительные идентификаторы	**Недействительные идентификаторы**
XH2	2XH
MOTOR3F, Motor3F	3FMOTOR
Motor3F_Off, Motor3F_OFF	MOTOR3F__Off
SQ12	SQ$12
Delay_12_5	Delay_12.5
Delay	Продлева
_3KL22	__3KL22
KM10a	KM 10a

Пример 3.2 Идентификаторы

```
TYPE
  _Phase          : ( wye, delta);
END_TYPE

VAR_GLOBAL CONSTANT
  _3KL22          : REAL := 3.22;
END_VAR

VAR_GLOBAL
  SQ12  AT %X0.0  : BOOL;
  KM10a AT %Y0.0  : BOOL;
  XH2             : INT;
END_VAR

FUNCTION_BLOCK MOTOR3F
  VAR_INPUT
    Start         : BOOL;
  END_VAR
  VAR
    Delay_12_5    : TIME;
    Status        : _Phase;
  END_VAR
  VAR_OUTPUT
    Motor3F_Off   : BOOL;
  END_VAR

END_FUNCTION_BLOCK
```

3.1.2 Литеральные константы

Литеральные константы предназначены для прямого представления значений переменных.

Литеральные константы можно разделить на три группы:

- *цифровые литеральные константы*
- *последовательности знаков*
- *повременные литеральные константы*

Если хотим подчеркнуть Тип данных записываемой литеральной константы, можно запись литеральной константы начать названием типа данных следуемый знаком # (напр. **`REAL#12.5`**). В случае повременных литеральных констант указание типа обязательно (напр. **`TIME#12h20m33s`**).

3.1.2.1 Цифровые литеральные константы

Цифровая литеральная константа определяется как число (постоянная величина) в десятичной схеме или в схеме с другим основанием чем десять (напр. двоичные, восьминные и шеснадцатиместные цифры). Цифровые литеральные константы разделяем на литеральные константы «integer» и литеральные константы «rea». Простое подчеркивание, установленное между цифрами цифровой литеральной константы на его значение не оказывает влияние, разрешается для улучшения разборчивости. Примеры различных цифровых литеральных констант указаны в Табл.3.2 .

Табл.2 Примеры цифровых литеральных констант

Описание	**Цифровая литеральная константа - пример**	**Примеч.**
Литеральная константа «Integer»	14 INT#–9 12_548_756	 -9 12 548 756
Литеральная константа «Real»	–18.0 REAL#8.0 0.123_4	 0,1234
Литеральная константа «Real» с показателем степени	4.47E6 652E–2	4 470 000 6,52
Литеральная константа с основой 2	2#10110111	183 десятичные
Литеральная константа с основой 8	USINT#8#127	87 десятичные
Литеральная константа с основой 16	16#FF	255 десятичные
литеральная константа «Bool» FALSE TRUE	 FALSE BOOL#0 TRUE BOOL#1	 0 1

Пример 3.3 Цифровые литеральные константы

```
VAR_GLOBAL CONSTANT
  Const1          : REAL  := 4.47E6;
  Const2          : LREAL := 652E-2;
END_VAR

VAR_GLOBAL
  MagicNum        : DINT  := 12_548_756;
  Amplitude       : REAL  := 0.123_4;
  BinaryNum       : BYTE  := 2#10110111;
  OctalNum        : USINT := 8#127;
  HexaNum         : USINT := 16#FF;
  LogicNum        : BOOL  := TRUE;
END_VAR

FUNCTION Parabola : REAL
  VAR_INPUT
    x,a,b,c   : REAL;
  END_VAR

  IF a <> 0.0 THEN
    Parabola := a*x*x + b*x + c;
  ELSE
    Parabola := 0.0;
  END_IF;
END_FUNCTION


PROGRAM ExampleLiterals
  VAR
    x,y    : REAL;
  END_VAR

  y := Parabola(x := x, a := REAL#2.0, b := Const1, c := 0.0 );
END_PROGRAM
```

3.1.2.2 Литеральные константы последовательности знаков

Последовательность знаков – это последовательность без знаков (пустая последовательность) или более знаков, которая начата и закончена простыми кавычками (‘). Примеры: ‘‘ (пустая последовательность), ‘темпетарута‘ (заполненная последовательность длиной семь, содержащая слово температура).

Знак доллара - $, используется как префикс, который позволяет введение специальных знаков в последовательности. Специальные знаки, которые не печатаются, используются напр. для обработки по формату текста для принтера или на дисплей. Если знак доллара находится перед двумя шестнадцатиместными цифрами, последовательность интерпретируется как шестнадцатиместная репрезентация восьмибитового кода знака. Напр. последовательность ‘$0D$0A‘ понимается как репрезентация двух кодов, а именно - 00001101 и 00001010. Первый код представляет в ASCII таблице знак Enter, (CR, десятичные 13) а второй код - интервал между строками (LF, десятичные 10).

Литеральные константы последовательности знаков, т. наз. стринги, используются напр. для обмена текстами между различными ПЛК или между ПЛК и следующими

компонентами системы автоматизации, или при программировании текстов, которые изображаются на блоках управления или панелях оператора.

Табл.3 Специальные знаки в последовательностях

Запись	Значение
$$	Знак доллара
$'	Знак простой апостроф
$L или $l	Знак Line feed (16#0A)
$N или $n	Знак New line
$P или $p	Знак New page
$R или $r	Знак Carriage return (16#0D)
$T или $t	Знак табулятора (16#09)

Табл.4 Примеры литеральных констант последовательности знаков

Пример	Примечание
''	Пустая последовательность, длина 0
'temperature'	Заполненная последовательность, длина 11 знаков
'Character $'A$''	Последовательность содержащая ковычки (Character 'A')
' End of text $0D$0A'	Последовательность законченная знаками CR и LF
' Price is 12$$'	Последовательность содержащая знак $
'$01$02$10'	Последовательность содержащая 3 знака с кодами 1,2 и 16

Пример 3.4 Последовательности знаков

```
PROGRAM ExampleStrings
  VAR
    message     : STRING := '';  //  пустая последовательность
    value       : INT;
    valid       : BOOL;
  END_VAR

  IF valid THEN
    message := 'Temperature is ';
    message := CONCAT(IN1 := message, IN2 := INT_TO_STRING(value));
    message := message + ' [C]';
  ELSE
    message := 'Temperature is not available !';
  END_IF;
  message := message + '$0D$0A';
END_PROGRAM
```

3.1.2.3 Литеральные константы времени

При управлении в сущности нам нужны два различных типа данных, которые каким-либо способом связаны со временем. В первую очередь это ***информация о продолжительности***, т.е. о времени, которое истекло или должно истечь в связи с каким-либо событием. Во-вторых - это информация об „абсолютном времени“, состоящем из *данных и* согласно khondáře (Данные) и из *повременных информаций в рамках одного дня, т. наз.* ***дневного времени*** (Time of Day). Эта информация о времени может использоваться для синхронизации начала или конца управляемого события принимая во внимание абсолютные рамки времени. Примеры повременных литеральных констант в Табл.3.6.

Продолжительность. Литеральная константа времени на период продолжительности вводится некоторым из ключевых слов **`T#`**, **`t#`**, **`TIME#`**, **`time#`**. Сама информация о времени выражена в повременных единицах: часы, минуты, секунды и милисекунды. Сокращения для отдельных частей повременной информации указаны в Табл.3.5. Они могут выражаться маленькой и большой буквой.

Табл.5 Сокращения для повременных информаций

Сокращение	Значение
мс, MS	Милисекунды (Miliseconds)
с, S	Секунды (Seconds)
м, M	Минуты (Minutes)
ч, H	Часы (Hours)
д, D	Дни (Days)

Дневное время и дата. Репрезентация информации о данных и времени в рамках дня такая же как в ISO 8601. Префикс может быть кратким или длинным. Ключевые слова для даты - **`D#`** или **`DATE#`**. Для информации о времени в рамках одного дня используются ключевые слова **`TOD#`** или **`TIME_OF_DAY#`**. Для совокупности информации о „абсолютном времени” ключевые слова **`DT#`** или **`DATE_AND_TIME#`**. Величина букв опять не имеет значения.

Табл.6 Примеры различных повременных литеральная констант

Описание	Примеры
Продолжительность	T#24ms, t#6m1s, t#8.3s t#7h_24m_5s, TIME#416ms
Дата	D#2003-06-21 DATE#2003-06-21
Дневное время	TOD#06:32:15.08 TIME_OF_DAY#11:38:52.35
Дата и дневное время	DT#2003-06-21-11:38:52.35 DATE_AND_TIME#2003-06-21-11:38:52.35

Пример 3.5 Литеральные константы времени

```
VAR_GLOBAL
  myBirthday          : DATE := D#1982-06-30;
  firstManOnTheMoon   : DT   := DT#1969-07-21-03:56:00;
END_VAR

PROGRAM ExampleDateTime
  VAR
    coffeeBreak  : TIME_OF_DAY := TOD#10:30:00.0;
    dailyTime    : TOD;
    timer        : TON;
    startOfBreak : BOOL;
    endOfBreak   : BOOL;
  END_VAR

  dailyTime    := TIME_TO_TOD( GetTime());
  startOfBreak := dailyTime > coffeeBreak AND dailyTime < TOD#12:00:00;
  timer(IN := startOfBreak, PT := TIME#15m, Q => endOfBreak);
END_PROGRAM
```

3.2 Типы данных

Для программирования на одном из языков согласно норме IEC 61 131-3 определены т. наз. ***элементарные***, предварительно определенные типы данных, (Элементary данныха types), также определены ***родовые*** типы данных (Generic data type) для родвственных групп типов данных. И наконец в распоряжении имеется механизм, которым пользователь может создавать собственные ***производные*** *(пользовательские)* типы данных (Derived data type, Type definition).

3.2.1 Элементарные типы данных

Элементарные типы данных характеризуются шириной данных (количеством битов) а также диапазоном значений. Перечень поддерживаемых типов данных указан в Табл.3.7.

Табл.7 Элементарные типы данных

Ключевое слово	По-английски	Тип данных	Битов	Диапазон значений
BOOL	Boolean	Цифра Боола	1	0,1
SINT	Short integer	Короткое целое число	8	–128 127
INT	Integer	Целое число	16	–32 768 - +32 767
DINT	Double integer	Целое число, двойная длина	32	–2 147 483 648 - +2 147 483 647
USINT	Unsigned short integer	Краткое целое число без знака	8	0 až 255
UINT	Unsigned integer	Целое число без знака	16	0 - 65 535
UDINT	Unsigned double integer	Целое число без знака, двойная длина	32	0 - +4 294 967 295
REAL	Real (single precision)	Число в бегущей запятой (простая точность)	32	±2.9E-39 - ±3.4E+38 Согласно IEC 559
LREAL	Long real (double precision)	Число в бегущей запятой (двойная точность)	64	Согласно IEC 559
TIME	Duration	Продолжительность	24d 20:31:23.647	
DATE	Data (only)	Дата	От 1.1.1970 00:00:00	
TIME_OF_DAY или TOD	Time of day (only)	Дневное время	24d 20:31:23.647	
DATE_AND_TIME или DT	Data and time of day	„Абсолютное время“	От 1.1.1970 00:00:00	
STRING	String	Последовательность	Макс.255 знаков	
BYTE	Byte(bit string of 8 bits)	Последовательность 8 битов	8	Диапазон не декларирован
WORD	Word (bit string of 16 bits)	Последовательность 16 битов	16	Диапазон не декларирован
DWORD	Double word (bit string of 32 bits)	Последовательность 32 битов	32	Диапазон не декларирован

Инициализация элементарных типов данных

Важным принципом при программировании согласно норме IEC 61 131-3 является то, что все переменные в программе имеют инициализирующее (начальное) значение. Если пользователь не укажет другое, будет переменная инициализирована неявным (предварительно определенным, дефо) значением согласно используемому типу данных. Предварительно определенное начальное значение для элементарных типов данных – обычно ноль, у данных - D#1970-01-01. Совокупность предварительно определенных начальных значений указана в Табл.3.8.

Табл.8 Предварительно определенные начальные значения для элементарных типов данных

Тип данных	Исходное значение (Initial Value)
BOOL, SINT, INT, DINT	0
USINT, UINT, UDINT	0
BYTE, WORD, DWORD	0
REAL, LREAL	0.0
TIME	T#0s
DATE	D#1970-01-01
TIME_OF_DAY	TOD#00:00:00
DATE_AND_TIME	DT#1970-01-01-00:00:00
STRING	’ ’ (пустой стрингер)

3.2.2 Родовые типы данных

Родовые типы данных выражают всегда целую группу (род) типов данных введены префиксом **ANY**. Напр. под запись **ANY_BIT** понимаются все типы данных указанные в следующем перечне: **DWORD, WORD, BYTE, BOOL**. Перечень родовых типов данных указан в Табл.3.9. Названия родовых типов данных начинающиеся **ANY_** согласно IEC не являются ключевыми словами. Предназначены только для обозначения группы у типов с такими же свойствами.

Табл.9 Перечень родовых типов данных

<table>
<tr><th colspan="5">ANY</th></tr>
<tr><th>ANY_BIT</th><th colspan="3">ANY_NUM</th><th>ANY_DATE</th><th rowspan="3">TIME
STRING</th></tr>
<tr><td rowspan="2">BOOL
BYTE
WORD
DWORD</td><th colspan="2">ANY_INT</th><th>ANY_REAL</th><td rowspan="2">DATE
DATE_AND_TIME
TIME_OF_DAY</td></tr>
<tr><td>INT
SINT
DINT</td><td>UINT
USINT
UDINT</td><td>REAL
LREAL</td></tr>
</table>

3.2.3 Производные типы данных

Производные типы, то есть типы специфицированные производителем или пользователем, могут декларироваться с помощью текстовых конструкций **TYPE**...**END_TYPE**. Названия новых типов, их типы данных или же их инициализационные значени указаны в рамках данной конструкции. Эти производные типы данных потом можно использовать совместно с элементарными типами данных в декларациях переменных. Определение производного типа данных глобальное, т.е. может использоваться в любых частях программы для ПЛК. Производные типы данных наследуют свойства типа, из которого были образованы.

3.2.3.1 Простые производные типы данных

Простые производные типы данных исходят из прямо элементарных типов данных. Чаще всего причиной введения нового типа данных бывает отличающееся инициализационное значение нового типа, которое можно присвоить прямо в декларацию типа с помощью присваивающего оператора „:=“. Если не указано инициализационное значение в декларации нового типа, наследуется новый тип инициализационного значения типа, из которого был модифицирован.

К простым производным типам данных принадлежит также перечень значений (Enumerated данныха type). Как правило, он используется для названия свойств или же варианта вместо назначения цифрового кода к каждому варианту, что улучшает четкость программы. Инициализационное значение типа, указанного в перечне – это всегда значение первого элемента, указанного в перечне.

Пример 3.6 Пример простых производных типов данных

```
TYPE
  TMyINT       : INT;                  // простой производный тип данных
  TRoomTemp    : REAL := 20.0;         // новый тип данных с инициализацией
  THomeTemp    : TRoomTemp;
  TPumpMode    : ( off, run, fault); // новый тип декларированный
                                       // в перечне значений
END_TYPE

PROGRAM SingleDerivedType
  VAR
    pump1Mode    : TPumpMode;
    display      : STRING;
    temperature  : THomeTemp;
  END_VAR

  CASE pump1Mode OF
    off   : display := 'Pump no.1 is off';
    run   : display := 'Pump no.1 is running';
    fault : display := 'Pump no.1 has a problem';
  END_CASE;
END_PROGRAM
```

Простые переменные, которые имеют декларированный пользовательский тип, могут использоваться там, где может использоваться переменная с „родительским“ типом. То есть напр. переменная „**temperature**“ (температура) из примеру 3.6 может использоваться там, где могут использоваться переменные типа **REAL**. Это правило может применяться рекурсивно.

Новым типом данных может быть также ***массив*** (Array) или ***структура*** (Structure).

3.2.3.2 Производные типы данных массив

Одномерные массивы

Массив – это упорядоченный ряд элементов одинакового типа данных. Каждый элемент массива имеет закрепленный индекс, с помощью которого можно приступить к элементу. Иначе говоря, значение индекса определяет, с каким элементом массива будем работать. Индекс может принимать только значения в диапазоне определения массива. Если значение индекса превысит декларированный размер массива, то будет объявлена т. наз. ошибка «Run-time» (ошибка, сигнализированная во время работы системы). Одномерный массив - это массив, котортое имеет только один индекс, как это указано на Рис. 3.1.

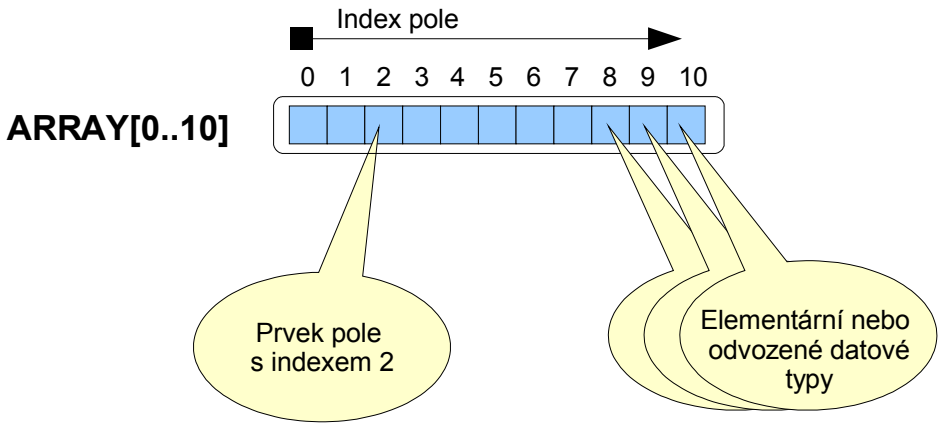


Рис. 4 Одномерный массив

Элемент массива может быть элементарного или производного типа данных. Массивы инстанций POU до сих пор не поддерживаются. Декларации производного типа данных массива указаны на примере 3.7. Декларация проводится с помощью ключевого слова **ARRAY**, зав которым следует размер массива в квадратных скобках. Размер массива задает диапазон допустимых индексов. За размером массива потом указано ключевое слово **OF** с введением типа данных для элемента массива. Индекс первого элемента массива должен быть положительное число или ноль. Отрицательные индексы не допускаются. Максимальная величина массива ограничена диапазоном памяти переменных в системе управления.

Декларация типа массива может также содержать инициализацию отдельных элементов (см. тип **TByteArray** и **TRealArray**). Инициализационные значения указаны в декларации типа массива за оператором присвоения „:=“ в квадратных скобках. Если указано меньше инициализационных значений чем отвечает размеру массива, то элементы с неуказанными инициализационными значениями инициализированы на начальное значение согласно используемому типу данных. Для инициализации большого количества элементов массива одинакового значения можно использовать т. наз. повторитель. В данном случае на месте для инициализационного значения указывыается количество повторений последуемых инициализационных значений в круглых скобках. Например записью **25(-1)** будем инициализировать 25 элементов массива значений -1.

Пример 3.7 Производный тип данных одномерный массив

```
TYPE
  TVector     : ARRAY[0..10]  OF INT;
  TByteArray  : ARRAY[1..10]  OF BYTE := [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
  TRealArray  : ARRAY[5..9]   OF REAL := [ 11.2, 12.5, 13.1];
  TBigArray   : ARRAY[1..999] OF SINT := [ 499( -1), 0, 499( 1)];
END_TYPE

PROGRAM Example1DimArray
  VAR
    index      : INT;
    samples    : TVector;
    buffer     : TByteArray;
    intervals  : TRealArray;
    result     : BOOL;
  END_VAR

  FOR index := 0 TO 10 DO
    samples[index] := 0;            // удалить все образцы
  END_FOR;
  result := intervals[5] = 11.2;    // TRUE
  result := intervals[8] = 0.0;     // TRUE
END_PROGRAM
```

Многоразмерные массивы

Многоразмерные массивы - это массивы, где для доступа к одному элементу массива нужен более чем один индекс. Поле имеет потом два и более размеров, которые могут для каждый индекса отличаться. Двухразмерный массив можно представить себе как матрицу элементов как это указано на Рис.3.2. Элементы многоразмерных массивов могут быть элементарными или производными типами данных, так же как у массивов одномерных.

Программа перевода в среде «Mosaic» поддерживает работу максимально с четырехразмерными массивами.

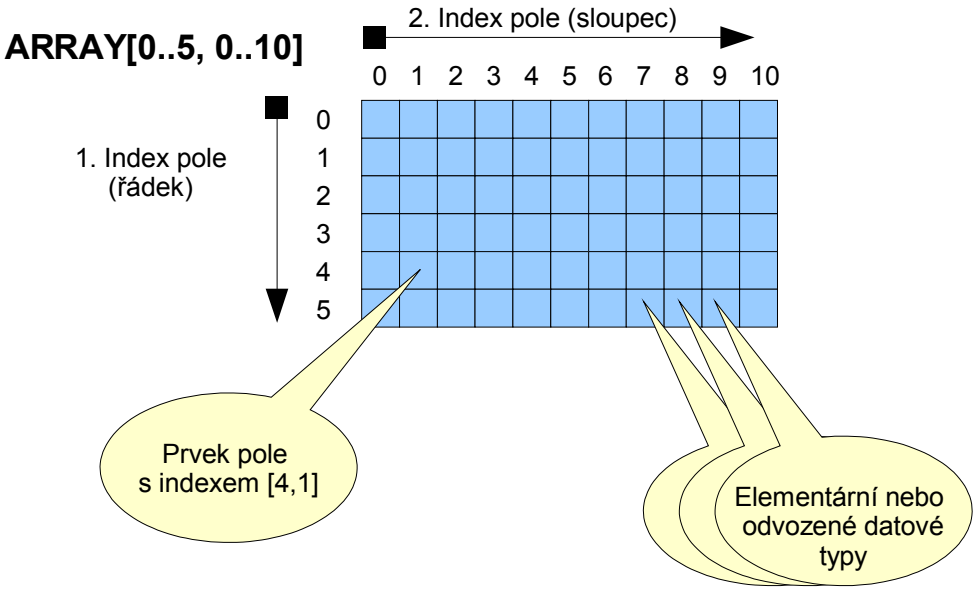


Рис. 5 Двухразмерный массив

Инициализация у многоразмерных массивов проводится так же как для одномерных массивов с учетом того, что сначала инициализированы все элементы для первого размера (т.е. например элементы массив[0,0], массив[0,1], массив[0,2] – массив[0,n]) и после этого действия повторяются для следующего значения первого индекса. Так же как последнее инициализированы элементы массив[m,0], массив[m,1], массив[m,2] и наконец массив[m,n]. При инициализации многоразмерных массивов можно также использовать повторитель для инициализации болешего количества элементов одновремено, как это указано на примере 3.8 у типа **`TThreeDimArray1`**. В коментарии указана та же декларация без использования повторителей.

Пример 3.8 Производный тип данных многоразмерный массив

```
TYPE
  TTwoDimArray   : ARRAY [1..2,1..4] OF SINT := [ 11, 12, 13, 14,
                                                  21, 22, 23, 24 ];

  TThreeDimArray : ARRAY [1..2, 1..3, 1..4] OF BYTE :=
                                                [ 111, 112, 113, 114,
                                                  121, 122, 123, 124,
                                                  131, 132, 133, 134,
                                                  211, 212, 213, 214,
                                                  221, 222, 223, 224,
                                                  231, 232, 233, 234 ];

  TThreeDimArray1 : ARRAY [1..2, 1..3, 1..4] OF BYTE :=
                                                [ 4(11), 4(12), 4(13),
                                                  4(21), 4(22), 4(23) ];
(*
  TThreeDimArray1 : ARRAY [1..2, 1..3, 1..4] OF BYTE :=
                                                [ 11, 11, 11, 11,
                                                  12, 12, 12, 12,
                                                  13, 13, 13, 13,
                                                  21, 21, 21, 21,
                                                  22, 22, 22, 22,
                                                  23, 23, 23, 23 ];
```

```
*)
END_TYPE


PROGRAM ExampleMultiDimArray
  VAR
    twoDimArray   : TTwoDimArray;
    threeDimArray : TThreeDimArray;
    element       : BYTE;
    result        : BOOL;
  END_VAR

   result  := twoDimArray[1, 4] = 14;     // TRUE
   element := threeDimArray[ 2, 1, 3];    // element = 213
END_PROGRAM
```

Так же как производного типа данных массива можно также декларировать прямо переменную типа массива, как это указано в гл.3.

3.2.3.3 Производный тип данных структура

Структуры - это типы данных, которые содержат так же как массива более элементов (статей). Но в отличие от массивов все элементы в структуре не должны быть одного типа данных. Структуру можно вывести как из элементарных так из уже производных типов данных. Структура может быть построена иерархически, что означает, что элементом структуры может быть уже определенная структура. Ситуация описана на Рис.3.3.

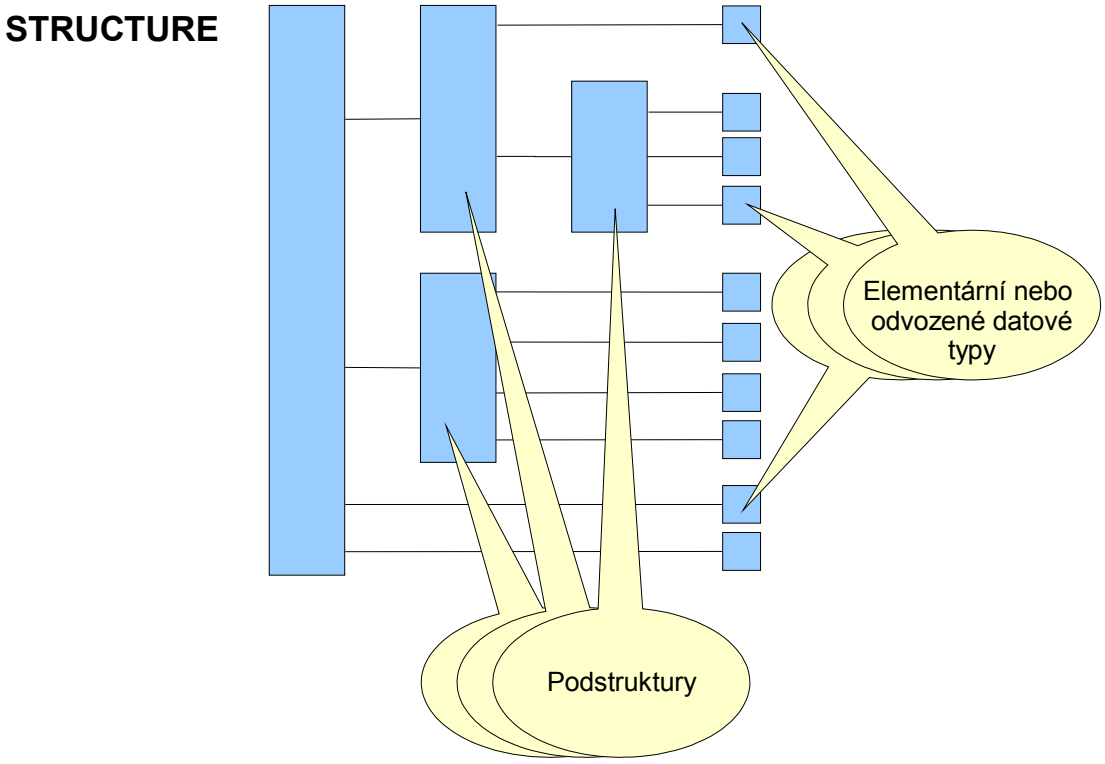


Рис. 6 Структура

Определение нового типа данных структура проводится с помощью ключевых слов **`STRUCT`** и **`END_STRUCT`** в рамках конструкций **`TYPE`** … **`END_TYPE`**. Внутри **`STRUCT`** … **`END_STRUCT`** указаны названия отдельных элементов структуры и их типы данных. Так же как это было в случае предыдущих производных типа данных, можно и структуры инициализировать введением значений у элемента за знаком „**`:=`**“.

Если создать переменную типа структуры, то доступ к отдельным элементам структуры будет „**`названиеПеременной.названиеЭлемента`**“ как это указано на примере 3.9.

Пример 3.9 Производный тип данных структура

```
TYPE
  Tproduct :
    STRUCT
      name         : STRING := 'Engine M11';
      code         : UINT;
      serie        : DINT;
      serialNum    : UDINT;
      expedition   : DATE;
    END_STRUCT;
END_TYPE

PROGRAM ExampleStruct
  VAR
    product     : Tproduct;
    product 1    : Tproduct;
  END_VAR

  product.code       := 700;
  product.serie      := 0852;
  product.serialNum  := 12345;
  product.expedition := DATE#2002-02-13;
END_PROGRAM
```

Инициализация переменных типа структуры проводится с помощью названий элементов структуры при декларации переменных. Разница между инициализацией типа данных структура и инициализацией переменной типа структура указана на примерах 3.9 и 3.10. Функциональная разница очевидна. В то время как на примере 3.9 будет каждая переменная типа **TProduct** элемент **name** автоматически инициализирована на значении **'Engine M11'**, на примере 3.10 неявна инициализация элемента **name** пустая последовательность заменена последовательностью **'Engine M11'** только в случае переменной **product**.

Пример 3.10 Инициализация структуры переменного типа

```
TYPE
  TProduct :
    STRUCT
      name         : STRING;
      code         : UINT;
      serie        : DINT;
      serialNum    : UDINT;
      expedition   : DATE;
    END_STRUCT;
END_TYPE

PROGRAM ExampleStruct
  VAR
    product: TProduct := ( name := 'Engine M11');
    product1    : TProduct;
  END_VAR
```

```
  product.code       := 700;
  product.serie      := 0852;
  product.serialNum  := 12345;
  product.expedition := DATE#2002-02-13;
END_PROGRAM
```

3.2.3.4 Комбинация структур и массивов в производных типах данных

Массивы и структуры можно в определении производных типов данных произвольно комбинировать. Элементом структуры могут быть массивы или элементом массива может быть структура как это указано на примере 3.11.

Пример 3.11 Структура как элемент массива

```
VAR_GLOBAL CONSTANT
  NUM_SENSORS   : INT := 12;
END_VAR

TYPE
  TLimit :
    STRUCT
      low         : REAL := 12.5;
      high        : REAL := 120.0;
    END_STRUCT;

  TSensor :
    STRUCT
      status      : BOOL;
      pressure    : REAL;
      calibration : DATE;
      limits      : TLimit;
    END_STRUCT;

   TSensorsArray  : ARRAY[1..NUM_SENSORS] OF TSensor;
END_TYPE

PROGRAM ExampleArrayOfStruct
  VAR
    sensors : TSensorsArray;
    i       : INT;
  END_VAR

  FOR i := 1 TO NUM_SENSORS DO
    IF (sensors[i].pressure >= sensors[i].limits.low)  AND
       (sensors[i].pressure <= sensors[i].limits.high)
    THEN
       sensors[i].status := TRUE;
    ELSE
       sensors[i].status := FALSE;
    END_IF;
  END_FOR;
END_PROGRAM
```

3.2.4 Тип данных «pointer»

Тип данных «pointer» является расширением нормы IEC 61 131. Другими словами «pointer» не определеется упомянутой нормой и программы, в которых будет этот тип данных использоваться, невозможно будет использовать для ПЛК программированные в другой среде, а только в «Mosaic».

Причиной того, что тип данных между нормированными типами данных отсутствует является однозначно безопасность программирования. Неправильное использование типа «pointer» может привести к полному сбою программы, что при управлении технологией недопустимо. При этом ошибку невозможно обнаружить даже в фазе перевода программы или во время работы программы. Опыт в языке C, где Тип данных «pointer» часто используется, доказывают, что большая часть некорректного поведения программы причинена именно неправильной работой с «pointer». С другой стороны существует наверное очень мало программ, написанных на языке C, где Тип данных «pointer» не используется. Что из этого следует? «Pointer» может быть очень хорошим слугой, но и злым господином. Ответственность за правильность программы с «pointer» лежит только на программаторе, так как средства, которые ему в других ситуациях помогают обнаруживать ошибки (программа перевода, стандартный контроль, контроль «run-time» и т.п.) в случае «pointer» не действуют. Преимуществом «pointer» является более высокая эфективность программирования. «Pointer» позволяет в ряде случаев более короткие и благодаря этому более быстрые программы, главным образом если в программе используются структуры, массивы и их комбинация. И наконец последней причиной просто является факт, что существуют проблемы, которые можно эффективно решить только с использованием «pointer».

«Pointer» собственно говоря, является показателем переменной, которая может быть как элементарного так и производного типа. Декларация «pointer» проводится с помощью ключевого слова **`PTR_TO`** за которым следует название типа данных, на который «pointer» показывает. Тип данных «pointer» можно использовать там, где можно использовать элементарные типы данных. POU не оказывает поддержку «Pointer».

Переменная типа «pointer» содержит собственно говоря адрес какой-либо другой переменной. С «pointer» можно работать двумя способами. Можно изменять его значение (повышать, снижать и т.д.) и таким образом менять то, на какую переменную показывает «pointer». Потом можно само собой работать с значением переменной, на которую указывает «pointer». Первая упомянутая операция называется арифметика «pointer», вторая операция в большинстве случаев обозначается как определение «pointer».

Арифметика «Pointer»

Первая операция, которую должна каждая программа с «pointer» провести - заполнить адрес переменной, на которую будет «pointer» указывать. Неявная инициализация типа данных «pointer» составляет -1, что собственно говоря означает, что «pointer» не показывает ни на одну переменную. Это также единственный случай, который может быть уловлен контролем «run-time» системы управления и сигнализирован как ошибка.

Инициализация «pointer», т.е. его заполнение адресом переменной, на которую будет указывать, проводится с помощью системной функции **`ADR()`**. Параметром данной функции является имя переменной, адрес которой хотим заполнить в «pointer». Например запись **`myPtr := ADR( myVar)`** заполнит адрес переменной **`myVar`** в «pointer» **`myPtr`**. Другими словами «pointer» **`myPtr`** будет указывать на переменную **`myVar`**.

С переменной типа «pointer» можно проводить арифметические операции с целью изменения адреса переменной. В выражениях можно тип **PTR_TO** комбинировать с типами данных **ANY_INT**. Если переменная **myVar** будет расположена в памяти по адресу %MB100 и переменная **yourVar** будет лежать в памяти по адресу %MB101, то выражение **myPtr := myPtr + 1** увеличивает значение «pointer» на 1, поэтому «pointer» будет указывать на переменную **yourVar** (вместо исходной **myVar**). Само собой только при предположении, что обе переменные типа данных, который занимает в памяти один бит. Арифметика в случае типа **PTR_TO** работает обычно, что означает, что после прочтения значений 15 будет «pointer» всегда указывать 15 битов дальше в памяти.

Дереференция «pointer»

Дереференция «pointer» - это операция, которая позволяет работать с переменной, на которую «pointer» указывает. Для дереференции используется знак ^. Поэтому запись **value := myPtr^** заполнит в переменные **value** значения переменных **myVar** (само собой при предположении, что **myPtr** указывает на **myVar** и переменная **value** того же типа как переменная **myVar**).

Пример 3.12 «Pointer»

```
VAR_GLOBAL
  arrayINT   : ARRAY[0..10] OF INT;
END_VAR

PROGRAM ExamplePtr
  VAR
    intPTR   : PTR_TO INT;
    varINT   : INT;
  END_VAR

  intPTR  := ADR( arrayINT[0]);          // init ptr
  intPTR^ := 11;                         // arrayINT[0] := 11;
  intPTR  := intPTR + sizeof( INT);      // ptr to next item
  intPTR^ := 22;                         // arrayINT[1] := 22;
  intPTR  := intPTR + sizeof( INT);      // ptr to next item
  varINT  := intPTR^;                    // varINT := arrayINT[2];
END_PROGRAM
```

Пример 3.12 использует для повышения адреса, на который «pointer» **intPTR** указывает, функцию **sizeof()**. Эта функция возвращает количество битов заданного типа данных или переменной.

Следующий пример указывает, как легко можно при работе с «pointer» сделать ошибку. Программа такая же как в примере 3.12 и отличается только инициализацией «pointer» **intPTR**. В то время как в первом случае инициализация проводится в каждом цикле командой **intPTR := ADR(arrayINT[0])**, во втором примере «pointer» инициализируется уже в декларации переменные **intPTR : PTR_TO INT := ADR(arrayINT[0])**. Это приведет к тому, что первый цикл программы после повторного запуска системы будет хотя и правильный, но уже во втором цикле будет «pointer» начинаться с адресом элемента **arrayINT[2]** вместо **arrayINT[0]**. При циклическом проведении программы это означает, что программа в примере 3.13 в кратко времени опишет всю память переменных значений **INT#11** и

INT#22, что нам еще не нужно. Поэтому необходимо помнить, что использование «pointer» требует всегда повышенную осторожность.

Пример 3.13 Ошибочная инициализация «pointer»

```
VAR_GLOBAL
  arrayINT    : ARRAY[0..10] OF INT;
END_VAR

PROGRAM ExamplePtrErr
  VAR
    intPTR    : PTR_TO INT := ADR( arrayINT[0]);
    varINT    : INT;
  END_VAR

                                        // for 1st cycle only !!!
  intPTR^ := 11;                        // arrayINT[0] := 11;
  intPTR  := intPTR + sizeof( INT);     // ptr to next item
  intPTR^ := 22;                        // arrayINT[1] := 22;
  intPTR  := intPTR + sizeof( INT);     // ptr to next item
  varINT  := intPTR^;                   // varINT := arrayINT[2];
END_PROGRAM
```

3.3 Переменные

Согласно норме IEC 61 131-3 ***переменные*** в сущности это средство для идентификации информационных объектов, их содержание может меняться, то есть данные приделенные к входам, выходам или памяти ПЛК. Переменная может быть декларирована некоторыми элементарными типами данных некоторых производных (пользовательских) типов данных.

Этим программирование согласно IEC 61 131-3 приближается к обычным. Вместо используемых раньше адресов технического обеспечения или символов здесь определены переменные так, как они используются в вышших программных языках. Переменные - это идентификаторы (названия) приделенные программатором, которые предназначены в сущности для бронирования места в памяти и содержат значения данных программ.

3.3.1 Декларация переменных

Каждая декларация POU (то есть каждая декларация программы, функции или функционального блока) для начала должна иметь как минимум одну декларационную часть, которая специфицирует типы данных переменных используемых в POU. Эта декларационная часть имеет вид текста и использует одно из ключевых слов **VAR**, **VAR_TEMP**, **VAR_INPUT**, **VAR_OUTPUT**. За ключевым словом **VAR** может быть на выбор приведен квалификатор **CONSTANT**. За приведенными ключевыми словами следует одна или более деклараций переменных разделенных точкой с запятой и законченная ключевым словом **END_VAR**. Составной частью декларации переменных может быть декларация их начальных (инициализационных) значений.

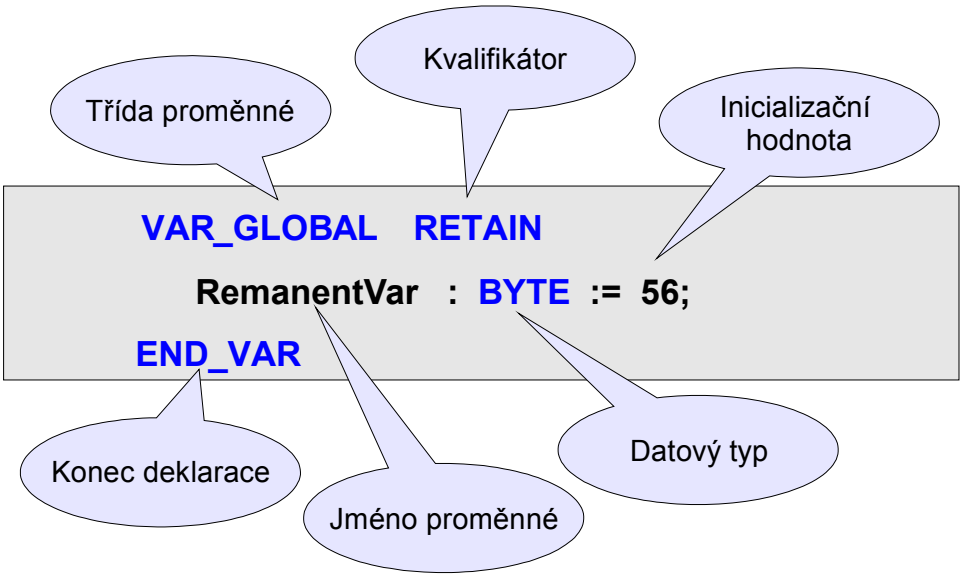


Рис. 7 Декларация переменных согласно IEC

Диапазон действия деклараций расположенных в декларационной части POU ***локальные*** для той программной организационной единицы, в которой декларация указана. Это означает, что декларированные переменные не будут доступны остальным POU кроме explicitního передачи параметров через переменные, которые были декларированы как ***входных переменных*** (**VAR_INPUT**) или же **выходные *переменные*** (**VAR_OUTPUT**). Единственным исключением из данного правила являются переменные, которые были декларированы как ***глобальные***. Эти переменные определены вне деклараций всех POU и начинаются ключевым словом **VAR_GLOBAL**. За ключевым словом **VAR_GLOBAL** может быть на выбор указан квалификатор **RETAIN** или **CONSTANT**.

3.3.1.1 Классы переменных

Класс переменной определяется ее использованием и диапазоном действительности (scope). С данной точки зрения можно переменные разделить на следующие группы:

- **глобальные переменные**
  - **VAR_GLOBAL** - незаавансированные переменные
  - **VAR_GLOBAL RETAIN** - заавансированные переменные
  - **VAR_GLOBAL CONSTANT** - постоянные величины
  - **VAR_EXTERNAL** - экстренные переменные
- **локальные переменные**
  - **VAR** - локальные переменные
  - **VAR_TEMP** - временные переменные
- **переменные для передачи параметров**
  - **VAR_INPUT** - входные переменные
  - **VAR_OUTPUT** - выходные переменные
  - **VAR_IN_OUT** - вход-выходные

Табл.10 Классы переменных

Класс переменной	Значение	Určení
VAR_INPUT	входные	Для передачи входных параметров в POU Эти переменные выделяются между остальными POU и из них также настраиваются
VAR_OUTPUT	выходные	Для передачи выходных параметров из POU Эти переменные выделяются между остальными POU, где можно проводить только их считывание Изменение значений данных переменных можно проводить только в рамках POU, в которые были переменные декларированы
VAR_IN_OUT	вход / выходные	Для непрямого доступа к переменным находящимся вне POU Переменные можно считывать и меняться их значение внутри и вне POU
VAR_EXTERNAL	глобальные	Переменные определенные в мнемокоде ПЛК
VAR_GLOBAL	глобальные	Переменные, которые доступны со всех POU
VAR	локальные	Вспомогательные переменные используемые в рамках POU Из остальных POU невидимы, то есть их можно считывать или же менять их значение только в рамках POU, в которых они декларированы Эти переменные могут сохранять значения и между отдельными вызовами соответствующей POU
VAR_TEMP	локальные	Вспомогательные переменные используемые в рамках POU Из остальных POU невидимы Эти переменные возникают при входе в POU и исчезают после окончания POU – не могут сохранять значения между двумя вызовами POU

Табл.11 Использование классов в отдельных POU

Класс переменной	PROGRAM	FUNCTION_BLOCK	FUNCTION	Вне POU
VAR_INPUT	да	да	да	нет
VAR_OUTPUT	да	да	нет	нет
VAR_IN_OUT	да	да	да	нет
VAR_EXTERNAL	да	да	да	нет
VAR_GLOBAL	нет	нет	нет	да
VAR	да	да	да	нет
VAR_TEMP	да	да	да	нет

3.3.1.2 Квалификаторы в декларации переменных

Квалификаторы позволяют определить дополнительно свойства декларированных переменных. Ключевое слово для квалификатора указывается за ключевым словом классы (**VAR,**...). В декларации переменных можно использовать следующие квалификаторы :

- **RETAIN** – заавансированные переменные (переменные, которые сохраняют значения также при выключении питания ПЛК)
- **CONSTANT** – постоянные значения величин (значение переменной не может быть изменено)
- **R_EDGE** – передний торец переменной
- **F_EDGE** – нисходящий торец переменной

Табл.12 Использование квалификаторов в декларации переменных

Класс переменной	Значение	RETAIN	CONSTANT	R_EDGE F_EDGE
VAR	локальные	нет	да	нет
VAR_INPUT	входные	нет	нет	да
VAR_OUTPUT	выходные	нет	нет	нет
VAR_IN_OUT	вход / выходные	нет	нет	нет
VAR_EXTERNAL	глобальные	нет	нет	нет
VAR_GLOBAL	глобальные	да	да	нет
VAR_TEMP	локальные	нет	нет	нет

3.3.2 Глобальные переменные

С точки зрения доступности можно переменные разделить на ***глобальные*** и ***локальные.***

Глобальные переменные - это такие переменные, которые доступны со всех POU. Их определение начинается ключевым словом **VAR_GLOBAL** и не указаны внутри какого-либо POU как это указано на примере 3.14. Глобальная переменная может быть размещена на конкретном адресе в памяти ПЛК с помощью ключевого слова **AT** в декларации переменных. Если ключевое слово **AT** отсутствует, необходимое место в памяти программы перевода будет приделено автоматически.

Если в декларации указан квалификатор **CONSTANT** речь идет о определеных переменных, значение которых точно определено декларацией и не возможно ее в программе изменить. Поэтому, собственно говоря это не переменные в полном смысле слова а постоянные величины. А если они элементарного типа данных, программа перевода им не приделяет какое-либо место в памяти, только в выражениях использует соответствующую постоянную величину.

Переменные классы **VAR_EXTERNAL** могут быть как глобальные так и локальные. Если декларация переменных данного класса указана внутри POU, речь идет о местной переменной, в противном случае - о глобальной переменной.

Пример 3.14 Декларация глобальных переменных

PROGRAM в мнемокоде:

```
#reg word mask                      ; декларация переменной в мнемокоде
P 0
        ld      $1111
        wr      mask
E 0
```

PROGRAM в языкеST:

```
VAR_EXTERNAL
  mask            : WORD;           // ссылка на переменную в мнемокоде
END_VAR

VAR_GLOBAL RETAIN
  maxTemp         : REAL;           // заавансированная переменная
END_VAR

VAR_GLOBAL CONSTANT
  PI        : REAL := 3.14159;      // постоянная величина
END_VAR

VAR_GLOBAL
  globalFlag      : BOOL;
  suma            : DINT := 0;
  temp   AT %XF10 : REAL;          // температура
  minute AT %S7   : USINT;
END_VAR

PROGRAM ExampleGlobal

  globalFlag := mask = 16#1111;                 // true
  максTemp := МАКС(IN1 := temp, IN2 := максTemp);
END_PROGRAM
```

3.3.3 Локальные переменные

Локальные переменные декларированы внутри POU и их действительность и видимость ограничена на ту POU, в которой они декларированы. В остальных POU их невозможно использовать. Декларация локальных переменных начинается ключевым словом **VAR** или **VAR_TEMP**.

Переменные декларированные в классе **VAR** - это т. наз. статические переменные. Этим переменным при переводе уделяется определенное место в памяти переменных, которые во время проведения программы не меняются. Это означает, что чем больше переменных в классе **VAR** определим, тем более памяти будет загружено. Следующим

важным свойством переменных классов **VAR** является то, что их значения сохраняются между двумя вызовами POU, в которых они декларированы.

Переменные декларированные в классе **VAR_TEMP** - это переменные, которые динамично созданы в момент, когда начинается расчет POU с соответствующей декларацией. В момент, когда расчет POU заканчивается, динамически выделенная память освобождена и переменные классы **VAR_TEMP** исчезают. Из этого следует, что декларация переменных в классе **VAR_TEMP** не влияет на расход памяти. Эти переменные также не могут сохранять значения между двумя вызовами POU, или после окончания POU перестают существовать.

Разница между переменными классов **VAR** и **VAR_TEMP** также в их инициализации. Переменные классы **VAR** инициализированы только при повторном запуске системы в то время как переменные классы **VAR_TEMP** инициализированы всегда при распределении памяти (т.е. при каждом начале расчета соответствующей POU). Указанные свойства указаны на следующем примере.

Пример 3.15 Декларация локальных переменных

```
PROGRAM ExampleLocal
  VAR
    staticCounter   : UINT;
    staticVector    : ARRAY[1..100] OF BYTE;
  END_VAR
  VAR_TEMP
    tempCounter     : UINT;
    tempVector      : ARRAY[1..100] OF BYTE;
  END_VAR

  staticCounter := staticCounter + 1;
  tempCounter   := tempCounter   + 1;
END_PROGRAM
```

Значения локальных переменных **staticCounter** при повторном вызове программы **ExampleLocal** будут плавно увеличиваться, так как каждый следующий вызов начнет расчет из значений **staticCounter** из прошлого вызова. И наоборот - значения переменных **tempCounter** будеут в конце программы **ExampleLocal** всегда 1 независимы от количества вызова программы, так как эта переменная возникнет и инициализирована значением 0 всегда при вызове программы **ExampleLocal**.

На примере 3.15 можно также указать разницу в расходе памяти переменных. Переменная **staticVector** занимает 100 битов памяти переменных, в то время как переменная **tempVector** совсем не влияет на величину занятой памяти переменных.

3.3.4 Входные и выходные переменные

Входные и выходные переменные предназначены для передачи параметров между POU. С помощью этих переменных мы можем определить входной и выходной интерфейс POU.

Для передачи параметров в направлении к POU предназначены переменные класса **VAR_INPUT** и речь идет о **входных** переменных. Для передачи параметров в направлении от POU предназначены переменные класса **VAR_OUTPUT** и речь идет о **выходных** переменных. Если представим себе напр. **функциональный блок** как интегрированный контур, то переменные **VAR_INPUT** представляют входные сигналы контура и переменные **VAR_OUTPUT** представляют его выходные сигналы.

Определение переменных типа **BOOL** в классе **VAR_INPUT** может быть дополнено квалификаторами **R_EDGE** и **F_EDGE**, которые позволяют детектировать передний или же нисходящий торец переменных. Переменные определенные с квалификатором **R_EDGE** будут принимать значение **true** только в случае, если состояние переменных изменяется из значения **false** на значение **true**. Такой переменной является переменная **in** в примере 3.16. Функциональный блок **FB_EdgeCounter** В данном примере будет считывать из переднего тоорца (изменения значения «false» на значение «true») входные переменные **in**.

Пример 3.16 Детекция переднего торца входных переменных

```
FUNCTION_BLOCK FB_EdgeCounter
  VAR_INPUT
    in          : BOOL R_EDGE;
  END_VAR
  VAR_OUTPUT
    count       : UDINT;
  END_VAR

  IF in THEN count := count + 1; END_IF;
END_FUNCTION_BLOCK


PROGRAM ExampleInputEdge
  VAR_EXTERNAL
    AT  %X0.0   : BOOL;
  END_VAR
  VAR
    edgeCounter : FB_EdgeCounter;
    howMany     : UDINT;
  END_VAR

  edgeCounter(in := %X0.0, count => howMany);
END_PROGRAM
```

Параметры передаваемые посредничеством входных и выходных переменных - это передаваемые значения. Другими словами это означает, что при вызове POU необходимо передача значений у входных переменных. После возвращения из POU необходима передача значений у выходных переменных.

Переменные класса **VAR_IN_OUT** могут служить в качестве входных и выходных одновременно. Параметры, передаваемые POU посредничеством переменных классов **VAR_IN_OUT** не передаются значением, а референцией. Это означает, что при вызове POU передается адрес переменной вместо ее значения, что позволяет использовать переменную при необходимости в качестве входных или выходных.

Разница между передачей параметров значения и референции указано на примере 3.17.

Пример 3.17 Разница в использовании переменных VAR_INPUT и VAR_IN_OUT

```
TYPE
  TMyUsintArray : ARRAY[1..100] OF USINT;
END_TYPE

FUNCTION Suma1  : USINT
  VAR_INPUT
    vector      : TMyUsintArray;
    length      : INT;
  END_VAR
  VAR
    i           : INT;
    tmp         : USINT := 0;
  END_VAR

  FOR i := 1 TO length DO tmp := tmp + vector[i]; END_FOR;
  Suma1 := tmp;
END_FUNCTION

FUNCTION Suma2  : USINT
  VAR_IN_OUT
    vector      : TMyUsintArray;
  END_VAR
  VAR_INPUT
    length      : INT;
  END_VAR
  VAR
    i           : INT;
    tmp         : USINT := 0;
  END_VAR

  FOR i := 1 TO length DO tmp := tmp + vector[i]; END_FOR;
  Suma2 := tmp;
END_FUNCTION

PROGRAM ExampleVarInOut
  VAR
    buffer      : TMyUsintArray := [1,2,3,4,5,6,7,8,9,10];
    result1,
    result2     : USINT;
  END_VAR

  result1 := Suma1( buffer, 10);        // 55
  result2 := Suma2( buffer, 10);        // 55

END_PROGRAM
```

Задание данного примера было создание функции, которая заключается в сумме заданного количества элементов массива типа **USINT**.

Функция **Suma1** используется для входной переменной **vector** класса **VAR_INPUT**, что означает, что при вызове данной функции должна передача значений у всех элементов массива **buffer** на входные переменные **vector**. В данном случае это означает 100 битов данных. Расчет происходит над переменной **vector**.

Функция **Suma2** имеет переменную **vector** определенную в классе **VAR_IN_OUT** и так при вызове данной функции передается адрес переменной **buffer** вместо значений всех

элементов. Это только 4 бита по сравнению с 100 битами в первом случае. Входная переменная **vector** содержит адрес переменной **buffer** и расчет происходит над переменной **buffer**, которая через переменную **vector** непрямо адресована.

3.3.5 Простые и сложные переменные

С точки зрения типа данных можно переменные разделить на ***простые*** а ***сложные***. Простые переменные - это переменные основного типа. Сложные переменные - это тип ***массива*** или ***структуры,*** или же их комбинация. Норма IEC 61 131-3 обозначает эти переменные как многоэлементные переменные (multi-элемент variables).

3.3.5.1 Простые переменные

Простая переменная определена как переменная, которая представляет простой информационный элемент одного из элементарных типов данных или пользовательского типа данных (перечень значений или тип производной рекурсивно так, что можно постепенно в обратном порядке вернуться к перечню значений или элементарным типам данных). Образцы простых переменных приведены в примере 3.18.

Пример 3.18 Простые переменные

```
TYPE
  TColor        : (white, red, gree, black);
  TMyInt        : INT := 100;
END_TYPE

VAR_GLOBAL
  basicColor    : TColor := red;
  lunchTime     : TIME   := TIME#12:00:00;
END_VAR

PROGRAM ExapleSimpleVar
  VAR
    tmpBool     : BOOL;
    count1      : INT;
    count2      : TMyInt;
    currentTime : TIME;
  END_VAR
  VAR_TEMP
    count3      : REAL := 100.0;
  END_VAR

END_PROGRAM
```

3.3.5.2 Массивы

Массив - это массив информационных элементов одинакового типа данных, на которые можно ссылаться с помощью одного или более индексов заключенных в скобках и отделенных запятыми. Индекс должен быть одним из типов включенных в родовые типы **ANY_INT**. Максимальное количество индексов (размер массива) - 4 и максимальный диапазон индексов должен отвечать типу **INT**.

Переменную типа массив можно определить двумя способами. Или же сначала определена производный тип данных массив и после этого основана переменная данного типа. Это например переменная **rxMessage** в примере 3.19. Или можно массив определить прямо в декларации переменных, см. переменная **sintArray** в том же примере. Для обеих способов декларации массива действительны правила, указанные в главе 3.2.3.2. Также способ записи инициализационных значений одинаков.

Пример 3.19 Массивы переменных

```
TYPE
  TMessage    : ARRAY[0..99] OF BYTE;
END_TYPE

VAR_GLOBAL
  delay       : ARRAY [1..5] OF TIME := [ TIME#1h,
                                          T#10ms,
                                          time#3h_20m_15s,
                                          t#15h5m10ms,
                                          T#3d];
END_VAR

PROGRAM ExampleArrayVar
  VAR
    rxMessage   : TMessage;
    txMessage   : TMessage;
    sintArray   : ARRAY [1..2,1..4] OF SINT := [ 11, 12, 13, 14,
                                                 21, 22, 23, 24 ];
  END_VAR
  VAR_TEMP
    pause       : TIME;
    элемент     : SINT;
  END_VAR

   pause   := delay[3];              // 3h 20m 15s
   элемент := sintArray[2, 3];       // 23
END_PROGRAM
```

3.3.5.3 Структуры

Структурированная переменная - это переменная, которая декларирована с типом, который был перед этим специфицирован как структура данных, т.е. типа данных состоящего из массива названных элементов. Декларация производного типа структуры описана в главе 3.2.3.3.

Прямая декларация структуры в декларации переменных не поддерживается.

В примере 3.20 имеется определенная переменная **presure**, которая имеет тип структуры **Tmeasure**. В определении данного типа используется следующая структура **Tlimit**. Пример указывает также инициализацию всех элементов переменной **presure**, в том числе погруженных структур. Одновременно в примере указано, каким способом в программе имеется доступ к отдельным элементам структурированных переменных (напр. **presure.lim.low**). Конструкция **AT %XF10** объяснена в следующей главе.

Пример 3.20 Структурированные переменные

```
TYPE
  TLimit :
    STRUCT
      low, high : REAL;
    END_STRUCT;

  TMeasure :
    STRUCT
      lim       : TLimit;
      value     : REAL;
      failure   : BOOL;
    END_STRUCT;
END_TYPE

VAR_GLOBAL
  AT      %XF10 : REAL;
  presure       : TMeasure := ( lim     := ( low := 10, high := 100.0),
                                value   := 0,
                                failure := false);
END_VAR

PROGRAM ExampleStrucформа

  presure.value := %XF10;                 // input sensor
  IF presure.value < presure.lim.low  OR
     presure.value > presure.lim.high
  THEN
    presure.failure := TRUE;
  ELSE
    presure.failure := FALSE;
  END_IF;
END_PROGRAM
```

3.3.6 Расположение переменных в памяти ПЛК

Размещение переменных в памяти ПЛК проводится автоматически программой перевода. Если по какой-либо причине необходимо уложить переменную на конкретный адрес, это можно специфицировать в декларации переменных с помощью ключевого слова **AT**, за которым следует запись адреса переменной.

Запись адреса переменной

Для записи адреса переменных используется специальной знак процента, „**%**“, ***префикс расположения*** (Location prefix) и ***префикс широты данных*** (Size prefix). За данными знаками следует один или более знаков типа UINT разделенных точками.

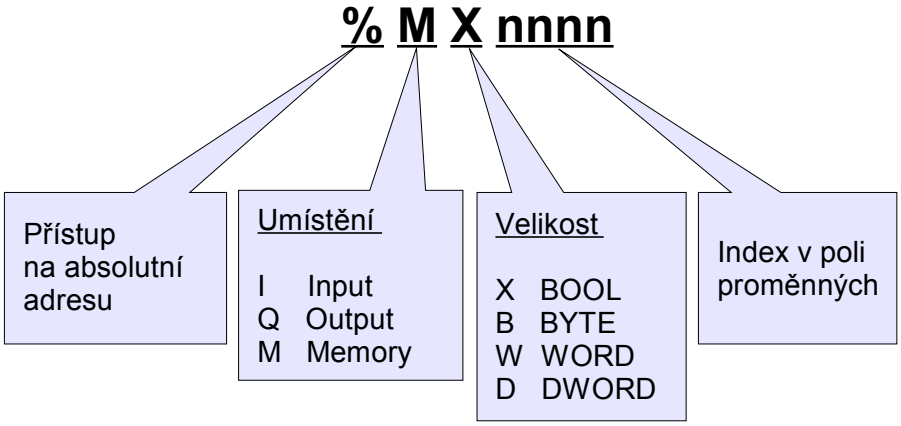


Рис. 8 Прямой адрес в памяти ПЛК согласно IEC

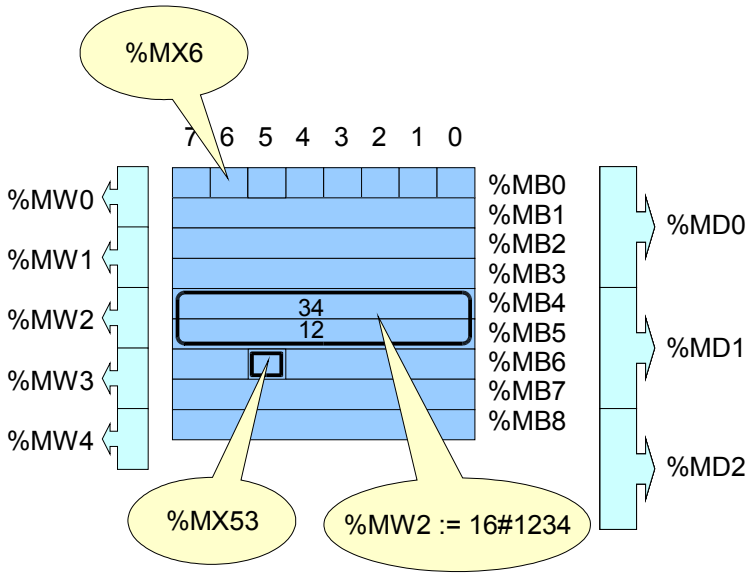


Рис. 9 Обозначение памяти ПЛК согласно IEC

Запись прямого адреса в программах для ПЛК можно также проводить способом, традиционно используемым в среде «Mosaic». Программа перевода автоматически распознает, какой способ записи прямого адреса использовался.

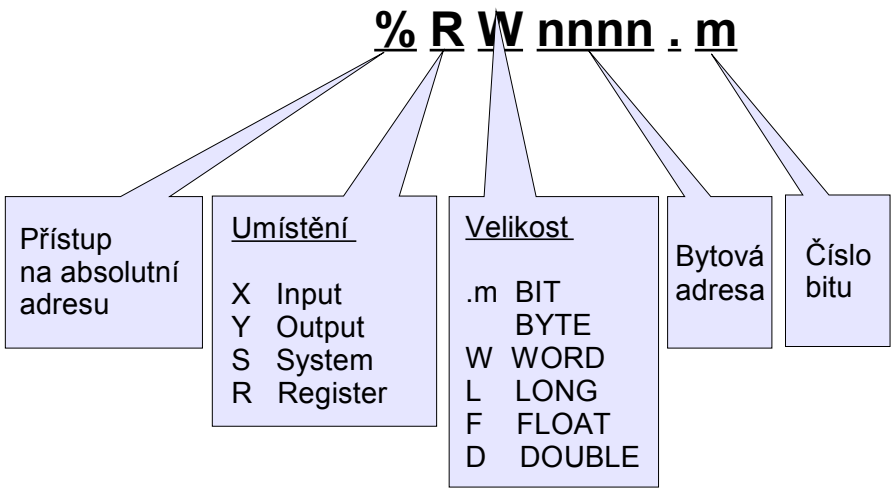


Рис. 10 Традиционная запись прямого адреса переменных в ПЛК

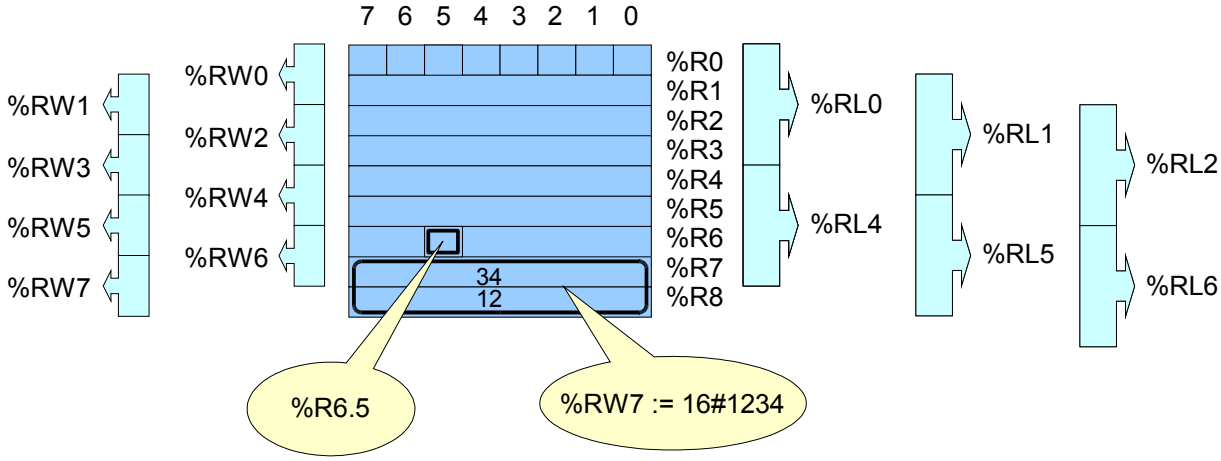


Рис. 11 Традиционное обозначение памяти ПЛК в среде «Mosaic»

Записи **`%MB10`** (согласно IEC) и **`%R10`** (традиционная) обозначают то же место в памяти. Запись **`%RW152.9`** обозначает девятый бит переменной величины **`WORD`**, которая записана в памяти от адреса **`152`**. У переменных, которые в памяти занимают более одного бита, на самом низком адресе укладываются значения наименьшего бита, значение наибольшего бита находятся на самом высоком адресе (Little Endian).

Спецификацию прямого адреса в декларации переменных можно использовать только в классах **VAR_GLOBAL** и **VAR_EXTERNAL**. Ключевое слово **AT**, которое вводит прямой адрес переменной, записывается между названием переменной и спецификацией типа данных.

Переменные, которые в декларации имеют только прямой адрес без названия переменных называются прямо представленные переменные. При доступе к этим переменным в программе вместо названия переменной используется ее адрес. Это случай переменных **%MB121** и **%R122** в примере 3.21.

Пример 3.21 Спецификация адреса в декларации переменных

```
VAR_GLOBAL
  SymbolicVar AT %MB120 : USINT;
              AT %MB121 : USINT;
              AT %R122  : USINT := 242;
  counterOut  AT %Y0.0  : BOOL;              // ПЛК output
END_VAR

PROGRAM ExampleDirectVar
  VAR_EXTERNAL
    AT %S6              : USINT;             // second counter
    AT %X0.0, AT %X0.1  : BOOL;              // ПЛК input
  END_VAR
  VAR
    counter : CTU;
  END_VAR

  SymbolicVar := %MB121 + %R122;
  counter(CU := %X0.0, R := %X0.1, PV := 100, Q => counterOut);
END_PROGRAM
```

Прямой адрес используются для декларации тех переменных, расположение которых не должно изменяться при изменении программы. Примером могут послужить переменные предназначенные для программы визуализации или переменные, которые представляют входу или же выходы ПЛК.

Если в декларации переменных не указан адрес, то расположение символических переменных в память ПЛК (назначение адреса) проведет программа перевода. При этом она также обеспечит, чтобы переменные в памяти не перекрывались.

У прямо презентируемых переменных о расположении переменные в памяти решает программатор и он также должен обеспечить, чтобы не произошла нежелательная коллизия переменных (перекрытие их адресов в памяти).

3.3.7 Инициализация переменных

Модель программирования согласно IEC 61 131 обеспечивает, чтобы каждая переменная после повторного запуска системы управления имела закрепленное начальное (инициализационное) значение. Это значение может быть:

- Значение, которое переменная имела в момент прекращения расчета – типично при выпадении питания системы управления (retained value)
- Пользователем специфицированное исходное значение (указанное в декларации переменных)
- Предварительно определенное (дефо) исходное значение согласно типу данных

Пользователь может декларировать, что должна быть переменная ***ретентивная*** (авансированная, то есть то, что должно сохраниться - последнее значение) с помощью квалификатора **RETAIN**. Этот квалификатор можно использовать только для глобальных переменные.

Инициализационное значение переменных можно специфицировать в рамках декларации переменных. Если не инициализировано значение в декларации переменных, будет переменная инициализирована на начальное значение согласно используемому типу данных.

Исходное значение переменных

Исходное значение переменных после повторного запуска системы определяется согласно следующим правилам:

- Если стартовая операция т. наз. теплый повторный запуск, то начальное значение ретентивных переменных будут их ретентивные (последние сохранившиеся) значения
- Если стартовая операция т. наз. холодный повторный запуск, то начальное значение ретентивных переменных будут пользователем специфицированы на начальное значение
- Неретентивные переменные будут инициализированы на значения, специфицированные пользователем или на предварительно определенные значения для соответствующего типа данных и всех переменных, где не специфицировано исходное значение пользователем
- Переменные, которые представляют вход системы ПЛК, будут инициализированы согласно состоянию сигналов, присоединенных к этим входам
- Переменные представляющие выходы системы будут инициализированы значением 0, которое отвечает состоянию „без напряжения“

У переменных в классе **VAR_EXTERNAL** не могут приделяться начальные значения, так как собственно говоря речь идет о ссылках на переменные, которые декларированы в другом месте программы. Инициализацию не возможно декларировать такжеу переменных класса **VAR_IN_OUT**, так как эти переменные содержат только «pointer» на переменные а не на сами переменные.

Пример 3.22 Инициализация переменных

```
TYPE
  MY_REAL : REAL := 100.0;
END_TYPE

VAR_GLOBAL RETAIN
  remanentVar1 : BYTE;
  remanentvar2 : BYTE := 56;
END_VAR
```

```
PROGRAM ExampleInitVar
  VAR
    localVar1   : REAL;
    localVar2   : REAL := 12.5;
    localVar3   : MY_REAL;
  END_VAR
  VAR_TEMP
    tempVar1    : BYTE;
    tempVar2    : REAL;
  END_VAR

    tempVar1    := remanentvar AND remanentVar2;
    tempVar2    := localVar1 + localVar2;
END_PROGRAM
```

При холодном повторном запуске системы будет авансированная переменная **remanentVar1** инициализационное значение **0** согласно дефо инициализационное значения типа **BYTE**. Переменная **remanentVar2** будет иметь инициализационное значение 56, так как это значение установлено в декларации переменных.

При теплом повторном запуске системы будут иметь переменные **remanentVar1** и **remanentVar2** такие значения, какие имели эти переменные при выключении питания системы.

Переменная **localVar1** будет независимо от типа повторного запуска инициализирована значением **0.0**, так как в декларации переменных нет инициализационных значений и поэтому используется предварительно определенное инициализационное значение типа данных **REAL**. Переменная **localVar2** будет после повторного запуска всегда инициализирована значением **12.5**. Переменная **localVar3** будет после повторного запуска инициализирована значением **100.0**, так как это инициализационное значение производных типа данных **MY_REAL**.

3.4 Программные организационные единицы

Программные организационные единицы (Programa Organisation Units, POUs) - это ***функции***, ***функциональные блоки*** и ***программы***. Могут быть поставлены от производителя или их может написать пользователь.

Программные организационные единицы ***не являются рекурсивными***, то есть вызов одной программной организационной единицы не должен причинить вызове другой программной организационной единицы того же типа! Упрощенно можно сказать, что POU не может вызывать сама себя.

3.4.1 Функции

В целях языков программирования для ПЛК функции определены как программная организационная единица, которая после проведения сотрет всегда один информационный элемент (может быть составлен из большего количества значений, как напр. массив или структура). Вызов функции может использоваться в текстовых языках как операнд в выражении.

Функции не содержат какую-либо внутреннюю информацию состояния, то есть вызов функции с одинаковыми аргументами (входными параметрами) создаст всегда такое же значение (выход).

Декларация функции

Текстовая декларация функции состоит из следующих элементов:

- Ключевое слово **`FUNCTION`**, за которым указано название декларированной функции, двоеточия и Тип данных значения, которое будет функцию возвращать
- Определение входных переменных **`VAR_INPUT`**, где указаны спецификации названий и типов входных параметров функции
- Определение локальных переменных **`VAR`** или же **`VAR_TEMP`**, где указаны спецификации названий и типов внутренних переменных функций
- Определение постоянных величин **`VAR CONSTANT`**
- Тело функции (Function body) записано на одном из языков IL, ST, LD или FBD. Тело функции специфицируют операции, которые должны проводиться над входными параметрами с целью отнесения одного или более значений переменных, которые имеют то же имя, какое имеют функции, а которое представляет величину возврата функции
- Заключительного ключевого слова **`END_FUNCTION`**

Вызов функции

В языке ST можно вызвать функцию записью названия функции с передаваемыми параметрами в круглых скобках. Количество а Тип данных передаваемых параметров должен отвечать входным переменным в определении функции. Если в вызове не указаны названия входных переменных функции, то порядок параметров должен точно отвечать порядку входных переменных в определении функции. Если параметры приделены к названиям входных параметров (formal call), то порядок параметров при вызове функции не имеет значения.

Пример 3.23 Определение функции и ее вызов в языке ST

```
FUNCTION MyFunction : REAL
  VAR_INPUT
    r, h    : REAL;
  END_VAR
  VAR CONSTANT
    PI      : REAL := 3.14159;
  END_VAR

  IF r > 0.0 AND h > 0.0
  THEN MyFunction := PI * r**2 * h;
  ELSE MyFunction := 0.0;
  END_IF;
END_FUNCTION


PROGRAM ExampleFunction
  VAR
    v1, v2  : REAL;
  END_VAR

  v1 := MyFunction( h := 2.0, r := 1.0);
  v2 := MyFunction( 1.0, 2.0);
END_PROGRAM
```

Функции **MyFunction** в примере 3.23 имеют определенные две входные переменные **r** и **h** типа **REAL**. Величина возврата данной функции имеет тип **REAL** и представлена названием **MyFunction**. В вызове данной функции **v1 := MyFunction(h := 2.0, r := 1.0)** указаны названия входных переменных. В данном случае не зависит от порядка входных параметров в скобках. Вызов **v2 := MyFunction(1.0, 2.0)** названия входных переменных не содержит и поэтому входными параметрами предполагаются такой порядок, в котором определены входные переменные в декларации функции. Оба вызова в указанном примере равноценны и показывают одинаковый результат.

3.4.1.1 Стандартные функции

Стандартные функции используемые во всех языках программирования для ПЛК подробно определены в норме IEC 61 131-3 в главе 2.5.1.5. Совокупность стандартных функций, которые поддерживаются программой перевода в среде «Mosaic», указана в данной главе.

Перегрузка функции (Overloading)

О функции или операции говорим, что она ***перегружена*** (*overloaded)*, если может работать над элементами входных данных различных типов в рамках родового названия типа. Напр. перегружена функция считывания родового типа **ANY_NUM** может работать над типами данных **LREAL**, **REAL**, **DINT**, **INT** и **SINT**. Если система ПЛК поддерживает перегруженную функцию или операции, то данная функция может применяться на все типы данных данного родового типа, которые поддерживает система.

Информация о том, какие функции перегружены приведена ниже. Пользовательские определенные функции не могут быть перегружены.

Если все формальные входные параметры - стандартные функции того же родового типа, то все актуальные параметры должены быть того же типа. Если это необходимо, могут использоваться в этих целях функции для *преобразования* типа. Выходные значения функции будут того же типа как актуальные входу.

Расширяемые функции (Extensible)

Некоторые стандартные функции ***расширяемы*** *(extensible)*, то есть могут иметь переменное количество входов. У этих функций предполагается, что операции определенных функций будут проводиться над всеми применяемыми входами. Если функции расширяемы, максимальное количество входов не ограничено.

Разделение стандартных функций

Стандартные функции разделяются на несколько основных групп :

- Функции для преобразования типа
- Цифровые функции
  - цифровые функции одной переменной
  - арифметические функции большего количества переменных
- Функции над последовательностью битов
  - ротация битов
  - функции Боола
- Функции выбора
- Функции сравнения
- Функции последовательности знаков
- Функции с типами даты и времени
- Функции над типами данных „перечень“

Колонка с названием **Ovr** в следующих таблицах задает, если функция перегружена (overloaded). Колонка с названием **Ext** несет информацию о том, если соответствующая функции расширяема (extensible). Точную спецификацию стандартных функций см. приложение.

Табл.13 Стандартные функции, группа преобразования типа

Стандартные функции, группа преобразования типа					
Наименование функции	**Тип данных входа**	**Тип данных выхода**	**Описание функции**	**Ovr**	**Ext**
…_TO_…	ANY	ANY	Преобразования типа данных указанного на первом месте на Тип данных указанный на втором месте	да	нет
TRUNC	ANY_REAL	ANY_INT	„Обрезка“	да	нет

Табл.14 Стандартные функции, группа цифровые функции одной переменной

Стандартные функции, группа цифровой функции одной переменной				
Наименование функции	**Тип данных входа / выхода**	**Описание функции**	**Ovr**	**Ext**
ABS	ANY_NUM / ANY_NUM	Абсолютное значение	да	нет
SQRT	ANY_REAL / ANY_REAL	Корень	да	нет
LN	ANY_REAL / ANY_REAL	Натуральный логарифм	да	нет
LOG	ANY_REAL / ANY_REAL	Десятичный логарифм	да	нет
EXP	ANY_REAL / ANY_REAL	Натуральная показательная функция	да	нет
SIN	ANY_REAL / ANY_REAL	Синус входного угла указанного в радианах	да	нет
COS	ANY_REAL / ANY_REAL	Косинус входного угла указанного в радианах	да	нет
TAN	ANY_REAL / ANY_REAL	Тангенс входного угла указанного в радианах	да	нет
ASIN	ANY_REAL / ANY_REAL	Дуга синуса	да	нет
ACOS	ANY_REAL / ANY_REAL	Дуга косинуса	да	нет
ATAN	ANY_REAL / ANY_REAL	Дуга тангенса	да	нет

Табл.15 Стандартные функции, группа цифровые функции - арифметические функции большее количество переменных

Стандартные функции, группа цифровые функции - арифметические функции большее количество переменных					
Наименование функции	**Тип данных входа / выхода**	**Символ**	**Описание функции**	**Ovr**	**Ext**
ADD	ANY_NUM, .. ANY_NUM / ANY_NUM	+	Сумма OUT:=IN1+ IN2+…+INn	да	да
MUL	ANY_NUM, .. ANY_NUM / ANY_NUM	*	Произведение OUT:=IN1* IN2*…*INn	да	да
SUB	ANY_NUM, ANY_NUM / ANY_NUM	-	Разница OUT:=IN1-IN2	да	нет
DIV	ANY_NUM, ANY_NUM / ANY_NUM	/	Доля OUT:=IN1/IN2	да	нет
MOD	ANY_NUM, ANY_NUM / ANY_NUM		Модуло OUT:=IN1 модульо IN2	да	нет
EXPT	ANY_REAL, ANY_NUM / ANY_REAL	**	Возвышение в степень OUT:=IN1**IN2	да	нет
MOVE	ANY_NUM / ANY_NUM	:=	Перемещение, включение OUT:=IN	да	нет

Табл.16 Стандартные функции, группа функции над последовательностью битов - ротация битов

Стандартные функции, группа функции над последовательностью битов - ротация битов				
Наименование функции	**Тип данных входа / выхода**	**Описание функции**	**Ovr**	**Ext**
SHL	ANY_BIT, N / ANY_BIT	Смещение влево OUT := IN смещен влево на N битов, справа дополнено нулями	да	нет
SHR	ANY_BIT, N / ANY_BIT	Смещение вправо OUT := IN смещен вправо на N битов, слева дополнено нулями	да	нет
ROR	ANY_BIT, N / ANY_BIT	Ротация вправо OUT := IN ротация вправо на N битов, справа дополнено на ротируемые слева биты	да	нет
ROL	ANY_BIT, N / ANY_BIT	Ротация влево OUT := IN ротация влево на N битов, слева дополнено на ротируемые справа биты	да	нет

Табл.17 Стандартные функции, группа функции над последовательностью битов - функции Боола

Стандартные функции, группа функции над последовательностью битов – функции Боола					
Наименование функции	**Тип данных входа / выхода**	**Символ**	**Описание функции**	**Ovr**	**Ext**
AND	ANY_BIT, .. ANY_BIT / ANY_BIT	&	Log. произведение, „и одновременно“, OUT:=IN1& IN2&…&INn	да	да
OR	ANY_BIT, .. ANY_BIT / ANY_BIT		Log. сумма, „или“, включенная сумма, OUT:=IN1 OR IN2 OR … OR INn	да	да
XOR	ANY_BIT, .. ANY_BIT / ANY_BIT		Исключенная сумма, „или же/ или“, эксклюзивная сумма OUT:=IN1 XOR IN2 XOR … XOR INn	да	да
NOT	ANY_BIT / ANY_BIT		Отрицание, „нет“, OUT:=NOT IN1	да	нет

Табл.18 Стандартные функции, группа функции выбора

Стандартные функции, группа функции выбора				
Наименование функции	**Тип данных входа / выхода**	**Описание функции**	**Ovr**	**Ext**
SEL	BOOL, ANY, ANY / ANY	Бинарный подбор OUT := IN0 if G = 0 OUT := IN1 if G = 1	да	нет
МАКС	ANY, .. ANY / ANY	Максимум OUT := МАКС(IN1, IN2, .. INn)	да	да
MIN	ANY, .. ANY / ANY	Минимум OUT := MIN(IN1, IN2, .. INn)	да	да
LIMIT	MN, ANY, MX / ANY	Ограничитель OUT := MIN(МАКС(IN, MN), MX)	да	нет

Табл.19 Стандартные функции - функции сравнения

Стандартные функции, группа функции сравнения				
Наименование функции	**Тип данных входа / выхода**	**Описание функции**	**Ovr**	**Ext**
GT	ANY, .. ANY / BOOL	Убывающая последовательность OUT:=(IN1> IN2)& (IN2>IN3)&…&(INn-1>INn)	да	да
GE	ANY, .. ANY / BOOL	Монотонная последовательность в направлении вниз OUT:=(IN1>= IN2)& (IN2>=IN3)&…&(INn-1>=INn)	да	да
EQ	ANY, .. ANY / BOOL	Равенство OUT:=(IN1= IN2)& (IN2=IN3)&…&(INn-1=INn)	да	да
LE	ANY, .. ANY / BOOL	Монотонная последовательность в направлении вверх OUT:=(IN1<= IN2)& (IN2<=IN3)&…&(INn-1<=INn)	да	да
LT	ANY, .. ANY / BOOL	Возростающая последовательность OUT:=(IN1< IN2)& (IN2<IN3)&…&(INn-1<INn)	да	да
NE	ANY, ANY / BOOL	Нетравенство OUT := (IN1<>IN2)	да	нет

Табл.20 Стандартные функции, группа функции над последовательностью знаков

Стандартные функции, группа функции над последовательностью знаков				
Наименование функции	**Тип данных входа / выхода**	**Описание функции**	**Ovr**	**Ext**
LEN	STRING / INT	OUT := LEN(IN); Длина последовательности IN	нет	нет
LEFT	STRING, ANY_INT / STRING	OUT := LEFT(IN, L); Из входной последовательности IN переместить L знаков слева на выходную последовательность	да	нет
RIGHT	STRING, ANY_INT / STRING	OUT := RIGHT(IN, L); Из входной последовательности IN переместить L знаков справа до выходных последовательностей	да	нет
MID	STRING, ANY_INT, ANY_INT / STRING	OUT := MID(IN, L, P); Из входных последовательностей IN переместить от P-того знака L знаков к выходной последовательности	да	нет
CONCAT	STRING, …. STRING /STRING	OUT := CONCAT(IN1, IN2, ...); Присоединение отдельных входных последовательностей к выходной последовательности	нет	да
INSERT	STRING, STRING, ANY_INT / STRING	OUT := INSERT(IN1, IN2, P); Вкладывание последовательности IN2 в последовательность IN1 начиная с P-той позиции	да	да
DELETE	STRING, ANY_INT, ANY_INT / STRING	OUT := DELETE(IN, L, P); Стирание L знаков из последовательности IN начиная с P-той позиции	да	да
REPLACE	STRING, STRING, ANY_INT, ANY_INT / STRING	OUT := REPLACE(IN1, IN2, L, P); Замен L знаков последовательности IN1 знаками последовательности IN2, вкладывание от P-того места	да	да
FIND	STRING, STRING / INT	OUT := FIND(IN1, IN2); Умножение позиции первого знака первого проявления последовательности IN2 в последовательности IN1	да	да

Табл.21 Стандартные функции, группа функции с типами даты и времени

Стандартные функции, группа функции с типами даты и времени					
Наименование функции	**IN1**	**IN2**	**OUT**	**Ovr**	**Ext**
ADD_TIME	TIME	TIME	TIME	нет	нет
ADD_TOD_TIME	TIME_OF_DAY	TIME	TIME_OF_DAY	нет	нет
ADD_DT_TIME	DATE_AND_TIME	TIME	DATE_AND_TIME	нет	нет
SUB_TIME	TIME	TIME	TIME	нет	нет
SUB_DATE_DATE	DATE	DATE	TIME	нет	нет
SUB_TOD_TIME	TIME_OF_DAY	TIME	TIME_OF_DAY	нет	нет
SUB_TOD_TOD	TIME_OF_DAY	TOD	TIME	нет	нет
SUB_DT_TIME	DATE_AND_TIME	TIME	DATE_AND_TIME	нет	нет
SUB_DT_DT	DATE_AND_TIME	DT	TIME		
MULTIME	TIME	ANY_NUM	TIME	да	нет
DIVTIME	TIME	ANY_NUM	TIME	да	нет
CONCAT_DATE_TOD	DATE	TIME_OF_ DAY	DATE_AND_TIME	нет	нет
Функции преобразования типа					
DATE_AND_TIME_TO_TIME_OF_DAY, ДАННЫХ_TO_TIME DATE_AND_TIME_TO_DATE, ДАННЫХ_TO_DATE					

TOD … TIME_OF_DATE
DT … DATE_AND_TIME

3.4.2 Функциональные блоки

При программировании согласно IEC 61 131-3 ***функциональный блок*** это такая организационная единица программы, которая после проведения сотрет одно или более значений. Из функциональных блоков можно создавать многократные названные ***инстанции*** (копии). Каждая инстанция имеет соответствующий идентификатор (*наименование инстанции*) и структуру данных, которая содержит ее входные, внутренние и выходные переменные. Все значения переменных в данной структуре данных сохраняются от одного исполнения функционального блока к следующим его исполнениям. Вызов одного функционального блока с одинаковыми аргументами (входными параметрами), не должен всегда вести к одинаковым выходным значениям. Инстанции функционального блока образуются использованием декларированного типа функционального блока в рамках класса **VAR** или **VAR_GLOBAL**.

Любой функциональный блок, который уже был декларирован, может быть опять использован в декларации другого функционального блока или программы.

Диапазон действия инстанции функционального блока - местный для той программной организационной единицы, в которой *инстанционирован*, (т.е. где создана его названная копия), если он не декларирован как глобальный.

Нижеприведенный пример указывает действия при декларации функционального блока, созданные его инстанцией в программе и наконец его вызовет (исполнение).

Пример 3.24 Функциональный блок в языке ST

```
FUNCTION_BLOCK fbstartСтоп              // декларация FB
  VAR_INPUT
    start       : BOOL R_EDGE;          // входная переменная
    стоп        : BOOL R_EDGE;
  END_VAR
  VAR_OUTPUT
    output      : BOOL;                 // входная переменная
  END_VAR

  output := (output OR start) AND not стоп;
END_FUNCTION_BLOCK


PROGRAM ExampleFB
  VAR
    startСтоп   : fbstartСтоп;          // инстанции FB
    running     : BOOL;
  END_VAR

  // вызов инстанции функционального блока
  startСтоп( стоп := FALSE, start := TRUE, output => running);

  // альтернативный способ вызова FB
  startСтоп.start := TRUE;
  startСтоп.стоп  := FALSE;
  startСтоп();
  running         := startСтоп.output;

  // вызов с неполным перечнем параметров
  startСтоп( start := TRUE);
  running := startСтоп.output;

END_PROGRAM
```

Входные и выходные переменные инстанции функционального блока могут быть представлены как элементы типа данных структуры.

Если инстанции функционального блока глобальные, то могут также декларироваться как ретентивные. В данном случае действительно только для внутренних и выходных параметров функционального блока.

Снаружи инстанции доступны только входные и выходные параметры функционального блока, то есть внутренние переменные функционального блока остаются от пользователя функционального блока скрытыми. Отнесение значений снаружи в выходные переменные функционального блока не разрешается, это значение приделяет только изнутри сам

функциональный блок. Приделение значений у входа функционального блока разрешено в любом вышестоящем POU (стандартно это составная часть вызова функционального блока).

Декларация функционального блока

- Разделительные ключевые слова для декларации функциональных блоков - это **FUNCTION_BLOCK**…**END_FUNCTION_BLOCK**
- Функциональный блок может иметь более чем один выходной параметр, декларированный в классе **VAR_OUTPUT**
- Значения переменных, которые передаются функциональному блоку в классе **VAR_IN_OUT** или **VAR_EXTERNAL** могут быть модифицированы внутри функционального блока.
- В декларации входных переменных функционального блока могут быть использованы квалификаторы **R_EDGE** и **F_EDGE**. Эти квалификаторы обозначают функцию детекции граней на входах Боола. Этим вызывается неявная декларация функционального блока **R_TRIG** или **F_TRIG**.
- Конструкция определенная для инициализации функций используется для декларации дефо значений входов функционального блока и для начального значения его внутренних и выходных переменных

С помощью класса **VAR_IN_OUT** в функциональный блок могут быть переданы только переменные (передача инстанций функциональных блоков не поддерживается). Каскад переменных **VAR_IN_OUT** разрешается.

3.4.2.1 Стандартные функциональные блоки

Стандартные функциональные блоки подробно определены в норме IEC 61 131-3 в главе 2.5.2.3.

Стандартные функциональные блоки разделены на следующие группы (см. Табл.3.22)

- Бистабильные элементы
- Детекция грани
- Счетчики
- Таймеры

Стандартные функциональные блоки уложены в библиотеке StdLib_Vxx_*.mlb, где Vxx – это версия библиотеки.

Табл.22 Перечень стандартных функциональных блоков

Наименование стандартного функционально-го блока	**Наименование входного параметра**	**Наименование выходного параметра**	**Описание**
Бистабильные элементы (триггерные схемы)			
SR	S1, R	Q1	главные настройки (сцепления)
RS	S, R1	Q1	главные смазки (размыкания)
Детекция hrany			
R_TRIG	CLK	Q	детекция переднего торца
F_TRIG	CLK	Q	детекция нисходящего торца
Счетчики			
CTU	CU, R, PV	Q, CV	предварительный счетчик
CTD	CD, LD, PV	Q, CV	обратный счетчик
CTUD	CU, CD, R, LD, PV	QU, QD, CV	двусторонний (реверсивный) счетчик
Таймеры			
TP	IN, PT	Q, ET	пульсирующий таймер
TON (T--0)	IN, PT	Q, ET	опоздание переднего тоорца
TOF (0--T)	IN, PT	Q, ET	опоздание нисходящего торца

Названия, значение и типы данных переменных используемые в стандартных функциональных блоках :

Название входа / выхода	**Значение**	**Тип данных**
R	Смазочный (повторного включения) вход	BOOL
S	Регулировочный (многотипный) вход	BOOL
R1	Главный смазочный вход	BOOL
S1	Главный регулировочный вход	BOOL
Q	Выход (стандартный)	BOOL
Q1	Выход (только у триггерных схем)	BOOL
CLK	Часовой (синхронизирующий) сигнал	BOOL
CU	Вход для прямого считывания	BOOL
CD	Вход для обратного считывания	BOOL
LD	Настройки предварительной настройки счетчика	BOOL
PV	Предварительный выбор счетчика	INT
QD	Выход (обратного счетчика)	BOOL
QU	Выход (прямого счетчика)	BOOL
CV	Актуальные значения (счетчики)	INT
IN	Вход (таймеры)	BOOL
PT	Предварительный выбор таймера	TIME
ET	Актуальные значения таймера	TIME
PDT	Предварительный выбор - дата и время	DT
CDT	Актуальные значения - дата и время	DT

3.4.3 Программы

Программа в норме IEC 61 131-1 определена как „логическая совокупность элементов языков программирования и конструкций необходимых для задуманной обработки сигналов, которые необходимы для управления машины или процесса системой программируемых контроллеров“.

Иначе говоря функции и функциональные блоки можно прировнать к подпрограммам (subroutines) в то время как программа POU – это основная программа (main program). Декларация и использование *программ* идентично декларации и использованию функциональных блоков.

Различия в доступе к программам по сравнению с функциональным блоком:

- Ключевые слова ограничивают декларации программы – **PROGRAM**… .**END_PROGRAM**
- *Программы* могут быть инстанционированы только в рамках *источников (Resources)*, как это указано в гл. 3.5. И наоборот, *функциональные блоки* могут быть инстанционированы только в рамках *программ* или других *функциональных блоков*
- Программы могут вызывать функции и функциональные блоки. Наоборот вызов программ из функций или функциональных блоков не возможен

Пример 3.25 POU Программа в языке ST

```
PROGRAM проверка
  VAR
    motor1 : fbMotor;
    motor2 : fbMotor;
  END_VAR

  motor1( startMotoru := sb1, стопMotoru := sb2,
          hvezda => km1, trojuhelnik => km2);
  motor2( startMotoru := sb3, стопMotoru := sb4,
          hvezda => km3, trojuhelnik => km4);

END_PROGRAM
```

При записи программы в языке ST важно осознать, что **PROGRAM** POU так же как функциональный блок является только „инструкцией“, в которой определена структура данных и алгоритмы, проведенные над данной структурой данных. Для выполнения определенных программ необходимо оформить ее инстанцию и причислить (ассоциировать) программу к одной из стандартных задач, в которыхх потом будет проводиться. Эти задачи описаны в нижеприведенной главе.

3.5 Конфигурационные элементы

Конфигурационные элементы описывают свойства программы «run-time» и причисляют проведение программы к конкретному техническому обеспечению в ПЛК. Таким образом представляют главную инструкцию целой программы для ПЛК.

При программирование систем ПЛК используются нижеприведенные конфигурационные элементы :

- ***Конфигурация*** *(Configuration)* – обозначает конкретную систему ПЛК, которая будет проводить все запрограмированные POU
- ***Источник*** *(Resource)* – обозначает конкретный модуль процессора в ПЛК, который обеспечивает ход программы
- ***Задача*** *(Task)* **–** причисляет задачу (процесс), в рамках которого будет соответствующая **PROGRAM** POU проводиться

Пример 3.26

```
CONFIGURATION Плк1
  RESOURCE CPM
    TASK FreeWheeling(Number := 0);
    PROGRAM prg WITH FreeWheeling : проверка ();
  END_RESOURCE
END_CONFIGURATION
```

В среде программирования «Mosaic» все конфигурационные элементы генерируются автоматически после выполнения конфигурационных диалогов.

3.5.1 Конфигурация

Конфигурация отмечает систему ПЛК, которая предоставит источники для проведения пользовательской программы. Другими словами конфигурация отмечает систему управления, для которой предназначена пользовательская программа.

Декларация конфигурации

- Ключевые слова ограничиваются *конфигурацией* **CONFIGURATION….END_ CONFIGURATION**
- За ключевым словом **CONFIGURATION** указано названия конфигурации, в среде программирования «Mosaic» наименование конфигурации отвечает другому проекту
- *Конфигурация* предназначена как рамка для определений *Источника (Resource)*

Пример 3.27

```
CONFIGURATION наименование_конфигурация

  // декларация источника

END_CONFIGURATION
```

3.5.2 Источники

Источник определяет, какой модуль в рамках ПЛК предоставит расчетную операцию для выполнения пользовательской программы. В ПЛК серии ТС700 это всегда модуль процессора системы.

Декларация источника

- Ключевые слова ограничивающие *источник* - **RESOURCE**….**END_ RESOURCE**
- За ключевым словом **RESOURCE** указано название источника, в среде программирования «Mosaic» это наименование неявно „CPM“
- *Источники (Resources)* могут быть декларированы только в рамках *конфигурации*

Пример 3.28

```
CONFIGURATION Плк1
  RESOURCE CPM

    // декларация задач

    // причисление программ к декларированным задачам

  END_RESOURCE
END_CONFIGURATION
```

3.5.3 Задачи

В целях нормы IEC 61 131-3 *задачы* определены как операционный элемент управления, который способен вызвать периодически или на основании появления восходящего торца специфицированных переменных Боола проведения массива программных организационных единиц (POUs). Данными программными организационными единицами могут быть *программы* и в них декларированные *функциональные блоки.*

В среде «Mosaic» понятие задача тождественно традиционно используемому понятию процесс.

Декларация задач

- Ключевое слово для ообозначения *задачи* **TASK**
- За ключевым словом **TASK** указано наименование задачи
- За названием задачи указаны свойства задачи, конкретная цифра соответствующего процесса
- *Задачи (Tasks)* могут быть декларированы только в рамках декларации *источника (Resource)*

Причисление программ к задачам

- Причисление программ к конкретной задаче уложено ключевым словом **PROGRAM**, которое одновременно автоматически состоит из инстанции указанной программы
- За ключевым словом **PROGRAM** следует наименование инстанции программы
- Ключевое слово **WITH** вводит наименование задачи, к которому будет программа причислена
- В заключение за двумя точками указано наименование ассоциированной программы, в том числе спецификация входных и выходных параметров
- С одной задачей может быть ассоциировано несколько программ, порядок их выполнения в рамках задачи отвечает порядку, в котором были ассоциированы
- Причисление программ может быть декларировано только в рамках декларации *источника (Resource)*

Пример 3.29

```
CONFIGURATION PLC1
  RESOURCE CPM
    TASK FreeWheeling(Number := 0);
    PROGRAM prg WITH FreeWheeling : Test ();
  END_RESOURCE
END_CONFIGURATION
```

4 ТЕКСТОВЫЕ ЯЗЫКИ

В норме IEC 61 131-3 определены два текстовые языки: язык перечня инструкций (IL, Instruction List) и язык структурированного текста (ST, Structured Text). Оба эти языки - это программа перевода поддерживаемая в среде «Mosaic».

4.1 Язык перечня инструкций IL

Язык перечня инструкций (Instruction List) - это язык низкого уровня типа «assembler». Этот язык принадлежит к строчечно ориентированным языкам.

4.1.1 Инструкции в IL

Перечень инструкций состоит из последовательности (последовательностей) *инструкций*. Каждая инструкция (команда) начинается на новой строчке и содержит *оператора*, который может быть дополнен *модификаторами*, и если это необходимо для конкретной инструкции, то содержит один или более *операндов* разделенных полосами. На месте операндов могут быть любые репрезентации данных, определенные для литеральных констант (см. гл. 3.1.2) и переменные (см. гл. 3.3).

В целях идентификации может быть перед инструкцией указан *сигналов,* за которым следует двоеточия (:). Сигналы предназначены к оообозначению места в программе для инструкции вызова или скока. На последнем месте на строчке инструкции может быть приведен коментарий. Между инструкциями могут быть вложены пустые строчки. Демонстрация программы в языке IL указана в примере 4.1.

Пример 4.1 Программа в языке IL

```
VAR_GLOBAL
  AT %X1.2       : BOOL;
  AT %Y2.0       : BOOL;
END_VAR

PROGRAM Example_IL
  VAR
    tmp1, tmp2   : BOOL;
  END_VAR

Step1:  LD       %X1.2    // load byt from ПЛК input
        AND      tmp1     (* AND temporary variable *)
        ST       %Y2.0    (* store to PLC output *)
                          (* empty instruction *)
Step2:                    (* label *)
        LDN      tmp2
END_PROGRAM
```

4.1.2 Операторы, модификаторы и операнды

Стандартные операторы вместе с допустимыми модификаторами указаны в Табл. 4.1 - Табл. 4.4. Если в таблицах не указано другое, семантика операторов следующая:

```
результат := результат ОПЕРАТОР операнд
```

Это означает, что значение выражения, которое оценивается, заменено его новым значением, которое обработано из его актуального значения с помощью оператора или же операнды. Напр. инструкции **AND %X1** толкуются следующим образом:

```
результат := результат AND %X1
```

Операторы сравнения интерпретируются с актуальным результатом слева от знака сравнения и операнды вправо от знака сравнения. Результатом сравнения является переменная Боола. Напр. инструкции **LT %IW32** будут иметь в результате „1“ Боола, если актуальный результат меньше чем значение входного слова номер 32, во всех остальных случаях будет в результате иметь „0“ Боола.

Модификатор **N** отмечает отрицание Боола операнды. Напр. инструкции **ORN %X1.5** интерпретируется следующим образом:

```
результат := результат OR NOT %X1.5
```

Модификатор левой скобки **(**отмечает, что оператор должен быть „отложен“, т.е. действия оператора одложены, (deferred), покуд не обозначен оператор правой скобки **)**. Напр. последовательность инструкций

```
AND(        %X1.1
OR          %X1.3
)
```

интерпретирована следующим образом:

```
результат := результат AND (%X1.1 OR %X1.3)
```

Табл.23 Операторы и модификаторы для типа данных ANY_BIT

ANY_BIT Операторы		
Операторы	**Модификатор**	**Описание функции**
LD	**N**	Главный актуальный результат на обозначении равняется операнде
AND	**N, (**	AND Боола
OR	**N, (**	OR Боола
XOR	**N, (**	XOR Боола
ST	**N**	Запишет актуальный результат на место операнды
S		Отрегулирует операнду Боола на „1“ Операция будет проведена только если актуальный результат Боола равен „1“
R		Отменит операнд Боола на „0“ Операция проведется только если актуальный результат Боола „1“
)		Оценка отлоденной операции

Некоторые операторы могут быть дополнены большим количеством модификаторов одновремено. Например оператор **AND** имеет в комбинации с модификаторами четыре различных формы как это указано в таблице Табл. 4.2.

Табл.24 Модификация оператора AND

AND	AND Боола
AND(	Отлаженный (deferred) AND Боола
ANDN	AND Боола с отрицательным операндом
ANDN(	Отлаженный AND Боола с отрицательным результатом

Табл.25 Операторы и модификаторы для типа данных ANY_NUM

ANY_NUM Операторы		
Операторы	**Модификатор**	**Описание функции**
LD	**N**	Отрегулирует актуальный результат на значение, равное операнду
ST	**N**	Запишет актуальные результаты на место операнда
ADD	**(**	Присчитывать операнд к результату
SUB	**(**	Отсчитывать операнд от результата
MUL	**(**	Умножать результат операндом
DIV	**(**	Делить результат на операнд
GT	**(**	Сравнение результат > операнд
GE	**(**	Сравнение результат >= операнд
EQ	**(**	Сравнение результат = операнд
NE	**(**	Сравнение результат <> операнд
LE	**(**	Сравнение результат <= операнд
LT	**(**	Сравнение результат < операнд
)		Оценка последней отложенной операции

Табл.26 Операторы и модификаторы для скока и вызова

ANY_BIT Операторы		
Операторы	**Модификатор**	**Описание функции**
JMP	**C, N**	Скок на сигнал
CAL	**C, N**	Вызов функционального блока
Func_name		Вызов функции
RET	**C, N**	Возвращение из функции или функционального блока

Модификатор **C** отмечает, что причисленная инструкция должна проводиться только в случае, если актуальный оцененный результат - „1“ Боола (или же „0“ Боола, если ператор дополнен модификатором **N**).

4.1.3 Определение пользовательской функции в языке IL

Пример 4.2 Пользовательская функции в языке IL

```
FUNCTION UserFun : INT
  VAR_INPUT
    val    :    INT;            // input value
    minVal :    INT;            // минимум
    maxVal :    INT;            // максимум
  END_VAR

        LD      val             // load input value
        GE      minVal          // проверка if val >= minVal
        JMPC    NXT_TST         // jump if OK
        LD      minVal          // low limit value
        JMP     VAL_OK
NXT_TST:
        LD      val             // load input value
        GT      maxVal          // проверка if val > максVal
        JMPCN   VAL_OK          // jump if not
        LD      maxVal          // high limit value
VAL_OK: ST      UserFun         // return value
END_FUNCTION
```

4.1.4 Вызов функций в языке IL

Функции в языке IL вызываются расположением названия функции в поле оператора. Актуальные результаты используются как первый параметр функции. Если требуются следующие параметры, они вносятся в поле операнда и разделяются медлу собой запятыми. Значение возвращенной функции после успешного проведения инструкции **RET** или после достижения физического конца функции станет актуальным результатом.

Следующие два возможных способа вызова отвечают вызову функции в языке ST. За названием функции в круглых скобках указан перечень параметров передаваемых функции. Перечень параметров может быть или же с введением названий параметров (formal call) или без них(informal call).

В примере 4.3 указаны все описанные способы вызова. Вызванная функция определена пользовательски в примере 4.2.

Пример 4.3 Вызов функции в языке IL

```
VAR_GLOBAL
  AT %YW10 : INT;
END_VAR

PROGRAM Example_IL1
  VAR
     count     : INT;
  END_VAR

        // calling function, first parameter is current result
        LD      Count
        UserFun 100, 1000
        ST      %YW10

        // calling function using an informal call
        UserFun( Count, 100, 1000)
        ST      %YW10

        // calling function using a formal call
        UserFun( val := Count, minVal := 100, maxVal := 1000)
        ST      %YW10

END_PROGRAM
```

4.1.5 Вызов функциональных блоков в языке IL

Функциональные блоки в языке IL вызываются условно или неусловно с помощью оператора **CAL** (Call). Как указано в нижеприведенном примере, функциональный блок можно вызвать двумя способами.

Функциональный блок можна вызвать его названием, за которым будет указан в скобке перечень параметров (formal call). Вторая мвозможность - запись параметров в соответствующие места в памяти инстанции функционального блока и его вызов (informal call). Оба способа можно комбинировать.

Пример 4.4 Вызов функционального блока в языке IL

```
VAR_GLOBAL
  in1   AT %X1.0   : BOOL;
  out1  AT %Y1.0   : BOOL;
END_VAR

PROGRAM Example_IL2
  VAR
    timer       : TON;
    timerValue  : TIME;
  END_VAR

  // calling FB using an informal call
  LD    in1
  ST    timer.IN        // parameter IN
  LD    T#10m12s
  ST    timer.PT        // parameter PT
  CAL   timer           // calling FB TON
  LD    timer.ET
  ST    timerValue      // timer value
  LD    timer.Q
  ST    out1            // timer output

  // calling FB using an informal call
  CAL   timer( IN := in1, PT := T#10m12s, Q => out1, ET => timerValue)

  // dather way
  LD    in1
  ST    timer.IN
  CAL   timer( PT := T#10m12s, ET => timerValue)
  LD    timer.Q
  ST    out1

END_PROGRAM
```

[TECO_HTML_TO_HTML_GENERATOR]
[TITLE] 4.2 Введение [/TITLE]
[GROUP] Язык ST [/GROUP]

[KEYWORDS] [/KEYWORDS]
[GLOBALS] [/GLOBALS]
[HIDDEN] [/HIDDEN]
[HIDDEN_GLOBALS] [/HIDDEN_GLOBALS]
[NOTHING] [/NOTHING]
[/TECO_HTML_TO_HTML_GENERATOR]

4.2 Язык структурированного текста ST

Язык структурированного текста является одним из языков, определенных нормой IEC 61 131-3. Это очень исполнительный высший язык программирования, который имеет корни в известных языках Ada, Pascal и C. Он объективно ориентирован и содержит все важные элементы современного языка программирования, в том числе разветвление (**IF-THEN-ELSE** а **CASE OF**) и итеративные контуры (**FOR**, **WHILE** и **REPEAT**). Эти элементы могут быть

внедрены. Этот язык является отличным инструментом для определения комплексных функциональных блоков.

Алгоритм записанный в языке ST можно разделить на отдельные ***команды*** *(statements)*. Команды используются для расчета и причисления значений, управления тока выполнения программы и для вызова или же окончания POU. Часть команды, которая расчитывает значения называется ***выражение***. Выражения производят значения, необходимые для проведения команд.

4.2.1 Выражения

Выражение - это конструкция, из которой после оценки сотрется значение соответствующее одному из типов данных, которые были определены в главе 3.2.

Выражение состоит из ***операторов и операнд***. Операндом может быть литеральная константа, переменная, вызов функции или друое выражение.

Операторы языка структурированного текста ST - это перечени, упорядоченные в Табл.4.5.

Табл.27 Операторы в языке структурированного текста ST

Оператор	Операция	Приоритет
( )	Скобки	Наивысший
**	Возведение в степень	
- **NOT**	Знак ДОполнение	
* / **MOD**	Умножение Деление Модуль	
+ -	Считывание Отсчитывание	
<, >, <= ,>=	Сравнение	
= <>	Равенство Неравенство	
&, AND	AND Боола	
XOR	Эксклюзивный OR Боола	
OR	OR Боола	Наинизший

Для операнды операторов действуют такие же ограничения как для входа соответствующих функций определенных в главе 3.4.1.1. Напр. результат оценки выражение **`A**B`** такой же, как результат оценки функции **`EXPT( A, B)`**, как это указано в Табл.3.14.

Оценка выражения состоит в приложение операторов на операнду, а именно - с учетом приоритета, выраженного в Табл.4.5. Сначала применяются операторы с наивысшим приоритетом в выражении, потом следующие операторы в направлении к более низкому приоритету до тех пор, пока оценка не будет закончена.Операторы сы одинаковм приоритетом оцениваются так, как это записано в выражении - в направлении слева на право.

Пример 4.5 Приоритет операторов при оценке выражений

```
PROGRAM EXAMPLE
  VAR                                  // локальные переменные
    A       : INT := 2;
    B       : INT := 4;
    C       : INT := 5;
    D       : INT := 8;
    X,Y     : INT;
    Z       : REAL;
  END_VAR

  X := A + B - C * ABS(D);             // X = -34
  Y := (A + B - C) * ABS(D);           // Y = 8
  Z := INT_TO_REAL( Y);
END_PROGRAM
```

Оценкой выражений **A + B- C * ABS(D)** в примере 4.5 получим значение –34. Если требуется другой порядок оценки, чем это указано, надо использовать скобки. Для одинакового значения переменных оценкам выражения **(A + B - C) * ABS(D)** получим значения 8.

Функции вызываются как элементы выражениеů, которые состоят из названия функции, за которымследует перечень аргументов в скобках.

Если оператор имеет две операнды, то самый левый оператор будет оцениваться как первый. Например выражение произведения двух тригонометрических функций **COS(Y) * SIN(X)** будет поэтому оценен в следующем порядке: расчет выражения **COS(Y)**, расчет выражения **SIN(X)** и только после этого расчет произведения **COS(Y) и SIN(X)**.

Выражения Боола могут оцениваться только в диапазоне, необходимом для получения однозначного конечного значения. Например если справедливо, что **C <= D**, то может быть из выражения **(C > D) & (F < A)** оценена только первая скобка **(C > D)**. Ее значение, принимая во внимание предположение – нулевое и поэтому достаточно для того, чтобы было нулевым все логическое произведение. Второе выражение **(F < A)** нет необходимости оценивать.

Если оператор в выражении может быть представлен как одна z přetíčtonýchфункция определенная в главе 3.4.1.1, происходит преобразование операнды и в результате согласно правилам и примерам, указанным в данной главе.

4.2.2 Совокупность команд в языке ST

Перечень команд языка структурированного текста ST - это совокупность указанная в Табл.4.6. Команды - это законченная точка с запятой. Со знаком конец строчка в данном языке ведет себя так же как с знаком пробела (space).

Табл.28 Перечень команд языка структурированного текста ST

Команда	Описание	Пример	Примечание
:=	Причисление	A **:=** 22;	Причисление значений расчитанных на правой стороне в идентификатор на левой стороне
	Вызов функционального блока	**Инстанции FB**(пара 1 := 10, пара 2 := 20);	Вызов функционального блока с передачей параметров
IF	Команда выбора	**IF** A > 0 **THEN** B := 100; **ELSE** B := 0; **END_IF;**	Подбор альтернативы в условном выражении BOOL
CASE	Команда выбора	**CASE** код **OF** 1 : A := 11; 2 : A := 22; **ELSE** A := 99; **END_CASE;**	Подбор блока команд условного значения выражения „код“

FOR	Итерактивная команда контур FOR	**FOR** i := 0 **TO** 10 **BY** 2 **DO**
		j := j + i;
		END_FOR;
			Многократный контур блока команд с начальным и конечным условием и значением отоэлектрона

WHILE	Итерактивная команда контур WHILE	**WHILE** i > 0 **DO**
		n := n * 2;
		END_WHILE;
			Многократный контур блока команд с условием окончания контура на начало

REPEAT	Итерактивная команда контур REPEAT	**REPEAT**
		k := k + i;
		UNTIL i < 20;
		END_REPEAT;
			Многократный контур блока команд с условием окончания контура в конце

EXIT	Окончание контура	**EXIT;**	Преждевременное окончание итерактивной команды
RETURN	Возвращение	**RETURN**;	Отход с правой выполненой POU и возвращение к вызываемой POU
;	Пустая команда	**;;**	

4.2.2.1 Команда причисления

Команда причисления заменяет актуальные значения простых или сложных переменных результатов, которые возникнут после оценки выражения. Команда причисления состоит из ссылки на переменную на левой стороне, за ней следует оператор причисления „:=“, за которым указано выражение, которое необходимо определить.

Команда причисления очень сильная. Может причислить простую переменную но и целую структуру. данных. Как указано на примере 4.6, где команда причисления **`A := B`** использованы для замены значений простые переменные **`A`** актуальные значения простых

переменных **B** (обе переменные основного типа **INT**). Но причисление можно удачно использовать такжеі для сложных переменных **AA := BB** и потом этой командой причисления переписываются все статьи сложной переменной **AA** статьями сложной переменной **BB**. Переменные должны быть одинакового типа данных.

Пример 4.6 Причисление простых и сложных переменных

```
TYPE
  tyRECORD : STRUCT
    SerialNo     : UDINT;
    color        : (red, green, white, blue);
    quality      : USINT;
  END_STRUCT;
END_TYPE

PROGRAM EXAMPLE
  VAR                          // локальные переменные
    A, B    : INT;
    AA, BB : tyRECORD;
  END_VAR

  A  := B;                        // причисление простых переменных
  AA := BB;                       // причисление сложных переменных
END_PROGRAM
```

Команда причисления может использовать также для причисления возвратные значения функции, а именно - расположенные названия функции на левой стороне оператора причисления в корпусе декларации функции. Величина возврата функции будет результатом последней оценки данной команды причисления.

Пример 4.7 Причисление возвращения значений функции

```
FUNCTION EXAMPLE : REAL
  VAR_INPUT                      // входные переменные
    F, G    : REAL;
    S       : REAL  := 3.0;
  END_VAR

  ПРИМЕР := F * G / S;          // величина возврата функции
END_FUNCTION
```

4.2.2.2 Команда вызова функционального блока

Функция может быть вызвана как составная часть оценки выражения, как было указано в данной главе, абзац выражение.

Функциональные блоки вызываются командой, которая состоит из названия инстанции функционального блока, за которым следует перечень названных входных параметров с причисленными значениями. Порядок, в котором параметры перечня при

вызове функционального блока указаны, не имеет заначения. При каждом вызове функционального блока не должны быть причислены все входные параметры. Если к какому-либо параметру не причислено значение перед вызовом функционального блока, используется значение причисленное как последнее (или начальное значение, если не было еще проведено какое-либо причисление).

Пример 4.8 Команда вызова функционального блока

```
// декларация функционального блока
FUNCTION_BLOCK fb_RECTANGLE
  VAR_INPUT
    A,B                : REAL;          // входные переменные
  END_VAR
  VAR_OUTPUT
    circumference, surface: REAL;       // выходные переменные
                                        // контур, площадь
  END_VAR

  circumference := 2.0 * (A + B); surface := A * B;
END_FUNCTION_BLOCK

// глобальные переменные
VAR_GLOBAL
  RECTANGLE : fb_RECTANGLE;             // глобальные инстанции FB
                                        // ПРЯМОУГОЛЬНИК
END_VAR

// декларация программы
PROGRAM main
  VAR
    o,s     : REAL;                   // локальные переменные
  END_VAR

  // вызов FB с полным переченем параметров
  RECTANGLE( A := 2.0, B := 3.0, circumference => o , surface => s);
  IF  o > 20.0 THEN
    ....
  END_IF;

  // вызов FB с неполным переченем параметров
  RECTANGLE( B := 4.0, A := 2.5);
  IF  RECTANGLE.circumference  > 20.0 THEN
    ....
  END_IF;
END_PROGRAM
```

4.2.2.3 Команда IF

Команда **IF** специфицирует, что необходимо проводить группе команд только в случае, если причисленное выражение Боота обозначено как верное (**TRUE**). Если условие

неверно, не проводится какая-либо команда или проводится группа команд, которые указаны за ключевым словом **ELSE** (или за ключевым словом **ELSIF**, если ему причисленое условие верно).

Пример 4.9 Команда IF

```
FUNCTION EXAMPLE : INT
  VAR_INPUT
    code      : INT;                      // входная переменная
  END_VAR

  IF code < 10 THEN EXAMPLE := 0;         // при code < 10 функция вернет 0
  ELSIF code < 100 THEN EXAMPLE := 1;     // при 9 < code < 100 вернет 1
  ELSE EXAMPLE := 2;                     // при code > 99 функция вернет 2
  END_IF;
END_FUNCTION
```

4.2.2.4 Команда CASE

Команда **CASE** содержит выражение, которое принадлежит к переменным типа **INT** (т. наз. „селектор“), и перечень группы команд, где каждая группа обозначена одним или более приложенными цифрами или диапазоном приложенных цифр. Этим выражается то, что будет проводиться первая группа команд, к пределам которой принадлежат расчитанное значение селектора. Если расчитанное значение не подходит ни к одной группе команд, будет проведена последовательность команд, которые указаны за ключевым словом **ELSE** (если в команде **CASE** имеется). В противном случае не будет проведена какая-либо последовательность команд.

Пример 4.10 Команда CASE

```
FUNCTION EXAMPLE : INT
  VAR_INPUT
    code     : INT;                 // входная переменная
  END_VAR

  CASE code OF
    10      : EXAMPLE := 0;       // при codee = 10 функция вернет 0
    20,77   : EXAMPLE := 1;       // при codee = 20 или code = 77  вернет 1
    21..55  : EXAMPLE := 2;       // при 20 < code < 56 функция вернет 2
    100     : EXAMPLE := 3;       // при codee = 100 функция вернет 3
  ELSE
              EXAMPLE := 4;       // в противном случае  функция вернет 4
  END_CASE;
END_FUNCTION
```

4.2.2.5 Команда FOR

Команда **FOR** используется, если количество итераций можно определить предварительно, в противном случае используются конструкции **WHILE** или **REPEAT**.

Команда **FOR** индицирует, что последовательность команд должна проводиться повторно до появления ключевого слова **END_FOR**, причем увеличиваются значения управляемых переменных контуров **FOR**. Переменная управления, исходное значение и конечное значение - это выражение одинакового типа интегров (**SINT**, **INT** или **DINT**) и не должен изменяться под влиянием каких-либо повторяемых команд. Команда **FOR** увеличивает или уменьшает значение переменной управления цикла от начального до конечного значения, а именно – после приращений, предназначенных значениям выражения (дефон это приращение равняется единице). Проверка условия окончания проводится на начале каждого повторения, поэтому если исходное значение переменной управления циклом превысит окончательное значение, последовательность команд не будет проведена.

Пример 4.11 Команда FOR

```
FUNCTION FACTORIAL : UDINT
  VAR_INPUT
    code    : USINT;                    // входная переменная
  END_VAR
  VAR_TEMP
    i       : USINT;                   // вспомогательная переменная
    tmp     : UDINT := 1;              // вспомогательная переменная
  END_VAR

  FOR i := 1 TO code DO
    tmp := tmp * USINT_TO_UDINT( i);
  END_FOR;
  FACTORIAL := tmp;
END_FUNCTION
```

4.2.2.6 Команда WHILE

Команда **WHILE** приведет к тому, что последовательность команд до ключевого слова **END_WHILE** будет повторяться до того времени, пока не будут причисленные выражения Боола неверными. Если причисленное выражение Боола в начале неверно, то последовательность команд не проводится совсем. Контур **FOR ... END_FOR** можно переписать с использованием конструкций **WHILE** … **END_WHILE**. Пример 4.12 можно с использованием команды **WHILE** переписать следующим образом:

Пример 4.12 Команда WHILE

```
FUNCTION FACTORIAL : UDINT
  VAR_INPUT
    code    : USINT;                    // входная переменная
  END_VAR
  VAR_TEMP
    i       : USINT;                   // вспомогательная переменная
    tmp     : UDINT := 1;              // вспомогательная переменная
  END_VAR

  i := code;
  WHILE i <> 0 DO
    tmp := tmp * USINT_TO_UDINT( i);  i := i - 1;
  END_WHILE;
  FACTORIAL := tmp;
END_FUNCTION
```

Если, команда **WHILE** использована в алгоритме, для которого не гарантировано выполнение условия окончания или проведения команды **EXIT**, то системы управления объявляют ошибку цикла.

4.2.2.7 Команда REPEAT

Команда **REPEAT** приведет к тому, что последовательность команд до ключевого слова **UNTIL** будет повторяться (как минимум один раз) до тех пор, пока не будут причисленные выражения Боола неверными. Контур **WHILE**…**END_WHILE** можно переписать с использованием конструкций **REPEAT** … **END_REPEAT**, что опять покажем на том же примере :

Пример 4.13 Команда REPEAT

```
FUNCTION FACTORIAL : UDINT
  VAR_INPUT
    code    : USINT;                    // входная переменная
  END_VAR
  VAR_TEMP
    i       : USINT := 1;              // вспомогательная переменная
    tmp     : UDINT := 1;              // вспомогательная переменная
  END_VAR

  REPEAT
    tmp := tmp * USINT_TO_UDINT( i);  i := i + 1;
  UNTIL i > code
  END_REPEAT;
  FACTORIAL := tmp;
END_FUNCTION
```

Если команда **REPEAT** использована в алгоритме, для которого не гарантировано выполнение условия окончания или проведения команды **EXIT**, то системы управления объявляют ошибку цикла.

4.2.2.8 Команда EXIT

Команда **EXIT** используется для окончания итераций перед выполнением условия окончания.

Если команда **EXIT** расположена внутри погруженной итеративной конструкции (команды **FOR**, **WHILE**, **REPEAT**), уход будет провиден из самых глубоких контуров, в которых **EXIT** расположен, то есть управление пердается на следующую команду за первым окончанием контура (**END_FOR**, **END_WHILE**, **END_REPEAT**), который следует за командой **EXIT**.

Пример 4.14 Команда EXIT

```
FUNCTION FACTORIAL : UDINT
  VAR_INPUT
    code    : USINT;                    // входная переменная
  END_VAR
  VAR_TEMP
    i       : USINT;                   // вспомогательная переменная
    tmp     : UDINT := 1;              // вспомогательная переменная
  END_VAR

  FOR i := 1 TO code
    IF i > 13 THEN
      tmp := 16#FFFF_FFFF; EXIT;
    END_IF;
    tmp := tmp * USINT_TO_UDINT( i);
  END_FOR;
  FACTORIAL := tmp;
END_FUNCTION
```

При расчете факториала для числа боле чем 13 будет результат больше чем максимальное число, которое можно записать в переменный тип **UDINT**. Эта ситуация в примере 4.14 решена с помощью команды **EXIT**.

4.2.2.9 Команда RETURN

Команда **RETURN** используетс для ухода функции, функционального блока или программы перед их окончанием.

В случае использования команды **RETURN** в функции необходимо отрегулировать выход функции (переменную, которая называется так же как функция) перед проведением команды **RETURN**. В противном случае не будут выходные значения функции определены.

Если будет команда **RETURN** использована в функциональном блоке, программатор должен обеспечить настройку выходных переменных функционального блока перед проведением команды. Ненастроенные выходные переменные будут иметь значения, соответствую-

щие инициализационному значению для соответствующего типа данных или значения настроенного в предыдущем вызове функционального блока.

Пример 4.15 Команда RETURN

```
FUNCTION FACTORIAL : UDINT
  VAR_INPUT
    code    : USINT;                    // входная переменная
  END_VAR
  VAR_TEMP
    i       : USINT;                   // вспомогательная переменная
    tmp     : UDINT := 1;              // вспомогательная переменная
  END_VAR

  IF code > 13 THEN FACTORIAL := 16#FFFF_FFFF; RETURN; END_IF;
  i := code;
  WHILE i <> 0 DO
    tmp := tmp * USINT_TO_UDINT( i);  i := i - 1;
  END_WHILE;
  FACTORIAL := tmp;
END_FUNCTION
```

При расчете факториала для числа большего чем 13 будет результат больше чем максимальное число, которое можно записать в переменный тип **UDINT**. Эта ситуация в примере 4.15 решена с помощью команды **RETURN**.

5 ГРАФИЧЕСКИЕ ЯЗЫКИ

В норме IEC 61 131-3 определены два графические языки: язык контактных схем (LD, Ladder Diagram, язык контактных схем) и язык функциональной блок-схемы (FBD, Function Block Diagram). Оба эти языки имеют поддержку в среде «Mosaic».

5.1 Общие элементы графических языков

Так же как и у текстовых языков содержит каждая декларация POU в графическом языке Декларационную и операционную часть. Декларационная часть совершенно похожа на текстовые языки, исполнительная часть разделена на т. наз. контуры (в английской литературе обозначаются ка networks). Каждый контур состоит из следующих элементов:

- сигналов контура
- коментариев контура
- графиуки контура

Наибольший контур

Каждый контур может быть оснащен сигналом, что ж является определенный пользователем идентификатор законченный знаком двоеточия. Сигнал потом может быть целью скока при разветвлении исполнительной программы POU. Диапазон действия контура и его сигналов *локальные* в рамках программные организационные единицы, в которые контур установлен. Сигналы контура не обязательны.

В среде программирования «Mosaic» кроме того каждый контур оснащен порядковым номером контура. Это генерируется автоматически и предназначено для лучшей ориентации в сложных POU. При вкладывании нового контура нижеприведенные контуры автоматически перечисляются. Графический эдитор потом позволяет быстрый поиск контуров в POU по их номерам.

Коментарий контура

Между сигналами контура и графикой контура можно поместить коментарий контура. Он может быть многострочный и может содержать знаки национальной азбуки. Коментарий контура не обязателен.

Графика контурu

Графика контура содержит графические элементы соединенные соединителями. Графическим элементом может быть например сцепление, блок таймера или выходная катушка. Соединители (соединительные линии) определяют ток информации, например из выхода таймера на выходную катушку. Каждый графический элемент может быть на выбор оснащен коментарием.

Направление тока в контурах

Графические языки используются для представления тока “в задуманном количестве” через один или более контуров представляющих алгоритм управления. Это задуманное количество можем понимать как:

- *“поток энергии”*, аналогичный потоку электроэнергии в электромеханических системах реле, который обычно используется в схемах реле
- *“поток сигнала”*, аналогичный потоку сигналов между элементами системы перерабатывающей сигналы, которые обычно используются в функциональных диаграмах блоков (схемах)

Соответствующее “задуманное количество” проходит по линиям между элементами сети согласно следующим правилам:

- Поток энергии в языке LD проходит слева направоо.
- Поток сигнала в языке FBD проходит от выхода (на правой стороне) функционального блока к входu (на левой стороне) следующего подключенного функционального блока

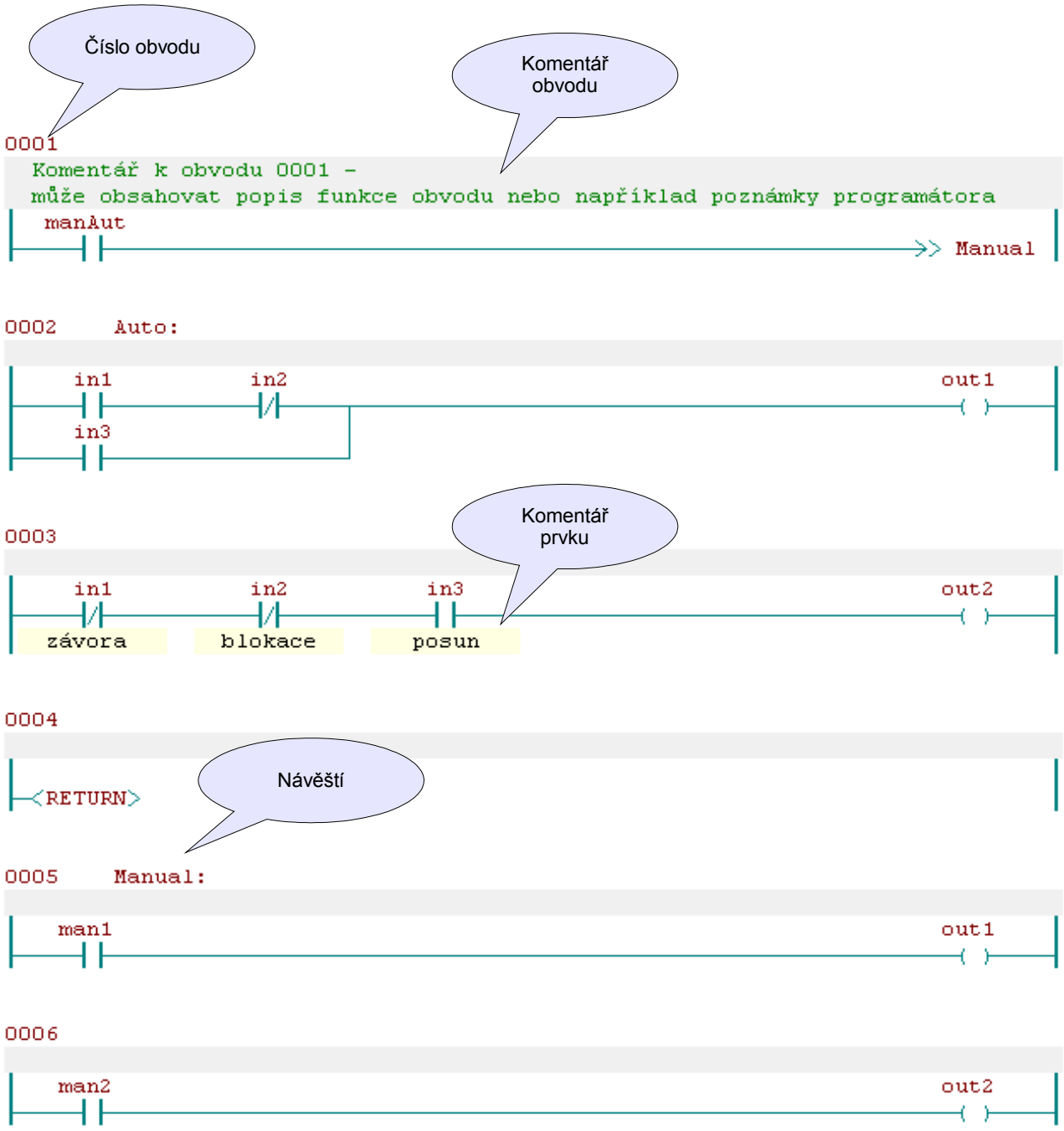


Рис. 12 Общие элементы графических языков

5.2 Язык контактных схем LD

Язык контактных схем (Ladder Diagram) проходит из электромеханических релейных контуров и основан на графическом представлении релейной логики. Этот язык первоначально преназначен для обработки сигналов Боола.

Как уже было сказано, исполнительная часть POU в языке LD состоит из контуров (networks). Контур в языке LD ограничен т. наз. питательными сборными шинами (power rails) на левой и правой стороне. С левой сборной шины „ведет“ логическая едничка (TRUE) во все к ней присоединенные графические элементы, типичные соединительные и разъединительные контакты. В зависимости от их состояния потом логическая единица пропускается или непропускается в следующий элемент подключенный в контуре. Последний элемент вправо бывает выходным и подключен к правой сборной шине. Типическим представителем выходного элемента является катушка.

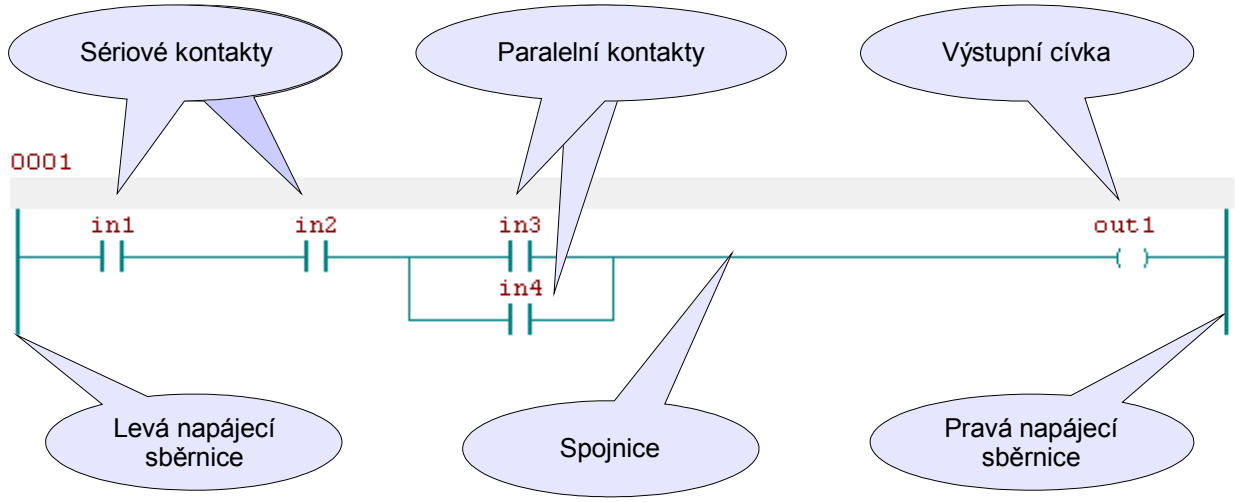


Рис. 13 Серийные и параллельные соединения элементов в контуеи

5.2.1 Графические элементы в языке LD

Контур в языке LD может содержать нижеприведенные графические элементы :

- питательная сборная шина
- соединители
- контакты и катушки
- графические элементы для управления работой программы (скоки)
- графические элементы для вызова функций или функциональных блоков

Графические элементы могут быть соединены серийно или парнольно. На рис. 5.2 переменные **`in1`** и **`in2`** соединены в серии (AND) с парнольно соединенными переменными **`in3`** и **`in4`** (OR). Эти переменные называются контактами и программа проверяет их значение (читает их). Переменная **`out1`** называется катушкой и программа на нее проводит запись.

Контур на рис.5.2 реализует выражение **`out1 := in1 AND in2 AND (in3 OR in4);`**

5.2.1.1 Питательная сборная шина

Контур в языке LD ограничен слева вертикальной линией, которая называется *левая питательная сборная шина*, и направо вертикальной линией, которая называется *правая питательная сборная шина*. Состояние левой питательной сборной шины всегда “ON”. Для правой питательной сборной шины не определено какое-либо состояние

5.2.1.2 Соединители в языке LD

Элементы соединителей могут быть расположены горизонтально или вертикально. Состояние соединителя может быть обозначено “ON” или “OFF”, а именно - согласно его значению по Боолю - 1 или 0. Понятие *состояние соединителя* – это синоним к понятию *ток энергии*.

Горизонтальные соединители индицированы горизонтальной линией. Горизонтальные соединители передают состояние элемента, который находится непосредственно влево, к элементу, который находится непосредственно вправо от него.

Вертикальные соединители состоят из вертикальной линии перекрещивающейся один или более раз с горизонтальными соединителями с каждой стороны. Состояние вертикальных соединителей представляет включающее OR состояние ON горизонтальных соединителей по его левой стороне, то есть состояние вертикальных соединителей будет:

- OFF, если состояние всех присоединенных горизонтальных соединителей с их левой стороны OFF
- ON, если состояние одного или более присоединенных соединителей с их левой стороны ON

Состояние вертикального соединителя копируется во все присоединенные горизонтальные соединители направо от них. Состояние вертикальных соединителей не копируется ни в одну из присоединенных горизонтальных соединителей влево от нее.

Табл.29 Соединители в языке FBD

Графический объект	Наименование	Функции
	Горизонтальные соединители	Горизонтальные соединители копируют состояние элемента присоединенного на левой стороне к элементу присоединенному на правой стороне
	Вертикальные соединители с горизонтальным присоединением	Состояние горизонтального соединителя влево копируется во всех горизонтальных соединителях вправо
	Вертикальные соединители с горизонтальным присоединением OR	Состояние горизонтального соединители вправо является результатом логической функции OR состояния всех горизонтальных соединителей влево

5.2.1.3 Контакты и катушки

Контакт позволяет логическую операцию между состоянием горизонтального соединителя слева и переменную, которая ке контакту причислена. Тип логической операции зависит от типа контакта. Итоговое значение передается на соединители вправо. Контакт не влияет на значение причисленной переменной Боола.

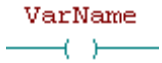
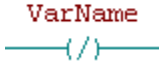
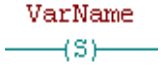
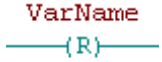
Контакты могут быть сцепляющими или расцепляющими. Сцепляющий контакт так же как электромеханический контакт в состоянии покоя расцеплен (значение переменной FALSE) и после подведения напряжения сцепляется (значенийа переменной TRUE). Функции расцепляющего контакта имеет обратное значение. В состоянии покоя (без подведения напряжения) контакт сцеплен (т.е. проверочное значение TRUE) и после подведения напряжения контакт расцепляется (проверочное значение FALSE). Функции контактов в языке LD объяснены в табл.5.2.

Табл.30 Контакты в языке LD

Графический объект	Наименование	Функции
	Сцепляющий контакт (Open contact)	Правые соединители := леввые соединители AND VarName; (Копируют состояние левого соединителя в правый соединитель если состояние переменной **VarName** - **TRUE**, в противном случае - в правый соединитель записывается **FALSE**)
	Расцепляющий контакт (Closed contact)	Правые соединители := левые соединители AND NOT VarName; (Копируют состояние левого соединителя в правый соединитель если состояние переменной **VarName** - **FALSE**, в противном случае - в правый соединитель записывается **FALSE**)

Катушка копирует состояние соединителя влево от нее в соединитель вправо и одновременно запишет это состояние в причисленную переменную Боола. Типы катушек и их функции указаны в табл.5.3.

Табл.31 Катушки в языке LD

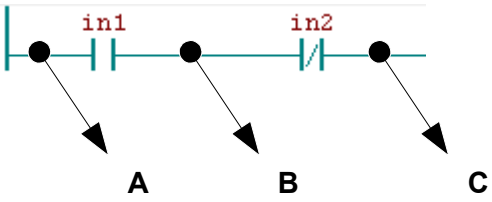
Графический объект	Наименование	Функции
	Катушка (Coil)	Переменная := левый соединитель; (Копирует состояние левого соединителя в переменной **VarName** и одновременно в правый соединитель)
	Отрицательная катушка (Negated coil)	Переменная := NOT левый соединитель; (Копирует состояние левого соединителя в переменной **VarName** и одновременно в правый соединитель)
	Катушка Set (Set coil)	Отрегулирует переменные **VarName** значенийu **TRUE** v случае, что состояние левой соединители je **TRUE**, в противном случае оставит переменную в первоначальном состоянию. Состояние правой соединители копируют состояние левой соединители.
	Катушка Reset (Reset coil)	Отрегулирует переменные **VarName** значенийu **FALSE** v случае, что состояние левой соединители je **TRUE**, в противном случае оставит переменную в первоначальном состоянию. Состояние правой соединители копируют состояние левой соединители.

Оценка потока энергии в контурах

Поток энергии в контурах оценивается слева слева направо. При расчете программф отдельные контурф в POU оцениваются в порядке сверху вниз.

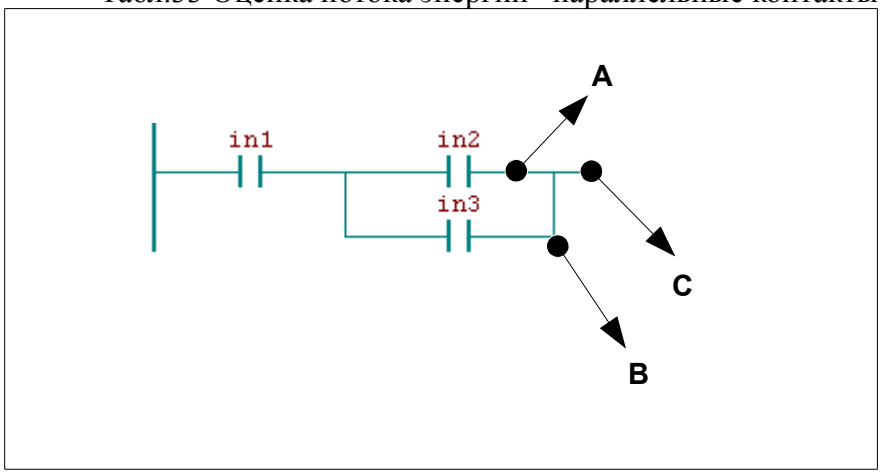
Пример оценки серийных контактов указан в табл.5.4. Указанный контур реализует выражение **`C := in1 AND NOT in2`**. Пример оценки парнольных контактов указан в табл.5.5. Указанный контур реализует выражение **`C := in1 AND (in2 OR in3)`**.

Табл.32 Оценка потока энергии – серийные контакты



in1	in2	NOT in2	A	B	C
0	0	1	1	0	0
0	1	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	0

Табл.33 Оценка потока энергии– параллельные контакты



in1	in2	in3	A	B	C
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

5.2.1.4 Управление проведением программы в языке LD

Для управления проведением программы в языке LD имеем две возможности : скок на определенный контур в актуальное POU и окончания POU. Графические символу указаны в табл.5.6.

Скоки изображаются горизонтальной линией, законченной двойной стрелкой. Передача управления программе на предназначенные сигналы контура произойдет, если значение Боола соединительной линии 1 (TRUE). Соединительная линия для условия скока может начинаться у переменной Боола, у выхода Боола функции или функционального блока или на левой питательной сборной шины. Безусловный скок является поэтому специальным случаем условного скока. Целью скока является сигнал сети в рамках программной организационной единицы, в которой скок появится. Невозможно скакать, это возможно только в рамках одной POU.

Условное возвращение из функций и функциональных блоков проводится использованием конструкций RETURN. Проведение программы передается обратно в вызывающую POU, если вход Боола 1 (TRUE). Проведение программы будет продолжаться в нормальном ходе, если вход Боола имеет значение 0. Безусловное возвращение возникнет на

физическом конце функции или функционального блока или с помощью элемента RETURN, который присоединен к левой питательной сборной шине.

Табл.34 Передача управления программы в языке LD

Графический объект	Наименование	Функции
	Безусловный скок (jump)	Скок на контур с сигналом **Label**
	Условный скок (Conditional jump)	Скок на контур с сигналом **Label** если переменная **VarName** имеет значение **TRUE**, в противном случае программа продолжается в решении нижеприведенного контура

Графический объект	Наименование	Функции
	Безусловное возвращение из POU (Return)	Завершит POU и вернет управление в вызывающий POU. POU также завершен, если будут решены все ее контуры
	Условное возвращение из POU (Conditional return)	Завершит POU а вернет управление в вызывающий POU если переменная **VarName** имеет значение **TRUE**, в противном случае программа продолжает решать нижеприведенный контур

5.2.1.5 Вызов функций и функциональных блоков в языке LD

Язык LD поддерживает вызов функций и функциональных блоков. Вызываемые POU в схеме представлены прямоугольником. Входные переменные представлены соединителями слева, выходные переменные - соединителями справа. Названия входных и выходных формальных параметров указаны внутри прямоугольника напроти соединителям, через которые присоединяются актуальные значения параметров (переменные или постоянные величины). У расширенных функций (напр. ADD, XOR, и т.д.) названия входных параметров не приводятся. Наименование функции или тип функционального блока указан в верхней части прямоугольника. Наименование инстанции функционального блока указано над прямоугольником. Прямоугольники функций изображены зеленым цветом, функциональные блоки - синим.

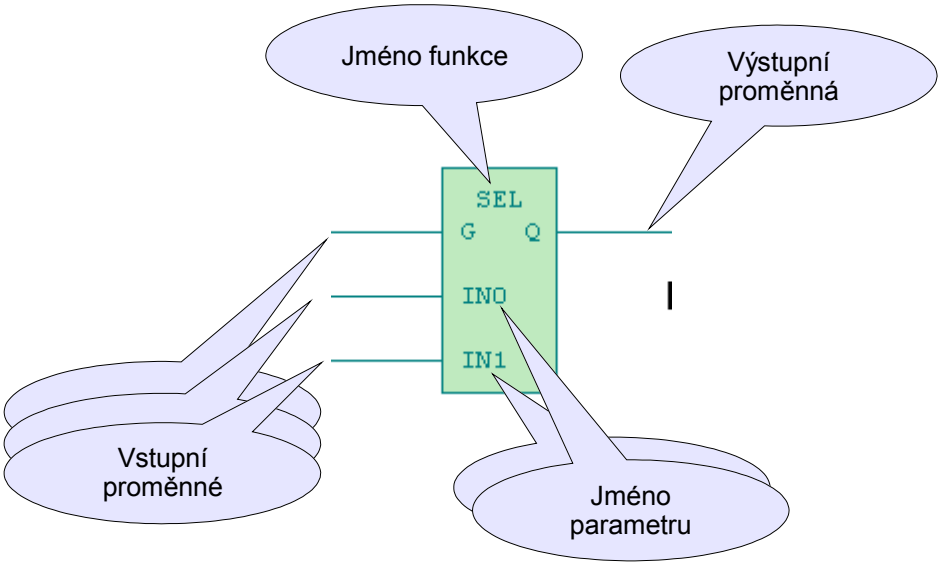
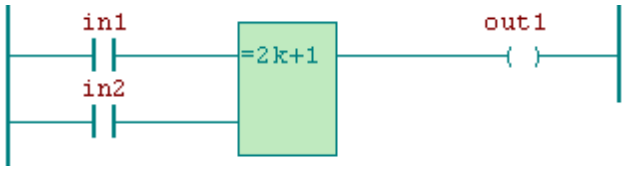
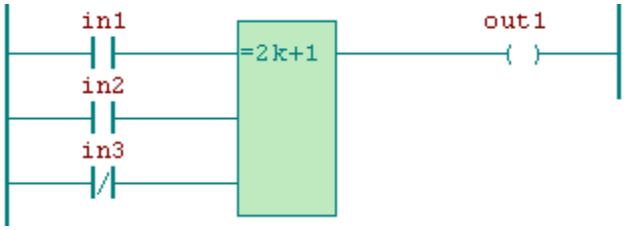
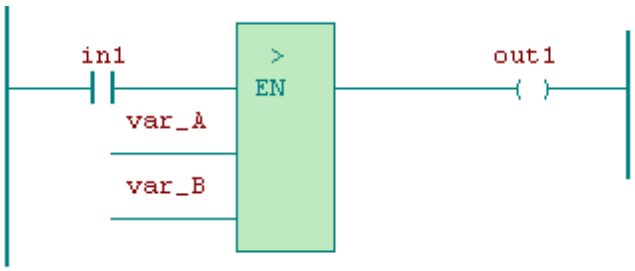
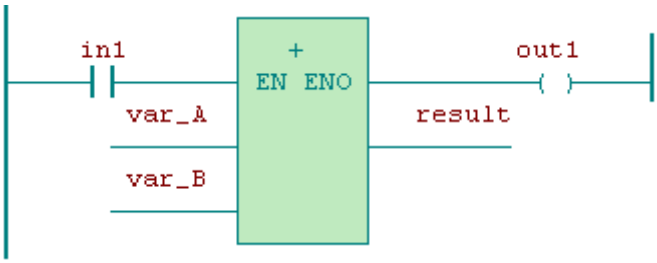


Рис. 14 Графическая репрезентация функции в языке LD

Вызов функций

Если функции имеют хотя бы один вход типа BOO, этот вход присоединен к левому соединителю в контуре. Если функция имеет выход типа BOOL этот выход присоединен к правому соединителю в контуре. В противном случае для подключения функции к контуру использованы неявной переменной Боола EN и ENO. EN - входная переменная типа BOOL, которая обуславливает вызов функции. Если на вход EN подведено значение TRUE, вызов функции будет проведен. В противном случае функция не будет вызвана. В каждом случае значение входа EN копируются на выход функции ENO. Подключение выхода ENO не обязательно. Использование EN / ENO типично например в случае арифметических функций.

Табл.35 Вызов функций в языке LD

Контур	Описание
	Вызов стандартных функций **XOR** Контур реализует выражение **out1 := IN1 XOR in2**
	Вызов стандартных функций **XOR** с расширенным количеством входов Контур реализует выражение **out1 := in1 XOR in2 XOR NOT in3**
	Вызов функции **GT** с использованием неявного входа **EN**. Неявный выход **ENO** не используется. Если вход **EN** имеет значение **TRUE** контур реализует выражение **out1 := var_A > var_B** В противном случае значение переменной **out1** не учитывается.
	Вызов функции **ADD** с использованием неявного входа **EN** и неявного выхода **ENO**. Если вход **EN** имеет значение **TRUE** контур реализует выражение **result := var_A + var_B** В противном случае значение переменной **result** не учитывается. Выход **ENO** копирует состояние входа **EN**.

Вызов функциональных блоков

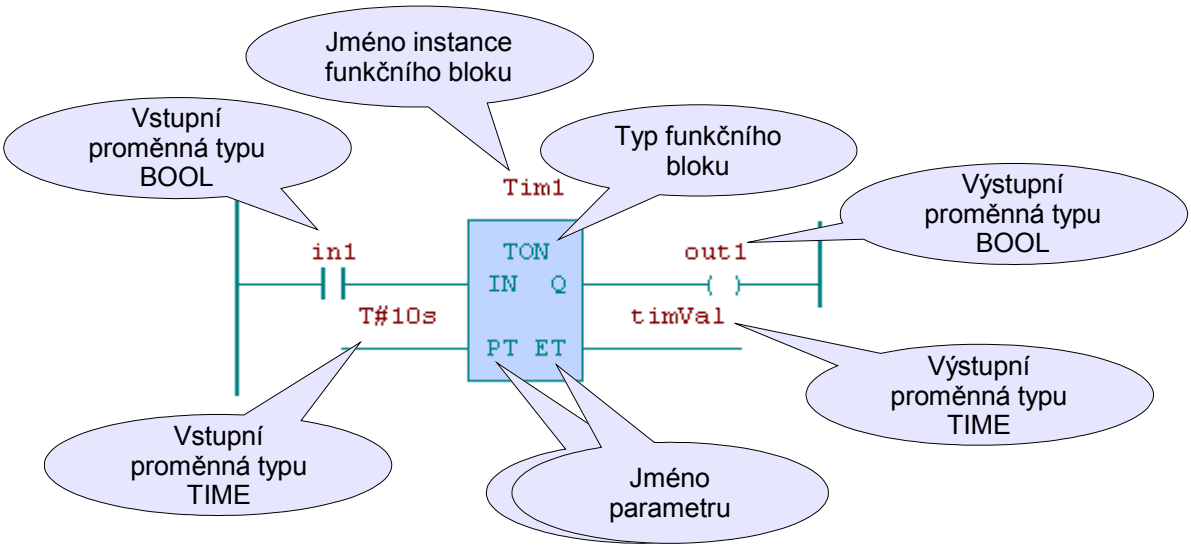
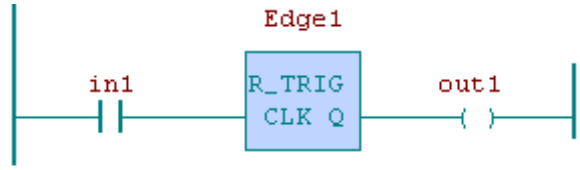
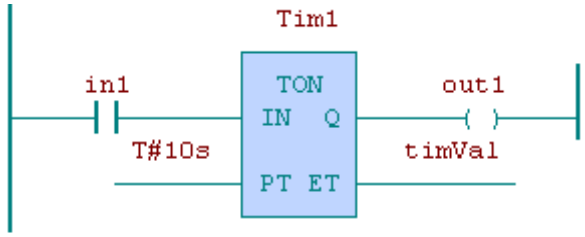
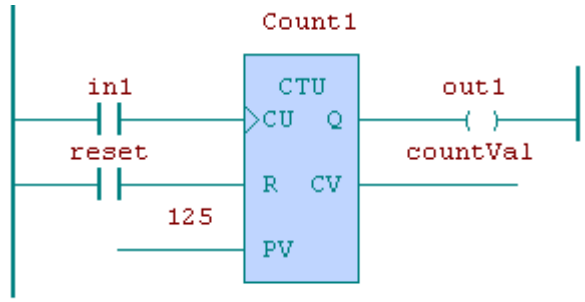
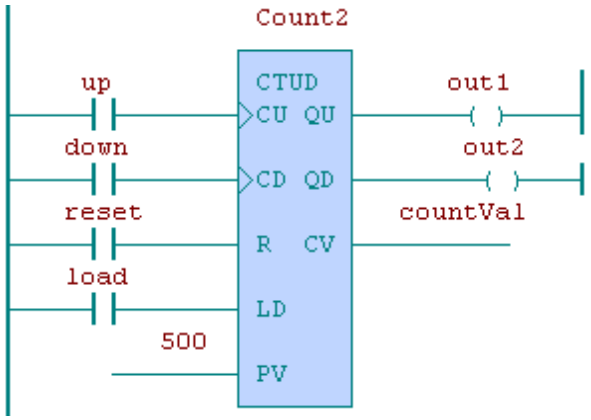


Рис. 15 Вызов функционального блока в языке LD

Для вызова функциональных блоков в языке LD действуют такие же правила, как для вызова функций. Чтобы можнобыло подключить функциональный блок к контуру в языке LD, должен существовать один из входов типа BOOL (так как ток сигнала в LD контура начинается на левой питательном соединителе, на который можно присоединить только элементы BOOL). Если функциональный блок не имеет вход типа BOOL, можно использовать неявный вход EN (enable), который обуславливает выполнение функционального блока. Этот вход имеет автоматически все функции и функциональные блоки, что обеспечивает среду программирования. Вход EN будет в распоряжении и для функциональных блоков определенных пользователем, и в случае, если определение блока такой вход не приводит. То же самое распространяется на неявный выход ENO (Enable Output). Так же как и функция значение входа EN копируется на выход ENO.

Табл.36 Вызов функциональных блоков в языке LD

Контур	Описание
	Вызов стандартного функционального блока **R_TRIG** Выход **out1** настроен только на случай перехода переменные **in1** из значения **0** на значение **1** (передний торец)
	Вызов стандартного функционального блока **TON** Входная переменная **PT** (передвыбор таймера) типа **TIME** и поэтому не подключен к левой питательной сборной шине. В данном случае в данную переменную записывается постоянная величина **T#10s** (10 секунд)
	Вызов стандартного функционального блока **CTU** Вход **CU** в функциональный блок **CTU** определен следующим образом: **VAR_INPUT** **CU : BOOL R_EDGE;** **END_VAR** По этой причине входные соединители данного сигнала закончены знаком оценка переднего торца 
	Вызов стандартного функционального блока **CTUD** Входы **CU** и **CD** типа **BOOL** с детекцией передних торцов. Вход **PV** (Preset Value) не принадлежит к типу **BOOL** и поэтому не подключен к питательной сборной шине. В данном случае к данному входу записывается постоянная величина **500**. То же самое касается выхода **CV** если он не принадлежит к типу **BOOL** и также не подключен к питательной сборной шине. Его значение записывается к переменной **countVal**.

5.3 Язык функциональной блок-схемы FBD

Язык функциональной блок-схемы (Function Block Diagram) основан на соединении функциональных блоков и функций. Так же как в языке LD и в языке FBD функции и функциональные блоки представлены прямоугольником. Разница состоит в том, что в языке LD можно соединители между элементами переносить только указанного типа BOOL в то время как в языке FBD могут соединители между графическими элементами переноситься значения любого типа.

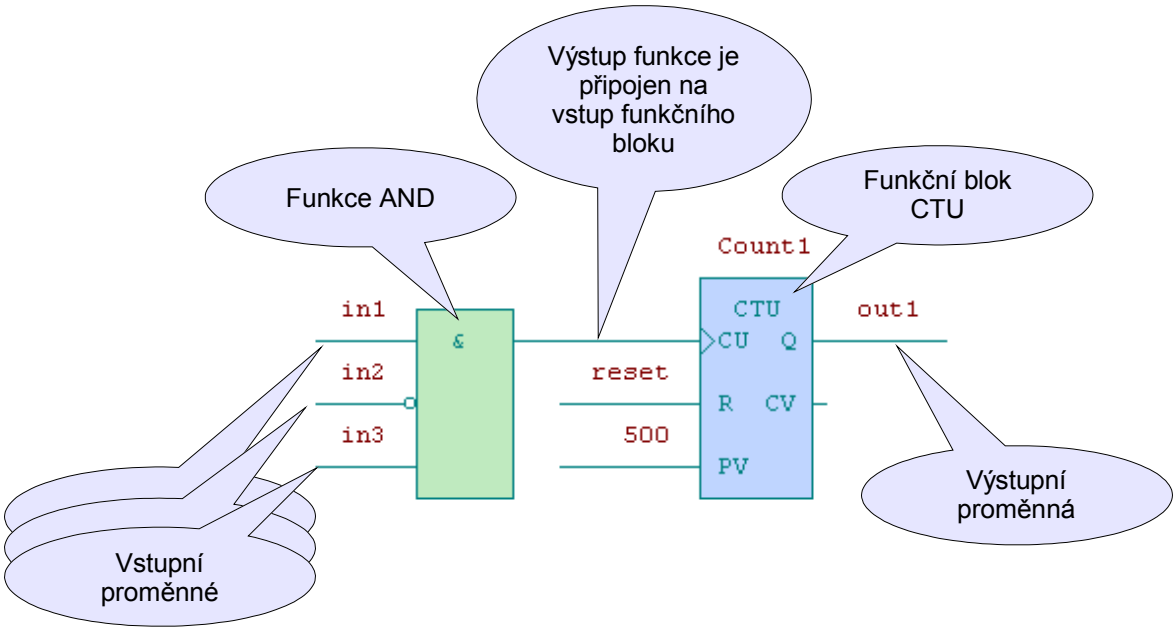


Рис. 16 Графика контура в языке FBD

5.3.1 Графические элементы в языке FBD

Контур в языке FBD может содержать нижеприведенные графические элементы :

- соединители
- графические элементы для управления проведения программы (скоки)
- графические элементы для вызова функций или функциональных блоков

Язык FBD не содержит какие-либо следующие графические элементы такие как контакты или катушки в языке LD. Элементы языка FBD поддерживаются соединителями тпотока сигнала. Выходы функциональных блоков не соединены между собой. Особенно конструкции “wired OR” языка LD не разрешаются в языке FBD. Вместо этого используется блок Боока OR.

Контур в языке FBD может быть начерчен двумя способами, как это указано на рис.5.6. Способ изображения можно в любой момент переключить. Контур реализует выражение **`result := (var1 + var2) – var3`**.

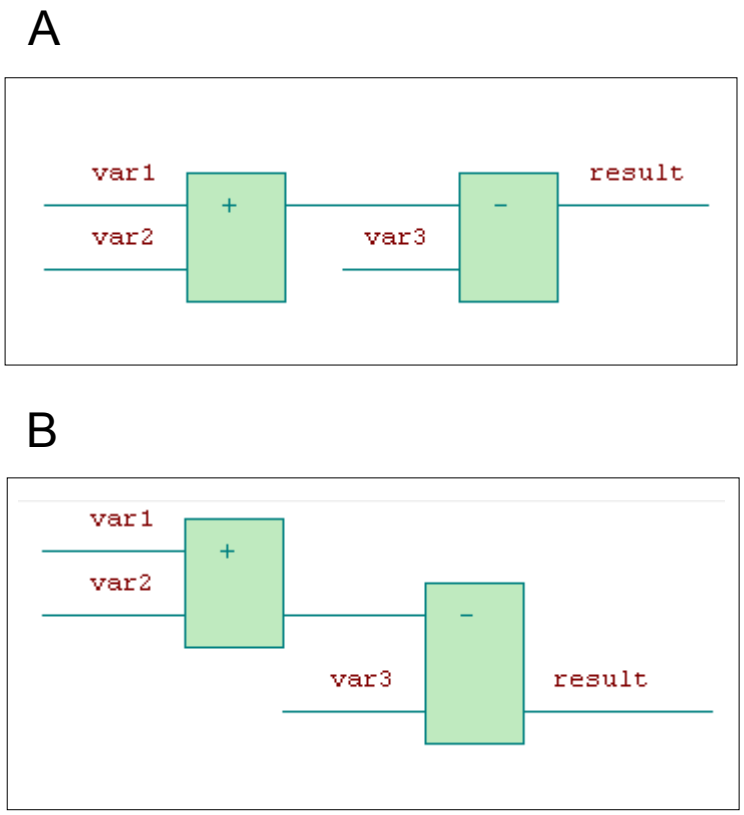


Рис. 17 Способы изображения контура в языке FBD

Оценка потока сигнала в контурах

Поток сигнала в контурах оценивается слева направо. При расчете программы потом отдельные контуры в POU оцениваются в порядке сверху вниз. В контуре на рис.5.6 будут сначала считаны переменные **`var1`** и **`var2`** и после этого будет отчислена переменная **`var3`**. Результат будет записан в переменные **`result`**.

5.3.1.1 Соединители в языке FBD

Элементы соединителей могут быть горизонтального или вертикального исполнения. Состояние соединителя представляет значение присоединеной переменной. Понятие *состояние соединителя* является синонимом понятия *поток сигнала*.

Горизонтальные соединители индицированы горизонтальной линией. Горизонтальные соединители передают состояние элемента, который находится непосредственно влево, к элемента, который находится непосредственно вправо от него.

Вертикальные соединители состоят из вертикальной линии присоединяющей один или более горизонтальных соединителей к правой стороне. Состояние вертикального соединителя копируется во все присоединенные горизонтальные соединители справа от не-

го. Состояние вертикальных соединителей не копируется к каким-либо присоединенным горизонтальым соединителям слева отнего.

Табл.37 Соединители в языке FBD

Графический объект	Наименование	Функции
	Горизонтальные соединители	Горизонтальные соединители копируют состояние элемента присоединенного на левой стороне к элементу присоединенному на правой стороне
	Вертикальные соединители с горизонтальным присоединением	Состояние горизонтального соединители влево копируется во все горизонтальные соединители справа
	Неразрешенная конструкция	Эта конструкция (обозначается как wired OR) не разрешена в языке FBD. Вместо нее можно использовать стандартную функцию OR Боола.

5.3.1.2 Управление проведением программы в языкеFBD

Для управления проведением программеы так же как и в языке LD имеется две возможности : скок на определенный контур в актуальные POU и окончания POU. Графические символы для языка FBD указаны в табл.5.10.

Скоки изображаются горизонтальной линией законченной двойной стрелкой. Передача управления программы на предназначенные сигналы контура произойдет, если значение Боола соединительной линии равняется 1 (TRUE). Соединительная линия при условии скоku может начинаться у переменные Боола или у выхода Боола функции или функциональной блока. Если условие не указано, речь идет о неусловный скок. Целью скока является сигнал сети в рамках программной организационной единицы, в которой скок появится. Невозможно проводить скоки в иных случаях, чем в рамках одной POU.

Условное возвращение из функций и функциональных блоков внедряется использованием конструкций RETURN. Проведение программы передается оборатно на POU, если вход Боола равен 1 (TRUE). Проведение программы будет продолжаться в нормальном ходе, если вход Боола имеет значение 0. Неусловное возвращение возникнет на физическом конце функции или функционального блока или с помощью неусловного элемента RETURN.

Табл.38 Передача управления программы в языке FBD

Графический объект	Наименование	Функции
	Неусловный скок (jump)	Скок на контур с сигналом **Label**
	Условный скок (Conditional jump)	Скок на контур с сигналом **Label** если переменная **VarName** имеет значение **TRUE**, в противном случае программа продолжает решать нижеприведенный контур
	Неусловное возвращение из POU (Return)	Закроет POU и вернет управление в вызывающий POU. POU тоже закроется, если будут решены все ее контуру
	Условное возвращение из POU (Conditional return)	Закроет POU а вернет управление в вызывающий POU если переменная **VarName** имеет значение **TRUE**, в противном случае программа продолжает решать нижеприведенные контуры

5.3.1.3 Вызов функций и функциональных блоков в языке FBD

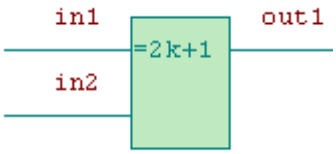
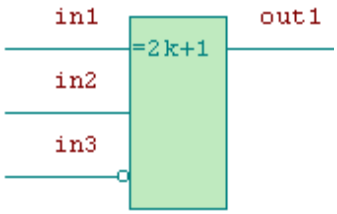
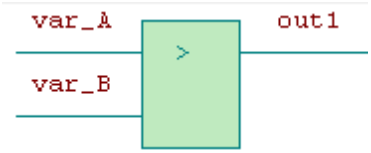
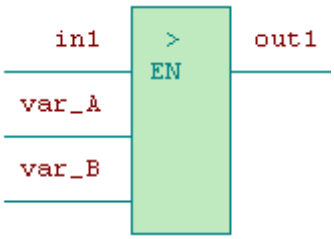
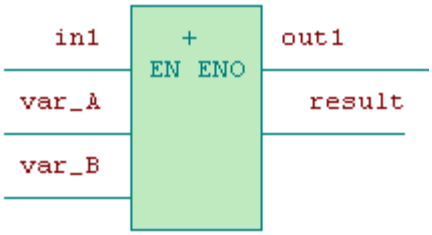
Графическая репрезентация функций и функциональных блоков очень похожа. Эти POU в схеме представлены прямоугольником так же как в языке LD. Входные переменные представлены соединителями слева, выходные переменные - соединителями справа. Названия входных и выходных формальных параметров указаны внутри прямоугольника напроти соединителям, через которые подключаются актуальные значения параметров (переменные или постоянные величины). У расширенных функций (напр.ADD, XOR, и т.д.) названия входных параметров не приводятся. Наименование функции или тип функционального блока указан в верхней части прямоугольника. Наименование инстанции функционального блока указано над прямоугольником. Прямоугольники функций обозначены зеленым цветом, функциональные блоки- синим.

В языке FBD не должны функции или функциональный блок иметь какой-либо вход типа BOOL для того, чтобы его можно было подключить к контуру. В принципе, нет необходимости использовать неявный вход EN, но это не запрещено. То же самое касается также неявного выхода ENO. Если EN и ENO используются, их значение и поведение такое же как в языке LD.

Вызов функций

Примеры вызова функций в языке FBD указаны в табл.5.11.

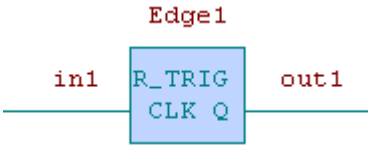
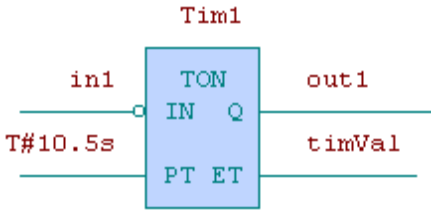
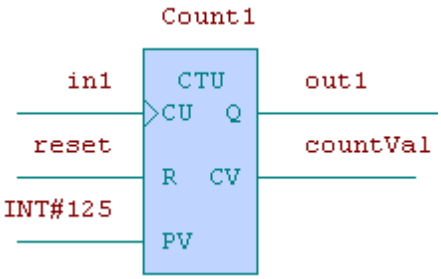
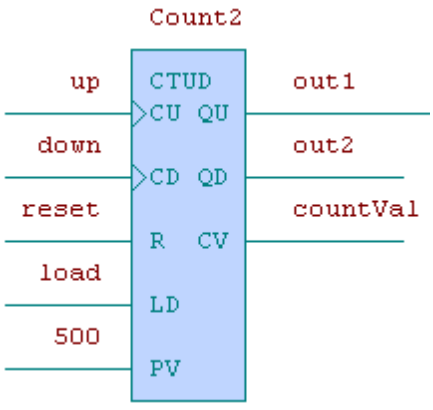
Табл.39 Вызов функций в языке FBD

Контур	Описание
	Вызов стандартной функции **XOR** Контур реализует выражение **out1 := IN1 XOR in2**
	Вызов стандартной функции **XOR** с расширенным количеством входов Контур реализует выражение **out1 := in1 XOR in2 XOR NOT in3**
	Вызов функции **GT** без использования **EN** и **ENO** Контур реализует выражение **out1 := var_A > var_B**
	Вызов функции **GT** с использованием неявного входа **EN**. Неявный выход **ENO** не используется. Если вход **EN** имеет значение **TRUE** контур реализует выражение **out1 := var_A > var_B** В противном случае значение переменной **out1** не учитывается.
	Вызов функции **ADD** с использованием неявного входа **EN** и неявного выхода **ENO**. Если вход **EN** имеет значение **TRUE** контур реализует выражение **result := var_A + var_B** В противном случае значение переменной **result** не учитывается. Выход **ENO** копирует состояние входа **EN**.

Вызов функциональных блоков

Примеры вызова функций в языке FBD указаны в табл.5.12.

Табл.40 Вызов функциональных блоков в языке FBD

Контур	Описание
	Вызов стандартного функционального блока **R_TRIG** Выход **out1** настроен только на случай перехода переменной **in1** из значения **0** в значение **1** (передний торец)
	Вызов стандартного функционального блока**TON** Входная переменная **in1** отрицательна. Входная переменная **PT** (передвыбор таймера) типа **TIME** и в данную переменную записывает постоянные величины **T#10.5s** (10,5 секунды)
	Вызов стандартного функционального блока **CTU** Вход **CU** в функциональном блоке **CTU** определена следующим образом: **VAR_INPUT** **CU : BOOL R_EDGE;** **END_VAR** По этой причине входные соединители данного сигнала закончены знкаом оценки переднего тоорца 
	Вызов стандартного функционального блока **CTUD** Входы **CU** и **CD** типа **BOOL** с детекцией передних торцов. Во вход **PV** (Preset Value) записывается постоянная величина **500**. Выход **CV** (Counter Value) записывается в переменную **countVal**.

6 ПРИЛОЖЕНИЯ

6.1 Директивы

Программы записанные на некотором из текстовых языков могут содержать директивы для программы перевода, которые позволяют управлять ее работой. Директивы записываются в фигурные скобки.

Например директива **{$DEFINE new_name}** определяет наименование new_name.

6.1.1 Директива PUBLIC

Директива **{PUBLIC}** предназначена для оообозначения общественной переменной. Описание обозначенной таким образом переменной будет при переводе записано в массив с аффиксом „.pub“, который предназначен для переноса определений переменных в программы визуализациии т.п.

Эти директивы можно использовать в рамках декларации типа данныхили в рамкахдекларация переменные.

Синтаксис записи ниже приведен:

```
TYPE MyINT {PUBLIC} : INT; END_TYPE
VAR
  Var1 {PUBLIC} : BOOL;
  Var2 {PUBLIC} AT %R2000 : BYTE;
END_VAR
```

6.1.2 Директивы для условного первода программ

Для условного первода программы введены нижеприведенные директивы:

```
{$IF <vyraz>}
{$IFDEF <name>}
{$IFNDEF <name>}
{$DEFINED <name>}
{$UNDEF <name>}
{$END_IF}
{$ELSE}
{$DEFINED(<name>)}
{$ELSEIF <name>}
```

Эти директивы можно использовать как в Декларационной так и в исполнительной части программы.

6.1.2.1 Директивы $IF … $ELSE … $END_IF

Директива **{$IF <выражение>}** предназначена для условного первода если удовлетворено выражение. Может быть на выбор условлено и ответвление **{$ELSE}**. Условно переводимая часть программы закончена директивой **{$END_IF}**. Выражение должно содержать только переменные определенные как **VAR_GLOBAL CONSTANT**, постоянные величины, или **{$DEFINED(<name>)}**. Операторы в выражении могут быть только:

'>' - больше
'<' - меньше
'=' - равняется
NOT - отрицание в выражении
AND - произведение Боола
OR - сумма Боола
')' - скобка
'(' - скобка

Синтаксис записи:
{$IF <выражение>} .... [{$ELSE}....] {$END_IF}

6.1.2.2 Директивы $IFDEF и $IFNDEF

Эти директивы предназначены для условного первода. Программа, которая следует директиву **{$IFDEF <name>}** переводится в случае, если наименование указанное в директиве существует (определено). Наоборот программа указанная за директивой **{$IFNDEF <name>}** будет переведена только в случае, если наименование указанное в директиве не определено. Эти директивы можно комбинировать с директивами **{$ELSE}** и **{$ELSEIF}** и создавать таким образом альтернативно переводимые части программ. Конец условного перевода обозначен директивой **{$END_IF}.**

Синтаксис записи:
{$IFDEF <name>} .... [{$ELSE}....] {$END_IF}
{$IFNDEF <name>} .... [{$ELSE}....] {$END_IF}

6.1.2.3 Директивы $DEFINED и $UNDEF

Эти директивы предназначены для добавления или отмена определения названия. Директива **{$DEFINED <name>}** добавит определение названия **<name>**. Наименование потом можно использовать в директивах **{$IFDEF <name>}** и **{$IFNDEF <name>}**. Директива **{$UNDEF <name>}** отменит определение названия, указанного в директиве.

Синтаксис записи:

```
{$DEFINED <name>}
{$UNDEF <name>}
```

6.1.2.4 Директива DEFINED

Данная директива предназначена для проверки действительности определения названия **<name>** и ее можно использовать в комбинации с директивой **{$IF <выражение>}** как составную часть выражения.

Синтаксис записи:

```
DEFINED (name)
```

Пример:

```
{$IF DEFINED( ALFA) OR DEFINED( BETA)}
  VAR counter : INT; END_VAR
{$ELSE}
  VAR counter : DINT; END_VAR
{$END_IF}
```

6.1.3 Директивы ASM и END_ASM

Директива **{ASM}** предназначена для вкладывания программы в мнемокод в программе на некоторых из IEC языков. Конец укладываемого мнемокода обозначен директивой **{END_ASM}**.

Синтаксис записи:

```
{ASM}
{END_ASM}
```

6.1.4 Директива ST_WARNING

Директива **{ST_WARNING}** предназначена для подавления предупредительных сообщений программы перевода ST. Директива **{ST_WARNING OFF}** обозначает место в программе, с которого будут предупредительные сообщения ST программы перевода подавляться. Директива **{ST_WARNING ON}** обозначит место в программе, с которого будут предупредительные сообщения ST программы перевода опять выдаваться.

Синтаксис записи:

```
{ST_WARNING ON}
{ST_WARNING OFF}
```

6.1.5 Директива OFFSET_REG

Директива **{OFFSET_REG=10000}** предназначена настройке базового адреса в памяти %R (%M), куда будут сняты переменные и инстанции. Первая переменная будет размещена на адресе на один уровень выше, чем это указано в директиве (%R10001). Директива **{END_OFFSET_REG}** закончит смещеннное размещение переменных. Размещение переменных в памяти ПЛК будет после этого продолжаться на адресе на 2 уровня выше, чем была перед использованием директивы **{OFFSET_REG=..}**.

Внимание!
Эти директивы выведут из работы автоматический контроль перекрытия адресов переменных. При перекрытии переменных не будет программа перевода сообщать о возникновении какой-либо ошибки!

Синтаксис записи:
{OFFSET_REG=xxx} где xxx адрес %R, где начнет новую съемку
{END_OFFSET_REG}

6.2 Забронированные ключевые слова

В нижеприведенной таблице указаны ключевые слова, использование которых забронировано для языка программирования IEC 61 131-3 и их нельзя использовать для пользовательских определенных символов.

Табл.41 Забронированные ключевые слова

A	ABS ANY ANY_NUM AT	ACOS ANY_BIT ANY_REAL ATAN	ACTION ANY_DATE ARRAY	ADD ANY_INT ASIN
B	BOOL	BY	BYTE	
C	CAL CD CONFIGURATION CTU	CALC CDT CONSTANT CTUD	CALCN CLK COS CU	CASE CONCAT CTD CV
D	D DINT DT	DATE DIV DWORD	DATE_AND_TIME DO	DELETE DS
E	ELSE END_CONFIGURATION END_IF END_STEP END_VAR EQ	ELSIF END_FOR END_PROGRAM END_STRUCT END_WHILE ET	END_ACTION END_FUNCTION END_REPEAT END_TRANSITION EN EXIT	END_CASE END_FUNCTION_BLOCK END_RESOURCE END_TYPE ENO EXP

	EXPT			
F	FALSE FOR	F_EDGE FROM	F_TRIG FUNCTION	FIND FUNCTION_BLOCK
G	GE	GT		
I	IF INT	IN INTERVAL	INITIAL_STEP	INSERT
J	JMP	JMPC	JMPCN	
L	L LEFT LN LWORD	LD LEN LOG	LDN LIMIT LREAL	LE LINT LT
M	MAX MOVE	MID MUL	MIN MUX	MOD
N	N	NE	NEG	NOT
O	OF	ON	OR	ORN
P	P PV	PRIORITY	PROGRAM	PT
Q	Q	QI	QU	QD
R	R READ_WRITE REPLACE RETC ROL R_EDGE	RI REAL RESOURCE RETCN ROR	R_TRIG RELEASE RET RETURN RS	READ_ONLY REPEAT RETAIN RIGHT RTC
S	S SEMA SINGLE SR STRING	ST SHL SINT ST STRUCT	SD SHR SL STEP SUB	SEL SIN SQRT STN
T	TAN TIME_OF_DAY TON TYPE	TASK TO TP	THEN TOD TRANSITION	TIME TOF TRUE
U	UDINT USINT	UINT	ULINT	UNTIL
V	VAR VAR_INPUT	VAR_ACCESS VAR_IN_OUT	VAR_EXTERNAL VAR_OUTPUT	VAR_GLOBAL
W	WHILE	WITH	WORD	
X	XOR	XORN		

СОДЕРЖАНИЕ