# Структурированный язык управления SCL

Структурированный язык управления SCL (Structured Control Language) является языком программирования высокого уровня для SIMATIC S7. Язык базируется на стандарте DIN EN 61131-3 (часть "Structured Text" ["Структурированные тексты"]) и имеет сертификат совместимости с PLC Base Level [Базовый уровень] версии V4.01 при использовании интернациональных мнемоник (в данной книге изначально использованы мнемоники, принятые в Германии). Язык SCL оптимизирован для программирования программируемых контроллеров (PLC). SCL содержит в себе элементы языка Паскаль (Paskal) наряду с типичными для PLC элементами, такими, например, как "вход" ("input") и "выход" ("output").

SCL особенно подходит для программирования сложных алгоритмов или для задач, относящихся к области управления данными. Язык SCL поддерживает характерную для STEP 7 блочную структуру, а также позволяет создавать S7-программы, включающие в себя фрагменты на базовых языках программирования STL, LAD и FBD.

Программное обеспечение S7-SCL является опционным (то есть, поставляемым по отдельному заказу) программным продуктом. ПО S7-SCL может быть поставлено вместе с базовым пакетом STEP 7 Basic Package. Описание S7-SCL в данной книге базируется на версии языка SCL V.5.1

При инсталляции ПО S7-SCL данный язык программирования полностью интегрируется с утилитой SIMATIC Manager и может после этого использоваться наряду с базовыми языками программирования (например, STL). Используя редактор SCL-программ, пользователь может создавать в S7-проекте исходные программы, которые он должен затем скомпилировать с помощью SCL-компилятора. Программа пользователя содержит скомпилированные SCL-блоки; эта программа, кроме того, может содержать скомпилированные блоки, написанные на других языках программирования. Пользователю также предоставляется возможность протестировать блоки, созданные с использованием языка SCL, в интерактивном (online) режиме в CPU с использованием отладчика SCL Debugger.

**Элементы языка SCL** в синтаксисе инструкций отличаются от элементов других (базовых) языков программирования (имеются в виду операторы, выражения, присвоение значений). Однако, все они совместно используют типы данных, адресные области, символьные имена и блочную структуру.

Используя **управляющие операторы (control statements)**, пользователь может организовывать ветвление программы, выполнять программные циклы. С помощью использования операций переходов можно прерывать

последовательное выполнение программы, а затем продолжать ее выполнение, начиная с другой точки блока.

С использованием SCL пользователь может запрограммировать **блоки** и затем - организовать их вызов в программе (как говорится, вставить блоки в программу), что может быть выполнено как с использованием языка SCL, так и с использованием другого языка программирования для S7. Используя язык SCL, Вы можете получить доступ к любой системной функции.

Стандартные функции, такие как функции преобразования, доступны в виде SCL-функций; кроме того, пользователь, используя языки SCL или STL, может запрограммировать свои собственные функции. И, наконец, для обработки переменных сложных типов пользователь в своей программе может использовать IEC-функции из базового пакета программного обеспечения STEP 7 Basic Package.

**27 Введение, элементы языка**

Интеграция с системой SIMATIC; адресация; операторы; выражения; присвоение значений.

**28 Операторы управления**

IF, CASE, FOR, WHILE, REPEAT, CONTINUE, EXIT, GOTO, RETURN.

**29 SCL-блоки**

Вызовы блоков; передача параметров; переменная OK; механизм EN/ENO.

**30 SCL-функции**

Функции таймеров; функции счетчиков; функции преобразования; математические функции; функции сдвига и циклического сдвига; программирование собственных функций пользователя с использованием языков программирования SCL и STL.

**31 IEC-функции**

Функции преобразования; функции сравнения; STRING-функции (функции для работы с переменными типа STRING); функции обработки даты и времени; функции для работы с числами.

# 27 Введение. Элементы языка

В данной главе рассматриваются требования, которые должны выполняться при программировании на языке SCL. В главе 2 "Программное обеспечение STEP 7" и в главе 3 "SIMATIC S7-программа" содержится детальное описание данного вопроса, поэтому Вы найдете здесь ссылки на эти главы.

В главе 2 "Программное обеспечение STEP 7" предоставлено введение в вопросы, касающиеся средств программирования, таких как, редактор символов (symbol editor), редактор SCL-программ (SCL program editor), компилятор и отладчик. В главе также говорится о программно-аппаратной среде программирования на языке SCL.

В главе 3 "SIMATIC S7-программа" представлена структура программы пользователя. В главе описываются различные варианты выполнения программы, структура блоков, а также перечислены все требуемые для программирования блоков ключевые слова. Вы найдете там также введение в темы "адресация переменных" и "типы данных, поддерживаемые STEP 7".

Примеры, рассматриваемые в данной главе, Вы можете найти на прилагаемой дискете в библиотеке STL_Book library в разделе "27 Language Elements" ("Элементы языка").

## 27.1 Интеграция с SIMATIC

### 27.1.1 Инсталляция (установка)

Инсталляция средств программирования на языке SCL требует наличия утилиты SIMATIC Manager соответствующей версии. SCL устанавливается посредством программы установки SETUP; при инсталляции всех поддерживаемых языков и примеров программирования для установки требуется около 8 Мб на жестком диске. Кроме того, Вам потребуется также выполнить авторизацию SCL (подтвердить права на использование), для чего служит специальная дискета.

### 27.1.2 Создание проекта

Утилита SIMATIC Manager является ключевым средством также и для работы с языком программирования SCL. Для того, чтобы создать программу на SCL, пользователь должен сначала запустить SIMATIC Manager и создать проект также, как и при создании программ на одном из стандартных языков программирования (см. раздел 2.1 "Базовый пакет STEP 7 (STEP 7 Basic Package)"). При этом пользователь может создать проект "вручную" или воспользоваться для его создания специальной программой Wizard -"мастером создания проектов".

При конфигурировании станции достаточно назначить CPU, чтобы SIMATIC Manager создал разделы для связанной с проектом S7-программы. Можно также создать S7-программу непосредственно в проекте и назначить CPU позднее.

Кроме того, можно также использовать уже готовый проект. Необходимо, чтобы были созданы разделы: *S7 Program [S7 программа], Blocks [Блоки],* а также *Symbols [Символы]* для таблицы символов. Если какой-либо объект не существует, необходимо создать его. Для этого нужно выделить раздел более низкого уровня и выбрать опцию меню *Insert (Вставка)*.

Если используется уже существующий проект, то в этом случае в нем уже могут присутствовать исходные программы на STL или скомпилированные блоки, созданные, скажем, с помощью FBD. Это не нарушает работы SCL-редактора. Пользователь может даже вызывать в своей SCL-программе ранее созданные и скомпилированные блоки, независимо от языка, который использовался для их создания.

### 27.1.3 Редактирование SCL-программы

Выделите раздел *Sources [Исходные программы]* и затем выберите опции меню: *Insert -> SCL Source (Вставка -> Исходная программа SCL)* (такая опция меню будет доступна для использования, только в том случае, если Вы установили SCL в своей системе). Теперь Вы можете переименовать вставленный объект *SCL Source(1).* Дважды щелкнув кнопкой мыши на этом объекте, вызовите редактор SCL-программ, который отобразит при открытии на экране монитора "пустой" исходный SCL-файл. Теперь Вы можете вводить SCL-программу.

Как использовать для ввода программ SCL-редактор, описывается в разделе 2.5.4 "Редактор SCL-программ (SCL-Program Editor)". Написание программы начинается с ввода (редактирования) блока. Структура блока и необходимый набор ключевых слов описываются в разделе 3.5 "Программирование кодовых блоков на SCL".

Здесь для начала мы рассмотрим простой пример: мы запрограммируем функцию ограничителя "Delimiter", которая должна ограничивать входные величины, адаптируя их к заданному диапазону значений (между верхним и нижним пределами); кроме того, в примере мы запрограммируем вызов этой функции в организационном блоке (см. рис. 27.1).

```
FUNCTION Delimiter : INT
VAR_INPUT
   MAX : INT;                              //максимальное значение
   IN  : INT;                              //входное значение
   MIN : INT;                              //минимальное значение
END_VAR
BEGIN
IF IN > MAX THEN Delimiter := MAX;      //адаптация к верхнему значению
   ELSIF IN < MIN THEN Delimiter := MIN;//адаптация к нижнему значению
   ELSE Delimiter := IN;                //допустимое входное значение
END_IF;
END_FUNCTION

ORGANIZATION_BLOCK Mainjprogram
VAR_TEMP
   SINFO : ARRAY [1..20] OF BYTE;
END_VAR
BEGIN
Result := Delimiter (MAX := Maximum, IN := INPUT_VALUE, MIN := Minimum);
END_ORGANIZATION_BLOCK
```

Рис. 27.1 Пример программы функции ограничителя "Delimiter"

Пример программы начинается с объявления типа блока ограничителя "Delimiter" (функция FC) и типа для значения функции (INT). Далее следует описание параметров блока: параметры MAX (максимальное значение, верхняя граница), IN (величина входного сигнала) и MAX (максимальное значение, нижняя граница) объявляются входными параметрами (INPUT), относящимися к типу данных INT. Сама программа следует за разделом объявления переменных. В соответствии с программой, если входная величина IN больше, чем заданная максимальная величина, то функция принимает значение, равное максимальной величине. Если это не так, и входная величина IN меньше, чем заданная минимальная величина, то функция принимает значение, равное минимальной величине. Если оба рассмотренных случая не дали положительного результата, то функция принимает значение, равное входной величине IN.

Теперь мы запрограммируем вызов этой функции в организационном блоке "Main Program". Программируя на языке SCL, Вы должны также зарезервировать 20 байтов в области временных локальных данных для стартовой информации организационного блока, даже если Вы не планируете их использовать.

В отличие от стандартных языков программирования, в языке SCL функция FC с возвращаемым значением функции является

"реальной" функцией, которая вставляется в выражение вместо адреса (идентификатора), и при этом обеспечивается совместимость типов данных. При вызове в организационном блоке "Main Program" функции ограничителя "Delimiter" ее значение присваивается глобальной переменной "Result" ("Результат"); теперь эта переменная будет содержать в себе значение "Input_value", ограниченное предельными значениями "Maximum" и "Minimum".

Исходная SCL-программа может содержать, как минимум, один блок. Вы можете также создать несколько исходных SCL-программ, которые затем должны скомпилировать в определенном порядке с использованием управляющего файла компилятора.

После написания исходной SCL-программы сохраните ее, используя опции меню: *File -> Save (Файл -> Сохранить)*. Так как в примере мы использовали символьную адресацию, то мы должны заполнить таблицу символов (Symbol Table) перед компилированием исходной программы.

### 27.1.4 Заполнение таблицы символов (Symbol Table)

Заполнение таблицы символов (Symbol Table) в SCL производится так же как и в стандартных языках программирования (см. раздел 2.5.2 "Таблица символов (Symbol Table)"). Вы также можете вводить информацию в уже существующую и частично заполненную таблицу символов. При этом в любой данной S7-программе может присутствовать только одна единственная таблица символов. Таблица символов в разделе *S7 Program [S7 программа]* представляется объектом *Symbols [Символы].*

Вы можете вызвать редактор символов (symbol editor), выбрав опции меню: *Options -> Symbol Table (Опции -> Таблица символов)* в редакторе SCL-программ или дважды щелкнув кнопкой манипулятора "мышь" на объекте *Symbols [Символы]* в окне утилиты SIMATIC Manager. После того, как откроется таблица (пустая, новая или ранее частично заполненная), Вы можете ввести символьные имена Ваших параметров (см. табл. 27.1). После заполнения таблицы, ее необходимо сохранить.

Таблица 27.1 Таблица символов (Symbol Table) для примера функции ограничителя "Delimiter"

| Symbol (символ) | Address (адрес) | Data Type (тип данных) | Comment (комментарий) |
|---|---|---|---|
| Main program | OB1 | OB1 | Исполняемый блок циклически выполняемой программы |
| Delimiter | FC271 | FC271 | Функция для ограничения переменной типа INT |
| Input_value | MW10 | INT | Величина входного сигнала |
| Maximum | MW12 | INT | Верхний предел |
| Minimum | MW14 | INT | Нижний предел |
| Result | MW16 | INT | Сигнал на выходе из функции ограничения |

### 27.1.5 Компилирование SCL-программы

Для того, чтобы выполнить компилирование исходной SCL-программы откройте раздел *Sources [Исходные программы]* (если он еще не открыт). Компилятор Вы можете найти, используя опции меню: *Options -> Customize* (*Опции -> Установки пользователя*) на вкладке "Compiler" ["Компилятор"]. (При необходимости создания блоков выберите опцию "Blocks generated" ("Создание блоков")).

Компилирование исходной SCL-программы может быть выполнено с помощью опций: *File -> Compile (Файл -> Компиляция)*; скомпилированные блоки сохраняются в разделе *Blocks [Блоки].* Более подробная информация о компиляции блоков представлена в разделе 2.5.4 "Редактор SCL-программ (SCL-Program Editor)".

Вы можете организовать пакетный режим компиляции сразу нескольких исходных программ; использование специального управляющего файла позволяет проводить компиляцию этих исходных файлов в любом порядке.

Необходимо отметить, что вызываемые блоки или функции должны быть доступны в момент компилирования или как уже скомпилированные блоки, хранящиеся в разделе *Blocks [Блоки]*, или как (заведомо не имеющие ошибок) исходные программы, хранящиеся в разделе *Sources [Исходные программы],* или как стандартные функции, хранящиеся в библиотеке стандартов Standard Library.

### 27.1.6 Загрузка SCL-блоков

Если программатор (программирующее устройство PLC) подключен к CPU, то с помощью опций меню: *PLC -> Load (PLC -> Загрузка)* Вы можете загрузить скомпилированные блоки в пользовательскую память CPU. Сам CPU должен быть при этом в режиме STOP (СТОП), потому что порядок следования загружаемых блоков может отличаться от порядка следования вызовов блоков.

Более подробная информация об этом и других аспектах, на которые Вы должны обратить Ваше внимание, представлена в разделе 2.6 "Интерактивный режим (Online mode)".

Вы можете также обрабатывать блоки в "интерактивном" ("online") и в "автономном" ("offline") окнах утилиты SIMATIC Manager.

### 27.1.7 Тестирование SCL-блоков

SCL-отладчик позволяет проводить тестирование отдельных блоков с помощью функции "Program Status" ("Состояние программы") в режиме последовательного мониторинга выбранных регистров и переменных или в пошаговом режиме. С помощью функции "Program Status" ("Состояние программы") Вы можете наблюдать, как меняются значения переменных в

процессе выполнения программы. В пошаговом режиме Вы можете останавливать выполнение программы в точке прерывания и выполнять программу оператор за оператором, отслеживая состояние переменных (см. раздел 2.7 "Тестирование программы").

Таблица переменных (VAT) также может использоваться для тестирования SCL-программы. С помощью этой таблицы Вы можете устанавливать значения переменных и затем наблюдать при выполнении программы результаты таких назначений.

### 27.1.8 Адреса и типы данных

**Адресные области**

Адреса и переменные, применяемые при программировании на языке SCL, соответствуют адресам и переменным, применяемым при написании программ на стандартных языках программирования (см. раздел 1.5 "Адресные области"):

- входы I, выходы Q, меркеры M;
- периферийные входы PI;
- периферийные выходы PQ;
- адреса глобальных данных D;
- временные и статические локальные данные (только символьная адресация);
- организационные блоки OB, функциональные FB, функции FC как с возвращаемым значением, так и без него; блоки данных DB.

Функции таймеров T и функции счетчиков C обрабатываются в SCL-программах как "стандартные функции" (см. раздел 30.1 "Функции таймеров T" и раздел 30.2 "Функции счетчиков C").

Примечание:

Адреса глобальных данных имеют отличающиеся адресные идентификаторы по сравнению со стандартными языками программирования. Более подробное изложение вопроса, посвященного адресным идентификаторам в SCL, представлено в разделе 27.2.1 "Абсолютная адресация".

В SCL функции, которые возвращают "значение функции", могут использоваться в выражениях как адреса.

**Типы данных**

Назначение типа данных определяет:

- тип и значение (компонентов) данных (например, integer [целая], character string [строка символов]);
- разрешенные диапазоны (например, числовой диапазон, длина строки символов);
- разрешенные операции, допустимые для обработки данных определенного типа;

- способ написания констант.

Типы данных, применяемые при программировании на языке SCL, такие же как те, что применяются при написании программ на стандартных языках программирования (в разделе 3.7 "Переменные и константы" представлен соответствующий обзор в табличной форме, а в главе 24 "Типы данных" Вы найдете детальное описание вопроса).

Численные значения могут быть представлены как десятичные числа, как шестнадцатеричные числа, как восьмеричные числа (8#17 соответствует 16#F или 15dec) и как двоичные числа.

**Классы типов данных**

В связи с возможностью группирования значений, в SCL определяют классы типов данных, которые представляют одинаковое поведение внутри одного класса:

- класс ANY_INT включает в себя данные типов INT и DINT;
- класс ANY_NUM включает в себя данные типов INT, DINT и REAL;
- класс ANY_BIT включает в себя данные типов BOOL, BYTE, WORD и DWORD.

Указанные классы типов данных были введены, чтобы сделать яснее описание операторов; переменные не могут быть описаны с помощью данных классов типов данных.

**Запись констант**

Константы - это фиксированные значения, которые в общем случае не изменяются при выполнении программы. Константы используются для предопределения начальных значений переменных при описании последних или для объединения (комбинирования) их в программе с другими переменными (например, при применении в качестве граничных значений).

В языке SCL константа не определяет "свой" тип данных, пока она не будет обработана в арифметической операции. Например, константа 1234 может относиться к типу данных INT или к типу данных REAL, в зависимости от применения:

```
int1    := int2 + 1234;   //константа INT
real1   := real2 + 1234;  //константа REAL
```

В языке SCL Вы можете назначать тип данных для константы (так называемая запись константы со спецификацией типа - "type-defined"). Используя соответствующий префикс, Вы можете, например, предопределить переменную WORD в разделе объявлений с помощью десятичного, шестнадцатеричного, восьмеричного или двоичного числа. Ниже представлен пример, в котором переменная, имеющая в каждом из случаев одинаковое значение, имеет различное представление:

```
W1   :  WORD  :=  W#1234 ;                  //десятичное
W2   :  WORD  :=  W#16#04D2 ;               //шестнадцатеричное
W3   :  WORD  :=  W#8#2322 ;                //восьмеричное
W4   :  WORD  :=  W#2#0000_0100_1101_0010; //двоичное
```

**Тип данных при абсолютной адресации**

Абсолютный адрес всегда принадлежит к классу типов данных ANY_BIT (например, двойное слово меркеров MD10 имеет тип данных DWORD). Операнд может иметь тип данных (отличный от ANY_BIT) только в случае, если имеет символьное имя ("когда он обращен к переменной"), или после преобразования типа данных.

```
MW14    := SHL(IN := MW12, N := 2);
real1   := real2 + DWORD_TO_REAL(MD10);
```

**Тип данных STRING**

Строка символов должна вводиться в одинарных кавычках. С данным типом могут также использоваться непечатаемые управляющие символы; они должны вводиться в формате $hh (здесь hh означает значение ASCII символа в шестнадцатеричной форме).

```
string1  := '$0A$0D';   //новая строка
```

Для продолжения строки символов на следующей строке или для вставки комментария в разрыв строки символов (если не нужно отображать этот комментарий при печати и при выводе на дисплей) предназначены две специальные комбинации символов '$>' и '<$':

```
string2  := 'ABCDEFGHIJKLMNOP$>'//продолжение следует
            '<$QRSTUVWXYZ';
```

## 27.1.9 Виды типа данных (Data Type Veiws)

В SCL Вы можете назначать дополнительные типы данных для уже объявленных переменных, или, более точно, Вы можете назначить дополнительные виды типа данных (data type veiws). При этом становится возможным обращаться к содержимому переменной в целом и по частям как к данным, относящимся к разным типам.

Пример:

Пусть, Вы объявляете входной параметр с именем *Station* и типом STRING. Вы должны переслать данную переменную *Station* в вызываемый блок для того, чтобы выполнить обработку, например, путем добавления численного значения. Кроме того, Вам необходимо высчитать текущую длину *Station*. Для этой цели Вы применяете дополнительный вид типа данных, например, в форме структуры из двух байтов. Первый байт этой структуры содержит максимальную длину строки, второй байт этой структуры содержит текущую длину строки. Дополнительный вид типа данных принимает имя *Len,* а компоненты структуры соответственно называются *max* и *cur.*

```
VAR_INPUT
   Station : STRING24 := ' ';
```

```
Len AT Station : STRUCT
   max : BYTE;     //Максимальная длина
   cur : BYTE;     //Текущая длина
   END_STRUCT;
END_VAR
...
IF WORD_TO_INT(Len.cur) > 12
       THEN ...
END_IF;
...
```

В примере сначала Вы объявляете переменную с "исходным" типом данных и с любым предопределением ее значения. Затем Вы назначаете дополнительный вид типа данных (data type veiws) с ключевым словом AT:

```
View AT Variable : Data_type;   //комментарий
```

Вы можете применить несколько видов типа данных (data type veiws) для переменной, давая при этом им различные имена. При этом предопределение их фиксированными значениями (инициализация) не допускается.

Требуемый объем памяти для вида типа данных (data type veiw) не должен превышать объема памяти для самой переменной, для которой этот вида типа данных назначается (новый вид типа данных должен соответствовать переменной).

Вы можете использовать виды типа данных (data type veiws) для переменной как любые другие переменные, но только локально в блоке. В вышеприведенном примере в вызывающем блоке входной параметр *Station* инициализируется строкой символов: вид типа данных недоступен ему как байтовая структура.

Вид типа данных (data type veiw) может применяться посредством параметров блока и временных и статических локальных данных. Вид типа данных (data type veiw) должен быть объявлен в том же самом разделе объявления, где объявляются переменные.

В таблице 27.2 показано, какие виды типа данных (data type veiws) Вы можете применять с переменными разных типов данных. Если, к примеру, переменная размещается в области временных локальных данных функции FC, и если она относится к сложному типу данных, то виды типа данных (data type veiws), которые могут применяться с такой переменной могут относиться к одному из следующих типов данных: простой тип, сложный тип, тип данных POINTER или ANY.

Переменные типов TIMER, COUNTER или BLOCK_xx не допускают применения к себе видов типа данных (data type veiws).

В представленной ниже таблице 27.2 приняты следующие обозначения: типы данных для видов типов данных:

E - (elementary) - простой тип (BOOL, CHAR, BYTE, WORD, INT, DINT, REAL, S5TIME, TIME, DATE, TIME_OF_DAY);
C - (complex) - сложный тип (DATE_AND_TIME, STRING, ARRAY, STRUCT и UDT);
P - (POINTER) - указатель;
A - (ANY) - ANY-тип.

Таблица 27.2 Разрешенные виды типа данных (data type veiws)

| Блок | Объявление переменной в блоке | Типы данных переменных | | | |
|---|---|---|---|---|---|
| | | Простые (E) (elementary) | Сложные (C) (complex) | POINTER (P) | ANY (A) |
| FC | VAR_INPUT | E | C | | |
| | VAR_OUTPUT | E | C | | |
| | VAR_IN_OUT | E | C | | |
| | VAR [1)] | E C | E C A | | C |
| | VAR_TEMP | E C | E C A | | C |
| FB | VAR_INPUT | E C | E C P A | C | C |
| | VAR_OUTPUT | E C | E C | | |
| | VAR_IN_OUT | E | C | | |
| | VAR | E C | E C | | |
| | VAR_TEMP | E C | E C A | | C |

[1)] для временных локальных данных

## 27.2 Адресация

### 27.2.1 Абсолютная адресация

При абсолютной адресации адреса назначаются в соответствии с началом адресной области; например, I 1.0 (0-ой бит входного 1-го байта). Абсолютная адресация в SCL соответствует абсолютной адресации в стандартных языках программирования (см. раздел 3.3 "Адресация переменных") за исключением способа идентификации адресов глобальных данных, в котором есть отличие (см. табл. 27.3).

Таблица 27.3 Идентификация адресов при абсолютной адресации

| Адресная область | Бит | Байт | Слово | Двойное слово |
|---|---|---|---|---|
| Входы | I y.x | IBy | IWy | IDy |
| Выходы | Qy.x | QBy | QWy | QDy |
| Периферийные входы | - | PIBy | PIWy | PIDy |
| Периферийные выходы | - | PQBy | PQWy | PQDy |
| Меркеры | My.x | MBy | MWy | MDy |
| Адреса глобальных данных | DBz.DXy.x | DBz.DBy | DBz.DWy | DBz.DDy |
| | DBz.Dy.x | | | |

x = адрес бита, y = адрес байта, z = номер блока данных

В языке программирования SCL доступ к адресам глобальных данных возможен только способом полной адресации. Блок данных может быть переменной типа BLOCK_DB (см. также раздел 27.2.3 "Косвенная адресация в SCL").

Примечание:

В SCL не допускается присутствие разделителей (типа "пробел" или "табуляция") между адресом и идентификатором адреса.

Различия по сравнению со стандартными языками программирования: в SCL не применяется абсолютная адресация временных и статических локальных данных; вызов блока данных с частичным адресом невозможен; вычисление номера и размера глобального экземплярного блока данных невозможно.

## 27.2.2 Символьная адресация

При символьной адресации символьные имена должны быть назначены абсолютным адресам и переменным. Для глобальных данных имена назначаются в таблице символов; для локальных данных имена назначаются в разделе объявления переменных блока.

Символьная адресация в SCL соответствует символьной адресации в стандартных языках программирования (см. раздел 3.3 "Адресация переменных"). Может иметь место также использование смешанных абсолютно-символьных (mixed absolute/symbolic) идентификаторов, например таких:

```
DB10.Setpoint
"Motor1Data".DW12
```

Данные идентификаторы допустимы для доступа к глобальным данным с использованием полного адреса.

В SCL Вы можете назначать имена константам в разделе объявления блока и использовать в программе эти имена как символы.

## 27.2.3 Косвенная адресация в SCL

**Косвенное назначение глобальных адресов**

Косвенное использование глобальных адресов основывается на абсолютной адресации. При этом для указания расположения данных в памяти в квадратных скобках указывается переменная INT (две переменные INT в случае адресации бита):

- I[*byteindex.bitindex*];
- MB[*byteindex*],

где *byteindex* и *bitindex* являются константами или переменными, которые могут быть изменены в процессе обработки программы, или выражениями типа INT.

Таким способом Вы можете адресовать следующие области:

- периферийные входы PI и периферийные выходы PQ
  (в обоих этих случаях адрес бита не указывается);
- входы I, выходы Q, меркеры M;
- адреса глобальных данных D
  (блок данных и адрес данных);
- временные и статические локальные данные
  (только символьная адресация);
- Функции таймеров T и функции счетчиков C
  (для обоих этих функций адрес бита не указывается).

**Косвенное назначение адресов глобальных данных**

Косвенное использование адресов глобальных данных основывается на абсолютной адресации, но при этом адреса данных также как адреса блоков данных могут быть изменены в процессе обработки программы.

Вы можете использовать или абсолютный адрес или символьный адрес для блока данных:

- DB10.DX[*byteindex.bitindex*];
- MotorData.DW[*byteindex*],

где *byteindex* и *bitindex* являются константами или переменными, которые могут быть изменены в процессе обработки программы, или выражениями типа INT.

Применяя функцию преобразования WORD_TO_BLOCK, Вы можете назначить косвенный адрес блоку данных. Номер блока данных DB определяется либо как переменная, либо как выражение с типом данных WORD (см. пример на рис. 27.2).

- WORD_TO_BLOCK_DB[*dbindex*].DW0,

где *dbindex* является переменной, которая может быть изменена в процессе обработки программы, или выражением с типом данных WORD.

Если блок данных адресован косвенным способом, то для доступа к адресу данных не может использоваться символьное имя.

**Обращение к адресам данных с помощью параметра блока BLOCK_DB**

Если к блоку данных возможен доступ через параметр блока BLOCK_DB, то обращение к адресам данных в блоке может быть организовано и абсолютным, и косвенным способом (см. рис. 27.2). Пусть входной параметр *Data* имеет тип BLOCK_DB:

- Data.DW0;
- Data.DX2.0;
- Data.DW[*byteindex*];
- Data.DX[*byteindex.bitindex*],

где *byteindex* и *bitindex* являются константами или переменными, которые могут быть изменены в процессе обработки программы, или выражениями типа INT.

Если блок данных адресован через параметр блока BLOCK_DB, то для доступа к адресу данных не может использоваться символьное имя.

```
//***********************************************************************
//Пример косвенного использования глобальных адресов
k := 120; FOR i := 48 TO 62 BY 2 DO
MW[k] := PIW[i]; k := k + 2; END_FOR;

//***********************************************************************
//Косвенная адресация блоков данных
//Индекс DB имеет тип данных WORD
M0.0 := WORD_TO_BLOCK_DB(dbindex_w).DX0.0;
M0.0 := WORD_TO_BLOCK_DB(dbindex_w).DX[byteindex,bitindex];

//Индекс DB имеет тип данных INT
M0.0 := WORD_TO_BLOCK_DB(INT_TO_WORD(dbindex_i)).DX0.0;
M0.0 := WORD_TO_BLOCK_DB(INT_TO_WORD(dbindex_i)).DX[byteindex,bitindex];

//***********************************************************************
//Косвенная адресация посредством параметра блока
// Для имени "Data" и параметра с типом BLOCK_DB
M0.0 := Data.DX0.0;                  //абсолютная адресация
M0.0 := Data.DX[byteindex,bitindex]; //косвенная адресация
//***********************************************************************
```

Рис. 27.2 Пример косвенного использования глобальных адресов

**Адресация массивов**

В SCL в качестве индекса массива Вы можете использовать или константу, или переменную, или выражение типа INT, поэтому индекс может быть изменен в процессе выполнения программы. Вы можете также организовать доступ к части массива как к переменной (см. раздел 27.5.4 "Присвоение значений массивам").

При предопределении массивов отдельным размерностям массивов могут назначаться множители повторения.

## 27.3 Операторы

Выражения обеспечивают получение определенных значений. Выражение может включать в себя один адрес (идентификатор) данных (одну переменную) или несколько адресов (идентификаторов) данных (несколько переменных), которые объединяются с помощью операторов.

Пример:

```
a + b;
```

здесь *a* и *b* - это адреса (идентификаторы данных = переменные),

"+" - это оператор.

Порядок выполнения операторов в выражении определяется их взаимным расположением и приоритетом. Этот порядок выполнения операторов можно регулировать с помощью скобок. С точки зрения операторов выражения могут быть смешанными в соответствии с правилами, которые регламентируют в SCL комбинирование данных (как исходных, так и результатов вычислений), относящихся к разным типам данных.

Язык программирования SCL поддерживает операторы, список которых приводится в таблице 27.4.

Операторы, относящиеся к одному приоритетному классу, выполняются последовательно слева направо.

Таблица 27.4 Список операторов, поддерживаемые языком программирования SCL

| Комбинирование | Наименование | Оператор | Приоритет |
|---|---|---|---|
| Скобки | (*Выражение*) | ( , ) | 1 |
| Арифметические | Степень | ** | 2 |
| | Унарный плюс, унарный минус (знак) | +, - | 3 |
| | Умножение, деление | *, /, DIV, MOD | 4 |
| | Сложение, вычитание | +, - | 5 |
| Сравнения | Меньше, меньше или равно, больше, больше или равно, | <, <=, >, >= | 6 |
| | Равно, не равно | =, <> | 7 |
| Двоичные | Логическая операция отрицания (инвертирование) | NOT | 3 |
| | Логическая операция "И" | AND, & | 8 |
| | Логическая операция "Исключающее ИЛИ" | XOR | 9 |
| | Логическая операция "ИЛИ" | OR | 10 |
| Присвоение | Операция присваивания | := | 11 |

термин "унарный" означает, что оператор относится к одному адресу (переменной)

## 27.4 Выражения

Выражение - это формула, которая обеспечивает получение определенного значения. Выражение содержит адреса (идентификаторы) данных (переменные) и операторы. В простейшем случае выражение может включать в себя один адрес (идентификатор) данных, одну переменную или константу, а также может содержать знак или оператор

инвертирования.

Выражение может содержать адреса (идентификаторы) данных (переменные), объединенные в группы с помощью операторов. Сами выражения могут объединяться в группы с помощью операторов; при этом такие комбинированные выражения могут иметь очень сложную структуру. Порядок выполнения операторов в выражении обычно регулируется с помощью скобок.

Результат, полученный при выполнении выражения, может быть присвоен переменной или параметру блока, или этот результат может быть использован для проверки критерия условия в инструкции управления (control instruction).

Выражения могут быть разделены в соответствии со способом объединения (комбинирования) данных на арифметические выражения, логические выражения и выражения сравнения.

### 27.4.1 Арифметические выражения

Арифметическое выражение или состоит из численного значения или оно объединяет два значения или выражения с помощью арифметических операторов.

Пример:

```
Voltage * Current
```

В таблице 27.5 представлен перечень допустимых типов данных для арифметических выражений, а также для результатов арифметических выражений.

Спецификация класса типов данных ANY_NUM означает то, что тип данных первого и второго операнда может относиться к типу данных INT, DINT или REAL. Если Вы объединяете с помощью оператора операнд типа INT с операндом типа DINT, то результат должен быть типа DINT; если Вы объединяете с помощью оператора операнд типа INT или DINT с операндом типа REAL, то результат должен быть типа REAL. Перед тем как сформировать исполняемый блок, редактор выполняет (незаметно для пользователя) преобразование данных, то есть приводит данные из выражения к требуемому типу (см. также таблицу 30.4 "Функции неявного преобразования (Implicit Conversion Functions)").

В случае операции деления, второй операнд (делитель) не должен быть равным нулю.

На рис. 27.3 представлен пример арифметических выражений в сочетании с выражением присвоения значений.

### 27.4.2 Выражения сравнения

Выражение сравнения сравнивает значения двух операндов и выдает результат в виде булева значения; если условие сравнения выполняется,

то результат равен TRUE (ИСТИНА), иначе - результат равен FALSE (ЛОЖЬ).

Пример:

```
Voltage1 > Voltage2
```

Сравниваемые в выражении сравнения операнды должны относиться к одному типу данных или к одному классу типов данных (ANY_INT, ANY_NUM или ANY_BIT).

В таблице 27.5 представлен перечень допустимых типов данных для выражений сравнения, а также для результатов этих выражений.

Таблица 27.5 Допустимые типы данных операндов и результатов в SCL-выражениях

| Операция | Оператор | 1 операнд | 2 операнд | Результат |
|---|---|---|---|---|
| **Арифметические выражения** | | | | |
| Возведение в степень | ** | ANY_NUM | INT | REAL |
| Умножение | * | ANY_NUM<br>TIME | ANY_NUM<br>ANY_INT | ANY_NUM<br>TIME |
| Деление | / | ANY_NUM | ANY_NUM | ANY_NUM |
| Деление нацело | DIV | ANY_INT<br>TIME | ANY_INT<br>ANY_INT | ANY_INT<br>TIME |
| Деление по модулю | MOD | ANY_INT | ANY_INT | ANY_INT |
| Сложение | + | ANY_NUM<br>TIME<br>TOD<br>DT | ANY_NUM<br>TIME<br>TIME<br>TIME | ANY_NUM<br>TIME<br>TOD<br>TOD |
| Вычитание | - | ANY_NUM<br>TIME<br>TOD<br>DATE<br>TOD<br>DT | ANY_NUM<br>TIME<br>TIME<br>DATE<br>TOD<br>TIME | ANY_NUM<br>TIME<br>TOD<br>TIME<br>TIME<br>DT |
| **Выражения сравнения** | | | | |
| Меньше, меньше или равно,<br>больше, больше или равно, | <, <=, >, >= | ANY_NUM<br>CHAR или STRING<br>TIME<br>DATE<br>TOD | ANY_NUM<br>CHAR или STRING<br>TIME<br>DATE<br>TOD | BOOL<br>BOOL<br>BOOL<br>BOOL<br>BOOL |
| Равно, не равно | =, <> | ANY_BIT | ANY_BIT | BOOL |
| **Логические выражения** | | | | |
| Логическая операция отрицания | NOT | ANY_BIT | - | ANY_BIT |
| Логическая операция "И" | AND, & | ANY_BIT | ANY_BIT | ANY_BIT |
| Логическая операция "Исключающее ИЛИ" | XOR | ANY_BIT | ANY_BIT | ANY_BIT |
| Логическая операция "ИЛИ" | OR | ANY_BIT | ANY_BIT | ANY_BIT |

Для улучшения ясности в выражениях сравнения рекомендуется использовать скобки.

Выражения сравнения могут быть объединены с логическими операторами, например, следующим образом:

```
(Value1 > 40)  AND NOT  (Value2 = 20)
```

Сравнение переменных типов CHAR выполняется в соответствии с кодом символов ASCII. Пользователю доступны IEC-функции для выполнения операций сравнения переменных, относящихся к типу данных STRING и DT. IEC-функции представляют собой загружаемые FC блоки, которые находятся в библиотеке стандартов *Standard Library* в разделе *IEC Function Blocks.*

На рис. 27.3 представлены несколько примеров выражений сравнения в сочетании с выражениями присвоения значений.

```
(********************* Операции присвоения ************************

Automatic := TRUE;                    //Присвоение значения константы
Setpoint  := StartSetpoint;           //Присвоение переменной
Deviation := ActualValue - Setpoint; //Присвоение выражения
Display   := INT_TO_WORD(Deviation); //Присвоение значения функции

(******************* Арифметические выражения **********************

Power     := Voltage * Current;
Volume    := 4/3 * PI * Radius**3;
Solutionl := -P/2 + SQRT(SQR(P/2)-Q);
MeanValue := (Motor[1].Power + Motor[2].Power)/2;

(********************* Выражения сравнения ************************

TooLarge  := Voltage_Act > Voltage_Set;
Warning   := (Voltage * Current) >= 20_000;
M101.0    := Setpoint = ActualValue;
IF Deviation > 2_000 THEN Display := 16#F002; END_IF;

(********************Логические выражения ************************

Q4.0    := II.0 & II.1;
ON      := (Manual_on OR Auto_pn) AND NOT Fault;
MW30    := MW32 AND Mask;
Pulses  := (Edge_mem_bits XOR ID16) AND ID16; Edge_mem_bits := ID16;

(*****************************************************************
```

Рис. 27.3 Примеры операций присвоения, арифметических, логических выражений и выражений сравнения

### 27.4.3 Логические выражения

Логическое выражение объединяет два операнда или выражения, относящихся к классу типов данных ANY_BIT, в соответствии с логикой AND (И), OR (ИЛИ) или XOR (Исключающее ИЛИ).

Пример:

```
Automatic  AND NOT  Manual_on
```

Логическое выражение также включает в себя (булево) инвертирование; эта операция аналогична изменению знака значения.

Логическое выражение выдает значение, относящееся к классу типов данных ANY_BIT. Результат логического выражения относится к типу BOOL, если оба операнда также имеют тип BOOL. Если один или оба операнда имеют тип BYTE, WORD или DWORD, то результат будет иметь тип данных более "требовательного к памяти" операнда.

На рис. 27.3 представлены несколько примеров логических выражений в сочетании с выражениями присвоения значений.

## 27.5 Присвоение значений

С помощью операции присвоения значения одна переменная получает значение другой переменной или значение выражения. Слева от оператора присваивания ":=" находится переменная, которая принимает значение другой переменной или выражения, которые в свою очередь находятся справа от оператора присваивания.

Тип данных, находящихся с двух сторон от знака присваивания, должен быть идентичен. Исключение составляет только случай присваивания в функции "неявного изменения типа данных" ("Implicit data type conversion"): если тип данных переменной характеризуется по крайней мере таким же размером в битах или имеет больший размер в битах, чем тип данных выражения, то тип данных выражения "неявно" конвертируется (значение выражения автоматически конвертируется в требуемый тип данных и присваивается переменной). Другими словами, "неявное изменение типа данных" (с помощью функции преобразования типа данных) является необходимым.

### 27.5.1 Присвоение значений в случае простых типов данных

С помощью операции присвоения значение константы, переменной, адреса или выражение может быть присвоено переменной или адресу (см. рис. 27.3).

Абсолютные адреса (например, MW 10) имеют тип данных ANY_BIT; они имеют тип данных, определяемый размером занимаемой области (типы BOOL, BYTE, WORD, DWORD). Если Вы хотите назначить значение отличающегося типа данных абсолютному адресу, то используйте преобразование типа данных, или назначьте этому адресу имя и требуемый тип данных в таблице символов.

### 27.5.2 Присвоение значений переменным типов DT и STRING

Каждой DT-переменной может быть назначено значение другой DT-

переменной или DT-константы.

Каждой STRING-переменной может быть назначено значение другой STRING-переменной или строки символов. Если назначаемая строка символов длиннее, чем переменная, стоящая слева от оператора присваивания, то на этапе компиляции пользователь получит предупреждающее сообщение.

Нельзя выполнять предопределение в разделе объявления в области временных локальных данных. Если Вы используете функции обработки STRING-операндов, например, IEC-функции, с помощью которых STRING-переменная проверяется (так, как выходной параметр), то Вы должны запрограммировать предопределенное выходное значение.

### 27.5.3 Присвоение значений структурам

С помощью операции присвоения одной STRUCT-переменной может быть назначено значение другой STRUCT-переменной только в том случае, если:

- структуры этих данных согласованы;
- компоненты структур согласованы с точки зрения типов данных;
- компоненты структур согласованы с точки зрения имен.

Отдельные компоненты структур могут быть обработаны как переменные одного типа данных; например, значение компонента структуры *Motor1.Setpoint* типа INT может быть присвоено другой INT-переменной, или какое-либо целое (INT) значение может быть присвоено данному компоненту структуры.

### 27.5.4 Присвоение значений массивам

С помощью операции присвоения одной ARRAY-переменной может быть назначено значение другой ARRAY-переменной только в том случае, если согласованы типы данных элементов этих массивов, а также граничные значения индексов (наименьшее и наибольшее значения) в каждой размерности, а также число размерностей для этих массивов совпадают.

Отдельные компоненты массивов могут быть обработаны так же как переменные соответствующего типа данных.

В случае использования массивов с несколькими размерностями, Вы можете работать с частями массивов как с массивами соответствующей размерности (ARRAY-переменными соответствующей размерности): если отбрасывать те или иные индексы массива, то можно получать массивы с меньшей размерностью по сравнению с исходным массивом данных.

Пример: пусть исходный массив задан следующим образом:

```
Field1 : ARRAY [1..8,1..16] OF INT;
```

Таким образом, массив `Field1` представляет собой двумерный массив.

Исходя из условий данного примера, Вы можете:

- обращаться к массиву в целом, используя идентификатор `Field1`;
- обращаться к части массива, используя идентификатор `Field1[i]` (что соответствует строке матрицы);
- обращаться к элементу массива, используя идентификатор `Field1[i,j]`.

Вы можете также присваивать часть массива `Field1[i]` другому массиву (то есть, другой ARRAY-переменной), имеющему соответствующую размерность, например:

```
Field12 := Field1[i],
```

где:

i может принимать значения от 1 до 8;

массив `Field2` объявлен следующим образом:

```
Field2  :  ARRAY [1..16] OF INT;
```

# 28 Операторы управления (Control Statements)

С помощью операторов управления (control statements) пользователь может организовать ветвление программы, циклическое выполнение отдельных фрагментов программы и осуществлять переход в программе блока для выполнения другой ее части. Язык программирования SCL поддерживает следующие операторы управления:

- IF (оператор для выполнения ветвления в программе по условию, проверяемому в отношении булевой переменной (или параметра типа BOOL));
- CASE (оператор для выполнения ветвления в программе по условию, проверяемому в отношении целой переменной (или параметра типа INT));
- FOR (оператор для организации в программе циклов с переменной - счетчиком циклов);
- WHILE (оператор для организации в программе циклов, инициируемых при выполнении определенного условия);
- REPEAT (оператор для организации в программе циклов с завершением по условию);
- CONTINUE (оператор для завершения текущего прохода цикла в программе);
- EXIT (оператор для выхода из цикла в программе);
- GOTO (оператор для продолжения выполнения программы, начиная с метки перехода);
- RETURN (оператор для выхода из программы блока).

## 28.1 Оператор IF

Оператор IF управляет выполнением той или иной части программы в зависимости от состояния булевой переменной. С помощью оператора IF пользователь может запрограммировать выполнение различных, определяемых условиями, ветвей программы.

```
IF  condition
  THEN  statements;
END_IF;
```

Здесь `condition` - это адрес или выражение с типом BOOL. Если `condition` имеет значение TRUE (ИСТИНА), то выполняются операторы после ключевого слова THEN. Если `condition` имеет значение FALSE (ЛОЖЬ), то выполняются операторы после ключевого слова END_IF. Ключевое слово END_IF завершает оператор IF.

```
IF  condition
   THEN  statements1;
   ELSE  statements0;
END_IF;
```

В данном примере, как и в предыдущем, `condition` имеет значение TRUE (ИСТИНА) или FALSE (ЛОЖЬ). Если `condition` имеет значение TRUE (ИСТИНА), то выполняются операторы после ключевого слова THEN. Если `condition` имеет значение FALSE (ЛОЖЬ), то выполняются операторы после ключевого слова ELSE.

```
IF  condition1
   THEN statements1;
   ELSEIF  condition2
     THEN  statements2;
   ELSE statements0;
END_IF;
```

Оператор `IF` может быть вложенным. В данном примере, как и в предыдущем, если `condition1` имеет значение TRUE (ИСТИНА), то выполняются операторы `statements1` после первого ключевого слова THEN, и далее программа будет выполняться, начиная с операторов после ключевого слова END_IF. Если `condition1` имеет значение FALSE (ЛОЖЬ), то выполняется проверка условия `condition2` после ключевого слова ELSEIF. Если `condition2` имеет значение TRUE (ИСТИНА), то выполняются операторы `statements2` после второго ключевого слова THEN, и далее программа будет выполняться, начиная с операторов после ключевого слова END_IF. Если `condition2` имеет значение FALSE (ЛОЖЬ), то выполняются операторы `statements0` после ключевого слова ELSE.

Пользователь может использовать любое количество комбинаций ключевых слов ELSEIF ... THEN ... между ключевыми словами IF ... THEN ... и ELSE. Ключевое слово ELSE и последующие операторы не являются обязательными.

Пример:

Если переменная *Actual_value* больше, чем переменная *Setpoint*, то выполняются операторы, идущие после ключевого слова THEN. Если, наоборот, переменная *Actual_value* меньше, чем переменная *Setpoint*, то выполняются операторы, идущие после ключевого слова ELSEIF. Если оба выражения сравнения не выполняются, то выполняются операторы после ключевого слова ELSE.

```
IF  Actual_value > Setpoint
   THEN greater_than := TRUE;
        less_than    := FALSE;
        equal_to     := FALSE;
```

```
   ELSEIF  Actual_value < Setpoint
       THEN  greater_than := FALSE;
             less_than    := TRUE;
             equal_to     := FALSE;
       ELSE  greater_than := FALSE;
             less_than    := FALSE;
             equal_to     := TRUE;
END_IF;
```

## 28.2 Оператор CASE

Оператор CASE позволяет выбрать для выполнения нужную последовательность операторов в программе в зависимости от значения целой переменной (параметра типа INT).

Общая структура программы с оператором CASE может иметь следующую форму:

```
CASE Selection OF
  CONST1 :    statements1;
  CONST2 :    statements2;
  ...
  CONSTx :    statementsx;
  ELSE   :    statements0;
END_CASE;
```

Здесь `Selection` - это адрес или выражение с типом INT. Если `Selection` имеет значение `CONST1`, то выполняются операторы `statements1`. Если `Selection` имеет значение `CONST2`, то выполняется цепочка операторов `statements2` и так далее.

Если `Selection` имеет значение, находящееся за рамками списка значений, указанных для проверки, то выполняется цепочка операторов после ключевого слова `ELSE`. При этом цепочка операторов, начинающаяся с ключевого слова `ELSE`, не является обязательной.

Список значений CONST1, CONST2 и т.д. состоит из целых (INT) констант. Для этих констант могут использоваться несколько вариантов форматов записи при использовании оператора CASE, то есть в качестве константы-"переключателя" CONSTx могут быть указаны:

- одиночное целое (INT) число;
- диапазон целых (INT) чисел (например: 15..20);
- смесь разделенных запятыми отдельных целых (INT) чисел и диапазонов целых чисел (например: 21,25,31..33).

При этом каждое значение константы-"переключателя" CONSTx должно быть уникально, оно может появится в списке одного оператора CASE только один раз.

Оператор CASE может быть вложенным.

Вместо отдельной цепочки операторов в списке оператора CASE может стоять другой оператор CASE.

Пример:

Значение, присваиваемое переменной *Error_number*, зависит от переменной *ID*.

```
CASE ID OF
   0      :    Error_number := 0;
   1,3,5  :    Error_number := ID + 128;
   ...
   6..10  :    Error_number := ID;
   ELSE   :    Error_number := 16#7F;
END_CASE;
```

## 28.3 Оператор FOR

Оператор FOR для организации в программе циклов с переменной - счетчиком циклов. Выполнение внесенного в цикл фрагмента программы будет повторяться столь долго, пока переменная "счетчик циклов" будет оставаться в указанном диапазоне значений.

Общая структура программы с оператором FOR может иметь следующую форму:

```
FOR i := limit1
      TO limit2
      BY step
      DO statements;
END_FOR;
```

В начале обработки данного оператора начальное значение `limit1` присваивается счетчику циклов `i`. Вы можете сами определять переменную-счетчик циклов; это должна быть переменная с типом INT или с типом DINT. Начальное значение `limit1` может быть выражением с типом INT или DINT, также как и конечное значение `limit2` и шаг изменения переменной-счетчика цикла `step`.

В начале выполнения цикла переменная-счетчик цикла получает указанное после оператора присваивания начальное значение. В то же самое время рассчитываются и запоминаются конечное значение `limit2` и шаг изменения переменной-счетчика цикла `step` (изменение этих значений во время выполнения цикла не будет иметь никаких последствий). После этого проверяется условие завершения выполнения цикла, и если условие не выполняется, то выполняется программа внутри цикла.

После каждого прохода программы цикла счетчик цикла увеличивается на величину шага приращения `step` (если шаг указан как положительное число) или уменьшается на величину шага приращения `step` (если шаг указан как отрицательное число).

При программировании цикла строка `BY step` (то есть, начиная с ключевого слова BY) не является обязательной. Если такое условие для шага изменения переменной-счетчика цикла отсутствует, то шаг (по умолчанию) принимается равным +1. Если величина переменной-счетчика цикла выходит за пределы диапазона значений между начальным значением и конечным значением `(limit1...limit2)`, то программа продолжает выполняться сразу после завершающего ключевого слова END_FOR.

Последний проход цикла выполняется при значении переменной-счетчика цикла, равном конечному значению, или (если шаг приращения таков, что конечное значение `limit2` не может быть достигнуто точно) при значении переменной-счетчика, равном разности конечного значения и шага приращения `(limit2 - step)`. После полного окончания обработки цикла значение переменной-счетчика цикла равно ее значению при последнем проходе цикла, суммированному с величиной шага приращения.

Оператор FOR может быть вложенным: внутри цикла с использованием оператора FOR могут быть запрограммированы другие циклы с использованием оператора FOR с другими переменными-счетчиками цикла.

Внутри цикла с использованием оператора FOR может быть запрограммирован переход к началу цикла (с использованием оператора управления CONTINUE) или полный выход из цикла для продолжения выполнения программы, начиная сразу же после ключевого слова END_FOR, (с использованием оператора управления EXIT).

Пример:

Пусть, необходимо считать слова с PIW 128 по PIW 142 из области периферии в область меркеров в слова с MW 128 по MW 142.

```
FOR i := 128 TO 142 BY 2 DO
    MW[i] := PIW[i];
END_FOR;
```

## 28.4 Оператор WHILE

Оператор WHILE служит для организации в программе циклов, выполнение которых продолжается все время, пока выполняется определенное условие.

Общая структура программы с оператором WHILE может иметь следующую форму:

```
WHILE  Condition  DO
    Statements;
END_WHILE;
```

Здесь `Condition` - это адрес или выражение с типом BOOL. Пока выполняется определенное условие (то есть, пока `Condition = TRUE(ИСТИНА)`), будут циклически выполняться выражения `Statements`.

Перед каждым проходом выполняется проверка условия `Condition`. Если условие не выполняется (`Condition = FALSE(ЛОЖЬ)`), то программа продолжает выполняться, начиная сразу же после ключевого слова END_WHILE. Такой вариант событий возможен даже, если не было ни одного прохода программы цикла (то есть до первого прохода цикла), что означает, что операторы `Statements` при этом ни разу не будут выполнены.

Оператор WHILE может быть вложенным; при этом внутри одного цикла с оператором WHILE могут размещаться другие циклы с оператором WHILE.

Внутри цикла с использованием оператора WHILE может быть запрограммирован переход к началу цикла (с использованием оператора управления CONTINUE) или полный выход из цикла для продолжения выполнения программы, начиная сразу же после ключевого слова END_WHILE, (с использованием оператора управления EXIT).

Пример:

Пусть, необходимо найти в блоке данных DB10 битовое значение 16#FFFF. Слово данных DW0 содержит или 16#FFFF, или интервал перед следующим словом, которое снова содержит или 16#FFFF, или интервал перед следующим словом.

```
i := 0;
WHILE DB10.DB[i] := 16#FFFF DO
    i := i + WORD_TO_INT(DB10.DB[i]);
END_WHILE;
```

## 28.5 Оператор REPEAT

Оператор REPEAT служит для организации в программе циклов, выполнение которых продолжается все время, пока не встретится условие завершения обработки цикла.

Общая структура программы с оператором REPEAT может иметь следующую форму:

```
REPEAT
    Statements;
    UNTIL Condition
END_REPEAT;
```

Здесь `Condition` - это адрес или выражение с типом BOOL. Пока не выполняется определенное условие (то есть, пока `Condition = FALSE(ЛОЖЬ)`), будут циклически выполняться выражения `Statements`.

После каждого прохода цикла выполняется проверка условия `Condition`. Если условие выполняется (`Condition = TRUE(ИСТИНА)`), то цикл далее не обрабатывается и выполнение программы будет продолжено сразу же после ключевого слова END_REPEAT.

Таким образом, программа цикла будет обработана по крайней мере один раз, даже если при первом проходе цикла выполняется условие завершения его обработки, что означает, что операторы `Statements` будут выполнены по крайней мере один раз.

Оператор REPEAT может быть вложенным; при этом внутри одного цикла с оператором REPEAT могут размещаться другие циклы с оператором REPEAT.

Внутри цикла с использованием оператора REPEAT может быть запрограммирован переход к началу цикла (с использованием оператора управления CONTINUE) или полный выход из цикла для продолжения выполнения программы, начиная сразу же после ключевого слова END_REPEAT, (с использованием оператора управления EXIT).

Пример:

Пусть, необходимо вызывать системную функцию SFC 25 COMPRESS в программе перезапуска, пока не будет завершено "сжатие" памяти пользователя.

```
REPEAT
    SFC_ERROR := COMPRESS(
         BUSY := busy,
         DONE := done);
    UNTIL done
END_REPEAT;
```

## 28.6 Оператор CONTINUE

Оператор CONTINUE служит для завершения текущего прохода цикла в программе, организованного с помощью операторов FOR, WHILE или REPEAT.

После выполнения оператора CONTINUE проверяются условия для выполнения следующего прохода программы цикла (в случае циклов, организованных с помощью операторов REPEAT или WHILE) или (в случае циклов, организованных с помощью оператора FOR) переменная-счетчик цикла изменяется на величину шага приращения, и далее следует проверка, находится ли эта переменная в допустимых для нее пределах?

Если проверяемое условие соблюдается, то после оператора CONTINUE выполняется следующий проход программы цикла с самого начала.

Пример:

Пусть, необходимо установить меркеры с помощью двух вложенных циклов с использованием оператора FOR; если байтовый адрес (i) равен 0, а битовый адрес (k) меньше 2, то операторы тела внутреннего цикла FOR не выполняются (установка битов должна начинаться с меркера M0.3).

```
FOR i := 0 TO 2 DO
  FOR k := 0 TO 7 DO
     IF (k<2 & i=0)
     THEN CONTINUE;
     END_IF;
     M[i,k] := TRUE;
  END_FOR;
END_FOR;
```

## 28.7 Оператор EXIT

Оператор EXIT служит для полного завершения обработки цикла (с выходом из него), организованного с помощью операторов FOR, WHILE или REPEAT. При этом выход из цикла с оператором EXIT не зависит от выполнения условий, проверяемых в цикле, и может производиться из любой точки цикла. При выходе из цикла с оператором EXIT программа продолжает выполняться сразу же после ключевых слов END_FOR, END_WHILE или END_REPEAT.

Выход из цикла с оператором EXIT происходит немедленно из точки внутри программы цикла, где этот оператор встретился.

Пример:

Пусть, необходимо установить меркеры с помощью двух вложенных циклов с использованием оператора FOR; если байтовый адрес (i) равен 2, а битовый адрес (k) больше 5, то выполнение внутреннего цикла FOR прерывается (установка битов должна заканчиваться на меркере M2.5).

```
FOR i := 0 TO 2 DO
  FOR k := 0 TO 7 DO
     IF (k=2 & i>5)
     THEN EXIT;
     END_IF;
     M[i,k] := TRUE;
  END_FOR;
END_FOR;
```

В данном примере выполнение цикла FOR прекращается при определенном условии для счетчика цикла *k* с помощью оператора EXIT. Выполнение внешнего цикла FOR с счетчиком *i* не связан с выходом из цикла. Тем не менее, пример составлен таким образом, что выход из цикла FOR производится на последнем проходе цикла с счетчиком *i*.

## 28.8 Оператор RETURN

Оператор RETURN служит для безусловного выхода из текущего блока.

Выполнение программы при этом будет продолжено либо в вызывающем блоке, либо в операционной системе (если выход с оператором RETURN происходит в организационном блоке).

Оператор RETURN не является обязательным оператором в конце блока.

Оператор RETURN пересылает состояние сигнала переменной OK на выход ENO завершаемого (с RETURN) блока.

Пример:

Пусть, необходимо организовать выход из блока по условию.

```
IF Error <> 0 THEN RETURN;
END_IF;
```

## 28.9 Оператор GOTO

Оператор GOTO служит для продолжения выполнения программы начиная с другой точки.

Пример:

Пусть, необходимо организовать выход из блока по условию.

```
        GOTO M1;
        ...;        //Пропущенные операторы
        ...;        //Пропущенные операторы
M1:     ...;        //Точка - цель перехода
```

Связь между оператором GOTO и точкой перехода обеспечивается меткой перехода. Вы должны объявить метки перехода в разделе объявлений блока между ключевыми словами: LABEL и END_LABEL. Имя меток перехода имеет такую же структуру, как и имя локальной переменной блока.

Метка перехода должна быть уникальной (не должна повторяться). Вы можете организовать в программе несколько переходов с помощью нескольких операторов перехода GOTO к одной и той же метке перехода.

После выполнения перехода к некоторой метке программа продолжает выполняться со строки, отмеченной данной меткой. Метка перехода и выражение в этой же строке должны быть разделены двоеточием.

За меткой перехода всегда следует оператор (выражение). Допустим также "пустой" оператор:

```
Label1: ;
```

Итак, если Вы используете оператор GOTO, то помните следующие правила:

- ель перехода должна быть внутри того же блока программы, что и оператор GOTO;
- цель перехода должна быть однозначно определена;
- Вы не можете перейти "извне" внутрь цикла, но из цикла переход возможен.

Пример:

```
LABEL
      M1, M2, M3, End;
END_LABEL
...
CASE Selection OF
1 : GOTO M1;
2 : GOTO M2;
3 : GOTO M3;
ELSE GOTO End;
END_CASE;
M1: ... statement1 ...;
GOTO End;
M2: ... statement2 ...;
GOTO End;
M3: ... statement3 ...;
END:;
```

Примечание:

Оператор GOTO не определен как стандартный. Язык SCL поддерживает все операторы и функции, необходимые для структурного программирования, поэтому без оператора GOTO можно обойтись.

# 29 SCL-блоки

## 29.1 SCL-блоки: общая информация

В языке программирования SCL используется точно такая же структура блока, как и в стандартных языках программирования. Вы можете запрограммировать отдельные блоки с использованием языка программирования SCL, а затем вызывать их, скажем, в FBD-блоке, и Вы также можете из SCL-блоков вызывать блоки, созданные с использованием языка STL.

Для того, чтобы обрабатывать в пользовательской программе блоки, созданные с использованием различных языков программирования, интерфейс блока должен иметь стандартную структуру. Это особенно касается инициализации входа EN и выхода ENO (см. раздел 29.4 "Механизм EN/ENO").

Примеры программ, рассматриваемые в данной главе, Вы можете найти на прилагаемой дискете в библиотеке SCL_Book library в программе "29 Block Calls" ("Вызов блоков").

### Структура программы пользователя

Организационный блок представляет собой интерфейс между операционной системой и пользовательской программой. Организационные блоки вызываются операционной системой CPU при свершении определенных событий, таких, например, как прерывания. Для управления программируемым контроллером пользовательская "обычная" программа выполняется в циклическом режиме. Эта программа располагается в организационном блоке OB 1, который используется практически всеми пользовательскими программами. Начало пользовательской программы соответствует первому сегменту в блоке OB 1 (см. раздел 3.1 "Обработка программы").

В соответствии со своим пониманием функциональной структуры пользовательской программы Вы можете разбить ее в OB1 на отдельные подпрограммы (или "блоки" - "blocks"). Пользовательская программа размещается в кодовых блоках (Code Block) и в блоках данных (Data Block), где хранятся данные пользователя. Кодовые блоки - это и есть собственно подпрограммы, которые Вы должны вызывать для выполнения (см. раздел 20.1 "Организация программы").

### Блоки

Система STEP 7 поддерживает функции FC и функциональные блоки FB как кодовые блоки. Функциональные блоки FB вызываются вместе с блоками данных, в которых хранятся локальные переменные блока (как

бы "память" блока). Такой блок данных, назначаемый вызову FB, называется "экземплярным блоком данных" ("*instance data block*"); он может быть или собственно блоком данных, или может быть частью блока данных более высокого уровня ("higher-level" data block). Функции FC не имеют блоков данных, но они могут иметь "функциональное значение" ("*function value*"). Это функциональное значение позволяет, например, в одном арифметическом выражении использовать в качестве операнда функцию FC (точнее, ее функциональное значение) наряду с другими переменными (см. раздел 3.2 "Блоки").

Оба упомянутых блока могут иметь параметры блока. Параметры блока позволяют параметризировать "правила" его обработки (функцию блока).

При программировании блока Вы можете объявить параметр блока как "входной параметр" ("*input parameter*") (VAR_INPUT), если необходимо только считывать (сканировать) значение параметра, или как "выходной параметр" ("*output parameter*") (VAR_OUTPUT), если необходимо только записывать значение параметра, или как "входной/выходной параметр" ("*in-out parameter*") (VAR_IN_OUT), если в программе необходимо как считывать, так и записывать значение параметра.

Обращение к параметру блока в программном блоке выполняется с "использованием формального параметра" ("*formal parameter*") по его имени (имени параметра блока). Формальный параметр служит своего рода макетом для "фактического параметра" ("*actual parametr*"), используемого CPU во время выполнения программы. При вызове блока фактические параметры назначаются параметрам блока; они представляют собой значения, которые должны поступить в вызываемый блок, и с которыми этот блок обрабатывается.

## 29.2 Программирование SCL-блоков

Средства программирования SCL-блоков описаны в главе 2 "Программное обеспечение STEP 7"; соответствующий список ключевых слов Вы можете найти в разделе 3.5 "Программирование кодовых блоков на SCL". Блоки данных и данные пользовательского типа UDT в основном программируются точно так же как и в языке программирования STL (см. раздел 3.6 "Программирование блоков данных" и раздел 24.3 "Пользовательский тип данных").

Для того, чтобы обозначить разницу в программировании различных кодовых блоков, а точнее - разницу при использовании различных языков программирования, рассмотрим реализацию функции ограничителя "Delimiter" из главы 27 "Введение. Элементы языка" в соответствии со следующим планом реализации программы:

- в виде функции FC 291 без возвращаемого значения функции;
- в виде функции FC 292 с возвращаемым значением функции;
- в виде функционального блока FB 291 со своим собственным блоком данных DB 291;
- в виде функционального блока FB 291 как локального экземпляра в функциональном блоке FB 290.

После этого мы организуем вызов всех блоков в функциональном блоке (FB 290 с DB 290 в качестве экземплярного блока данных). Во всех случаях программа всегда будет одна и та же. Изменяются только объявление и инициализация параметров.

Примечание:

Так как программа функции ограничителя "Delimiter" не сохраняет локальные данные и возвращает свое значение, то вариант программной реализации в виде функции FC с функциональным значением является оптимальным типом блока.

### 29.2.1 Функции FC без возвращаемого значения функции

Функция FC без возвращаемого значения функции имеет тип данных VOID. В нашем примере функция FC 291 имеет входные параметры MAX, IN, MIN и выходной параметр OUT.

```
FUNCTION FC291 : VOID
VAR_INPUT
   MAX : INT;
   IN  : INT;
   MIN : INT;
END_VAR
VAR_OUTPUT
   OUT : INT;
END_VAR;
BEGIN
IF IN > MAX THEN OUT := MAX;
   ELSIF IN < MIN THEN OUT := MIN;
   ELSE OUT := IN;
END_IF;
END_FUNCTION
```

Все выходные параметры простых типов в функции должны быть соответствующим образом определены заданным алгоритмом и должны вычисляться в процессе обработки программы. Входные параметры функции могут быть только считаны, а выходные параметры могут быть только записаны.

### 29.2.2 Функции FC с возвращаемым значением функции

Функция FC с возвращаемым значением функции имеет тип данных, соответствующий функциональному значению (возвращаемому значению функции). В нашем примере функция FC 292 имеет входные параметры MAX, IN, MIN и функциональное значение (возвращаемое значение

функции), которое имеет адрес (имя) функции или в абсолютной или в символьной форме. Тип данных функционального значения должен быть определен после имени блока, отделенный от него двоеточием.

```
FUNCTION FC292 : INT
VAR_INPUT
   MAX : INT;
   IN  : INT;
   MIN : INT;
END_VAR
BEGIN
IF IN > MAX THEN FC292 := MAX;
   ELSIF IN < MIN THEN FC292 := MIN;
   ELSE FC292 := IN;
END_IF;
END_FUNCTION
```

В качестве типа данных для функционального значения Вы можете использовать любые простые типы, а также следующие типы данных: DATE_AND_TIME, STRING и пользовательские типы UDT. Типы данных ARRAY, STRUCT, POINTER и ANY в данном случае не допускаются.

Если функциональное значение относится к типу STRING, то для него будет зарезервирована длина, определенная в установках компилятора (но не максимальная длина, заданная в квадратных скобках в разделе объявления параметров).

Все выходные параметры простых типов в функции должны быть соответствующим образом определены заданным алгоритмом и должны вычисляться в процессе обработки программы. Входные параметры функции могут быть только считаны, а выходные параметры могут быть только записаны.

В соответствии с программой функции FC функциональное значение должно быть определено в выражении, относящемся к тому же типу данных. Определение этого значения и присвоение его возвращаемому значению функции должно быть выполнено также в процессе обработки программы.

### 29.2.3 Функциональный блок FB

Функциональному блоку соответствует экземплярный блок, в котором хранятся переменные функционального блока (функциональный блок или вызывается со своим собственным блоком данных, или он использует блок данных вызывающего функционального блока).

В нашем примере мы хотим использовать это и объявим предельные значения для функции ограничителя статическими локальными переменными. Только входная переменная IN и выходная переменная OUT остаются параметрами блока.

```
FUNCTION_BLOCK FB291
VAR_INPUT
   IN  : INT;
END_VAR
VAR_OUTPUT
   OUT : INT;
END_VAR;
VAR
   MAX : INT := 10_000;
   MIN : INT := -5_000;
END_VAR
BEGIN
IF IN > MAX THEN OUT := MAX;
   ELSIF IN < MIN THEN OUT := MIN;
   ELSE OUT := IN;
END_IF;
END_FUNCTION_BLOCK
```

Входные параметры блока могут быть только считаны, а выходные параметры могут быть только записаны.

Различают два варианта вызовов: вызов с собственным блоком данных или вызов в режиме локального экземпляра. Вариант последовательного вызова блока не должен рассматриваться при программировании функционального блока. Тем не менее, необходимо обеспечить, чтобы при использовании локального экземпляра по крайней мере один параметр блока или одна статическая локальная переменная были доступны: длина экземпляра не должна быть равна нулю.

Примечание:

Входные и выходные параметры сложных типов сохраняются в экземплярном блоке данных в виде значений, а входные-выходные параметры сохраняются в виде указателей на фактические параметры (см. раздел 26.3.2 "Хранение параметров в функциональных блоках").

## 29.2.4 Временные локальные данные

Все кодовые блоки имеют область для временных локальных данных, которую Вы можете использовать для промежуточного хранения данных в блоке. При программировании на языке SCL Вы можете использовать временные локальные данные точно также, как при программировании на стандартных языках. Для получения более подробной информации Вы можете обратиться к разделу 18.1.5 "Временные локальные данные".

Вы должны объявить временные локальные данные в разделе объявлений блока после ключевого слова VAR_TEMP. Здесь допускается применять все простые, сложные, пользовательские (UDT) типы данных, а также типы данных POINTER и ANY. Для типа данных ANY существуют специальные правила использования (см. далее).

Временные локальные данные не могут быть предопределены значениями на этапе объявления. Вот почему, при назначении L-стека редактор резервирует для переменных типа STRING область с размером, который вводится на вкладке "Compiller" ("Компилятор"), выбираемой с использованием опций меню: *Option -> Customize (Опции -> Установки пользователя)*.

Если временные локальные данные должны получить конкретные значения, они должны быть сначала записаны. Это также касается временных переменных типа STRING, полученных в качестве выходного параметра, например, при использовании IEC-функций. При записи значения IEC-функция проверяет корректность введенной информации о размере строковой переменной. Вы можете обратиться к ней, назначая значение (любое значение) переменной в программе перед ее использованием.

При программировании на языке SCL Вы можете объявлять переменные, относящиеся к одному типу данных, списком:

```
VAR_TEMP
   VALUE1, VALUE2, VALUE3 : INT;
   ...
END_VAR
```

Необходимо отметить, что при программировании на языке SCL временные локальные данные могут быть адресованы только символьным способом.

**Тип данных ANY**

Временные локальные данные типа ANY могут хранить адреса инструкций, а также глобальных переменных или локальных переменных блока:

```
ANY_VAR := MW10;
ANY_VAR := Setpoint;
ANY_VAR := DB10.Field;
```

Временные локальные данные типа ANY могут быть предопределены посредством ключевого слова NIL, указывающим на "нуль":

```
ANY_VAR := NIL;
```

Пример:

Пусть, различные записи данных должны копироваться в "почтовый ящик" для пересылки ("send mailbox") с помощью системной функции SFC 20 BLKMOV в зависимости от идентификатора:

```
VAR_TEMP
   Address := ANY;
END_VAR
...
CASE Identifier OF
1: Address := DataRecord1;
2: Address := DataRecord2;
...
```

```
ELSE Address := NIL;
END_CASE;
SFC_ERROR := BLKMOV(
   SRCBLK := Address,
   DSTBLK := SendMailbox);
```

Вы можете редактировать отдельные компоненты ANY-указателя, такие как номер DB или адрес, непосредственно с помощью видов типа данных (см. раздел 27.1.9 "Виды типа данных (Data Type Veiws)").

### 29.2.5 Статические локальные данные

Статические локальные данные - это "память" функционального блока. Эти данные располагаются в экземплярном блоке данных. При этом значения данных сохраняются до тех пор, пока не будут изменены из программы точно также, как изменяются значения переменных в глобальных блоках данных.

В статических локальных данных Вы также можете объявлять локальные экземпляры функциональных блоков и системных функциональных блоков. Для получения более подробной информации Вы можете обратиться к разделу 18.1.6 "Статические локальные данные".

Статические локальные данные объявляются с помощью ключевых слов VAR и END_VAR. В статических локальных данных допускается применять все простые, сложные, пользовательские (UDT) типы данных, а также типы данных POINTER и ANY.

При программировании на языке SCL Вы можете списком объявлять переменные, относящиеся к одному типу данных. Переменные, объявленные таким способом, не могут получить предопределение (инициализирующее значение).

Пример:

```
VAR
   VALUE1, VALUE2, VALUE3 : INT;
   VALUE4                 : INT:=3;
   ...
END_VAR
```

Необходимо отметить, что при программировании на языке SCL статические локальные данные в функциональном блоке могут быть адресованы только символьным способом.

Так как статические локальные данные располагаются в блоке данных, то доступ к ним может быть организован таким же образом, как к глобальным данным. Доступ к этим данным обеспечивается с использованием полного адреса, с определением блока данных и адреса данных.

## 29.2.6 Параметры блока

Параметры блока обеспечивают связь между вызывающим и вызываемым блоками. Эти параметры могут быть объявлены как входные (Input), входные-выходные (In/Out) и выходные (Output) параметры (см. раздел 19.1.3 "Объявление параметров блока").

Входные (Input) параметры могут только быть считаны, выходные (Output) параметры могут только быть записаны. Поэтому, если Вам требуются такие параметры, которые могут быть считаны, потом изменены и, наконец, снова записаны, то Вам необходимо использовать входные-выходные (In/Out) параметры.

В случае использования функций FC, параметры блока являются указателями на фактические параметры или на другие указатели. В случае использования функциональных блоков FB, параметры блока сохраняются в экземплярных блоках данных (см. раздел 26.3 "Сохранение данных при передаче параметров").

При программировании на языке SCL Вы можете списком объявлять переменные, относящиеся к одному типу данных. Переменные, объявленные таким способом, не могут получить предопределение (инициализирующее значение).

Пример:

```
VAR_INPUT
   VALUE1, VALUE2, VALUE3 : INT;
   ...
END_VAR
```

Так как параметры блока размещаются в блоке данных, то доступ к ним может быть организован таким же образом, как к глобальным данным. Доступ к этим данным обеспечивается с использованием полного адреса, с определением блока данных и адреса данных.

```
Result := DB279.DW20;
Result := DB279.Total;
Result := Totalizer.Total;
Result := Totalizer.DW20;
```

Фактически дальнейшая обработка возможна только для значений выходных (Output) параметров (см. вопросы, касающиеся вызовов блоков в разделе 29.3.3 "Функциональный блок со своим собственным блоком данных", а также раздел 29.3.4 "Функциональный блок как локальный экземпляр").

**Предварительная инициализация параметров блоков**

Предварительная инициализация параметров блоков не обязательна и допускается только в отношении таких функциональных блоков, параметры которых сохраняются как значение. Это распространяется на любые параметры блоков с простыми типами данных, а также на входные (Input) и выходные (Output) параметры сложных типов данных.

Если инициализация параметров блоков не производится, то редактор

в качестве начальных значений параметров будет использовать нулевое значение, наименьшее значение или пробел, в зависимости от типа данных. Для параметров типа BLOCK_DB в качестве значения по умолчанию принимается DB1 (DB0 не допускается, так как он не существует).

Если Вы не задаете размер для переменных типа STRING, то компилятор сам установит 254 байтов в качестве максимальной длины и 0 байтов - в качестве текущего значения размера данных, или компилятор использует установки, взятые с вкладки "Compiler" ("Компилятор"), которая открывается посредством выбора опций меню: *Options -> Customize (Опции -> Установки пользователя).*

## 29.2.7 Формальные параметры

Формальные параметры используются для адресации параметров в программе блока. Формальные параметры имеют такие же имена, как и параметры блока, и используются в выражения программы в качестве операндов.

**Формальные параметры простых типов данных**

Вы можете использовать формальные параметры простых типов данных в любых выражениях в качестве операндов, имеющих такой же тип данных; Вы можете также передавать их значения в параметры вызываемых блоков.

Вы можете назначать несколько видов типа данных для параметров блока простых типов и таким путем организовать доступ к ним для различных формальных параметров.

**Формальные параметры сложных типов данных и типов данных пользователя (UDT)**

Вы можете использовать формальные параметры сложных типов данных и пользовательских типов данных в выражениях присвоения (assignment) в качестве операндов, имеющих такой же тип данных; Вы можете передавать их значения в параметры вызываемых блоков. Вы можете также работать с отдельными компонентами сложных типов данных ARRAY, STRUCT и UDT.

Вы можете назначать несколько видов типа данных для параметров блока сложных типов и таким путем организовать доступ к ним для различных формальных параметров. Такая возможность может быть особенно полезной для типов данных DT и STRING, обработать отдельные байты которых Вам не удастся другим способом.

**Формальные параметры параметрических типов TIMER и COUNTER**

Формальные параметры параметрических типов TIMER и COUNTER могут быть обработаны с использованием SIMATIC-функций таймеров и SIMATIC-функций счетчиков (см. раздел 30.1 "Функции таймеров" и раздел 30.2 "Функции счетчиков"). Значения формальных параметров таких типов могут быть переданы в параметры вызываемых блоков.

**Формальные параметры параметрического типа BLOCK_xx**

С помощью формальных параметров типа BLOCK_xx Вы можете получить доступ к адресам данных в блоке данных (см. раздел 27.2.3 "Косвенная адресация на SCL").

Формальные параметры параметрических типов BLOCK_FB и BLOCK_FC при использовании языка программирования SCL могут только передаваться в вызываемые блоки (нет команд для обработки формальных параметров такого типа в блоке).

**Формальные параметры типов данных POINTER и ANY**

Формальные параметры типов данных POINTER и ANY могут быть переданы целиком в вызываемые блоки при использовании языка программирования SCL. Исключение составляют фактические параметры, размещенные во временных локальных данных, передача которых не допускается.

Вы можете назначать несколько видов типа данных для параметров блока типов данных POINTER и ANY и таким путем организовать доступ к ним для различных формальных параметров. Такая возможность может быть особенно полезной для типов данных ANY для того, чтобы модифицировать ANY-указатель в процессе выполнения программы.

## 29.3 Вызов SCL-блоков

При программировании блоков на языке программировании SCL различают блоки с функциональным значением (function value) и блоки без функционального значения.

Функциональные блоки FB и функции FC без функционального значения являются просто "ветвями" программы (имеют смысл подпрограмм); к этой же группе блоков можно отнести системные функциональные блоки SFB и системные функции SFC без функционального значения.

Функции FC с функциональным значением могут использоваться в выражениях присваивания, а также в других выражениях в качестве операндов. В таблице 29.1 представлен обзор способов вызовов блоков.

Системные функциональные блоки SFB вызываются точно так же, как и функциональные блоки FB, а системные функции SFC вызываются точно так же, как функции FC. Если Вы вызываете системные функциональные блоки SFB с блоком данных, то блок данных размещается в пользовательской программе.

При вызове блока с параметрами параметры блока инициализируются *фактическими параметрами.* Фактические параметры - это такие значения констант, переменных или выражений, с которыми вызванный блок обрабатывается при выполнении программы, и в которых сохраняются результаты этой обработки.

Все параметры блока должны быть инициализированы при вызове функций FC и системных функций SFC.

При вызове функциональных блоков FB и системных функциональных блоков SFB инициализация параметров блока не обязательна. Выходные

параметры при вызове функциональных блоков FB и системных функциональных блоков SFB инициализируются посредством прямого обращения к экземплярным данным вместо использования фактических параметров при вызове.

Таблица 29.1 Вызовы SCL-блоков

| *Вызов функции* | |
|---|---|
| с функциональным значением | без функционального значения |
| переменная := FCx(...);<br>переменная := FC_*name*(...); | FCx(...);<br>FC_*name*(...); |
| *Вызов функционального блока* | |
| с блоком данных | как локальный экземпляр |
| FBx.DBx(...);<br>FB_*name*.DB_*name*(...); | local_*name*(...); |

## 29.3.1 Функции FC без функционального значения

Пример вызова функции FC без функционального значения:

```
FC291(MAX := Maximum,
       IN := InputValue,
      MIN := Minimum,
      OUT := Result);
```

При вызове используется либо абсолютный, либо символьный адрес блока, за которым в скобках следует список параметров.

Все параметры должны быть инициализированы, при этом порядок их следования может быть произвольным. Скобки должны быть указаны, даже если функция FC не имеет параметров.

Если функция имеет единственный входной параметр, то имя параметра при инициализации может быть опущено.

Пример:

Пусть, необходимо запрограммировать преобразование INT-переменной Speed в STRING-переменную Display:

```
Display := I_STRING(Speed);
```

## 29.3.2 Функции FC с функциональным значением

Пример вызова функции FC с функциональным значением:

```
Result := FC292(
   MAX := Maximum,
    IN := InputValue,
   MIN := Minimum);
```

Функции FC с функциональным значением могут использоваться в любых выражениях в качестве операндов с таким же типом данных, например, в выражениях присваивания. В данном примере глобальной переменной *Result* присваивается функциональное значение функции FC 292.

При вызове используется либо абсолютный, либо символьный адрес блока, за которым в скобках следует список параметров.

Все параметры должны быть инициализированы, при этом порядок их следования может быть произвольным. Скобки должны быть указаны, даже если функция FC не имеет параметров.

Если функция имеет единственный входной параметр, то имя параметра при инициализации может быть опущено.

Если Вы используете при вызове блока входной параметр EN, и если этот вход имеет значение FALSE (ЛОЖЬ), то функциональное значение не будет определено (функциональному значению не будет присвоено никакой величины).

### 29.3.3 Функциональный блок со своим собственным блоком данных

Экземплярный блок данных специфицируется, когда производится вызов функционального блока. Он может быть либо запрограммирован в исходной программе (после функционального блока и перед его вызовом), либо сгенерирован в среде программирования на SCL, если он еще существует. Экземплярный блок данных может быть запрограммирован на SCL также и инкрементным способом, без исходной программы, (см. раздел 3.6.1 "Инкрементное программирование блоков данных").

Любой свободный блок данных может использоваться как экземплярный блок данных. При этом пользователь свободен в выборе символьного имени в допустимых пределах.

```
DATA_BLOCK DB291
   FB291
BEGIN
END_DATA_BLOCK
```

Пример вызова функционального блока с экземплярным блоком данных:

```
FB291.DB291(IN := InputValue);
Result := DB291.OUT;
```

Вызов записывается как адрес функционального блока с последующим адресом экземплярного блока, отделенные точкой, и далее - список параметров в скобках. В качестве адресов могут использоваться либо абсолютный, либо символьный адрес блока.

Инициализация параметров функционального блока необязательна. Так как входные/выходные параметры сложных типов сохраняются как указатели, они должны быть инициализированы значащими величинами при первом вызове функционального блока. Если параметр блока не инициализирован, то он сохраняет свое последнее определенное значение. Скобки должны присутствовать в записи, даже если не инициализированы никакие параметры.

Все параметры могут быть адресованы также как глобальные данные с указанием имени экземплярного блока данных и имени параметра. В примере граничные значения определены константами. Они также могут быть инициализированы до вызова функционального блока посредством операторов присваивания:

```
DB291.MAX := Maximum;
DB291.MIN := Minimum;
```

Выходные параметры не могут быть инициализированы при вызове функционального блока. Если требуется, их значения считываются непосредственно из экземплярного блока данных и в дальнейшем обрабатываются без промежуточного хранения.

```
IF DB291.OUT > 10000 THEN ... END_IF;
DB291.MIN := Minimum;
```

### 29.3.4 Функциональный блок как локальный экземпляр

Функциональные блоки могут быть объявлены как "локальные экземпляры" (local instance) и вызваны в другом функциональном блоке. Такие функциональные блоки (вызываемые) сохраняют свои локальные данные в экземплярном блоке данных вызывающего функционального блока.

```
FUNCTION_BLOCK FB290
...
VAR
  Delimiter : FB291;
END_VAR
...
BEGIN
Delimiter (IN := InputValue);
Result := Delimiter.OUT;
...
END_FUNCTION_BLOCK
```

Объявление экземпляров выполняется в статических локальных данных; здесь Вы назначаете имя (например, Delimiter) и назначаете функциональный блок (FB291 или его символьное имя) как данных. К моменту компилирования функциональный блок, который должен вызываться, должен уже существовать или в виде ранее скомпилированного блока в разделе *Blocks (Блоки)*, или в виде (заведомо безошибочной) исходной программы, которая компилируется перед вызовом блока.

Точно такие же действия выполняются при вызове системного функционального блока SFB как локального экземпляра.

Вызов блока в виде локального экземпляра производится как инициализация имени переменной с последующим списком параметров в скобках. Инициализация параметров функционального блока необязательна.

Так как входные/выходные параметры сложных типов сохраняются как указатели, они должны быть инициализированы значащими величинами при первом вызове функционального блока. Если параметр блока не инициализирован, то он сохраняет свое последнее определенное значение. Скобки должны присутствовать в записи, даже если не инициализированы никакие параметры.

Пользователь может создать несколько локальных экземпляров с различными именами для одного и того же функционального блока.

Все параметры локального экземпляра могут быть адресованы также как компоненты структурированной переменной - в записи указываются имя локального экземпляра и имя параметра. В примере граничные значения определены константами. Они также могут быть инициализированы перед вызовом локального экземпляра посредством операторов присваивания:

```
Delimiter.MAX := Maximum;
Delimiter.MIN := Minimum;
```

Выходные параметры не могут быть инициализированы при вызове функционального блока (это также касается локальных экземпляров). Если требуется, их значения считываются как компоненты локального экземпляра:

```
Result := Delimiter.OUT;
```

Вы можете также получить доступ к параметрам локального экземпляра "со стороны" вызывающего функционального блока. Такой доступ организуется как доступ к адресам глобальных данных посредством записи спецификации блока данных (DB290), локального экземпляра (Delimiter) и имени параметра:

```
DB290.Delimiter.MAX := Maximum;
DB290.Delimiter.MIN := Minimum;
Result := DB290.Delimiter.OUT;
```

### 29.3.5 Фактические параметры

При вызове блока происходит инициализация параметров блока текущими значениями (фактическими параметрами - "actual parameter") с помощью операции присваивания (см. предыдущий раздел). В языках программирования STL и SCL в отношении фактических параметров применяются одинаковые операции (см. раздел 19.3 "Фактические параметры"), за исключением следующих моментов:

- Параметры блока сложных типов данных DT и STRING в SCL могут быть инициализированы значениями констант.
- Параметры блока типов POINTER в SCL не могут быть инициализированы значениями констант или с помощью указателя в формате P#Operand. Исключение: определение значения по умолчанию посредством "нулевого" указателя (NIL) разрешается.
- Параметры блока типов ANY не могут быть инициализированы значениями констант или с помощью ANY-указателя в форме P#(DB.(Операнд Тип Размер)). Исключение: определение значения по умолчанию посредством "нулевого" указателя (NIL) разрешается.

- Вы можете инициализировать параметры блока с помощью выражений, которые используют значение такого же типа данных как параметр блока. Например, функция FC с функциональным значением также может быть фактическим параметром.

Примечание:

Если Вы инициализируете формальный параметр типа POINTER или ANY временной переменной при вызове FB или FC, Вы не можете передавать этот параметр из вызванного блока в другой блок. Адреса временных переменных теряют свои значения при передаче в другой блок.

## 29.4 Механизм EN/ENO

При программировании на SCL пользователь имеет возможность проверить отдельные выражения (операции) на правильность выполнения, например, имеется возможность проверить, находится ли результат вычисления функции в допустимом численном диапазоне? Результат подобных проверок сохраняется в так называемой "OK-переменной". Пользователь может также связать назначение "OK-переменной" с вызывающим блоком посредством выходного параметра ENO блока. Наконец, имеется возможность выполнять вызов блока посредством EN в зависимости от условий.

Вы можете использовать заранее определенные переменные EN и ENO для всех блоков (FC, SFC, FB, SFB, а также для IEC-функций), для всех стандартных функций (например, функции сдвига и функции преобразования), кроме функций таймеров и функций счетчиков.

В главе 15 "Биты состояния", в разделе 15.4 "Использование двоичного результата" описывается, как используется механизм EN/ENO в стандартных языках программирования.

### 29.4.1 OK-переменная

В языке программирования SCL существует инициализируемая переменная с именем "OK" и типом данных BOOL. Эта переменная сообщает об ошибках, возникающих при выполнении программы в SCL-блоке, но только в случае, если Вы выбрали опцию "Set OK flag" ("Установить OK-флаг") на вкладке "Compiler" ("Компилятор"), которая открывается при выборе опций меню: *Options -> Customize (Опции -> Установки пользователя)* в редакторе SCL-программ.

Редактор или компилятор не проверяют, установлена данная опция или нет, если Вы используете OK-переменную в программе.

В начале блока OK-переменная имеет значение TRUE (ИСТИНА). При возникновении программной ошибки OK-переменная сбрасывается в состояние FALSE (ЛОЖЬ). Вы можете проверять OK-переменную с помощью соответствующих SCL-операций, а также можете присваивать этой переменной требуемые значения в любое время.

```
SUM := SUM + IN;
IF OK
     THEN (* не произошло ошибок *);
     ELSE (* ошибка в операции сложения *);
END_IF;
```

В данном примере на OK-переменную влияет результат обработки арифметического выражения и некоторых функций преобразования (см. раздел 30.5.2 "Явные функции преобразования"). Если происходит ошибка при выполнении стандартных функций, таких, как математические функции, об этом сообщается посредством выхода ENO (см. далее по тексту).

При выходе их блока значение OK-переменной назначается выходу ENO.

## 29.4.2 Выход ENO (ENO output)

Вызываемый блок сохраняет значение OK-переменной в выходе ENO ("ENO output" = "Enable output" = "Выход разблокирован"). ENO имеет тип данных BOOL. После вызова блока значение ENO может быть использовано для определения того, правильно ли был обработан блок (в этом случае ENO = TRUE [ИСТИНА]) или произошла ошибка (в этом случае ENO = FALSE [ЛОЖЬ]).

```
FC15 (In1:= ..., In2:= ...);
IF ENO
     THEN (* не произошло ошибок *);
     ELSE (* произошла ошибка *);
END_IF;
```

Если необходимо с помощью ENO после вызова блока передать в вызывающий блок сообщение о групповой ошибке, то установите соответствующим образом OK-переменную:

```
FC15 (In1:= ..., In2:= ...);
OK := ENO;
```

Вы можете также назначить значение для выхода ENO в блоке с помощью соответствующей установки OK-переменной:

```
IF (* произошла ошибка *)
   THEN OK := FALSE; RETURN;
END_IF;
```

ENO является не параметром блока, а последовательностью выражений, генерируемой редактором программ, в случае использования ENO. ENO не объявляется. ENO может быть немедленно проверен после вызова блока.

Если Вы управляете вызовом блока с помощью входа EN (см. следующий раздел), и вход EN имеет значение FALSE (ЛОЖЬ), так что блок не может быть запущен для выполнения, то выход ENO также будет иметь значение FALSE (ЛОЖЬ).

Примечание:

Если в блоке, созданном с использованием стандартных языков программирования, для сообщения об ошибках используется "двоичный результат" BR, то в SCL Вы можете проверять сообщения об ошибках с помощью выхода ENO после вызова блока (см. раздел 15.4 "Использование двоичного результата").

### 29.4.3 Вход EN (EN input)

Вы можете управлять вызовами блока с помощью входа EN. EN имеет тип данных BOOL. Если вход EN инициализирован значением TRUE (ИСТИНА), то вызываемый блок будет вызван для выполнения. Если EN инициализирован значением FALSE (ЛОЖЬ), то вызываемый блок соответственно не будет вызван для выполнения. При этом будет выполнен переход к следующему оператору после вызова блока.

```
FC15 (EN := I1.0,
      In1:= ...,
      In2:= ...);
(* FC15 выполняется только в случае, когда I1.0 = "1" *)
```

Если Вы не используете EN, то блок будет выполняться всегда.

EN является не параметром блока, а последовательностью выражений, генерируемой редактором программ, в случае использования EN. EN не объявляется. Вы должны использовать EN в списке параметров блока также, как любой другой входной параметр.

Вы можете также назначить значение для входа EN значением ENO; в этом случае вызываемый блок будет обрабатываться только в том случае, если обработка ранее вызванного блока успешно завершена.

Пример:

Если необходимо вызывать блок FC16 только в случае, если обработка FC15 успешно завершена, то фрагмент программы может выглядеть следующим образом:

```
FC15 (EN := I1.0,
      In1:= ...,
      In2:= ...);
FC16 (EN := ENO,
      In1:= ...,
      In2:= ...);
```

Если ранее никакой блок не вызывался на том же уровне вызовов, то ENO будет иметь значение TRUE (ИСТИНА).

Необходимо отметить, что функция FC или системная функция SFC принимает неопределенное значение (произвольное значение функции), если Вы управляете вызовом функции с помощью входа EN, и вход EN имеет значение FALSE (ЛОЖЬ).

# 30 SCL-функции

## 30.1 Функции таймеров

Таймеры в системной памяти CPU в языке программирования SCL адресуются так же, как функции с возвращаемым (функциональным) значением. Названия функций, соответствующих различным режимам таймера, представлены ниже:

- S_PULSE ("pulse timer" - "режим управляемого импульса")
- S_PEXT ("extended pulse" - "режим расширенного импульса")
- S_ODT ("ON delay" - "с задержкой включения")
- S_ODTS ("latching ON delay" - "с задержкой включения с памятью")
- S_OFFDT ("OFF delay" - "с задержкой выключения")

Все функции таймера имеют параметры, показанные в таблице 30.1.

Таблица 30.1 Параметры для SIMATIC-функций таймеров

| Параметр | Объявление | Тип данных | Значение |
|---|---|---|---|
| T_NO | INPUT | TIMER | Адрес таймера |
| S | INPUT | BOOL | Параметр запуска таймера |
| TV | INPUT | S5TIME | Установленное значение таймера |
| R | INPUT | BOOL | Параметр сброса таймера |
| Функциональное значение | OUTPUT | S5TIME | Текущее значение времени в двоично-десятичном коде (BCD) |
| Q | OUTPUT | BOOL | Состояние таймера |
| BI | OUTPUT | WORD | Текущее значение времени в двоичном коде (binary) |

Пример вызова функций таймеров:

```
Time_BCD  := S_PULSE(
     T_NO := Timer_address,
     S    := Start_input,
     TV   := Timer_duration,
     R    := Reset,
     Q    := Timer_status,
     BI   := Binary_time);
```

Диаграммы работы функций таймеров подробно рассмотрены в главе 7 "Функции таймеров (Timer Functions)".

Необходимо отметить, что функция разблокирования таймера (см. гл. 7 "Функции таймеров (Timer Functions)" не поддерживается в SCL.

При инициализации параметров функций таймеров применяются следующие правила:

- параметр T_NO должен быть всегда инициализирован
- пара параметров S и TV могут быть не инициализированы
- параметр Q может быть не инициализирован
- параметр BI может быть не инициализирован

В дополнение к SIMATIC-функциям таймеров в специализированных CPU поддерживаются также IEC-функции таймеров как системные функциональные блоки SFB:

- SFB 3 TP
  Pulse generation - функция генерации импульса
- SFB 4 TON
  ON delay - функция генерации импульса с задержкой включения
- SFB 5 TOF
  OFF delay - функция генерации импульса с задержкой выключения

Данные функции описаны в разделе 7.7 "IEC-функции таймеров (IEC-Timer Functions)".

Функциональные блоки хранятся в "стандартной библиотеке" *Standard library* в разделе *System Function Blocks (Системные функциональные блоки).*

Примеры применения SIMATIC-функций таймеров и IEC-функций таймеров Вы можете найти на дискете, прилагаемой к книге, в библиотеке "SCL_Book" в исходном файле "Timer Functions" ("Функции таймеров") в разделе "30 SCL Functions" ("30 SCL-функции").

## 30.2 Функции счетчиков

Счетчики в системной памяти CPU в языке программирования SCL адресуются так же, как функции с возвращаемым (функциональным) значением. Названия функций, соответствующих различным режимам счетчика, представлены ниже:

- S_CU ("up counter" - "функция счетчика прямого счета")
- S_CD ("down counter" - "функция счетчика обратного счета")
- S_CUD ("up-down counter" - "функция счетчика прямого и обратного счета")

Параметры для всех SIMATIC-функций счетчиков представлены в таблице 30.2.

Таблица 30.2 Параметры для SIMATIC-функций счетчиков

| Параметр | Объявление | Тип данных | Значение |
|---|---|---|---|
| C_NO | INPUT | COUNTER | Адрес счетчика |
| CU | INPUT | BOOL | Прямой счет |
| CD | INPUT | BOOL | Обратный счет |
| S | INPUT | BOOL | Параметр запуска счетчика |
| PV | INPUT | S5TIME | Установленное значение счетчика |
| R | INPUT | BOOL | Параметр сброса счетчика |
| Функциональное значение | OUTPUT | WORD | Текущее значение счетчика в двоично-десятичном коде (BCD) |
| Q | OUTPUT | BOOL | Состояние счетчика |
| CV | OUTPUT | WORD | Текущее значение счетчика в двоичном коде (binary) |

Пример вызова функций счетчиков:

```
BCD_Count_Value  := S_CU(
     C_NO := Count_address,
     CU   := Count_up,
     S    := Set_input,
     PV   := Count_value,
     R    := Reset,
     Q    := Counter_status,
     BI   := Binary_count_value);
```

Диаграммы работы функций счетчиков подробно рассмотрены в главе 8 "Функции счетчиков (Counter Functions)".

Необходимо отметить, что функция разблокирования счетчиков не поддерживается в SCL.

При инициализации параметров функций счетчиков применяются следующие правила:

- применение параметра CD не допускается вместе с функцией счетчика S_CU
- применение параметра CU не допускается вместе с функцией счетчика S_CD
- параметр C_NO должен быть всегда инициализирован
- в зависимости от выбранного режима счетчика должен быть установлен один из параметров CD или CU
- пара параметров S и PV могут быть не инициализированы
- параметр Q может быть не инициализирован
- параметр CV может быть не инициализирован

В качестве константы для инициализации параметра PV счетчика

может быть применено целое число типа INT с диапазоном значений от 0 до 999 или шестнадцатеричное число с диапазоном значений от 16#000 до 16#3E7.

В дополнение к SIMATIC-функциям счетчиков в специализированных CPU поддерживаются также IEC-функции счетчиков как системные функциональные блоки SFB:

- SFB 0 CTU
  ("up counter" - "функция счетчика прямого счета")
- SFB 1 CTD
  ("down counter" - "функция счетчика обратного счета")
- SFB 2 CTUD
  ("up-down counter" - "функция счетчика прямого и обратного счета")

Данные функции описаны в разделе 8.6 "IEC-функции счетчиков (IEC-Counter Functions)". Функциональные блоки хранятся в "стандартной библиотеке" *Standard library* в разделе *System Function Blocks (Системные функциональные блоки).*

Примеры применения SIMATIC-функций счетчиков и IEC-функций счетчиков Вы можете найти на дискете, прилагаемой к книге, в библиотеке "SCL_Book" в исходном файле "Counter Functions" ("Функции счетчиков") в разделе "30 SCL Functions" ("30 SCL-функции").

## 30.3 Математические функции

В языке программирования SCL поддерживаются следующие математические функции:

- Тригонометрические функции:

  | | |
  |---|---|
  | SIN | функция синуса |
  | COS | функция косинуса |
  | TAN | функция тангенса |

- Обратные тригонометрические функции (Arc-функции):

  | | |
  |---|---|
  | ASIN | функция арксинуса |
  | ACOS | функция арккосинуса |
  | ATAN | функция арктангенса |

- Логарифмические функции:

  | | |
  |---|---|
  | EXP | экспоненциальная функция по основанию *e* |
  | EXPD | экспоненциальная функция по основанию 10 |
  | LN | натуральный логарифм |
  | LOG | десятичный логарифм |

- Тригонометрические функции:

  | | |
  |---|---|
  | ABS | функция выделения абсолютного значения |
  | SQR | функция нахождения квадрата числа |
  | SQRT | функция взятия квадратного корня |

Математические числа обрабатывают числа форматов INT, DINT и REAL.

При использовании в функции числа в формате INT или DINT в качестве входного параметра, оно автоматически преобразуется в число в формате REAL.

Математические функции работают с числами, внутренне представляемыми в формате REAL; результат также получается в формате REAL. Исключение составляет функция ABS, которая выдает результат того же типа, к которому относилось исходное число.

Тригонометрические функции рассматривают входные параметры, как углы, выраженные в радианах, в диапазоне от 0 до 2π (где π = 3,141593e+00), что соответствует диапазону углов от 0° до 360°.

| Функция | Допустимый диапазон | Возвращаемое значение |
|---|---|---|
| ASIN | -1 ... +1 | -π/2 ... +π/2 |
| ACOS | -1 ... +1 | 0 ... π |
| ATAN | нет ограничений | -π/2 ... +π/2 |

Примеры:

Расчет реактивной мощности Reactive_power, которая определяется произведением напряжения Voltage на ток Current и на синус фазового сдвига между ними:

```
Reactive_power  := Voltage * Current * SIN(φ);
```

Расчет объема жидкости Volume, который определяется произведением π числа PI на квадрат радиуса основания сосуда Radius и на уровень заполнения сосуда Level:

```
Volume  := PI * SQR(Radius) * Level;
```

Расчет длины гипотенузы по известным величинам катетов:

```
c  := SQRT(SQR(a) + SQR(b));
```

## 30.4 Функции сдвига (Shifting) и циклического сдвига (Rotating)

Общий для функций сдвига (Shifting) и циклического сдвига (Rotating) способ вызова имеет вид:

```
Result  := Function(
    IN  := Input_value,
     N  := Shift_number);
```

Функции сдвига и циклического сдвига имеют два входных параметра. Параметр N показывает число битов, на которое необходимо произвести операцию сдвига или циклического сдвига, этот параметр относится к типу INT. Параметр IN определяет переменную, с которой необходимо выполнить операцию сдвига или циклического сдвига, этот параметр относится к классу типов ANY_BIT (то есть, к типам BOOL, BYTE, WORD,

DWORD). При этом значение функции относится к тому же типу, что и входное значение.

Примеры:

```
MW14  := SHL(IN := MW12, N := 2);

res_dword := ROR(
       IN := in_dword,
        N := shift_int);
```

Таблица 30.3 Функции сдвига (Shifting) и циклического сдвига (Rotating)

| | | |
|---|---|---|
| SHL | Сдвиг влево | Входное значение IN сдвигается влево на N битов; освободившиеся позиции заполняются нулями. |
| SHR | Сдвиг вправо | Входное значение IN сдвигается вправо на N битов; освободившиеся позиции заполняются нулями. |
| ROL | Циклический сдвиг влево | Входное значение IN сдвигается влево на N битов; освободившиеся позиции заполняются значениями "вытолкнутых" из аккумулятора битов. |
| ROR | Циклический сдвиг вправо | Входное значение IN сдвигается вправо на N битов; освободившиеся позиции заполняются значениями "вытолкнутых" из аккумулятора битов. |

## 30.5 Функции преобразования (Conversion Functions)

При совместной обработке переменных с помощью операторов переменные должны (внутренне) относиться к одному типу данных. Это касается и инструкций присваивания, и операций инициализации параметров функций и параметров блоков. Если переменная не может быть обработана, так как принадлежит к другому типу данных, то ее тип должен быть изменен. Для этой цели применяются функции преобразования типов данных (Conversion Functions).

В языке программирования SCL поддерживаются два типа функций преобразования. Первый тип функций преобразования относится к классу A ("Class A"). Эти функции выполняются в SCL автоматически ("неявно"), так как они не связаны с потерей информации (например, преобразование данных типа BYTE в данные типа WORD). Второй тип функций преобразования относится к классу B ("Class B"). Эти функции пользователь должен инициировать самостоятельно, "явно", (например, преобразование данных типа REAL в данные типа INT). Любая потеря информации может быть предупреждена с помощью соответствующего контроля данных или же пользователь может проверять "OK-переменную" для проверки результатов выполнения операций (возможность задействования OK-переменной должна быть инициирована в *Compiler Properties [Свойствах компилятора]*).

Пользователь имеет возможность преобразовывать и обрабатывать переменные типов DATE_AND_TIME и STRING с помощью IEC-функций. Описание стандартной библиотеки *Standard Library* и раздела *IEC*

*Function Blocks [IEC функциональные блоки]*) Вы можете найти в главе 31 "IEC-функции".

### 30.5.1 Неявные функции преобразования (Implicit Conversion Functions)

Неявные функции преобразования типов данных выполняются в SCL автоматически ("неявно"). Тем не менее, Вы можете запрограммировать эти функции преобразования типов, например, для того, чтобы улучшить ясность или читаемость Вашей программы.

В таблице 30.4 показаны поддерживаемые в языке программирования SCL неявные функции преобразования типов данных.

Таблица 30.4 Неявные функции преобразования (Implicit Conversion Functions)

| Функция | OK | Особенности операции преобразования |
|---|---|---|
| BOOL_TO_BYTE | N | Заполнение нулями незаполненных позиций слева |
| BOOL_TO_WORD | N | |
| BOOL_TO_DWORD | N | |
| BYTE_TO_WORD | N | |
| BYTE_TO_DWORD | N | |
| WORD_TO_DWORD | N | |
| INT_TO_DINT | N | Заполнение знаком незаполненных позиций слева |
| INT_TO_REAL | N | - |
| DINT_TO_REAL | N | При преобразовании, помимо прочего, теряется точность |
| CHAR_TO_STRING | Y | Преобразование в строку символов, состоящую из одного символа |

При преобразовании данных из символьной переменной в строку символов (функция CHAR_TO_STRING), создается строка STRING, длина которой равна 1 (строка состоит из одного символа), и OK-переменная устанавливается в состояние FALSE (ЛОЖЬ).

Примеры:

```
                    //Преобразование типов данных
MB10       := M7.0;       //BOOL -> BYTE
real_var   := int_var;    //INT  -> REAL
string_var := char_var;   //CHAR -> STRING
```

В рассмотренном примере меркер M10.7 принимает значение меркера M7.0. Остальные биты устанавливаются в состояние "0".

### 30.5.2 Явные функции преобразования (Explicit Conversion Functions)

Пользователь должен самостоятельно инициировать в программе явные функции преобразования типов данных; тем не менее, при использовании некоторых из этих явных функций собственно преобразование данных не происходит и не выполняется никакой код (см. табл. 30.5 с комментарием "принимается без изменения"). На состояние переменной OK влияют некоторые из функций преобразования.

Примеры:

```
MB10      := CHAR_TO_BYTE(char_var);
int_var   := WORD_TO_INT(MW20);
real_var  := DWORD_TO_REAL(MD30);
```

Необходимо отметить, что в последнем примере преобразование данных не происходит. Битовая структура двойного слова меркеров принимается без изменений в переменную типа REAL.

В случае преобразования вида REAL_TO_xxx переменная OK устанавливается в состояние FALSE (ЛОЖЬ), даже в случае, если корректное число типа REAL недоступно.

## 30.6 Программирование Ваших собственных функций на SCL

Если Вы не можете найти подходящих функций среди набора стандартных SCL-функций и IEC-функций, язык SCL позволяет Вам создать свои собственные функции, которые Вы можете адаптировать к Вашим требованиям.

Подходящий тип блоков для этих целей - это функции FC с функциональным значением. Вопросы программирования функций FC с функциональным значением и вопросы по организации вызовов этих функций рассматриваются в разделе 29.2.2 "Функции FC с функциональным значением" и в разделе 29.3.2 "Функции FC с функциональным значением" соответственно.

Во многих случаях языковых ресурсов SCL не хватает для программирования требуемых функций. В таких случаях для создания пользовательских функций возможно также использование языка программирования STL (см. раздел 30.7 "Программирование Ваших собственных функций на STL"). Однако, принципиальная возможность использования видов типов данных ("data type views") в SCL позволяет обрабатывать сложные переменные. В разделе 27.1.9 "Виды типа данных (Data Type Views)" рассмотрено, какие виды типа данных для каких переменных Вы можете использовать.

**Поразрядная обработка переменных простых видов**

Например, Вам необходимо обработать отдельные биты переменной формата двойного слова (doubleword) некоторым образом, например, методом опроса битов или логического комбинирования с последующей

Таблица 30.4 Явные функции преобразования (Implicit Conversion Functions)

| Функция | Преобразование | OK | Примечания |
|---|---|---|---|
| BYTE_TO_BOOL | Принимается младший значащий бит | Y | Если бит в не принятой части переменной = "1", то OK= TRUE (ИСТИНА) |
| WORD_TO_BOOL | | Y | |
| DWORD_TO_BOOL | | Y | |
| WORD_TO_BYTE | Принимается младший значащий байт | Y | |
| DWORD_TO_BYTE | | Y | |
| DWORD_TO_WORD | Принимается младшее значащее слово | Y | |
| CHAR_TO_BYTE | При присваивании не изменяется | N | |
| BYTE_TO_CHAR | При присваивании не изменяется | N | |
| CHAR_TO_INT | Старший значащий байт заполняется нулями | N | |
| INT_TO_CHAR | Младший значащий байт принимается без изменения | Y | Если бит в левом байте = "1", то OK = TRUE (ИСТИНА) |
| STRING_TO_CHAR | Принимается первый символ | Y | Если длина переменной STRING <> "1", то OK = FALSE (ЛОЖЬ) |
| WORD_TO_INT | Принимается без изменения | N | |
| DWORD_TO_DINT | | N | |
| INT_TO_WORD | | N | |
| DINT_TO_DWORD | | N | |
| REAL_TO_DWORD | | N | Нет преобразования! |
| DWORD_TO_REAL | | N | Нет преобразования! |
| DINT_TO_INT | Копируются биты для знака | Y | Если численный диапазон нарушен, то OK = FALSE (ЛОЖЬ) |
| REAL_TO_INT | Округляется до целого INT | Y | |
| REAL_TO_DINT | Округляется до целого DINT | Y | |
| ROUND | Преобразование REAL в DINT с округлением | Y | Если численный диапазон нарушен, то OK = FALSE (ЛОЖЬ) |
| TRUNC | Преобразование REAL в DINT без округления ("усечение" дробной части) | Y | |
| DINT_TO_TIME | Принимается без изменения | N | |
| DINT_TO_TOD | | Y | Если нарушен диапазон для TOD, то OK = FALSE (ЛОЖЬ) |
| DINT_TO_DATE | | Y | Если левое слово имеет назначение, то OK = FALSE (ЛОЖЬ) |
| DATE_TO_DINT | | N | |
| TIME_TO_DINT | | N | |
| TOD_TO_DINT | | N | |
| WORD_TO_BLOCK_DB | Принимается без изменения | N | |
| BLOCK_DB_TO_WORD | Принимается без изменения | N | |

записью в другой бит. Для этой цели Вы можете применить виды типа данных к переменной в форме массива битов с последующей адресацией отдельных битов как элементов этого массива.

```
VAR_TEMP
  DW_VAR   :  DWORD;
  Pattern AT DW_VAR : ARRAY[0..31] OF BOOL;
END_VAR
...
Pattern[1] := Pattern[10]&Pattern[11]
...
```

В данном маленьком примере 10-й и 11-й биты переменной DW_VAR комбинируются в логической операции AND (И), и результат записывается в 1-й бит той же переменной.

### Обработка переменных типов DT и STRING

Переменные типов DT и STRING в языке программирования SCL обычно обрабатываются "целиком", например, при инициализации параметров функции или при пересылке данных из одного параметра в другой. Вы можете использовать IEC-функции из стандартной библиотеки STEP 7 Standard Library для обработки переменных типов DT или STRING.

При необходимости обработки части переменной типа DT или STRING посредством SCL-инструкций используйте виды типа данных (Data Type View) для переменной, которую нужно обработать. Байтовые массивы (OF BYTE) подходят для представления переменных типов DT и STRING (см. табл. 30.6).

### Пример SCL-функции

Рассмотрим пример функции "Hour" ("Час"), которая извлекает информацию о часе из данных в формате DT и передает эту информацию в возвращаемое значение функции Hour.

```
FUNCTION Hour : INT
VAR_INPUT
  DAT : DT;
  TMP AT DAT : ARRAY[1..8] OF BYTE;
END_VAR
BEGIN
Hour :=
  WORD_TO_INT(SHR(IN:=TMP[4],N:=4))*10 +
  WORD_TO_INT(TMP[4] AND 16#0F);
END_FUNCTION
(* Считывание данных о времени из CPU и
вызов функции "Hour" *)
SFC_ERROR := READ_CLK(DATE_TIME);
IF Hour (DATE_TIME) >= 18
  THEN FINISH_WORK := TRUE;
END_IF;
```

Таблица 30.6 Часто используемые виды типа данных (Data Type View)

| Тип переменной | Виды типа данных (Data Type View) | Особенности операции преобразования |
|---|---|---|
| Простой | Массив элементов типа BOOL | `TEMPVAR : DWORD;`<br>`VEIW AT TEMPVAR : ARRAY[0..31] OF BOOL;` |
| DT | Массив элементов типа BYTE | `TEMPVAR : DT;`<br>`VEIW AT TEMPVAR : ARRAY[1..8] OF BYTE;` |
| STRING | Массив элементов типа CHAR | `TEMPVAR : STRING[max];`<br>`VEIW AT TEMPVAR : ARRAY[1..max] OF CHAR;` |
| ARRAY | STRUCT | `TEMPVAR : ARRAY[0..255] OF BYTE;`<br>`VEIW1 AT TEMPVAR : STRUCT`<br>`name : data_type;`<br>`.... : ...`<br>`END_STRUCT;`<br>`VEIW2 AT TEMPVAR : STRUCT`<br>`name : data_type;`<br>`.... : ...`<br>`END_STRUCT;` |
| ANY | STRUCT | `TEMPVAR : ANY;`<br>`VEIW1 AT TEMPVAR : STRUCT`<br>`ID : BYTE;`<br>`TYP : BYTE;`<br>`ANZ : INT;`<br>`DBN : INT;`<br>`PTR : DWORD;`<br>`END_STRUCT;` |

**Различные виды массивов и структур**

Переменным типов ARRAY и STRUCT могут быть назначены виды типа данных (Data Type View), сами имеющие соответственно типы ARRAY и STRUCT. Применение таких видов типа данных может найти себе место при создании области "почтового ящика" ("mailbox") для передачи или приема фреймов сообщений.

Например, Вы можете установить максимальную длину области "почтового ящика" ("mailbox"), используя байтовый массив. Для каждого фрейма сообщения, который необходимо обработать в "почтовом ящике" ("mailbox"), Вы можете применить для работы с "почтовым ящиком" вид типа данных, который имеет структуру фрейма сообщения. Такой вид типа данных должен быть специально предназначен и приспособлен для фрейма сообщения, следовательно, он может иметь размер, меньший, чем размер области "почтового ящика" ("mailbox").

**Управление указателями ANY**

При создании в области временных локальных данных переменной типа

ANY компилятор интерпретирует эту переменную как указатель и передает ее прямо во входной параметр типа ANY, например, параметр вызываемого блока (см. раздел 29.2.4 "Временные локальные данные").

Вы можете управлять этим ANY-указателем в режиме выполнения программы с помощью видов типа данных (Data Type View), что позволит Вам динамически задавать различные исходные области данных для копирования блоков.

Пример:

Пусть, необходимо выполнить копирование данных из области, которая определяется переменными *DATABLOCK, DATASTART* и *NUM_OF_BYTES* в переменную, которая имеет имя *SEND_MAILBOX.*

```
FUNCTION_BLOCK
VAR_INPUT
  AREA         : ANY;
  DATABLOCK    : INT;
  DATASTART    : INT;
  NUM_OF_BYTES : INT;
END_VAR
VAR_TEMP
  SFC_ERROR    : INT;
  SEND_MAILBOX : ANY;
  VEIW AT SENDEFACH : STRUCT;
     ID  : WORD;
     TYP : BYTE;
     NUM : INT;
     DBN : INT;
     PTR : DWORD;
     END_STRUCT;
END_VAR
BEGIN
VEIW.ID   := 16#10;
VEIW.TYP  := 16#02;
VEIW.NUM  := NUM_OF_BYTES;
VEIW.DBN  := DATABLOCK;
VEIW.PTR  := INT_TO_WORD(8*DATASTART);
SFC_ERROR := BLKMOV(
  SRCBLK  := AREA,
  DSTBLK  := SEND_MAILBOX);
END_FUNCTION_BLOCK
(* Вызов функционального блока *)
COPY.COPYDATA(
  AREA         := SEND_MAILBOX,
  DATABLOCK    := 309,
  DATASTART    := 32,
  NUM_OF_BYTES := 32);
```

Вы можете найти еще несколько примеров по данной теме на прилагаемой дискете в библиотеке SCL_Book в разделе "General Examples" ("Общие примеры").

## 30.7 Программирование Ваших собственных функций на STL

Использование функций FC с функциональным значением позволяет Вам создавать свои собственные функции с использованием языка программирования SCL. Тем не менее, так как Вы можете совместно использовать в Вашей программе блоки, созданные с использованием различных языков программирования, то Вы можете также создавать функции на языке STL и затем вызывать их из SCL-программы. Это позволяет Вам использовать более обширный набор функций STL, такие, например, как прямой доступ к адресам переменных или адресация с использованием адресного регистра.

Вы можете программировать STL-блоки двумя различными способами: инкрементным или путем написания исходных текстов программ (см. раздел 3.4 "Программирование кодовых блоков на STL").

Для второго способа создания программ - путем написания исходных текстов программ - процедура идентична созданию программ SCL-блоков:

1. Создание исходного STL-файла в разделе исходных файлов *Source Files*.
2. Открытие исходного STL-файла в разделе исходных файлов *Source Files* двойным щелчком кнопки манипулятора "мышь" на объекте.
3. Программирование исходной программы на языке программирования STL (см. замечания ниже по тексту).
4. Если Вы выбрали символьные имена для функций, то заполните таблицу символов *Symbol Table*.
5. Компилирование исходной STL-программы, для того, чтобы в разделе блоков программы *Blocks (Блоки)* были сохранены скомпилированные блоки функций.
6. Теперь Вы можете вызывать новые функции таким же образом, как, например, стандартные функции в SCL-программе.

При написании исходных текстов STL-программ используются почти такие же ключевые слова, как и при программировании блоков на языке SCL (см. таблицу 3.3 в разделе 3.4.3 "Программирование кодовых блоков на STL, ориентированное на создание исходных файлов").

Основное различие, относящееся к функциям с функциональным значениям, заключается в том, что функциональное значение в STL-программе имеет имя RET_VAL (или ret_val). То есть, Вы назначаете значение функции переменной RET_VAL в программе.

В качестве наших небольших примеров мы выбрали функции сканирования, запуска и сброса функций таймера. Целью создания этих функций является упрощение использования функций таймеров. В главе 7 "Функции таймеров (Timer Functions)" рассматривается, как функции таймеров программируются с использованием языка STL.

Функция T_SCAN возвращает состояние указанного в параметре адреса таймера:

```
FUNCTION T_SCAN : BOOL
VAR_INPUT
  T_NO : TIMER;
END_VAR
BEGIN
  U T_NO;  = RET_VAL;
END_FUNCTION
```

Функция T_PULSE запускает таймер в режиме управляемого импульса:

```
FUNCTION T_PULSE : VOID
VAR_INPUT
  T_NO  : TIMER;
  START : BOOL;
  Time_value : S5TIME;
END_VAR
BEGIN
  U Start; L Time_value; SI T_NO;
END_FUNCTION
```

Функция T_RESET сбрасывает таймер при каждом ее вызове:

```
FUNCTION T_RESET : VOID
VAR_INPUT
  T_NO  : TIMER;
END_VAR
BEGIN
  Set; R T_NO;
END_FUNCTION
```

После компиляции эти функции могут быть использованы в SCL-программе, например, следующего вида:

```
IF NOT T_SCAN(T1)
   THEN T_PULSE (T_NO  := T2,
                 START := I1.0,
                 Time_value := S5T#5s);
   ELSE T_RESET(T3);
END_IF;
```

Вы можете найти эти примеры на прилагаемой к книге дискете в библиотеке SCL_Book в разделе "General Examples" ("Общие примеры").

## 30.8 Краткое описание примеров использования языка SCL

### 30.8.1 Пример "Conveyor" ("Конвейер")

В примере "Conveyor" ("Конвейер") представлено применение двоичных логических операций, функций установки/сброса и вызовов блоков. Он разработан для языка программирования STL. Если Вы освоили STL, и теперь желаете изучить SCL, то Вы найдете здесь предложения по преобразованию типичных STL-функций в SCL-функции.

На рисунке 30.1 показана программа и структура данных примера. Детальное описание примеров дано в разделах:

- 5.5 "Пример системы управления ленточным конвейером" (FC 11)
- 8.7 "Пример счетчика деталей" (FC 12)
- 19.5.1 "Пример: ленточный конвейер" (FC 21)
- 19.5.2 "Пример: счетчик деталей" (FC 22)
- 19.5.3 "Пример: подающий механизм" (FC 20)

Вы можете найти эти примеры на прилагаемой к книге дискете в библиотеке SCL_Book в разделе "Conveyor Example" ("Пример конвейера").

**Пример конвейера**

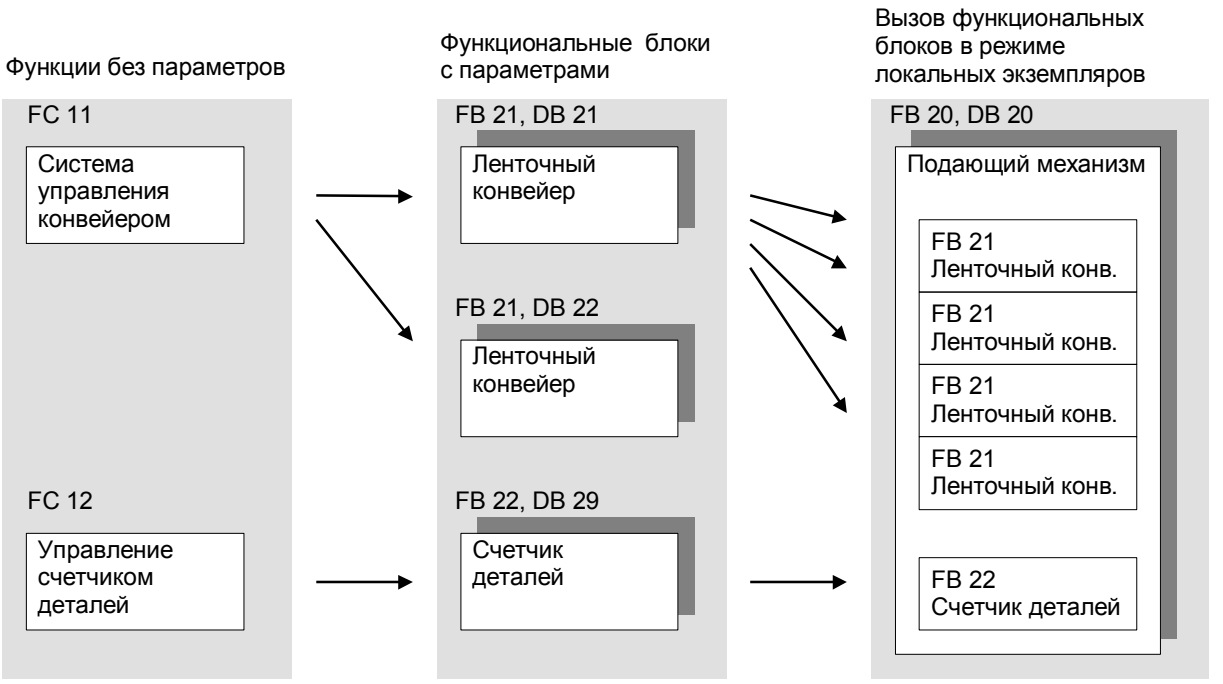


Рис. 30.1 Данные и структура программы примера ленточного конвейера

### 30.8.2 Пример фрейма сообщения

Пример "Message Frame" ("Фрейм сообщения") показывает, как обрабатывать пользовательские данные и как копировать данные из областей. В этом контексте разработан пример на языке программирования STL с использованием косвенной адресации посредством адресных регистров и с управлением ANY-указателем (см. раздел 26.4 "Краткое описание примера "Message Frame Example" (Пример фрейма сообщения)").

Те же функции можно было бы реализовать, используя язык программирования SCL, и в определенной степени более изящно. Здесь пользователь может использовать возможность обработки во время выполнения программы отдельных элементов массива с адресацией их посредством индексов (см. рис. 30.2).

Кроме того, язык программирования SCL лучше подходит для формулирования задачи (в результате программа получается яснее, что облегчает ее использование и снижает вероятность появления ошибок). Тем не менее, с помощью прямого доступа к переменным (чего нет в SCL) при использовании языка STL легко достигается требуемый результат при создании программы. Таким образом, при программировании одной и той же задачи при использовании языков SCL и STL применяются несколько различные подходы.

Вы можете найти программу для этого примера на прилагаемой к книге дискете в библиотеке SCL_Book в разделе "Message Frame Example" ("Пример фрейма сообщения").

### 30.8.3 Общие примеры

В общих примерах представлена обработка переменных сложных типов и управление ANY-указателем с помощью видов типа данных.

Следующие функции выполняют преобразования типов данных средствами языка SCL:

- FC 61 DT_TO_STRING
  функция для извлечения даты и преобразования в переменную типа STRING
- FC 62 DT_TO_DATE
  функция для извлечения даты и преобразования в переменную типа DATE
- FC 63 DT_TO_TOD
  функция для извлечения времени суток и преобразования в переменную типа TOD

Следующие функциональные блоки позволяют получить доступ к переменным сложных типов, а также - выполнить обработку данных в кольцевом буфере и в FIFO регистре:

- FB 61 Variable_length ("длина переменной")
- FB 62 Checksum ("контрольная сумма")
- FB 63 Ring_buffer ("кольцевой буфер")
- FB 64 FIFO_register ("FIFO регистр")

В исходных программах "STL Functions" ("STL-функции") и "Call STL Functions" ("Вызов STL-функций") показано, как записывать на STL простые функции и как вставлять эти функции в Вашу SCL-программу. Вы научитесь использовать SIMATIC таймеры и счетчики, используя язык программирования SCL, так же как в стандартных языках программирования.

Вы можете найти рассмотренные примеры на прилагаемой к книге дискете в библиотеке SCL_Book в разделе "General Example" ("Общие примеры").

**Пример работы с фреймом сообщения**

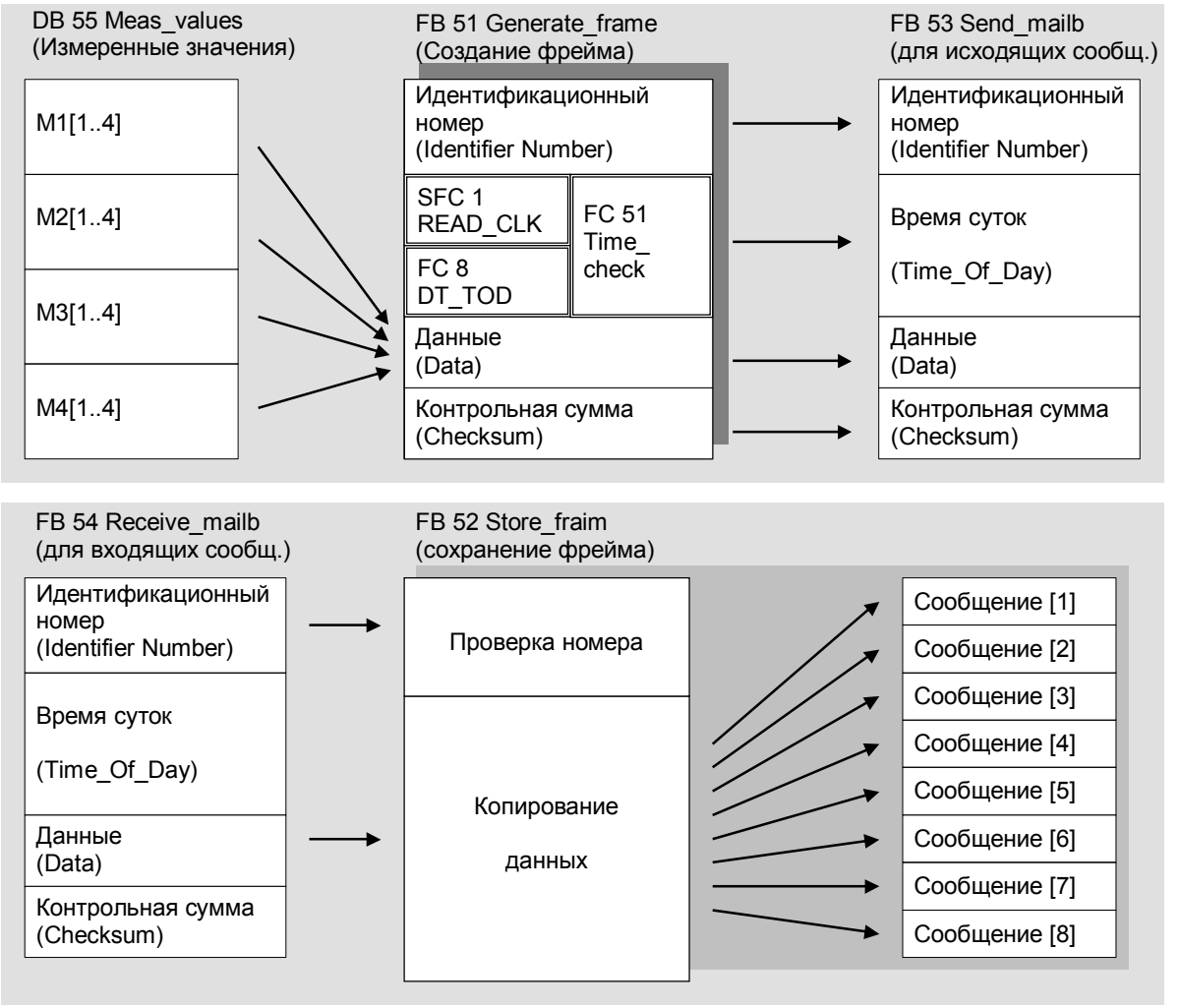


Рис. 30.2 Данные и структура программы для примера работы с фреймом сообщения

# 31 IEC-функции

IEC-функции - это загружаемые функции FC; эти функции поддерживаются в STEP 7. IEC-функции размещаются в стандартной библиотеке *Standard Library* в разделе *IEC Function Blocks (Функциональные блоки IEC стандарта)*. Они являются стандартными функциями в SCL и могут также использоваться в других языках, например, в таких как STL. Вы можете найти обзор всех IEC-функций в Приложениях. Эти функции могут быть разбиты на следующие функциональные группы:

- Функции преобразования
- Функции сравнения для данных типа DATE_AND_TIME
- Функции сравнения для данных типа STRING
- Функции для данных типа STRING
- Функции для данных типа Date/Time-of-day
- Функции для численных данных

Вызов функций представляется в SCL-нотации. Если Вы используете IEC-функции в STL-программе, то функциональные значения имеют имя RET_VAL и являются первым выходным параметром функции.

Пример:

Вызов в SCL:

```
CompResult := EQ_STRING(
        S1 := string1,
        S2 := string2);
```

Вызов в STL:

```
CALL EQ_STRING(
        S1 := string1,
        S2 := string2,
        RET_VAL := CompResult);
```

Некоторые IEC-функции устанавливают двоичный результат BR в качестве сообщения о групповой ошибке. BR может быть проверен в SCL посредством выхода ENO, а в STL непосредственным использованием двоичной проверки или с помощью функции перехода.

## 31.1 Функции преобразования (Conversion Functions)

### Общая информация

Функции преобразования преобразуют типы переменных. Значение, которое необходимо преобразовать, является входным параметром функции, а функциональное значение является результатом преобразования и относится к другому типу данных.

*Общий вид вызова:*

```
var_out := функция_преобразования(var_in);
```

где:

var_in - входной параметр

var_out - выходной параметр (функциональное значение)

Некоторые функции преобразования устанавливают двоичный результат BR слова состояния или выход ENO в состояние FALSE (ЛОЖЬ), если в процессе выполнения преобразования типа данных происходит ошибка. В таком случае преобразование не выполняется.

Пример:

Пусть, целая величина INT, содержащаяся в переменной *Speed*, должна быть преобразована в строку символов, которая должна быть записана в переменную *Display*.

```
Display := I_STRNG(Speed);
IF ENO
   THEN (* преобразование выполнено корректно *);
   ELSE (* произошла ошибка *);
END_IF;
```

Если Вы назначаете функциональное значение типа STRING переменной типа STRING, размещенной во временных локальных данных, то Вы в программе должны присвоить этой переменной определенное значение с заданной длиной (так как предопределение ее значения во временных локальных данных посредством объявления невозможно).

Определенная область (число байтов) резервируется для переменной типа STRING, объявленной в области временных локальных данных. Вы можете установить длину в окне свойств компилятора Compiler Properties. Если Вы не объявляете размера переменной типа STRING, то для длины будет принято значение 254(+2) байта.

### FC 33 S5TI_TIM
### Преобразование типа данных S5TIME в TIME

Функция FC 33 S5TI_TIM выполняет преобразование данных формата S5TIME в данные формата TIME.

Функция FC 33 S5TI_TIM не сообщает об ошибках.

**FC 40 TIM_S5TI**
**Преобразование типа данных TIME в S5TIME**

Функция FC 40 TIM_S5TI выполняет преобразование данных формата TIME в данные формата S5TIME. При преобразовании данных происходит округление данных.

Если входной параметр больше, чем позволяет представить формат S5TIME (то есть значение больше величины TIME#02:46:30.000), то в результате выходной параметр будет иметь значение TIME#999.3, а двоичный результат или выходной параметр ENO получит значение FALSE (ЛОЖЬ).

**FC 16 I_STRNG**
**Преобразование типа данных INT в STRING**

Функция FC 16 I_STRNG выполняет преобразование данных формата INT в данные формата STRING. Результат (строка символов) будет представлен с "лидирующим" знаком (определенное количество цифр плюс знак).

Если строка символов, определенная как функциональное значение, слишком короткая, то преобразование не происходит, а двоичный результат или выходной параметр ENO получает значение FALSE (ЛОЖЬ).

**FC 5 DI_STRNG**
**Преобразование типа данных DINT в STRING**

Функция FC 5 DI_STRNG выполняет преобразование данных формата DINT в данные формата STRING. Результат (строка символов) будет представлен с "лидирующим" знаком (определенное количество цифр плюс знак).

Если строка символов, определенная как функциональное значение, слишком короткая, то преобразование не происходит, а двоичный результат или выходной параметр ENO получает значение FALSE (ЛОЖЬ).

**FC 30 R_STRNG**
**Преобразование типа данных REAL в STRING**

Функция FC 30 R_STRNG выполняет преобразование данных формата REAL в данные формата STRING. Результат (строка символов) будет представлен как строка из 14 разрядов:

±v.nnnnnnnE±xx , где
- ± - знак числа;
- v - число перед десятичной точкой;
- n - один из 7-ми разрядов десятичной дроби после десятичной точки;
- x - один из 2-х разрядов показателя степени с основанием 10 (которую символизирует символ E).

Если строка символов, определенная как функциональное значение, слишком короткая или, если во входном параметре находится значение, не являющееся корректным числом с плавающей запятой, то преобразование не происходит, а двоичный результат или выходной параметр ENO получает значение FALSE (ЛОЖЬ).

**FC 38 STRNG_I**
**Преобразование типа данных STRING в INT**

Функция FC 38 STRNG_I выполняет преобразование данных формата STRING в данные формата INT. Первый символ строки может быть знаком или цифрой, а все последующие символы должны быть цифрами.

Если длина строки символов равна 0 или больше 6, или если строка содержит неразрешенные символы, или если значение после преобразования выходит за пределы допустимых величин для данных формата INT, то преобразование не происходит, а двоичный результат или выходной параметр ENO получает значение FALSE (ЛОЖЬ).

**FC 37 STRNG_DI**
**Преобразование типа данных STRING в DINT**

Функция FC 37 STRNG_DI выполняет преобразование данных формата STRING в данные формата DINT. Первый символ строки может быть знаком или цифрой, а все последующие символы должны быть цифрами.

Если длина строки символов равна 0 или больше 11, или если строка содержит неразрешенные символы, или если значение после преобразования выходит за пределы допустимых величин для данных формата DINT, то преобразование не происходит, а двоичный результат или выходной параметр ENO получает значение FALSE (ЛОЖЬ).

**FC 39 STRNG_R**
**Преобразование типа данных STRING в REAL**

Функция FC 39 STRNG_R выполняет преобразование данных формата STRING в данные формата REAL. Исходная строка символов должна иметь следующий формат:

±v.nnnnnnnE±xx , где
- ± - знак числа;
- v - число перед десятичной точкой;
- n - один из 7-ми разрядов десятичной дроби после десятичной точки;
- x - один из 2-х разрядов показателя степени с основанием 10.

Если исходная строка содержит меньшее число символов, чем 14, или, если эта строка имеет иную структуру, нежели указано выше, значение, или, если значение после преобразования выходит за пределы допустимых величин для данных формата REAL, то преобразование не происходит, а двоичный результат или выходной параметр ENO получает значение FALSE (ЛОЖЬ).

## 31.2 Функции сравнения (Comparison Functions)

Функции сравнения позволяют сравнивать значения двух переменных и выдавать результат сравнения в виде функционального значения. Функциональное значение равно TRUE (ИСТИНА), если результат сравнения положительный, и равно FALSE (ЛОЖЬ), если результат сравнения отрицательный. Функции сравнения не сообщают о возникновении ошибок. Различают функции сравнения для работы с переменными типа DT и для работы с переменными типа STRING.

*Общий вид вызова:*

```
Result := функция_сравнения_DT(
          DT1 := DT_var1,
          DT2 := DT_var2);

Result := функция_сравнения_STRING(
          S1 := STRING_var1,
          S2 := STRING_var2);
```

**FC 9 EQ_DT**
**Сравнение переменных типа DT (проверка на равенство переменных)**

Функция FC 9 EQ_DT выполняет операцию сравнения двух переменных типа DATE_AND_TIME, при этом осуществляется проверка на равенство этих переменных. Функциональное значение равно TRUE (ИСТИНА), только в том случае, если результат сравнения положительный, то есть, если значение параметра DT1 равно значению параметра DT2.

**FC 28 NE_DT**
**Сравнение переменных типа DT (проверка на неравенство переменных)**

Функция FC 28 NE_DT выполняет операцию сравнения двух переменных в формате DATE_AND_TIME, при этом осуществляется проверка на неравенство этих переменных. Функциональное значение равно TRUE (ИСТИНА), только в том случае, если результат сравнения положительный, то есть, если значение параметра DT1 не равно значению параметра DT2.

**FC 14 GT_DT**
**Сравнение переменных типа DT**
**(проверка переменных на отношение по формуле "больше, чем")**

Функция FC 14 GT_DT выполняет операцию сравнения двух переменных в формате DATE_AND_TIME, при этом осуществляется проверка этих переменных на отношение по формуле "больше, чем". Функциональное значение равно TRUE (ИСТИНА), только в том случае, если результат сравнения положительный, то есть, если значение параметра DT1 больше (то есть, имеет более позднее значение времени), чем значение параметра DT2.

**FC 12 GE_DT**
**Сравнение переменных типа DT**
**(проверка переменных на отношение по формуле "больше или равно")**

Функция FC 12 GE_DT выполняет операцию сравнения двух переменных в формате DATE_AND_TIME, при этом осуществляется проверка этих переменных на отношение по формуле "больше или равно". Функциональное значение равно TRUE (ИСТИНА), только в том случае, если результат сравнения положительный, то есть, если значение параметра DT1 больше (то есть, имеет более позднее значение времени), чем значение параметра DT2 или равно этому значению (то есть, если оба параметра содержат одинаковое значение времени).

**FC 23 LT_DT**
**Сравнение переменных типа DT**
**(проверка переменных на отношение по формуле "**мен**ьше, чем")**

Функция FC 23 LT_DT выполняет операцию сравнения двух переменных в формате DATE_AND_TIME, при этом осуществляется проверка этих переменных на отношение по формуле "меньше, чем". Функциональное значение равно TRUE (ИСТИНА), только в том случае, если результат сравнения положительный, то есть, если значение параметра DT1 меньше (то есть, имеет более раннее значение времени), чем значение параметра DT2.

**FC 18 LE_DT**
**Сравнение переменных типа DT**
**(проверка переменных на отношение по формуле "меньше или равно")**

Функция FC 18 LE_DT выполняет операцию сравнения двух переменных в формате DATE_AND_TIME, при этом осуществляется проверка этих переменных на отношение по формуле "меньше или равно". Функциональное значение равно TRUE (ИСТИНА), только в том случае, если результат сравнения положительный, то есть, если значение параметра DT1 меньше (то есть, имеет более раннее значение времени), чем значение параметра DT2 или равно этому значению (то есть, если оба параметра содержат одинаковое значение времени).

**FC 10 EQ_STRNG**
**Сравнение переменных типа STRING (проверка на равенство переменных)**

Функция FC 10 EQ_STRNG выполняет операцию сравнения двух переменных в формате STRING, при этом осуществляется проверка на равенство этих переменных. Функциональное значение равно TRUE (ИСТИНА), только в том случае, если результат сравнения положительный, то есть, если строка символов в параметре S1 идентична строке символов в параметре S2.

**FC 29 NE_STRNG**
**Сравнение переменных типа STRING (проверка на неравенство переменных)**

Функция FC 29 NE_STRNG выполняет операцию сравнения двух переменных в формате STRING, при этом осуществляется проверка на неравенство этих переменных. Функциональное значение равно TRUE (ИСТИНА), только в том случае, если результат сравнения положительный, то есть, если строка символов в параметре S1 отличается от строки символов в параметре S2.

**FC 15 GT_STRNG**
**Сравнение переменных типа STRING**
**(проверка переменных на отношение по формуле "больше, чем")**

Функция FC 15 GT_STRNG выполняет операцию сравнения двух переменных в формате STRING, при этом осуществляется проверка этих переменных на отношение по формуле "больше, чем". Функциональное значение равно TRUE (ИСТИНА), только в том случае, если результат сравнения положительный, то есть, если строка символов в параметре S1 больше, чем строка символов в параметре S2. Переменные сравниваются

в соответствии с их ASCII-кодом (например, имеет ли 'A' большее значение кода, чем 'a' ?), начиная с левого символа строки. Первый отличающийся символ определяет результат операции сравнения. Если, начиная с первого символа, все символы одной строки идентичны соответствующим символам другой строки, то более длинная строка символов принимается, как имеющая большее значение в операции сравнения переменных S1 и S2.

**FC 13 GE_STRNG**
**Сравнение переменных типа STRING**
**(проверка переменных на отношение по формуле "больше или равно")**

Функция FC 13 GE_STRNG выполняет операцию сравнения двух переменных, имеющих формат STRING, при этом осуществляется проверка этих переменных на отношение по формуле "больше или равно". Функциональное значение равно TRUE (ИСТИНА), только в том случае, если результат сравнения положительный, то есть, если строка символов в параметре S1 больше, чем строка символов в параметре S2 или обе строки идентичны. Переменные сравниваются в соответствии с их ASCII-кодом (например, имеет ли 'A' большее значение кода, чем 'a' ?), начиная с левого символа строки. Первый отличающийся символ определяет результат операции сравнения. Если, начиная с первого символа, все символы одной строки идентичны соответствующим символам другой строки, то более длинная строка символов принимается, как имеющая большее значение в операции сравнения переменных S1 и S2.

**FC 24 LT_STRNG**
**Сравнение переменных типа STRING**
**(проверка переменных на отношение по формуле "меньше, чем")**

Функция FC 24 LT_STRNG выполняет операцию сравнения двух переменных в формате STRING, при этом осуществляется проверка этих переменных на отношение по формуле "меньше, чем". Функциональное значение равно TRUE (ИСТИНА), только в том случае, если результат сравнения положительный, то есть, если строка символов в параметре S1 меньше, чем строка символов в параметре S2. Переменные сравниваются в соответствии с их ASCII-кодом (например, имеет ли 'A' меньшее значение кода, чем 'a' ?), начиная с левого символа строки. Первый отличающийся символ определяет результат операции сравнения. Если, начиная с первого символа, все символы одной строки идентичны соответствующим символам другой строки, то менее длинная строка символов принимается, как имеющая меньшее значение в операции сравнения переменных S1 и S2.

**FC 19 LE_STRNG**
**Сравнение переменных типа STRING**
**(проверка переменных на отношение по формуле "меньше или равно")**

Функция FC 19 LE_STRNG выполняет операцию сравнения двух переменных, имеющих формат STRING, при этом осуществляется проверка этих переменных на отношение по формуле "меньше или равно". Функциональное значение равно TRUE (ИСТИНА), только в том случае, если результат сравнения положительный, то есть, если строка символов в параметре S1 меньше, чем строка символов в параметре S2 или обе строки идентичны. Переменные сравниваются в соответствии с

их ASCII-кодом (например, имеет ли 'A' меньшее значение кода, чем 'a' ?), начиная с левого символа строки. Первый отличающийся символ определяет результат операции сравнения. Если, начиная с первого символа, все символы одной строки идентичны соответствующим символам другой строки, то менее длинная строка символов принимается, как имеющая меньшее значение в операции сравнения переменных S1 и S2.

## 31.3 Функции для данных типа STRING (STRING Functions)

Функции для данных типа STRING (STRING Functions) позволяют обрабатывать строки символов. Некоторые функции для данных типа STRING устанавливают двоичный результат BR слова состояния или выход ENO в состояние FALSE (ЛОЖЬ), если при выполнении функции возникает ошибка.

Функции для данных типа STRING (STRING Functions) позволяют проверить фактический параметр на корректность (валидность) (например: проверить, имеет ли параметр блока, использующий переменную типа STRING достаточную длину?). Если Вы объявляете переменную типа STRING в области временных локальных данных и затем используете ее как фактический параметр, то Вы должны сначала в программе присвоить этой переменной некоторое (любое) значение формата STRING с требуемой длиной. Причина этого требования заключается в том, что предопределение значений переменных в области временных локальных данных при объявлении невозможно. То есть, без определения значений для этих переменных их содержимое носит случайный характер, а в случае переменных типа STRING случайными значениями заполнены также байты, несущие информацию о максимальной и текущей длине такой переменной. Указанные байты принимают значащие величины при присвоении переменной определенного значения.

Определенная область (число байтов) резервируется для переменной типа STRING, объявленной в области временных локальных данных. Вы можете установить длину в окне свойств компилятора Compiler Properties. Если Вы не объявляете размера переменной типа STRING, то для длины будет принято значение 254(+2) байта.

**FC 21 LEN**
**Возвращение длины переменной типа STRING**

Вызов функции:

```
int := LEN(string);
```

Функция FC 21 LEN возвращает текущую длину строки символов (число значащих символов) как функциональное значение. Пустая строка имеет нулевую длину. Максимальная длина для строки символов составляет 254 байта.

Функция не выдает сообщений об ошибках.

**FC 11 FIND**
**Поиск в переменной типа STRING**

Вызов функции:

```
int:=FIND(IN1:=string,IN2:=string);
```

Функция FC 11 FIND выполняет поиск позиции подстроки IN2 (второй переменной типа STRING) в строке символов IN1 (в первой переменной типа STRING). Поиск осуществляется с крайнего левого символа строки; при первом обнаружении строки IN2 функция выдает результат операции. Если строка IN1 не содержит строки IN2, то функция возвращает нулевое значение.

Функция не выдает сообщений об ошибках.

**FC 20 LEFT**
**Возвращение указанного числа символов из левой части переменной типа STRING**

Вызов функции:

```
string:=LEFT(IN:=string,L:=int);
```

Функция FC 20 LEFT выполняет возврат указанного числа символов (L) строковой переменной, начиная с 1-го символа слева (то есть, возврат первых L символов). Если число L больше, чем текущая длина строки символов, то функция возвращает входное значение (IN) целиком. Если L = 0 или входное значение IN = 0, то функция возвращает "пустую" строку.

Если L имеет отрицательное значение, то функция также возвращает "пустую" строку, а двоичный результат BR слова состояния или выход ENO получают значение FALSE (ЛОЖЬ).

**FC 32 RIGHT**
**Возвращение указанного числа символов из правой части переменной типа STRING**

Вызов функции:

```
string:=RIGHT(IN:=string,L:=int);
```

Функция FC 32 RIGHT выполняет возврат указанного числа символов (L) строковой переменной, начиная с 1-го символа справа (то есть, возврат последних L символов). Если число L больше, чем текущая длина строки символов, то функция возвращает входное значение (IN) целиком. Если L = 0 или входное значение IN = 0, то функция возвращает "пустую" строку.

Если L имеет отрицательное значение, то функция также возвращает "пустую" строку, а двоичный результат BR слова состояния или выход ENO получают значение FALSE (ЛОЖЬ).

**FC 26 MID**
**Возвращение указанного числа символов из средней части переменной типа STRING**

Вызов функции:

```
string:=MID(IN:=string,L:=int,P:=int);
```

Функция FC 26 MID выполняет возврат указанного числа символов (L) из средней части строковой переменной, начиная с P-го символа слева включительно. Если сумма значений L и P по величине больше, чем текущая длина строки символов, то функция возвращает часть строковой переменной, начиная с P-го символа до конца входного значения IN. целиком. Если L = 0 или входное значение IN = 0, то функция возвращает "пустую" строку.

Во всех остальных случаях (если значение P больше значения L или меньше 1, или если значение P и/или L имеют нулевое или отрицательное значения, то функция также возвращает "пустую" строку, а двоичный результат BR слова состояния или выход ENO получают значение FALSE (ЛОЖЬ).

**FC 2 CONCAT**
**Операция конкатенации (сцепления) двух переменных типа STRING**

Вызов функции:

```
string:=CONCAT(IN1:=string,IN2:=string);
```

Функция FC 2 CONCAT выполняет операцию конкатенации (сцепления) двух переменных типа STRING и создает из них одну строку символов. Если в результате данной операции получается строка символов, имеющая большую длину, чем задано для выходного параметра (RET_VAL), то результирующая строка ограничивается заданным для нее форматом, а двоичный результат BR слова состояния или выход ENO получают значение FALSE (ЛОЖЬ).

**FC 17 INSERT**
**Операция вставки в переменную типа STRING**

Вызов функции:

```
string:=INSERT(IN1:=string,IN2:=string,P:=int);
```

Функция FC 17 INSERT выполняет операцию вставки строки символов из параметра IN2 в строку символов из параметра IN1 после P-го символа. Если параметр P равен 0, то вторая строка *string2* вставляется перед первой строкой *string1*; если параметр P больше, чем текущая длина первой строки символов, то вторая строка вставляется после первой строки.

Если параметр P имеет отрицательное значение, в результате данной операции получается "пустая" строка символов, а двоичный результат BR слова состояния или выход ENO получают значение FALSE (ЛОЖЬ). Двоичный результат BR или выход ENO также получают значение FALSE (ЛОЖЬ), если в результате данной операции получается строка символов, имеющая большую длину, чем задано для выходного параметра; в таком случае результирующая строка ограничивается заданной для нее максимальной длиной.

**FC 4 DELETE**
**Операция удаления части строки из переменной типа STRING**

Вызов функции:

```
string:=DELETE(IN:=string,L:=int,P:=int);
```

Функция FC 4 DELETE выполняет операцию удаления L символов из исходной строки символов, начиная с P-го символа включительно. Если L = 0 и/или P = 0, или если значение P больше, чем текущее значение длины входной строки IN, то функция возвращает исходную строку. Если сумма значений L и P по величине больше, чем текущая длина строки символов, то при выполнении функции удаляется часть входной строковой переменной, начиная с P-го символа до ее конца.

Если значение P и/или L имеют отрицательное значение, то функция также возвращает "пустую" строку, а двоичный результат BR слова состояния или выход ENO получают значение FALSE (ЛОЖЬ).

**FC 31 REPLACE**
**Операция замены части строки в переменной типа STRING**

Вызов функции:

```
string:=REPLACE(IN1:=string,IN2:=string,L:=int,P:=int);
```

Функция FC 31 REPLACE выполняет операцию замены L символов в исходной строке символов (IN1), начиная с P-го символа включительно, второй строкой символов (IN2). Если L = 0, то функцией возвращается исходная строка (IN1). Если P = 0 или P = 1, то замена L символов в исходной строке символов (IN1), начиная с 1-го символа включительно. Если значение P больше, чем текущее значение длины входной строки (IN1), то функция вставляет вторую строку после исходной строки (сцепляет эти строки).

Если значение P и/или L имеют отрицательное значение, то функция также возвращает "пустую" строку, а двоичный результат BR слова состояния или выход ENO получают значение FALSE (ЛОЖЬ). Двоичный результат BR или выход ENO также получают значение FALSE (ЛОЖЬ), если в результате данной операции получается строка символов, имеющая большую длину, чем задано для выходного параметра; в таком случае результирующая строка ограничивается заданной для нее максимальной длиной.

## 31.4 Функции для данных типа Date/Time-of-Day (Date/Time-of-Day Functions)

Функции для данных типа Date/Time-of-Day (Date/Time-of-Day Functions) позволяют обрабатывать переменные типов DATE, TIME-OF-DAY и DATE_AND_TIME.

Некоторые функции для данных типа Date/Time-of-Day устанавливают двоичный результат BR слова состояния или выход ENO в состояние FALSE (ЛОЖЬ), если при выполнении функции возникает ошибка.

**FC 3 D_TOD_DT**
**Объединение переменных форматов DATE и TIME_OF_DAY в переменную, имеющую формат DATE_AND_TIME**

Вызов функции:

```
date_and_time:= D_TOD_DT(IN1:=date,IN2:=time_of_day);
```

Функция FC 3 D_TOD_DT объединяет данные форматов DATE (D#) и TIME_OF_DAY (TOD#) и преобразует их в данные формата DATE_AND_TIME (DT#). Входное значение IN1 принимать значение между границами DATE#1990-01-01 и DATE#2089-12-31.

Функция не сообщает об ошибках.

### FC 6 DT_DATE
### Извлечение значения даты DATE из переменной формата DATE_AND_TIME

Вызов функции:

```
date:=DT_DATE(date_and_time);
```

Функция FC 6 DT_DATE позволяет извлекать данные формата DATE (D#) из данных типа DATE_AND_TIME (DT#). Значение DATE должно находиться между граничными значениями DATE#1990-1-1 и DATE#2089-12-31.

Функция не сообщает об ошибках.

### FC 7 DT_DAY
### Извлечение информации о дне недели из переменной формата DATE_AND_TIME

Вызов функции:

```
int:=DT_DAY(date_and_time);
```

Функция FC 7 DT_DAY позволяет извлекать данные о дне недели из переменной формата DATE_AND_TIME (DT#).

День недели представляется в формате данных INT (от 1 до 7):

1 Воскресенье
2 Понедельник
3 Вторник
4 Среда
5 Четверг
6 Пятница
7 Суббота

Функция не сообщает об ошибках.

### FC 8 DT_TOD
### Извлечение данных о времени суток из переменной формата DATE_AND_TIME

Вызов функции:

```
time_of_day:=DT_TOD(date_and_time);
```

Функция FC 8 DT_TOD позволяет извлекать данные о времени суток (данные формата TIME_OF_DAY (TOD#)) из данных формата DATE_AND_TIME (DT#).

Функция не сообщает об ошибках.

### FC 1 AD_DT_TM
### Добавление временного промежутка к определенному значению времени суток

Вызов функции:

```
date_and_time:=AD_DT_TM(T:=date_and_time,D:=time);
```

Функция FC 1 AD_DT_TM позволяет добавить заданный промежуток времени (значение в формате TIME (T#)) к определенному значению времени суток (значение в формате DATE_AND_TIME (DT#)) и выдает в качестве результата новое значение времени суток (значение в формате DATE_AND_TIME (DT#)). Исходное значение времени суток (параметр T) должно находиться в диапазоне:
от DT#1990-01-01-00:00:00.000 до DT#2089-12-31-23:59:59.999.

Функция не выполняет проверки входного значения.

Если результат сложения не находится внутри указанного выше диапазона, то результат ограничивается соответствующим значением, и при этом двоичный результат BR слова состояния или выход ENO устанавливаются в состояние "FALSE" ("ЛОЖЬ").

### FC 35 SB_DT_TM
### Вычитание временного промежутка из определенного значения времени суток

Вызов функции:

```
date_and_time:=SB_DT_TM(T:=date_and_time,D:=time);
```

Функция FC 35 SB_DT_TM позволяет вычесть заданный промежуток времени (значение в формате TIME (T#)) из определенного значения времени суток (значение в формате DATE_AND_TIME (DT#)) и выдает в качестве результата новое значение времени суток (значение в формате DATE_AND_TIME (DT#)). Исходное значение времени суток (параметр T) должно находиться в диапазоне:
от DT#1990-01-01-00:00:00.000 до DT#2089-12-31-23:59:59.999.

Функция не выполняет проверки входного значения.

Если результат вычитания не находится внутри указанного выше диапазона, то результат ограничивается соответствующим значением, и при этом двоичный результат BR слова состояния или выход ENO устанавливаются в состояние "FALSE" ("ЛОЖЬ").

### FC 34 SB_DT_DT
### Вычитание одного значения времени суток из другого

Вызов функции:

```
date_and_time:=SB_DT_DT(T1:=date_and_time,
T2:=date_and_time);
```

Функция FC 34 SB_DT_DT позволяет выполнить операцию вычитания одного значения времени суток из другого (оба значения имеют формат DATE_AND_TIME (DT#)). Функция выдает в качестве результата значение интервала времени (значение в формате TIME (T#)).

Значения времени должны находиться в диапазоне:
от DT#1990-01-01-00:00:00.000 до DT#2089-12-31-23:59:59.999.

Функция не выполняет проверки входного значения.

Если первое значение времени (параметр T1) больше (имеет более позднее значение), чем второе значение (параметр T2), то результат положителен; если первое значение времени меньше (имеет более раннее значение), чем второе значение, то результат отрицателен.

Если результат вычитания не находится внутри указанного выше диапазона для данных формата TIME, то результат ограничивается соответствующим значением, и при этом двоичный результат BR слова состояния или выход ENO устанавливаются в состояние "FALSE" ("ЛОЖЬ").

## 31.5 Функции для обработки численных данных (Numerical Functions)

Функции для обработки численных данных (Numerical Functions) сохраняют неизменным численное значение и при этом двоичный результат BR слова состояния или выход ENO устанавливаются в состояние "FALSE" ("ЛОЖЬ"), если:

- параметризированная переменная относится к недопустимому типу данных;
- параметризированная переменная относится к другому, а не к ожидаемому, типу данных;
- переменная формата REAL представлена некорректным числом (ожидается формат числа с плавающей запятой и значение числа должно принадлежать допустимому диапазону).

**FC 22 LIMIT**
**Ограничитель (Delimiter)**

Вызов функции:

```
any_num:=LIMIT(MN:=any_num,IN:=any_num,MX:=any_num);
```

Функция FC 22 ограничивает числовое значение переменной IN задаваемыми граничными значениями: MN и MX. В качестве входных значений допускаются переменные типов INT, DINT и REAL. Все входные значения (фактические параметры) должны принадлежать к одному и тому же типу данных.

Нижнее граничное значение (параметр MN) должно быть меньше, чем верхнее граничное значение (параметр MX).

Функция сообщает о возникновении ошибки, если выполняется любое из вышеуказанных условий возникновения ошибки, а также, если: нижнее граничное значение (параметр MN) больше, чем верхнее граничное значение (параметр MX).

**FC 25 MAX**
**Выбор максимального числа из трех исходных**

Вызов функции:

```
any_num:=MAX(IN1:=any_num,IN2:=any_num,IN3:=any_num);
```

Функция FC 25 MAX позволяет выбрать наибольшее из числовых значений трех входных переменных. В качестве входных значений допускаются значения переменных, принадлежащих к типам данных INT, DINT и REAL. Все входные значения (фактические параметры) должны принадлежать к одному и тому же типу данных.

**FC 27 MIN**
**Выбор минимального числа из трех исходных**

Вызов функции:

```
any_num:=MIN(IN1:=any_num,IN2:=any_num,IN3:=any_num);
```

Функция FC 27 позволяет выбрать наименьшее из числовых значений трех входных переменных. В качестве входных значений допускаются значения переменных, принадлежащих к типам данных INT, DINT и REAL. Все входные значения (фактические параметры) должны принадлежать к одному и тому же типу данных.

**FC 36 SEL**
**"Двоичный переключатель" (Binary selection)**

Вызов функции:

```
any:=SEL(G:=bool,IN0:=any,IN1:=any);
```

Функция FC 36 SEL позволяет выбрать одно из двух значений переменных (IN0 и IN1) в зависимости от состояния переключателя (параметр G). В качестве входных значений IN0 и IN1 допускаются переменные, принадлежащие к любым простым типам (кроме переменных типа BOOL). Обе входные переменные и выходная переменная должны принадлежать к одному типу данных.