Обработка переменных

Данный раздел книги рассматривает вопросы обработки сложных переменных. Для обработки сложных переменных требуются в первую очередь знание структуры данных различных типов, умение использовать косвенную адресацию и способность определять адреса переменных во время выполнения программы.

Доступ к переменным простых **типов данных (data types)** возможен непосредственно с помощью STL-операторов, если Вы имеете дело с двоичными логическими операциями, с операциями с памятью (memory functions) или с операциями загрузки (load) или пересылки (transfer). Для сложных типов данных и пользовательских типов данных прямой доступ в настоящее время возможен только к их отдельным компонентам. И если Вам все-таки необходимо организовать доступ к этим типам данных, то Вы должны знать внутреннюю структуру таких переменных.

Косвенная адресация (indirect addressing) предоставляет доступ к операндам, адреса которых заранее неизвестны (до момента выполнения программы). Пользователь может выбирать между косвенной адресацией посредством памяти (memory-indirect addressing) и косвенной адресацией посредством регистра (register-indirect addressing). Вы можете дождаться выполнения программы, для использования области операнда. Косвенная адресация позволяет Вам получать доступ к переменным сложных типов и пользовательских типов, используя абсолютную адресацию.

При использовании **прямого доступа к переменным (direct variable access)** используется текущий адрес локальной переменной. Если Вы определили адреса, Вы можете обработать соответствующие локальные переменные (в том числе и параметры блока), относящиеся к любому типу данных. В двух предыдущих главах содержится информация, требуемая для этих целей.

Несколько развернутых примеров собраны в разделе 26.4 "Краткое описание примера фрейма сообщения". В данном разделе объясняется, как обрабатываются сложные переменные. Примеры "Данные фрейма сообщения" ("Message Frame Data"), "Подготовка фрейма сообщения" ("Preparing a Message Frame"), и "Контроль времени" ("Clock Check") имеют дело с данными пользовательского типа и применяют переменные сложного типа с использованием системных и стандартных функций. В примерах "Контрольная сумма" ("Checksum") и "Преобразование данных" ("Data Item Conversion") описывается, как обращаться к параметрам, относящимся к сложным типам данных, с помощью косвенной адресации.

В примере "Сохранение фрейма сообщения" ("Save Message Frame") показано, как использовать системную функцию SFC 20 BLKMOV для пересылки данных из области, адрес которой неизвестен до начала выполнения программы.

24 Типы данных

Простые, сложные и пользовательские типы данных; объявление и структура разных типов данных.

25 Косвенная адресация

Указатели на область, указатели на DB, ANY-указатели; косвенная адресация посредством памяти (memory-indirect addressing) и косвенная адресация посредством регистра (register-indirect addressing); внутризонная (area-internal) адресация и межзонная (area-crossing) адресация; использование адресных регистров.

26 Прямой доступ к переменным

Адресация локальных переменных; сохранение переменных;. сохранение данных при передаче параметров; "переменная" ANY-указатель; пример фрейма сообщения.

24 Типы данных

Тип данных определяет их свойства и характеристики, особое представление содержимого одного или больше, чем одного связанных адресов и допустимых областей. STEP 7 предоставляет пользователю возможность предопределения данных, которые он может компилировать, как и определенные им пользовательские типы данных. Типы данных доступны везде. Они могут использоваться в любом блоке.

В разделе 3.7 "Переменные и константы" представлен обзор всех типов данных и соответствующее представление констант.

Данная глава содержит детальную информацию о простых и сложных типах данных; в главе показана структура соответствующих переменных. Вы узнаете, как создаются и используются пользовательские типы данных.

Вы можете найти примеры для типов данных на дискете, прилагаемой к данной книге, в разделе "Variable Handling" ("Обработка переменных") в функциональных блоках FB 101, FB 102 и FB 103 или в исходном файле Chap_24.

24.1 Простые типы данных

Переменные простых типов имеют максимальную длину, равную одному двойному слову; поэтому они могут быть обработаны с помощью функций загрузки (load) и пересылки (transfer) или с помощью двоичных логических операций.

24.1.1 Объявление простых типов данных

Данные простого типа могут занимать один бит, один байт, одно слово и одно двойное слово.

Объявление (Declaration)

```
varname : datatype := pre-assignment;
```

varname - имя переменной

datatype - простой тип данных

pre-assignment - (предопределенное) фиксированное значение переменной

Идентификаторы типов данных (например, BOOL, REAL) являются ключевыми словами; они могут быть записаны также и в нижнем регистре. Переменная простого типа данных может быть объявлена в таблице символов глобальной или локальной переменной в разделе объявлений.

Предопределение (Pre-assignment)

Переменная может быть предопределена во время объявления (в отличие от параметров функции или временных переменных). Значение предопределения должно относиться к тому же типу данных, что и сама переменная.

Применение

Вы можете применять переменные простых типов в качестве фактических параметров для соответствующим образом объявленных параметров блока (того же типа данных POINTER или ANY) или Вы можете организовать к ним доступ посредством обычных STL-операторов (например, двоичная проверка, функции загрузки [load]).

Сохранение переменных

Переменные простых типов сохраняются таким же образом, как соответствующие адреса. Для этого допустимо использовать все адресные области, включая параметры блоков.

24.1.2 Типы данных BOOL, BYTE, WORD, DWORD, CHAR

Переменная типа BOOL представляет собой значение бита (например, входа I 1.0). Переменные типов данных BYTE, WORD и DWORD представляют собой последовательности битов, состоящие соответственно из 8, 16 и 32 битов; при этом отдельные биты не проверяются. В главе 3 "SIMATIC S7" рассматриваются возможные представления как констант.

Специальной формой этих типов данных являются BCD-числа (числа в двоично-десятичном коде) и значения счетчиков, которые используются в сочетании с собственно функцией счетчика, а также тип данных CHAR, представляющий символ в коде ASCII (см. рис. 24.1).

BCD-числа (числа в двоично-десятичном коде)

BCD-числа не имеют специального идентификатора в STL. Вы можете вводить BCD-число с типом данных 16# (шестнадцатеричное), используя только цифры из ряда 0 … 9.

BCD-числа встречаются при загрузке значений таймеров и счетчиков и при работе с функциями преобразования. Тип данных S5TIME# может использоваться для задания значения таймера при запуске функции таймера (см. ниже), а для определения значения счетчика используется тип данных: 16# или C#. Значение счетчика с типом данных C# - это BCD-число из диапазона 000 … 999, в котором знак всегда равен 0.

В общем случае BCD-числа - это беззнаковые числа. При использовании в функции преобразования знак BCD-числа хранится в крайней левой (старшей) тетраде. Это ведет к потере одной тетрады BCD-числа для его диапазона.

В случае, если BCD-число хранится в шестнадцатиразрядном слове, знак хранится в старшей тетраде в бите 15. При этом, если состояние сигнала данного бита равно "0", то знак числа положительный, если состояние сигнала равно "1", то знак - отрицательный. Знак не влияет на значения отдельных тетрад. Аналогичное назначение применяется для 32-разрядного слова.

Диапазон числа составляет от 0 до 999 для 32-разрядных BCD-чисел.

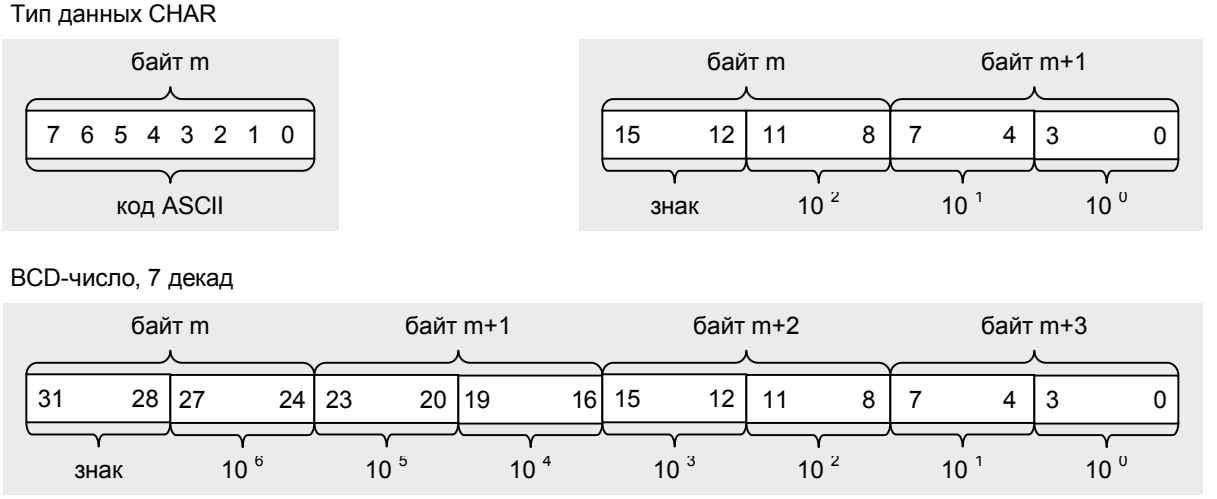


Рис. 24.1 Представление BCD-чисел (чисел в двоично-десятичном коде) и данных типа CHAR

CHAR

Переменная типа CHAR (character - символ) занимает один байт. Тип данных CHAR представляет отдельный символ в формате ASCII. Например, символ 'A'. Вы можете также записать любой печатаемый символ в одинарных кавычках.

В операторах загрузки (load) в языке программирования STL используются также некоторые специальные символы, показанные в таблице 24.1. Например, оператор L '$$' загружает знак доллара в коде ASCII.

Кроме того, Вы можете использовать другие специальные формы типа данных CHAR, когда загружаете в аккумулятор символы в коде ASCII. Например, оператор L 'a' загружает один символ (в данном случае символ a) с выравниванием вправо в аккумуляторе; оператор L 'aa' загружает два символа, а оператор L 'aaaa' соответственно загружает четыре символа.

Таблица 24.1 Специальные символы для переменной типа CHAR

Символ CHAR	Hex	Значение
$$	24_{hex}	Символ доллара
$'	27_{hex}	Одинарная кавычка
$L или $l	$0A_{hex}$	Новая строка
$P или $p	$0C_{hex}$	Новая страница
$R или $r	$0D_{hex}$	Возврат каретки
$T или $t	09_{hex}	Табулятор

24.1.3 Представление чисел

В данном разделе совместно рассматриваются типы данных INT, DINT и REAL. На рис. 24.2 представлено назначение битов для этих типов данных.

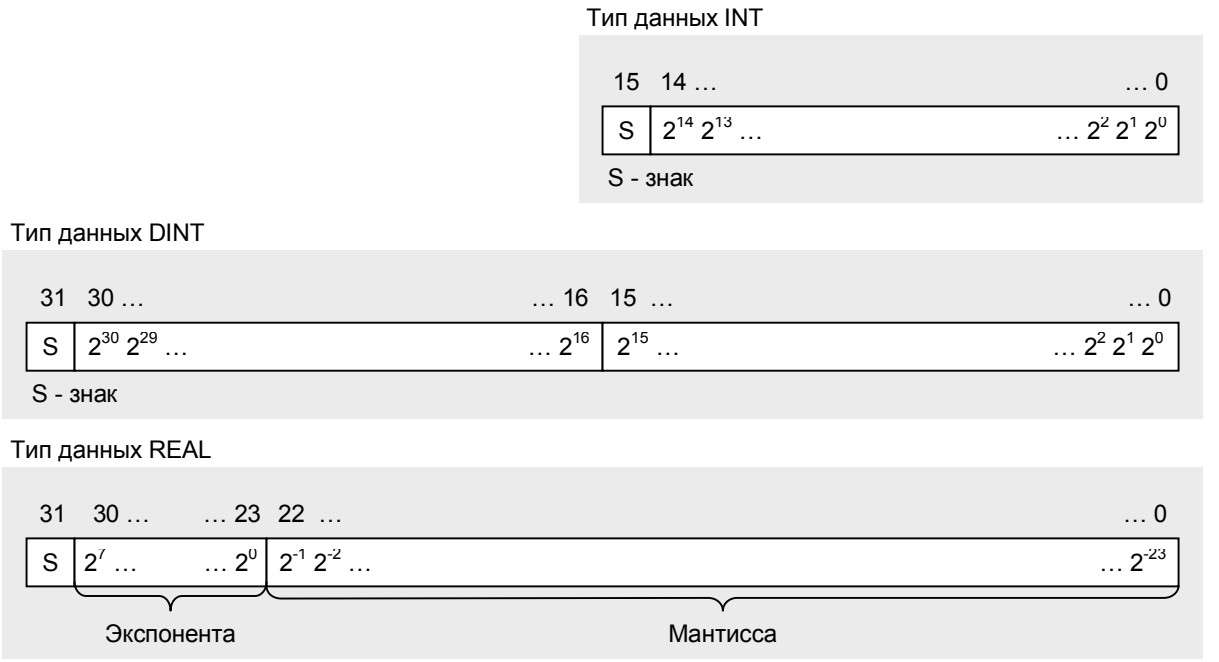


Рис. 24.2 Назначение битов для данных типов INT, DINT и REAL

INT

Переменная типа INT представляет собой целое (целое число), которое сохраняется как 16-разрядное число с фиксированной запятой. Тип данных INT не имеет специального идентификатора.

Переменная типа данных INT занимает одно слово (word).

Состояния сигналов от бита 0 до бита 14 определяют модуль числа; состояние бита 15 определяет знак числа (S). Если состояние сигнала данного бита равно "0", то число положительное, если состояние сигнала равно "1", то число отрицательное. Отрицательные числа представляются как инвертированные числа (two's complement).

Диапазон INT-числа составляет:

от + 32 767 ($7FFF_{hex}$)

до - 32 768 (8000_{hex}).

DINT

Переменная типа DINT представляет собой целое число, которое сохраняется как 32-разрядное число с фиксированной запятой. Целое число хранится в формате DINT, если оно больше +32767 или меньше, чем -32768 или, если перед этим числом стоит специальный идентификатор L#.

Переменная типа данных DINT занимает одно двойное слово (duobleword). Состояния сигналов от бита 0 до бита 30 определяют модуль числа; состояние бита 31 определяет знак числа. Если состояние сигнала данного бита равно "0", то число положительное, если состояние сигнала равно "1", то число отрицательное. Отрицательные числа представляются как инвертированные числа (two's complement).

Диапазон DINT-числа составляет:

от + 2 147 483 647 ($7FFF\ FFFF_{hex}$)

до - 2 147 483 648 ($8000\ 0000_{hex}$).

Пример для STL: с помощью оператора L -100 Вы загружаете в аккумулятор целое INT-число, а с помощью оператора L# -100 Вы загружаете в аккумулятор целое DINT-число. Для данных двух случаев различие заключается в назначении битов левого слова аккумулятора: в случае с INT-числом левое слово аккумулятора содержит значение 0000_{hex}, а в примере с DINT-числом левое слово аккумулятора содержит значение $FFFF_{hex}$.

Пример для SCL: если Вы определили значение константы как -100, то в случаях, когда предстоит выполнить операцию с данным числом и переменной DINT-формата, редактор автоматически преобразует данное INT-число в DINT-формат ("неявное" преобразование типа).

REAL

Переменная типа REAL представляет собой дробное число, которое сохраняется как 32-разрядное число с плавающей запятой. Целое число сохраняется в формате REAL, если в нем за десятичной точкой записан ноль.

Пример для STL: в то время как 100 и L#100 указывает на целое число соответственно в INT- и в DINT-формате, в REAL формате Вы должны записать число 100 либо как 100.0, либо как 1.0e+2 (спецификация с десятичной точкой, в первом случае без экспоненты, во втором случае с экспонентой).

Пример для SCL: в формате REAL Вы можете определить значение константы в любом численном представлении. Значение 100, например, в случаях, когда предстоит выполнить операцию с данным числом и переменной REAL-формата, редактор автоматически преобразует данное число в REAL-формат ("неявное" преобразование типа).

В экспоненциальном представлении Вы можете определить целое или дробное число с помощью 7 значащих цифр со знаком с последующим символом "e" или "E". Вслед за символом "e" или "E" пишется значение показателя степени по основанию 10 ("экспонента"). Преобразование числа REAL-формата во внутреннее представление числа с плавающей запятой выполняется системой STEP 7.

Для REAL-чисел существует различие между числами, которые могут быть представлены с общей точностью ("нормированные" или "нормализованные" ("normalized") числа с плавающей запятой) и числами, которые представляются как числа с ограниченной точностью ("ненормированные" или "денормализованные" ("denormalized") числа с плавающей запятой). Диапазон значений нормированных чисел с плавающей запятой составляет:

от $-3.402\,823 \cdot 10^{+38}$ до $-1.175\,494 \cdot 10^{-38}$

и

от $+1.175\,494 \cdot 10^{-38}$ до $+3.402\,823 \cdot 10^{+38}$.

Диапазон значений ненормированных чисел с плавающей запятой составляет:

от $-1.175\,494 \cdot 10^{-38}$ до $-1.401\,298 \cdot 10^{-45}$

и

от $+1.401\,298 \cdot 10^{-45}$ до $+1.175\,494 \cdot 10^{-38}$.

S7-300 CPU не могут выполнять вычисления с ненормированными числами с плавающей запятой. Битовая структура в этом случае такова, что ненормированное число с плавающей запятой представляется как ноль. Если результат вычисления попадает в диапазон ненормированных чисел с плавающей запятой, то значение этого результата представляется нулевым с установкой битов состояния OV и OS ("нарушение численного диапазона").

CPU выполняет вычисления с полной точностью представления чисел с плавающей запятой. Из-за ошибок округления при преобразовании результаты, отображаемые программатором, могут отличаться от теоретически точного представления результата.

Переменная типа REAL внутренне состоит из трех компонентов:
- знак,
- 8-разрядная экспонента по основанию 2,
- 32-разрядная мантисса.

Знак может принимать значения "0" (положительное число) и "1" (отрицательное число). Экспонента сохраняется как константа, увеличенная на 1 (смещена на +127), так что она имеет диапазон значений от 0 до 255. Мантисса представляет собой дробную часть. Целая часть мантиссы (integer component) не хранится, так как она или всегда равна 1 (в случае нормированных чисел с плавающей запятой), или всегда равна 0 (в случае ненормированных чисел с плавающей

запятой). В таблице 24.2 показаны внутренние границы диапазона числа с плавающей запятой.

Таблица 24.2 Внутренние границы диапазона числа с плавающей запятой

Знак	Экспонента	Мантисса	Значение
0	255	не равно 0	Некорректное число с плавающей запятой (не число)
0	255	0	+ бесконечность
0	1…254	любое значение	Положительное нормированное число с плавающей запятой
0	0	не равно 0	Положительное ненормированное число с плавающей запятой
0	0	0	+ ноль
1	0	0	- ноль
1	0	не равно 0	Положительное ненормированное число с плавающей запятой
1	1…254	любое значение	Положительное нормированное число с плавающей запятой
1	255	0	- бесконечность
1	255	не равно 0	Некорректное число с плавающей запятой (не число)

24.1.4 Представление времени

В данном разделе совместно рассматриваются типы данных S5TIME, DATE, TIME и TIME_OF_DAY. На рис. 24.3 представлено назначение битов для этих типов данных.

Один тип данных TIME_OF_DAY в рассматриваемой категории типов принадлежит также к сложным типам данных, так как для его представления требуется 8 байтов.

S5TIME

Переменные типа S5TIME используются для инициализации таймера из SIMATIC-функций для базовых языков программирования STL, LAD и FBD (язык SCL использует для этих целей представление данных типа TIME). Тип данных S5TIME занимает 16-разрядное слово со структурой "1+3 тетрады" (см. рис. 24.3).

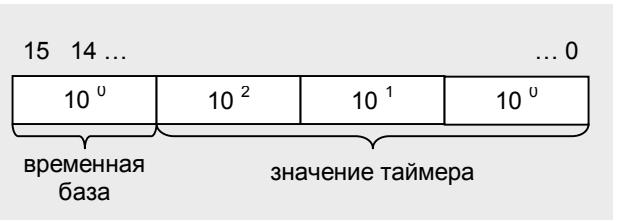
Время задается в часах, минутах, секундах и миллисекундах. Преобразование числа S5TIME-формата во внутреннее представление выполняется системой STEP 7. Внутренне число представляется как BCD-число (число в двоично-десятичном коде) из диапазона от 000 до 999. Временная база (time base) может принимать следующие значения: 10 мс (0000), 100 мс (0001), 1 с (0010), 10 с (0011). Время определяется как результат произведения временной базы на значение таймера.

Примеры:

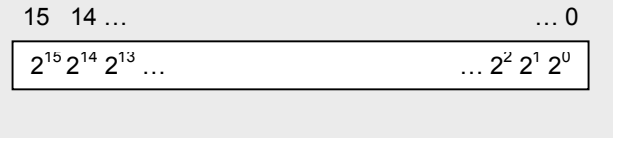
S5TIME#500ms (= 0050_{hex})

S5T#2h46m30s (= 3999_{hex})

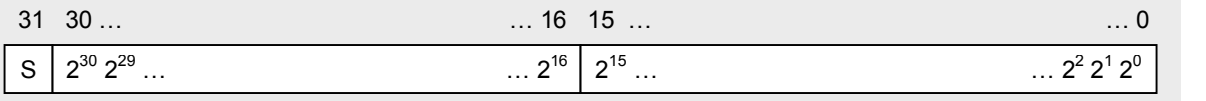
Тип данных S5TIME



Тип данных DATE



Тип данных TIME



S - знак

Тип данных TIME_OF_DAY

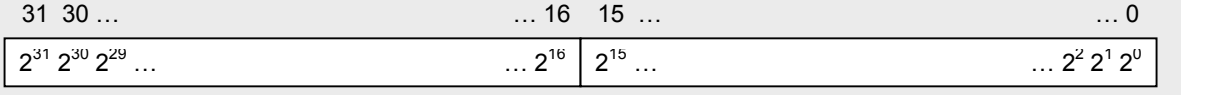


Рис. 24.3 Назначение битов для типов S5TIME, DATE, TIME и TIME_OF_DAY

DATE

Переменная типа DATE хранится в машинном слове как беззнаковое число с фиксированной запятой. Содержание переменной соответствует числу дней, начиная с 01.01.1990 г. Представление числа содержит год, месяц и день, разделенные тире.

Примеры:

DATE#1990-01-01 (= 0000_{hex})

D#2168-12-31 (= $FF62_{hex}$)

TIME

Переменная типа TIME занимает одно двойное слово. Представление числа содержит спецификации дня (d), часов (h), минут (m), секунд (s) и

миллисекунд (ms); при этом отдельные спецификации могут быть пропущены. Содержание переменной интерпретируется как число миллисекунд и сохраняется в 32-разрядном числе со знаком с фиксированной запятой.

Примеры:

TIME#24d20h31m23s647ms (= 7FFF FFFF$_{hex}$)

TIME#0ms (= 0000 0000$_{hex}$)

T#-24d20h31m23s647ms (= 8000 0000$_{hex}$)

В SCL данное представление используется для выражения длительности в функциях SIMATIC-таймера. Редактор автоматически преобразует число, заданное в формате TIME, в число формата S5TIME (структура 1+3 тетрады) с округлением вниз, если это необходимо.

"Десятичное" представление для данных типа TIME также возможно, например, TIME#2.25h или T#2.25h. Такое представление в SCL допустимо только для положительных значений.

Примеры:

TIME#0.0h (= 0000 0000$_{hex}$)

TIME#24.855134d (= 7FFF FFFF$_{hex}$)

TIME_OF_DAY

Переменная типа TIME_OF_DAY занимает одно двойное слово. Содержание переменной интерпретируется как число миллисекунд (ms), начиная с начала дня (0.00 часов) и сохраняется в 32-разрядном беззнаковом числе с фиксированной запятой. Спецификация числа состоит из часов, минут и секунд, разделенных двоеточием. В спецификации возможно указывать число миллисекунд с записью после числа секунд и отделением от этого числа точкой. Значение для миллисекунд может быть опущено.

Примеры:

TIME_OF_DAY#00:00:00 (= 0000 0000$_{hex}$)

TOD#23:59:59.999 (= 0526 5BFF$_{hex}$)

24.2 Сложные типы данных

К сложным типам данных относятся такие данные, которые целиком не могут быть непосредственно обработаны с помощью STL-операторов, но допускаются в SCL-выражениях. Система STEP 7 определяет следующие четыре сложных типа данных:

- DATA_AND_TIME
  дата и время суток (в двоично-десятичном или в BCD-коде)
- STRING
  символьная строка, в которой может быть до 254 символов

- ARRAY
  массив ("field" - "поле", комбинация переменных одного типа)
- STRUCT
  структура (комбинация переменных разного типа)

Данные типов STRING, ARRAY и STRUCT должны быть заранее определены (предопределены) пользователем - для типа STRING задается длина символьной строки, а для типов ARRAY и STRUCT определяются состав и размеры данных-компонентов.

Вы можете объявить переменные сложных типов только в блоках глобальных данных, в экземплярных блоках данных как временные локальные данные или как параметры блока.

24.2.1 Тип данных DATA_AND_TIME

Данные типа DATA_AND_TIME заключают в себе информацию о дате и времени суток. Вместо идентификатора DATA_AND_TIME Вы можете использовать сокращение DT.

Объявление (Declaration)

varname : DATA_AND_TIME *:= pre-assignment;* *или* *varname :* DT *:= pre-assignment;*

varname - имя переменной

pre-assignment - предопределенное значение переменной

DATA_AND_TIME или DT - ключевые слова; они могут быть записаны и в нижнем регистре.

Предопределение (Pre-assignment)

Переменная может быть предопределена на этапе объявления (не как параметр блока в функции, как входной или выходной параметр в функциональном блоке или как временная переменная). Значение предопределения должно относиться к тому же типу данных, что и сама переменная. Значение предопределения должно иметь следующий формат:

DATA_AND_TIME#год-месяц-день-часы:минуты:секунды.миллисекунды
или
DT#год-месяц-день-часы:минуты:секунды.миллисекунды

Спецификация миллисекунд может быть опущена (см. табл. 24.3).

Применение

Переменные типа DT Вы можете применять в качестве соответствующим образом объявленных параметров блока (того же типа данных - DT или ANY); например, они могут быть скопированы с помощью системной функции SFC 20 BLKMOV. Для обработки этих переменных используются

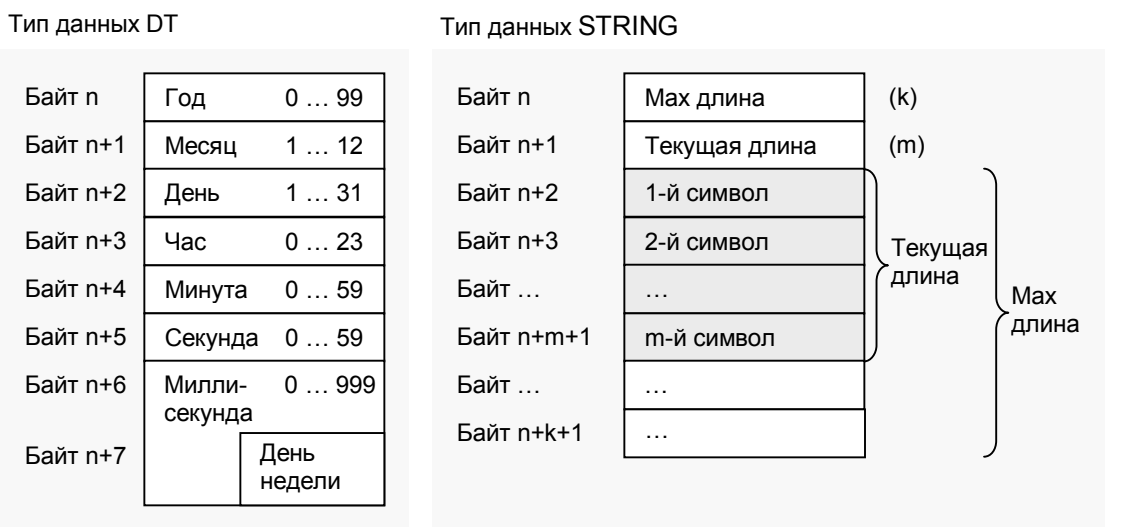
стандартные функциональные блоки ("IEC-functions" - "IEC-функции").

Структура переменных

Переменные типа DATA_AND_TIME или DT занимают 8 байтов (см. рис 24.4). Переменные начинаются на границе слов (начиная с байта с четным адресом). Все спецификации доступны в BCD-формате (в формате двоично-десятичного числа).

Таблица 24.3 Примеры объявления переменных типов DT и STRING

Имя	Тип данных	Начальное значение	Комментарий
Date1	DT	DT#1990-01-01-00:00:00	Минимальное значение переменной типа DT
Date2	DATA_AND_TIME	DATA_AND_TIME# 2089-12-31-23:59:59.999	Максимальное значение переменной типа DT
First_name	STRING[10]	'Jack'	Переменная типа STRING, заняты 4 символа из 10
Last_name	STRING[7]	'Daniels'	Переменная типа STRING, заняты все 7 символов из 7
NewLine	STRING[2]	'RL'	Переменная типа STRING, занята специальными символами
EmptyString	STRING[16]	' '	Переменная типа STRING, без заполнения



День недели: 1 = Воскресенье
7 = Суббота

Рис. 24.4 Структура переменных типов DT и STRING

24.2.2 Тип данных STRING

Данные типа STRING - символьная строка, в которой может быть до 254 символов.

Объявление (Declaration)

varname : STRING[Max число] *:= pre-assignment;*

varname - имя переменной

pre-assignment - предопределенное значение переменной

STRING - ключевое слово; оно может быть записано также в нижнем регистре.

Max число - параметр, определяющий максимальное число символов (длину строки), которые могут быть записаны в данную переменную. Таким образом, такая переменная может содержать от 0 до 254 символов. Данный параметр при записи также может быть опущен; при этом редактор будет использовать для такой переменной стандартную (максимально возможную) длину, равную 254 байта. При использовании функций FC редактор не допускает спецификацию параметра "Max число", и он будет использовать для такой переменной стандартную длину, равную 254 байта.

Предопределение (Pre-assignment)

Переменная может быть предопределена на этапе объявления (не как параметр блока в функции, как входной или выходной параметр в функциональном блоке или как временная переменная). Значение предопределения задается символами в ASCII-коде, заключенными в одинарные кавычки, или с предшествующим знаком доллара, в случае применения особых символов (см. табл. 24.3).

Если значение предопределения короче, чем объявленная максимальная длина, то неиспользованные позиции в поле переменной не занимаются. При дальнейшей обработке такой STRING-переменной учитываются только символы, занимающие позиции в поле переменной. Как значение предопределения допускается "пустая строка" ("emptystring").

Применение

Переменные типа STRING Вы можете применять в качестве параметров блоков типа STRING или ANY; например, они могут быть скопированы с помощью системной функции SFC 20 BLKMOV. Для обработки этих переменных используются стандартные функциональные блоки ("IEC-functions" - "IEC-функции").

Информацию об особенностях применения этих переменных в SCL Вы можете найти в разделе 27.5.2 "Назначения для переменных типов DT и STRING".

Структура переменных

Переменная типа STRING (строка символов) имеет максимальную длину 256 символов, из которых 254 байтов могут использоваться собственно для хранения данных. Переменные типа STRING начинаются на границе слов (начинаются с байта с четным адресом).

При применении STRING-переменных пользователь определяет для них максимальную длину. "Текущая" длина (то есть, фактически занимаемая строкой символов, равная числу "непустых" символов) определяется при

предопределении STRING-переменной или при ее обработке. В первом байте переменной содержится значение максимальной (заданной при объявлении переменной) длины; во втором байте содержится значение текущей длины переменной. Начиная с 3-го по счету байта располагаются данные - строка символов в формате ASCII (см. рис 24.4).

24.2.3 Тип данных ARRAY

Данные типа ARRAY представляет собой массив (или "поле" - "field"), содержащий в себе комбинацию фиксированного числа переменных одинакового типа.

Объявление (Declaration)

fieldname : ARRAY[*minIndex..maxIndex*] OF *datatype := pre-assignment;*
или
fieldname : ARRAY[$minIndex_1$ ..$maxIndex_1$,...,$minIndex_6$ ..$maxIndex_6$,]
OF *datatype := pre-assignment;*

ARRAY и OF - ключевые слова; они могут быть записаны также и в нижнем регистре.

fieldname - имя переменной типа ARRAY массив (поле);

pre-assignment - предопределенное значение переменной;

minIndex - нижний предел массива (поля);

maxIndex - верхний предел массива (поля);

Указанные пределы являются целыми числами типа INT из диапазона: -32768 … +32767; значение *maxIndex* должно быть больше или равно значению *minIndex.* Массив может быть многомерным. При этом каждая размерность должна также определяться своими нижним и верхним значениями. Максимальное число размерностей для массива ARRAY составляет 6:

$minIndex_1..maxIndex_1,.., minIndex_6..maxIndex_6$

datatype - параметр, определяющий тип данных для элементов массива; в рассматриваемом контексте это может быть любой тип данных, даже пользовательский, исключая собственно только тип ARRAY.

Предопределение (Pre-assignment)

Пользователь может предопределить на этапе объявления отдельные элементы массива (в отличие от параметров функции, проходных параметров в функциональном блоке или временных переменных). Тип данных предопределения должен соответствовать типу, заявленному для переменной.

Нет необходимости определять все элементы массива; если число предопределяемых элементов меньше числа элементов массива, то только первые элементы получают значения. Число предопределяемых элементов не может быть больше числа элементов массива. Значения

предопределения должны разделяться запятыми. Многократное назначение одного и того же значения может быть выполнено с помощью взятия значения в скобки и указания перед скобками коэффициента повторения (см. табл. 24.4).

Таблица 24.4 Примеры объявления переменных типа ARRAY

Имя	Тип данных	Начальное значение	Комментарий
Meas_Val	ARRAY[1..24]	0.4, 1.5, 11 (2.6, 3.0)	Переменная типа ARRAY с 24 элементами типа REAL
	REAL		
Time_Of_Day	ARRAY[-10..10]	21 (TOD#08:30:00)	Массив переменных TOD с 21 элементом
	TIME_OF_DAY		
Result	ARRAY[1..24,1..4]	96 (L#-0)	Двумерный массив с 96 элементами
	DINT		
Character	ARRAY[1..2,3..4]	2 ('a'), 2 ('b')	Двумерный массив с 4 элементами
	CHAR		

Применение

Переменные типа ARRAY Вы можете применять в качестве параметров блоков типа ARRAY с такой же структурой или как параметры блоков типа ANY. Например, они могут быть скопированы с помощью системной функции SFC 20 BLKMOV. Вы можете также определить отдельные элементы массива в качестве параметров блока типа ARRAY, если параметры блока относятся к тому же типу данных, что и эти элементы.

Если отдельные элементы массива относятся к простым типам данных, то Вы можете обрабатывать их с помощью обычных STL-операторов.

Доступ к элементу массива обеспечивается по имени массива и индексу в квадратных скобках. Индекс имеет фиксированное значение в STL и не может быть изменен во время выполнения программы (не может быть переменной).

Индекс в SCL может быть переменной или выражением с данными типа INT; значение индекса может быть изменено во время выполнения программы.

Многомерные массивы

Массив может быть многомерным. Максимальное число размерностей для массива ARRAY составляет 6. Многомерный массив подобен одномерному. Во время объявления переменных диапазоны индексов по каждой размерности отделяются друг от друга запятыми и заключаются в квадратные скобки.

Обеспечивая доступ к элементам многомерного массива в STL, Вы всегда должны указывать индексы всех размерностей массива. В SCL возможно организовывать доступ к части массива (см. раздел 27.5.4 "Назначение массивов").

Структура переменных

Переменные типа ARRAY всегда начинаются на границе слов (т.е., начинаются с байта с четным адресом). Переменная типа ARRAY занимает в памяти область до следующей границы слова.

Элементы массива типа BOOL начинаются с младшего бита; элементы массива типа BYTE и CHAR начинаются с расположенного справа байта (см. рис. 24.5, левая часть). При этом элементы массива следуют один за другим по порядку.

В многомерных массивах элементы хранятся, выстроенными в цепочки (в порядке следования размерностей), начиная с первой размерности (см. рис. 24.5, правая часть). Для элементов, имеющих длину 1 бит или 1 байт, новая размерность всегда начинается *со следующего байта*, а для элементов других типов новая размерность начинается *со следующего слова* (то есть, со следующего четного байта).

[1)] n = четный байт

Рис. 24.5 Структура переменной типа ARRAY

24.2.4 Тип данных STRUCT

Тип данных STRUCT определяет структуру данных, состоящую из фиксированного числа компонентов, которые могут относиться к различным типам данных.

Объявление (Declaration)

```
structname : STRUCT
    komp1name : datatype := pre-assignment;
    komp2name : datatype := pre-assignment;
    …
    END_STRUCT;
```

STRUCT и END_STRUCT - ключевые слова; они могут быть записаны также и в нижнем регистре.

structname - имя структуры;

pre-assignment - предопределенное значение переменной;

komp1name, komp2name - имена отдельных компонентов структуры;

datatype - тип данных имена отдельных компонентов структуры;

Предопределение (Pre-assignment)

Пользователь может предопределить на этапе объявления отдельные компоненты структуры (в отличие от параметров функции, проходных параметров в функциональном блоке или временных переменных). Тип данных предопределения должен соответствовать типу, заявленному для переменной.

Таблица 24.5 Пример объявления переменной типа STRUCT

Имя		Тип данных	Начальное значение	Комментарий
MotCont		STRUCT		Переменная простой структуры с 4 компонентами
	On	BOOL	FALSE (ЛОЖЬ)	Переменная MotCont.On типа BOOL
	Off	BOOL	TRUE (ИСТИНА)	Переменная MotCont.Off типа BOOL
	Delay	S5TIME	S5TIME#5s	Переменная MotCont.Delay типа S5TIME
	maxSpeed	INT	5000	Переменная MotCont.maxSpeed типа S5TIME
		END_STRUCT		

Применение

Переменные типа STRUCT Вы можете целиком применять в качестве параметров блоков типа STRUCT с такой же структурой или как параметры блоков типа ANY. Например, Вы можете скопировать с помощью системной функции SFC 20 BLKMOV содержимое переменной типа STRUCT. Вы можете также определить отдельные компоненты структуры в качестве параметров блока типа STRUCT, если параметры блока относятся к тому же типу данных, что и эти компоненты.

Если отдельные компоненты структуры относятся к простым типам данных, то Вы можете обрабатывать их с помощью обычных STL-операторов.

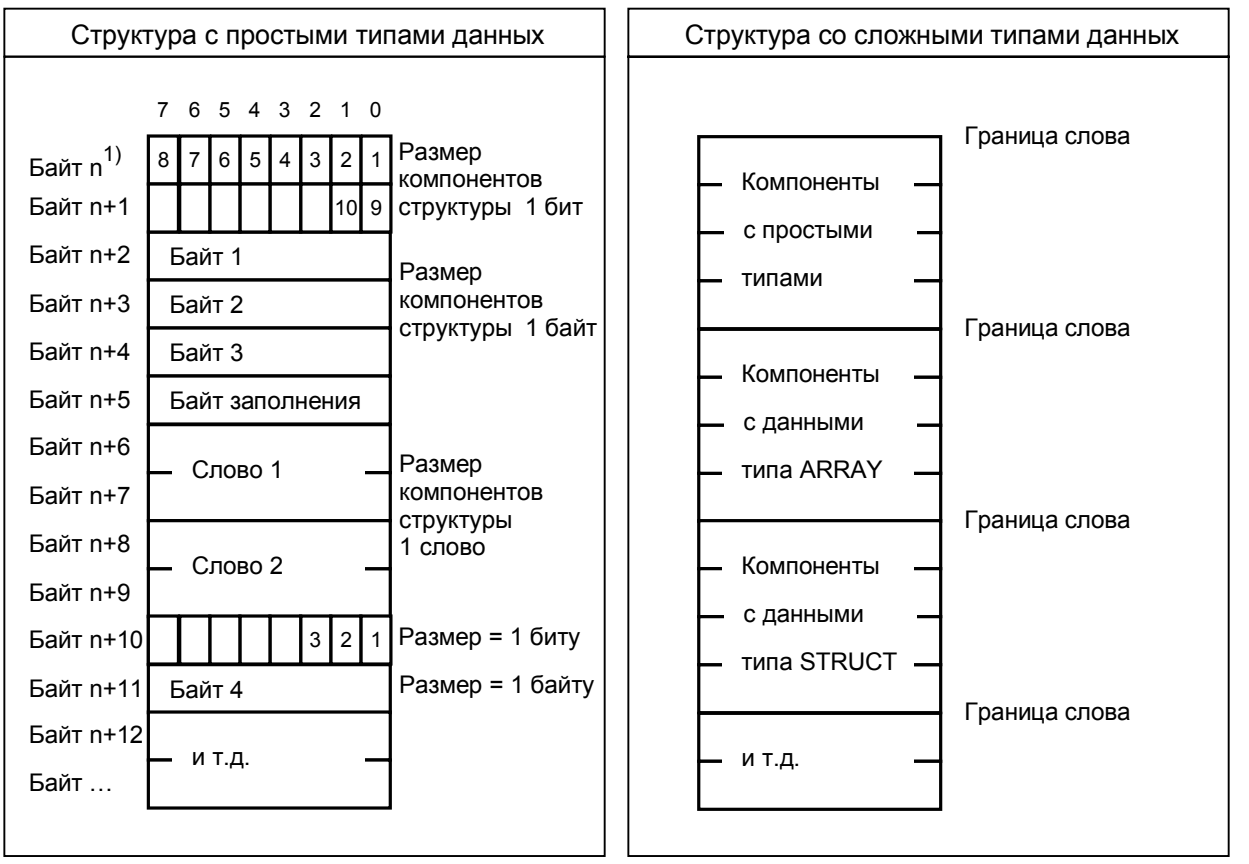
Доступ к компоненту структуры обеспечивается с использованием имени структуры и имени компонента, разделенных точкой.

Структура переменных

Переменные типа STRUCT всегда начинаются на границе слов (то есть, начинаются с байта с четным адресом); отдельные компоненты располагаются в памяти в том порядке, в котором они были описаны при объявлении. Переменная типа STRUCT занимает в памяти область до следующей границы слова.

Элементы массива типа BOOL начинаются с младшего бита; элементы массива типа BYTE и CHAR начинаются с расположенного справа байта (см. рис. 24.6). Компоненты других типов начинаются на границе слов.

Структура может быть *вложенной*. Вложенная структура - это такая структура, которая сама является компонентом другой структуры. Глубина вложения для структур может достигать шести вложений. Если отдельные компоненты структуры относятся к простым типам данных, то Вы можете обрабатывать их с помощью обычных STL-операторов. Отдельные имена компонентов структуры в полном имени разделяются точками.



1) n = четный байт

Рис. 24.6 Структура переменной типа STRUCT

24.3 Пользовательский тип данных

Пользовательский тип данных (UDT - "User-defined Data Type") определяет некоторую структуру данных (как комбинацию компонентов, которые могут относиться к различным типам данных), относящуюся к глобальным данным. Вы можете использовать пользовательский тип данных (UDT), если такая структура данных часто встречается в Вашей программе или Вы просто желаете дать имя структуре данных.

Вы можете создать структуру данных UDT либо при инкрементном редактировании программы, либо в текстовом редакторе, создавая исходный файл программы. Данные типа UDT могут быть созданы либо с использованием STL, либо с использованием SCL языков программирования одинаковым образом (Вы можете также запрограммировать данные UDT инкрементным способом с использованием языка SCL, если эти данные расположены в объекте *Blocks* (*Блоки*)).

Данные UDT являются глобальными, то есть, будучи один раз объявленными, они могут быть использованы во всех блоках. Доступ к данным UDT может быть организован с использованием имени (доступ по символу); для этого данным UDT Вы должны назначить абсолютный адрес в таблице символов (symbol table). Типу данных UDT (в таблице символов) соответствует абсолютный адрес.

Если Вы хотите передать в переменную структуру данных, определенную в UDT, назначьте переменной при ее объявлении пользовательский тип данных нужной структуры, как это делается при назначении "обычных" типов данных. Доступ к данным типа UDT может быть организован как по абсолютному адресу (UDT0 … UDT65535), так и по имени (символьная адресация).

Вы можете также определить тип данных UDT целому блоку данных. При программировании блока данных Вы можете назначить требуемый тип UDT блоку как структуре данных.

Пример "Данные фрейма сообщения" ("Message Frame Data") в разделе 26.4 "Краткое описание примера фрейма сообщения" объясняет, как работать с данными пользовательского типа.

24.3.1 Инкрементное программирование данных, определенных пользователем (UDT)

Вы можете создавать данные пользовательского типа (UDT) или с помощью утилиты SIMATIC Manager, выбрав сначала объект *Blocks (Блоки),* а затем - опции меню: *Insert -> S7 Block -> Data Type (Вставка -> S7 Block -> Тип данных)* или в редакторе, выбрав опции меню: *File -> New (Файл -> Создать)* и задав затем "UDTn" в строке "Имя объекта".

Двойной щелчок на объекте *UDT* в окне программы позволит открыть окно таблицы объявления данных, которая выглядит точно также, как таблица объявления блока данных. Структура UDT программируется точно также, как блок данных: с отдельными строками для имени (Name), типа (Type),

начального значения (Initial value) и комментария (Comment). Единственное различие между таблицами заключается в том, что здесь невозможно переключение к обзору данных (data view). (С помощью данных UDT Вы не можете создавать никаких переменных, а только собираете данные разных типов в одну структуру; по этой причине здесь не может быть фактических значений для компонентов структуры).

Начальные значения, которые Вы запрограммируете для структуры UDT, пересылаются в переменные при объявлении последних.

24.3.2 Применение данных UDT при создании исходных текстов программы

При программировании, ориентированном на создание исходных текстов программ, создание данных, определенных пользователем (UDT), выполняется при вводе данных типа STRUC ("структура"), заключенных между ключевыми словами TYPE и END_TYPE.

Объявление (Declaration)

```
TYPE udtname :
    STRUCT
    komp1name : datatype := pre-assignment;
    komp2name : datatype := pre-assignment;
    …
    END_STRUCT;
END_TYPE
```

TYPE, END_TYPE, STRUCT и END_STRUCT - ключевые слова; они могут быть записаны также и в нижнем регистре.

udtname - имя данных, определенных пользователем (UDT); вместо *udtname* Вы можете использовать абсолютный адрес UDT*n*.

pre-assignment - предопределенное значение переменной;

komp1name, komp2name - имена отдельных компонентов структуры;

datatype - тип данных отдельных компонентов структуры; здесь могут быть использованы все типы данных, кроме POINTER и ANY.

Предопределение (Pre-assignment)

Предопределение данных, определенных пользователем (UDT), производится точно также, как предопределение структуры STRUCT. структура данных также полностью соответствует типу данных STRUCT.

При предопределении данных, определенных пользователем (UDT), способ записи констант, принятый в языке программирования STL, применяется также и в SCL (см. обзор в разделе 3.7.3 "Простые типы данных").

Таблица 24.6 Пример объявления переменной типа UDT

Имя		Тип данных	Начальное значение	Комментарий
		STRUCT		
	Identifier	WORD	W#16#F200	Компонент UDT Identifier типа WORD
	Number	INT	0	Компонент UDT Number типа INT
	TimeofDay	TIME_OF_DAY	5000	Компонент UDT TimeofDay типа TIME_OF_DAY
		END_STRUCT		

25 Косвенная адресация

Косвенная адресация дает Вам возможность доступа к адресам, которые неизвестны до начала выполнения программы. С помощью косвенной адресации Вы также можете выполнять повторяющиеся разделы, части программы, такие, например, как циклы, и при этом при каждом проходе цикла Вы можете назначать разные адреса данных. В данной главе рассматриваются вопросы использования косвенной адресации при программировании на STL. Использование косвенной адресации при программировании на SCL рассматривается в разделе 27.2.3 "Косвенная адресация в SCL".

Так как при косвенной адресации адреса не вычисляются до выполнения программы, существует опасность того, что области памяти будут непреднамеренно перезаписаны. *В этом случае поведение программируемого контроллера может быть непредсказуемым! Будьте крайне осторожны при использовании косвенной адресации!*

Примеры для данной главы также представлены на дискете, прилагаемой к книге, в разделе "Variable Handling" ("Обработка переменных") в функциональном блоке FB 125 или в исходном файле Chap_25.

25.1 Указатели

Адрес при использовании косвенной адресации должен иметь такую структуру, в которой содержится адрес бита, адрес байта и, если необходимо, адрес области (*address area*). Поэтому такой адрес имеет специальный формат, имеющий название *указатель (Pointer).* Указатель используется для "указания" на адрес.

Система STEP 7 различает три типа указателей:

- указатели на область (area pointers)
  длина такого указателя составляет 32 бита и содержит определенный адрес;
- указатели на DB (DB pointers)
  длина такого указателя составляет 48 битов и содержит кроме адреса (как в указателе на область) номер блока данных;
- ANY-указатели (ANY pointers)
  длина такого указателя составляет 80 битов и содержит кроме информации, общей с указателем на DB, например, тип данных операнда.

Только первый тип указателей, т.е. указатели на область (area pointers), имеют значение для косвенной адресации. Указатели на DB и ANY-указатели используются при передаче параметров блока. Так как эти типы указателей содержат указатели на область, в данной главе описывается также и структура указателей на DB и ANY-указателей.

25.1.1 Указатели на область (area pointers)

Указатели на область (area pointers) содержат адрес операнда и, возможно, также адрес области, где находится операнд. Если адрес (идентификатор) области не указывается, то это *"внутризонный указатель" (area-internal pointer)*, иначе, если адрес (идентификатор) области присутствует, то это *межзонный указатель (area-crossing pointer)*.

Вы можете использовать указатель на область (area pointer) непосредственно и загружать (*load*) в аккумулятор или в адресный регистр, так как длина такого указателя составляет 32 бита. Ниже представлена форма записи для представления константы для двух случаев:

P#y.x для случая "внутризонного указателя" (area-internal pointer), например, P#22.0 и

P#Zy.x для случая "межзонного указателя" (area-crossing pointer), например, P#M22.0,

где x = адрес бита, x = адрес байта, Z = "адрес области". Как "адрес области" Вы используете идентификатор области. По значению бита 31 определяется тип указателя (внутризонный указатель или межзонный указатель) (см. рис. 25.1).

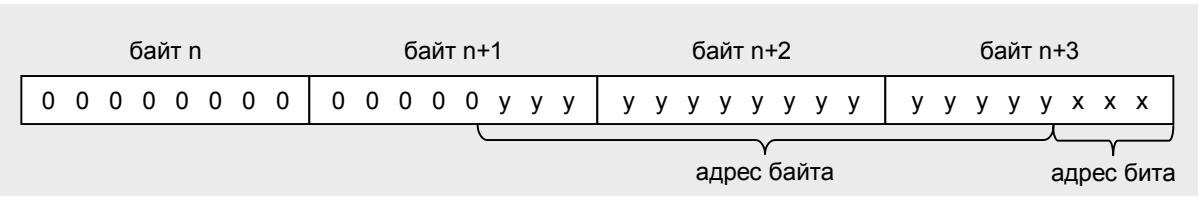
Указатель на область (area pointers) содержит адрес бита, который всегда должен быть определен, даже для числовых операндов. Для числовых операндов адрес бита равен 0 (нулю).

С помощью указателя на область P#M22.0 Вы можете, например, адресовать меркер (memory bit) M 22.0, а также байт в области меркеров (memory byte) MB 22, слово в области меркеров (memory word) MW 22 или двойное слово в области меркеров (memory duoble word) MW 22.

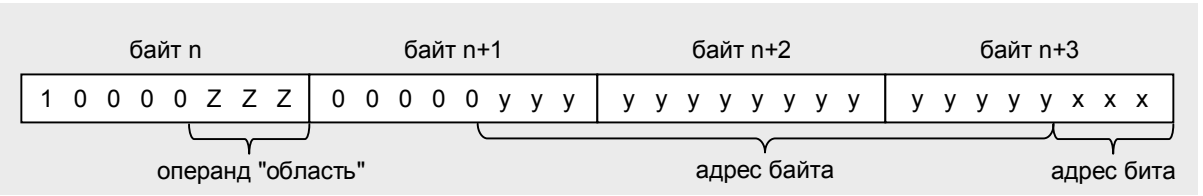
25.1.2 Указатели на DB (DB pointers)

Указатели на DB (DB pointers) содержит кроме адреса (как в указателе на область) номер блока данных в виде положительного целого числа типа INT. Этот номер идентифицирует блок данных, в то время как другая составная часть указателя содержит адрес в области глобальных данных или в области данных экземпляра. Во всех остальных случаях первые два байта указателя на DB будут содержать нуль.

Внутризонный указатель



Межзонный указатель



Адресная область:

0	0	0	Периферийные входы/выходы (P)
0	0	1	Входы (I)
0	1	0	Выходы (Q)
0	1	1	Меркеры (M)
1	0	0	Глобальные данные (DBX)
1	0	1	Экземплярные данные (DIX)
1	1	0	Временные локальные данные (L) [1]
1	1	1	Временные локальные данные вызывающего блока (V) [2]

[1] Не применяется при межзонной адресации

[2] Только при передаче параметра блока

Типы, используемые с ANY-указателем

Простые типы		**Сложные типы**
01	BOOL	0E DT
02	BYTE	13 STRING
03	CHAR	**Типы параметра**
04	WORD	17 BLOCK_FB
05	INT	18 BLOCK_FC
06	DWORD	19 BLOCK_DB
07	DINT	1A BLOCK_SDB
08	REAL	1C COUNTER
09	DATE	1D TIMER
0A	TOD	**Нуль-указатель (Zero-pointer)**
0B	TIME	
0C	S5TIME	00 NIL

Рис. 25.1 Структура указателя для косвенной адресации

Вы знакомы с форматом записи указателя по материалу, касающемуся вопроса полной адресации данных. Здесь также идентификатор блока данных и адрес данных разделяются точкой:

P#DataBlock.DataAddress

Пример:

P#DB 10.DBX 20.5

Вы не можете загружать данный указатель в аккумулятор; тем не менее, Вы можете применять его в качестве параметра блока с типом POINTER для указания адреса данных (кроме SCL). STEP 7 использует данный тип указателя для передачи фактических параметров.

25.1.3 ANY-указатели (ANY pointer)

Указатель ANY (ANY pointer) содержит кроме информации, общей с указателем на DB, тип данных адреса и коэффициент повторения. Это позволяет использовать такой указатель ANY для указания на область данных.

Указатель ANY используется в двух вариантах: для переменных с типом данных и для переменных с типом параметра. Если Вы используете указатель для переменных с типом данных, то указатель ANY содержит указатель на DB, тип и множитель повторения. Если Вы используете указатель для переменных с типом параметра, то указатель ANY содержит в дополнение к типу только номер вместо указателя на DB. Для функций таймера или счетчика тип повторяется в байте (n+6); байт (n+7) содержит B#16#00. Во всех остальных случаях эти два байта содержат значение W#16#0000.

Первый байт указателя ANY содержит синтаксическую структуру ID; в STEP 7 это всегда 10_{hex}. Тип определяет тип данных переменных, для которых данный указатель ANY применяется. Переменные простых типов, DT и STRING принимают тип, показанный на рисунке 25.1, и размер 1.

Если Вы используете переменную типа массива ARRAY или структуры STRUCT (а также пользовательского типа UDT) в качестве параметра ANY, то редактор создает указатель ANY для массива или структуры. Этот указатель ANY будет содержать идентификатор для байта BYTE (02_{hex}) как тип и число байтов, определяя этим самым длину переменной как ее размер (quantity). Тип данных отдельного элемента массива или структуры при этом не имеют значения. Указатель ANY с двойным количеством байтов таким образом указывает на массив из слов (WORD).

Исключение: указатель на массив, состоящий из элементов типа CHAR также используется с типом CHAR (03_{hex}).

Вы можете применять указатель ANY в качестве параметра блока типа "параметр ANY", если необходимо указать на переменную или на адресную область.

Представление констант для *типов данных* выполняется следующим образом:

P#[DataBlock.]Address Type Quantity = P#[БлокДанных.]Адрес Тип Размер.

Примеры:

- P#DB 11.DBX 30.0 INT 12
  область на 12 слов в блоке DB 11, начиная с DBB 30;
- P#M 16.0 BYTE 8
  область на 8 байт, начиная с MB 16;
- P#I 18.0 WORD 1
  слово входов IW 18;
- P#I 1.0 BOOL 1
  вход I 1.0.

Представление констант для *типов данных* выполняется следующим образом:

L# Number Type Quantity = L# Номер Тип Размер.

Примеры:

- L# 10 TIMER 1
  функция таймера T 10;
- L# 2 COUNTER 1
  функция счетчика Z 2.

Редактор использует указатель ANY, если тот согласован по типу и по размеру со спецификацией в представлении константы.

Необходимо отметить, что местоположение в памяти в указателе ANY для типов данных должно быть задано адресом бита.

Спецификация константы посредством указателя ANY имеет смысл, если Вы хотите получить доступ к области данных, для которой Вы не объявили переменной. В принципе, Вы можете также применить переменные или адреса в качестве параметра ANY. Например, представление "P#I 1.0 BOOL 1" идентично "I 1.0" или соответствующему символьному адресу.

С помощью типа параметра ANY Вы можете также объявить переменные во временных локальных данных. Вы можете использовать эти переменные для создания указателя ANY, который можно модифицировать во время выполнения программы (см. раздел 16.3.3 "Переменная "указатель ANY"").

Если Вы не выполняете никакого предопределения при объявлении параметра ANY в функциональном блоке, то редактор назначает значение 10_{hex} синтаксической структуре ID и 00_{hex} - всем остальным байтам. После этого редактор представляет такие (пустые) указатели ANY (при просмотре данных) следующим образом:

P#P0.0 VOID 0.

25.2 Типы косвенной адресации в STL

В данном разделе рассматриваются вопросы использования косвенной адресации в языке программирования STL; информацию по вопросам применения косвенной адресации в языке программирования SCL Вы можете найти в разделе 27.2.3 "Косвенная адресация в SCL".

25.2.1 Общая информация

Косвенная адресация возможна только с использованием абсолютных адресов. Вы не сможете применять косвенную адресацию с использованием символьных адресов (Вы должны также иметь возможность получать прямой доступ к отдельным элементам массива в STL). Если Вы желаете иметь косвенный доступ к переменной, Вы должны знать абсолютный адрес переменной. STL поддерживает прямой доступ к переменным (см. следующую главу).

Абсолютная адресация может применяться как:

- непосредственная адресация (immediate addressing);
- прямая адресация (direct addressing);
- косвенная адресация (indirect addressing).

Адресация посредством параметров блока - особая форма косвенной адресации: назначая фактические параметры параметрам блока, Вы определяете адреса, которые будут обработаны во время выполнения программы.

Мы имеем дело с *непосредственной адресацией (immediate addressing)*, когда численное значение (значение числа) задается непосредственно в операторе. Примером непосредственной адресации могут служить операция загрузки (load) значения константы в аккумулятор, операция сдвига фиксированного значения, а также установка или сброс результата логической операции посредством операций SET (УСТАНОВКА) и CLR (ОЧИСТКА).

С помощью *прямой адресации (direct addressing)* Вы получаете прямой доступ к адресу, например, A I 1.2 или L MW 122. Значение, с которым Вы хотите выполнить операцию, или значение, которое Вы хотите загрузить в аккумулятор, расположено по указываемому адресу, то есть в ячейке памяти. Вы адресуете данную ячейку памяти, указывая ее адрес непосредственно в операторе STL.

С помощью косвенной адресации (indirect addressing) STL-выражение указывает, где адрес может быть найден, вместо указания самого адреса. Мы различаем два типа косвенной адресации, различие между которыми определяется способом указания на адрес. Это:

- **косвенная адресация посредством памяти (memory-indirect addressing)**, которая использует адрес в системной памяти для определения адреса; пример: в выражении T QW [MD 220] адрес слова выходов, в которые должна быть сделана пересылка, размещен в двойном слове меркеров MD 220;
- **косвенная адресация посредством регистра (register-indirect addressing)**, которая использует адресный регистр для определения расположения адреса; пример: в выражении T QW [AR1,P#2.0] адрес слова выходов, в которые должна быть сделана пересылка, находится на 2 байта выше адреса, размещенного в адресном регистре AR1.

Вы можете использовать косвенную адресацию посредством регистра (register-indirect addressing) в двух вариантах: как *внутризонную косвенную адресацию посредством регистра (area-internal register-indirect addressing)* и как *межзонную косвенную адресацию посредством регистра (area-crossing register-indirect addressing).*

С помощью внутризонной косвенной адресации посредством регистра (area-internal register-indirect addressing) Вы можете запрограммировать в операторе адресную область, адрес для которой указывается в адресном регистре. Следовательно, адрес в адресном регистре может быть изменен в пределах адресной области (например: выражение L MW [AR1,P#0.0] обеспечивает загрузку [load] слова меркеров, адрес которого помещен в AR1).

С помощью межзонной косвенной адресации посредством регистра (area-crossing register-indirect addressing) Вы определяете в выражении только длину (размер) адреса (бит, байт, слово, двойное слово). Адрес для адресной области указывается в адресном регистре, при этом он может изменяться динамически (например: выражение L W [AR1,P#0.0] обеспечивает загрузку [load] слова, информация об адресной области и адрес которого находятся в AR1).

25.2.2 Косвенная адресация (Indirect Addresses)

Адреса, которые могут быть определены косвенно, могут быть разделены на две категории:

- адреса, для которых может быть назначен простой тип данных;
- адреса, для которых может быть назначен тип параметра.

Вы можете использовать косвенную адресацию посредством памяти и косвенную адресацию посредством регистра с первой категорией из указанных адресов, но со второй категорией адресов Вы можете использовать только косвенную адресацию посредством памяти (см. таблицу 25.1). Операнды, которые не могут иметь битовый адрес, также не требуют адреса бита в указателе, так что 16-разрядный номер вполне достаточен в качестве адреса (беззнаковое целое INT-число).

Таблица 25.1 Косвенная адресация

Адреса, которые могут быть определены косвенно	Адресация	Указатель
Периферийные входы/выходы (I/O), входы, выходы, меркеры, глобальные данные, экземплярные данные, временные локальные данные.	Косвенная адресация посредством памяти и косвенная адресация посредством регистра	Указатели на область: или внутризонный способ указания, или межзонный способ указания
Таймеры, счетчики, функции, функциональные блоки, блоки данных	Косвенная адресация посредством памяти	16-разрядный номер

Область указателей имеет теоретический размер в пределах от 0 до 65535 (байтовых адресов или номеров). На практике верхняя граница для адресов в каждом случае зависит от CPU. Диапазон адресов для битов (битовых адресов) занимает значения от 0 до 7.

25.2.3 Косвенная адресация посредством памяти (memory-indirect addressing)

При косвенной адресации посредством памяти (memory-indirect addressing) адрес указывается посредством адресованной ячейки памяти. Адрес должен иметь размер двойного слова, если требуется использовать указатель на область (area pointer), или же он должен иметь размер слова (WORD), если требуется при косвенной адресации использовать число в качестве указателя.

Адрес операнда может находиться в одной из ниже перечисленных адресных областей:

- область меркеров
  как абсолютный адрес или символьная переменная;
- L-стек (область временных локальных данных)
  как абсолютный адрес или символьная переменная;
- блок глобальных данных
  как абсолютный адрес;
  при использовании адресации глобальных данных пользователь должен обеспечить, чтобы требуемый блок данных был открыт с помощью DB-регистра; если, например, Вы указываете адрес адреса глобальных данных косвенным образом посредством двойного слова в глобальных данных, то все операции (вычисление адреса) должны быть произведены с учетом того, что все данные находятся в одном и том же блоке данных.
- экземплярный блок данных
  как абсолютный адрес или символьная переменная;
  существуют определенные ограничения при использовании экземплярных данных в качестве адреса; (об этом см. ниже).

Если Вы используете экземплярные данные в качестве адресов в функциях, обрабатывайте их точно таким же образом, как и адреса глобальных данных; при этом Вы должны использовать DI-регистр вместо DB-регистра. Символьная адресация не допускается в этом случае. Вы можете также использовать экземплярные данные в качестве адресов в функциональных блоках, но только если Вы скомпилировали их как блоки CODE_VERSION1 (без поддержки "мультиэкземплярности").

Косвенная адресация с указателем на область (area pointer)

Указатель на область (area pointer), используемый для косвенной адресации посредством памяти, всегда является внутризонным указателем (area-internal pointer). Это означает, что он содержит адрес байта и адрес бита. Если Вы хотите адресовать числовые операнды, то Вы должны всегда в качестве адреса бита указывать 0.

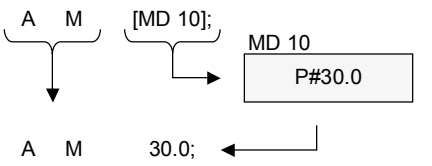
Пример: двойное слово MD 10 содержит указатель P#30.0. Выражение A M [MD 10] дает доступ к меркеру, адрес которого размещен в двойном слове MD 10; следовательно, с помощью этого выражения проверяется меркер M 30.0 (рис. 25.2).

С помощью выражения L MW [MD 10] Вы можете загрузить слово меркеров MW 30 в аккумулятор.

Вы можете использовать косвенную адресацию посредством памяти для всех бинарных операндов при использовании двоичных логических операций и операций с памятью, а также любых числовых операндов при использовании операций загрузки (load) и передачи (transfer).

Косвенная адресация посредством памяти с указателем на область

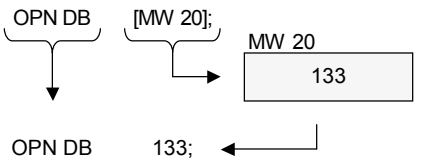
```
L    P#30.0;          //загрузка указателя в аккумулятор
                      //accumulator 1
T    MD 10;           //сохранение в двойном слове
                      //меркеров
L    MW [MD 10];      //назначение дискретного адреса
…
A    M [MD 10];       //назначение двоичного адреса
```



Косвенная адресация посредством памяти по номеру

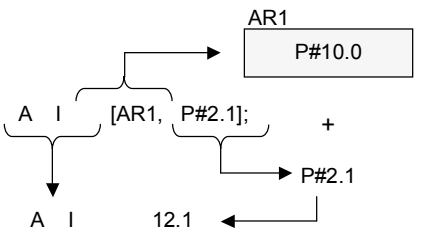
```
L    133;             //загрузка указателя в аккумулятор
                      //accumulator 1
T    MW 20;           //сохранение в слове меркеров

OPN DB [MW 20];       //открытие глобального блока DB
…
SP   T [MW 20];       //запуск таймера
```



Косвенная внутризонная адресация посредством регистра

```
LAR1 P#10.0;          //загрузка указателя в адресный
                      //регистр 1
…
L    MW [AR1,P#4.0]; //назначение дискретного
                      //адреса
…
A    I  [AR1,P#2.1];  //назначение двоичного адреса
```



Косвенная межзонная адресация посредством регистра

```
LAR1 P#M12.0;         //загрузка указателя в адресный
                      //регистр 1
…
L      B [AR1,P#4.0]; //назначение дискретного
                      //адреса
…
=      I  [AR1,P#0.7]; //назначение двоичного адреса
```

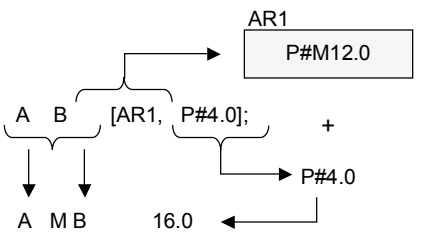


Рис. 25.2 Разновидности косвенной адресации

Косвенная адресация с использованием номера

Номер, используемый для косвенной адресации таймеров, счетчиков и блоков имеет длину 16 битов. Операнд, имеющий размер слова (WORD),

достаточен для хранения номера.

Пример: слово MW 20 содержит число (номер) 133. Выражение

OPN DB [MW 20]

открывает блок глобальных данных, номер которого размещен в слове меркеров MW 20. С помощью выражения

SP T [MW 20]

Вы можете запустить таймер T 133 в режиме импульса.

С помощью косвенной адресации Вы можете использовать любые операции с таймерами и счетчиками. Вы можете открывать блоки данных или с использованием DB-регистра (OPN DB […]), или с использованием DI-регистра (OPN DI […]). Если слово адреса содержит 0, то CPU выполняет NOP-операцию.

Вы можете косвенно адресовать вызовы кодовых блоков, используя UC FC[..] и CC FC[..] или UC FB[..] и CC FB[..]. Вызов блоков посредством операций UC или CC просто приводит к смене блока; при этом не происходит ни передачи параметров блока, ни открытия экземплярного блока.

25.2.4 Косвенная внутризонная адресация посредством регистра (Register-Indirect Area-Internal Addressing)

При косвенной внутризонной адресации посредством регистра (register-indirect area-internal addressing) адрес указывается посредством одного из двух адресных регистров. Содержимое адресного регистра является внутризонным указателем.

При косвенной адресации посредством регистра (register-indirect addressing) задается смещение в дополнение к информации адресного регистра. Это смещение добавляется к содержимому адресного регистра при выполнении соответствующей операции (без изменения содержимого адресного регистра). Указанное смещение имеет формат внутризонного указателя (area-internal pointer). Вы всегда должны задавать смещение при использовании данного вида адресации, и при этом Вы должны определять его константой. При косвенной адресации числовых операндов смещение всегда должно иметь нулевой битовый адрес (=0). При этом максимальное значение составляет P#8191.7.

Пример: Адресный регистр AR1 содержит указатель на область (area pointer) P#10.0 (при использовании оператора LAR1 Вы можете загрузить указатель непосредственно в адресный регистр AR1, об этом см. ниже). Выражение

A I [AR1.P#2.1]

добавляет указатель к адресному регистру AR1 и, таким образом, формирует адрес входа, состояние которого должно быть проверено. С помощью выражения

L MW [AR1.P#4.0]

Вы можете загрузить слово меркеров MW 14 в аккумулятор.

Внутризонная адресация с использованием межзонных указателей

Если адресный регистр содержит межзонный указатель (area-crossing pointer), и если Вы используете данный адресный регистр для операций с внутризонной адресацией данных, то адрес области в адресном регистре будет игнорироваться.

Пример: Следующие инструкции обеспечивают загрузку (load) в адресный регистр AR1 межзонного указателя (area-crossing pointer) на бит DBX 20.0 в глобальных данных, и после этого обеспечивается внутризонная адресация посредством адресного регистра AR1 в двойное слово меркеров. При выполнении следующего оператора выполняется загрузка в аккумулятор двойного слова MD 20.

```
LAR1    P#DBX 20.0;
L       MD[AR1,P#0.0];
```

25.2.5 Косвенная межзонная адресация посредством регистра (Register-Indirect Area-Crossing Addressing)

При косвенной межзонной адресации посредством регистра (register-indirect area-crossing addressing) адрес располагается в одном из двух адресных регистров. Содержимое адресного регистра является межзонным указателем (area-crossing pointer).

При межзонной адресации Вы дополнительно устанавливаете адресную область посредством указателя на область (area pointer) в адресный регистр. Если Вы используете косвенную адресацию Вы задаете только идентификатор ID для спецификации длины адреса: нет спецификации для бита, "B" - для байта, "W" - для слова, "D" - для двойного слова.

Как и при внутризонной адресации Вы используете в данном случае смещение, которое Вы должны определить с помощью фиксированного значения для битового адреса. Содержание адресного регистра не изменяется при задании смещения.

Пример: Адресный регистр AR1 содержит указатель на область (area pointer) P#12.0 (при использовании инструкции

```
LAR1 P#M12.0
```

Вы можете загрузить указатель непосредственно в адресный регистр AR1, об этом см. ниже). Инструкция

```
L B [AR1.P#4.0]
```

добавляет указатель P#4.0 к адресному регистру AR1 и, таким образом, формирует адрес слова меркеров, которое должно быть загружено (MB 16 в данном случае). С помощью инструкции

```
=  [AR1.P#0.7]
```

Вы можете назначить результат логической операции (т.е. RLO) меркеру M 12.7.

Вы не можете (в настоящее время) использовать межзонную адресацию для временных локальных данных (при этом CPU переходит в режим STOP). Вы должны использовать внутризонную адресацию, если

адресуемая область расположена в области временных локальных данных.

25.2.6 Резюме

Итак, в каких случаях использовать те или иные способы адресации? Если это возможно, то используйте косвенную внутризонную адресацию посредством регистра (register-indirect area-internal addressing). Язык программирования STL более всего приспособлен для использования такой адресации. Вы можете видеть указание на адресную область, доступ к которой необходим для выполнения операции. И кроме того, CPU обрабатывает программу с косвенной внутризонной адресацией посредством регистра (register-indirect area-crossing addressing) быстрее всего.

С другой стороны, косвенная адресация посредством памяти (memory-indirect addressing) обеспечивает преимущество, если в выполняющуюся программу включены больше, чем два указателя. Однако необходимо учитывать период "валидности" (корректности) указателя:

так, указатель в области меркеров может корректно использоваться без каких-либо ограничений при обработке программы целиком и даже на протяжении нескольких циклов сканирования программы;

указатель в блоке данных может корректно использоваться пока блок находится в открытом состоянии;

указатель в области временных локальных данных может корректно использоваться только во время обработки блока.

Если адресные области также должны быть доступны с использованием переменной адресации, то косвенная адресация посредством регистра является правильным выбором.

В таблице 25.2 представлены сравнительные данные вариантов косвенной адресации. Все показанные фрагменты программ ведут к одному и тому же результату - к установке выхода Q 4.7.

Таблица 25.2 Сравнительные данные вариантов косвенной адресации

Косвенная адресация посредством памяти	Косвенная внутризонная адресация посредством регистра	Косвенная межзонная адресация посредством регистра
L P#4.7	LAR1 P#4.7	LAR1 P#Q4.7
T MD 24		
S Q [MD 24]	S Q [AR1, P#0.0]	S [AR1, P#0.0]

25.3 Использование адресных регистров

Ниже показаны инструкции, которые возможны в языке программирования STL в связи с использованием адресных регистров: в форме списка инструкций (см. рис. 25.3.1) и в графической форме (см. рис. 25.3.2):

LAR1	-	Загрузка в адресный регистр AR1
LAR2	-	Загрузка в адресный регистр AR2
	P#Zy.x	межзонного указателя
	P#y.x	внутризонного указателя
LAR1	-	Загрузка в адресный регистр AR1 содержимого
LAR2	-	Загрузка в адресный регистр AR2 содержимого
	MD y	двойного слова меркеров
	LD y	двойного слова локальных данных
	DBD y	двойного слова глобальных данных
	DID y	двойного слова экземплярных данных [1)]
LAR1		Загрузка в адресный регистр AR1 содержимого аккумулятора Accu 1
LAR2		Загрузка в адресный регистр AR2 содержимого аккумулятора Accu 1
LAR1	AR2	Загрузка в адресный регистр AR1 содержимого адресного регистра AR2
TAR1	-	Пересылка содержимого адресного регистра AR1 в
TAR2	-	Пересылка содержимого адресного регистра AR2 в
	MD y	двойное слово меркеров
	LD y	двойное слово локальных данных
	DBD y	двойное слово глобальных данных
	DID y	двойное слово экземплярных данных [1)]
TAR1		Пересылка содержимого адресного регистра AR1 в аккумулятор Accu 1
TAR2		Пересылка содержимого адресного регистра AR2 в аккумулятор Accu 1
TAR1	AR2	Пересылка содержимого адресного регистра AR1 в адресный регистр AR2
CAR	-	Обмен содержимым между адресными регистрами
+AR1	-	Сложить содержимое аккумулятора Accu 1 и адресного регистра AR1
+AR2	-	Сложить содержимое аккумулятора Accu 1 и адресного регистра AR2
+AR1	P#Zy.x	Прибавить указатель к содержимому адресного регистра AR1
+AR2	P#y.x	Прибавить указатель к содержимому адресного регистра AR2

[1)] При использовании адресации такого типа необходимо учитывать ряд ограничений (см. ниже "Особенности косвенной адресации")

Рис. 25.3.1 Список инструкций языка программирования STL с использованием адресных регистров

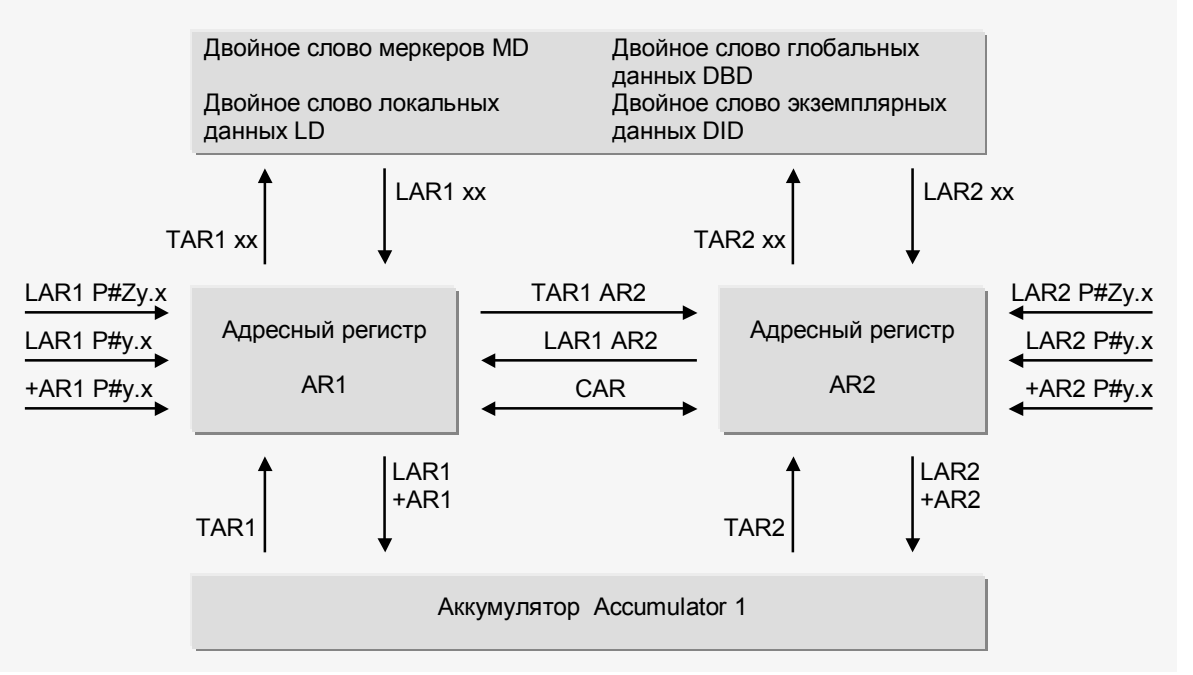


Рис. 25.3.2 Схема использования в STL адресных регистров

25.3.1 Загрузка в адресный регистр

Оператор LAR*n* загружает указатель на область (area pointer) в адресный регистр AR*n.* Исходными данными для загрузки в адресный регистр Вы можете выбрать внутризонный или межзонный указатель или двойное слово из области меркеров, из области временных локальных данных, из области глобальных данных или из области экземплярных данных. При этом содержимое двойного слова должно соответствовать формату указателя на область (area pointer).

В случае, если Вы не задаете адрес, оператор LAR*n* загружает содержимое аккумулятора accumulator 1 в адресный регистр AR*n.*

Если Вы используете инструкцию LAR1 AR2, то при выполнении данной инструкции происходит копирование содержимого адресного регистра AR2 в адресный регистр AR1.

Примеры:

```
LAR2  P#20.0;      //Загрузка указателя P#20.0 в AR2
L     P#24.0;      //Загрузка указателя P#24.0 в Accu1
LAR1  ;            //Загрузка содержимого аккумулятора
                   //Accumulator 1 в AR1
LAR1  MD 120;      //Загрузка содержимого MD 120 в AR1
LAR1  AR2;         //Загрузка содержимого AR2 в AR1
```

25.3.2 Пересылка из адресного регистра

Оператор TAR*n* выполняет пересылку указателя на область (area pointer) целиком из адресного регистра AR*n.* Как область назначения Вы можете выбрать двойное слово из области меркеров, из области временных локальных данных, из области глобальных данных или из области экземплярных данных.

В случае, если Вы не задаете адрес, оператор TAR*n* пересылает содержимое адресного регистра AR*n* в аккумулятор accumulator 1. При этом предыдущее содержимое аккумулятора accumulator 1 сдвигается в аккумулятор accumulator 2; предыдущее содержимое аккумулятора accumulator 2 теряется. Содержимое аккумуляторов accumulator 3 и accumulator 4 остается без изменения.

Если Вы используете инструкцию TAR1 AR2, то при выполнении данной инструкции происходит копирование содержимого адресного регистра AR1 в адресный регистр AR2.

Примеры:

```
TAR2  MD 140;      //Пересылка содержимого AR2 в MD 140
TAR1  ;            //Пересылка содержимого AR1 в
                   //аккумулятор Accumulator 1
TAR1  AR2;         //Пересылка содержимого AR1 в AR2
```

25.3.3 Обмен содержимым между адресными регистрами

Оператор CAR выполняет обмен содержимым между адресными регистрами AR1 и AR2.

Пример:

Пусть 8 байтов данных пересылаются между областью меркеров (MB 100) и областью данных (DB 20.DBB 200). Направление пересылки указывается в меркере M 126.6. Если меркер M 126.6 имеет состояние "0", то адресные регистры обмениваются содержимым. Если необходимо таким способом организовать пересылку данных между двумя блоками данных, загрузите оба регистра блока данных (с помощью инструкций OPN DB и OPN DI) содержимым адресных регистров и производите обмен данными с помощью оператора TDB.

```
      LAR1  P#M100.0;
      LAR2  P#DBX200.0;
      OPN   DB 20;
      A     M 126.6;
      JC    OV;
      CAR   ;
OV:   L     D[AR1,P#0.0];
      T     D[AR2,P#0.0];
      L     D[AR1,P#4.0];
      T     D[AR2,P#4.0];
```

Примечание: системная функция SFC 20 BLKMOV применяется для передачи больших объемов данных.

25.3.4 Операция сложения с содержимым адресного регистра

Пользователь может запрограммировать операцию сложения некоторого значения с содержимым адресного регистра для того, чтобы, например, инкрементировать адрес при каждом следующем проходе цикла в программе. Вы можете задать прибавляемое значение или как константу (как внутризонный указатель), или это значение может быть расположено в правом слове аккумулятора accumulator 1. Тип указателя в адресном регистре (внутризонный указатель или межзонный указатель) и адресной области при этом сохраняются.

Операции сложения с содержимым указателей

Инструкции +AR1 P#y.x и +AR2 P#y.x прибавляют указатель к содержимому указанного адресного регистра AR1 или AR2.

Необходимо отметить, что в данных инструкциях указатель на область (area pointer) может иметь максимальную величину P#4095.7. Если аккумулятор содержит величину, большую, чем P#4095.7, то число интерпретируется как инвертированное число с фиксированной запятой (two's complement) и вычитается.

Пример:

Пусть необходимо сравнить значение в машинном слове в области данных с некоторой величиной. Если сравниваемая величина больше, чем значение в области данных, то устанавливается меркер (получает значение "1"), в противоположном случае меркер сбрасывается (получает значение "0").

```
      OPN  DB 14;
      LAR1 P#DBX20.0;
      LAR2 P#M10.0;
      L    Quantity_Data;
Loop: T    LoopCounter;
      L    ComparisonVal;
      L    W[AR1,P#0.0];
      >I   ;
      =    [AR2,P#0.0];
      +AR1 P#2.0;
      +AR2 P#0.1;
      L    LoopCounter;
      LOOP Loop;
```

Операции сложения с содержимым аккумулятора

Инструкции +AR1 и +AR2 интерпретируют значение в аккумуляторе Accu1

как целое число формата INT, представляют его с соответствующим знаком как 24-разрядное число и прибавляют к содержимому указанного адресного регистра AR1 или AR2. Таким образом, указатель может быть также уменьшен. Нарушение верхней границы допустимого диапазона байтового адреса (0 … 65535) не имеет значения: "избыточные" биты отрезаются (см. рис. 25.4).

Необходимо отметить, что битовый адрес размещается в битах с номерами от 0 до 2. Если необходимо инкрементировать байтовый адрес в аккумуляторе Accumulator 1, то увеличение надо начинать с бита 3 (то есть, прибавляемое значение Вы должны сдвинуть на три разряда влево).

Пример: Пусть в блоке данных DB 14 16 байтов, адреса которых вычисляются с помощью указателя в двойном слове меркеров MD 220 и смещения в байте меркеров MB 18, должны быть очищены. Перед прибавлением к AR1 содержимое MB 18 должно быть смещено вправо на 3 разряда (SLW 3).

```
OPN   DB 14;
LAR1  MD 220;
L     MB 18;
SLW   3;
+AR1  ;
L     0;
T     DBD[AR1,P#0.0];
T     DBD[AR1,P#4.0];
T     DBD[AR1,P#8.0];
T     DBD[AR1,P#12.0];
```

Примечание:

Системная функция SFC 21 FILL используется для заполнения больших областей определенным значением бита.

Аккумулятор Accumulator 1

Адресный регистр

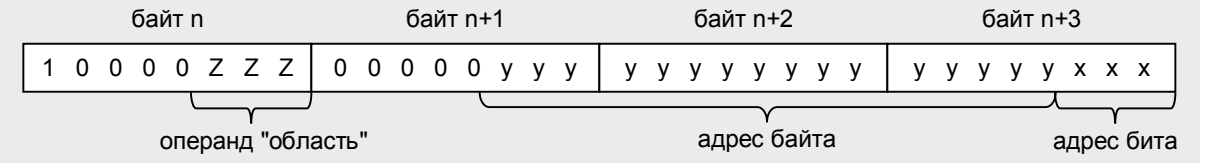


Рис. 25.4 Прибавление значения к адресному регистру

25.4 Особенности косвенной адресации

25.4.1 Использование адресного регистра AR1

В языке программирования STL адресный регистр AR1 используется для обращения к параметрам блока, которые пересылаются как DB-указатели. При использовании функций адресный регистр AR1 может применяться для обращения ко всем параметрам блока сложных типов данных, а в случае функциональных блоков это касается входных/выходных параметров сложных типов.

При использовании доступа к параметрам блока этого вида, например, для того, чтобы проверить битовый компонент структуры или чтобы записать целое INT-значение в элемент массива, содержание адресного регистра AR1 изменяется и также, изменяется содержание DB-регистра. Это происходит тогда, когда Вы передаете параметры блока данного типа в вызываемые блоки.

Если Вы используете адресный регистр AR1 (помимо косвенной адресации), при этом не должен быть возможен доступ к параметру блока (в соответствии с изложенным выше материалом). Другими словами, Вы должны сохранять содержимое адресного регистра AR1 до обращения к параметрам и вновь загружать регистр после выполнения операции доступа.

Пример:

Вы загружаете указатель в AR1 и используете этот адресный регистр для косвенной адресации. В определенный момент Вы хотите загрузить (load) значение компонента структуры *Motor.Act.* Перед загрузкой *Motor.Act* Вы сохраняете содержимое DB-регистра и адресного регистра AR1; после выполнения операции загрузки вышеназванного компонента структуры, Вы восстанавливаете состояния регистров (см. рис. 25.5, верхняя часть).

25.4.2 Использование адресного регистра AR2

При использовании функциональных блоков с возможностью работы в "мультиэкземплярном режиме" (блок версии 2), адресный регистр AR2 используется в STEP 7 как "базовый адресный регистр" ("Base address register") для экземплярных данных. При вызове экземпляра адресный регистр AR2 содержит P#DBX0.0 и при всех обращениях к параметрам блоков или к статическим локальным данным в FB используется косвенная внутризонная адресация области DI посредством данного регистра.

При вызове локального экземпляра производится инкрементирование "базового адреса" ("base address") при помощи +AR2 P#y.x, так что обращение к данным может быть выполнено относительно данного адреса внутри вызываемого функционального блока, который использует экземплярный блок данных вызывающего функционального блока. Таким образом, функциональные блоки могут быть вызваны и как автономные экземпляры, и как локальные экземпляры (к тому же в любой точке в функциональном блоке и даже несколько раз).

```
//********** Сохранение адресного регистра AR1 *********
...
VAR_TEMP
   AR1_Memory   : DWORD;
   DB_Memory    : WORD;
END_VAR
...
//Косвенная адресация с регистром AR1 и DB-регистром
LAR1  P#y.x;
OPN   DB z;
...
//Сохранение содержимого регистров
L     DBNO;
T     DB_Memory;
TAR1  AR1_Memory;
//Обращение к параметрам блока сложного типа
L     Motor. Act;
//Восстановление содержимого регистров
OPN   DB [DBMemory];
LAR1  AR1_Memory;
T     DBW[AR1,P#0.0];       //Сохранение   загруженного
                            //значения
//********** Сохранение адресного регистра AR2 *********
...
VAR_TEMP
   AR2_Memory : DWORD;
   DI_Memory  : WORD ;
END_VAR
...
//Сохранение содержимого регистров
L     DINO;
T     DI_Memory;
TAR2  AR2_Memory;
//Косвенная адресация с регистром AR2 и DI-регистром
LAR2  P#y.x;
OPN   DI z;
...
L     DIW[AR2,P#0.0]
...
//Восстановление содержимого регистров
OPN   DI [DI_Memory];
LAR2  AR2_Memory;
```

Рис. 25.5 Примеры сохранения адресных регистров AR1 и AR2

Если Вы программируете функциональный блок как блок версии 1,то есть, без "мультиэкземплярного режима", то STEP 7 не использует адресный регистр AR2.

Итак, Вы хотите использовать адресный регистр AR2 в функциональном блоке с возможностью работы в "мультиэкземплярном режиме", Вы должны сначала сохранить содержимое адресного регистра AR2 и DI-регистра до использования и вновь восстанавливать данные в регистрах после выполнения операции доступа. Вы не должны программировать никаких обращений к параметрам блока или к статическим локальным данным в области, в которой Вы используете адресный регистр AR2.

При использовании адресного регистра AR2 внутри функций никаких ограничений не накладывается.

Пример:

В функциональном блоке Вам необходимо использовать регистр AR2 и DI-регистр для выполнения косвенной адресации. Прежде всего Вы должны сохранить содержимое DI-регистра и адресного регистра AR2. При этом Вы не должны выполнять обращения к параметрам блоков или к статическим локальным данным до того, пока Вы не восстановите содержимое регистров (см. рис. 25.5, нижняя часть).

25.4.3 Ограничения на использование статических локальных данных

При использовании функциональных блоков, скомпилированных с CODE_VERSION1 (без "мультиэкземплярного режима"), Вы можете использовать все операторы (инструкции), описанные в данной главе без ограничений.

При использовании функциональных блоков с возможностью работы в "мультиэкземплярном режиме", редактор (Editor) обращается к экземплярным данным посредством адресного регистра AR2; то есть, все эти операции доступа носят косвенный характер. Это же касается косвенной адресации и обработки адресных регистров. Если Вы используете абсолютную адресацию для экземплярных данных, в которых Вы храните указатели на область (area pointers), редактор использует абсолютные адреса. Тем не менее, как только Вы начнете использовать символьную адресацию, редактор отбросит эту часть программы как попытку "двойной косвенной адресации" (duoble indirect addressing).

В таблице 25.3 представлены два таких примера: если Вы используете косвенную адресацию посредством памяти в функциональных блоках с возможностью работы в "мультиэкземплярном режиме", то Вы не можете непосредственно использовать указатель, который Вы желаете сохранять в статических локальных данных. Вы должны скопировать указатель во временные локальные данные, и после этого можете работать с ним. При этом Вы не можете непосредственно загрузить (load) указатель на статические локальные данные в адресный регистр и не можете также переслать содержимое адресного регистра непосредственно в указатель (второй пример).

Таблица 25.3 Различие в программировании FB версий 1 и 2

Для FB, скомпилированных с параметром CODE_VERSION1 (без "мультиэкземплярного режима")	Для FB, скомпилированных с параметром CODE_VERSION2 (с "мультиэкземплярным режимом")
VAR sPointer : DWORD; END_VAR	VAR sPointer : DWORD; END_VAR VAR_TEMP tPointer : DWORD; END_VAR
L MW[sPointer];	L sPointer; T tPointer; L MW[tPointer];
LAR1 sPointer;	L sPointer; LAR1;
TAR1 sPointer;	TAR1; T sPointer;

26 Прямой доступ к переменным

В данной главе рассматривается, как реализовать прямое обращение к локальным переменным с использованием их абсолютных адресов. "Обычные" STL-операторы позволяют обращаться к локальным переменным простых типов. Локальные переменные сложных типов, а также параметры блоков типа POINTER или ANY не могут быть обработаны "целиком". Для обработки таких переменных Вы сначала должны вычислить начальный адрес, по которому переменная хранится, а затем Вы можете обращаться к отдельным ее частям, используя косвенную адресацию. Таким же путем Вы можете также обрабатывать параметры блоков, относящиеся к сложным типам данных.

Примеры для данной главы также представлены на дискете, прилагаемой к книге, в библиотеке STL_Book в разделе "Variable Handling" ("Обработка переменных") в функциональном блоке FB 126 или в исходном файле Chap_26.

26.1 Загрузка адреса переменной

Следующие инструкции дают начальный адрес локальной переменной:

```
L     P#name;
LAR1  P#name;
LAR2  P#name;
```

Здесь операнд *name* - это имя локальной переменной. В этих инструкциях межзонный указатель (area-crossing pointer) загружается в аккумулятор Accumulator 1, в адресный регистр AR1 и в адресный регистр AR2 соответственно. Межзонный указатель содержит адрес первого байта переменной. Если имя *name* не может быть идентифицировано однозначно как локальная переменная, тогда добавьте символ "#" перед именем, так что инструкция изменится, например, следующим образом:

```
L     P##name;
```

В таблице 26.1 представлены области, в которых, в зависимости от вида блока, может быть размещена переменная *name*.

В функциях FC адрес параметра блока не может быть загружен непосредственно в адресный регистр.

Поэтому здесь следует использовать аккумулятор Accumulator 1, например:

```
L     P#name;
LAR1  ;
```

В функциональных блоках (FB), скомпилированных с ключевым словом CODE_VERSION1 (то есть, как блок версии 1, без "мультиэкземплярного режима"), абсолютный адрес переменной экземпляра может быть загружен (load).

В функциональных блоках (FB), скомпилированных как блок версии 2, то есть, с возможностью "мультиэкземплярного режима", абсолютный адрес (относительно адресного регистра AR2) переменной экземпляра может быть загружен (load), если переменная относится к статическим локальным данным или является параметром блока. Если необходимо вычислить абсолютный адрес переменной в экземплярном блоке данных, Вы должны сложить внутризонный указатель (area-internal pointer) (только адрес) из AR2 с загруженным адресом переменной.

Таблица 26.1 Разрешенные области для размещения адресов переменных

Инструкция	Тип данных переменной *name*	Блок			
		OB	FC	FB V1	FB V2
L P#*name*	Временные локальные данные	x	x	x	x
	Статические локальные данные	-	-	x	x [1)]
	Параметр блока	-	x	x	x [1)]
LAR*n* P#*name*	Временные локальные данные	x	x	x	x
	Статические локальные данные	-	-	x	x [1)]
	Параметр блока	-	-	x	x [1)]

[1)] Адрес переменной, связанный с адресным регистром AR2

Пример 1:

Загрузить адрес переменной в адресный регистр AR1.

```
TAR2  ;
AD    DW#16#00FF_FFFF;
LAR1  P#name;
+AR1  ;
```

Здесь адрес в AR2 загружается в аккумулятор и добавляется к содержимому AR1 с помощью оператора +AR1. В результате чего адрес переменной *#name* загружается в адресный регистр AR1.

Пример 2:

Загрузить адрес переменной в аккумулятор Accumulator 1.

```
TAR2  ;
AD    DW#16#00FF_FFFF;
L     P#name;
+D    ;
```

В данном примере в результате выполнения данного фрагмента программы в аккумуляторе Accumulator 1 оказывается адрес переменной *#name*.

Сложение с внутризонным указателем (area-internal pointer) может быть пропущено, если указатель имеет значение P#0.0. Это для случая, если Вы не используете функциональный блок как локальный экземпляр.

Необходимо отметить, что при выполнении инструкции "LAR2 P#name" перезаписывается содержимое адресного регистра AR2, который применяется при использовании функциональных блоков с возможностью "мультиэкземплярного режима" как "базовый адресный регистр" при адресации экземплярных данных!

С помощью этих (LAR*n*) операторов загрузки (load) Вы можете организовать доступ только к одной переменной целиком, но не можете организовать доступ к отдельным компонентам массивов, структур или локальных экземпляров. Вы не можете с помощью этих операторов загрузки получить доступ к переменным в блоках глобальных данных или в адресных областях входов, выходов, периферийных входов/выходов и в адресных областях меркеров.

В таблице 26.2 показано, как рассчитываются адреса переменных типов INT и STRING в статических локальных данных и как использовать эти адреса. Если Вы используете образец программы в функциональном блоке, который Вы вызываете как локальный экземпляр, Вы должны прибавить базовый адрес к адресу переменной, как показано выше.

Таблица 26.2 Загрузка адреса переменной (примеры)

```
//Объявление переменных (функциональный блок не является локальным
//экземпляром!)
//Назначение переменных начинается с адреса P#0.0
VAR
  Field     : ARRAY[1..22] OF BYTE;  //переменная ARRAY занимает 22 байта
  Number    : INT := 123;            //переменная INT занимает 2 байта
  FirstName : STRING := 'Joane';     //переменная STRING занимает 5 байтов
END_VAR
```

`LAR1  P#Number;`	Загрузка начального адреса *Number* в AR1 Регистр AR1 теперь содержит P#DIX22.0
`L     W[AR1,P#0.0];`	Соответствует выражению L DIW 22 *или* L *Number*
`LAR1  P#FirstName;`	Загрузка начального адреса *FirstName* в AR1 Регистр AR1 теперь содержит P#DIX24.0
`L     B[AR1,P#0.0];`	Загрузка 1-го байта (максимальная длина строки символов) в аккумулятор Accu 1
`L     B[AR1,P#2.0];`	Загрузка 3-го байта (1-го байта данных) в аккумулятор Accu 1
`L     'John'` `T     D[AR1,P#2.0];`	Запись имени 'John' в строку символов
`L     4;` `T     B[AR1,P#1.0];`	Корректировка текущего размера строки символов (=4) Переменная *FirstName* теперь содержит 'John'

26.2 Хранение переменных

26.2.1 Хранение переменных в блоках глобальных данных

Редактор охраняет отдельные переменные в блоках данных в порядке их объявления. При этом должны соблюдаться следующие правила:

- Первая "битовая" переменной (с длиной в 1 бит) из непрерывной последовательности объявленных переменных размещается в бите 0 следующего байта; за первой битовой переменной размещаются следующие битовые переменные.
- "Байтовая" переменная (с длиной в 1 байт) из непрерывной последовательности объявленных переменных размещается в следующем байте.
- Переменные, имеющие размер одного "машинного" слова и имеющие размер двойного слова, всегда размещаются, начиная с границы слова, то есть в байте с четным адресом.
- Переменные типов DT и STRING также всегда размещаются, начиная с границы слова.
- Переменные типа ARRAY также всегда размещаются, начиная с границы слова, и заполняют область до следующей границы слова. Это касается также битовых и байтовых массивов. Элементы массивов простых типов сохраняются, как описано выше. Элементы массивов сложных типов размещаются, начиная с границы слова. Каждая размерность массива размещается с выравниванием как автономный массив.
- Переменные типа STRUCT также всегда размещаются, начиная с границы слова, и заполняют область до следующей границы слова. Это касается также "чисто" битовых и байтовых структур. Компоненты структур простых типов сохраняются, как описано выше. Элементы структур сложных типов размещаются, начиная с границы слова.

Группируя отдельные битовые переменные и компонуя байтовые переменные, Вы можете оптимизировать размещение данных в блоках данных.

На рисунке 26.1 Вы можете видеть один пример неоптимального и один пример оптимального хранения данных.

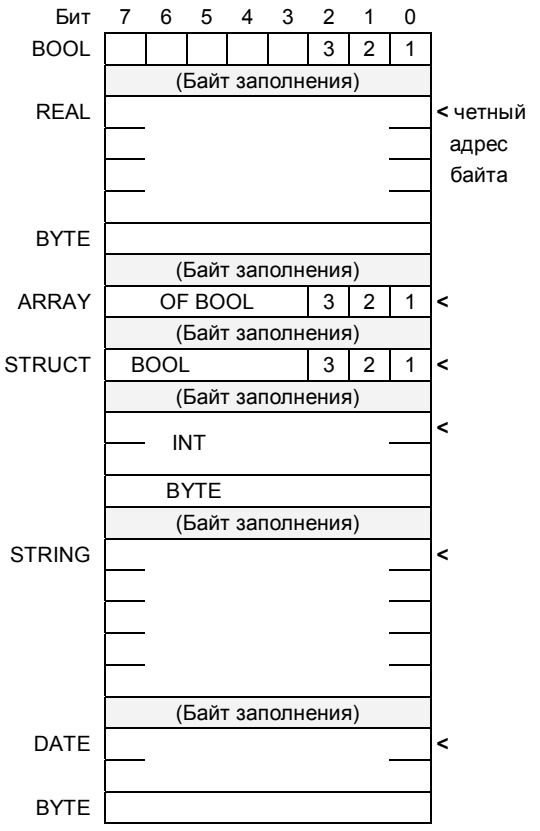
Необходимо отметить, что редактор всегда "заполняет" переменные типа ARRAY и типа STRUCT до следующей границы слова, что означает, что ни битовые, ни байтовые переменные не могут быть размещены для хранения в остающихся незанятыми областях памяти. Тем не менее, Вы можете оптимизировать размещение переменных внутри структуры.

Неоптимальное размещение данных

```
DATA_BLOCK StorageNonOptimized
STRUCT
  Bit1      : BOOL;
  Bit2      : BOOL;
  Bit3      : BOOL;
  Real1     : REAL;
  Byte1     : BYTE;
  Bit_field : ARRAY [1..3] OF BOOL;
  Structure : STRUCT
    S_Bit1  : BOOL;
    S_Bit2  : BOOL;
    S_Bit3  : BOOL;
    S_Int1  : INT;
    S_Byte  : BYTE;
  END_STRUCT;
  Character : STRING[3];
  Date1     : DATE;
  Byte2     : BYTE;
END_STRUCT
BEGIN
END_DATA_BLOCK
```

Оптимальное размещение данных

```
DATA_BLOCK StorageNonOptimized
STRUCT
  Bit1      : BOOL;
  Bit2      : BOOL;
  Bit3      : BOOL;
  Byte1     : BYTE;
  Real1     : REAL;
  Bit_field : ARRAY [1..3] OF BOOL;
  Structure : STRUCT
    S_Bit1  : BOOL;
    S_Bit2  : BOOL;
    S_Bit3  : BOOL;
    S_Byte  : BYTE;
    S_Int1  : INT;
  END_STRUCT;
  Character : STRING[3];
  Byte2     : BYTE;
  Date1     : DATE;
END_STRUCT
BEGIN
END_DATA_BLOCK
```



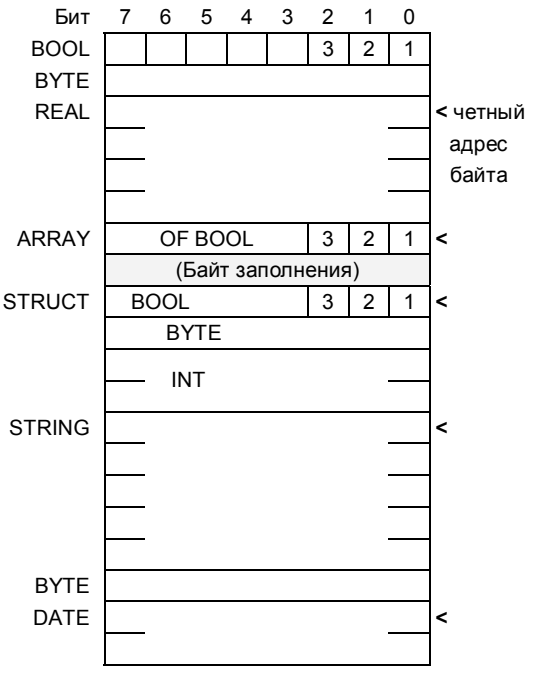


Рис. 26.1 Примеры размещения данных в блоке данных

26.2.2 Хранение переменных в блоках экземплярных данных

Редактор сохраняет отдельные переменные в экземплярах DB, размещая их в следующем порядке:

- Входные параметры;
- Выходные параметры;
- Входные/Выходные параметры;
- Локальные переменные (в том числе и переменные локальных экземпляров).

Порядок расположения переменных в соответствующей области памяти соответствует порядку их объявления. Каждая объявленная область (для одинаково объявленных переменных) начинается на границе "машинного" слова, то есть с байта с четным адресом. Внутри таких областей отдельные переменные располагаются, как описано в предыдущем разделе (как в глобальном блоке данных).

На рисунке 26.2 показан пример размещения переменных в экземплярном блоке данных.

26.2.3 Хранение переменных в области временных локальных данных

Размещение переменных в области временных локальных данных (в L-стеке) соответствует размещению в блоке глобальных данных. Размещение данных всегда начинается с байта 0 (относительный адрес).

Необходимо отметить, что в организационном блоке первые 20 байтов должны быть заняты стартовой информацией. Даже если Вы не используете стартовую информацию, первые 20 байтов должны быть объявлены (например, массивом, имеющим размер 20 байтов.).

Сам редактор также использует локальные данные, например, при передаче параметров в вызываемый блок. Редактор применяет объявленные с символьными именами временные локальные данные и временные локальные данные, которые он использует для выполнения своих функций, в порядке их объявления или в порядке, определяющем их использование. Временные локальные данные с абсолютной адресацией не используются здесь, так как при этом может произойти "перекрытие" данных (замена значений), если Вы не контролируете, какие именно данные использует редактор. Если Вы хотите или должны организовать доступ к локальным данным, используя абсолютную адресацию, то Вы, например, можете оставить свободным массив (field) в первой позиции списка объявленных локальных данных для задания требуемого числа байтов (слов, двойных слов). Затем Вы сможете обращаться, используя абсолютную адресацию, к данной области. Что касается организационных блоков, то здесь Вы должны будете разместить указанный массив после первых 20 байтов, которые должны быть отведены под стартовую информацию блока.

```
FUNCTION_BLOCK StorageExample
VAR_INPUT
  E_Bit1      : BOOL;
  E_Bit2      : BOOL;
  E_Bit3      : BOOL;
  E_Real1     : REAL;
END_VAR
VAR_OUTPUT
  A_BYTE 1    : BYTE;
  A_BYTE 2    : BYTE;
  A_BYTE 3    : BYTE;
END_VAR
VAR_IN_OUT
  D_BYTE 1    : BYTE;
  D_Bit1      : BOOL;
  D_Bit2      : BOOL;
  D_Bit3      : BOOL;
END_VAR
VAR
  Date1     : DATE;
  Character : STRING[3];
  Bit_field : ARRAY [1..3] OF BYTE;
END_VAR
BEGIN
END_FUNCTION_BLOCK
```

```
ORGANIZATION_BLOCK Cycle
VAR_TEMP
  SInfo   : ARRAY [1..20] OF BYTE;
  LData   : ARRAY [1..16] OF BYTE;
  Temp1   : STRING [36];
  Temp2   : BOOL;
  Temp3   : BOOL;
  Temp4   : BOOL;
  Temp5   : BYTE;
  Temp6   : INT;
END_VAR
BEGIN
//Доступ по абсолютным адресам
  ...
  T    LW 20;
  ...
  =    L 22.2;
//Доступ по символам
  T    Temp6;
  =    Temp3;
  T    LData[16];
//Загрузка адреса переменной
  L    P#Temp1;
  LAR1 P#Temp2;      ...
END_ORGANIZATION_BLOCK
```

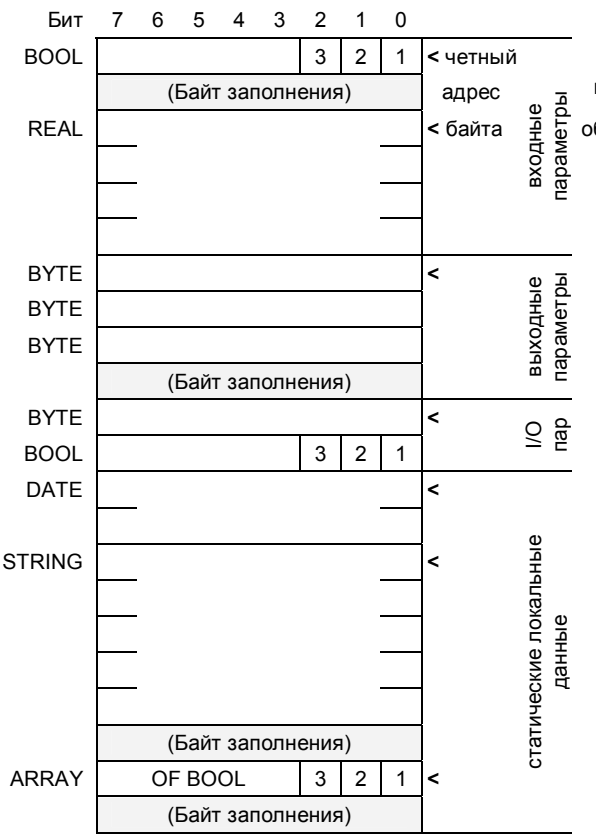


Рис. 26.2 Пример размещения данных в экземплярном DB

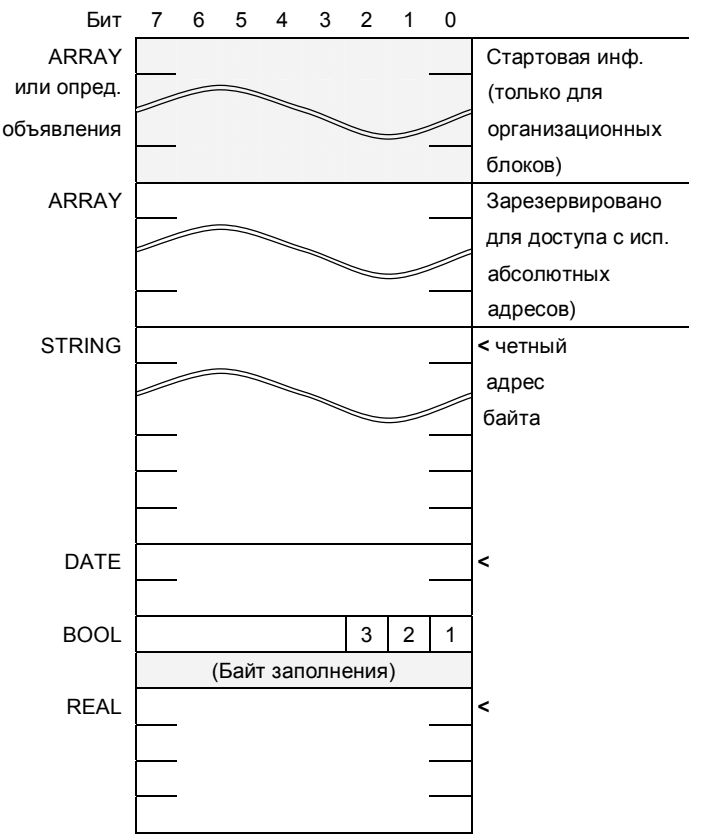


Рис. 26.3 Пример размещения данных L-стека в организационных блоках

На рисунке 26.3 показан пример размещения временных локальных данных в организационном блоке.

Массив "LData" располагается сразу после стартовой информации с байта LB 20 и простирается, например, вплоть до байта LB 35. Редактор не занимает эту область своими собственными временными данными, поэтому Вы можете использовать данную область для абсолютной адресации.

В функциях и функциональных блоках стартовая информация опускается. Если Вам необходимы временные локальные данные для абсолютной адресации, используйте массив как первую переменную в данных блоках; при этом начало массива будет располагаться в байте LB 0.

26.3 Сохранение данных при передаче параметров

Параметры блоков хранятся по-разному в функциях и функциональных блоках. Вам как пользователю нет необходимости вникать в это; Вы программируете параметры для блоков обоих типов одинаково. Тем не менее, эта разница чрезвычайно важна с точки зрения вопроса прямого доступа к параметрам блока.

26.3.1 Доступ к параметрам в функциях

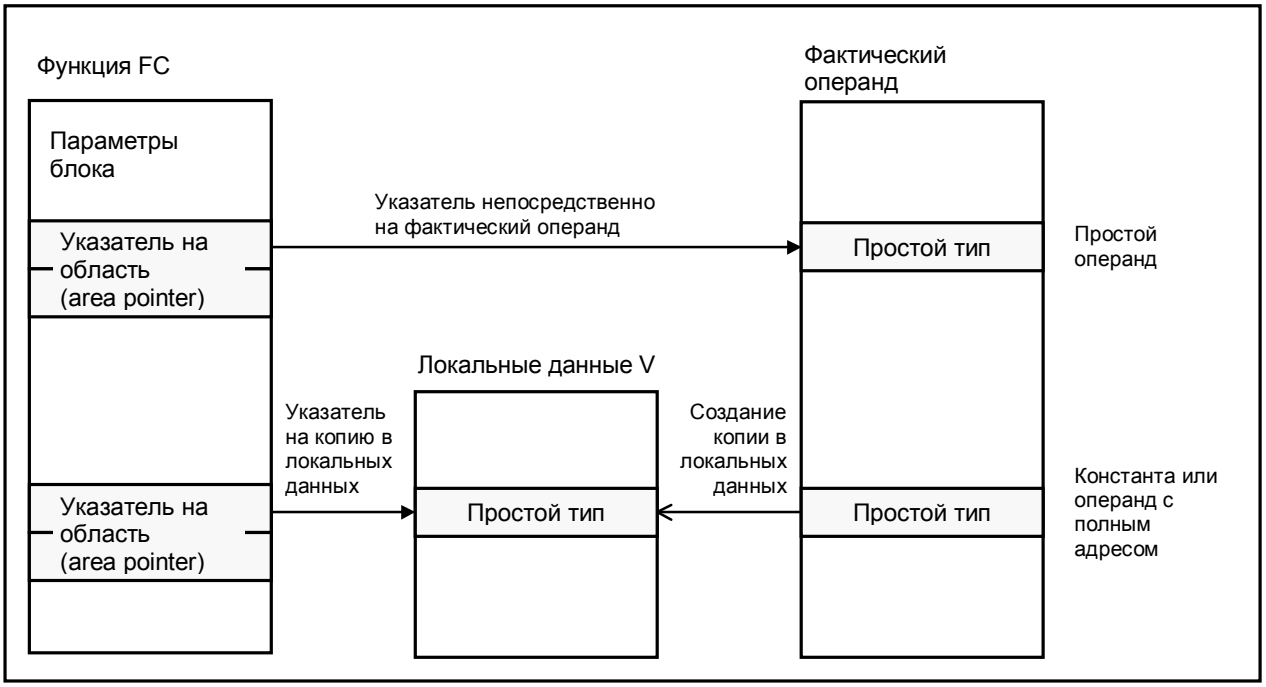
Редактор сохраняет данные о параметрах блоков для функции (имеются в виду параметры функции, относящиеся к типу "параметр блока"; далее по тексту - "параметры функции") в виде межзонных указателей области (area pointer) в коде блока в соответствующем операторе вызова; таким образом, каждый параметр функции требует для хранения одно двойное слово памяти. В зависимости от типа данных и от объявленного типа указатель указывает на собственно фактический параметр, на копию фактического параметра во временных локальных данных вызывающего блока (созданную редактором) или на указатель во временных локальных данных вызывающего блока, который в свою очередь указывает на собственно фактический параметр (см. табл. 26.3). Исключение: для следующих типов параметров: TIMER, COUNTER и BLOCK_xx указателем является 16-разрядное число, расположенное в левом слове параметра блока (функции).

Таблица 26.3 Доступ к параметрам функции

Тип данных	INPUT (входной)	IN_OUT (входной-выходной)	OUTPUT (выходной)
	Параметр является указателем на область (area pointer) для:		
Простой тип	значения	значения	значения
Сложный тип	DB-указателя	DB-указателя	DB-указателя
TIMER, COUNTER и BLOCK	номера	недопустимый	недопустимый
POINTER	DB-указателя	DB-указателя	DB-указателя
ANY	ANY-указателя	ANY-указателя	ANY-указателя

В случае простых типов данных параметр блока указывает непосредственно на фактический параметр (см. 26.4).

Указатель на фактический операнд или его значение



Указатель на другой указатель

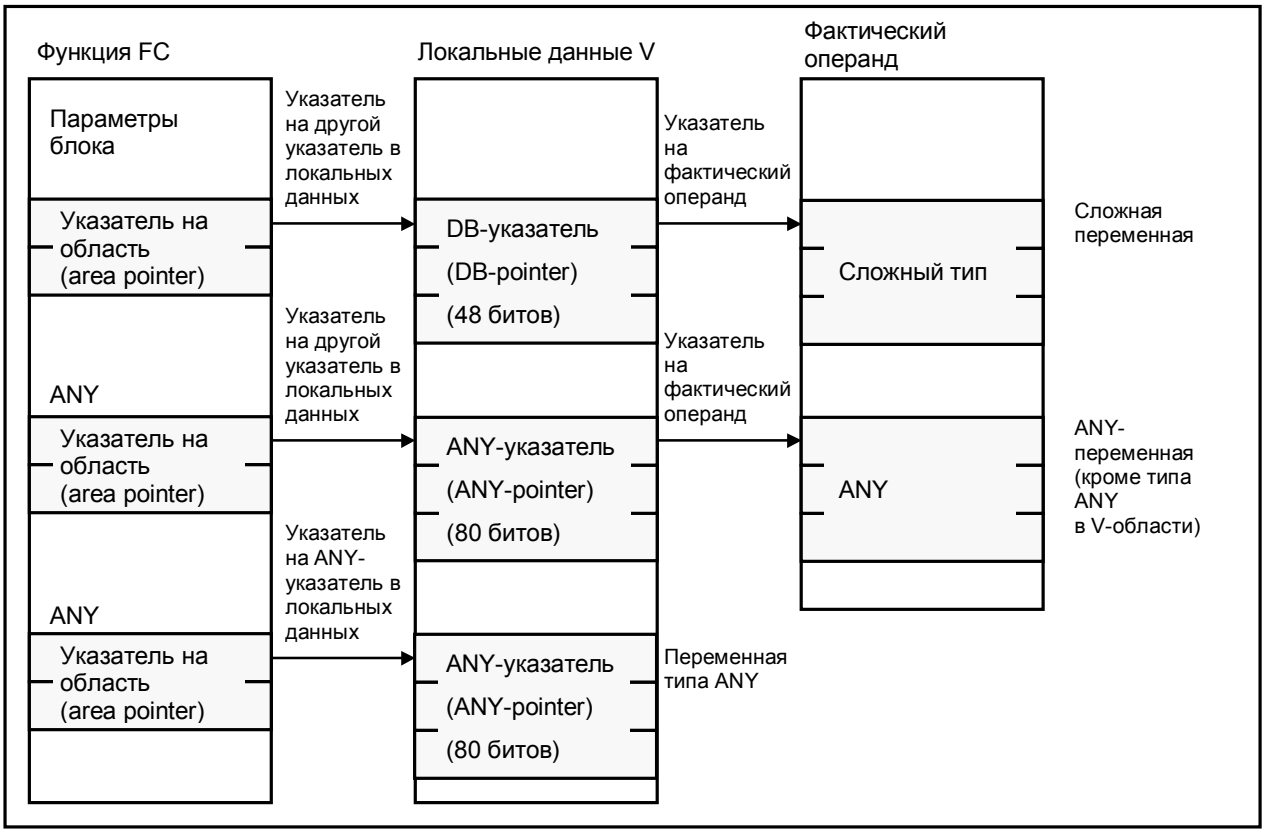


Рис. 26.4 Передача параметров в функции

Тем не менее, с помощью указателя на область (area-pointer) как параметра блока Вы не можете получить доступ ни к каким константам или к операндам, расположенным в блоках данных. По этой причине еще на этапе компилирования редактор копирует константы или фактические параметры, размещенные в блоке данных (операнд с полным адресом), в область временных локальных данных вызывающего блока и "направляет" указатель (area-pointer) на эти копии данных. Подобная область параметров называется "V" (временные локальные данные "предыдущего" блока или V-область).

Перед текущим вызовом функции FC имеет место копирование в V-область данных, полученных при вызове функции: входных параметров и входных-выходных параметров, а после вызова функции имеет место копирование возвращенных входных-выходных параметров, выходных параметров, а также значения функции. Поэтому здесь действует правило, согласно которому Вы можете только проверить входные параметры и сделать запись в выходные параметры. Если Вы передаете значение во входной параметр, используемый в операнде с полным адресом, то это значение будет сохранено во временных локальных данных вызывающего блока и будет в дальнейшем "забыто" (утеряно), так как в дальнейшем не будет никакого "обратного" копирования в "фактическую" ("actual") переменную в блоке данных. Похожая история и с загрузкой соответствующего выходного параметра: так как никакого копирования не происходит из фактической переменной из блока данных в V-область, Вы сможете загрузить (load) лишь "неопределенное" значение из V-области в данном случае.

Из-за операции копирования выходной параметр, как и значение функции, **должен** быть перезаписан определенным значением при простом типе данных в блоке, если операнд с полным адресом рассматривается или может рассматриваться как фактический параметр. Если Вы не назначаете значения для выходного параметра (например, из-за преждевременного выхода из блока или из-за обхода в программе соответствующей инструкции), значение во временных локальных данных также не будет инициализировано. Оно будет иметь значение, которое имело место перед вызовом блока. В дальнейшем выходной параметр будет перезаписан этим "неопределенным" значением.

В случае сложных типов данных, таких как DT, STRING, ARRAY, STRUCT, а также UDT, фактические параметры размещаются или в блоке данных, или в V-области. Так как указатель на область (area-pointer) не может обеспечить доступ к фактическим параметрам в блоке данных, то редактор создает DB-указатель в V-области во время процесса компиляции. Этот DB-указатель указывает на фактический параметр в блоке данных (если номер DB не равен 0), или указывает на V-область (если номер DB равен 0). DB-указатели для всех объявленных типов в V-области создаются до фактического вызова функции FC.

В случае типов параметра, таких как TIMER, COUNTER и BLOCK_xx, параметр блока содержит номер (16-разрядное число, располагающееся с выравниванием влево в 32-разрядном параметре) вместо указателя на область (area-pointer).

В случае, если тип параметра POINTER, то параметр обрабатывается точно также, как данные сложных типов.

В случае, если тип параметра ANY, то редактор создает 10-байтовый ANY-указатель в V-области, который указывает на любую (ANY)

переменную. Такой же подход распространяется на сложные типы данных.

Редактор делает исключение в случае, если Вы используете для параметра блока типа ANY фактический параметр, который размещается в области временных локальных данных и относится к типу данных ANY. В этом случае редактор не создает никаких ANY-указателей, а вместо этого направляет "указатель на область" (area-pointer) (параметр блока) прямо на фактический параметр (в этом случае ANY-указатель может быть модифицирован в процессе выполнения программы, см. раздел 26.3.3 "Переменная" ANY-указатель).

26.3.2 Хранение параметров в функциональных блоках

Редактор сохраняет параметры для функционального блока в его экземплярном блоке данных. При вызове функционального блока редактор создает код, который обеспечивает копирование значений фактических параметров в экземплярный блок данных перед "фактическим" вызовом функционального блока, а затем копирует значения этих параметров назад из экземплярного блока в фактические параметры после вызова блока. При просмотре скомпилированного блока Вы не можете видеть этот код, создаваемый редактором, но Вы можете заметить его присутствие по косвенным признакам - по загрузке памяти.

В экземплярном блоке данных параметры блока хранятся либо как значения (в виде 16-разрядных чисел), либо как указатель, указывающий на фактический параметр (см. табл. 26.4). Если параметр хранится как значение, то требуемая для этого память зависит от типа данных параметра блока. Так, число занимает два байта, указатели занимают 6 байтов (DB-указатели) или 10 байтов (ANY-указатели).

Таблица 26.4 Доступ к параметрам функционального блока

Тип данных	INPUT (входной)	IN_OUT (входной-выходной)	OUTPUT (выходной)
Простой тип	значение	значение	значение
Сложный тип	значение	DB-указатель	значение
TIMER, COUNTER и BLOCK	номер	недопустимый	недопустимый
POINTER	DB-указатель	DB-указатель	недопустимый
ANY	ANY-указатель	ANY-указатель	недопустимый

Связь между параметрами блоков, назначениями экземплярных данных и фактическими параметрами показана на рис. 26.5. При копировании значений фактических параметров сложных типов в экземплярный блок данных (входные параметры) и при копировании значений параметров из экземплярного блока в фактические параметры (выходные параметры) редактор использует системную функцию SFC 20 BLKMOV, параметры которой он встраивает в область временных локальных данных вызывающего блока.

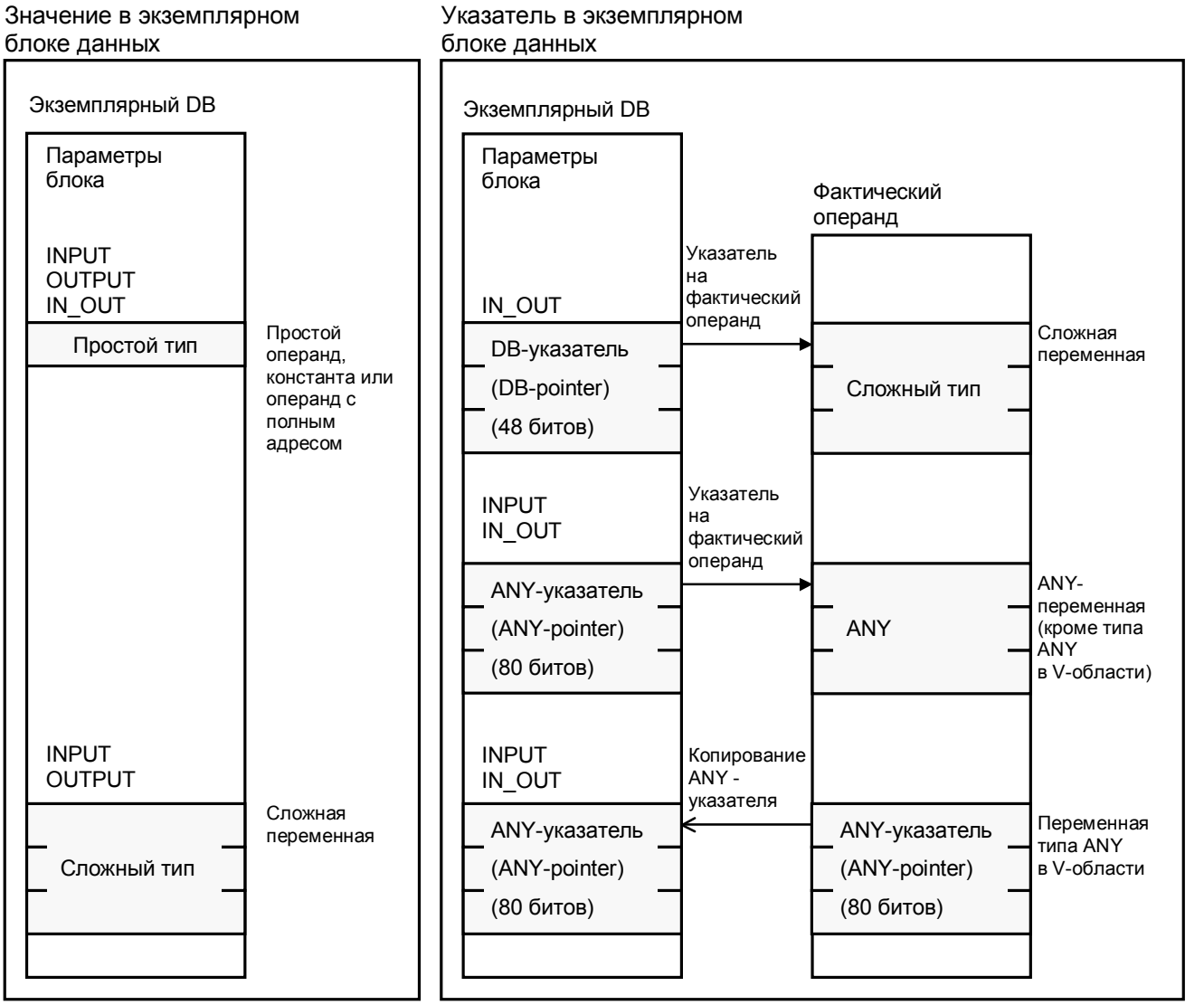


Рис. 26.5 Передача параметров при использовании функциональных блоков

Перед текущим вызовом функционального блока и после такого вызова производится копирование и передача параметров блока в экземплярный блок данных с использованием последовательности операторов кода, вносимого редактором при компиляции. Перед текущим вызовом функционального блока производится копирование входных и входных-выходных параметров блока, а после вызова функционального блока производится копирование входных/выходных и выходных параметров блока. Поэтому здесь действует правило, согласно которому Вы можете только проверить входные параметры и сделать запись в выходные параметры. Если Вы, к примеру, передаете (новое) значение во входной параметр, то текущее значение фактического параметра будет потеряно. Если Вы загружаете (load) выходной параметр, это значит, что Вы загружаете (старое) значение из экземплярного блока данных, а не значение фактического параметра.

Так как параметры блока хранятся в экземплярном блоке данных, то нет

необходимости инициализировать их всякий раз при вызове функционального блока. Если не была проведена инициализация параметров блока, то в программе будут использоваться "старые" значения входных и входных-выходных параметров блока, а что касается выходных параметров блока, то Вы сможете считать их значения, относящиеся к другой точке выполняемой программы. Вне функционального блока Вы можете получить доступ к переменным, которые хранятся в экземплярном блоке данных, таким же способом, как и к переменным в блоке глобальных данных (с использованием символьного имени блока данных и имени параметра блока). Это касается также и статических локальных данных.

Если Вы используете временную локальную переменную типа ANY в ANY-параметре, то редактор копирует содержание этой переменной в ANY-указатель (в параметр блока) в экземплярном блоке данных.

26.3.3 "Переменная" ANY-указатель (ANY-pointer)

ANY-параметры могут быть параметризированы только такими областями данных или переменными, которые уже определены на этапе компиляции.

Пример:

Копирование переменной в область данных с помощью системной функции SFC 20 BLKMOV:

```
CALL SFC 20 (
   SRCBLK  := "ReceiveMailbox".Data,
   DSTBLK  := P#DB63.DBW0.0 BYTE 8,
   RET_VAL := SFC20Error);
```

Во время работы программы можно модифицировать или переопределить переменную или область данных, так как редактор использует фиксированный ANY-указатель на фактический параметр во временных локальных данных (см. далее в данном разделе).

Но есть здесь одно исключение, которое касается случая, когда фактический параметр располагается во временных локальных данных и имеет тип данных ANY. Тогда редактор не создает никакого ANY-указателя, вместо этого он интерпретирует эту ANY-переменную как ANY-указатель на фактический параметр. Это означает, что ANY-переменная должна иметь такую же структуру как ANY-указатель.

Во время работы программы Вы можете модифицировать такие ANY-переменные во временных локальных данных и тем самым специфицировать другие фактические параметры, соответствующие ANY-параметрам.

Для применения таких "переменных" ANY-указателей Вы должны выполнить следующее:

- Применение временной локальной переменной типа ANY (имя ANY-переменной может быть выбрано произвольным в допустимых рамках

для локальных переменных блока):

```
VAR_TEMP
   ANY_POINTER  : ANY;
END_VAR
```

- Инициализация ANY-переменной значениями:

Адрес, известный во время выполнения программы	Адрес, неизвестный во время выполнения программы
	`LAR1 P#ANY_POINTER;`
`L W#16#1002;`	`L W#16#1002;`
`T LW 0;`	`T LW[AR1,P#0.0];`
`L 16;`	`L 16;`
`T LW 2;`	`T LW[AR1,P#2.0];`
`L 63;`	`L 63;`
`T LW 4;`	`T LW[AR1,P#4.0];`
`L P#DBB0.0;`	`L P#DBB0.0;`
`T LD 6;`	`T LD[AR1,P#6.0];`

- Инициализация ANY-параметра, например, в блоке SFC 20:

```
CALL SFC 20 (
     SRCBLK  := "ReceiveMailbox".Data,
     DSTBLK  := P#DB63.DBW0.0 BYTE 8,
     RET_VAL := SFC20Error);
```

Данная процедура не ограничивается только функциональным блоком SFC 20 BLKMOV; Вы можете применять ее в отношение всех ANY-параметров любых блоков.

Пример:

Пусть необходимо создать блок, копирующий одну область данных в другую. Исходная область и область назначения должны быть параметризируемыми. Мы используем функцию SFC 20 BLKMOV для копирования.

Блок - функция FC - имеет следующие параметры:

```
VAR_INPUT
     QDB   : INT; //Исходный блок данных
     SSTA  : INT; //Начальный адрес исходных данных
     NUMB  : INT; //Число байтов
     DDB   : INT; //Блок данных назначения
     DSTA  : INT; //Начальный адрес области назначения
END_VAR
```

Возвращаемое значение функции должно содержать сообщение об ошибках функции SFC 20. Кроме того бит слова состояния BR устанавливается в состояние "0" в случае ошибки.

Для работы с локальными данными блока достаточно использовать две ANY-переменные: одну - как указатель для исходной области, а вторую - как указатель для области назначения:

```
VAR_TEMP
     SANY  : ANY; //ANY-указатель исходной области
     DANY  : ANY; //ANY-указатель области назначения
END_VAR
```

Так как мы знаем адреса ANY-указателей во временных локальных данных, мы можем запрограммировать их, используя абсолютную адресацию, например, при подготовке ANY-указателя области назначения:

```
L    W#16#1002;    // тип BYTE
T    LW 0;
L    NUMB;         // число байтов
T    LW 2;
L    QDB;          // исходный DB
T    LW 4;
L    SSTA;         // начало исходных данных
SLD  3;
OD   DW#16#8400_0000;
T    LD 6;
```

Указатель области назначения с начальным адресом LB 10 программируется аналогичным образом.

Остается только инициализировать блок SFC 20:

```
CALL SFC 20 (
  SRCBLK  := SANY,
  DSTBLK  := DANY,
  RET_VAL := RET_VAL);
```

Значение функции RET_VAL для SFC 20 инициализируется значением RET_VAL нашей функции FC.

Полностью данный маленький пример Вы можете найти на дискете, прилагаемой к данной книге (функция FC 47 в программе "General Examples" ("Общие примеры")).

Таким образом, ANY-указателю может быть назначено любое значение; например, Вы можете менять тип в слове 2 или указатель на область (area pointer); так что в принципе Вы можете адресовать любые переменные и области данных, например, область меркеров.

Примечание:

Если ANY-указатель, размещенный во временных локальных данных, указывает на переменную, которая также размещена во временных локальных данных вызывающего блока, то V-область должна быть

введена как область-операнд ("operand area"), так как с точки зрения вызванного блока данная переменная размещена во временных локальных данных предшествующего ("predecessor") блока.

26.4 Краткое описание примера "Message Frame Example" (Пример фрейма сообщения)

Описываемая далее программа-пример углубит Ваше понимание процесса работы со сложными переменными. Программа состоит из различных блоков, каждый из которых иллюстрирует определенный аспект данной темы. Технологические функции, заложенные в примерах, таких, например, как "Generate_Frame" ("Создание фрейма") и "Checksum" ("Контрольная сумма"), предназначены только для того, чтобы сделать яснее способы решения проблемы, и, где это необходимо, изложены лишь вкратце.

Вы можете найти эту программу на дискете, приложенной к данной книге, под заголовком "Message Frame Example" ("Пример фрейма сообщения"). В книге примеры представлены пояснительными текстами и рисунками.

Программа-пример состоит из следующих разделов:

- Message frame data (UDT 51, UDT 52, DB 61, DB 62, DB 63) (раздел "Фрейм сообщения" иллюстрирует вопросы обработки определяемых пользователем структур данных);
- Clock Check (FC 61)
  (раздел "Контроль времени" иллюстрирует вопросы использования системных и стандартных блоков);
- Checksum (FC 62)
  (раздел "Контрольная сумма" иллюстрирует вопросы использования прямого доступа к переменным);
- Generate frame (FB 51)
  (раздел "Создание фрейма" иллюстрирует вопросы использования системной функции SFC 20 BLKMOV с фиксированными адресами);
- Store frame (FB 52)
  (раздел "Хранение фрейма сообщения" иллюстрирует вопросы использования "переменной" ANY-указатель);
- Date conversion (FC 63)
  (раздел "Преобразование даты" иллюстрирует вопросы обработки переменных сложных типов данных).

Пример "Message frame data"
("Фрейм сообщения")

Пример "Message frame data" ("Фрейм сообщения") показывает, как Вы можете определять часто встречающиеся структуры данных, таких как Ваша собственная структура, то есть, относящаяся к UDT-типу, и как использовать данный тип данных при объявлении переменных и параметров.

Программа-пример организована как база данных для входящих и исходящих сообщений: "почтовый ящик" исходящих сообщений ("send mailbox") со структурой фрейма сообщения, "почтовый ящик" входящих сообщений ("receive mailbox") с такой же структурой и входной ("receive") кольцевой буфер для промежуточного хранения входящих фреймов сообщений (см. рис. 26.6).

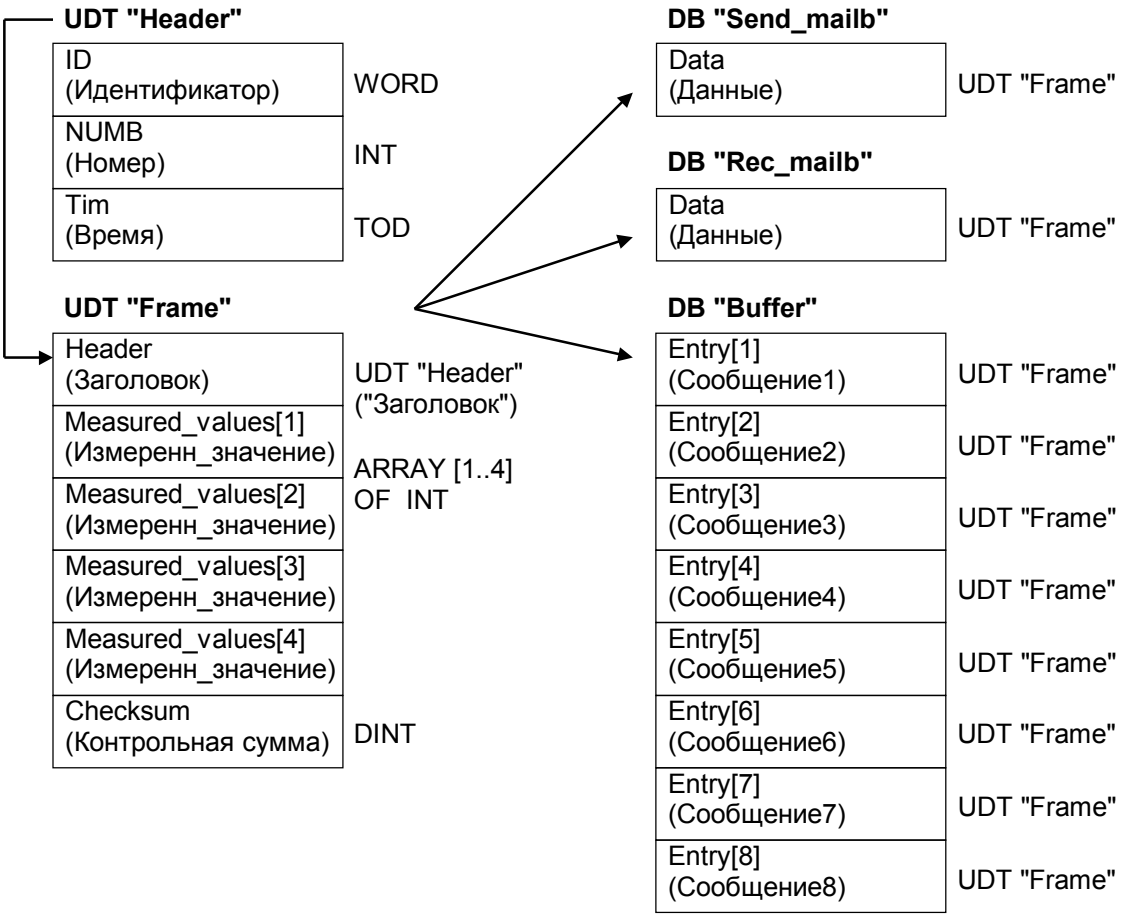


Рис. 26.6 Структура данных для примера "Message frame data" ("Фрейм сообщения")

Так как структура данных сообщения встречается часто, то мы можем определить пользовательский тип данных (UDT) с именем *Frame*. Фрейм содержит заголовок (frame header); мы можем также дать имя структуре заголовка фрейма. "Почтовый ящик" исходящих сообщений ("send mailbox") и "почтовый ящик" входящих сообщений ("receive mailbox") должны представлять собой блоки данных, каждый из которых будет содержать переменные со структурой *Frame.* Наконец, кольцевой буфер для промежуточного хранения входящих фреймов сообщений должен представлять из себя блок данных с массивом из 8 элементов, которые также имеют структуру данных *Frame*.

Сначала мы определяем структуру UDT *Header* , затем - структуру UDT *Frame*.

Структура UDT *Frame* содержит структуру UDT *Header*, массив *Measured_values* из 4 элементов и переменную *Checksum*. Все компоненты при инициализации получают значение 0.

И в блоке данных "Send_mailb", и в блоке данных "Rec_mailb" определена переменная *Data* со структурой *Frame*. Переменные теперь могут быть отдельно инициализированы в разделах инициализации блоков данных.

В нашем примере компонент *ID* в каждом случае получает свое значение, отличающееся от полученного при инициализации пользовательского типа UDT. Блок данных "Buffer" содержит переменную *Entry* в виде массива, состоящего из 8 элементов, каждый из которых имеет структуру *Frame*. В этом случае также отдельные компоненты структуры могут быть инициализированы различными значениями в разделе инициализации (например: Entry[1].Header.Numb := 1).

Данный пример, в свою очередь используемый в последующих примерах, содержит следующие объекты:

UDT 51 - пользовательский тип данных Header,

UDT 52 - пользовательский тип данных Frame,

DB 61 - блок данных "Send_mailb" ("почтовый ящик" исходящих сообщений),

DB 62 - блок данных "Rec_mailb" ("почтовый ящик" входящих сообщений),

DB 63 - блок данных "Buffer".

Пример "Clock Check"
("Контроль времени")

Пример "Контроль времени" показывает, как использовать системные и стандартные блоки (для операций проверки наличия ошибок, для операций копирования из библиотеки, для переименования).

Функция "Clock_check" используется для вывода значения суточного времени, считываемого из встроенных в CPU часов реального времени, в виде значения функции. Для этих целей нам потребуется системная функция SFC 1 READ_CLK, которая считывает дату и время суток из встроенных часов реального времени, в формате даты DATE_AND_TIME или DT. Так как нам нужно только значение времени суток, то нам потребуется еще IEC-функция FC 8 DT_TOD. Данная функция позволяет выбрать значение суточного времени в формате TIME_OF_DAY или TOD из данных формата DT (см. рис. 26.7).

Спецификация времени для часов реального времени хранится в блоке данных "Data66". Данная информация нам еще потребуется для использования в примере "Date conversion" ("Преобразование даты"). Если бы в дальнейшем нам не потребовалась данная информация, мы могли бы также объявить временную локальную переменную вместо переменной *CPU_Tim*.

Проверка на наличие ошибок (Error evaluation)

Системные функции сообщают об ошибках посредством бита

двоичного результата BR и с помощью возвращаемого значения функции RET_VAL. О том что произошла ошибка, свидетельствует значение BR = "0"; также при этом значение функции RET_VAL становится отрицательным - устанавливается бит 15. Стандартные IEC-функции сообщают об ошибках только посредством двоичного результата BR. В примере представлены два способа проверки наличия ошибок. В функции "Clock_check" двоичный результат BR сначала установлен в "1". В случае появления ошибки двоичный результат BR сбрасывается в "0". При этом на выход поступает неверное значение суточного времени. После вызова функции "Clock_check" Вы также можете проверить наличие ошибок, используя проверку бита двоичного результата BR.

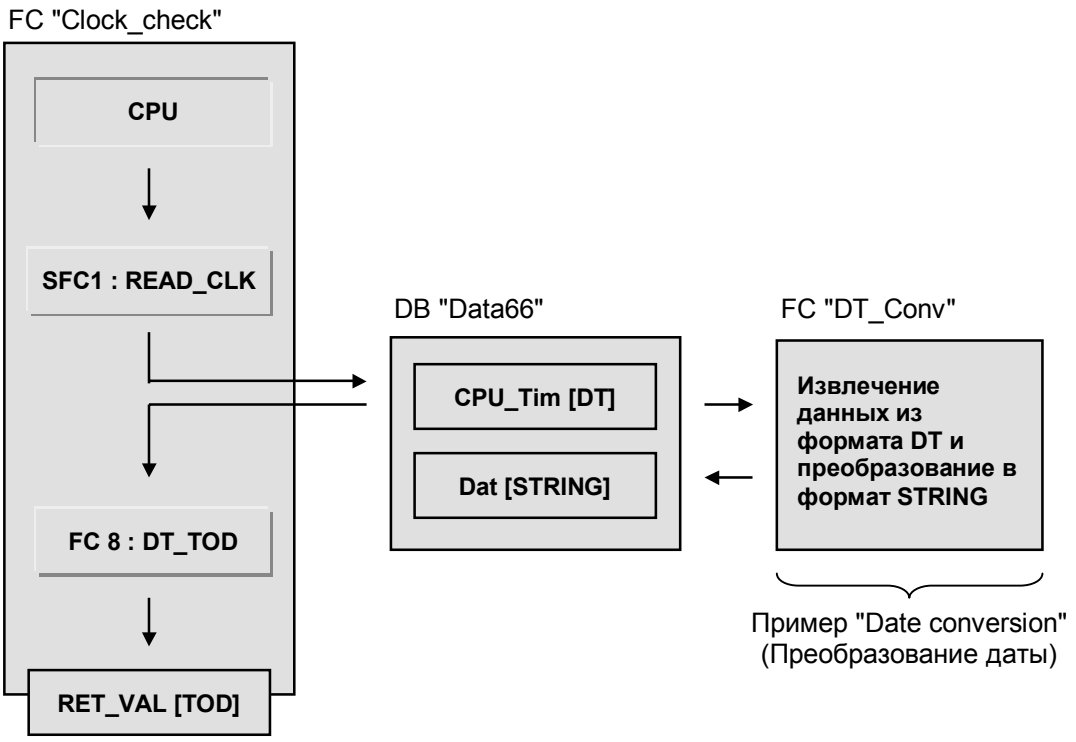


Рис. 26.7 Пример "Clock check" ("Проверка времени")

Программирование системных функций в автономном (offline) режиме

Перед компилированием программы или перед вызовом ее для корректировки в инкрементном режиме "автономная" программа пользователя должна включать в себя системную функцию SFC 1 и стандартную функцию FC 8. Обе функции входят в комплект поставки системы STEP 7. Вы можете найти эти функции в библиотеке блоков из поставляемого программного обеспечения. (Для системных функций, встроенных в CPU, библиотека блоков (block library) содержит описание интерфейса вместо собственно программных кодов этих функций. Эти системные функции могут быть вызваны в автономном (offline) режиме с помощью упомянутого описания интерфейса; описание интерфейса не пересылается в CPU. Загружаемые функции, такие, как IEC-функции хранятся в библиотеке как исполняемые программы.)

Выбрав следующие опции: *File -> Open (Файл -> Открыть)* в Simatic Manager, Вы можете выбрать "стандартную библиотеку" *Standard Library* и открыть библиотеку *System Function Blocks (Системные функциональные блоки)*. В разделе блоки Вы можете найти все описания интерфейсов для системных функций. Если Вы еще не открыли окно Вашего проекта, Вы можете вывести оба окна одно под другим, выбрав следующие опции: *Window -> Arrange -> Vertically (Окно -> Настроить -> Вертикально),* и с помощью манипулятора "мышь" "перетащить" выбранные системные функции в Вашу программу, используя метод "Drag-n-Drop" (Отметьте требуемую функцию SFC с помощью мыши, удерживайте нажатой левую кнопку мыши, "перетащите" захваченный объект на раздел *Blocks* или на его открытое окно и отпустите кнопку). Вы можете скопировать стандартную функцию FC 8 таким же образом. Эту функцию Вы можете найти в библиотеке *IEC Function Blocks (Функциональные блоки стандарта IEC).* Функция FC 8 является загружаемой функцией, следовательно, она занимает часть пользовательской памяти, в отличие от SFC 1.

Если стандартный блок вызывается из каталога программных элементов редактора (Editor's ProgramElement Catalog) из раздела "Libraries" во время инкрементного программирования, то этот блок автоматически копируется в раздел блоков *Blocks* и вводится в таблицу символов.

Переименование стандартных функций

Пользователь имеет возможность переименовывать стандартные функции. Для этого Вы можете выделить стандартную функцию (например, FC 8) в окне проекта и снова щелкнуть кнопкой мыши на имени выбранного объекта. Вокруг имени появится рамка, и в ней Вы можете задать новое имя (например, FC 98). Если теперь Вы нажмете F1, пока выделена стандартная функция (переименованная в FC 98), то Вы получите контекстную справку (функция Help), касающуюся изначальной стандартной функции FC 8.

Если при выполнении операции копирования окажется, что существуют два одноименных блока, то появится диалоговое окно, в котором следует выбрать один из двух вариантов выполнения копирования: копирование с переименованием блока или копирование с перезаписью существующего блока.

Символьная адресация функций

В таблице символов пользователь может назначить символьные имена для системных и стандартных функций, так что эти функции могут быть адресованы в дальнейшем посредством символьных имен. Пользователь свободен в выборе имен в допустимых пределах, регламентированных для имен блоков. В примере символьные имена были заданы для каждого блока (для лучшей идентификации).

Пример "Checksum"
("Контрольная сумма")

Данный пример разъясняет использование прямого доступа к параметру блока типа ANY с вычислением адреса переменной и с использованием косвенной адресации (см. 26.8).

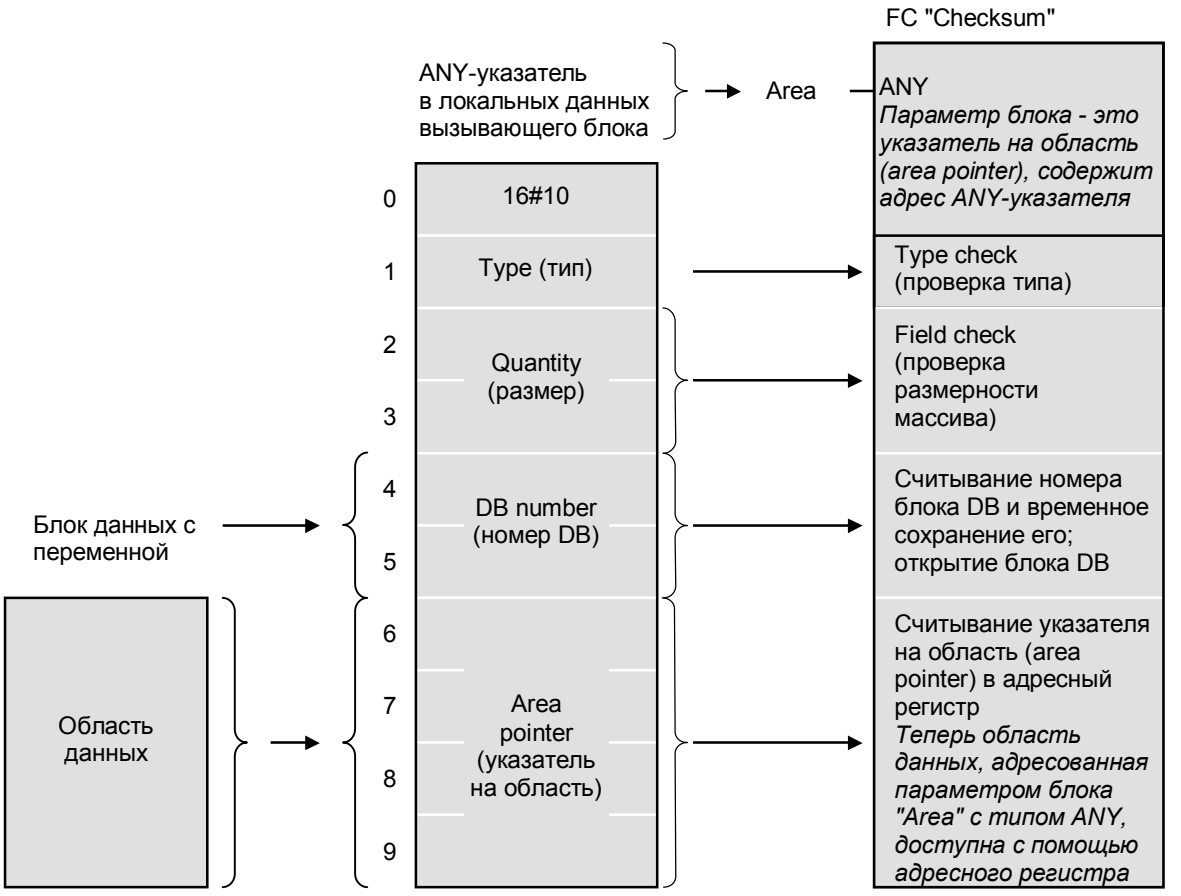


Рис. 26.8 Пример "Checksum" ("Контрольная сумма")

Контрольная сумма должна быть сгенерирована исходя из структуры данных простым сложением всех байтов без учета корректности выполнения (переполнение, нарушение диапазона значений для DINT).

Все структуры данных (STRUCT и UDT) обрабатываются редактором как массивы с байтовыми элементами (размер элемента равен 1 байту), если применяются в параметре блока с типом ANY. Следовательно, с помощью этой программы Вы можете сгенерировать контрольную сумму не только для массивов с байтовыми компонентами (ARRAY OF BYTE), но также и для переменных типа структура (STRUCT). Если Вы хотите использовать программу с переменными другого типа, то Вы должны изменить соответствующую проверку (идентификатор ID типа данных в ANY-указателе).

Функция генерации контрольной суммы использует прямой доступ для получения абсолютного адреса параметра блока (или, более точно - адреса, по которому редактор сохранил ANY-указатель). Сначала выполняется проверка идентификатора ID типа данных (на соответствие типу BYTE) и вводится множитель повторения (>1). В случае ошибки двоичный результат устанавливается в "0" и выполнение функции прерывается с возвращаемым значением функции, равным 0.

Начальный адрес фактического параметра (в режиме выполнения программы) находится в ANY-указателе. Он загружается в адресный

регистр AR1. Если переменная размещена в блоке данных, то этот блок также открывается.

Следующий сегмент добавляет значения всех байтов, составляющих фактический параметр. Циклическое выполнение программы будет продолжаться до тех пор, пока переменная *Quantity* не будет иметь нулевое значение (при каждом проходе цикла значение переменной декрементируется). После этого сумма передается в возвращаемое значение функции.

Пример "Generate frame"
("Создание фрейма")

Пример "Generate frame" ("Создание фрейма") показывает использование системной функции SFC 20 BLKMOV для копирования сложных переменных.

Блок данных "Send_mailb" должен быть заполнен данными фрейма сообщения. Здесь используется функциональный блок с идентификатором ID и порядковым номером в его экземплярном блоке. Собственно данные расположены в глобальном блоке данных; они копируются в блок "почтовый ящик для исходящих сообщений" ("send mailbox") с помощью системной функции BLKMOV.

Значение времени суток поступает от часов реального времени, встроенных в CPU, с помощью функции "Clock_check" (см. выше), затем генерируется контрольная сумма методом простого сложения всех байтов заголовка фрейма сообщения и данных (см. пример "Checksum" ["Контрольная сумма"]). На рис. 26.9 показаны структура программы и структура данных.

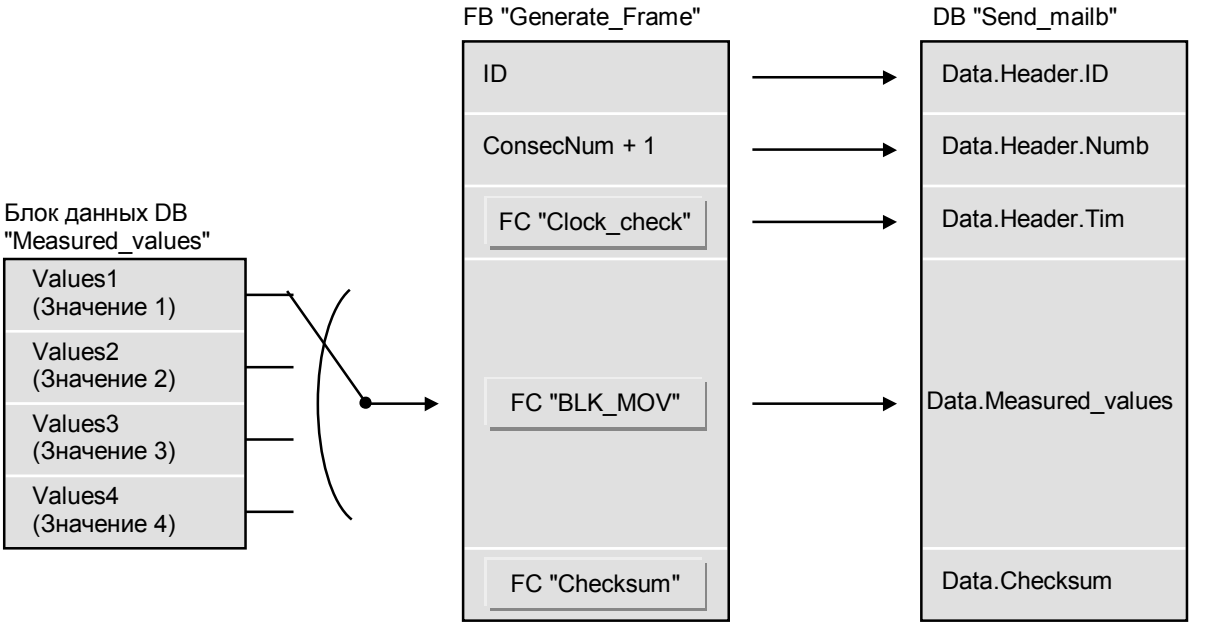


Рис. 26.9 Пример "Generate frame" ("Создание фрейма")

Первый сегмент в функциональном блоке FB "Generate_frame" передает идентификатор ID, сохраненный в экземплярном блоке данных в заголовок фрейма. Порядковый номер ConsecNum инкрементируется (+1) и также поступает в заголовок фрейма.

Второй сегмент содержит вызов функции "Clock_check", которая считывает суточное время из часов реального времени и в формате TIME_OF_DAY передает эту информацию в заголовок фрейма.

В третьем сегменте Вы можете видеть принципы использования системной функции SFC 20 BLKMOV для копирования переменных, выбранных в процессе работы программы, без использования косвенной адресации. Следовательно, нет необходимости знать абсолютный адрес и структуру переменной.

Принцип предельно прост: требуемая функция копирования выбирается с помощью распределителя переходов. Здесь при выборе перехода допускаются номера от 1 до 4. Пример "Buffer entry" ("Входной буфер") демонстрирует такую же функциональность, но на этот раз, с набором "переменных назначения" и с вычисляемым указателем в процессе работы программы (см. 26.10).

В следующем сегменте программы генерируется контрольная сумма с учетом заголовка фрейма и данных фрейма. Так как функция "Checksum" генерирует контрольную сумму по единой области данных, то сначала заголовок фрейма и данные фрейма объединяются во временную переменную *Block*. После этого содержимое переменной *Block* побайтно суммируется, и результат сохраняется как контрольная сумма в исходящем фрейме.

Функциональный блок FB "Generate_Frame" запрограммирован таким образом, что он может вызываться для генерации фрейма по фронту сигнала.

Пример "Store frame"
("Хранение фрейма сообщения")

В примере "Store frame" ("Хранение фрейма сообщения") показано, как используется "переменная" ANY-указатель).

Фрейм в блоке данных "Rec_mailb" должен быть внесен в следующую позицию в блоке данных "Buffer". Локальная переменная блока Entry определяет позицию в кольцевом буфере; адрес кольцевого буфера высчитывается, исходя из значения в данной позиции (см. рис. 26.10).

Если номер фрейма блоке "почтовый ящик для входящих сообщений" ("receive mailbox") изменился, то входящий фрейм должен быть записан в буфере в следующей позиции. Буфер должен представлять собой блок данных, который может накапливать до 8 фреймов. После прихода восьмого фрейма следующий, девятый фрейм, должен быть внесен вновь в первую позицию.

Функциональный блок "Store_Frame" сравнивает номер пришедшего фрейма с сохраненным номером в блоке данных "Rec_mailb". Если номера фреймов различаются, то номер, который сохранен, корректируется, и фрейм, находящийся в блоке "почтовый ящик для входящих сообщений" ("receive mailbox"), копируется в блок данных "Buffer" в следующую позицию. Системная функция SFC 20 BLKMOV управляет копированием.

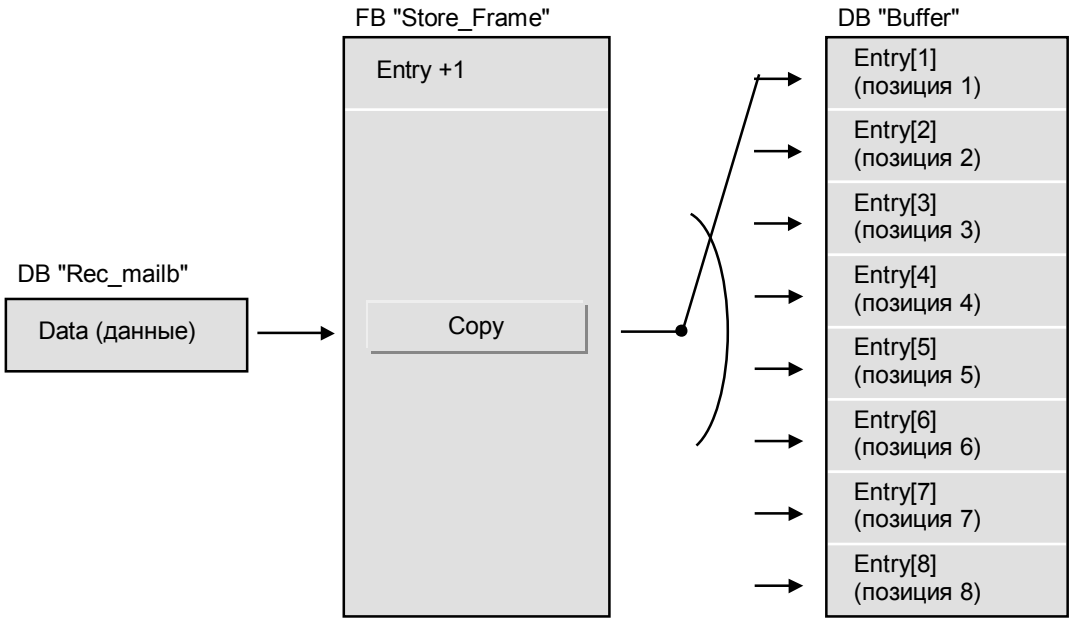


Рис. 26.10 Пример "Store frame" ("Хранение фрейма сообщения")

Область назначения может быть различной, в зависимости от состояния переменной *Entry*. Производится расчет абсолютного адреса области назначения, создается ANY-указатель в переменной *ANY_pointer* и передается в SFC в параметр DSTBLK.

Необходимо отметить, что для косвенной адресации временных локальных переменных Вы можете использовать только внутризонную адресацию.

Структура данных *Frame* имеет размер, равный 20 байтов (заголовок: 8 байтов, собственно данные: 8 байтов и контрольная сумма: 4 байта). Переменная *Receive* в блоке данных "Rec_mailb", следовательно, также имеет длину 20 байтов, точно также как и каждый компонент массива *Entry* в блоке данных "Buffer" имеет длину 20 байтов. Следовательно, отдельные элементы *Entry[n]* начинаются с байтового адреса $n \times 20$, где n соответствует номеру элемента переменной *Entry*.

Пример "Date conversion"
("Преобразование даты")

Пример "Date conversion" ("Преобразование даты") иллюстрирует вопросы обработки переменных сложных типов данных с использованием прямого доступа к переменным и косвенной адресации посредством обоих адресных регистров.

Глобальный блок данных "Data66" содержит переменные *CPU_Tim* (тип данных *DATE_AND_TIME*) и *Dat* (тип данных *STRING*). Дата должна считываться из переменной *CPU_Tim* и сохраняться в виде строки символов в формате "*YYMMTT"* в переменной *Dat*.

Следующая программа в функции "DT_Conv" использует адресный регистр A1 и DB-регистр для указателя на входной параметр Tim.

Адресный регистр A2 и DI-регистр в программе используются для указателя на значение функции (в соответствии с переменной *Dat,* относящейся к типу STRING, в блоке данных "Data66"). Данная программа размещена в функции, так что и DB-регистр, и DI-регистр, а также оба адресных регистра предоставляют пользователю неограниченный доступ.

В первом сегменте программы рассчитывается адрес фактического параметра для параметра блока *Tim,* остающийся действующим ("валидным") во время выполнения программы, после чего этот адрес сохраняется в DB-регистре в AR1. Фактический параметр сложного типа может быть размещен только в блоке данных (глобальных данных или экземплярных данных) или во временных локальных данных вызывающего блока (в V-области). Если фактический параметр находится в блоке данных, то номер этого блока данных будет загружен в DB-регистр, и указатель на область (area pointer) в адресном регистре AR1 будет содержать адресную область блока данных DB. Если фактический параметр находится в V-области, то в DB-регистр будет загружен нуль, и указатель на область (area pointer) в адресном регистре AR1 будет содержать адресную область V.

Во втором сегменте программы содержится "эквивалентная" программа для значения функции, адрес которого размещается в адресном регистре AR2 и в DI-регистре. Чтобы можно было использовать косвенную адресацию также посредством DI-регистра, адресная область DI должна быть введена в адресный регистр AR2. Тем не менее, в зависимости от занимаемой фактическим параметром области памяти здесь определяется или DB для блока данных, или V - для V-области. Установкой 24-го бита в адресном регистре AR2 в состояние "1" мы можем изменить адресную область с DB на DI, но мы никак не сможем изменить адресную область V.

Подготовленное таким путем значение максимальной длины, зафиксированное для фактического параметра в значении функции, может быть использовано в следующем сегменте. Эта длина должна состоять по крайней мере из 6 символов. Если длина короче 6 символов, то в двоичный результат BR записывается значение "0" (в противоположном случае BR = "1"), после чего завершается обработка блока. Таким образом, Вы можете контролировать наличие ошибок обработки с помощью проверки двоичного результата после вызова функции "DT_Conv".

В следующем сегменте программы считываются год и месяц из переменной *Tim* (в формате двоично-десятичного [BCD] числа), после чего эти значения преобразуются в символы ASCII (с предшествующим символом "3") и записываются в качестве возвращаемого значения функции. Так же выполняется обработка данных, которые определяют день.

Обработка программы заканчивается корректировкой длины в значении функции.