МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНБАСЬКА ДЕРЖАВНА МАШИНОБУДІВНА АКАДЕМІЯ

Проектування систем автоматизації

Розділ 3

Розробка програмного забезпечення систем автоматизації

Конспект лекцій

для студентів спеціальності
«Автоматизація та комп‘ютерно-інтегровані технології»

Краматорськ 2018

УДК 621.3

Проектування систем автоматизації. Розділ 3: Розробка програмного забезпечення систем автоматизації. Конспект лекцій для студентів спеціальності «Автоматизація та комп˜ютерно-інтегровані технології») / Укл. О. О. Сердюк. - Краматорськ: ДДМА, 2018. - 112 с.

Викладені особливості структурної організації програм у системах автоматизації SIMATIC. Наведені методики створення користувацької програми на мовах програмування STL, SCL, LAD, FBD, S7-Graph.

Укладач СЕРДЮК Олександр Олександрович, доц.
Відп. за випуск КЛИМЕНКО Галина Петрівна, проф.

ЗМІСТ

1 ПРОЕКТУВАННЯ СТРУКТУРИ ПРОГРАМИ 5
1.1 Структурна організація програмного забезпечення в CPU 5
1.2 Особливості використання блоків 9
1.3 Методика створення логічних блоків 13
1.4 Адресація змінних у блоці.....15
1.5 Призначення типів даних 19
Контрольні питання.....23
2 ПРОГРАМУВАННЯ ПРИСТРОЇВ ЛОГІЧНОГО КЕРУВАННЯ.....24
2.1 Програмування двійкових логічних операцій.....24
2.2 Програмування операції з пам'яттю й передачі даних.....32
2.3 Програмування таймерів 40
2.4 Програмування лічильників 44
2.5 Використання функцій порівняння.....47
2.6 Програмування арифметичних і математичних функцій 50
2.7 Застосування функцій перетворення типів даних.....56
2.8 Програмування функцій зрушення 60
2.9 Контроль стану операції й програмування переходу у програмі.....63
2.10 Застосування інструкцій для виклику й завершення блоків 68
2.11 Методика створення програми на мовах LAD, STL, FBD.....70
Контрольні питання.....73
3 ПРОГРАМУВАННЯ МОВОЮ SCL.....75
3.1 Призначення адрес і типів даних у мові SCL.....75
3.2 Правила використання виражень і операторів.....79
3.3 Особливості застосування операторів керування програмою.....81
3.4 Особливості програмування SCL-блоків.....88
3.5 Особливості програмування SCL-функцій.....94
Контрольні питання..... .98
4 ПРОГРАМУВАННЯ МОВОЮ S7-GRAPH..... .99
4.1 Особливості мови S7-GRAPH..... .99
4.2 Програмування дій і умов 103
4.3 Установка параметрів.....107
4.4 Створення структури й установка режимів системи керування 109
Контрольні питання.....111
ЛІТЕРАТУРА.....112

1 ПРОЕКТУВАННЯ СТРУКТУРИ ПРОГРАМИ

1.1 Структурна організація програмного забезпечення в CPU

В CPU завжди виконуються дві програми: *операційна система* й *програма користувача*.

Операційна система

Кожний CPU містить операційну систему, яка організує функції й послідовності в CPU, не пов'язані з конкретним завданням керування.

Операційна система забезпечує наступні функції:

- виклик програми користувача;
- обробку "теплого" і "гарячого" перезапуску;
- відновлення таблиці образа процесу для входів і вивід таблиці образа процесу для виходів;
- виявлення переривань і виклик організаційних блоків переривань;
- виявлення й обробка помилок;
- керування областями пам'яті;
- обмін інформацією із пристроями програмування й іншими комунікаційними партнерами.

Параметри операційної системи встановлюються за замовчуванням і міняти їх не можна.

Програма користувача

Програма користувача містить ті функції, які необхідні для реалізації конкретного завдання автоматизації.

Завдання програми користувача полягають у наступному:

- обробка даних процесу (зчитування й аналіз вхідних сигналів, а також обчислення функцій переходу й значень вихідних сигналів);
- реакція на переривання;
- обробка порушень у нормальному виконанні програми.

Програмне забезпечення STEP 7 дозволяє структурувати *користувацьку* програму, інакше кажучи, розбивати програму на окремі автономні програмні секції, які простіше модифікувати й налагодити. Важливо також і те, що такі програмні секції можна використовувати неодноразово для повторюваних технологічних функцій процесу. При структурному програмуванні програмні секції представляються організаційними блоками (OB), функціональними блоками (FB), функціями (FC) і блоками даних (DB).

Функціонування структурованої програми користувача полягає у викликах функціональних блоків (рис. 1.1).

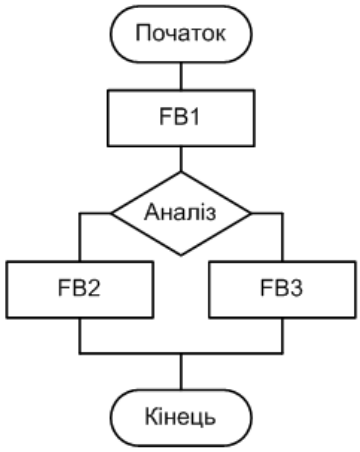


Рисунок 1.1 – Сутність функціонування структурованої програми

Кількість блоків може бути любою. Для організації циклічного виконання програми обов'язкове застосування організаційного блоку OB1.

У принципі можна записати всю користувацьку програму в одному блоці OB1. Таке програмування називається лінійним. Лінійне програмування доцільне застосовувати тільки при створенні простих програм.

Фази циклічної обробки програми

Користувацька програма, записана в OB1, складається з функціональних блоків, які працюють із поточними значеннями входів і формують поточні значення виходів. Для того, щоб під час циклічної обробки програми мати несуперечливий образ сигналів процесу, CPU звертається не безпосередньо до периферійних входів (PI) і виходів (PQ) на модулях уведення/виводу, а до області внутрішньої пам'яті CPU, яка містить образ входів (I) і виходів (Q).

Циклічна обробка програми містить у собі наступні фази:

1. Операційна система запускає час контролю циклу.
2. CPU переписує значення виходів із таблиці *образа виходів* процесу у модулі виводу.
3. CPU зчитує стани входів (у модулях уведення) і записує їх у таблицю *образа входів* процесу.
4. CPU виконує програму користувача.
5. Операційна система виконує завдання, які чекають своєї черги (фоновий режим).
6. CPU перезапускає час контролю циклу й починає новий цикл.

Циклічна обробка програми може бути перервана в результаті подачі команди STOP, виходу з ладу живлення, а також виникнення інших несправностей або помилок в програмі.

Час виконання циклу

Час виконання циклу – це час, необхідний операційній системі для виконання циклічної програми й системних операцій, наприклад, переривань.

Час виконання циклу ТС не однаковий в кожному циклі. На рисунку 1.2 показаний випадок збільшення часу виконання циклу CPU. Тут час виконання циклу ТС_1 більше часу циклу ТС_2 через переривання за часом дня, який виконується організаційним блоком ОВ10.

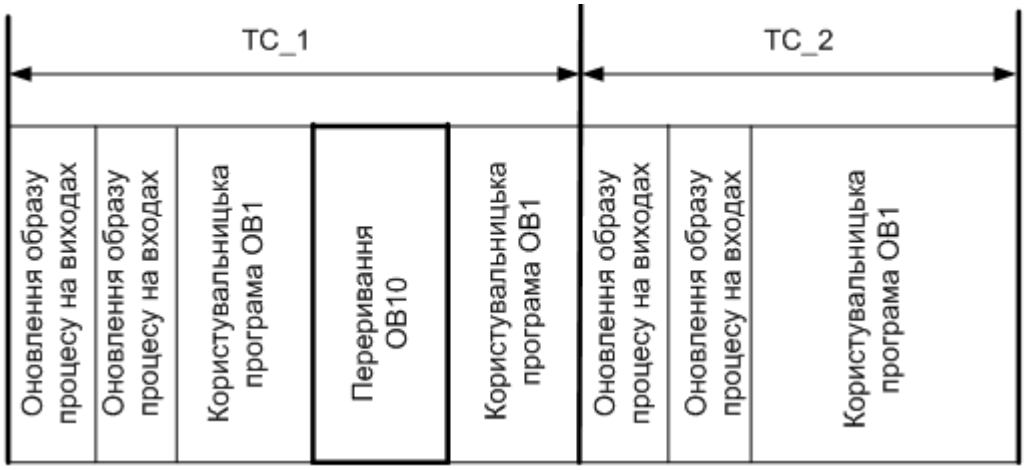


Рисунок 1.2 – Виникнення різного часу виконання циклів CPU

За допомогою STEP 7 можна задавати максимальний і мінімальний час циклу. Якщо пройде максимальний час циклу, а програма користувача не закінчиться, то в CPU можливі дві процедури: або він переходить у режим STOP, або буде викликаний блок ОВ80, щоб визначити, як CPU повинен реагувати на таку помилку.

Якщо час робочого циклу повинен бути однаковим, наприклад, у завданнях регулювання, то слід установити мінімальну тривалість циклу. Відмінність між максимальним і мінімальним часом буде являти собою резерв для обслуговування переривань і фонових завдань.

Ієрархія створення й викликів блоків у програмі користувача

Функціонування структурованої програми користувача полягає у викликах функціональних блоків. Для цього використовуються спеціальні команди, які можуть бути запущені тільки в логічних блоках.

Порядок викликів блоків називається *ієрархією викликів*. Кількість блоків, які можуть бути вкладені друг у друга (глибина вкладення) залежить від конкретного CPU.

Порядок викликів блоків диктує, відповідно, порядок їх створення.

Якщо побудувати ієрархію викликів в організаційному блоці ОВ1, як

показано, наприклад, на рисунку 1.3, то сформулювати порядок створення блоків можна в такий спосіб:

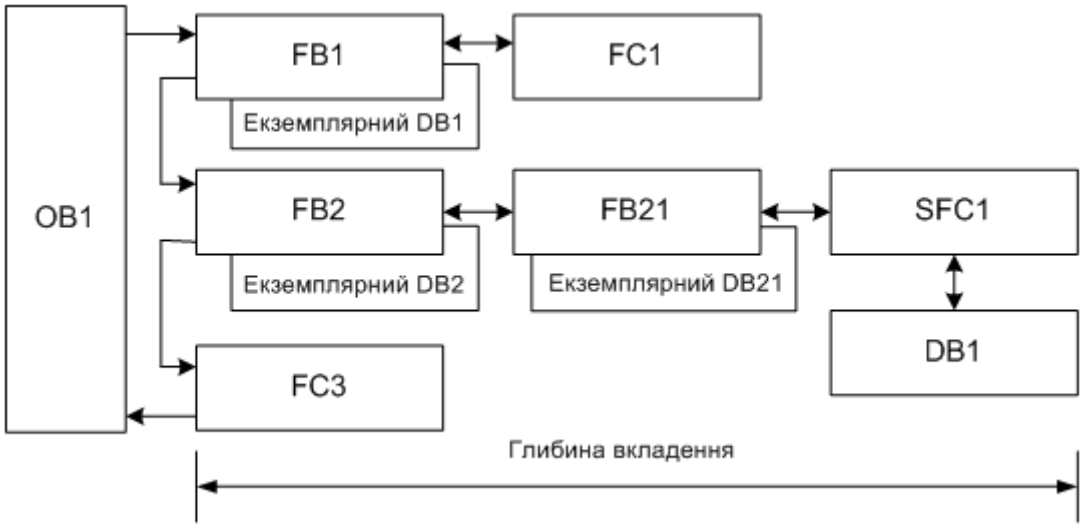


Рисунок 1.3 - Порядок викликів блоків усередині циклу програми

• Блоки створюються *зверху вниз*, тобто починати потрібно з верхнього ряду блоків.

• Оскільки кожний викликуваний блок уже повинен існувати, то усередині ряду блоки повинні створюватися *справа наліво*.

Враховуючи цей порядок, для наведеного на рисунку 1.3 прикладу визначимо таку послідовність створення блоків:

1. Для верхнього ряду спочатку повинен бути створений блок з функцією FC1, потім функціональний блок FB1 з його екземплярним блоком даних DB1.

2. Для наступного ряду спочатку створюється блок системних даних DB1 для системної функції SFC1, після цього складається програма функціонального блоку FB21, для якого потім створюється блок даних DB21. Далі розробляється функціональний блок FB2 з його екземплярним блоком даних DB2.

3. Для останнього ряду створюється блок з функцією FC3.

4. На закінчення створюється блок OB1.

У програмі STEP 7 використовуються різні операнди – сигнали входів/виходів, меркери, лічильники, таймери, блоки даних і функціональні блоки. Звертання до цих операндів здійснюється через їхні абсолютні адреси. Однак програма буде значно легше читатися, якщо замість цих адрес застосувати символьні імена. STEP 7 може автоматично перетворювати символьні імена в необхідні абсолютні адреси. Однак для цього символьні

імена вже повинні бути призначені абсолютним адресам. Так, наприклад, якщо призначити символьне ім'я Motor_On адресі Q 4.0, то потім можна використовувати Motor_On, як адресу в операторові програми.

Слід урахувати, що символьні адреси полегшують розуміння відповідності елементів програми й апаратних засобів керування процесом.

Розрізняють локальні й глобальні символи. Їхні відмінності зводяться до наступного:

1. Глобальні символи діють у всій програмі користувача, а локальні – у межах блоку, для якого вони визначені.

2. Глобальні символи повинні бути записані в таблиці символів, а локальні символи призначаються в таблицях опису змінних блоків даних.

3. Глобальні символи можуть визначати всі змінні, а локальні – тільки параметри блоку (вхідні, вихідні, статичні й тимчасові).

У розділі кодів програми глобальні символьні імена з таблиці символів відображаються в лапках "...", а локальним символьним іменам з таблиці опису змінних блоку передує символ "#".

Уводити лапки або символ "#" немає необхідності. При введенні програми редактор додає ці символи автоматично.

По всій таблиці символів необхідно використовувати тільки мнемонічні позначення, пропоновані редактором.

1.2 Особливості використання блоків

Організаційні блоки OB (Organization blocks)

Організаційні блоки служать своєрідним *інтерфейсом* між операційною системою й користувацькою програмою.

Операційна система CPU викликає організаційні блоки при виникненні особливої події, наприклад, при запуску програми користувача, або при апаратному перериванні.

Головна програма перебуває в організаційному блоці OB1. Інші організаційні блоки також мають постійні призначені номери, прив'язані до певних подій.

Функціональні блоки FB (Function blocks) і блоки даних DB (Data blocks)

Функціональні блоки є частинами програми й створюються для розв'язку певних завдань. Вони мають область пам'яті для змінних, яка розташована в блоці даних DB. Кожному функціональному блоку або,

точніше, *виклику* функціонального блоку буде призначений свій блок DB. Таким чином, якщо один функціональний блок викликається, наприклад, п'ять разів, то буде створено п'ять екземплярів DB.

Постійно призначений блок даних називається *екземплярним блоком даних, або локальним екземпляром*. Виклик функціонального блоку й екземплярного блоку даних називається *екземпляром виклику* або, для стислості, *екземпляром*. Функціональні блоки можуть зберігати свої змінні не тільки у своєму екземплярному блоці даних, але й у екземплярному блоці даних того блоку, який зробив виклик.

Крім екземплярних блоків даних існують блоки глобальних даних (global data blocks). Блоки глобальних даних у користувацькій програмі кодовому блоку не призначаються.

При створенні блоку даних необхідно визначити, у якій формі будуть зберігатися дані, тобто вказати типи даних. При цьому операційна система процесора відводить у пам'яті відповідний ресурс для зберігання даних у потрібному форматі. Слід урахувати, що перед створенням екземплярного блоку даних відповідний FB уже *повинен існувати*, інакше неможливо буде вказати *номер FB*, для якого створюється екземплярний блок даних.

Функції FC (Functions)

Функції використовуються для програмування часто повторюваних або специфічних завдань автоматики. Функція може мати призначені їй параметри, значення яких вона може повертати в блок, який викликав функцію. Функції не зберігають інформацію й не мають призначених блоків даних.

Оскільки FC створюється для формального параметра (деякого образа), а обробляє фактичне значення, то формальні параметри потрібно зіставити фактичним значенням.

Якщо, наприклад, формальному параметру "Start" відповідає вхід "E 3.6" (фактичний параметр), то при виклику функції вона замінить "Start" значенням, установленим на вході E 3.6. Інакше кажучи, формальні вхідні й вихідні параметри, використовувані FC, зберігаються як покажчики на фактичні параметри логічного блоку, який викликав FC.

Системні й стандартні блоки

Системні й стандартні блоки є компонентами *операційної системи*. Системні блоки (функції SFC і функціональні блоки SFB) можуть містити дані в системних блоках даних (SDB). Вони забезпечують важливі системні функції, доступні користувачеві, наприклад, функції керування внутрішнім годинником CPU або комунікаційні функції. При цьому системні й стандартні

блоки не займають місця в користувацькій пам'яті – вони розташовуються в операційній системі. Однак для системних блоків потрібно створювати екземплярні блоки даних і завантажувати їх в CPU як частину програми користувача.

Використання декількох екземплярних DB для одного FB

Схема з декількома екземплярами DB дозволяє за допомогою одного FB управляти декількома однотипними пристроями. Так, наприклад, FB, створений для деякого класу двигунів, може управляти різними двигунами, використовуючи для кожного з них певний набір даних. При використанні цього методу для декількох двигунів потрібний тільки один функціональний блок (рис. 1.4).

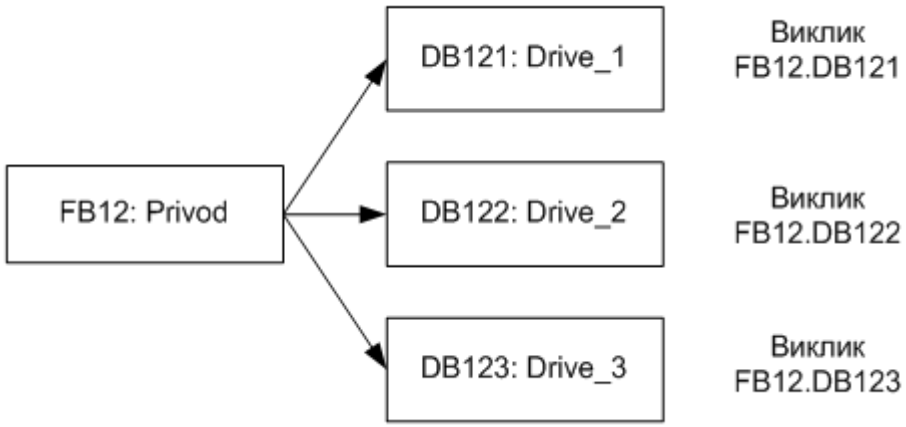


Рисунок 1.4 – Декілька екземплярних блоків даних для одного FB

Використання одного екземплярного DB для декількох екземплярів FB

Екземпляри даних для декількох двигунів можна одночасно передавати в один екземплярний DB. Для цього потрібно запрограмувати виклики в одному FB і описати (типом FB) статичні змінні для кожного екземпляра в розділі описів.

На рисунку 1.5 виклик здійснює блок FB12 "Privod", а змінні мають тип даних FB13, тобто блоку, який викликається. Екземпляр виклику визначається за допомогою призначень Drive_1, Drive_2 і Drive_3.

У цьому прикладі FB13 не має потреби у власному екземплярному блоці даних, тому що дані його екземплярів зберігаються в блоці даних DB20, який належить FB12 – блоку, який його викликав.

При використанні одного екземплярного DB для декількох екземплярів FB заощаджується пам'ять і оптимізується використання блоків даних.

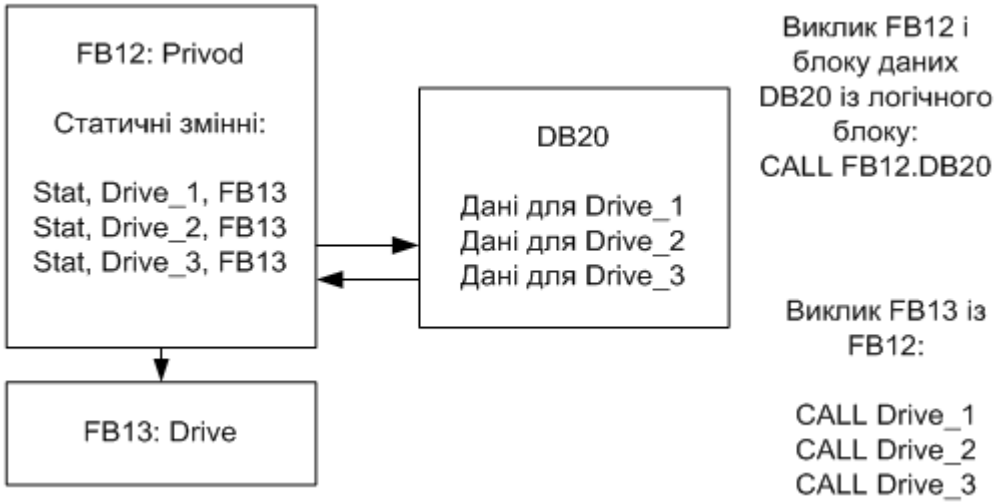


Рисунок 4.5 - Використання одного екземплярного DB для декількох екземплярів FB

Використання глобальних блоків даних

Глобальні блоки даних застосовуються для зберігання користувацьких даних, до яких можуть звернутися всі блоки.

Кожний FB, FC або OB може читати дані із глобального DB або записувати дані в цей DB. Ці дані зберігаються в DB після виходу з нього. Глобальний і екземплярний DB можуть бути відкриті одночасно.

На рисунку 1.6 показані методи доступу до блоків даних.

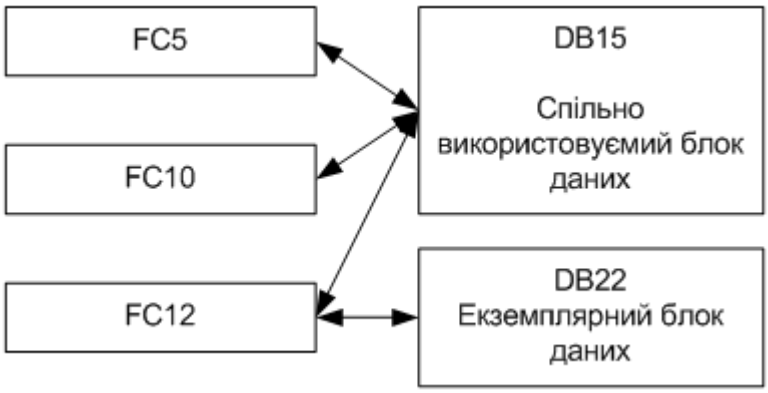


Рисунок 4.6- Варіанти доступу до блоків даних

Потрібно враховувати, що коли викликається логічний блок FC, FB або організаційний блок OB, то він повинен на час виклику зайняти місце в області локальних даних, тобто в L-стеці. При цьому логічний блок відкриває область пам'яті в DB. На відміну від даних, які перебувають в L-стеці, дані в DB не видаляються після завершення роботи логічного блоку.

1.3 Методика створення логічних блоків

Логічні блоки OB, FB і FC містять у собі три розділи: розділ опису змінних, розділ кодів, а також розділ властивостей. У розділі (таблиці) опису змінних визначаються параметри, їх системні атрибути й локальні змінні блоку. У розділі кодів створюється користувацька програма. Властивості блоку містять мітки часу або шлях, який уводиться системою. Крім того, у властивостях блоку можна ввести системні атрибути для блоків, а також його власні деталі, які ставляться до імені, сімейства, версії й автору.

У принципі не має значення, у якому порядку редагуються ці частини логічного блоку.

Результатом опису змінних є наступне:

- При описі *тимчасових* змінних блоку пам'ять для неї резервується в L-стеці, а при описі *статичних* змінних пам'ять виділяється в екземплярному DB, який буде приєднаний при виклику.
- При призначенні вхідних і вихідних параметрів визначається інтерфейс виклику блоку в програмі.
- При описі змінних у функціональному блоці формується також структура даних для кожного екземплярного блоку даних DB, пов'язаного із цим функціональним блоком.

Установка системних атрибутів передбачає призначення спеціальних властивостей параметрам передачі повідомлень і конфігурації з'єднань, а також визначення функцій взаємодії з оператором.

Таблиця оголошення змінних і розділ кодів логічних блоків тісно зв'язані один з одним, тому що імена з таблиці опису змінних використовуються в розділі кодів. Для контролю зроблених оголошень використовується інтерфейс блоку.

Вікно інтерфейсу блоку відображає в лівій частині (браузері) список дозволених для декларування типів (IN, OUT, IN_OUT, STAT, TEMP), а в правій частині представляє детальний огляд змінних (рис. 1.7).

Програмування послідовності операцій для логічного блоку здійснюється в *розділі кодів* шляхом уведення відповідних команд. Після введення команди редактор негайно виконує перевірку синтаксису й відображає помилку червоним курсивом.

Розділ кодів логічного блоку містить у собі ряд сегментів (ланцюгів). Окремі частини розділу кодів можна редагувати в будь-якому порядку. При цьому можна вказати ім'я сегмента, а також увести коментарі до сегментів або окремих команд.

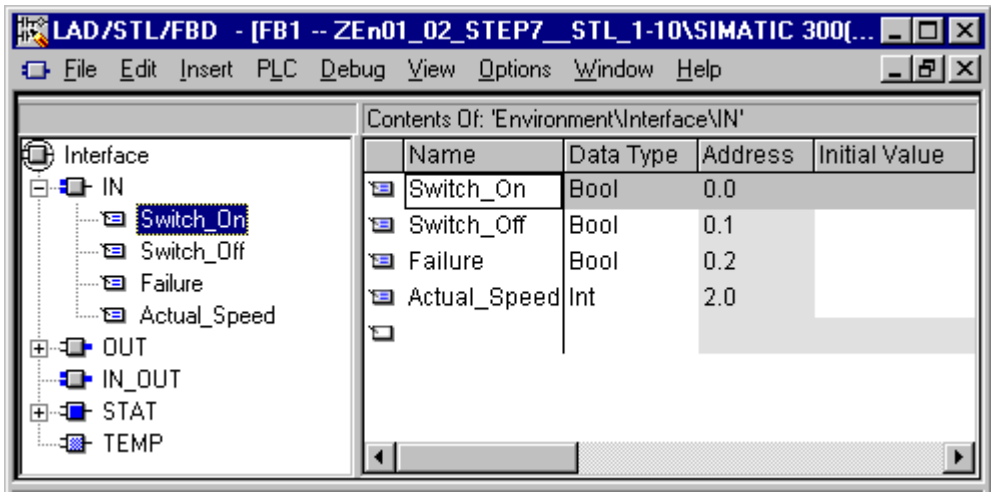


Рисунок 1.7 - Приклад таблиці опису змінних

Завдяки коментарям програма легше читається, що підвищує ефективність пошуку помилок. Коментарі є важливою частиною програмної документації й повинні використовуватися скрізь.

Інтерфейс блоків (Block Interface)

Таблиця оголошення змінних містить інтерфейс блоку, який складається із вхідних і вихідних параметрів блоку, а також статичних локальних даних. Тимчасові локальні дані не належать інтерфейсу блоку. Змінні, які входять в інтерфейс блоку, необхідно ініціалізувати (задати початкові значення) при виклику блоку.

Редактор програм перевіряє відповідність параметрів, заданих при ініціалізації блоку, його інтерфейсу. Для цього редактор використовує *мітки часу*. Мітки часу необхідні для того, щоб інтерфейс викликуваного блоку створювався раніше, чим інтерфейс блоку, який зробив виклик. Іншими словами – останні зміни інтерфейсу повинні бути виконані до його об'єднання із блоком. Редактор програм обновляє мітку часу інтерфейсу при зміні числа параметрів, типу даних, а також значень параметрів.

Конфлікт тимчасових міток (Time stamp conflict)

Якщо інтерфейс викликуваного блоку має більш пізню тимчасову мітку щодо блоку, який його викликає, то виникає "конфлікт тимчасових міток" – Time stamp conflict. Це може трапитися, наприклад, при спробі відкрити для редагування вже скомпільований блок. У цьому випадку редактор виділить некоректний виклик блоку червоним кольором.

Конфлікт тимчасових міток виникає в наступних випадках:

- Інтерфейс викликуваного блоку має більш пізню тимчасову мітку (younger), чим код викликуваного блоку. Конфлікту не буде, якщо спочатку

змінні будуть оголошені, а після цього буде написаний програмний код, у якому ці змінні використовуються.

- Інтерфейс ініціалізації не погоджений з інтерфейсом блоку.
- Функціональний блок має більш пізню тимчасову мітку, чим його екземплярний блок даних.
- Інтерфейс локального екземпляра має більш пізню тимчасову мітку, чим екземпляр, який його викликав (стосується функціональних блоків).
- Користувацький тип даних UDT має більш пізню тимчасову мітку, чим блок, у якому цей тип оголошений.

Перевірка блоку на консистентність (Check Block Consistency)

Для повної перевірки програми потрібно використовувати функцію перевірки консистентності блоку – Check Block Consistency. Ця функція знімає більшість конфліктів інтерфейсу й указує на місця в програмі, які вимагають редагування.

1.4 Адресація змінних у блоці

При адресації змінних використовуються два основні варіанти:

- абсолютна адресація (absolute addressing);
- символьна адресація (symbol addressing).

Абсолютна адресація змінних

При абсолютній адресації використовуються чисельні адреси, починаючи з адреси "0". Такий принцип застосовується для кожної адресної області.

При символьній адресації використовуються символьні імена, які задаються для глобальних адрес у таблиці символів (Symbol Table), а для локальних – у розділі оголошення змінних усередині блоків.

Для розширення можливостей абсолютної адресації використовується непряма адресація (indirect addressing), при якій адреса (місце розташування) у пам'яті обчислюється під час виконання програми.

Абсолютні адреси входів і виходів розраховуються, виходячи з початкової адреси модуля, яка зазначена в таблиці конфігурації, і типу сигналу, який підводить до модуля.

Дискретний сигнал містить один біт інформації. Прикладами дискретних сигналів є *вхідні* сигнали від кінцевих вимикачів, кнопок і т.п., які надходять на дискретні вхідні модулі, а також *вихідні* сигнали, які управляють лампами, контакторами й т.п. і надходять на дискретні вихідні модулі.

Аналоговий сигнал містить 16 біт інформації, що відповідає одному каналу. Він займає в контролері машинне слово (word), тобто 2 байта. Динамічний діапазон сигналу відповідає динамічному діапазону змінної, яка обробляється й зберігається. Динамічний діапазон зміни сигналу й інтерпретація цього сигналу, наприклад, відносне положення, узяті разом, визначають тип даних змінної.

Дискретним сигналам відповідають змінні типу BOOL (булева змінна), аналоговим сигналам – типу INT (цілочисельна змінна).

Визначальним фактором для адресації змінної є її тип, від якого залежить необхідна величина області пам'яті для розміщення змінної.

У системі STEP 7 існують 4 типу даних для абсолютної адресації:

1 біт тип даних BOOL;

8 біт тип даних BYTE або інший 8-бітовий тип даних;

16 біт тип даних WORD або інший 16-бітовий тип даних;

32 біта тип даних DWORD або інший 32-бітовий тип даних.

На рисунку 1.8 показана схема адресації для вихідних даних, які представляються бітом, байтом, словом і подвійним словом.

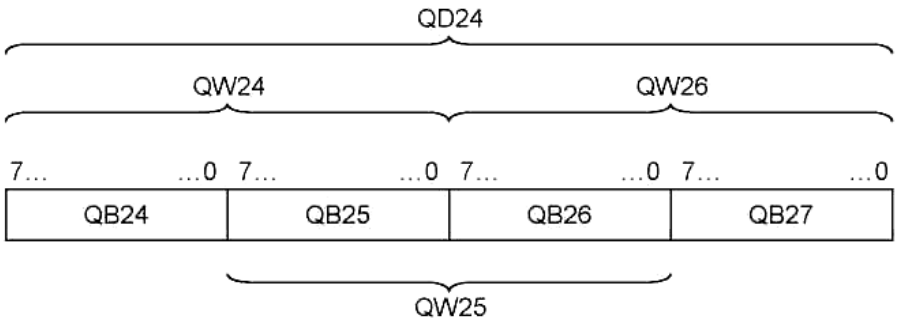


Рисунок 1.8 - Схема адресації даних, представлених бітом, байтом, словом і подвійним словом

Посилання на змінні типу BOOL здійснюються за допомогою ідентифікатора адреси, номера байта й відділеного десятковою крапкою номера біта. Нумерація байтів починається з нуля в кожній адресній області. Біти усередині байтів нумеруються від 0 до 7.

Приклади:

I 1.0 вхідний біт з номером 0 у байті номер 1.

Q 16.4 вихідний біт з номером 4 у байті номер 16.

Для змінних типу BYTE як абсолютна адреса використовується ідентифікатор адреси й номер байта, у якому перебуває значення змінної. Ідентифікатор адреси доповнюється символом B.

Приклади:

IB 2 вхідний байт номер 2.

QB 18 вихідний байт номер 18.

Змінні типу WORD складаються із двох байтів (слово). Як абсолютна адреса використовується ідентифікатор адреси й номер молодшого байта машинного слова, у якому втримується значення змінної. Ідентифікатор адреси такий змінної доповнюється символом W.

Приклади:

IW 4 вхідне слово номер 4, містить байти 4 і 5.

QW 20 вихідне слово номер 20, містить байти 20 і 21.

Змінні типу DWORD складаються із чотирьох байтів (подвійне слово). Як абсолютна адреса використовується ідентифікатор адреси й номер молодшого байта подвійного слова, у якому втримується значення змінної. Ідентифікатор адреси доповнюється символом D.

Приклади:

ID 8 вхідне подвійне слово містить байти 8, 9, 10 і 11.

QD 24 вихідне подвійне слово містить байти 24, 25, 26 і 27.

При адресації даних в DB вказується номер цього блоку даних.

Приклади:

DB 10.DBX 2.0 біт даних 2.0 у блоці даних DB 10.

DB 11.DBB 14 байт даних 14 у блоці даних DB 11.

DB 20.DBW 20 слово даних 20 у блоці даних DB 20.

DB 22.DBD 10 подвійне слово даних 10 у блоці даних DB 22.

Непряма (побічна) адресація

Непряма адресація (indirect addressing) дозволяє розраховувати адреси в області даних під час виконання програми. Мови програмування STL і SCL використовують різні методи для непрямої адресації. В STL розрізняють наступні види адресації:

- Непряма адресація *за допомогою пам'яті* (Memory-indirect-addressing). Так, наприклад, IW [MD 200] означає, що адреса перебуває в подвійному слові пам'яті меркерів.
- Непряма *внутрішньозонна адресація* виконується *за допомогою регістру* (Register-indirect area-internal addressing). Так, наприклад, IW [AR1, P#2,0] означає, що адреса *слова даних з області входів* перебуває в адресному регістрі AR1 і при виконанні оператора він повинен бути збільшений на величину зрушення P#2,0.

• Непряма *міжзонна адресація за допомогою регістру* AR. Так, наприклад, W [AR1, P#0,0] означає, що *область і адреса* перебувають в адресному регістрі AR1. При виконанні оператора адреса повинна бути збільшена на величину зрушення P#0,0 (у даному прикладі зрушення відсутнє).

Подвійні слова адресної області для даних (DBD і DID), меркерів (MD) і тимчасових локальних даних (LD) можуть використовуватися для зберігання адрес при непрямій адресації за допомогою пам'яті.

Непряму адресацію за допомогою регістру можна застосовувати з використанням двох адресних регістрів – AR1 і AR2.

При використанні мови програмування SCL адресні області складаються з поля, елементи якого доступні побічно й окремо.

Наприклад, MW[*index*] – це звертання до слова пам'яті, адреса якого розміщена у змінній *index*. Змінна *index* може визначатися в процесі виконання програми.

Символьна адресація змінних

Символьна адресація використовує імена (символи) замість абсолютних адрес. Розроблювач призначає ці імена самостійно. Ім'я повинне починатися з букви й може містити до 24 символів. В STL не дозволено використовувати ключові слова як імена (символи). Для того, щоб використовувати ключові слова як імена, в SCL потрібно вставити перед іменем символ ґрати "#".

При присвоєнні імен входам ураховується регістр написання символу. Для імен виходів редактор використовує регістр і нотацію (форму запису), які були застосовані при оголошенні символу.

У таблиці символів (symbol table) *глобальні* символи можна призначити наступним об'єктам:

- Блоки даних і кодові блоки.
- Входи, виходи, периферійні входи й периферійні виходи.
- Меркери, таймери й лічильники.
- Користувацькі типи даних.
- Таблиці змінних.

Глобальний символ може містити пробіли, спеціальні символи й національні символи, наприклад, умляут. Кожний такий символ повинен бути унікальним (однозначно належати одній адресі) у цій програмі.

Локальні символи діють тільки усередині блоку, у якому вони описані. Такі ж символи можуть бути застосовані в іншому блоці в іншому контексті для позначення зовсім інших об'єктів. Редактор відображає локальні символи

(імена), вставляючи зі спереду символ "#".

У випадку використання *масивів* доступ до окремих елементів масивів забезпечується використанням імені масиву з індексом. Так, наприклад, ім'я MSERIES[1] належить першому елемента масиву MSERIES. У випадку програмування на STL індекс повинен бути константою (INT). У випадку програмування на SCL індекс може бути як цілим числом (INT), так і вираженням.

У *структурах* кожний елемент імені (subname) відділяється від інших елементів десятковою крапкою, наприклад, FRAME.HEADER.CNUM.

Компоненти користувацьких типів даних адресуються точно також як і компоненти структур.

Символьна адресація даних припускає використання повної адреси, включаючи адресу блоку даних. Наприклад, якщо блок даних із символьною адресою MVALUES містить змінні MVALUE1, MVALUE2 і MTIME, то ці змінні можуть бути адресовані так:

MVALUES. MVALUE1
MVALUES. MVALUE2
MVALUES. MTIME

1.5 Призначення типів даних

Усі дані в програмі користувача повинні бути ідентифіковані типом даних. Доступні наступні типи даних:

- елементарні типи даних;
- складені типи даних (комбінуються з елементарних типів);
- параметричні типи даних.

Команди працюють із об'єктами даних певного розміру. Команди двійкової логіки працюють із бітами. Команди завантаження й передачі в STL, а також команди пересилання в LAD і FBD працюють із байтами (B), словами (W) і подвійними словами (DW).

Елементарні типи даних мають певний розмір, варіант вистави (формат) і діапазон. Елементарні типи можуть представлятися в наступних форматах:

- Тип BOOL представляється одним бітом зі значеннями TRUE і FALSE.
- Тип BYTE – це один байт, який представляється шістнадцятирічним числом без знаку, наприклад, B#16#1F.

- Тип WORD представляється двома байтами у двійковому (2#0001_0000_0000_0000), шістнадцятирічному (W#16#1000) або BCD (C#157) форматі.
- Тип DWORD представляється подвійним словом двійковим, шістнадцятирічним, а також десятковим числом без знака, наприклад, B#(1, 14, 100, 120).
- INT – це ціле десяткове число зі знаком і діапазоном від -32768 до +32767 (розмір 16 біт).
- DINT – це ціле десяткове число зі знаком і діапазоном від -2.147.483.648 до +2.147.483.647 (розмір 32 біта).
- REAL – число із плаваючою крапкою розміром 32 біта й форматом IEEE, наприклад, 1.254567e+12.

Складені типи даних

Складені типи даних визначають групи даних, які займають більше 32 біт, або групи даних, які складаються з інших типів даних.

STEP 7 допускає наступні складені типи даних:

- DATE_AND_TIME – дата й час. Цей тип даних зберігає рік, місяць, день, години, хвилини, секунди, мілісекунди й день тижня;
- STRING – символьний рядок. Цей тип визначає одномірний масив довжиною максимум 254 символу (тип даних CHAR). Символьний рядок може передаватися тільки як одне ціле;
- ARRAY – масив. Цей тип поєднує групу даних одного типу, утворюючи в такий спосіб одне ціле;
- STRUCT – структура, поєднує дані різних типів, утворюючи одне ціле;
- UDT – типи даних, обумовлені користувачем.

Дані типу DATE_AND_TIME (DT)

Дані типу DATE_AND_TIME зберігаються у двійково-десятковому форматі в 8 байтах. Можливі два варіанти вистави даних:

- DATE_AND_TIME# 12-25-8:01:1.23;
- DT# 12-25-8:01:1.23

Для роботи з типом даних DATE_AND_TIME використовуються спеціальні функції стандарту МЕК (ICE):

- Перетворення дати й часу у формат DATE_AND_TIME за допомогою функції FC3: D_TOD_DT.
- Одержання дати з формату DATE_AND_TIME за допомогою функції FC6: DT_DATE
- Одержання дня тижня з формату DATE_AND_TIME за допомогою

функції FC7: DT_DAY

• Одержання часу часи з формату DATE_AND_TIME за допомогою функції FC8: DT_TOD

Масиви

При створенні масиву необхідно зробити наступне:

• Описати масив за допомогою ключового слова ARRAY і привласнити масиву ім'я.

• Визначити розмір масиву, використовуючи індекси, тобто номера першого й останнього елемента по окремих вимірах масиву (максимум 6 вимірів). Індекс уводять у квадратних дужках, розділяючи виміру за допомогою коми, а номера першого й останнього елемента виміру двома крапками. Наприклад, тривимірний масив буде визначати індекс [1..5,-2..3,30..32].

• Указати тип даних, які повинні втримуватися в масиві.

Звертання до даних у масиві проводиться через індекс певного елемента в масиві. Індекс використовується в з'єднанні із символьним іменем.

Так, наприклад, якщо масив має ім'я Motor, а елемент масиву – символьне ім'я Heat_2x3, то обіг до другого елемента першої розмірності цього масиву запишеться таким способом:

Motor.Heat_2x3[1,2].

Структури

При визначенні структури необхідно описати дані в DB або в розділі опису змінних логічного блоку.

Таблиця 1.1 ілюструє опис структури Motcont, яка складається із двох булевых змінних On і Off, тимчасової затримки Delay типу S5TIME і цілочисельної змінної maxSpeed.

Таблиця 1.1 - Приклад опису структури

Ім'я		Тип даних	Початкове значення	Коментар
Motcont		**STRUCT**		Змінна простої структури
	On	BOOL	FALSE	Змінна *Motcont.On*
	Off	BOOL	TRUE	Змінна *Motcont.Off*
	Delay	S5TIME	S5TIME#5s	Змінна *Motcont.Delay*
	maxSpeed	INT	5000	Змінна *Motcont.maxSpeed*
		END_STRUCT		

Типи даних, обумовлені користувачем

Типи даних, обумовлені користувачем (UDT), можуть поєднувати елементарні й складені типи даних. Можна привласнювати UDT імена й використовувати їх багато раз.

Замість уведення всіх типів даних по одному або у вигляді структури досить лише визначити "UDT20" як тип даних і STEP 7 автоматично виділить відповідний простір пам'яті.

Параметричні типи

Крім елементарних і складених типів даних використовуються також параметричні типи даних.

До цього типу належать наступні дані:

• TIMER або COUNTER – визначає конкретний таймер або конкретний лічильник, який буде використовуватися під час виконання блоку. При параметризації формальний параметр типу TIMER або COUNTER забезпечується значенням, тобто після введення "T" або "C" уводиться позитивне ціле число.

• BLOCK – визначає конкретний блок, використовуваний як вхід або вихід. Опис цього параметра визначає використовуваний тип блоку (FB, FC, DB і т.д.) і значення, які задається як фактична адреса блоку, наприклад, "FC101" при використанні абсолютної адресації або "Valve" при символьній адресації.

• POINTER – указує адресу змінної замість її значення. Так, наприклад, покажчик для адресації даних, які починаються з M 50.0, буде мати такий формат: P#M50.0.

• ANY – використовується, коли тип даних фактичного параметра невідомий або коли можна використовувати будь-який тип даних.

Обмеження типів даних для блоків

В STEP 7 є обмеження на призначення елементарних, складових і параметричних типів даних.

Оскільки викликати OB не можна, то в OB не може бути вхідних, вихідних або прохідних параметрів. Таким чином, в OB можуть бути тільки *тимчасові* змінні елементарних або складових типів.

Для FB кількість обмежень на призначення типів менше. При описі вхідних параметрів FB обмежень немає, але для вихідного параметра не можна призначати параметричні типи. Тимчасові змінні можуть мати тип даних ANY. Усі інші параметричні типи заборонені.

Контрольні питання

1. Які функції забезпечує операційна система?
2. Які функції забезпечує програма користувача?
3. Що являє собою *структурування* користувацької програми?
4. Чим відрізняється структурне програмування від лінійного?
5. Які фази містить у собі циклічна обробка програми?
6. Від чого залежить час виконання циклу?
7. Для чого призначаються організаційні блоки програми?
8. Для чого створюються функціональні блоки FB?
9. Для чого використовуються функції FC?
10. Для чого створюються блоки даних DB?
11. У чому полягають особливості системних функцій і блоків SFC, SFB?
12. У чому полягає відмінність між формальними й фактичними параметрами даних?
13. Які основні об'єкти утворюють ієрархію структури проекту?
14. У якому порядку створюються блоки програми?
15. У чому полягає відмінність між глобальними й локальними змінними?
16. Які три частини необхідно редагувати при створенні блоків?
17. Що створює операційна система контролера в результаті опису змінних?
18. Яку роль відіграють мітки часу при створенні блоків?
19. При яких умовах виникає конфлікт тимчасових міток?
20. Як здійснюється адресація даних?
21. Які дані належать до елементарних типів?
22. Які дані належать до складених типів?
23. Які дані належать до параметричних типів?

2 ПРОГРАМУВАННЯ ПРИСТРОЇВ ЛОГІЧНОГО КЕРУВАННЯ

Для програмування пристроїв логічного керування зазвичай застосовуються текстова мова STL (Statement List) і дві графічні мови – LAD (Ladder Diagram) і FBD (Function Block Diagram). Розробка програми обраною мовою здійснюється в одному редакторі – «LAD, STL, FBD: Programming S7 Blocks». Редактор дозволяє переходити від одної вистави програми до іншої, наприклад, від вистави програми мовою LAD до вистави мовою FBD, або STL. Такий перехід стає можливим завдяки однаковому набору функцій у цих трьох мовах.

У мовах LAD, STL, FBD використовуються *базові функції, функції для обробки чисел і функції керування в програмі*.

Базові функції містять у собі двійкові логічні операції, операції з пам'яттю, функції передачі даних, а також функції таймерів і лічильників.

Функції для обробки чисел – це функції порівняння, математичні й арифметичні функції, а також функції перетворення й зрушення.

Функції керування в програмі містять у собі функції переходів і функції обробки блоків.

2.1 Програмування двійкових логічних операцій

Особливості бінарної логіки в STL

У мові STL застосовуються двійкові логічні операції A (AND – логічне І), O (OR – логічне АБО) і X (Exclusive OR – Виключаюче АБО). При обчисленні булевих функцій із цими операціями перевіряється результат на рівні логічної одиниці “1”. Для перевірки логічного “0” використовуються функції з модифікатором N, тобто функції AN, ON і XN, відповідно.

Потрібно взяти до уваги, що в схемах SIMATIC S7 стан сигналу має значення "1" при наявності напруги на вході модуля введення, а значення "0", якщо напруга на вході відсутнє.

Суворо кажучи, при обчисленні булевої функції CPU не зв'язує стан сигналу перевіреного біта, а скоріше формує результат перевірки. При цьому у випадках перевірки на стан "1" результат перевірки ідентичний стану сигналу, а у випадках перевірки на стан "0" результат перевірки інвертується щодо стану сигналу.

Перевірку стану біта ініціює вираження, яке містить оператор перевірки (check statement). Це вираження містить правило логічної операції, тобто

алгоритм, згідно з яким результат перевірки стану біта буде прирівнюватися збереженому в процесорі *результату логічної операції* (RLO).

Результат логічної операції CPU надалі використовується для обробки двійкових сигналів. Значення RLO формується й модифікується за допомогою операторів перевірки. Якщо RLO має значення "1", то це означає, що умова двійкової логічної операції виконана, якщо RLO має значення "0, то умова двійкової логічної операції не виконана.

У процесі виконання логічної операції беруть участь вхідний модуль, вихідний модуль і CPU. Після перевірки стану сигналу датчика, підключеного до каналу вхідного модуля, CPU зв'язує його з результатом логічної операції RLO, який був збережений після виконання попередньої логічної операції, а потім формує відповідний вихідний сигнал.

Ця схема показана на рисунку 2.1.

Рисунок 2.1 - Схема виконання перевірки двійкової логічної операції

Результат *поточної* логічної операції запам'ятовується й зберігається як *новий результат логічної операції*. Таким чином, обробка RLO здійснюється на кожному логічному кроці операції.

Кожний логічний крок складається з виражень із операторами

перевірки, які називаються операторами сканування, і виражень із умовними операторами. Перший оператор перевірки, який виконується за умовним оператором, називається *первинним опитуванням* (first check).

Умовні оператори – це такі оператори, виконання яких залежить від результату логічної операції RLO. Ці оператори включають операції призначення (assign), установки (set) і скидання (reset) двійкових розрядів, а також запуск таймерів і лічильників.

Умовні оператори (за рідкісним винятком) виконуються тільки тоді, коли результат логічної операції RLO має стан "1", і не виконуються, коли RLO має стан "0". Умовні оператори не впливають на результат логічної операції RLO, і, таким чином, RLO протягом послідовного виконання декількох умовних операторів залишається незмінним.

Приклади логічних кроків з операторами А:

= Q 4.0 умовний оператор (conditional statement), який призначає поточний RLO вихідному сигналу з адресою Q 4.0 (на схемі рис. 2.1 перехід від меркера збереження до модуля виводу);

A I 2.0 первинне опитування (first check) – це сканування входів на схемі рис. 2.1;

A I 2.1 оператор перевірки (check statement) – це двійкова логічна операція на рис. 2.1;

= Q 4.3 умовний оператор (conditional statement).

У наведеному прикладі використаний умовний оператор присвоєння "=". Цей оператор призначає результат логічної операції RLO біту, зазначеному у вираженні. Якщо результат логічної операції має значення "1", то біт установлюється, якщо результат логічної операції має значення "0", то біт скидається.

Перший оператор перевірки, який виконується за умовним оператором, має особливе значення, тому що він створює новий результат логічної операції. При цьому *старе* значення результату логічної операції RLO губиться. *Первинне опитування завжди відповідає початку логічної операції.* Алгоритм первинного опитування (AND, OR або Exclusive OR) не відіграє при цьому ніякої ролі.

Якщо датчик, підключений до входу, має нормально розімкнуті контакти, то рівень "1" установлюється при його активації. Якщо датчик має нормально замкнені контакти, то рівень "1" є присутнім на вході в неактивному стані датчика.

Тип використовуваного датчика необхідно враховувати при розробці програми. У випадку, якщо один з нормально розімкнутих контактів замінити

на нормально замкнений, то для збереження логіки керування (керування активними значеннями сигналів) потрібно замінити оператор перевірки A на AN. Із цього випливає, що врахування типу датчика необхідно для правильного вибору оператора перевірки.

При програмуванні складних двійкових логічних операцій оператор AND має більш високий пріоритет і виконується перед операторами OR і Exclusive OR, які мають однаковий пріоритет. Для збереження необхідного порядку обчислення складного логічного вираження іноді потрібно тимчасово зберегти значення RLO у деякій точці програми. Із цією метою можуть використовуватися вкладені вираження. Як і при записі виражень булевої алгебри, вкладені оператори забезпечують певний порядок виконання функцій.

У мові програмування STL можна використовувати наступні вкладені оператори:

O функція OR для вкладення функцій AND;
A(відкриваюча дужка з функцією AND;
O(відкриваюча дужка з функцією OR;
X(відкриваюча дужка з функцією Exclusive OR;
AN(відкриваюча дужка з функцією NOT-AND;
ON(відкриваюча дужка з функцією NOT-OR;
XN(відкриваюча дужка з функцією Not-Exclusive OR;
) закриваюча дужка.

Коли CPU зустрічає *відкриваючу* дужку, він запам'ятовує поточне значення RLO, а потім обробляє вираження в дужках, тобто вкладене вираження.

Коли CPU зустрічає *закриваючу* дужку, він зв'язує значення RLO для вкладеного вираження з раніше запам‘ятованим значенням RLO. Це зв'язування здійснюється згідно з функцією, яка записана при відкриваючій дужці.

Оператор перевірки, який виконується за *відкритою* дужкою, є *первинним опитуванням*, тому що CPU повинен створити новий результат логічної операції RLO *для вкладеного вираження*.

У практиці обчислення булевих виражень нерідко виникає необхідність в об'єднанні окремих функцій. При об'єднанні AND-функцій оператором OR логічні операції можуть бути записані в булевій алгебрі без використання дужок. Тут діє правило, згідно з яким функції AND виконуються в першу чергу, а функція OR, яка зв'язує функції AND, виконується в другу чергу.

Приклад:

```
A     Key0;
A     Key1;
O     ;
A     Var1;
A     Var2;
=     Out1;
```

У даному прикладі оператор O перебуває між двома функціями AND. Сигнал Out1 установлюється в "1", якщо (Key0 I Key1) АБО (Var1 I Var2) установлені в "1".

При об'єднанні OR оператором AND функції повинні бути записані в булевій алгебрі *з використанням дужок*, за допомогою яких вказується, що функції OR виконуються в першу чергу.

На рисунку 2.2 показане вікно STL-редактора із фрагментом програми, у якій застосовуються дужки для зміни порядку формування результату логічної операції RLO.

Особливості операцій бінарної логіки в LAD

Графічна мова програмування LAD заснована на виставі комутаційних схем. Елементами комутаційної схеми є нормально розімкнуті й нормально замкнені контакти, а також котушки реле. З'єднання цих елементів групується в ланцюг (network). Один або кілька ланцюгів Network 1, Network 2 і т.д. створюють розділ кодів логічного блоку. На рисунку 2.3 представлений вид вікна LAD-редактора. Зверніть увагу на те, що програмний код на рисунках 2.2-2.3 однаковий.

У мові LAD застосовуються два типи контактів – *нормально розімкнуті контакти типу NO*, які скануються з очікуванням стану «1», і *нормально замкнені контакти типу NC*, які скануються з очікуванням стану «0».

Ланцюг може складатися з одного контакту або ж великої кількості з'єднаних контактів. Ланцюг завжди повинен бути завершеним логічною змінною, наприклад, котушкою (coil). Котушка управляє двійковим операндом за допомогою *результату логічної операції RLO*.

До операнду можна звернутися через контакт, використовуючи абсолютну або символічну адресу.

Нормально розімкнутий контакт відповідає скануванню з очікуванням сигнального стану «1». Якщо струм тече в точці контактного плану, то це означає, що бітова логіка виконується й відповідний двійковий операнд має стан «1». Результат логічної операції (RLO) теж рівний «1».

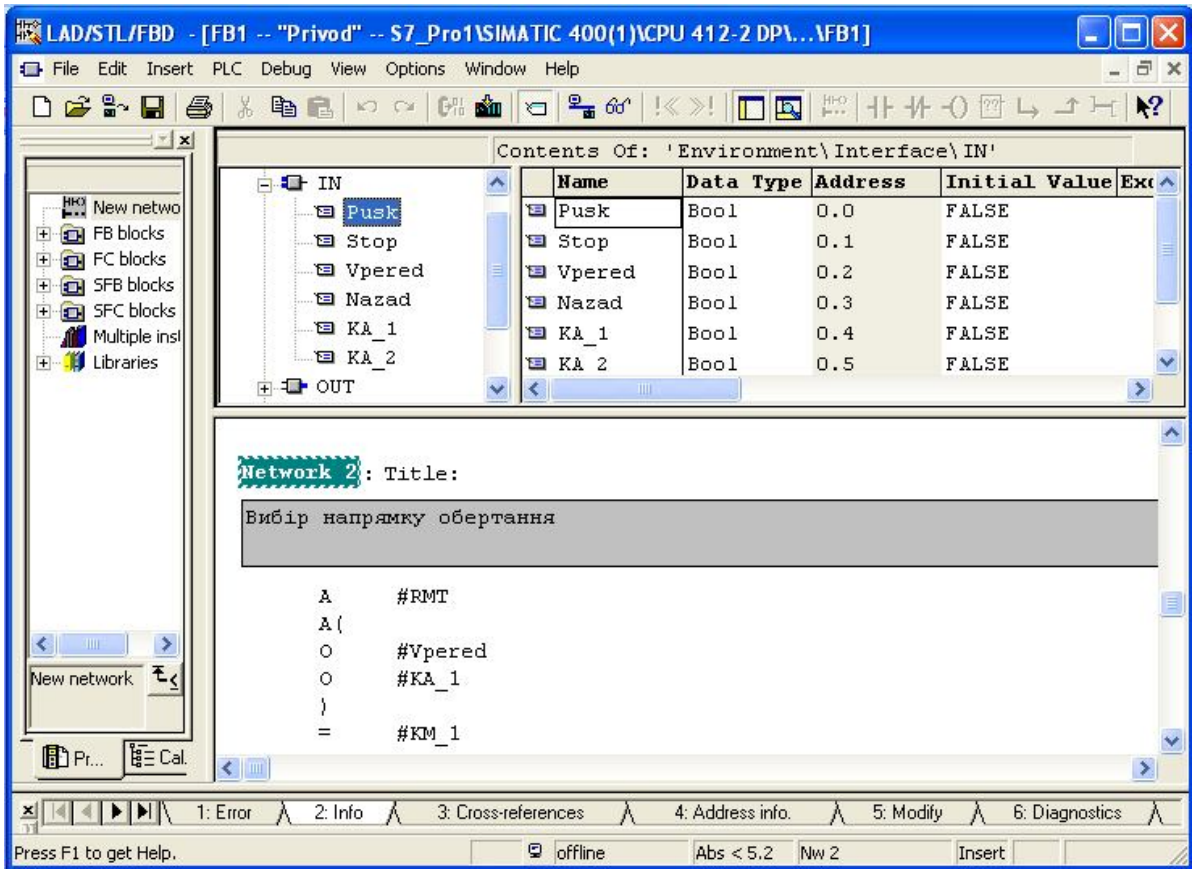


Рисунок 2.2 – Вид вікна програмування мовою STL

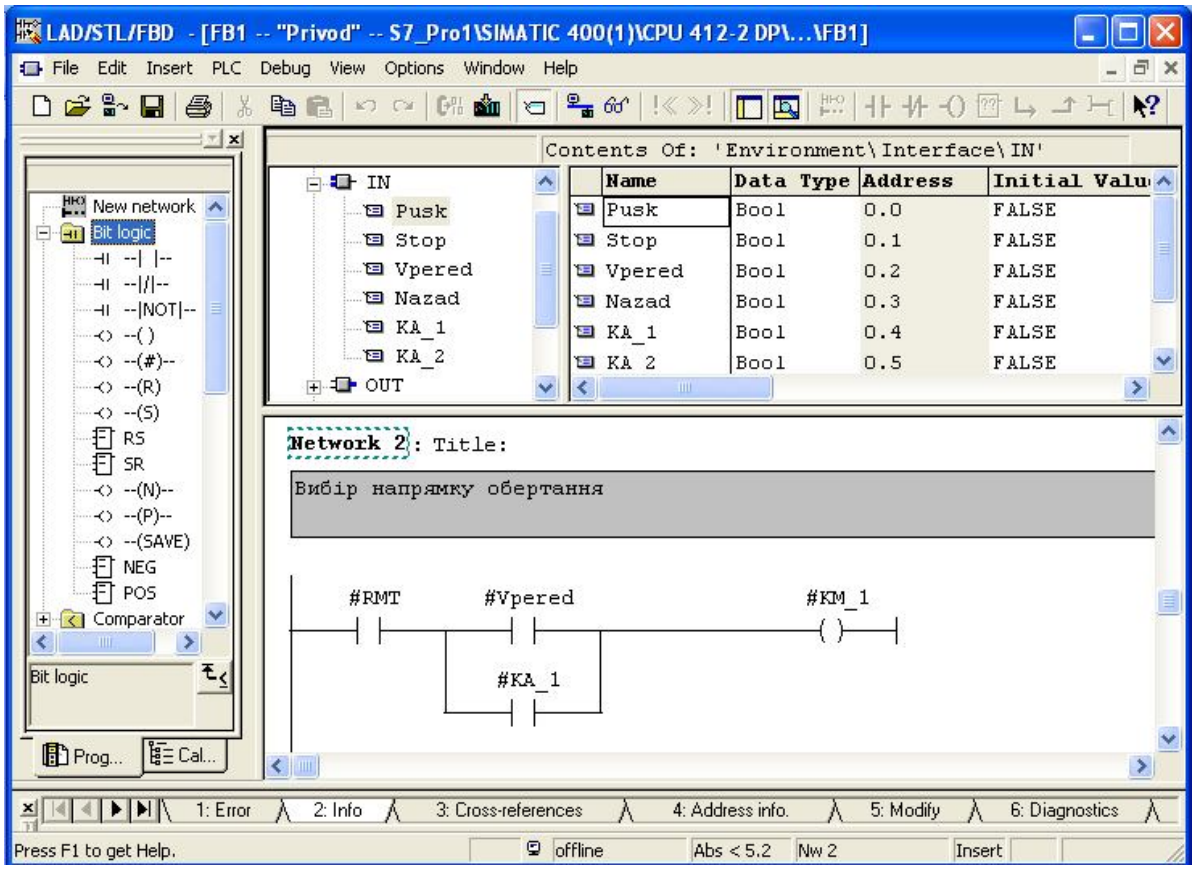


Рисунок 2.3 – Вид вікна програмування мовою LAD

Операції бінарної логіки в FBD

Мова програмування FBD (Function Block Diagram), інакше функціональний план, заснований на використанні графічних символів логічних елементів, відомих з булевої алгебри й мікросхемотехніки.

Приклад програми в FBD представлено на рисунку 2.4.

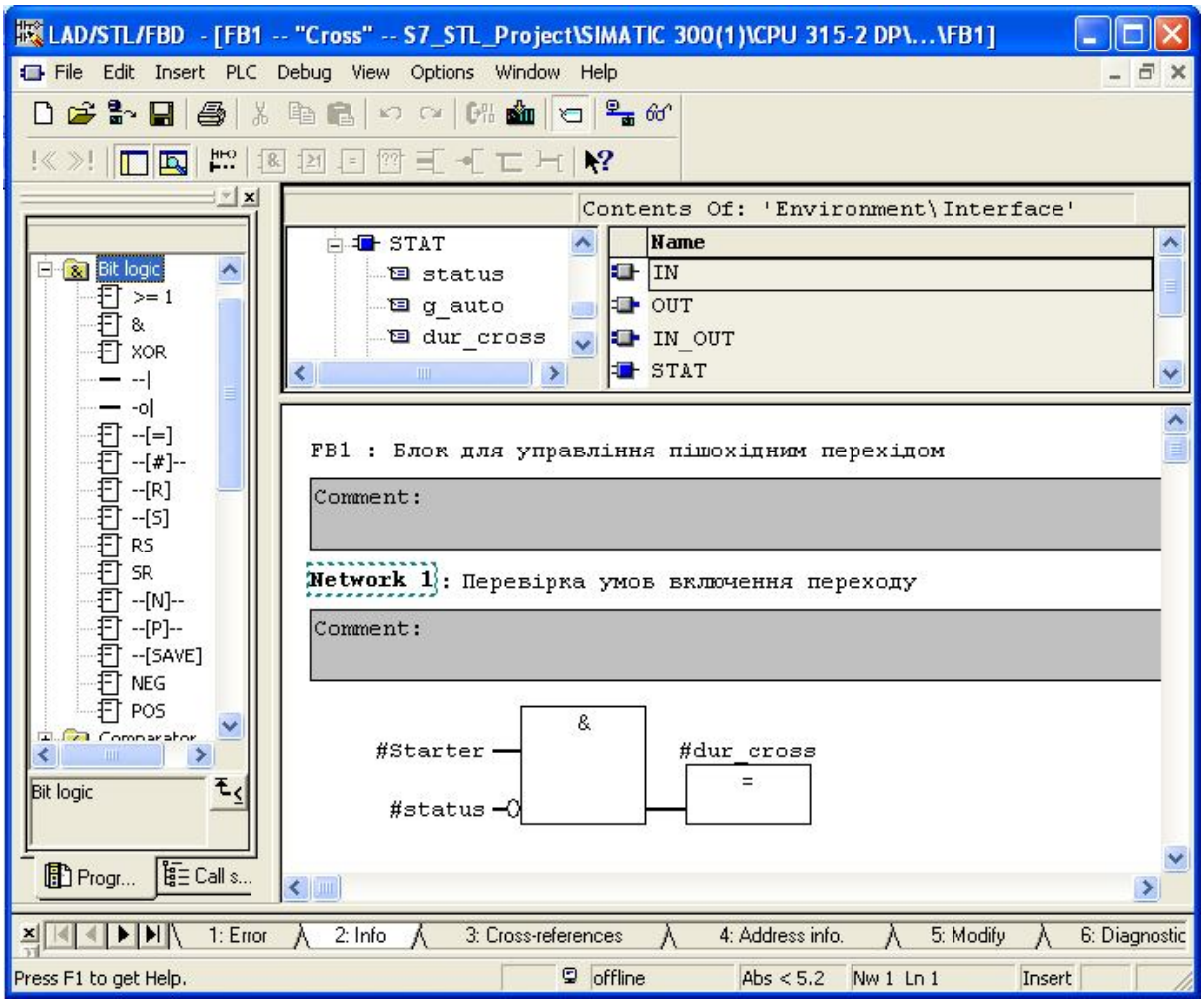


Рисунок 2.4 - Приклад програми в FBD

Для виконання логічних операцій до входів блоків можна підключити наступні операнди:

- вхідні й вихідні біти;
- меркери;
- таймери й лічильники;
- біти глобальних і локальних даних;
- біти слова стану (результати оцінки й обчислень).

Кожний операнд може бути адресований абсолютно або символічно. Логічна схема або логічна операція завжди повинна бути завершена оператором, наприклад, присвоювання. У результаті присвоювання бінарний

операнд одержує значення результату логічної операції (RLO).

Операнд може скануватися з очікуванням «1» або «0» (рис. 2.5). Сканування з очікуванням «0» розпізнається по символу заперечення “N” на вході функції.

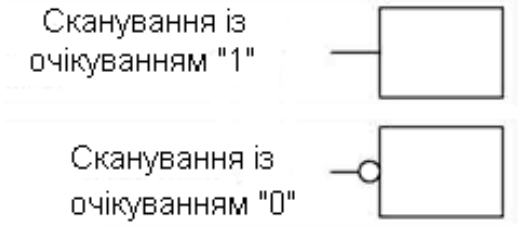


Рисунок 2.5 - Позначення сканування з очікуванням «1» і «0»

З погляду функціональності два методи сканування бінарних операндів дозволяють використовувати NO-контакти й NC-контакти ідентично.

Вихід бінарної функції завжди повинен бути приєднаний до наступного функціонального блоку. У найпростішому випадку можна просто з'єднати вихід із блоковим елементом Assign (*Присвоєння*). Щоб привласнити сигнальний стан бінарного операнда іншому бінарному операнду зазвичай використовується функція AND. При цьому операнд підключається до одного із входів функції, а інший вхід видаляється.

Бінарні функції можна вільно комбінувати, наприклад, можна об'єднати кілька функцій AND по функції OR. Кількість функцій у логічній операції (схемі або ланцюгу) теоретично не обмежено.

Приклад комбінації бінарної логіки представлено на рисунку 2.6.

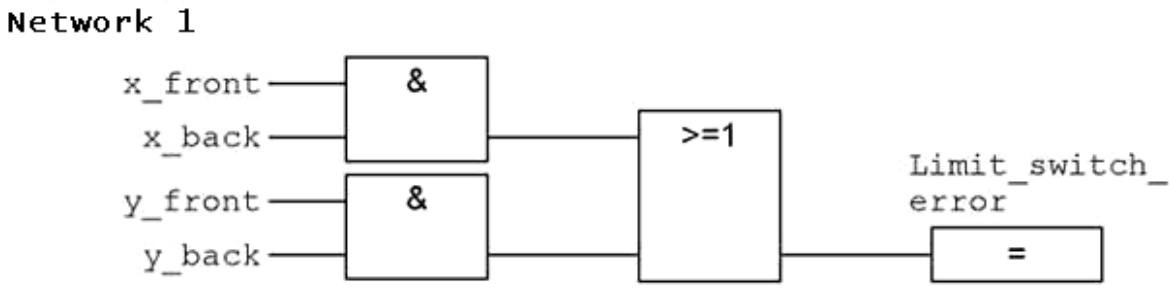


Рисунок 2.6 - Схема виявлення помилки кінцевих вимикачів

У ланцюзі Network 1 здійснюється спостереження кінцевих вимикачів (limit switches) осей X і Y, які не можуть бути активовані попарно. Схема формує повідомлення про помилку роботи кінцевих вимикачів.

Інвертування (заперечення) результату логічної операції можна використовувати при скануванні бінарного операнда, що еквівалентно скануванню з очікуванням сигнального стану «0», а також на виході бінарної функції, коли умова не виконана, тобто коли RLO = 0.

2.2 Програмування операції з пам'яттю й передачі даних

Операції з пам'яттю в STL

Операції з пам'яттю використовуються в поєднанні із двійковими логічними операціями, щоб впливати на значення сигналів (станів) біт з використанням результату логічної операції RLO.

До операцій з пам'яттю належать:

- функція Assign (*Присвоєння*) для *динамічного* керування бітами;
- функції *статичного* керування бітами – Set (*Установка біта*) і Reset (*Скидання біта*);
- функції перевірки наявності фронту сигналу.

Функція Assign має наступний синтаксис:

= *Bit*;

Ця функція привласнює RLO зазначеному біту.

Функції Set і Reset мають синтаксис:

S *Bit;*

R *Bit;*

Функції Set і Reset виконуються тільки у випадку, якщо результат логічної операції RLO має значення "1". Якщо результат логічної операції "0", то інструкції Set і Reset не міняють стани біта, зазначеного як операнд у цих інструкціях.

Для перевірки наявності фронту сигналу використовуються дві функції:

FP *Bit* – функція перевірки наявності переднього (зростаючого) фронту сигналу;

FN *Bit* – функція перевірки наявності заднього (спадаючого) фронту сигналу.

Наявність переднього фронту сигналу свідчить про перехід сигналу від рівня "0" до рівня "1". Відповідно, наявність заднього фронту сигналу свідчить про перехід сигналу від рівня "1" до рівня "0".

У логічних перемикаючих схемах еквівалентом функції перевірки наявності фронту сигналу є *контактний формувач імпульсу*. При включенні

реле цей формувач генерує імпульс, який свідчить про наявність зростаючого фронту сигналу. При вимиканні реле цей формувач генерує імпульс, який свідчить про наявність убутного фронту сигналу.

До біта, зазначеного як операнд, звертаються як до "меркеру фронту". Стан цього сигналу може бути перевірений в будь-який момент у наступних циклах сканування програми. Бітом у функції перевірки наявності фронту сигналу може бути меркер, біт із блоку глобальних даних або біт із статичних локальних даних функціональних блоків.

Отже, меркер фронту зберігає "старе" значення RLO, яке CPU використовував при останній перевірці наявності фронту сигналу. При кожній новій перевірці наявності фронту сигналу CPU порівнює поточне значення RLO зі станом меркера фронту. Фронт сигналу буде виявлений, якщо сигнали першої й другої перевірки будуть мати різні стани. Таким чином, стан біта "1" означає факт виявлення фронту сигналу. Цей стан біта зберігається, як правило, протягом одного циклу сканування програми, тому що при наступній перевірці CPU фронту сигналу не буде.

Перевірка наявності фронту у двійковій логічній операції може бути використана для керування таймером, лічильником або операцією з пам'яттю. Двійкові операції перевірки можуть розташовуватися між операцією перевірки наявності фронту й функцією, яка запускається.

Приклад:

```
O     Var_1;
O     Var_2;
FP    Merker_1;
A     Var_3;
S     Out_1;
A     Var_4;
FN    Merker_2;
R     Out_1;
```

У прикладі вихід Out_1 установлюється в момент, коли виконується OR-умова й вхід Var_3 установлений в "1". Вихід Out_1 скидається в момент, коли приходить негативний фронт на вхід Var_4.

Функції передачі даних в STL

У всіх операціях передачі даних використовується акумулятор – спеціальний регістр у процесорі, який виконує функції проміжного буфера.

Напрямок, у якому відбувається передача даних, вказується у використовуваній для передачі інструкції.

Функція завантаження складається з оператора L (код операції завантаження) і константи, змінної або адреси з ідентифікатором адреси, уміст якої функція буде завантажувати в акумулятор 1.

Приклади:

L +1500 Завантаження константи;

L IW 32 Завантаження слова із прямою адресацією;

L Actval Завантаження змінної (символьна адресація).

CPU виконує функцію завантаження незалежно від результату логічної операції RLO і біта стану. Функція завантаження не впливає на результат логічної операції й не впливає на біти стану. При завантаженні адреси, константи або змінної в акумулятор 1 поточний уміст акумулятора 1 пересилається в акумулятор 2. Попередній уміст акумулятора 2 при цьому губиться.

Функція вивантаження складається з оператора T і адреси в області пам'яті, по якій дані повинні бути відправлено з акумулятора 1.

Приклади:

T MW120; переносить уміст акумулятора в область пам'яті меркерів (байти 120 і 121) (застосована абсолютна адресація);

T Setpoint; переносить уміст акумулятора в змінну (застосована символьна адресація).

CPU виконує функцію вивантаження незалежно від результату логічної операції RLO і біта стану. Функція вивантаження не впливає на результат логічної операції й не впливає на біти стану.

Функція вивантаження пересилає вміст акумулятора 1 по одному байту, слову, або подвійному слову. При цьому вміст акумулятора 1 залишається незмінним (копіюється), що дозволяє багаторазово пересилати дані з акумулятора 1.

Для одночасної обробки двох чисельних значень потрібні два проміжні буфери. У ролі таких буферів виступають акумулятор 1 і акумулятор 2. Усі CPU мають такі спеціальні регістри. Крім того, CPU S7-400 мають два додаткових проміжних буфери – акумулятор 3 і акумулятор 4, які використовуються переважно в арифметичних операціях.

Кожний акумулятор має 32-розрядний вміст, тоді як усі області пам'яті мають байтову структуру. Обмін інформацією між областями пам'яті й акумулятором 1 може відбуватися побайтно, по 1 машинному слову й по 1 подвійному машинному слову.

В операціях пересилання можуть брати участь:

IB вхідний байт;

IW вхідне слово;
ID вхідне подвійне слово;
QB вихідний байт;
QW вихідне слово;
QD вихідне подвійне слово;
MB байт меркерів;
MW слово меркерів;
MD подвійне слово меркерів;
LB байт локальних даних;
LW слово локальних даних;
LD подвійне слово локальних даних;
DBB байт глобальних даних;
DBW слово глобальних даних;
DBD подвійне слово глобальних даних;
DIB байт в екземплярному DB;
DIW слово в екземплярному DB;
DID подвійне слово в екземплярному DB;
STW слово стану.

Функції акумуляторів дозволяють пересилати значення з одного акумулятора в інший або переміщати байти усередині акумулятора 1. До функцій акумуляторів належать:

PUSH Зрушення вмісту акумулятора "уперед";
POP Зрушення вмісту акумулятора "назад";
TAK Обмін умістом між акумуляторами 1 і 2;
ENT Зрушення вмісту акумулятора "уперед" (без акумулятора 1);
LEAVE Зрушення вмісту акумулятора "назад" (без акумулятора 1)

Перші три функції PUSH, POP і TAK використовуються в CPU, які мають тільки два акумулятори (S7-300 CPU). Усі п'ять функцій використовуються в CPU, які мають чотири акумулятори (S7-400 CPU).

Функції обміну байтами в акумуляторі 1:

CAW міняє місцями два байти в молодшому слові акумулятора 1. При цьому байти старшого слова акумулятора залишаються незмінними.

CAD міняє місцями всі байти в акумуляторі 1. При цьому самий старший байт стає самим молодшим по номеру, а середні два байти міняються місцями.

Для пересилання даних застосовуються також системні функції SFC 20 BLKMOV (копіювання області даних), SFC 21 FILL (вставка даних в область призначення) і SFC 81 UBLKMOV (безперервне копіювання області даних).

Елементи пам'яті LAD

Для роботи з пам'яттю в мові LAD доступні такі функції:

- одиночна котушка, якій привласнений (призначений) RLO;
- котушки R і S, як операції з пам'яттю;
- блокові елементи RS і SR, як функції, які працюють із пам'яттю;
- коннектори (midline outputs), як проміжні буфери;
- котушки P и N, як елементи виявлення фронту імпульсу;
- блокові елементи POS і NEG для виявлення фронту операндів.

Одиночна котушка, як термінатор, тобто завершальний елемент ланцюга, призначає потік електроенергії (електричний струм) прямо операнду, розташованому при котушці. Котушки установки й скидання S і R (set coil, reset coil) також можуть завершувати ланцюг.

Позначення котушок наведено на рисунку 2.7.

Рисунок 2.7 - Позначення котушок установки й скидання

Якщо струм тече в котушці установки S, то операнд, записаний над котушкою, установлюється в сигнальний стан «1». Якщо струм тече в котушці скидання, то операнд над котушкою має сигнальний стан «0». При відсутності струму в котушці установки або скидання бінарний операнд залишається незмінним.

Функції котушок установки й скидання можуть бути об'єднані в блоковому елементі функції для роботи із блоком пам'яті (memory box), як показано на рисунку 2.8. Загальний бінарний операнд розташовується над блоковим елементом.

Вхід S (set input) блокового елемента в цьому випадку відповідає котушці установки, вхід R (reset input) – котушці скидання. Сигнальний стан двійкового операнда, призначеного функції для роботи з пам'яттю, перебуває на виході Q функції пам'яті.

Блокові елементи SR і RS відрізняються пріоритетом входу скидання. Якщо на обидва входи одночасно подається рівень «1», то функції пам'яті реагують по-різному – функція пам'яті SR скидається, а функція пам'яті RS установлюється. Оскільки оператори виконуються послідовно, то в блоковому елементі SR CPU спочатку встановить операнд пам'яті, тому що вхід S

обробляється першим, однак потім знову скине його при обробці входу скидання. Функція пам'яті із пріоритетом скидання є «нормальною» формою функції для роботи з пам'яттю, тому що стан скидання (сигнальний стан «0») зазвичай безпечніше або менш ризикований.

Рисунок 2.8 - Блоки елементів пам'яті SR і RS

Коннектори (midline outputs) є проміжними буферами в контактному або функціональному планах. Коннектор є одиночною котушкою в точці ланцюга. Двійковий операнд над коннектором зберігає RLO для цієї точки. Сам коннектор не впливає на електричний струм.

Позначення коннектора наведено на рисунку 2.9.

Рисунок 2.9 - Позначення коннектора в LAD і FBD

Сканувати бінарний операнд над коннектором можна в іншій точці програми за допомогою NO- і NC-контактів (рис. 2.10). В одному ланцюзі можуть бути запрограмовані декілька коннекторов для різних точок.

Блокові елементи пам'яті в FBD

В FBD блокові елементи пам'яті використовуються з метою впливу на сигнальні стани бінарних операндів за допомогою результату логічної операції RLO.

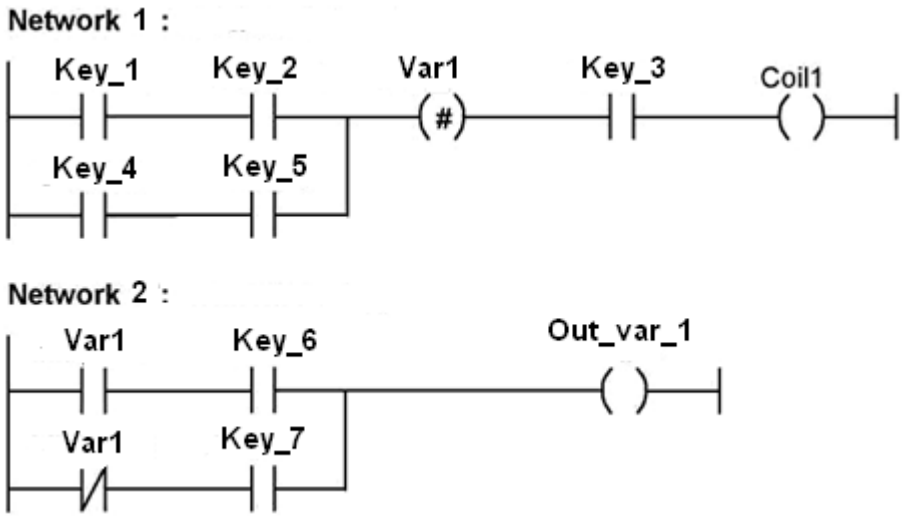


Рисунок 2.10 - Приклад використання коннектора в LAD

Для роботи з пам'яттю доступні такі функції:

- блоковий елемент присвоювання (assign box);
- блокові елементи установки S (set) і скидання R (reset), як індивідуально програмувальні функції пам'яті.
- блокові елементи RS і SR, як цілком закінчені функції пам'яті.
- блоковий елемент коннектора (midline output box), як проміжний буфер.
- блокові елементи P и N, як елементи оцінки (виявлення) фронту результату логічної операції.
- блокові елементи POS і NEG, як визначники фронту операндів.

Блоковий елемент присвоювання, як термінатор ланцюга, привласнює результат логічної операції безпосередньо операнду, суміжному із блоковим елементом.

Позначення блокового елемента присвоювання наведене на рис. 2.11.

Рисунок 2.11 - Позначення блокового елемента присвоювання

Блокові елементи установки й скидання також можуть завершувати логічну операцію (рис. 2.12). Ці блокові елементи активуються тільки тоді, коли результат логічної операції, яка направляється в блоковий елемент, рівняється «1».

Рисунок 2.12 - Позначення блокових елементів установки й скидання

Елементи оцінки фронту імпульсу в LAD і FBD

Виявлення фронту сигналу відбувається в програмі шляхом порівняння поточного RLO зі збереженим RLO. Якщо сигнальні стани різні, то це є наслідком фронту сигналу.

Для оцінки фронту застосовуються чотири елементи (рис. 2.13):

- меркер позитивного фронту;
- меркер негативного фронту;
- позитивний фронт операнда;
- негативний фронт операнда.

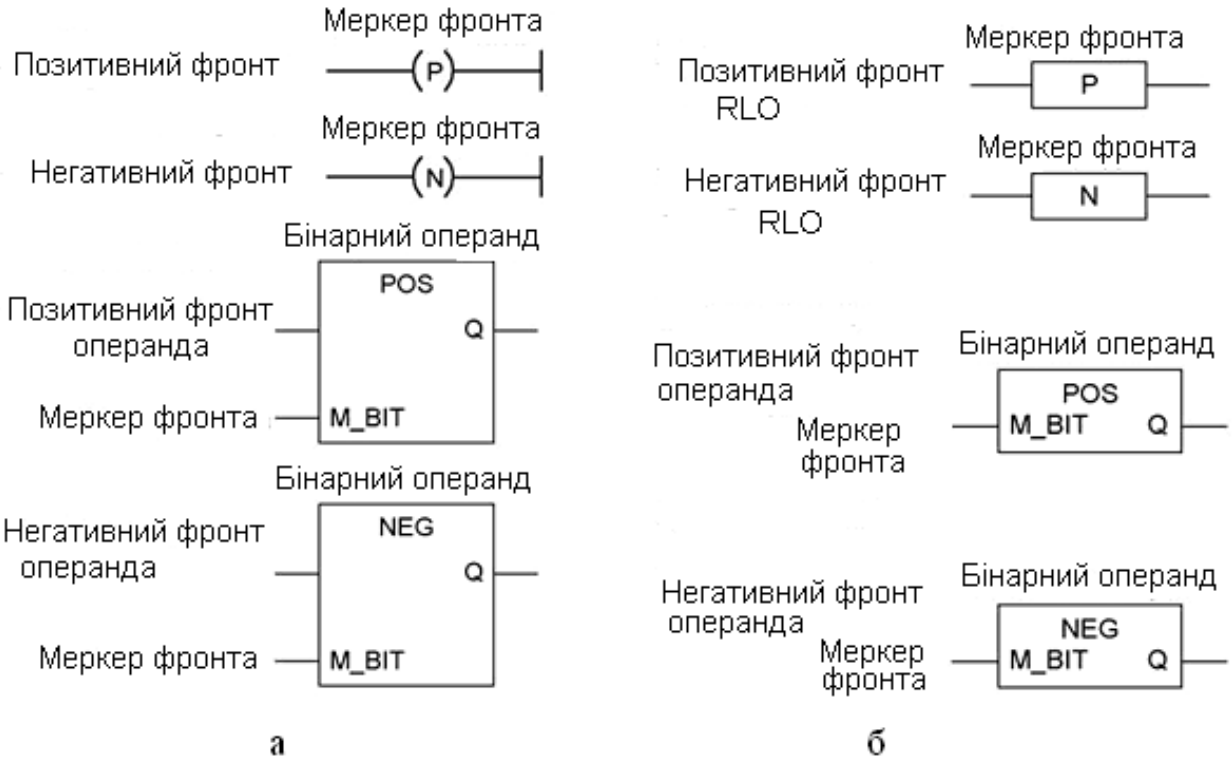


Рисунок 2.13 - Елементи оцінки фронту в LAD (а) і FBD (б)

2.3 Програмування таймерів

Таймери використовуються для керування за часом, наприклад, для забезпечення заданого часу очікування, для виміру відрізків часу або для генерації імпульсів.

Користувачеві доступні наступні типи таймерів:

- Таймер з керованим імпульсом (Pulse timer);
- Таймер з розширеним імпульсом (Extended pulse timer);
- Таймер із затримкою включення (On-delay timer);
- Таймер із затримкою включення й пам'яттю (Retentive On-delay timer);
- Таймер із затримкою вимикання (Off-delay timer).

Позначення типів і тимчасові діаграми таймерів наведено на рис. 2.14.

Тип таймеру	На STL	На SCL	Сигнал запуску
Імпульсний таймер (Pulse timer)	SP	S_PULSE	t
Таймер із розширеним імпульсом (Extended pulse timer)	SE	S_PEXT	t
Таймер із затримкою імпульса (On-delay timer)	SD	S_ODT	t
Таймер із затримкою включення (Retentive On-delay timer)	SS	S_ODTS	t
Таймер із затримкою виключення (Off-delay timer)	SF	S_OFFDT	t

Рисунок 2.14- Тимчасові діаграми таймерів

Інтерфейс таймера становлять вхідні сигнали запуску, скидання, завдання тривалості, а також вихідні сигнали стану таймера і його поточних значень. Призначення входів і виходів наведені в табл. 2.1.

Особливості використання таймерів у мові STL

Таймер існує в STL-програмі як блоковий елемент. Завантаження значення часу здійснюється через акумулятор 1.

Таймер запускається, якщо відбулася зміна значення результату логічної операції RLO. При цьому для таймера із затримкою вимикання (Off-delay

timer) RLO повинен змінити свій стан зі значення "1" на "0", а для всіх інших типів таймерів RLO повинен змінити свій стан зі значення "0" на "1".

Таблиця 2.1 - Призначення входів і виходів таймера

Назва	Тип даних	Опис
S	BOOL	Вхід запуску
TV	S5TIME	Специфікація тривалості
R	BOOL	Вхід скидання
BI	WORD	Поточне значення у двійковому коді
BCD	WORD	Поточне значення в BCD-форматі
Q	BOOL	Стан таймера

При запуску таймера з акумулятора 1 вибирається час запуску (running time) або тривалість роботи (duration). Ці параметри рекомендується завантажувати в акумулятор 1 безпосередньо перед запуском або у вигляді константи, або у вигляді змінної.

Приклади завдання тривалості імпульсу таймера константою:

```
L     S5TIME#10s;   // Завантажити тривалість 10 с;
L     S5T#1m10ms;   // Завантажити тривалість 1 хв 10 мс.
```

Приклади завдання тривалості імпульсу таймера змінної:

```
L     S5T#10m;      //Задати тривалість 10 хв;
T     MW20;         //Зберегти тривалість роботи;
L     MW20;         //Завантажити тривалість роботи.
```

Внутрішня структура тимчасового параметра "тривалість імпульсу" складається зі значення часу й тимчасової бази. Тривалість імпульсу таймера рівняється добутку цих величин.

Значення тимчасової бази (величини кроку за часом) використовується операційною системою CPU для декремента таймера (рис. 2.15).

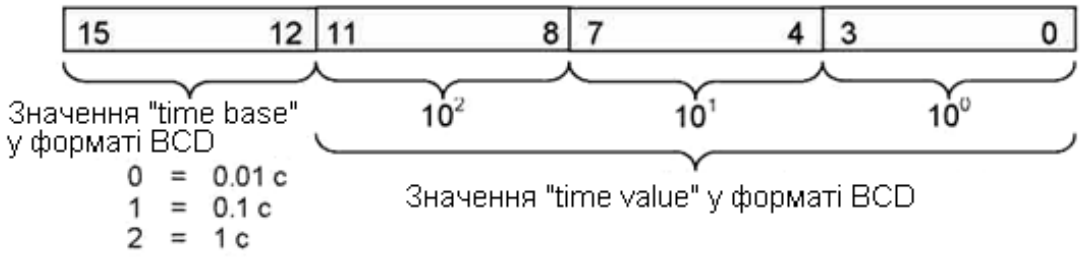


Рисунок 2.15 – Формування параметра "тривалість роботи"

Таким чином, зменшення значення тимчасової бази забезпечує більш точне обчислення проміжків часу згідно із заданим значенням тимчасової бази.

Слід ураховувати, що таймери можуть обновлятися асинхронно щодо процесу сканування програми користувача. При цьому стан таймера на початку циклу сканування відрізняється від його стану наприкінці циклу. Щоб зменшити помилку через асинхронне відновлення таймера в програмі, таймер необхідно обновляти тільки в одному місці програми.

Скидання таймера виконується по інструкції:

R T n;

По цій інструкції таймер скидається при RLO=1.

Запустити таймер знову можна інструкцією:

FR T n

В операційну систему CPU вбудовані IEC-функції таймерів. Вони доступні як системні функціональні блоки SFB:

- SFB 3 TP – генерація імпульсів;
- SFB 4 TON – генерація імпульсу із затримкою включення;
- SFB 5 TOF – генерація імпульсу із затримкою вимикання.

Параметри IEC-функцій таймерів представлені в табл. 2.2.

Таблиця 2.2 - Параметри IEC-функцій таймерів

Назва	Призначення	Тип даних	Опис
IN	INPUT	BOOL	Вхід запуску (Start input)
PT	INPUT	TIME	Тривалість імпульсу (Pulse length) або затримка включення (Delay duration)
Q	INPUT	BOOL	Стан таймера (Timer status)
ET	INPUT	TIME	Минулий час (Elapsed time)

Особливості програмування таймерів у мовах LAD і FBD

У програмах LAD і FBD таймер представляється або як блоковий елемент, як показано на рисунку 2.16, або окремими програмними елементами. Вистава окремих програмних елементів LAD показана на рисунку 2.17, а вистава окремими програмними елементами в FBD – на рисунку 2.18.

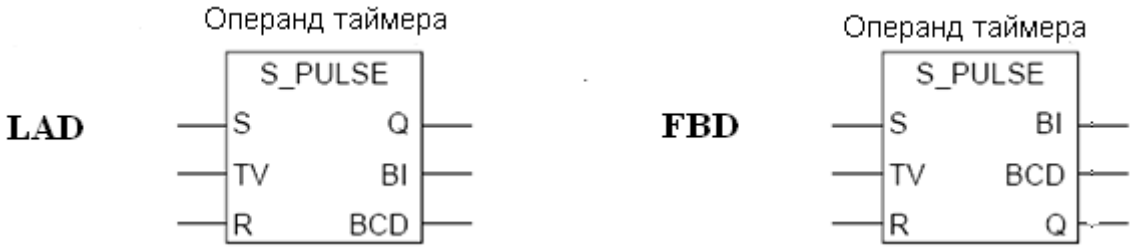


Рисунок 2.16 - Вистава таймера блоковим елементом

Над блоковим елементом розташований абсолютний або символічна адреса таймера. У самому блоковому елементі, як заголовок, зазначений режим таймера (S_PULSE означає Start pulse або запуск імпульсу).

Таймер запускається, коли результат логічної операції (RLO) міняється на вході запуску S (start input) блокового елемента або перед котушкою. Таймери запускаються при зміні RLO з «0» на «1», однак для таймера *затримки вимикання* RLO повинен помінятися з «1» на «0».

Рисунок 2.17 – Вистава таймера окремими елементами в LAD

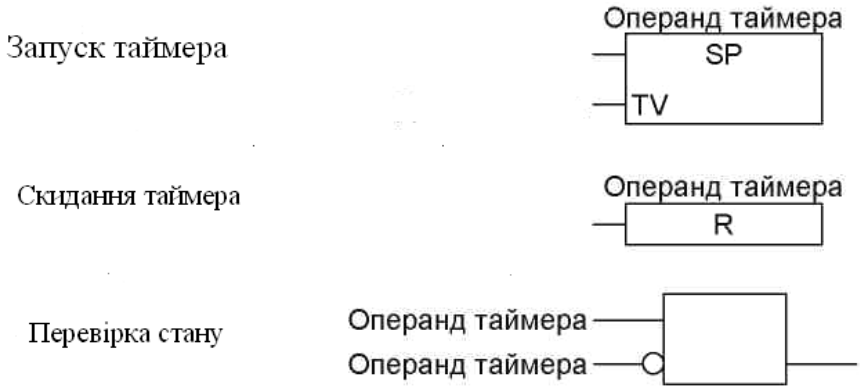


Рисунок 2.18 – Вистава таймера окремими елементами в FBD

Як завдання тривалості таймер ухвалює значення, зазначене під котушкою (блоковим елементом) або на вході TV. Задати тривалість можна

константою, операндом розміром у слово або змінної типу S5TIME.

Слід урахувати, що призначення для входів S і TV обов'язкові.

На рисунку 2.19 показаний приклад LAD-програми запуску таймера T1 із часом, який визначений операндом «Del_1».

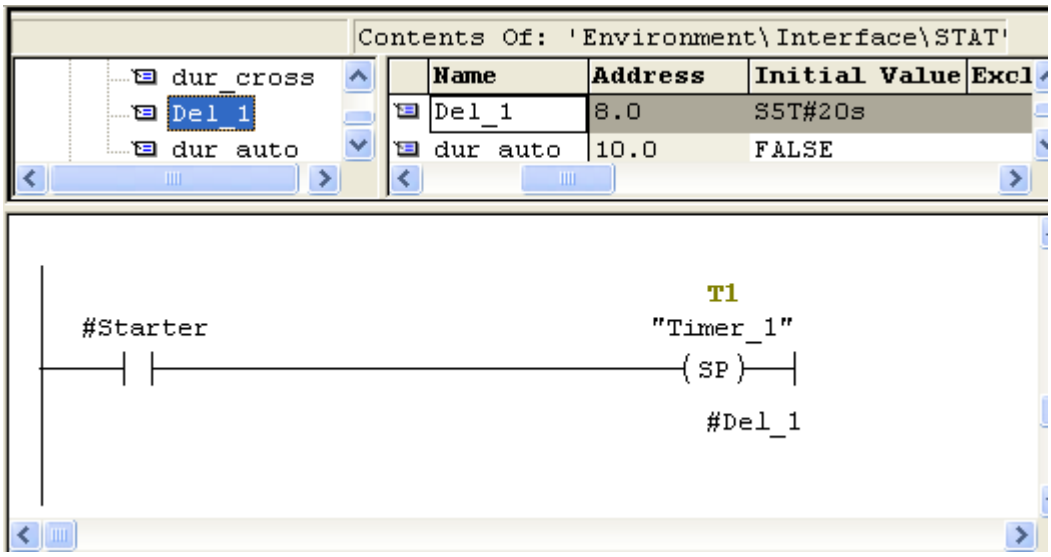


Рисунок 2.19 – Приклад запуску таймера, представленого котушкою

В LAD-програмі таймер скидається, коли електричний струм тече на вході скидання або в котушці скидання. В FBD-програмі таймер скидається, коли на вході скидання присутній сигнал «1». У скинутому стані перевірка стану таймера поверне «0».

Виходи BI і BCD (рис. 2.16) містять значення часу таймера у двійковому (BI) і двійково-десятковому (BCD) форматі. Значення виходу рівно поточному в момент зчитування. Якщо таймер активний, то значення часу відлічується від установленого убік зменшення (до нуля). Значення зазвичай зберігається в заданому операнді, тобто передаються до блокового елемента MOVE.

2.4 Програмування лічильників

Функції лічильників дозволяють вирішувати завдання рахунку безпосередньо в CPU. Лічильники дозволяють виконувати прямий і зворотний рахунок. При виконанні рахунку використовується "тридекадний" формат значення лічильника, який визначає діапазон значень від 000 до 999.

Швидкість рахунку лічильників залежить від часу сканування програми. Це пов'язане з тим, що для забезпечення процесу рахунку CPU повинен

виявляти на вході лічильника зміни вхідного сигналу, інакше кажучи, вхідний імпульс (або пауза) повинен бути присутнім на вході принаймні один цикл сканування програми. Таким чином, чим триваліше цикл сканування програми, тем повільніше швидкість рахунку лічильника.

Лічильник представляється зазвичай блоковим елементом, який показано на рисунку 2.20.

Рисунок 2.20 - Блоковий елемент лічильника

Блоковий елемент лічильника містить усі операції рахунку у формі функціональних входів і функціональних виходів. Над блоковим елементом розташований операнд – абсолютна або символічна адреса лічильника. У блоковому елементі, як заголовок, вказується тип лічильника. Наприклад, S_CUD – лічильник прямого й зворотного рахунку.

Призначення входів і виходів лічильника представлено в таблиці 2.3.

Таблиця 2.3

Назва	Тип даних	Опис
CU	BOOL	Вхід прямого рахунку
CD	BOOL	Вхід зворотного рахунку
S	BOOL	Вхід установки
PV	WORD	Вхід передустановки
R	BOOL	Вхід скидання
CV	WORD	Поточне значення у двійковому коді
CV_BCD	WORD	Поточне значення в BCD-коді
Q	BOOL	Стан лічильника

Програмування лічильника мовою LAD

При програмуванні лічильника можна включати режим прямого або зворотного рахунку, зупиняти лічильник, задавати для лічильника початкове значення (initial value). Існує також можливість визначати стан лічильника.

Установка лічильника здійснюється по інструкції:

S C n

Лічильник установлюється, якщо RLO переходить від "0" до "1" перед операцією установки S, тобто для установки лічильника завжди потрібен позитивний фронт.

Установити лічильник – це значить завантажити в лічильник початкове значення. Початкове значення (від 0 до 999), яке завантажується в лічильник, повинне перебувати в акумуляторі 1.

Для завдання константи можна використовувати C# або W#16# (тільки з десятковими числами).

Приклад:

```
L     C#200;          //Завантажити значення лічильника 200
T     MW 36;          //Зберегти значення лічильника
```

Скидання лічильника здійснюється командою:

R C n

Лічильник скидається, якщо RLO має значення "1" перш ніж у програмі зустрінеться операція скидання R. Скидання встановлює лічильник у нульове значення.

Прямий рахунок виконується по інструкції:

CU C n

Зворотний рахунок виконується по інструкції:

CD C n

Приклади програмування лічильника під номером C2

1) установка прямого рахунку лічильника C2:

```
A   I 2.1
CU  C 2
```

2) установка зворотного рахунку лічильника C2:

```
A   I 2.2
CD  C 2
```

3) установка лічильника C2 із завантаженням константи 500:

```
A   I 2.3
L   C#500
S   C 2
```

4) скидання лічильника C2:

```
A   I 2.4
R   C 2
```

5) опитування лічильника C2 з передачею рівня на вихід Q 13.0:

```
A   C 2
=   Q 13.0
```

Особливості програмування лічильника в LAD і FBD

Лічильник установлюється, коли сигнал на вході установки S міняється з «0» на «1». Коли лічильник установлюється, він ухвалює значення на вході PV або значення під котушкою установки.

Лічильник скидається, коли струм протікає на вході скидання або в котушці скидання. У цьому випадку перевірка лічильника NO-контактом поверне результат зчитування «0», а перевірка NC-контактом поверне «1».

Стан лічильника подається на вихід Q блокового елемента лічильника. Стан лічильника можна перевірити з використанням NO-контакту або NC-контакту. Вихід Q містить значення «1», коли поточний рахунок більше нуля. На виході Q утримується «0», якщо поточне значення рахунку дорівнює нулю. Вихід Q у блоковому елементі лічильника можна не підключати.

IEC-функції лічильників

IEC-функції лічильників (IEC Counter Functions) вбудовані в операційну систему CPU як системні функціональні блоки SFB.

При розробці програми доступні наступні функції лічильників:

- SFB 0 CTU – функція прямого рахунку;
- SFB 1 CTD – функція зворотного рахунку;
- SFB 2 CTUD – функція прямого й зворотного рахунку.

Ці SFB можна викликати з екземплярними блоками даних або використовувати їх як локальні екземпляри у функціональному блоці.

2.5 Використання функцій порівняння

Функції порівняння використовуються при обробці чисел типів INT, DINT і REAL. Їх перелік і позначення наведено в таблиці 2.4.

Таблиця 2.4 - Функції порівняння для даних типу INT, DINT і REAL

Операція порівняння	Тип даних		
	INT	DINT	REAL
Рівно	==I	==D	==R
Не рівно	<>I	<>D	<>R
Більше чим	>I	>D	>R
Більше чим або рівно	>=I	>=D	>=R
Менше ніж	<I	<D	<R
Менше ніж або рівно	<=I	<=D	<=R

Програмування операцій порівняння в LAD

При програмуванні функцій порівняння необхідно керуватися наступною схемою операцій:

1) завантаження першого числа;
2) завантаження другого числа;
3) застосування функції порівняння;
4) присвоєння результату.

При виконанні першої операції число завантажується в акумулятор 1. При виконанні другої операції вміст акумулятора 1 переміщається в акумулятор 2, а число із другої адреси завантажується в акумулятор 1. Тепер операція порівняння виконується між умістом двох акумуляторів.

Функція порівняння повертає двійковий результат типу BOOL, який може бути призначений двійковій змінній. Функції порівняння не міняють умісту акумуляторів. Вони завжди виконуються поза всяким зв'язком з якими-небудь умовами.

Приклад:

```
L     #actval;     //Завантаження першої змінної
L     #calibra;    // Завантаження другої змінної
>=R;               //Перевірка умови “більше або рівно”
S     #recali;     //Установка змінної recali в “1”
```

Функції порівняння повертають двійковий результат логічної операції RLO і можуть бути використані в поєднанні з іншими двійковими функціями. Функція порівняння на початку логічної операції завжди є первинним опитуванням (first check). Значення RLO, яке вертається функцією порівняння, може бути безпосередньо використано в логічних операціях.

Приклад:

```
L     MW 120;
L     512;
>     I;
A     Input1;
=     Output1;
```

У прикладі вихід *Output1* установлюється, якщо виконується умова порівняння, а вхід *Input1* при цьому має стан "1".

Функція порівняння усередині логічної операції повинна бути взята в дужки, тому що ця функція починає новий логічний етап – первинне опитування.

Приклад:

```
O     Input2;
O     (;
L     MW 12;
L     200;
<=    I;
);
O     Input3;
=     Output2;
```

У цьому прикладі вихід *Output2* установлюється, якщо вхід *Input2* або вхід *Input3* має стан "1", або якщо виконується умова порівняння.

Програмування операцій порівняння в LAD і FBD

На рисунку 2.21, як приклад, показаний блоковий елемент операції «рівно».

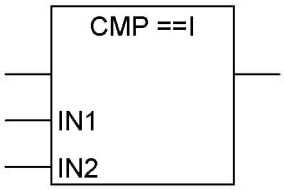


Рисунок 2.21 - Позначення блокового елемента порівняння «рівно»

Блоковий елемент функції порівняння має два входи IN1 і IN2, а також немарковані вхід і вихід. Немарковані вхід і вихід служать для приєднання інших (двійкових) програмних елементів.

Заголовок CMP у блоковому елементі ідентифікує операцію порівняння (compare) і тип виконуваного порівняння.

У ланцюзі LAD компаратор (блок порівняння) можна поставити замість контакту. Порівнювані (цілочисельні) значення подаються на входи IN1 і IN2, результат порівняння виводиться у вигляді булевої змінної на виході. Успішне порівняння еквівалентне замкненому контакту (струм протікає через компаратор). Якщо порівняння не успішне, то контакт розімкнутий. Вихід компаратора завжди повинен бути підключений.

У прикладі на рисунку 2.22 меркер M 99.0 скидається, якщо значення в слові MW 92 рівно 120, інакше він залишається без змін.

Контакти можна приєднати до й після функції порівняння. Самі блокові елементи порівняння можуть бути з'єднані послідовно або паралельно.

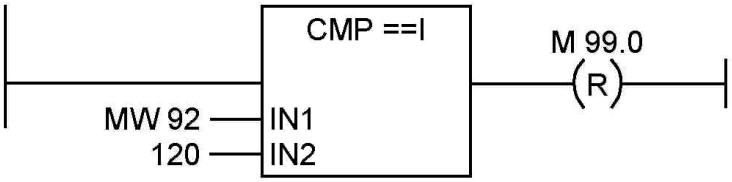


Рисунок 2.22 - Приклад застосування функції порівняння чисел INT

У випадку послідовного з'єднання функцій порівняння обоє порівняння повинні бути успішними, щоб у ланцюзі протікав струм.

У паралельній схемі струм буде протікати при виконанні однієї з умов.

2.6 Програмування арифметичних і математичних функцій

Програмування арифметичних функцій

Арифметичні функції забезпечують виконання базових арифметичних операцій із двома числовими значеннями, одне з яких перебуває в акумуляторі 1, а друге – в акумуляторі 2. Результат арифметичної операції записується в акумулятор 1. Біти стану CC0, CC1, OV і OS забезпечують інформацію, яка стосується виконання обчислень і результату.

Огляд доступних користувачеві арифметичних функцій наведено в таблиці 2.5.

Таблиця 2.5 - Арифметичні функції

Назва функції	Тип даних		
	INT	DINT	REAL
Додавання (Addition)	+I	+D	+R
Вирахування (Substraction)	-I	-D	-R
Множення (Multiplication)	*I	*D	*R
Розподіл (Division with quotient as result)	/I	/D	/R
Залишок від розподілу (Division with remainder as result)	-	MOD	-

Арифметичні функції програмуються за схемою:

1) завантаження першого числа;
2) завантаження другого числа;
3) виконання арифметичної функції;
4) запис результату.

При виконанні першої операції число записується в акумулятор 1. При завантаженні другого числа спочатку вміст акумулятора 1 переміщається в акумулятор 2, а потім друге число завантажується в акумулятор 1. Після цього виконується арифметична операція, у якій беруть участь два акумулятори. Результат операції зберігається в акумуляторі 1. Арифметичні функції виконуються поза всяким зв'язком з якими-небудь умовами.

Приклад програмування арифметичної операції:

```
L     MW12;     //Завантаження першого числа
L     250;      // Завантаження другого числа
/I;             //Поділ другого числа на перше (ціле число)
T     MW120;    //Запис результату
```

Інструкція поділу **/I** інтерпретує вміст молодших слів акумуляторів 1 і 2 як числа цілого типу (INT). Інструкція виконує поділ числа, яке перебуває в акумуляторі 2 (ділене), на число, яке перебуває в акумуляторі 1 (дільник), і зберігає обидва результати поділу – частку (молодше слово) і залишок (старше слово). Обидва значення мають тип INT.

Частка являє собою цілий результат операції розподілу. Вона рівняється нулю у двох випадках:

1) ділене рівняється нулю;
2) ділене менше ніж дільник.

Частка від розподілу негативна, якщо дільник менше нуля.

Після виконання інструкції біти стану CC0 і CC1 показують, яка частка від розподілу – негативна, дорівнює нулю або позитивна. Біти стану OV і OS указують на порушення дозволеного діапазону.

У випадку розподілу на нуль частка від розподілу й залишок вертаються з нульовими значеннями, а біти стану CC0, CC1, OV і OS установлюються в "1".

Якщо потрібно виконати кілька арифметичних операцій, то їх можна запрограмувати послідовно. У цьому випадку результат виконання першої операції використовується для обробки в наступній операції. Тимчасове зберігання даних забезпечується акумуляторами.

Приклад:

Result1:= Value1+Value 2-value3

```
L     Var1;
L     Var2;
+I;                   //Var1 + Var2 = сума
L     Var3;
-I;                   // Сума – Var3
```

```
T       Result1;
```

При виконанні арифметичної операції в CPU із двома акумуляторами перше завантажене число перебуває в акумуляторі 2 і може бути використане без повторного завантаження.

Приклад:

*Result2: = Var4 * (Var5)2*

```
L       Var5;
L       Var4;
*D;                     //Var4 * Var5
*D;                     //Var4 * Var5 * Var5
T       Result2;
```

Операції декремента й інкремента

Синтаксис інструкцій:

```
DEC     n       //Декремент
INC     n       //Інкремент
```

Ці операції слід програмувати за наступною схемою:

1) Завантаження адреси;
2) Вказівка операції (декремент або інкремент) і кроку;
3) Передача результату Result;

Операції декремента й інкремента виконуються незалежно від RLO.

Приклади операцій:

```
L       Incvar;
INC 5;
T       Incres;
```

У прикладі значення змінної Incvar збільшується на 5 і результат передається в змінну Incres.

```
L       Decvalue;
DEC 7;
T       Decres;
```

У прикладі значення змінної Decvalue зменшується на 7 і результат зберігається в змінній Decres.

Програмування математичних функцій

До математичних функцій належать наступні функції:

- синус (SIN), косинус (COS), тангенс (TAN);
- арксинус (ASIN), арккосинус (ACOS), арктангенс (ATAN);
- зведення у квадрат (SQR), добування квадратного кореня (SQRT);
- експонента (EXP), логарифм (LN).

Усі математичні функції обробляють числа у форматі REAL. Залежно від результату ці функції встановлюють біти стану CC0, CC1, OV і OS.

Як вхідне значення математичні функції використовують число, яке перебуває в акумуляторі 1. Це число обробляється згідно з інструкцією функції й знову зберігається в акумуляторі 1.

Математичні функції міняють тільки вміст акумулятора 1, уміст усіх інших акумуляторів залишається незмінним. Математичні функції завжди виконуються поза всяким зв'язком з якими-небудь умовами.

Приклади математичних функцій:

```
L     MD 10;     //Значення кута в подвійному слові
SIN;
T     MD 14;     //Запис синуса кута в подвійному слові
…     …
L     #exponent;
EXP;
T     #result;
```

Математичні функції виконуються згідно із правилами обробки чисел типу REAL, навіть коли застосовується абсолютна адресація й тип вхідного числа не описаний.

Особливості програмування арифметичних і математичних функцій в LAD і FBD

Арифметичні функції представляються в програмах блоковими елементами. Приклад арифметичного блокового елемента для функції підсумовування даних типу INT наведено на рисунку 2.23.

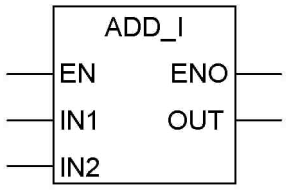


Рисунок 2.23 - Позначення блокового елемента підсумовування даних типу INT

Блоковий елемент арифметичної функції крім входу дозволу EN (enable input) і виходу дозволу ENO (enable output) має два входи IN1 і IN2, а також вихід OUT. Заголовок у блоковому елементі ідентифікує арифметичну дію, яка виконується. Позначення арифметичних функцій відповідно до типу даних наведені у таблиці 2.6.

Таблиця 2.6 - Позначення арифметичних функцій

Арифметична функція	Відповідність типу даних		
	INT	DINT	REAL
Додавання	ADD_I	ADD_DI	ADD_R
Вирахування	SUB_I	SUB_DI	SUB_R
Множення	MUL_I	MUL_DI	MUL_R
Розподіл (результат – частка)	DIV_I	DIV_DI	DIV_R
Розподіл із залишком	-	MOD_DI	-

Значення, які обчислюються, подаються на входи IN1 і IN2, результат обчислення виводиться на вихід OUT. Використовувані змінні повинні бути того самого типу даних, що й входи елемента.

Арифметична функція виконується, якщо на вході дозволу присутній сигнал «1». Якщо під час обчислення виникає помилка, то вихід дозволу встановлюється в «0. Якщо виконання функції не дозволене (EN = 0), то обчислення не виконується. Якщо головне реле керування (MCR) активоване, то вихід OUT установлюється в нуль.

Під час виконання арифметичної функції можуть виникнути наступні помилки:

- переповнення в обчисленнях з типами INT і DINT;
- зникнення значущих розрядів і переповнення в обчисленнях з типом REAL;
- неприпустиме число REAL в обчисленнях з типом REAL.

Приклад обчислення даних типу INT

На рисунку 2.24 значення слова в пам'яті меркерів MW 100 ділиться на 250, цілочисельний результат зберігається в меркері MW 102.

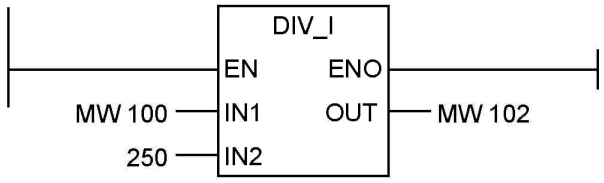


Рисунок 2.24 - Приклад застосування функції розподілу чисел INT

Арифметичний блоковий елемент може бути поміщений у будь-якому місці ланцюга. Пряме з'єднання з лівою (живильною) напрямною дозволяє підключити арифметичні блокові елементи паралельно. При цьому блокові

елементи самого верхнього ланцюга обробляються зліва направо, потім зліва направо обробляються елементи другого ланцюга і так далі. Для завершення ланцюга потрібно встановити котушку, якій можна призначити, наприклад, біт тимчасових локальних даних.

Якщо вихід ENO попереднього блокового елемента з'єднаний із входом EN наступного, то останній спрацьовує тільки тоді, коли попередній елемент був оброблений без помилок.

Математичні функції

В LAD і FBD представлені наступні математичні функції:

- синус, косинус, тангенс;
- арксинус, арккосинус, арктангенс;
- зведення у квадрат, добування квадратного кореня;
- експонентна функція по підставі *e*, натуральний логарифм.

Усі математичні функції працюють із числами типу REAL.

Блоковий елемент математичної функції має вхід IN і вихід OUT, а також вхід дозволу EN і вихід дозволу ENO. Заголовок у блоковому елементі ідентифікує виконувану математичну функцію.

Приклад позначення блокового елемента математичної функції *синус* наведено на рисунку 2.25.

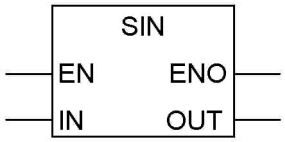


Рисунок 2.25 - Позначення блокового елемента "синус"

Вхідне значення подається на вхід IN, а результат математичної функції виводиться на вихід OUT. Вхід і вихід ставляться до типу даних REAL. Операнди, які адресуються з використанням абсолютних адрес, повинні мати розмір подвійного слова.

Математична функція виконується, якщо на вході дозволу (EN) присутній сигнал «1». Якщо при обчисленні виникне помилка, то вихід дозволу ENO установлюється в «0». Якщо виконання функції не дозволене (EN = 0), то обчислення не виконується й ENO також стає рівним «0».

Якщо головне реле керування (MCR) активно, то при виконанні математичної функції (EN = 1) вихід OUT установлюється в нуль. При цьому MCR не впливає на ENO.

У математичній функції можуть виникнути наступні помилки:

- вихід за межі діапазону й переповнення;
- неприпустиме число типу REAL як вхідне значення.

Приклад застосування математичної функції

Обчислення функції *синус* показано на рисунку 2.26. Значення подвійного слова пам'яті (меркерів) MD 110 містить величину в радіанах. Від цього значення береться синус і зберігається в подвійному слові пам'яті MD 104.

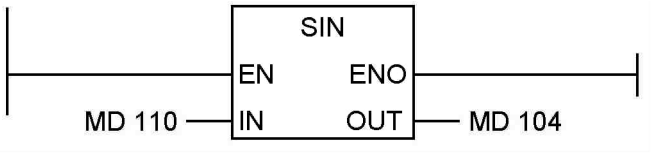


Рисунок 2.26 - Приклад обчислення функції синус

2.7 Застосування функцій перетворення типів даних

Функції перетворення конвертують тип даних, які перебувають в акумуляторі 1. Схема функцій перетворення представлена на рис. 2.27.

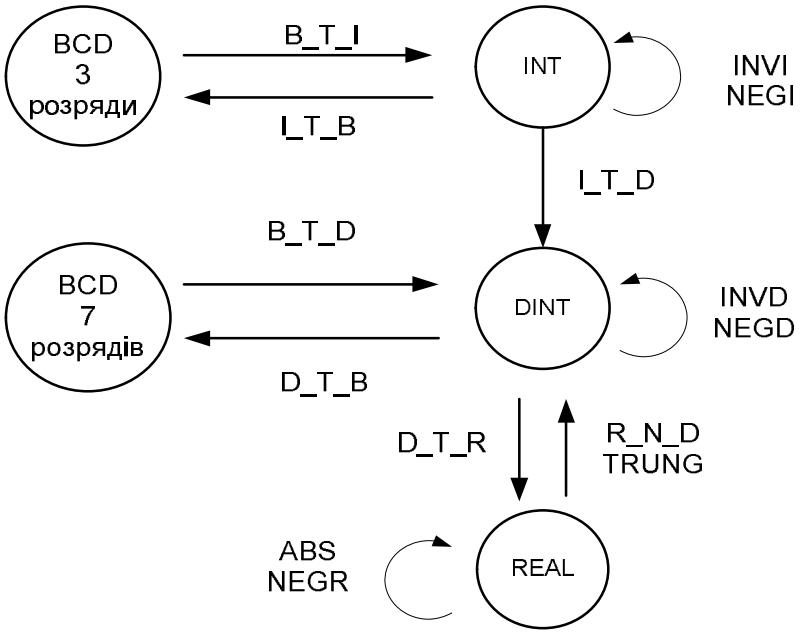


Рисунок 2.27 - Схема використання функцій перетворення

Функції перетворення впливають тільки на дані, які перебувають в акумуляторі 1. При цьому окремі функції перетворення впливають тільки на вміст молодшого слова в акумуляторі 1 (біти з 0 по 15), інші функції

впливають на вміст акумулятора в цілому. Функції перетворення завжди виконуються поза всяким зв'язком з умовами.

Програмування функцій перетворення в STL

Функції перетворення програмуються за наступною схемою:

1) завантаження операнда в акумулятор;
2) запис функції перетворення;
3) передача результату.

Приклад програмування функцій перетворення:

```
L     MW 120;
ITB;                    //Перетворення INT в BCD
T     MW 122;
```

Уміст акумулятора 1 можна використовувати в декілька послідовно виконуваних функціях перетворення.

Приклад:

```
L     BCD_Number;
BTI;                    //Перетворення BCD в INT
ITD;                    // Перетворення INT в DINT
DTR;                    // Перетворення DINT в REAL
T     REAL_Number;
```

У даному прикладі BCD-число перетвориться в число формату REAL.

Існує також кілька спеціальних функцій перетворення:

- INVI знаходження зворотного коду числа формату INT;
- INVD знаходження зворотного коду числа формату DINT;
- NEGI інвертування числа формату INT;
- NEGD інвертування числа формату DINT;
- NEGD інвертування числа формату REAL;
- ABS знаходження абсолютного значення числа формату REAL.

Усі функції перетворення не встановлюють біт стану.

Програмування функцій перетворення в LAD і FBD

На рисунку 2.28 показаний приклад блокової вистави функції перетворення INT в BCD (I_BCD).

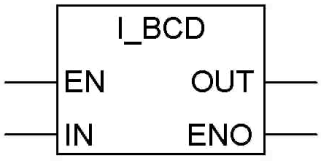


Рисунок 2.28 - Позначення функції перетворення I_BCD

Значення, яке підлягає конвертуванню, подається на вхід IN, результат конвертування формується на виході OUT. Тип даних входу й виходу залежить від функції перетворення. У функції перетворення DI_R (DINT в REAL), наприклад, вхід має тип DINT, а вихід – тип REAL.

Функція перетворення виконується, якщо на вході дозволу присутній сигнал «1». Якщо під час перетворення виникне помилка, то вихід ENO установлюється в «0», а якщо ні, то він установлюється в «1». Якщо виконання функції не дозволене (EN = 0), перетворення не відбувається й на виході ENO також установлюється нуль.

У таблиці 2.7 наведені функції перетворення для чисел типів INT і DINT. Змінні на входах і виходах повинні мати певні типи даних, а операнди, які адресуються абсолютно, відповідати їхнім розмірам.

Таблиця 2.7 – Функції перетворення чисел типів INT і DINT

Перетворення типу даних	Блоковий елемент	Тип даних параметра	
		IN	**OUT**
INT в DINT	I_DI	INT	DINT
INT в BCD	I_BCD	INT	WORD
DINT в BCD	DI_BCD	DINT	DWORD
DINT в REAL	DI_R	DINT	REAL

Не всі функції перетворення повідомляють про помилку. Помилка виникає тільки тоді, коли перевищується припустимий діапазон чисел (I_BCD, DI_BCD) або визначене недійсне число REAL.

Якщо вхідне значення для перетворення BCD_I або BCD_DI містить псевдотетраду, то виконання програми переривається, і викликається організаційний блок обробки помилок OB 121 (синхронні помилки роботи програми).

На рисунку 2.29 показаний приклад перетворення числа INT. Значення слова MW 120 інтерпретується як число типу INT і зберігається як BCD-число в пам'яті меркерів MW 122.

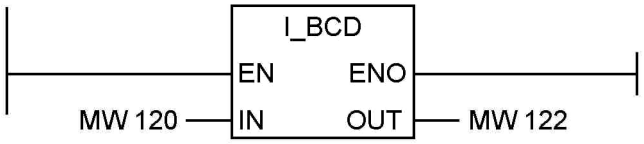


Рисунок 2.29 - Приклад перетворення числа INT

Є кілька функцій для перетворення числа у форматі REAL у число формату DINT (перетворення дробового числа в ціле число). Ці функції наведено в таблиці 2.8. Вони відрізняються по способу округлення.

Таблиця 2.8 - Перетворення чисел типу REAL у числа типу DINT

Перетворення типу даних з округленням	Блоковий елемент	Тип даних параметра	
		IN	OUT
До наступного більшого цілого числа	CEIL	REAL	DINT
До наступного меншого цілого числа	FLOOR	REAL	DINT
До наступного цілого числа	ROUND	REAL	DINT
Без округлення	TRUNC	REAL	DINT

До входів і виходів повинні бути застосовані змінні певного типу даних розміром у подвійне слово.

Функції CEIL, FLOOR, ROUND і TRUNC інтерпретують значення на вході IN як число у форматі REAL і перетворюють його в число формату DINT на виході OUT.

CEIL повертає ціле, яке більше або дорівнює конвертованому числу. Якщо значення на вході IN перебуває поза діапазоном, припустимим для чисел формату DINT, або воно не відповідає числу у форматі REAL, то CEIL установлює біти стану OV і OS. Перетворення при цьому не виконується.

Функція FLOOR повертає ціле, яке менше або дорівнює конвертованому числу. Якщо значення на вході IN перебуває поза діапазоном, припустимим для чисел у форматі DINT, або воно не відповідає числу у форматі REAL, то FLOOR установлює біти стану OV і OS. Перетворення при цьому не виконується.

Функція ROUND повертає наступне ціле число. Якщо результат перебуває точно між непарним і парним числом, перевага віддається парному числу. Якщо значення на вході IN перебуває поза діапазоном, припустимим для чисел у форматі DINT, або воно не відповідає числу у форматі REAL, то ROUND установлює біти стану OV і OS. Перетворення при цьому не виконується.

Функція TRUNC повертає цілу складову перетвореного числа, а дробова складова відтинається.

2.8 Програмування функцій зрушення

Функції зрушення дозволяють побітно зрушувати вліво або вправо дані, які перебувають в акумуляторі 1 (слово або подвійне слово). Біти, які зрушуються за межі слова (подвійного слова), губляться при операціях простого зрушення або переносяться на іншу сторону. Функції зрушення не впливають на вміст інших акумуляторів і на результат логічної операції.

Програмування функцій зрушення в STL

Доступні користувачеві функції зрушення наведено в таблиці 2.9. Їхнє позначення залежить від способу завдання кількості розрядів, на яку здійснюється зрушення. Число позицій для функції зрушення може бути задано двома шляхами – в акумуляторі 2 або в параметрі інструкції.

Таблиця 2.9 - Огляд функцій зрушення

Функція зрушення	W (слово)		DW (подвійне слово)	
	Число позиц. як параметр	Число поз. в Accum2	Число позиц. як параметр	Число поз. в Accum2
Зрушення вліво	Slwn	SLW	Sldn	SLD
Зрушення вправо	Srwn	SRW	Srdn	SRD
Зрушення зі знаком	Ssin	SSI	Ssdn	SSD
Циклічне зрушення вліво			Rldn	RLD
Циклічне зрушення вправо			Rrdn	RRD
Циклічне зрушення вліво через біт CC1			RLDA	
Циклічне зрушення вправо через біт CC1			RRDA	

Так, наприклад, зрушення слова даних вліво здійснюється інструкціями:

SLW n // Зрушення слова даних вліво на *n* розрядів

SLW // Зрушення слова даних вліво на кількість розрядів, зазначених в акумуляторі 2

Функція зрушення SLW дозволяє біт за бітом зрушувати вліво дані, які перебувають у молодшому слові акумулятора 1 (з 0 по 15). При цьому розряди, які звільняються при зрушенні зазначених біт, заповнюються

нулями. Біти, які перебувають у старшому слові акумулятора залишаються без зміни. Перенесення даних у біт 16 не виконується.

Якщо вміст акумулятора 1 (молодшого слова) інтерпретується як ціле типу INT, то зрушення **вліво** еквівалентне множенню на 2.

Для зрушення подвійного слова даних *вліво* слід застосовувати інструкції:

```
SLD     n;          //зрушення подвійного слова вліво на n розрядів
SLD;                //зрушення подвійного слова даних вліво на кількість розрядів, зазначених в акумуляторі 2
```

При зрушенні слова даних *вправо* застосовуються інструкції:

```
SRW     n;
SRW;
```

Якщо вміст акумулятора 1 (молодшого слова) інтерпретується як ціле число формату INT, то зрушення **вправо** еквівалентне розподілу на 2.

Функції зрушення встановлюють біт стану CC0 в "0", а біт CC1 у стан, у якому перебував останній переміщений біт.

Біти стану перевіряються за допомогою двійкового опитування або в операціях переходу.

Функція *циклічного* зрушення з бітом стану CC1 зміщає вміст акумулятора 1 на одну розрядну позицію.

Приклади програмування функцій зрушення:

```
L       MW 130;
SLW  4;             //зрушення слова на 4 розряду вліво
T       MW 132;
…       …
L       #actval;
SSI  2;             //зрушення слова даних на 2 розряду вправо зі знаком
T       #display;
```

При зрушенні числа INT зі знаком *інструкцією SSI* вивільнювані розряди заповнюються значенням знакового біта 15, тобто значенням "0", якщо число позитивне й значенням "1", якщо число негативне.

При зрушенні подвійного слова даних (числа типу DINT) зі знаком застосовується інструкція SSD. Розряди, які звільняються при зрушенні, заповнюються значенням біта 31, який містить знак числа формату DINT.

Операції циклічного зрушення RLD і RRD дозволяють біт за бітом зрушувати вліво (RLD) або вправо (RRD) дані, які перебувають у всіх розрядах акумулятора 1. Звільнений при зрушенні розряд заповнюється значенням біта, який був "виштовхнутим" з акумулятора останнім.

Циклічне зрушення може також виконуватися через біт стану CC1.

Програмування функцій зрушення на мовах LAD і FBD

У мовах LAD і FBD операції зрушення представляються блоковими елементами. На рисунку 2.30, як приклад, показаний елемент для зрушення слова вліво, а в таблиці 2.10 – огляд функцій зрушення.

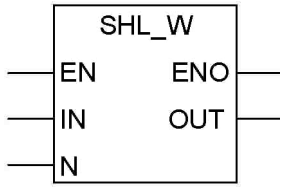


Рисунок 2.30 - Позначення блокового елемента функції зрушення

Таблиця 2.10 - Огляд функцій зрушення

Функція зрушення	Змінна – слово	Змінна – подвійне слово
Зрушення вліво	SHL_W	SHL_DW
Зрушення вправо	SHR_W	SHR_DW
Зрушення зі знаком	SHR_I	SHR_DI
Циклічне зрушення вліво Циклічне зрушення вправо		ROL_DW ROR_DW

Крім входу дозволу EN і виходу ENO блоковий елемент функції зрушення має вхід IN, вхід N і вихід OUT. Заголовок у блоковому елементі ідентифікує виконувану функцію зрушення.

Значення, які підлягають зрушенню, подаються на вхід IN. Кількість розрядів для зрушення задається на вході N. Результат видається на вихід OUT. Типи даних на вході й виході залежать від функції зрушення. Так, наприклад, вхід і вихід для функції зрушення SHR_DW (зрушення вправо змінної розміром у подвійне слово) повинні належати до типу DWORD.

Приклад застосування функцій зрушення

Зрушення змінних розміром у слово показано на рисунку 2.31. Значення в слові пам'яті MW 130 зрушується на 4 позиції вліво й зберігається в слові пам'яті MW 132.

Число зрушень на вході N може бути константою або змінною. Якщо число зрушень рівно 0, то функція не виконується, якщо більше 15, то вихідна змінна після виконання функції містить нуль.

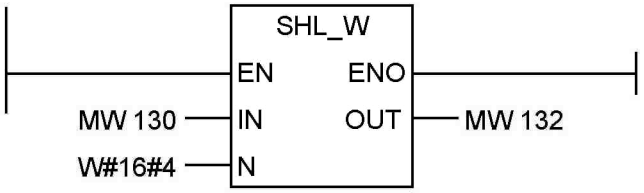


Рисунок 2.31 - Приклад програмування зрушення числа INT

При зрушенні подвійного слова, наприклад функцією SHR_DI, уміст змінному типу DINT на вході IN зрушується побітно вправо на кількість позицій, яка задана на вході N. Бітові розряди, звільнені при зрушенні, заповнюються значеннями біта 31, який є знаком числа DINT. Зрушення вправо числа у форматі DINT еквівалентно розподілу на 2^n, де показник ступеня *n* є числом зрушень.

Циклічні зрушення змінної *вліво* здійснюються за допомогою функції ROL_DW. Функція зрушує вліво вміст змінної типу DWORD на вході IN побітно на кількість позицій, задану числом на вході N. Бітові позиції, які вивільняються при зрушенні, заповнюються витиснутими бітовими розрядами. Результат утримується в змінній типу DWORD на виході OUT.

Число зрушень на вході N може бути константою або змінною. Якщо число зрушень рівно 0, то функція не виконується. Якщо воно рівно 32, то вміст вхідної змінної зберігається й установлюються біти стану. Якщо число зрушень рівно 33, здійснюється зрушення одного розряду, якщо рівно 34, зрушуються два розряди і так далі, тобто зрушення виконується по модулю 32.

Циклічне зрушення змінної *вправо* здійснюється за допомогою функції ROR_DW.

2.9 Контроль стану операції й програмування переходу в програмі

Біти стану – це двійкові прапори (індикаторні біти), які втримуються в слові стану (status word) і використовуються для керування операціями. У таблиці 2.11 представлені біти стану, доступні при програмуванні.

Перша колонка показує номер біта в слові стану. Біти з 0 по 3 і 8 – це "двійкові прапори". Вони застосовуються для керування двійковими функціями. Біти з 4 по 7 – це "числові прапори". Вони використовуються для індикації результатів арифметичних і математичних функцій.

Таблиця 2.11 – Формат слова стану (status word)

Біт	**Двійкові прапорці (binary flags)**	
0	/FC	Первинне опитування (first check)
1	RLO	Результат логічної операції
2	STA	Стан (status)
3	OR	Біт стану OR (OR status bit)
8	BR	Двійковий результат
	Числові прапорці (digital flags)	
4	OS	Для збереження інформації про переповнення
5	OV	Переповнення (Overflow)
6	CC0	Умовний код
7	CC1	Умовний код

Біт 0 - первинне опитування FC (first check)

Біт стану /FC управляє двійковими логічними операціями усередині арифметичного логічного пристрою керування. Двійковий логічний крок завжди починається із двійкової інструкції перевірки (первинного опитування) при /FC = "0". Первинне опитування встановлює /FC у стан "1". Двійковий логічний крок закінчується або присвоєнням двійкового значення, або умовним переходом. При цьому біт стану /FC скидається. Наступна двійкова перевірка (опитування) починається з нової двійкової логічної функції.

Біт 1 - результат логічної операції (RLO)

Біт стану RLO є проміжним буфером у двійкових логічних операціях. При первинному опитуванні процесор передає результат опитування в RLO, комбінує цей результат зі збереженим в RLO значенням і потім зберігає нове значення в RLO. Значення біта RLO можна встановлювати, знімати, інвертувати або зберігати в біті двійкового результату BR (біт 8).

Біт 2 - стан STA (status)

Значення біта STA відповідає стану сигналу зазначеного двійкового розряду (двійкової адреси) або стану "умовного біта", який перевіряється при аналізі двійкової логічної операції. Біт стану STA не впливає на обробку STL-операторів. Його можна використовувати для трасування двійкових логічних послідовностей або для налагодження програми.

Біт 3 – стан OR (OR status bit)

Біт OR status зберігає результат виконаної двійкової логічної операції AND (І) і показує наступній операції OR (АБО), що результат уже

зафіксований.

Біти переповнення 4 і 5 - OV (Overflow) і OS (Overflow stored)

Біт стану OV відображає факт перевищення діапазону припустимих чисельних значень (переповнення) або факт використання некоректних дійсних чисел (REAL). На стан біта OV впливають арифметичні й математичні функції, функції порівняння дійсних чисел типу REAL, а також окремі функції перетворення. Перевірити стан біта OV можна за допомогою операцій перевірки або оператором переходу JO.

Біт стану OS дублює й зберігає встановлений стан біта OV. При цьому, якщо біт стану OV надалі може бути скинутий при виконанні відповідної операції, то біт OS збереже інформацію про факт перевищення.

Біти стану CC0, CC1 (6 і 7)

Біти стану CC0, CC1 – це умовні біти, які відображають результати виконання функцій порівняння, арифметичних і математичних функцій, логічних операцій для слів даних, а також стан біта в операціях зрушення.

Біт 8 – BR (двійковий результат) допомагає реалізувати механізм EN/ENO для викликів блоків у з'єднанні із графічними мовами.

Усі біти стану, тобто слово стану STW можна завантажити в акумулятор 1 інструкцією:

```
L     STW;
```

Програмування переходів в STL-програмі

За допомогою функцій переходу (jump functions) можна перервати лінійне виконання програми й продовжити її виконання в іншій точці. Таке розгалуження програми може бути організоване або з перевіркою виконання певної умови, або без перевірки яких-небудь умов. Відповідно, у програмі повинен використовуватися або умовний, або безумовний перехід.

В STL використовуються наступні функції переходів:

JU	*мітка*	Безумовний перехід;
JC	*мітка*	Перехід, якщо RLO = "1";
JCN	*мітка*	Перехід, якщо RLO = "0";
JCB	*мітка*	Перехід, якщо RLO = "1", і збереження RLO;
JNB	*мітка*	Перехід, якщо RLO = "0", і збереження RLO;
JBI	*мітка*	Перехід, якщо BR = "1";
JNBI	*мітка*	Перехід, якщо BR = "0";
JZ	*мітка*	Перехід, якщо результат рівняється "0";
JN	*мітка*	Перехід, якщо результат не рівняється "0";
JP	*мітка*	Перехід, якщо результат більше "0";
JPZ	*мітка*	Перехід, якщо результат більше або рівняється "0";

JM *мітка* Перехід, якщо результат менше "0";

JMZ *мітка* Перехід, якщо результат менше або рівняється "0";

JUO *мітка* Перехід, якщо результат невірний;

JO *мітка* Перехід при переповненні (перевірка біта OV);

JOS *мітка* Перехід при переповненні (перевірка біта OS);

LOOP *мітка* Перехід циклічний.

Інструкція для кожної функції переходу містить оператор переходу й мітку переходу. Оператор переходу визначає умову, яка перевіряється, а мітка показує, у якій точці програми слід продовжувати обробку програми у випадку, коли умова для переходу виконується.

Мітка переходу може містити до 4 символів алфавіту й числового ряду, а також символ підкреслення. Мітка переходу не може починатися із цифри. Після мітки переходу ставиться двокрапка. Мітка переходу вказує на вираження, яке повинне оброблятися після виконання операції переходу.

Приклад програмування з розгалуженням:

```
L     C 1;        //Завантажити значення лічильника
L     50;         //Завантажити умовне значення
>I;               //Програмування умови
JC    M1;         //Умовний перехід
…     …           //Продовження програми при невиконанні умови
JU    M2;         //Безумовний перехід
M1: …     …       //Мітка продовження програми при виконанні умови
M2: …     …       //Мітка основної програми
```

У наведеному прикладі умовою переходу є позитивний результат операції порівняння RLO. При RLO = "1" виконується перехід до мітки продовження програми M1. При негативному результаті операції порівняння (RLO = "0") перехід до M1 не виконується.

Перехід може виконуватися в будь-якому напрямку програми, однак тільки усередині блоку. Мітка переходу повинна мати унікальний ідентифікатор.

При програмуванні мовою STL ідентифікатори міток зберігаються у відповідній частині блоку на носіях даних програматора PG. Із цієї причини зміни в програмі блоку, зроблені в CPU в інтерактивному режимі, повинні бути також виконані й у програматорі.

Особливості керування переходами в мовах LAD і FBD

Для читання біт стану в LAD передбачені як нормально розімкнуті контакти, так і нормально замкнені. Ці контакти перебувають у розділі Status bits браузера програмних елементів вікна редагування. Для читання біт стану

в FBD передбачені відповідні блокові елементи.

Функція переходу складається з операції переходу у формі котушки (в LAD) або блокового елемента (в FBD), а також мітки переходу, яка позначає місце в програмі для продовження обробки. Мітка переходу вказується над операцією переходу (рис. 2.32).

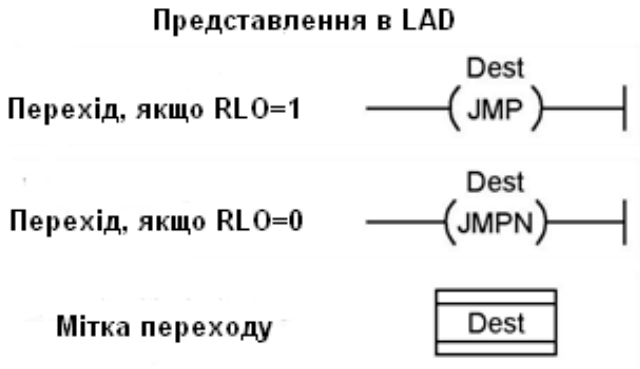


Рисунок 2.32 - Позначення функцій переходу (LAD)

Мітка переходу може містити в собі до 4 символів, які можуть бути буквами, цифрами й знаками підкреслення. Починається вона з букви. Мітка переходу *в блоковому елементі* позначає ланцюг, який буде оброблятися після закінчення операції переходу. Блоковий елемент із міткою переходу повинен розташовуватися на початку ланцюга. Вибрати цей елемент можна в каталозі програмних елементів у розділі LABEL (*Мітка*).

Переходи можна виконувати як уперед (у напрямку обробки програми), так і назад. Перехід може здійснюватися тільки в межах блоку, тобто місце призначення переходу повинне бути в тому ж блоці, що й функція переходу. Якщо використовується головне реле керування MCR, мітка переходу повинна перебувати в тій ж MCR-зоні або в MCR-області, що й функція переходу.

Місце переходу повинне мати унікальне позначення. Програмний редактор зберігає імена міток переходів у розділах відповідних блоків.

Абсолютним переходом, який завжди виконується, є функція переходу JMP. На рисунку 2.34 показаний приклад її застосування у програмі LAD.

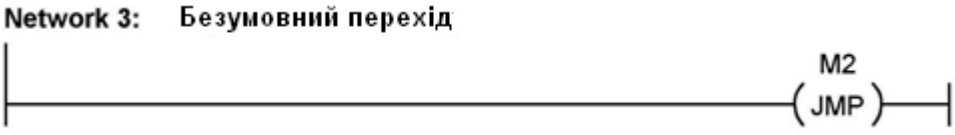


Рисунок 2.34 - Правило застосування безумовного переходу в LAD

Якщо котушка не має прямого з'єднання з лівою шиною, то виконується умовний перехід.

При RLO = 1 CPU перериває лінійний потік програми й продовжує обробку в ланцюзі, позначеною міткою переходу. Якщо попередня логічна операція не виконана, то CPU продовжує виконувати програму в наступному ланцюзі.

Умовний перехід при RLO = 0 виконується функцією JMPN, чия котушка не з'єднана прямо з лівою живильною шиною. Якщо RLO = 0 у випадку невиконання попередньої логічної операції, CPU перериває лінійний потік програми й продовжує обробку в тому ланцюзі, який позначений міткою переходу.

2.10 Застосування інструкцій для виклику й завершення блоків

Інструкції для виклику й завершення обробки блоків належать до функцій обробки кодових блоків.

Кодові блоки із своїми блоками даних викликаються оператором CALL. Екземплярний блок даних призначений для того, щоб зберегти поточні значення змінних кодового блоку до наступного циклу виконання програми. Якщо кодові блоки не мають параметрів, то їх можна викликати за допомогою операторів UC або CC. Обробка блоку припиняється оператором кінця блоку. Оператор BEC завершує обробку програми в блоці залежно від стану RLO, а оператори BEU і BE закінчують блок незалежно від умов.

Функції для кодових блоків наведені в таблиці 2.12.

Програма блоку обробляється, поки не зустрінеться оператор закінчення блоку. По закінченню викликаного блоку CPU вертається до виконання програми в блоці, який зробив виклик. Виконання цієї програми триває від оператора, який іде за оператором виклику блоку. Якщо виконання програми організаційного блоку завершується, CPU передає керування операційній системі.

Інформація про те, яка потрібно CPU для повернення в блок, зберігається в B-стеці. При кожному виклику блоку в B-стеці генерується новий елемент, який містить адресу повернення, уміст регістру даних і адреси локальних даних блоку, який зробив виклик.

Таблиця 2.12

Виклик функціонального блоку		
Із блоком даних і параметрами	Як локальний екземпляр з параметрами	Виклик безумовний і за умовою
CALL FB1, DB1 (In1: =Num1; In2: =Num2; In3: =Num3);	CALL name (In1: =Num1; In2: =Num2; In3: =Num3);	UC FB 5 CC FB 5
Виклик функції		
Зі значенням функції й з параметрами блоку	Без значення функції, з параметрами блоку	Виклик безумовний і за умовою
CALL FB1 (In1: =Num1; In2: =Num2; Ret_Val: = Num3);	CALL FB1 (In1: =Num1; In2: =Num2; Out: = Num3);	UC FB 5 CC FB 5
Оператори завершення блоку		
Умовне завершення блоку	Безумовне завершення блоку	Кінець блоку
BEC	BEU	BE

В інструкції виклику може бути список параметрів блоку. При введенні вихідного тексту програми список параметрів блоку перебуває між круглими дужками. Параметри блоку, наведені в списку, повинні бути розділені комами. При виклику функціонального блоку ініціалізувати усі параметри викликуваного блоку немає необхідності.

Функції FC викликаються шляхом завдання ідентифікатора функції абсолютним або символьним образом після оператора CALL. При виклику функції потрібно ініціалізувати усі параметри. Викликувані функції з функціональним значенням мають точно таку ж форму, як і функції без функціонального значення. Єдиний вихідний параметр, який відповідає функціональному значенню, має ім'я RET_VAL.

Системні функціональні блоки можна викликати в такий же спосіб, як і блоки, створені користувачем. Системні блоки доступні тільки в операційній системі CPU. При виклику системних блоків під час програмування в автономному (offline) режимі, редактору потрібен опис інтерфейсу виклику для того, щоб ініціалізувати параметри.

Опис інтерфейсу розташований у стандартній бібліотеці *Standard library* у системних функціональних блоках *System Function Blocks*. Звідси редактор копіює опис інтерфейсу в папку (розділ) автономного режиму "Blocks", коли викликається системний блок. Після цього скопійований опис інтерфейсу виклику з'являється як "нормальний" об'єкт блоку.

2.11 Методика створення програми на мовах LAD, STL, FBD

Для складання програми мовою STL треба спочатку визначитися – де буде розміщена ця програма. Якщо це буде головна програма із циклічним виконанням, то її потрібно записати в організаційному блоці OB1. Якщо програма розробляється для локального завдання логічного керування, то для її розміщення можливо потрібний функціональний блок.

У кожному разі спочатку треба створити проект.

Створення проекту відбувається в середовищі Simatic Manager. Після завдання імені проекту, вибору типу станції й типу процесорного модуля з'являється можливість розміщення програмних блоків.

Для цього в дереві проекту знайдемо й відкриємо папку Blocks. На правому полі вікна проекту в Simatic Manager перебуває перший блок – блок організації циклічного виконання програми OB1.

Нехай, наприклад, треба створити програму для керування пішохідним переходом і розмістити її в окремому функціональному блоці.

Клацнемо на полі правою кнопкою миші й виберемо в контекстному меню команди «Insert New Object» → «Function Block», як показано на рисунку 2.35. При цьому відкривається вікно для завдання параметрів блоку. На вкладці General – Part 1 у відповідних полях вносимо: номер блоку (нумерація блоків виконується автоматично, але її можна змінити), його символічне ім'я (Cross) і мову програмування (STL). Вікно Properties після виконання цих операцій показано на рисунку 2.36.

Після створення функціонального блоку треба визначити склад змінних, які будуть використовуватися в програмі. Насамперед, необхідно визначити глобальні змінні, які потрібно внести в таблицю Symbol Table. Щоб відкрити цю таблицю, треба у вікні редактора вибрати меню Options → Symbol Table. У таблицю символів можна включати вхідні й вихідні дані, меркери, периферійні дані, таймери, лічильники, функціональні блоки й функції, організаційні блоки, блоки даних і типи даних UDT. Редагувати Symbol Table можна в будь-який момент.

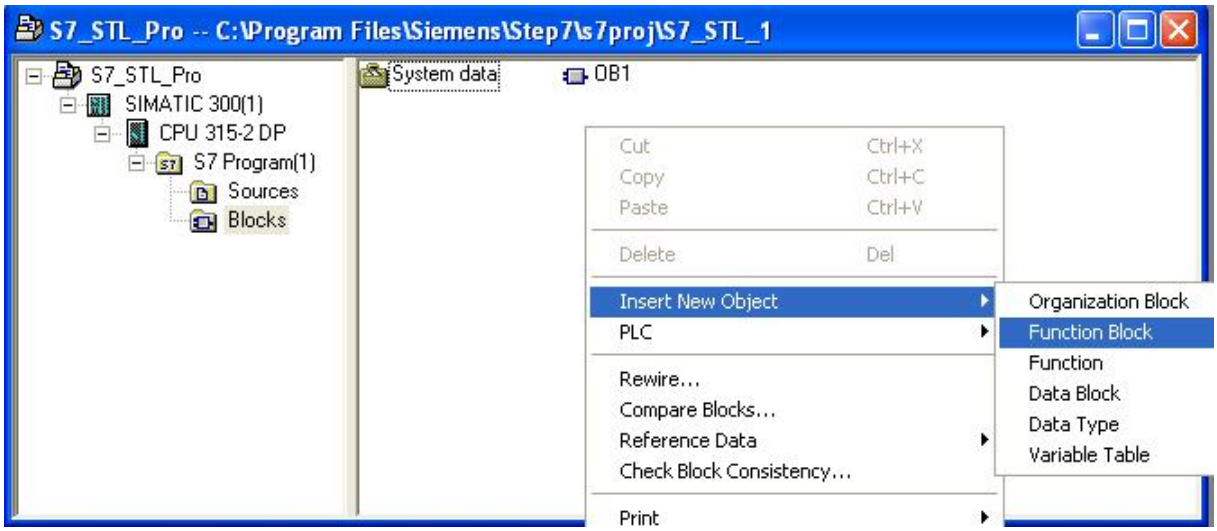


Рисунок 2.35 – Вибір команд у вікні Simatic Manager для створення функціонального блоку

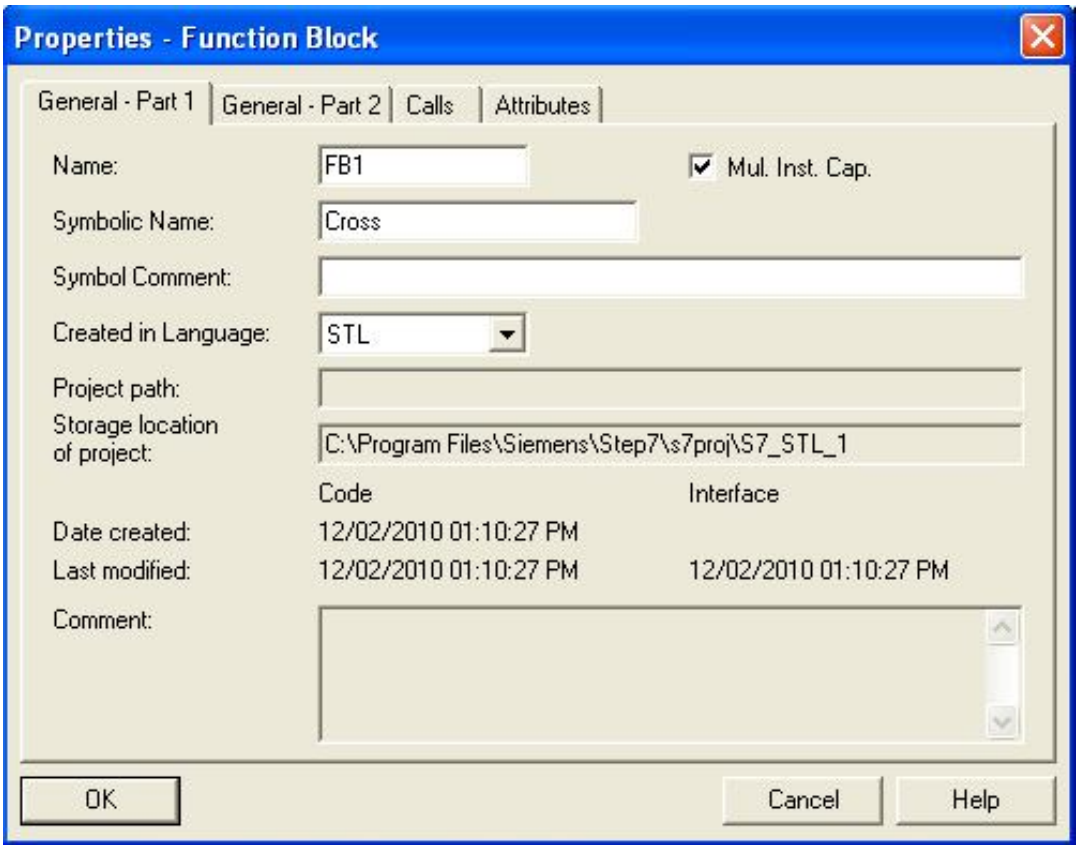


Рисунок 2.36 – Вид вікна Properties після завдання параметрів блоку

Далі визначаються локальні змінні блоку – його інтерфейс із програмою. До інтерфейсу належать вхідні дані (IN), вихідні дані (OUT), вхідні й вихідні параметри (IN_OUT), а також статичні змінні (STAT). Крім цього блок може мати тимчасові дані (TEMP), які не належать до інтерфейсу.

Оголошення символічних імен і типів усіх необхідних змінних здійснюється в розділі оголошень вікна редактора STL- програми, яке можна відкрити подвійним клацанням лівої кнопки на піктограмі створеного функціонального блоку у вікні Simatic Manager.

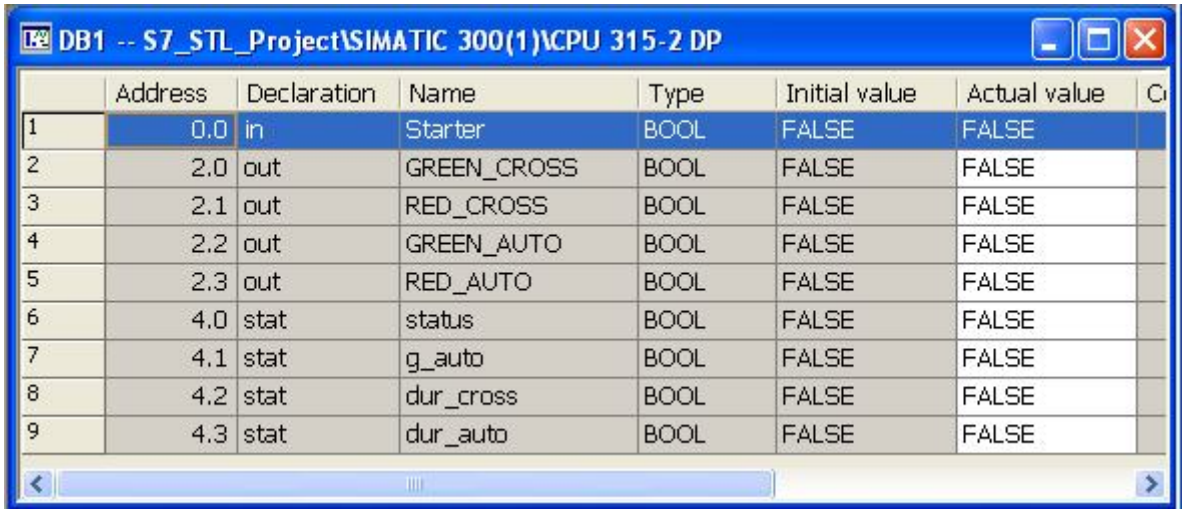
Після оголошення змінних у функціональному блоці потрібно створити блок даних DB. Слід прийняти до уваги, що у випадку внесення змін у розділ оголошень функціонального блоку після створення блоку даних, інтерфейс блоку даних уже не буде відповідати інтерфейсу функціонального блоку, тому блок даних повинен бути створений заново.

Для створення блоку даних потрібно у вікні проекту Simatic Manager розкрити папку Blocks, клацнути правою кнопкою миші на полі розташування блоків, а потім в списку, який відкрився, вибрати команди «Insert New Object» → «Data Block».

У вікні параметрів, яке відкривається при цьому, треба призначити номер і символічне ім'я блоку даних, перемкнути тип блоку з «Shared DB» на «Instance DB» (екземплярний) і вказати номер функціонального блоку. Після внесення цих даних потрібно закрити вікно кнопкою ОК. При цьому інтерфейс функціонального блоку буде скопійований у блок даних. На рисунку 2.37 для прикладу показаний уміст блоку даних DB1, створеного для функціонального блоку FB1.

Після створення функціонального блоку й екземплярного блоку даних залишається організувати виклик цих блоків у головній програмі керування, яка перебуває в OB1.

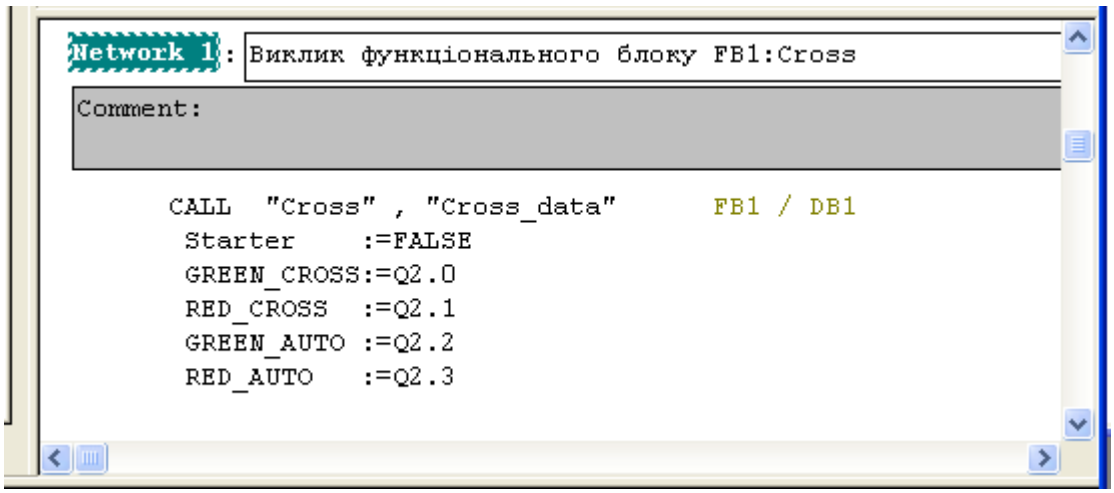
Відкривши OB1, у першому сегменті вводимо команду CALL FB1.DB1 і натискаємо Enter. При цьому редактор автоматично міняє абсолютні номери блоків на символічні імена “Cross” і “Cross_data”, де лапки означають, що це глобальні символи, а також виводить для ініціалізації вхідні й вихідні змінні. Вхідні змінні повинні мати конкретні значення, а для вихідних вказується адреса розташування. Приклад програмного елемента виклику показано на рисунку 2.38.



DB1 -- S7_STL_Project\SIMATIC 300(1)\CPU 315-2 DP

	Address	Declaration	Name	Type	Initial value	Actual value	C
1	0.0	in	Starter	BOOL	FALSE	FALSE	
2	2.0	out	GREEN_CROSS	BOOL	FALSE	FALSE	
3	2.1	out	RED_CROSS	BOOL	FALSE	FALSE	
4	2.2	out	GREEN_AUTO	BOOL	FALSE	FALSE	
5	2.3	out	RED_AUTO	BOOL	FALSE	FALSE	
6	4.0	stat	status	BOOL	FALSE	FALSE	
7	4.1	stat	g_auto	BOOL	FALSE	FALSE	
8	4.2	stat	dur_cross	BOOL	FALSE	FALSE	
9	4.3	stat	dur_auto	BOOL	FALSE	FALSE	

Рисунок 2.37 – Відображення даних у блоці даних DB1



Network 1: Виклик функціонального блоку FB1:Cross

Comment:

```
CALL  "Cross" , "Cross_data"      FB1 / DB1
 Starter     :=FALSE
 GREEN_CROSS:=Q2.0
 RED_CROSS   :=Q2.1
 GREEN_AUTO  :=Q2.2
 RED_AUTO    :=Q2.3
```

Рисунок 2.38- Організація виклику функціонального блоку в програмі

Контрольні питання

1. Якими операціями здійснюється перевірка стану біт?
2. Що включає логічний крок двійкової операції?
3. Яким чином ураховуються типи контактів при програмування перевірки стану?
4. Як програмуються вкладені операції перевірки біт?
5. Які функції належать до операцій з пам'яттю?
6. Як перевіряється наявність фронту сигналу?
7. Яким чином можна зберегти факт виявлення фронту сигналу?
8. Якими операціями здійснюється пересилання даних?
9. Які типи таймерів доступні користувачеві?
10. Як задається тривалість для таймера?

11. Яке значення має тимчасова база для завдання тривалості?
12. Від чого залежить швидкість роботи лічильників?
13. Які операції можна застосовувати при програмуванні лічильника?
14. У якому порядку програмуються операції порівняння?
15. У якому порядку програмуються арифметичні операції?
16. У якому порядку програмуються математичні операції?
17. У якому порядку програмуються функції перетворення?
18. Як програмуються функції зрушення?
19. З яких біт складається слово стану?
20. Які біти дозволяють використовувати функції переходів?
21. При яких умовах виконується перехід в іншу точку програми?
22. Які функції застосовуються для обробки кодових блоків?
23. З яких процедур складається обробка виклику блоку?
24. Якими процедурами завершується обробка блоку?
25. Що необхідно для виклику функції?
26. Що необхідно для виклику системного блоку?

3 ПРОГРАМУВАННЯ МОВОЮ SCL

3.1 Призначення адрес і типів даних у мові SCL

Структурована мова програмування SCL (Structured Control Language) є мовою високого рівня для SIMATIC S7. Мова базується на стандарті 61131-3 і містить у собі елементи мови Паскаль разом з типовими для PLC елементами, такими, як "вхід" і "вихід".

SCL доцільно застосовувати для програмування складних алгоритмів або для завдань, які ставляться до області керування даними.

Створення програми здійснюється в редакторі SCL. Вихідна програма компілюється за допомогою SCL-компілятора в програму користувача.

Незважаючи на те, що елементи мови SCL (оператори, вираження, присвоєння значень) відрізняються в синтаксисі від інструкцій STL, в SCL використовуються ті ж типи даних, адресні області, символьні імена й блокова структура.

Для того щоб створити програму на SCL, потрібно спочатку створити проект. Далі в проекті призначити CPU, тому що створювана програма залежна від типу CPU. Тільки після цього створюються необхідні блоки програми.

Створена SCL-програма може містити, як мінімум, один блок. Можна також створити декілька заготовок SCL-програм, які потім потрібно скомпілювати в певному порядку з використанням керуючого файлу компілятора.

При програмуванні мовою SCL застосовуються наступні адреси й змінні:

- входи I і виходи Q;
- периферійні входи PI і периферійні виходи PQ;
- адреси глобальних даних D;
- меркери M;
- тимчасові й статичні локальні данини із символьною адресацією;
- організаційні блоки OB, функціональні блоки FB, функції FC, а також блоки даних DB.

Функції таймерів T и функції лічильників З обробляються в SCL-програмах як *стандартні функції*.

При призначенні типу даних визначаються:

- характер і значення даних;

- дозволені діапазони, наприклад, числовий діапазон або довжина рядка символів;
- дозволені операції для обробки даних.

В SCL є можливість групування значень типів даних, які представляють однаковий обіг усередині одного класу. Групування дозволяє одержати наступні класи:

- клас ANY_INT, який містить у собі дані типи INT і DINT;
- клас ANY_NUM, який містить у собі дані типи INT, DINT і REAL;
- клас ANY_BIT, який містить у собі дані типи BOOL, BYTE, WORD і DWORD.

Зазначені класи типів даних дозволяють ясніше описати оператори. Однак для опису змінних при їхнім оголошенні застосовувати зазначені класи не можна.

Особливий клас даних представляють константи. Константи – це фіксовані значення, які при виконанні програми не міняються. Константи використовуються у якості початкових значень змінних або для їхнього об'єднання (комбінування) у програмі з іншими змінними, наприклад, граничними значеннями.

У мові SCL константа не визначає "свій" тип даних, поки вона не буде оброблена в арифметичній операції. Наприклад, константа 1234 може ставитися до типу даних INT або до типу даних REAL, залежно від застосування:

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

У мові SCL можна призначати тип даних для константи зі специфікацією "type-defined", використовуючи відповідний префікс. Можна, наприклад, визначити змінну WORD у розділі оголошень за допомогою десяткового, шістнадцятирічного, восьмеричного або двійкового числа.

Приклади:

```
W1:  WORD  :=W#1234;                    //десяткове
W2:  WORD  :=W#16#04D2;                 // шістнадцятирічне
W3:  WORD  :=W#8#2322;                  //восьмеричне
W4:  WORD  :=W#2#0000_0100_1101_0010;   //двійкове
```

Абсолютна адреса завжди належить до класу типів даних ANY_BIT. Так, наприклад, подвійне слово меркерів MD10 має тип даних DWORD.

Операнд може мати тип даних, відмінний від ANY_BIT тільки у двох випадках: якщо має символьне ім'я і якщо тип даних перетворений.

```
MW14 := SHL(IN := MW12, N := 2);          //Операнд має
                                          //символьне ім'я
real1 := real2 + DWORD_TO_REAL(MD10);     //Тип даних операнда
                                          //перетворений
```

Рядок символів повинен вводитися в одинарних лапках. З даним типом можуть також використовуватися керуючі символи, які не друкуються; вони повинні вводитися у форматі $hh, де hh означає значення ASCII символу:

```
string1:= '$0A$0D';        // новий рядок
```

При абсолютній адресації адреси призначаються згідно з початком адресної області; наприклад, I 0.0 (перший біт вхідного байта 0). Абсолютна адресація в SCL відповідає абсолютній адресації в стандартних мовах програмування за винятком адрес глобальних даних (табл. 3.1).

Таблиця 3.1 - Ідентифікація адрес при абсолютній адресації

Адресна область	Біт	Байт	Слово	Подвійне слово
Входи	Iy.x	Iby	Iwy	Idy
Виходи	Qy.x	Qby	Qwy	Qdy
Периферійні входи		Piby	Piwy	Pidy
Периферійні виходи		Pqby	Pqwy	Pqdy
Меркери	My.x	Mby	Mwy	Mdy
Адреси глобальних даних	Dbz.Dxy.x Dbz.Dy.x	Dbz.Dby	Dbz.Dwy	Dbz.Ddy

Примітка: x – адреса біта, y – адреса байта, z – номер блоку даних.

У мові програмування SCL доступ до адрес глобальних даних можливий тільки шляхом повної адресації. Доступ до блоку даних здійснюється викликом змінної типу BLOCK_DB.

При символьній адресації символьні імена повинні бути призначені абсолютним адресам і змінним. Для глобальних даних імена призначаються в таблиці символів, для локальних даних імена призначаються в розділі оголошення змінних блоку.

Символьна адресація в SCL відповідає символьної адресації в стандартних мовах програмування. Можна також використовувати змішані абсолютно-символьні ідентифікатори, наприклад:

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

Непряме використання глобальних адрес базується на абсолютній адресації. При цьому для вказівки розташування даних у пам'яті у квадратних дужках вказується:

– дві змінні INT, якщо дані представлені бітом;
– одна змінна INT, якщо дані представлені байтом.

Наприклад:

- I [*byteindex.bitindex*]; //Дані представлені бітом
- MB [*byteindex*]; //Дані представлені байтом

Тут *byteindex* і *bitindex* є вираженнями типу INT.

У такий спосіб можна адресувати наступні області:

- периферійні входи PI і периферійні виходи PQ (в обох цих випадках адреса біта не вказується);
- входи I, виходи Q, меркери M;
- адреси глобальних даних D (блок даних і адреса даних);
- тимчасові й статичні локальні дані (тільки символьна адресація);
- функції таймерів T и лічильників C (адреса біта не вказується).

Синтаксис адресації блоку:

DB10.DX [*byteindex.bitindex*]; //Адреса блоку й адреса даних

Motordata.DW [*byteindex*]; //Символьне ім'я блоку, адреса даних

Тут *byteindex* і *bitindex* є вираженнями типу INT.

Застосовуючи функцію перетворення WORD_TO_BLOCK, можна призначити непряму адресу блоку даних. Номер блоку даних DB визначається або як змінна, або як вираження з типом даних WORD:

WORD_TO_BLOCK_DB [*dbindex*].DW0.

Тут *dbindex* – змінна або вираження з типом даних WORD.

Якщо блок даних адресований не прямо, то для доступу до адреси даних не може використовуватися символьне ім'я. Так, наприклад, вхідний параметр Data можна викликати такими образами:

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

Тут *byteindex* і *bitindex* – константи, змінні або вираження типу INT. Індекс може бути змінений у процесі виконання програми.

3.2 Правила використання виражень і операторів

Оператори використовуються у вираженнях для одержання певних значень. Вираження – це формула, у якій існує певний порядок обчислення змінної. Цей порядок зазвичай регулюється за допомогою дужок. Результат, отриманий при виконанні вираження, може бути привласнений змінній або параметру блока, або може бути використаний для перевірки критерію умови в інструкції керування.

Вираження розділяються на арифметичні, логічні й вираження порівняння.

Приклад вираження: a + b;

Тут *a* і *b* – ідентифікатори даних (змінні), "+" – оператор.

Список операторів, які підтримує мова SCL, наведено в таблиці 3.2.

Таблиця 3.2 - Список операторів і їх пріоритетів у мові SCL

Комбінування	Назва	Оператор	Пріоритет
Дужки	<Вираження>	(…)	1
Арифметичні операції	Ступінь Знак (плюс, мінус) Множення, розподіл Додавання, вирахування	** +, – *, /, MOD, DIV +, –	2 3 4 5
Порівняння	Менше, менше або рівно Більше, більше або рівно Рівно, не рівно	<, <= >, >= =, <>	6 6 7
Двійкові	Логічне " НІ" Логічне "І" "Виключаюче АБО" Логічне "АБО"	NOT AND, & XOR OR	3 8 9 10
Присвоєння		:=	11

Оператори, які ставляться до одного класу, виконуються послідовно зліва направо.

Арифметичне вираження може складатися або із чисельних значень, або з їхніх символьних імен. Типи даних, які припустимі в арифметичних вираженнях, наведено в таблиці 3.3.

Приклади використання арифметичних виражень:

```
Power       := Voltage * Current;
```

Volume := 4/3*PI*Radius**3;

Meanvalue := (Motor[1].Power + Motor[2]. Power)/2;

Solution1 := -P/2 + SQRT(SQR (P/2) – Q);

Таблиця 3.3 - Типи даних, припустимі в арифметичних операціях

Операція	Оператор	1 операнд	2 операнд	Результат
Множення	*	ANY_NUM TIME	ANY_NUM ANY_INT	ANY_NUM TIME
Розподіл	/	ANY_NUM	ANY_NUM	ANY_NUM
Розподіл цілих	DIV	ANY_INT	ANY_INT	ANY_INT
Додавання Вирахування	+	ANY_NUM TIME	ANY_NUM TIME	ANY_NUM TIME
Зведення в ступінь	**	ANY_NUM	INT	REAL

Логічне вираження поєднує два операнда або вираження, які ставляться до класу типів даних ANY_BIT, згідно з логікою AND (І), OR (АБО) або XOR (Виключаюче АБО).

Приклади:

Q 4.0 := I 1.0 & I 1.1;

Pulses := (Edge_mem_bits XOR ID 16) AND ID 16;

MW 30 := MW 32 AND Mask;

Логічне вираження видає значення, яке належить до класу типів даних ANY_BIT. Результат логічного вираження належить до типу BOOL, якщо обидва операнда також мають тип BOOL. Якщо один із операндів має тип BYTE, WORD або DWORD, то результат буде мати той тип даних, який більш "вимогливий" до пам'яті операнда.

До логічного вираження належить також інвертування. Ця операція аналогічна зміні знака.

Приклад:

Automatic AND NOT Manual_on;

Вираження порівняння дозволяє одержати результат у вигляді булевого значення – при виконанні умови порівняння результат рівняється TRUE (ІСТИНА), а при її невиконанні результат рівняється FALSE (НЕПРАВДА).

Приклади:

Toollarge:= Voltageact > Voltageset;

Warning:= (Voltage*Current) > 20_000;

IF Deviatin > 2_000 THEN Display := 16#F002; END_IF;

Порівнювані операнди повинні належати до одного типу даних або до одного класу типів даних (ANY_INT, ANY_NUM або ANY_BIT).

За допомогою операції *присвоєння значення* одна змінна одержує значення іншої змінної або значення вираження. Ліворуч від оператора присвоювання ":=" перебуває змінна, яка ухвалює значення змінної, що перебуває справа від оператора присвоювання.

Типи даних, які перебувають із двох сторін від знака присвоювання, повинні бути ідентичними. Виключення представляє тільки випадок явного присвоювання типу даних.

Приклади:

```
Automatic   := TRUE;
Deviation   := Actualvalue – Setpoint;
Display     := INT_TO_WORD (Deviation);
```

Абсолютні адреси мають типи даних ANY_BIT, які обумовлені розміром – BOOL, BYTE, WORD, DWORD. Якщо необхідно призначити інший тип даних, то доцільно використовувати оператори перетворення.

У випадку використання масивів можна:

- звертатися до масиву в цілому;
- звертатися до частини масиву;
- звертатися до елемента масиву.

При перетворенні типів даних у масивах слід ураховувати, що типи даних елементів масивів повинні бути погоджені. При цьому повинні збігатися граничні значення індексів у кожній розмірності.

3.3 Особливості застосування операторів керування програмою

За допомогою операторів керування (control statements) користувач може організувати розгалуження програми, циклічне виконання окремих фрагментів програми й здійснювати перехід у програмі блоку для виконання іншої її частини.

Мова програмування SCL підтримує наступні оператори керування:

- IF – оператор для виконання розгалуження в програмі за умовою, яка перевіряється у відношенні до булевої змінної;
- CASE – оператор для виконання розгалуження в програмі за умовою, яка перевіряється у відношенні цілої змінної або параметра типу INT;
- FOR – оператор для організації в програмі циклів зі змінною, яка є лічильником циклів;

• WHILE – оператор для організації в програмі циклів, які ініціюються при виконанні певної умови;

• REPEAT – оператор для організації в програмі циклів із завершенням за умовою;

• CONTINUE – оператор для завершення поточного проходу циклу в програмі;

• EXIT – оператор для виходу із циклу в програмі;

• GOTO – оператор для переходу в нову точку програми, зазначену міткою;

• RETURN – оператор для виходу із програми блоку.

Оператор IF

Оператор IF управляє виконанням тієї або іншої частини програми залежно від стану булевої змінної.

Перший варіант синтаксису при застосуванні оператора:

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

Тут condition – це адреса або вираження з типом BOOL. Якщо condition має значення TRUE, то виконуються оператори після ключового слова THEN. Якщо condition має значення FALSE, то виконуються оператори після ключового слова END_IF. Ключове слово END_IF завершує оператор IF.

Другий варіант синтаксису:

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

У даному прикладі, як і в попередньому, condition має значення TRUE або FALSE. Якщо condition має значення TRUE, то виконуються оператори після ключового слова THEN. Якщо condition має значення FALSE, то виконуються оператори після ключового слова ELSE.

Третій варіант синтаксису:

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

Можна використовувати будь-яку кількість комбінацій ключових слів ELSEIF ... THEN ... між ключовими словами IF ... THEN ... і ELSE. Ключове слово ELSE і наступні оператори не є обов'язковими.

Приклад:

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

У наведеному прикладі, якщо змінна *Actual_value* більша, ніж змінна *Setpoint*, то виконуються оператори, які йдуть після ключового слова THEN. Якщо, навпаки, змінна *Actual_value* менше, чим змінна *Setpoint*, то виконуються оператори, які йдуть після ключового слова ELSEIF. Якщо обидва порівняння не виконуються, то виконуються оператори після ключового слова ELSE.

Оператор CASE

Оператор CASE дозволяє вибрати для виконання потрібну послідовність операторів у програмі залежно від значення якогось параметра типу INT.

Загальна структура програми з оператором CASE може мати наступну форму:

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

У цьому прикладі Selection – це адреса або вираження типу INT. Якщо Selection має значення CONST1, то виконуються оператори statements1. Якщо Selection має значення CONST2, то виконується ланцюжок операторів

statements2 і так далі. Якщо Selection має значення, яке перебуває за рамками списку значень, зазначених для перевірки, то виконується ланцюжок операторів після ключового слова ELSE.

Список значень CONST1, CONST2 і т.д. складається із цілих (INT) констант. Для цих констант можуть використовуватися кілька варіантів форматів записи. У якості константи-перемикача Constx можуть бути зазначені:

- одиночне ціле число;
- діапазон цілих чисел, наприклад, 15..20;
- суміш розділених комами окремих цілих чисел і діапазонів цілих чисел, наприклад, 21,25,31..33.

При цьому кожне значення константи-перемикача Constx повинне бути унікальним.

Приклад.

Нехай значення, яке привласнюється змінній *Error_number*, залежить від змінної *ID*. Тоді програма може виглядати так:

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

Оператор FOR

Оператор FOR застосовується для організації циклів з лічильником циклів. Виконання внесеного в цикл фрагмента програми буде повторюватися настільки довго, скільки змінна (лічильник циклів) буде залишатися в зазначеному діапазоні.

Загальна структура програми з оператором FOR може мати наступну форму:

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

При обробці даного оператора початкове значення limit1 привласнюється лічильнику циклів "i". Змінна, призначена лічильником циклів, повинна бути типу INT або DINT. Вона повинна мати початкове й

кінцеве значення, а також крок зміни step.

Після кожного проходу програми (циклу) лічильник циклу збільшується на величину кроку збільшення step, якщо крок зазначений як позитивне число, або зменшується на величину кроку step, якщо крок зазначений як негативне число.

При програмуванні циклу рядок BY step не є обов'язковим. Якщо така умова для кроку лічильника циклу відсутня, то крок (за замовчуванням) ухвалюється рівним +1. Якщо величина змінної лічильника циклу виходить за межі зазначеного діапазону, то програма переходить до оператора, який стоїть після ключового слова END_FOR.

Оператор FOR може бути вкладеним, тобто усередині циклу з оператором FOR можна запрограмувати інші цикли з оператором FOR, у яких як лічильники циклу використовуються інші змінні.

Усередині циклу з оператором FOR може бути запрограмований перехід до початку циклу (з використанням оператора керування CONTINUE) або повний вихід із циклу для продовження виконання програми, починаючи відразу ж після ключового слова END_FOR, (з використанням оператора керування EXIT).

Нехай, наприклад, необхідно привласнити значення слів з PIW 128 по PIW 142 області периферії словам в області меркерів – з MW 128 по MW 142.

Програма цієї процедури буде мати такий вид:

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

Оператор WHILE

Оператор WHILE служить для організації циклів, виконання яких триває увесь час, поки виконується певна умова.

Загальна структура програми з оператором WHILE може мати наступну форму:

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

У цьому прикладі Condition – це адреса або вираження типу BOOL. Поки виконується певну умову, тобто, поки Condition = TRUE, будуть циклічно виконуватися вираження, представлені операторами Statements.

Перед кожним проходом виконується перевірка умови Condition. Якщо умова не виконується (Condition = FALSE), програма переходить до оператора, розташованого після ключового слова END_WHILE. Такий перехід

можливий навіть без проходу програми циклу. Оператор Statements при цьому жодного разу не буде виконаним.

Оператор WHILE може бути вкладеним. При цьому усередині одного циклу з оператором WHILE можуть розміщатися інші цикли з оператором WHILE.

Усередині циклу з використанням оператора WHILE може бути запрограмований перехід до початку циклу з використанням оператора керування CONTINUE або повний вихід із циклу з використанням оператора керування EXIT. Після виходу програма буде виконувати оператор, розташований після ключового слова END_WHILE.

Оператор REPEAT

Оператор REPEAT служить для організації циклів, виконання яких триває увесь час, поки не зустрінеться умова завершення обробки циклу.

Загальна структура програми з оператором REPEAT може мати наступну форму:

REPEAT
Statements;
UNTIL Condition
END_REPEAT;

У цьому прикладі Condition – це адреса або вираження типу BOOL. Поки не виконується певна умова, тобто, поки Condition = FALSE, будуть циклічно виконуватися вираження Statements.

Після кожного проходу циклу виконується перевірка умови Condition. Якщо умова виконується (Condition = TRUE), то цикл далі не обробляється й виконання програми буде продовжено після ключового слова END_REPEAT.

Таким чином, програма циклу буде оброблена, принаймні, один раз, навіть якщо при першому проході циклу виконується умова завершення його обробки.

Оператор REPEAT може бути вкладеним. При цьому усередині одного циклу з оператором REPEAT можуть розміщатися інші цикли з оператором REPEAT.

Усередині циклу з використанням оператора REPEAT може бути запрограмований перехід до початку циклу з використанням оператора керування CONTINUE або повний вихід із циклу для продовження виконання програми, починаючи відразу ж після ключового слова END_REPEAT (з використанням оператора керування EXIT).

Нехай, наприклад, необхідно викликати системну функцію SFC 25 COMPRESS стільки разів, поки не буде завершений "стиск" пам'яті.

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

Оператор CONTINUE

Оператор CONTINUE служить для завершення поточного проходу циклу в програмі, організованої за допомогою операторів FOR, WHILE або REPEAT.

Після виконання оператора CONTINUE у програмі проводиться одна з наступних дій:

- якщо цикл організований за допомогою операторів REPEAT або WHILE, перевіряється умова для виконання наступного проходу циклу;
- якщо цикл організований з оператором FOR, проводиться зміна лічильника циклу на величину кроку збільшення й перевіряється умова – чи перебуває змінна лічильника циклів у припустимому діапазоні значень.

Нехай, наприклад, необхідно встановити меркери за допомогою двох вкладених циклів з використанням оператора FOR. Установка біт повинна починатися з меркера M 0.3. Умова завершення циклу формулюється при цьому таким способом: якщо байтова адреса (i) рівняється 0, а бітова адреса (k) менше 2, оператори тіла внутрішнього циклу FOR не виконуються.

Програма:

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

Оператор EXIT

Оператор EXIT служить для повного завершення обробки циклу, організованого за допомогою операторів FOR, WHILE або REPEAT. При цьому вихід із циклу з оператором EXIT не залежить від виконання умов, які перевіряються в циклі, і може виконуватися з будь-якої точки циклу. При виході із циклу з оператором EXIT програма продовжує виконуватися відразу ж після ключових слів END_FOR, END_WHILE або END_REPEAT.

Вихід із циклу з оператором EXIT здійснюється негайно із точки програми циклу, де цей оператор зустрівся.

Приклад:

Нехай, необхідно встановити меркери за допомогою двох вкладених циклів з використанням оператора FOR. Якщо байтова адреса (i) рівняється 2, а бітова адреса (k) більше 5, то виконання внутрішнього циклу FOR переривається, тобто установка біт повинна закінчуватися на меркері M 2.5.

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

У даному прикладі виконання циклу FOR припиняється при певній умові для лічильника циклу *k* за допомогою оператора EXIT.

Оператор RETURN

Оператор RETURN служить для безумовного виходу з поточного блоку й переходу в основну програму. При цьому оператор RETURN пересилає стан змінної OK на вихід ENO блоку, який завершується з ним.

Приклад:

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

3.4 Особливості програмування SCL-блоків

У користувацькій SCL-програмі використовуються кодові блоки й блоки даних. Кодові блоки – це викликувані для виконання підпрограми, а блоки даних – це описи атрибутів змінних, які беруть участь у виконанні цих підпрограм.

У мові програмування SCL використовується стандартна структура й стандартний інтерфейс блоку. Тому окремі блоки, запрограмовані з використанням SCL, можна викликати, наприклад, в FBD-блоці й, навпаки, з SCL-блоків можна викликати блоки, створені з використанням, наприклад, мови STL.

Система STEP 7 підтримує функції FC і функціональні блоки FB як кодові блоки. На відміну від функцій FC функціональні блоки FB

викликаються разом із блоками даних, у яких зберігаються локальні змінні блоку (пам'ять блоку).

Параметри блоку

Параметри блоку забезпечують зв'язок між викликаючим і викликуваним блоками. Ці параметри можуть бути оголошені як вхідні (Input), вихідні (Output) і вхідні-вихідні (In/Out) параметри. Вхідні параметри можуть бути тільки прочитані, вихідні параметри можуть бути тільки записані. Якщо параметри повинні бути спочатку прочитані, потім змінені й, зрештою, знову записані, то слід використовувати тип In/Out.

У функціях FC параметри блоку є покажчиками на фактичні параметри або на інші покажчики.

У функціональних блоках FB параметри блоку зберігаються в екземплярних блоках даних. Доступ до цих даних вимагає визначення блоку даних і адреси даних.

Приклади:

Result := DB1.DW2;

Result := DB2.Total;

Result := Privod_data.Speed;

Формальні параметри

Для адресації блоку використовуються формальні параметри. Вони мають такі ж імена, як і параметри блоку, і використовуються у вираженнях програми в якості операндів.

Формальні параметри параметричних типів TIMER і COUNTER можуть бути оброблені з використанням функцій таймерів і функцій лічильників. Значення формальних параметрів можуть бути передані у викликувані блоки.

За допомогою формальних параметрів типу BLOCK_xx можна одержати доступ до адрес даних у блоці даних. Формальні параметри параметричних типів BLOCK_FB і BLOCK_FC при використанні мови програмування SCL можуть тільки передаватися у викликувані блоки, тому що команд для обробки формальних параметрів такого типу в блоці немає.

Формальні параметри типів даних POINTER і ANY можуть бути передані у викликувані блоки цілком. Виключення представляють тимчасові локальні дані, передача яких не допускається.

Статичні локальні дані

Статичні локальні дані – це пам'ять функціонального блоку. Ці дані розташовуються в *екземплярному блоці даних*. При цьому значення даних зберігаються доти, поки не будуть змінені програмою. У статичних локальних

даних можна розміщати також локальні екземпляри функціональних блоків і системних функціональних блоків.

Статичні локальні дані оголошуються за допомогою ключових слів VAR і END_VAR.

У якості статичних локальних даних допускається застосовувати всі прості, складові, користувацькі (UDT) типи даних, а також типи даних POINTER і ANY.

Якщо значення ініціалізації задавати не потрібно, змінні можна повідомляти списком.

Приклад:

```
VAR
VAR1, VAR2, VAR3: INT;
VAR4: INT:=3;
END_VAR
```

Необхідно врахувати, що при програмуванні мовою SCL статичні локальні дані у функціональному блоці можуть бути адресовані тільки символьним образом.

Попередня ініціалізація параметрів блоків

Попередня ініціалізація параметрів блоків не обов'язкова й допускається тільки у відношенні таких функціональних блоків, параметри яких зберігаються як значення. Це поширюється на будь-які параметри блоків із простими типами даних, а також на вхідні (Input) і вихідні (Output) параметри складних типів даних.

Якщо ініціалізація параметрів блоків не зроблена, то редактор буде використовувати, як початкові значення параметрів, або нульове значення, або найменше значення, або пробіл (залежно від типу даних). Для параметрів типу BLOCK_DB як значення за замовчуванням ухвалюється DB1 (DB0 не існує).

Виклик SCL-блоків

При програмуванні блоків мовою програмуванні SCL розрізняють блоки з функціональним значенням і блоки без функціонального значення.

Функціональні блоки FB і функції FC без функціонального значення є просто "областями" програми, тобто підпрограмами.

Функції FC з функціональним значенням можуть використовуватися у вираженнях присвоювання, а також в інших вираженнях у якості операндів.

Системні функціональні блоки SFB викликаються точно так само, як і функціональні блоки FB, а системні функції SFC викликаються точно так

само, як функції FC. Слід урахувати, що системний функціональний блок SFB викликається із блоком даних, який розміщається в користувацькій програмі.

При виклику блоку його параметри ініціюються *фактичними параметрами*, які являють собою константи, змінні або вираження. Саме із цими параметрами блок обробляється при виконанні програми, і саме в них зберігаються результати цієї обробки.

При виклику функцій FC і системних функцій SFC усі параметри блоку повинні бути ініційовані. При виклику функціональних блоків FB і системних функціональних блоків SFB ініціалізація параметрів блоку не обов'язкова.

Вихідні параметри при виклику функціональних блоків FB і системних функціональних блоків SFB ініціюються за допомогою прямого звертання до екземплярних даних.

Приклад виклику функції FC без функціонального значення:

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

При виклику використовується або абсолютна, або символьна адреса блоку. Список параметрів приводиться в дужках. Дужки повинні бути зазначені, навіть якщо функція FC не має параметрів.

Порядок ініціалізації параметрів може бути довільним. Якщо функція має єдиний вхідний параметр, то ім'я параметра при ініціалізації може бути опущене.

Приклад ініціалізації з перетворенням INT-змінної Speed в STRING-змінну Display:

Display:= I_STRING(Speed);

Приклад виклику функції FC з функціональним значенням:

Result:= FC2 (MAX:= Maximum, IN:= Inputvar, MIN:= Minimum);

У даному прикладі глобальній змінній Result привласнюється функціональне значення функції FC2. Функції FC з функціональним значенням можуть використовуватися в якості операндів у будь-яких вираженнях.

Якщо при виклику блоку використовується вхідний параметр EN, і якщо цей вхід має значення FALSE, то функціональне значення не буде визначено, тобто функціональному значенню не буде привласнена ніяка величина.

Екземплярний блок даних специфікується при виклику функціонального блоку.

Приклад виклику функціонального блоку з екземплярним блоком даних:

```
FB2.DB2 (IN:= Inputvalue);
Result:= DB2.OUT;
```

У записі виклику спочатку записується адреса функціонального блоку, а потім, відділена крапкою, адреса екземплярного блоку даних. Далі в дужках представляється список параметрів.

Оскільки вхідні й вихідні параметри складних типів зберігаються як покажчики, вони повинні бути ініційовані значущими величинами при першому виклику функціонального блоку. Якщо параметр блоку не ініційований, то він зберігає своє останнє певне значення.

Усі параметри блоку даних можуть бути адресовані як глобальні дані із вказівкою імені екземплярного блоку даних і імені параметра. Вони також можуть бути ініційовані при виклику функціонального блоку за допомогою операторів присвоювання:

```
DB2.MAX:= Maximum;
DB2.MIN:= Minimum.
```

Функціональний блок, як локальний екземпляр

Функціональні блоки можуть бути оголошені як *локальні екземпляри* й викликані в іншому функціональному блоці. Такі функціональні блоки зберігають свої локальні дані в екземплярному блоці даних.

Приклад:

```
FUNCTION_BLOCK FB2
...
VAR
Delimiter : FB3;
END_VAR
BEGIN
Delimiter (IN := Inputvar);
Result := Delimiter.OUT;
...
END_FUNCTION_BLOCK
```

Оголошення екземплярів виконується в розділі статичних локальних даних за ключовим словом VAR. Тут локальному екземпляру (FB3) призначається ім'я, наприклад, Delimiter. До моменту компілювання викликуваний функціональний блок повинен уже існувати або у вигляді

раніше скомпільованого блоку в розділі Blocks (*Блоки*), або у вигляді підготовленої програми.

Аналогічні дії виконуються при виклику системного функціонального блоку SFB.

Застосування механізму EN/ENO

При програмуванні на SCL користувач має можливість перевірити окремі вираження (операції) на правильність їх виконання й залежно від цього ухвалювати розв'язок – виконувати наступний блок або не виконувати.

Для реалізації цієї можливості в мові програмування SCL існує змінна з іменем "OK" і типом даних BOOL. Ця змінна повідомляє про помилки, які виникають при виконанні програми в SCL-блоці. На початку блоку OK-змінна має значення TRUE. При виникненні програмної помилки вона скидається в стан FALSE. Результат перевірки повідомляється за допомогою спеціального виходу блоку ENO (Enable output – *вихід розблокований*) типу BOOL.

Перевіряти OK-змінну можна за допомогою відповідних SCL-операцій. Нижче наведений приклад одержання перевірки результату арифметичного вираження:

```
SUM:= SUM + IN;
IF OK
THEN                (* немає помилок *);
ELSE                (* помилка в операції додавання *);
END_IF;
```

Для того, щоб викликуваний блок зберіг значення OK-змінної на виході ENO, необхідно передбачити наступні призначення:

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

Після виклику блоку значення ENO може бути використане для визначення того, чи правильно був оброблений блок. Якщо блок був оброблений правильно, то ENO = TRUE. Якщо при обробці блоку виникла помилка, то ENO = FALSE.

Приклад програмування аналізу ENO:

```
FC5 (In1:= ..., In2:= ...);
IF ENO
THEN                (* немає помилок *);
ELSE                (* виникла помилка *);
END_IF;
```

Для керування викликами блоку використовується вхід EN типу BOOL. Якщо вхід EN ініційований значенням TRUE, то блок буде викликаний для

виконання, якщо EN ініційований значенням FALSE, те блок не буде викликаний для виконання. При цьому буде виконаний перехід до наступного оператора після виклику блоку.

Приклад виклику блоку із вказівкою значення входу EN:

FC1 (EN:= I 1.0, In1:= ..., In2:= ...); (*FC15 виконується тільки у випадку, коли I 1.0 = 1 *)

Якщо EN не використовується, блок буде виконуватися завжди.

Приклад програмування виклику блоку FC2 у випадку, якщо обробка FC1 успішно завершена:

FC1 (EN := I1.0, In1:= ..., In2:= ...);

FC2 (EN := ENO, In1:= ..., In2:= ...);

3.5 Особливості програмування SCL-функцій

До SCL-функцій належать наступні функції: *таймерів, лічильників, математичні, зрушення й перетворення*.

Програмування таймерів

У системній пам'яті CPU підтримується ряд таймерів, які відрізняються режимам роботи:

S_PULSE – режим керованого імпульсу (pulse timer);

S_PEXT – режим розширеного імпульсу (extended pulse);

S_ODTON – із затримкою включення (delay);

S_ODTS – із затримкою включення з пам'яттю (latching ON delay);

S_OFFDT – із затримкою вимикання (OFF delay).

Усі функції таймера мають параметри, показані в таблиці 6.4.

Таблиця 6.4 - Параметри SIMATIC-функцій таймерів

Параметр	Оголошення	Тип даних	Значення
T_NO	INPUT	TIMER	Адреса таймера
S	INPUT	BOOL	Параметр запуску
TV	INPUT	S5TIME	Установлене значення
R	INPUT	BOOL	Скидання таймера
Функціональне значення	OUTPUT	S5TIME	Поточне значення в BCD форматі
Q	OUTPUT	BOOL	Стан таймера
BI	OUTPUT	WORD	Поточне значення таймера у двійковому коді

Приклад виклику функцій таймерів:

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

При ініціалізації параметрів функцій таймерів слід ураховувати, що параметр T_NO повинен бути ініційований завжди.

На додаток до SIMATIC-функцій таймерів у спеціалізованих CPU підтримуються також IEC-функції таймерів. Ця підтримка забезпечується системними функціональними блоками SFB:

SFB 3 TP – функція генерації імпульсу;

SFB 4 TON – функція генерації імпульсу із затримкою включення;

SFB 5 TOF – функція генерації імпульсу із затримкою вимикання.

Ці функціональні блоки зберігаються в бібліотеці *Standard library* у розділі *System Function Blocks (Системні функціональні блоки)*.

Програмування лічильників

У системній пам'яті CPU підтримуються три функції лічильників:

S_CU – функція лічильника прямого рахунку (up counter);

S_CD – функція лічильника зворотного рахунку (down counter);

S_CUD – функція прямого й зворотного рахунку (up-down counter).

Параметри SIMATIC-функцій лічильників представлено в таблиці 3.5.

Таблиця 3.5 - Параметри SIMATIC-функцій лічильників

Параметр	Оголошення	Тип даних	Значення
C_NO	INPUT	COUNTER	Адреса лічильника
CU	INPUT	BOOL	Прямий рахунок
CD	INPUT	BOOL	Зворотний рахунок
S	INPUT	BOOL	Параметр запуску
PV	INPUT	S5TIME	Установлене значення
R	INPUT	BOOL	Скидання лічильника
Функціональне значення	OUTPUT	WORD	Поточне значення в BCD форматі
Q	OUTPUT	BOOL	Стан лічильника
CV	OUTPUT	WORD	Поточне значення

Приклад виклику функцій лічильників:

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

При ініціалізації параметрів функцій лічильників застосовуються наступні правила:

- не допускається застосування параметра CD разом з функцією лічильника S_CU, а також параметра CU разом з функцією лічильника S_CD (повинен бути встановлений лише один з параметрів CD або CU);
- параметр C_NO повинен бути ініційований завжди;
- параметри S і PV, а також параметри Q і CV можна не ініціювати.

Для ініціалізації параметра PV лічильника може бути застосоване ціле число типу INT, як константа.

У спеціалізованих CPU підтримуються також IEC-функції лічильників (як системні функціональні блоки SFB):

SFB 0 CTU – функція лічильника прямого рахунку;

SFB 1 CTD – функція лічильника зворотного рахунку;

SFB 2 CTUD – функція лічильника прямого й зворотного рахунку.

Програмування математичних функцій

У мові програмування SCL підтримуються наступні математичні функції:

Тригонометричні функції:

SIN (синуса), COS (косинуса), TAN (тангенса);

ASIN (арксинуса), ACOS (арккосинуса) ATAN (арктангенса).

Логарифмічні функції:

EXP	експонентна функція по підставі *e*;
EXPD	експонентна функція по підставі 10;
LN	натуральний логарифм;
LOG	десятковий логарифм;

Інші функції:

ABS	функція виділення абсолютного значення;
SQR	функція знаходження квадрата числа;

SQRT функція узяття квадратного кореня.

Математичні функції обробляють числа форматів INT, DINT і REAL. При використанні вхідного параметра у форматі INT або DINT, число автоматично перетвориться у формат REAL. Виключення представляє функція ABS, яка видає результат того ж типу, до якого ставилося вихідне число.

Тригонометричні функції розглядають вхідні параметри, як кути, виражені в радіанах, у діапазоні від 0 до 2π (де π = 3,141593e+00), що відповідає діапазону в градусах від 0° до 360°.

Приклади програмування математичних функцій:

1. Розрахунки реактивної потужності Reactive_power, яка визначається добутком напруги Voltage на струм Current і на синус фазового зрушення між ними:

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

2. Розрахунки обсягу рідини Volume, який визначається добутком числа PI на квадрат радіуса підстави посудини Radius і на рівень заповнення посудини Level:

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

3. Розрахунки довжини гіпотенузи по відомих величинах катетів:

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

Програмування функцій зрушення

Загальна форма виклику функцій зрушення (Shifting) і циклічного зрушення (Rotating) має вигляд:

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

Функції зрушення й циклічного зрушення мають *два вхідних параметри*.

Параметр IN визначає змінну, з якою необхідно виконати операцію зрушення або циклічного зрушення. Цей параметр належить до класу ANY_BIT, тобто, до типів BOOL, BYTE, WORD, DWORD. При цьому значення функції належить до того ж типу, що й вхідне значення.

Параметр N показує число біт, на яке необхідно зробити операцію зрушення або циклічного зрушення. Цей параметр належить до типу INT.

До функцій зрушення належать:

SHL – зрушення вліво;

SHR – зрушення вправо.

До функцій циклічного зрушення належать:

ROL та ROR – циклічне зрушення вліво та вправо, відповідно;

Приклади:

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

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

Використання функцій перетворення

У мові програмування SCL підтримуються *два типи функцій* перетворення – неявні і явні.

Неявні функції виконуються в SCL автоматично (неявно), тому що вони не пов'язані із втратою інформації. Прикладом може служити перетворення даних типу BYTE у дані типу WORD.

Явні функції користувач повинен ініціювати самостійно, явно. Прикладом може служити перетворення даних типу REAL у дані типу INT. Будь-яка втрата інформації може бути попереджена за допомогою відповідного контролю даних або ж користувач може перевіряти ОК-змінну для перевірки результатів виконання операцій.

Слід урахувати, що при використанні деяких явних функцій перетворення, по-перше, перетворення даних по суті не відбувається й не виконується ніякий код, а по-друге, деякі з функцій перетворення впливають на стан змінної ОК.

Контрольні питання

1. Для розв'язку яких завдань застосовується мова SCL?
2. Що треба визначити для призначення типу даних в SCL-операндів?
3. Які області даних використовуються при адресації в мові SCL?
4. Який тип даних визначає константа?
5. Який синтаксис застосовується для адресації змінних в SCL?
6. Що являють собою *логічні вираження* в мові SCL?
7. Що являють собою *математичні вираження* в мові SCL?
8. Як розподіляються пріоритети операторів?
9. За допомогою яких операторів можна організувати розгалуження програми?
10. При яких умовах застосовується оператор IF?
11. Які завдання вирішуються з використанням оператора CASE?
12. За допомогою яких операторів можна організувати циклічні процедури?
13. Якими операторами можна завершити циклічні процедури?
14. Як здійснюється виклик і завершення роботи блоків програми?

4 ПРОГРАМУВАННЯ МОВОЮ S7-GRAPH

4.1 Особливості мови S7-GRAPH

Мова програмування S7-GRAPH призначена для створення систем послідовного керування.

При використанні цієї мови процес розділяється на окремі кроки, забезпечуючи наочний огляд функціонування системи. Графічна вистава функціонального блоку FB S7-GRAPH називається *секвенсором*.

Секвенсор містить послідовність кроків, які повинні виконуватися в певному порядку, і послідовність переходів, які визначають умови для переходу до наступного кроку. Найпростіша структура секвенсора – лінійна послідовність кроків і переходів без розгалуження. Лінійний секвенсор починається із кроку й завершується переходом.

Приклад лінійного секвенсора показано на рисунку 4.1.

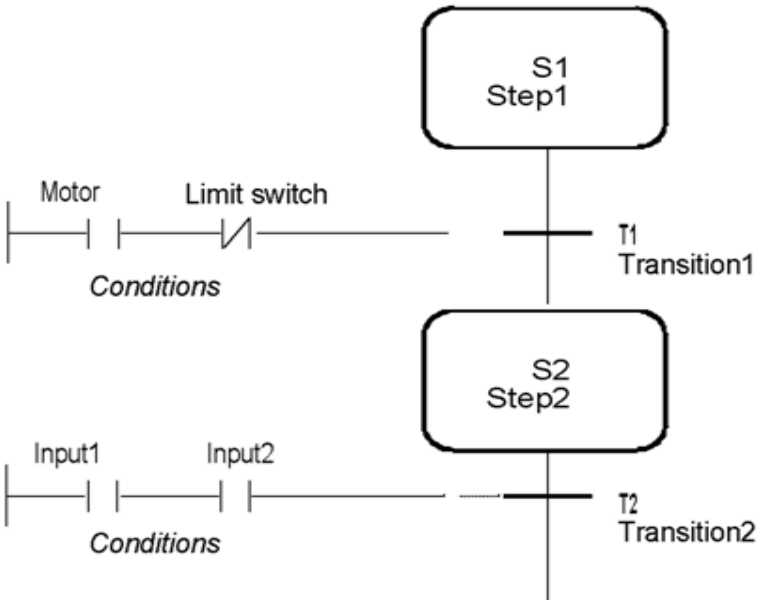


Рисунок 4.1 - Лінійна структура секвенсора

До створення програми секвенсора необхідно розробити концепцію керування технологічним процесом, розбивши процес на окремі кроки.

Основою для розробки концепції служить технологічна схема процесу й тимчасова діаграма процесу (рис. 4.2).

Складність системи послідовного керування залежить від завдання автоматизації. Однак, у загальному випадку, буде потрібно, принаймні, три блоки:

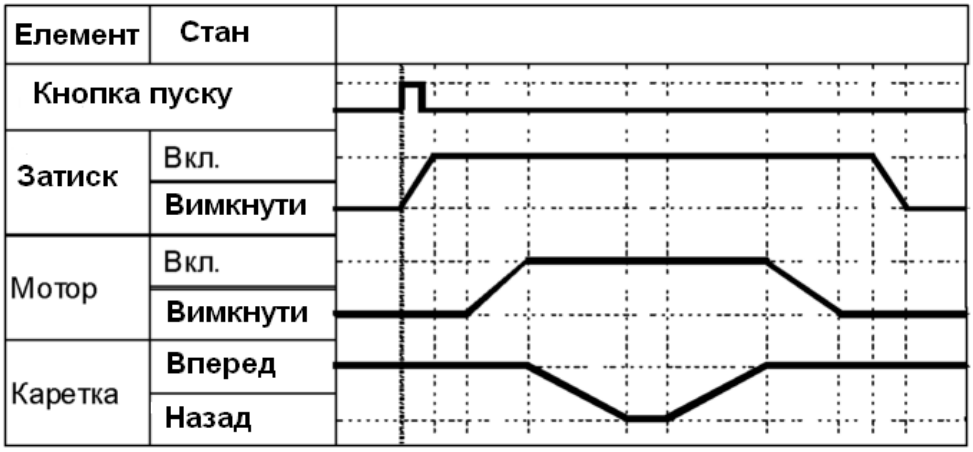


Рисунок 4.2 - Приклад тимчасової діаграми процесу

1. Блок STEP 7, у якому викликається функціональний блок S7-GRAPH. Це може бути організаційний блок (OB), функція (FC), або інший функціональний блок (FB).

2. Функціональний блок FB S7-GRAPH, який описує окремі підзадачі й взаємозалежності системи послідовного керування.

3. Екземплярний блок DB, який містить дані й параметри системи послідовного керування. Екземплярний DB призначається функціональному блоку FB S7-GRAPH і може бути створений системою автоматично.

Структура секвенсора повинна задовольняти наступним правилам:

- FB S7-GRAPH може містити до 256 кроків і переходів. Кроки й переходи вставляються тільки парами.
- Секвенсор може містити максимум 256 відгалужень, у тому числі до 125 альтернативних або до 249 паралельних гілок. Практично недоцільно створювати більш ніж 20-30 гілок.
- Гілка може бути замкнута тільки на гілку, яка розташована ліворуч.
- Наприкінці гілок після переходу можуть бути встановлені перегони. Вони ведуть до з'єднання з попереднім кроком у тій же послідовності або в іншій послідовності у тому ж FB.
- Останов секвенсора може бути встановлений після переходу наприкінці гілки.
- Постійні інструкції можуть бути визначені в спеціальних полях. Вони викликаються однократно в кожному циклі.
- Програмувати структуру секвенсора необхідно на рівні відображення "Секвенсор".

Пари крок-перехід

За замовчуванням FB S7-GRAPH завжди містить *одну* пару крок-перехід

до якої можна додати інші пари. Коли вставляються кроки й переходи, їм автоматично привласнюються номери.

Початковий крок – це крок секвенсора, який стає активним без попереднього опитування умов, тобто коли FB S7-GRAPH запускається в перший раз. Початковий крок – це не обов'язково *перший* крок секвенсора.

Коли секвенсор виконується циклічно, початковий крок, як і будь-який крок, стає активним тільки тоді, коли виконуються умови попереднього переходу, наприклад, "Повернення" (*Return*).

Запуск секвенсора з початкового кроку здійснюється при значенні вхідного параметра FB INIT_SQ = 1.

Стрибок – це перехід до вилученого кроку в даному або в іншому секвенсорі в межах одного FB. Стрибок завжди встановлюється за переходом і закриває секвенсор або шлях гілки до даної точки. Стрибок і точка його призначення графічно відображаються як стрілка, але саме з'єднання не зображується.

Лінійний секвенсор може бути розширений альтернативним і паралельним розгалуженням.

Альтернативна галузі

Альтернативна гілка починається з *переходу*. Виконується тільки та гілка, перехід до якої включається першим. Альтернативна гілка відповідає, таким чином, логіці АБО, у якій активний тільки один шлях. Кожна альтернативна гілка закінчується переходом (рис. 4.3).

Якщо на початку альтернативних гілок одночасно відкрито кілька переходів, пріоритет має крайня ліва гілка з відкритим переходом.

Паралельне розгалуження

Паралельне розгалуження відповідає обробці галузей по логіці І. Паралельне розгалуження починається *загальним переходом*, який активує перший крок *у всіх* паралельних гілках.

На рисунку 4.4 загальними переходами є переходи Т3 і Т7.

Паралельне розгалуження закінчується кроком, підключеним до *загального* фінального переходу. Фінальний перехід дозволяє наступний крок тільки тоді, коли виконані всі паралельні гілки.

Постійні умови

Умови, які потрібно виконувати в більш ніж одній точці секвенсора, можуть бути запрограмовані як постійні умови.

Для програмування умов можна використовувати елементи контактної схеми – нормально розімкнуті або нормально замкнені контакти, компаратори або елементи FBD. У постійній умові можна використовувати до 32 елементів.

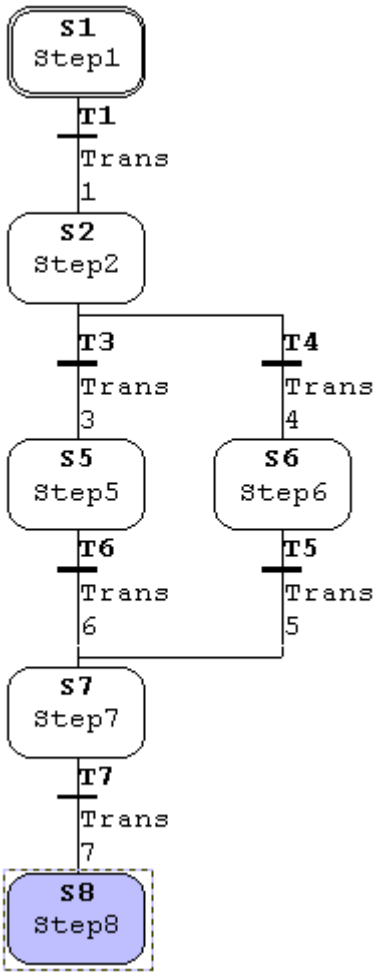


Рисунок 4.3 - Секвенсор з альтернативними гілками

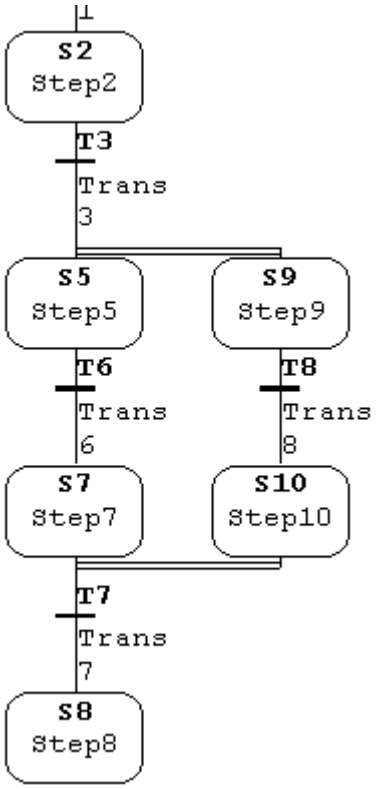


Рисунок 4.4 – Секвенсор з паралельними гілками

Результат обчислення умов зберігається елементом «котушка» в LAD або блоком пам'яті в FBD.

Виклики блоку

Блоки, створені на інших мовах програмування, можуть бути викликані з використанням постійних інструкцій або дій в FB S7-GRAPH. Після того, як викликаний блок буде виконаний, виконання FB S7-GRAPH буде продовжено.

В S7-GRAPH можна викликати функції (FC) і функціональні блоки (FB), запрограмовані на LAD, FBD, STL або SCL, а також системні функції (SFC) і системні функціональні блоки (SFB). Функціональним блокам і системним функціональним блокам повинні бути призначені екземплярні блоки даних DB. Імена блоків можна використовувати в абсолютному виді, наприклад, FC1 або символьно, наприклад, Motor1. При виклику блоків потрібно забезпечити формальні параметри викликуваного блоку дійсними значеннями.

4.2 Програмування дій і умов

Редагування пари крок-перехід

Після визначення структури секвенсора в FB S7-GRAPH можна почати програмувати окремі кроки й переходи.

Порядок програмування не має великого значення.

На рисунку 4.5 показані поля й області розміщення елементів програми при програмуванні кроку й переходу.

Програмування кроку починається з коментаря в полі 1. Коментар може містити до 2048 символів і не впливає на виконання програми.

У полях 2 і 3 уводяться умови блокування й супервізора, відповідно. У полі 4 перебуває символ кроку, а в полі 7 – символ переходу. Поле 5 призначене для опису умов переходу на мовах LAD, FBD, STL і SCL.

Дії, які повинні бути виконані в кроці, заносяться в таблицю, яка розміщається в зоні 6.

Дії управляють входами, виходами й меркерами, вони активують і деактивують кроки секвенсора, а також викликають блоки.

Отже, дії містять команди керування процесом. Вони виконуються в порядку "зверху вниз". Дія може складатися з події, наприклад, "S1 N" на рисунку 4.6 або команди, наприклад, "N M 4.2". У діях вказується адреса (M 4.2) або присвоєння, наприклад A:=B+C.

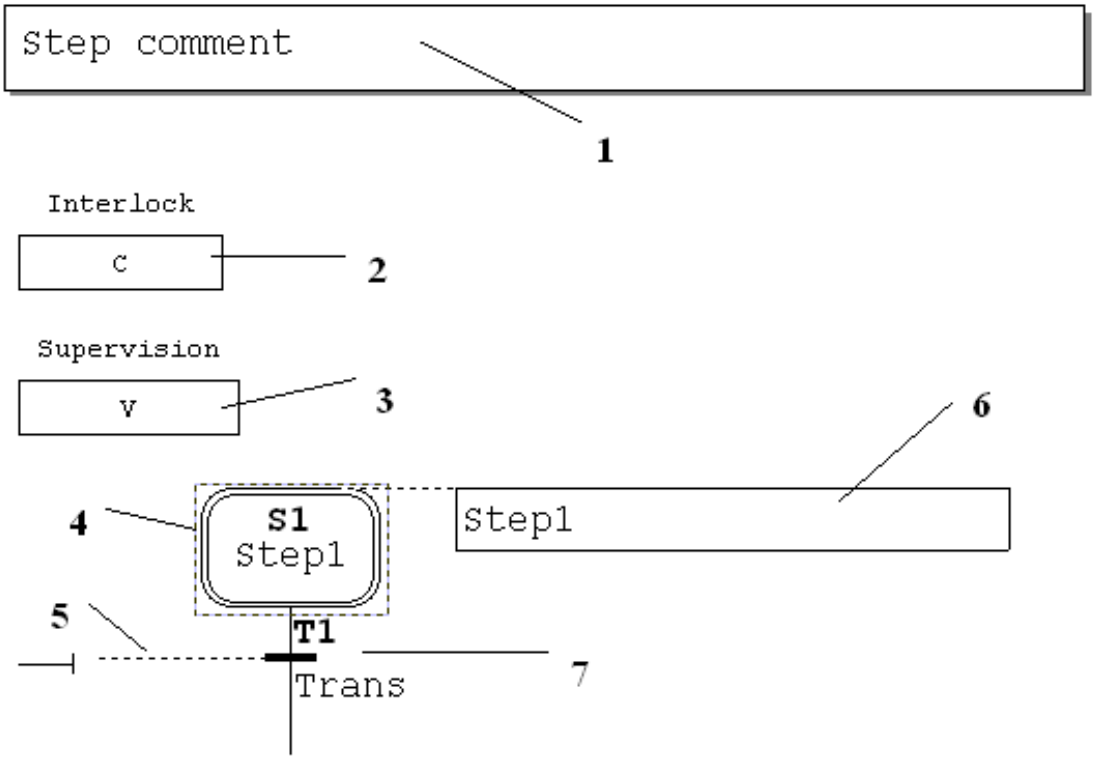


Рисунок 4.5 - Схема розташування областей програмування кроку й переходу

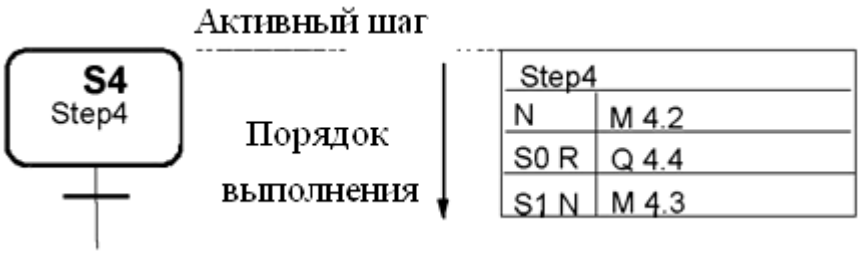


Рисунок 4.6 - Приклад програмування кроку

Дії можна розділити на наступні категорії:

- Стандартні дії.
- Дії, які залежать від подій.
- Дії для активації й деактивації кроків.
- Лічильники й таймери.
- Арифметичні дії.

Кроки, які не містять запрограмованих дій – це порожні кроки. Порожні кроки обробляються так само, як і активні.

Стандартні дії

Стандартні дії наведено в таблиці 4.1.

Таблиця 4.1

Команда	Ідентифікатор	Адреса	Значення
N[C]	Q, I, M, D	m,n	Адреса однократно одержує значення 1
S[C]	Q, I, M, D	m,n	Адреса встановлюється в 1
R[C]	Q, I, M, D	m,n	Адреса скидається в 0
D[C]	Q, I, M, D	m,n	Затримка включення на *n* секунд
L[C]	Q, I, M, D	m,n	Обмеження тривалості адреси на *n* секунд
CALL[C]	FB, FC, SFB, SFC	Номер блоку	Виклик блоку

Усі стандартні дії (команди) можуть комбінуватися з умовою блокування (додається символ C). Такі дії виконуються тільки тоді, коли блокування зняте.

У таблиці використовуються наступні позначення:

D адреса у форматі: Dbi.Dbxm.n;

m адреса байта;

n адреса біта;

SFB, FB системний функціональний блок або функціональний блок;

SFC, FC системна функція або функція.

Константа часу

Інструкції D[C] або L[C] вимагають завдання часу. Час програмується як константа із синтаксисом T#<const> і може бути заданий в необхідній комбінації:

<const>= nd (n доби), nh (n годин), nm (n хвилин), ns (n секунд), nms (n мілісекунд), де n - ціле число.

Приклад:

T#2D3H – константа часу: 2 дня й 3 години.

Дії, які залежать від подій

Дії можуть бути логічно скомбіновані з подіями.

Подія – ця зміна стану кроку або супервізора, блокування, підтвердження, повідомлення або установка реєстрації.

S1: Крок активується.

S0: Крок деактивується.

V1: З'явилася помилка супервізора (неполадка).

V0: Припинилася помилка супервізора (немає неполадок).

L0: Умова блокування включена.

L1: Умова блокування відключена, наприклад, неполадка.

C: Умови блокування задоволені.

A1: Повідомлення підтверджене.

R1: Установлена реєстрація (позитивний фронт на вході REG_EF або REG_S).

Якщо дія логічно скомбінована з подією, сигнал стану визначається по *виявленню фронту*. Це означає, що інструкції можуть виконуватися тільки в тому циклі, у якому відбувається подія.

При настанні подій S0, V0, L0, L1 можливе виконання наступних команд:

N – адреса однократно одержує значення 1;

S – адреса однократно встановлюється в 1;

R – адреса однократно скидається в 0;

CALL – однократно викликається блок.

Лічильники й таймери в діях

Лічильники в діях поводяться, як і в інших мовах програмування S7 – вони не переповняються зверху й знизу. При значенні лічильника рівному 0 біт стану лічильника також рівний 0, а в інших випадках рівний 1.

Для всіх типів подій (S, R, V, L, A) можна застосовувати чотири інструкції:

CS[C] – установка лічильника (завантаження початкового значення);

CU[C] – рахунок нагору;

CD[C] – рахунок униз;

CR[C] – скидання.

Арифметика в діях

У діях можна передбачити також інструкції із простими арифметичними вираженнями – присвоєння у вигляді A:=B, A:=func(B) і A:=B<operator>C.

Дії з арифметичними вираженнями вимагають інструкції N.

При прямому присвоєнні із синтаксисом A:=B використовуються наступні типи даних:

8 біт – BYTE, CHAR;

16 біт – ORD, INT, DATE, S5TIME;

32 біта – WORD, DINT, REAL, TIME, TIME_OF_DAY.

Присвоєння із вбудованою функцією записується у вигляді A:=func(B). Під вбудованими функціями розуміються функції перетворення й складні математичні функції.

Умови

Умови – це двійкові стани процесу, які представляються елементами LAD або блоками FBD. Умови – це події й стани, наприклад, вхід I 2.0 установлений. Умови використовуються в переході для дозволу наступного кроку, при виклику блоку, при введенні блокувань і в супервізорі.

Перехід передає керування наступному кроку секвенсора, якщо виконується логічна умова переходу, тобто, коли результат виконання сегмента рівний 1. Крок, який іде за переходом, стає активним.

Блокування – ця програмувальна умова для заборони виконання окремих дій у кроках, підданих блокуванню. Якщо логічні умови блокування виконуються, дії, скомбіновані із блокуванням, виконуються. Якщо логічні умови блокування не виконуються, це вважається порушенням. Запрограмоване блокування позначається буквою C поруч із кроком.

Супервізор – це програмувальна умова для контролю кроку, що впливає на спосіб, яким секвенсор передає керування від одного кроку до наступного. Запрограмований супервізор позначається на всіх рівнях вистави буквою V зліва від кроку.

Якщо логічна умова супервізора виконується, це означає неполадку й сигналізує про подію V1. Секвенсор не передає керування наступному кроку, активним залишається поточний крок. Якщо логічна умова супервізора не виконується, то неполадок немає й секвенсор передає керування наступному кроку.

4.3 Установка параметрів

Ім'я й номер кроку призначаються системою автоматично від stepl до step999, однак його можна змінити. Ім'я переходу призначається системою також автоматично від Transl до Trans999, його також можна змінити.

Ім'я кроку й переходу може містити максимум 24 символів (букв і цифр). Перший символ повинен бути буквою.

Призначення параметрів FB S7-GRAPH

Множина параметрів блоку S7-GRAPH залежить від передбаченого використання секвенсора й доступної пам'яті CPU.

У вікні опису змінних існує чотири стандартних набори параметрів:

- Мінімальний набір з 3 параметрами (Minimum).
- Стандартний набір з 21 параметром (Standard).

- Максимальний набір з 31 параметром (Maximum).
- Користувацький набір з 53 параметрами (User-specific).

Зі стандартних наборів можна вилучити не використовувані параметри. Та або інша множина параметрів вибирається згідно з розв'язуваним завданням.

Існують наступні рекомендації:

Множина параметрів Minimum використовується тоді, коли секвенсор працює тільки в автоматичному режимі й не потрібно додаткових функцій керування й моніторингу.

Множина параметрів Standard використовується тоді, коли секвенсор працює в різних режимах і потрібен зворотний зв'язок від процесу.

Множина параметрів Maximum використовується тоді, коли секвенсор працює в різних режимах і потрібен зворотний зв'язок від процесу, можливість підтвердження повідомлень і додаткові функції моніторингу для обслуговування системи.

Користувацький набір параметрів User-specific надає ті ж можливості, що й набір Maximum.

На рисунку 4.7 показаний функціональний блок S7-GRAPH зі стандартним набором вхідних і вихідних параметрів.

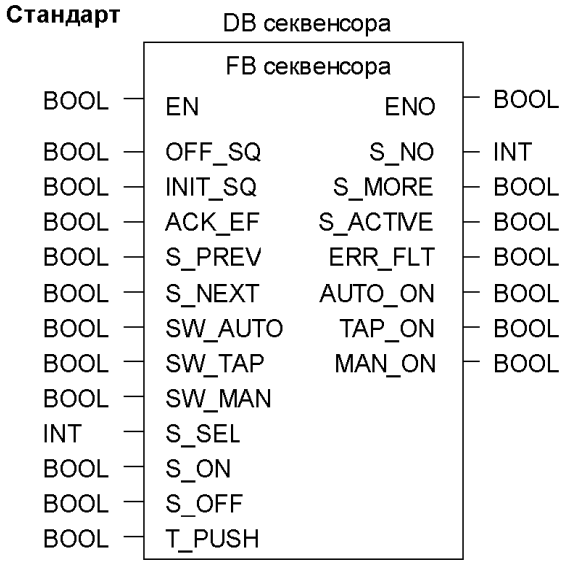


Рисунок 4.7 - Відображення блоку з набором параметрів Standard

Перелік вхідних параметрів для наборів Minimum і Standard наведено в таблиці 4.5, а вихідних – у таблиці 4.6.

Слід урахувати, що функціональний блок S7-GRAPH реагує на позитивний фронт вхідного параметра.

Таблиця 4.5 - Вхідні параметри блоку S7-GRAPH

Параметр	Тип даних	Опис	Minim.	Stand.
EN	BOOL	Вхід дозволу	•	•
OFF_SQ	BOOL	Деактивація секвенсора		•
INIT_SQ	BOOL	Скидання секвенсора	•	•
ACK_EF	BOOL	Підтвердження помилок, перемикання до наступного кроку		•
S_PREV	BOOL	В автоматичному режимі крок назад. У ручному режимі – показати номер попереднього кроку на S_NO		•
S_NEXT	BOOL	В автоматичному режимі крок уперед. У ручному режимі – показати номер наступного кроку на S_NO		•
SW_AUTO	BOOL	Включити автоматичний режим		•
SW_TAP	BOOL	Включити режим з підштовхуванням		•
SW_MA	BOOL	Включити ручний режим		•
S_SEL	INT	Уведення номера кроку		•
S_ON	BOOL	Відобразити крок у ручному режимі		•
S_OFF	BOOL	Виключити відображення кроку		•
T_PUSH	BOOL	Передача керування в покроковому режимі		•

4.4 Створення структури й установка режимів системи керування

Визначення структури програми й вбудовування секвенсора

Для кожного секвенсора S7-GRAPH створюється FB з екземплярним DB. Оскільки разом із програмами, створеними в S7-GRAPH, зазвичай потрібні інші програми, найкращий спосіб – це виклик всіх FB в одному блоці, як показано на рисунку 4.8.

Різні функції на окремих рівнях виконуються циклічно в тому порядку, у якому вони викликаються.

Рекомендується наступний порядок виклику:

- Секвенсори, як загальні функції більш високого рівня.

Таблиця 4.6 - Вихідні параметри блоку S 7-GRAPH

Параметр	Тип даних	Опис	Minim.	Stand.
ENO	BOOL	Дозвіл виходу	•	•
S_NO	INT	Відображення номера кроку		•
S_MORE	BOOL	Вибір іншого кроку		•
S_ACTIVE	BOOL	Відображення активного кроку		•
ERR_FLT	BOOL	Групова неполадка		•
AUTO_ON	BOOL	Індикація автоматичного режиму		•
TAP_ON	BOOL	Індикація режиму з підштовхуванням		•
MAN_ON	BOOL	Індикація ручного режиму		•

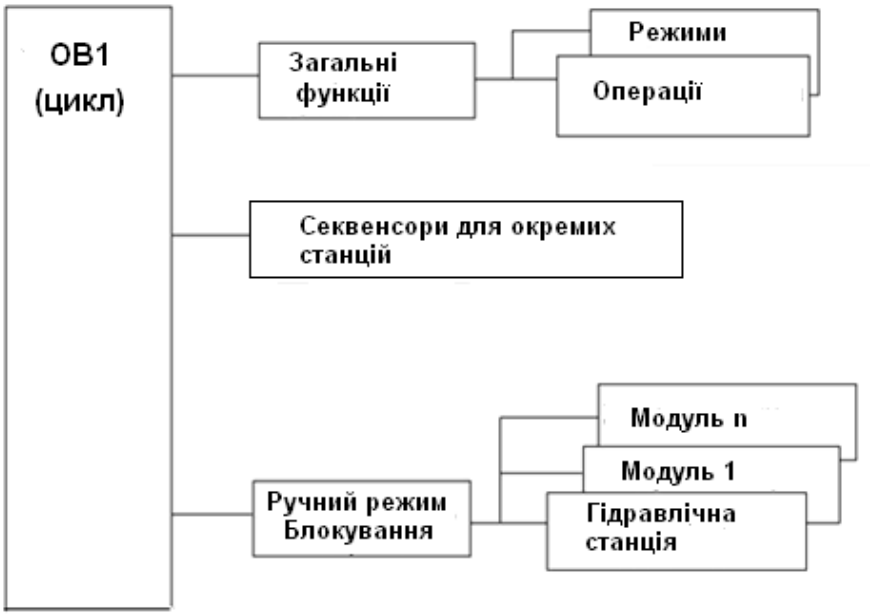


Рисунок 4.8 - Структура програми з використанням секвенсорів

• Секвенсори для окремих станцій, які викликаються в FB "Sequencers".

• За секвенсорами викликаються розділи програми для ручного режиму, блокувань і постійних функцій моніторингу для модулів.

Режими роботи

Залежно від ситуації, користувачеві потрібні різні режими роботи системи й установки.

У незв'язаних виробничих осередках, наприклад, при складанні корпуса, можливі наступні режими:

• Автоматичний (SW_AUTO), у якому керування передається

наступному кроку, якщо виконується умова переходу.

• З підштовхуванням (SW_TAP) – це різновид автоматичного режиму із зупинкою після *кожного* обробленого кроку (напівавтомат). Секвенсор передає керування, коли виконується умова переходу й виникає позитивний фронт (перехід з 0 в 1) у параметрі T_PUSH.

• Автомат з додатковими умовами дозволу кроку SW_TOP – це різновид автоматичного режиму при початковому запуску проекту з покроковим тестуванням керуючої системи.

• Ручний (SW_MAN) – це пряме керування модулями й функціями. Ручний режим вибирається, наприклад, для перевірки секвенсора. Ручний режим характеризується тим, що кроки можна вибирати й скасовувати вручну.

Якщо є панель вибору режиму, то перемикачем на керуючій панелі за допомогою простої логіки можна одержати необхідні сигнали для різних режимів роботи секвенсора.

Контрольні питання

1. Що називається секвенсором?
2. Що є основою для створення програми секвенсора?
3. Які блоки потрібні для застосування секвенсора в S7-програмі?
4. Які правила необхідно виконати при створенні секвенсора?
5. Що являє собою початковий крок секвенсора?
6. Які правила застосовуються для побудови альтернативних гілок?
7. Які правила застосовуються для побудови паралельних гілок?
8. Що розуміється під постійними умовами?
9. Як можна викликати функціональний блок FB S7-GRAPH?
10. Що програмується в кроці секвенсора?
11. Що програмується в переході секвенсора?
12. Які стандартні дії передбачені при програмуванні секвенсора?
13. Які події аналізуються секвенсором?
14. Яку арифметику можна застосовувати в діях при програмуванні секвенсора?
15. Що розуміється під терміном "блокування"?
16. Якими наборами параметрів можна скористатися при програмуванні блоку S7-GRAPH?
17. Який порядок передбачений при виклику секвенсорів у програмі?
18. Які режими роботи можливі при реалізації секвенсора?

ЛІТЕРАТУРА

1. Бергер Ганс. Автоматизация посредством STEP 7 с использованием STL и SCL и программируемых контроллеров SIMATIC S7-300/400. - 2001. [Электронный ресурс]. Режим доступа: http://mirknig.com/knigi/nauka_ucheba/1181227029-avtomatizaciya-posredstvom-step-7-s-ispolzovaniem.html
2. Бергер Ганс. Автоматизация с помощью программ STEP7 LAD и FBD. Изд. 2-е, перераб, 2001. [Электронный ресурс]. Режим доступа: http://www.twirpx.com/file/119315/
3. SIMATIC S7. Введение в STEP 7. [Электронный ресурс]. Режим доступа: http://www.twirpx.com/file/60377/
4. SIMATIC. Программирование с помощью STEP 5 V5.3. Руководство. Редакция 01/2004, A5E00261405-01.
5. Интерактивный каталог продуктов Siemens IA&DT. Техника автоматизации SIEMENS. [Электронный ресурс]. Режим доступау: https://eb.automation.siemens.com/goos/catalog/Pages/ProductData.aspx?catalogRegion=RU&language=ru&nodeid=10045207&tree=CatalogTree&regionUrl=%2fru#activetab=product&
6. Программируемые контроллеры SIMATIC S7. [Электронный ресурс]. Режим доступа: *http://automation-drives.ru/as/products/simatic_s7/*
7. Автоматизация в промышленности. Каталог Siemens CA01 2010. [Электронный ресурс]. Режим доступа: *http://www.sms-automation.ru/distribution/Siemens/catalog/index.php?nodeid=1102*
8. SIMATIC. HiGraph для S7-300/400. Руководство [Электронный ресурс]. Режим доступа: http://automation-drives.ru/as/download/doc/software/eng/HiGraph_V4.1_r.pdf
9. SIMATIC. S7-GRAPH V5.3 для S7-300/400. Программирование систем последовательного управления. Руководство. Редакция 02/2004, A5E00290656-01
10. SIMATIC. Программируемые контроллеры S7-400, M7-400. Руководство пользователя C79000–G7076–C400–01. Выпуск 2