STRUCTURED TEXT С НУЛЯ: СИНТАКСИС, ТИПЫ ДАННЫХ, ОПЕРАТОРЫ В CODESYS V3.5

Telegram: https://t.me/plcmasters

Электронные учебные пособия:
https://electricalschool.info/e-textbooks.html

Концептуальная структура издания

Материал построен по принципу восходящего усложнения: от архитектуры памяти контроллера и лексических правил языка к сложным комбинированным типам данных, математическому аппарату и отчуждаемым программным компонентам (POU).

Содержание

- *Пример 1:* Создание минимального рабочего проекта в эмуляторе CODESYS Control Win V3.

- *Пример 2:* Прямая адресация дискретного ввода-вывода через переменные конфигурации (%IX0.0, %QX0.0).

- *Пример 3:* Использование символических переменных без жесткой физической привязки.

- *Пример 4:* Отработка ошибок компиляции: пропущенная точка с запятой, необъявленная переменная, конфликт имен.

- *Пример 5:* Диагностика переменных через режим онлайн-мониторинга (Online Monitoring) и запись форсированных значений (Force Values).

• **Контрольные вопросы и упражнения к главе 1** (10 теоретических вопросов, 3 задачи на исправление синтаксических ошибок).

Глава 2. Фундаментальная система типов данных

Раздел раскрывает низкоуровневую физику хранения чисел в памяти ПЛК, границы диапазонов, переполнение и правила явного приведения типов.

• **2.1. Логический тип данных.** `BOOL`: хранение в памяти, правила интерпретации, литералы `TRUE` и `FALSE`.

• **2.2. Битовые строки.** `BYTE` (8 бит), `WORD` (16 бит), `DWORD` (32 бита), `LWORD` (64 бита). Представление в шестнадцатеричной (`16#FF`) и двоичной (`2#1010_1100`) формах.

• **2.3. Целочисленные знаковые и беззнаковые типы.** Сравнительная таблица: `SINT`, `USINT`, `INT`, `UINT`, `DINT`, `UDINT`, `LINT`, `ULINT`. Дополнительный код отрицательных чисел и явление аппаратного переполнения.

• **2.4. Вещественные числа с плавающей точкой.** Стандарт IEEE 754: одинарная точность REAL (32 бита) и двойная точность LREAL (64 бита). Потеря точности мантиссы, деление на ноль, значения NaN и Infinity.

• **2.5. Символьные строки.** STRING (ASCII / Windows-1251, 1 байт на символ) и WSTRING (Unicode UTF-16, 2 байта на символ). Спецификаторы длины STRING(80). Нуль-терминированные строки.

• **2.6. Типы физического времени.** Интервалы TIME и LTIME. Литералы времени (T#1h20m30s500ms).

• **2.7. Календарные типы.** DATE, TIME_OF_DAY (TOD), DATE_AND_TIME (DT).

• **2.8. Преобразование типов (Type Casting).** Неявные преобразования и строгие операторы явного приведения (TO_INT, REAL_TO_DINT, TIME_TO_UDINT).

• **Практикум главы 2:**

○ *Пример 6:* Демонстрация аппаратного переполнения INT (32767 + 1 = -32768) в онлайн-режиме.

○ *Пример 7:* Сравнение вещественных чисел: почему rVal1 = rVal2 опасно и как использовать ABS(rVal1 - rVal2) < EPSILON.

○ *Пример 8:* Извлечение отдельных миллисекунд из переменной типа TIME через приведение типов.

○ *Пример 9:* Форматирование строковых сообщений с конкатенацией числовых показаний датчиков.

○ *Пример 10:* Распаковка байта на 8 отдельных логических сигналов BOOL.

- *Пример 11:* Упаковка 16 флагов состояний в единое коммуникационное слово WORD для передачи по Modbus.

- *Пример 12:* Безопасное масштабирование сигнала датчика без потери дробной части.

• **Контрольные вопросы и упражнения к главе 2** (12 вопросов, 4 задачи на расчет занимаемой памяти и допустимость приведения).

Глава 3. Операторы присваивания, арифметика и битовая логика

Раздел формирует базу построения вычислительных выражений, раскрывая приоритеты операций и специфику выполнения вычислений на микропроцессорах контроллеров.

• **3.1. Оператор присваивания.** Простое присваивание :=, семантика копирования значений, порядок вычисления правых частей.

• **3.2. Арифметические операторы.** Сложение +, вычитание -, умножение *, деление /, взятие остатка MOD. Особенности целочисленного деления.

• **3.3. Логические (булевы) операторы.** AND, OR, XOR, NOT. Построение комбинаторных схем управления.

• **3.4. Операторы отношения и сравнения.** =, <>, <, >, <=, >=.

• **3.5. Побитовые сдвиги и вращения.** SHL (логический сдвиг влево), SHR (логический сдвиг вправо), ROL (циклический сдвиг влево), ROR (циклический сдвиг вправо). Применение для умножения/деления на степени двойки и фильтрации масок.

• **3.6. Выбор максимального, минимального и ограничение диапазона.** Стандартные математические операторы MIN, MAX, LIMIT, SEL, MUX.

• **3.7. Таблица старшинства и приоритета операторов.** Использование круглых скобок для предотвращения неявных коллизий вычисления.

• **Практикум главы 3:**

○ *Пример 13:* Реализация релейной логики «Пуск-Стоп» с самоподхватом математическим булевым выражением без IF.

○ *Пример 14:* Программный селектор аналогового датчика с резервированием по схеме 2 из 3 через SEL и LIMIT.

○ *Пример 15:* Циклическое бегущее реле на базе битового сдвига ROL.

○ *Пример 16:* Ограничение максимальной скорости нарастания аналогового задания привода (Ramp-ограничитель).

○ *Пример 17:* Деление секунд на часы, минуты и остаток секунд через целочисленные / и MOD.

○ *Пример 18:* Программная установка, сброс и инвертирование бита в управляющем слове по маске.

○ *Пример 19:* Вычисление евклидова расстояния между двумя точками на плоскости по теореме Пифагора.

• **Контрольные вопросы и упражнения к главе 3** (10 вопросов, 5 задач на расстановку скобок и вычисление результирующих значений).

Глава 4. Управление потоком выполнения: ветвления и циклы

Раздел посвящен алгоритмическим конструкциям, определяющим порядок исполнения инструкций, с жестким акцентом на исключение зависания задачи ПЛК.

- **4.1. Условный оператор IF-THEN-ELSE.** Полная и неполная формы. Вложенные конструкции ELSIF. Почему порядок условий имеет решающее значение для быстродействия.

- **4.2. Многопутевой выбор CASE-OF.** Синтаксис переключателя, одиночные метки, списки меток через запятую, диапазоны значений (1..10:), обязательная защитная ветвь ELSE. Преимущества CASE перед длинной лесенкой ELSIF.

- **4.3. Опасности циклических конструкций в ПЛК.** Понятие сторожевого таймера задачи (Task Watchdog). Почему бесконечный цикл WHILE TRUE приводит к падению контроллера в Hard Fault.

- **4.4. Детерминированный цикл FOR-TO-BY-DO.** Синтаксис, переменная счетчика, шаг цикла BY, условия досрочного завершения. Корректное использование при обходе массивов.

- **4.5. Итерационные циклы с предусловием и постусловием.** WHILE-DO и REPEAT-UNTIL. Условия гарантированного выхода из цикла, счетчики предельного числа итераций.

- **4.6. Операторы прерывания и пропуска.** EXIT (досрочный выход из цикла), CONTINUE (пропуск шага итерации), RETURN (досрочный выход из программы/блока).

- **Практикум главы 4:**

  - *Пример 20:* Реализация трехпозиционного селектора режимов («Ручной / Автомат / Наладка») через оператор CASE.

  - *Пример 21:* Поиск максимального и минимального значения в массиве температур через цикл FOR.

- *Пример 22:* Линейный поиск индекса аварийного датчика в массиве с использованием прерывания EXIT.
- *Пример 23:* Безопасный алгоритм расчета контрольной суммы CRC-8 с защитой от зависания через WHILE.
- *Пример 24:* Расчет факториала и степенных рядов с отсечкой по максимальному времени выполнения.
- *Пример 25:* Алгоритм сортировки массива методом пузырька (Bubble Sort) с оптимизацией флагом перестановок.
- *Пример 26:* Двухуровневый ступенчатый термостат с гистерезисом на базе каскадных операторов IF-ELSIF.

- **Контрольные вопросы и упражнения к главе 4** (12 вопросов, 4 задачи на оптимизацию ветвлений и аудит кода на зависание).

Глава 5. Пользовательские типы данных (DUT)

Глава учит структурировать сырые переменные в технологические модели механизмов и рецептурные наборы.

- **5.1. Концепция DUT (Data Unit Type).** Назначение, место объявления в дереве проекта, экспорт и повторное использование.
- **5.2. Статические массивы (ARRAY).** Одномерные и многомерные массивы. Определение границ (1..10, -5..5). Индексация, доступ к элементам, проверка выхода за границы массива (CheckBounds).
- **5.3. Перечислимые типы (ENUM).** Объявление состояний и режимов, задание явных числовых значений элементам. Прагмы {attribute 'qualified_only'} и {attribute 'strict'}.

• **5.4. Структуры данных (STRUCT).** Объединение разнородных переменных в единую технологическую сущность. Вложенные структуры. Точечная нотация обращения к полям (Pump1.rSpeed).

• **5.5. Псевдонимы типов (ALIAS) и диапазоны (SUBRANGE).** Ограничение допустимых значений переменной на уровне компилятора (TYPE T_Percentage : INT (0..100);).

• **5.6. Битовые объединения (UNION).** Разделение одной области памяти между переменными разного типа (например, одновременное представление 4 байт как одного DWORD или REAL).

• **Практикум главы 5:**

○ *Пример 27:* Создание паспорта электропривода: структура ST_MotorData (флаги, ток, скорость, температура, моточасы).

○ *Пример 28:* Создание кольцевого буфера на базе массива структур для регистрации событий.

○ *Пример 29:* Матрица рецептурного дозирования: двумерный массив ARRAY[1..10, 1..5] OF REAL.

○ *Пример 30:* Описание режимов работы упаковочной машины через типизированное перечисление E_MachineMode.

○ *Пример 31:* Быстрая распаковка телеграммы Modbus TCP через структуру с объединением UNION.

○ *Пример 32:* Функция инициализации полей структуры значениями по умолчанию.

○ *Пример 33:* Создание таблицы калибровочных точек датчика (массив структур PointX, PointY).

• **Контрольные вопросы и упражнения к главе 5** (10 вопросов, 3 задачи на проектирование структур и расчет выравнивания памяти).

Глава 6. Программные компоненты: функции (FUN) и функциональные блоки (FB)

Завершающий раздел выстраивает мост к профессиональной инкапсуляции алгоритмов, разделяя чистые вычисления без состояния и объекты с долговременной памятью.

• **6.1. Классификация POU по стандарту IEC 61131-3.** Программы (PROGRAM), функции (FUNCTION), функциональные блоки (FUNCTION_BLOCK). Сравнительный анализ жизненного цикла в оперативной памяти.

• **6.2. Анатомия функции (FUN).** Входные параметры (VAR_INPUT), возвращаемое значение по имени функции, временные переменные (VAR_TEMP). Почему функция не может хранить предысторию между вызовами.

• **6.3. Анатомия функционального блока (FB).** Входы (VAR_INPUT), выходы (VAR_OUTPUT), двунаправленные ссылки (VAR_IN_OUT), внутреннее статическое состояние (VAR). Понятие экземпляра (инстанса) блока.

• **6.4. Размещение экземпляров в памяти.** Создание нескольких экземпляров одного FB (например, 10 одинаковых насосов). Накладные расходы на экземпляр.

• **6.5. Работа с параметрами по ссылке и значению.** Передача тяжелых массивов и структур через VAR_IN_OUT для экономии стека и предотвращения копирования памяти.

• **6.6. Вызов стандартных блоков из библиотек.** Правила подключения и инстанцирования таймеров (TON, TOF), счетчиков (CTU) и триггеров (R_TRIG, F_TRIG).

• **Практикум главы 6:**

○ *Пример 34:* Чистая математическая функция: полиномиальное масштабирование аналогового сигнала FUN_ScaleLinear.

○ *Пример 35:* Логическая функция контроля четности байта FUN_ParityCheck.

○ *Пример 36:* Разработка функционального блока реверсивного пускателя FB_MotorStarter с контролем обратной связи.

○ *Пример 37:* Разработка функционального блока цифрового фильтра скользящего среднего FB_MovingAverage.

○ *Пример 38:* Разработка параметризуемого генератора импульсов заданной скважности FB_Blinker.

○ *Пример 39:* Вложенные вызовы: создание комплексного агрегата задвижки FB_Valve с использованием внутри экземпляров таймеров TON.

○ *Пример 40:* Сквозной курсовой проект: интеграция функций и блоков в главную программу PLC_PRG для управления автоматической сортировочной станцией.

• **Контрольные вопросы и упражнения к главе 6** (12 вопросов, 3 задачи на проектирование интерфейсов POU).

Приложения и методический аппарат

Вспомогательный справочный аппарат позволяет использовать пособие как настольное инженерное руководство:

• **Приложение А.** Сводная таблица стандартных операторов языка Structured Text с указанием типов операндов и ограничений.

• **Приложение Б.** Таблица приоритетов выполнения операций стандарта IEC 61131-3.

• **Приложение В.** Таблица соответствия префиксов венгерской нотации типам данных.

• **Приложение Г.** Эталонные решения и листинги к расчетно-практическим упражнениям глав 1–6.

ВВЕДЕНИЕ

Программирование систем промышленной автоматизации коренным образом отличается от создания программного обеспечения общего назначения для персональных компьютеров или веб-серверов. В промышленной среде во главу угла ставятся детерминированность вычислений, абсолютная предсказуемость времени реакции на внешние физические события и катастрофическая цена ошибки. Программа, управляющая химическим реактором, насосной станцией высокого давления или скоростным раскроечным порталом, не имеет права зависнуть в ожидании освобождения памяти, аварийно завершиться с системной ошибкой доступа к указателю или самопроизвольно изменить порядок выполнения инструкций.

Международный стандарт IEC 61131-3 (в отечественной нормативной базе адаптированный как ГОСТ Р МЭК 61131-3) зафиксировал унифицированные требования к языкам программирования программируемых логических контроллеров (ПЛК). Среди пяти стандартизированных языков особое место занимает Structured Text (ST, Структурированный текст). В отличие от

графических языков релейно-контактных схем (LD) или функциональных блоковых диаграмм (FBD), язык ST представляет собой высокоуровневый текстовый язык программирования с блочной структурой, синтаксически восходящий к языкам Pascal и Ada.

Исторически графические языки создавались для облегчения перехода инженеров-электриков от физических релейных шкафов к микропроцессорным устройствам. Однако по мере усложнения технологических задач визуальные схемы перестали справляться с лавинообразным ростом математических вычислений, реализацией сложных численных алгоритмов, протоколов сетевого обмена, манипуляцией массивами данных и сложными рецептурными матрицами. Графическая диаграмма, насчитывающая сотни параллельных веток и логических блоков, превращается в визуальный хаос, в котором поиск ошибки требует колоссальных временных затрат. Язык Structured Text лишен этого недостатка: он обеспечивает максимальную плотность и структурированность кода, полную прозрачность математических выражений, алгоритмическую гибкость и идеальную совместимость с современными распределенными системами контроля версий (такими как Git).

Исполнение программ на языке ST в контроллере жестко подчинено циклической модели задачи real-time операционной системы. Программа никогда не запускается однократно от начала до конца, как это происходит в скриптовых языках. Цикл задачи ПЛК непрерывно повторяется через строго фиксированные кванты времени: сначала системная шина контроллера обновляет снимок дискретных и аналоговых входов в памяти процесса, затем процессор вызывает программный код пользователя на языке ST от первой до последней инструкции, после чего сформированный образ выходов

копируется на физические исполнительные реле, контакторы и модули полевых сетей.

Цель настоящего учебного пособия — заложить фундаментальную алгоритмическую базу для инженера, начинающего путь в промышленной автоматизации. Пособие последовательно раскрывает устройство памяти контроллера, правила построения выражений, строгую типизацию данных стандарта, логические и математические операции, а также принципы проектирования отчуждаемых программных блоков в среде CODESYS v3.5 — признанном мировом стандарте инструментальной среды разработки для сотен семейств промышленных ПЛК.

ГЛАВА 1. АНАТОМИЯ СРЕДЫ РАЗРАБОТКИ И БАЗОВЫЙ СИНТАКСИС ST

1.1. Цикл задачи ПЛК и модель памяти процесса

Чтобы понимать логику выполнения любой инструкции на языке Structured Text, разработчик обязан представлять, как программа взаимодействует с оперативной памятью контроллера и системным планировщиком задач (Task Scheduler). В традиционном программировании для операционных систем Windows или Linux приложение может запросить у процессора паузу, заснуть на несколько секунд или ожидать нажатия клавиши в бесконечном цикле. В программируемом контроллере подобное поведение недопустимо: любая задержка при выполнении программы приведет к срабатыванию аппаратного сторожевого таймера (Hardware Watchdog), сбросу силовых цепей и аварийному останову установки.

Классический рабочий цикл сканирования циклической задачи ПЛК делится на три жестко синхронизированных этапа. На первом этапе процессорный модуль опрашивает модули ввода, расположенные на локальной шине или подключенные по промышленным сетям (EtherCAT, Profinet, Modbus), и копирует их состояние в специализированную область памяти — образ входов процесса (Process Image Input). На втором этапе управление передается коду на языке ST. Программа считывает переменные исключительно из образа входов, выполняет математические и логические вычисления и помещает результаты в промежуточную область — образ выходов процесса (Process Image Output). На третьем этапе системный планировщик физически передает сформированные байты из образа выходов в выходные транзисторные ключи или реле модулей вывода.

Из этой архитектуры вытекает важнейшее свойство языка ST: физические выходы контроллера не переключаются в тот же момент, когда в тексте программы выполняется оператор присваивания. Если в начале программы вы записали команду на включение пневмоцилиндра, а через десять строк кода по аварийному условию записали команду на его отключение, исполнительный механизм даже не успеет стронуться с места: на выходные клеммы попадет только то итоговое значение, которое осталось в памяти образа выходов на момент завершения последнего оператора программы.

Память процесса по стандарту IEC 61131-3 делится на три физические и логические зоны:

- Зона входов (префикс `%I`): область памяти, доступная программе только для чтения, куда драйверы ввода непрерывно записывают значения датчиков, кнопок и концевых выключателей.

• Зона выходов (префикс %Q): область памяти, куда программа записывает рассчитанные сигналы управления пускателями, электромагнитными клапанами и преобразователями частоты.

• Зона внутренних маркеров (префикс %M): память промежуточных переменных общего назначения, системных флагов и регистров обмена со SCADA-системами и панелями оператора.

В современных проектах на CODESYS v3.5 прямая адресация через физические адреса вида %IX0.0 или %QD4 уступает место символическим переменным. Разработчик оперирует осмысленными мнемоническими именами (xEmergencyStop, rReactorTemperature), а привязка этих имен к физическим каналам модулей осуществляется централизованно в конфигураторе ввода-вывода (I/O Mapping). Это изолирует алгоритмическую логику от аппаратной конфигурации: при замене модуля ввода или смене типа контроллера текст программы на ST остается абсолютно неизменным.

1.2. Структура программной единицы (POU)

Вся программная логика в стандарте IEC 61131-3 упаковывается в программные организационные единицы — POU (Program Organization Unit). Базовой единицей верхнего уровня является программа (PROGRAM), примером которой служит стандартный блок PLC_PRG, создаваемый по умолчанию в любом новом проекте CODESYS.

Текстовый файл или редактор POU в среде разработки всегда четко разделен на две изолированные части: секцию объявления переменных и секцию исполняемого кода. Смешивание объявлений и алгоритма, допустимое в некоторых скриптовых языках, в Structured Text категорически запрещено. Компилятор требует предварительного описания абсолютно каждого

идентификатора, определяя его тип данных, начальное значение и область видимости еще до того, как этот идентификатор появится в исполняемых инструкциях.

Секция объявления переменных всегда открывается ключевым словом, определяющим категорию памяти, и закрывается служебным словом END_VAR. В зависимости от назначения переменных используются следующие спецификаторы:

• VAR ... END_VAR — обычные локальные переменные программы или функционального блока. Они размещаются в статической памяти процесса, сохраняют свои значения между последовательными циклами сканирования задачи, но недоступны для чтения или модификации из других программных блоков.

• VAR_INPUT ... END_VAR — входные переменные, передаваемые в функциональный блок или функцию извне.

• VAR_OUTPUT ... END_VAR — выходные переменные, формируемые алгоритмом блока для передачи во внешний код.

• VAR_IN_OUT ... END_VAR — двунаправленные параметры, передаваемые по ссылке (адресу памяти), позволяющие модифицировать внешнюю переменную прямо внутри вызываемого алгоритма без накладных расходов на копирование.

• VAR_TEMP ... END_VAR — временные переменные, размещаемые на системном стеке выполнения. Они инициализируются заново при каждом входе в блок и полностью уничтожаются при выходе из него, не сохраняя предысторию между циклами контроллера.

• VAR RETAIN ... END_VAR — энергонезависимые реманентные переменные, которые сохраняют свои числовые значения при кратковременном отключении сетевого электропитания шкафа управления за счет резервной ионисторной или аккумуляторной памяти контроллера.

Под секцией объявления располагается рабочее тело программы. В теле POU операторы языка выполняются строго последовательно: сверху вниз, от первой инструкции до последней. Ни одна инструкция не может выполниться вне цикла сканирования задачи.

1.3. Лексические основы, идентификаторы и венгерская нотация

Текст программы на Structured Text формируется из токенов: идентификаторов, ключевых слов, числовых литералов, строковых констант, разделителей и знаков операций. Язык ST регистронезависим: идентификаторы MotorSpeed, motorspeed и MOTORSPEED указывают на одну и ту же область памяти. Тем не менее пренебрежение правилами регистра ведет к резкому падению читаемости проекта, поэтому в промышленной практике принято строгое следование единому стилю кодирования.

Правила формирования пользовательских идентификаторов жестко регламентированы:

• Идентификатор может состоять из букв латинского алфавита (A–Z, a–z), цифр (0–9) и символа подчеркивания _.

• Первым символом имени обязана быть буква или знак подчеркивания; идентификатор не может начинаться с цифры.

• Запрещено использование двух символов подчеркивания подряд (например, Sensor__Value недопустимо), а также знак подчеркивания в качестве финального символа.

• Имя переменной не должно совпадать со служебными зарезервированными словами стандарта (такими как IF, CASE, TON, REAL, RETURN).

В индустрии систем автоматизации де-факто стандартом стало применение модифицированной венгерской нотации, согласно которой имя переменной предваряется коротким префиксом строчными буквами, указывающим на ее фундаментальный тип данных и физический смысл. Это исключает грубые ошибки при чтении сложных участков кода.

Основной перечень префиксов венгерской нотации, принятый в среде CODESYS:

• x — логический булев тип (BOOL), например: xEmergencyStop, xValveOpened;

• b — однобайтовая битовая строка (BYTE), например: bControlByte;

• w — двухбайтовая битовая строка (WORD), например: wStatusWord;

• dw — четырехбайтовая битовая строка (DWORD), например: dwErrorCodeMask;

• i — 16-битное знаковое целое число (INT), например: iCurrentStep;

• di — 32-битное знаковое целое число (DINT), например: diEncoderPulses;

• ui — 16-битное беззнаковое целое число (UINT), например: uiBatchQuantity;

• udi — 32-битное беззнаковое целое число (UDINT), например: udiOperationCycles;

• r — вещественное число одинарной точности (REAL), например: rPressureBar;

• lr — вещественное число двойной точности (LREAL), например: lrCoordinateMm;

• t — интервал времени (TIME), например: tDelayBeforeStart;

• s — текстовая строка (STRING), например: sRecipeName;

• a — массив данных (ARRAY), за которым следует префикс типа элементов, например: aiTemperatures;

• st — структура пользовательского типа (STRUCT), например: stMotorData;

• e — элемент перечислимого типа (ENUM), например: eCurrentState;

• fb — экземпляр функционального блока (FUNCTION_BLOCK), например: fbPurgeTimer.

Использование осмысленных имен с префиксами превращает код в самодокументированную техническую спецификацию, понятную любому сервисному инженеру при пусконаладке на объекте.

1.4. Разделители, синтаксическая пунктуация и оператор присваивания

Синтаксис Structured Text требует скрупулезного соблюдения правил пунктуации. Главным символом языка является точка с запятой ;. Она служит терминатором (разделителем) завершенных операторов. Любая исполняемая инструкция, присваивание, вызов функционального блока или системная

команда обязаны оканчиваться символом точки с запятой. Пропуск этого знака — наиболее распространенная ошибка начинающих разработчиков, приводящая к тому, что компилятор пытается интерпретировать следующую строку как продолжение предыдущего выражения, формируя сообщения об ошибках в совершенно посторонних местах кода.

При этом точка с запятой не ставится после ключевых слов заголовков блоков, таких как `THEN` в условных операторах, `DO` в операторах циклов или заголовка `CASE ... OF`. Установка точки с запятой сразу после `THEN` приводит к созданию пустого тела условия, что полностью искажает запланированную логику работы механизма.

Фундаментальное значение имеет четкое разграничение двух внешне похожих символов:

1. **Оператор присваивания `:=` (двоеточие со знаком равенства).** Это динамическое действие, команда процессору контроллера вычислить математическое или логическое выражение, расположенное справа от оператора, и немедленно скопировать полученный результат в ячейку памяти переменной, имя которой указано слева.

2. **Оператор отношения `=` (одиночный знак равенства).** Это логическая операция сравнения. Она не меняет содержимое ячеек памяти, а лишь отвечает на вопрос, равны ли операнды справа и слева. Результатом оператора сравнения всегда является булево значение `TRUE` или `FALSE`.

Попытка использовать одиночный знак равенства для изменения значения переменной (например, запись вида `xMotor = TRUE;`) не выполнит включение исполнительного механизма, а вызовет фатальную ошибку сборки проекта компилятором CODESYS.

Круглые скобки () используются для группировки операндов внутри сложных математических и логических выражений для явного переопределения стандартного приоритета операций, а также для передачи списка фактических параметров при вызове функций и функциональных блоков. Запятая , применяется для разделения аргументов при вызовах и перечислении элементов данных.

Двоеточие : (без знака равенства) используется исключительно в секции объявления данных для отделения имени переменной от ее типа данных, а также в метках оператора множественного выбора CASE.

1.5. Культура комментирования и организация промышленного кода

Промышленный программный код создается один раз, но читается, модифицируется и анализируется сотни раз на протяжении десятилетий эксплуатации технологического оборудования. Отсутствие пояснений или небрежное комментирование приводят к колоссальным простоям производственных линий при модернизации или устранении сбоев силами дежурного персонала предприятия.

Среда CODESYS v3.5 поддерживает два официальных формата комментариев:

1. **Однострочный комментарий (//).** Все символы от двойного слеша до физического конца строки полностью игнорируются транслятором компилятора. Данный формат удобен для пояснения назначения конкретной строки кода, физического смысла переменной или уставки.

2. **Блочный многострочный комментарий ((* ... *)).** Все содержимое, заключенное между открывающей скобкой со звездочкой и

закрывающей звездочкой со скобкой, воспринимается как технический комментарий, даже если оно занимает десятки строк. Данный синтаксис стандарта IEC 61131-3 предпочтителен для оформления вводных паспортных шапок программных блоков, а также для временного исключения крупных участков кода из исполнения при отладке (комментирование блоков).

Грамотно организованный промышленный POU обязан предваряться стандартной структурированной шапкой, содержащей наименование агрегата, технологический узел, автора разработки, версию алгоритма и журнал внесенных изменений. Каждая ветвь аварийных блокировок обязана содержать текстовое пояснение физической причины останова со ссылкой на схему автоматизации (P&ID).

ПРАКТИКУМ ГЛАВЫ 1

Ниже представлены пять выверенных инженерных примеров возрастающей сложности, адаптированных для работы в среде CODESYS v3.5 SP16/SP18/SP20 под управлением программного контроллера CODESYS Control Win V3.

Пример 1. Создание минимального рабочего проекта и базовая инициализация

Данный пример демонстрирует скелет программы управления дискретным клапаном с непрерывным тактовым подсчетом рабочих циклов задачи ПЛК.

Секция объявления переменных программы PLC_PRG:

```
PROGRAM PLC_PRG
VAR
  // Логический флаг разрешения работы механизма
  xSystemEnable      : BOOL := FALSE;

  // Команда на включение электромагнитного клапана
  xValveCommand      : BOOL := FALSE;

  // Внутренний счетчик циклов сканирования ПЛК
  udiCycleCounter    : UDINT := 0;

  // Технологическая уставка предельного давления, бар
  rPressureThreshold  : REAL := 6.5;
END_VAR
```

Исполняемый код программы:

```
// Шаг 1: Инкрементирование системного счетчика циклов задачи контроллера
// Данная инструкция выполняется один раз за каждый скан (Task Cycle Time)
udiCycleCounter := udiCycleCounter + 1;

// Шаг 2: Базовая технологическая проверка условий
// Клапан активируется только в том случае, если система переведена в рабочий режим
IF xSystemEnable THEN
  xValveCommand := TRUE; // Подача напряжения на катушку распределителя
```

```
ELSE
   xValveCommand := FALSE; // Обесточивание катушки и возврат клапана пружиной
END_IF;
```

Построчный разбор логики:

• Строки объявления переменных задают начальные значения через оператор :=. Переменная udiCycleCounter инициализируется нулем при старте контроллера.

• В первой строке исполняемого кода выражение udiCycleCounter + 1 вычисляет сумму текущего значения счетчика и единицы, после чего оператор := сохраняет новое число обратно в ту же переменную. Этот прием называется программным счетчиком сканов.

• Блок IF ... THEN ... ELSE ... END_IF анализирует состояние флага xSystemEnable. Если переменная истинна (TRUE), выполняется первая ветка. Если флаг сброшен (FALSE), принудительно отрабатывает секция ELSE, гарантируя отключение силового выхода.

Пример 2. Прямая физическая адресация дискретного ввода-вывода

В данном примере реализовано аппаратное связывание физических клемм контроллера с логическими инструкциями языка ST с использованием прямой стандартизированной адресации IEC.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
   // Физическая кнопка «Пуск» на клемме 0 входного модуля (нормально
```

```
разомкнутая)
  xBtnStart         AT %IX0.0 : BOOL;

  // Физическая кнопка «Стоп» на клемме 1 входного модуля (нормально замкнутая)
  xBtnStop          AT %IX0.1 : BOOL;

  // Контактор включения электродвигателя насоса на клемме 0 выходного модуля
  xPumpContactor     AT %QX0.0 : BOOL;

  // Светосигнальная лампа индикации аварии на клемме 1 выходного модуля
  xAlarmLamp        AT %QX0.1 : BOOL;

  // Внутреннее реле состояния (память самоподхвата схемы)
  xPumpRunningState   : BOOL := FALSE;
END_VAR
```

Исполняемый код программы:// Классическая схема управления пускателем с самоподхватом:

```
// Насос должен включиться по кнопке «Пуск» и продолжать работать после ее отпускания.
// Отключение происходит по размыканию физической кнопки «Стоп».

// Физическая кнопка «Стоп» с контактом NC при исправной цепи подает сигнал TRUE.
// Нажатие кнопки или обрыв провода приводит к появлению сигнала FALSE.
```

```
IF (xBtnStart OR xPumpRunningState) AND xBtnStop THEN
   xPumpRunningState := TRUE;  // Включение и удержание рабочего состояния
ELSE
   xPumpRunningState := FALSE; // Сброс состояния при нажатии Стоп или обрыве
END_IF;

// Передача рассчитанного состояния на физический контактор насоса
xPumpContactor := xPumpRunningState;

// Индикация: если контактор отключен, горит лампа готовности/останова
xAlarmLamp := NOT xPumpRunningState;
```

Построчный разбор логики:

- Директива `AT %IX0.0` жестко связывает переменную `xBtnStart` с нулевым битом нулевого байта области входов процесса.
- В условии `IF` оператор `OR` реализует параллельное соединение контактов: питание схемы поддерживается либо нажатой кнопкой `xBtnStart`, либо уже взведенным внутренним флагом `xPumpRunningState`.
- Оператор `AND xBtnStop` выступает главным защитным размыкателем: если кнопка «Стоп» нажата (цепь разомкнута, значение на входе равно `FALSE`), логическое произведение дает `FALSE`, обесточивая насос.

Пример 3. Символическая адресация и логика защитной блокировки гидростанции

Пример демонстрирует современный подход без использования прямых физических адресов %I/%Q, реализуя технологическую защиту гидравлического насоса по сигналам датчиков уровня и температуры масла.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Дискретный датчик минимального уровня масла в гидробаке (TRUE = норма)
  xOilLevelOk       : BOOL := TRUE;

  // Термореле предельной температуры масла (TRUE = норма, FALSE = перегрев)
  xOilTempOk        : BOOL := TRUE;

  // Входной флаг команды включения от технологического автомата
  xAutoRunCommand    : BOOL := FALSE;

  // Сигнал управления катушкой гидрораспределителя
  xHydraulicValveOn  : BOOL := FALSE;

  // Битовая маска блокировок для отправки на панель оператора HMI
  wInterlockStatus   : WORD := 16#0000;
END_VAR
```

Исполняемый код программы:

```
// Формирование комплексного защитного условия:
// Разрешение на включение формируется ТОЛЬКО при одновременной норме уровня и температуры
IF xOilLevelOk AND xOilTempOk THEN
  // Защиты в норме: гидрораспределитель повторяет технологическую команду
  xHydraulicValveOn := xAutoRunCommand;
  wInterlockStatus  := 16#0000; // Ошибок нет
ELSE
  // Нарушение условий безопасности: безусловное обесточивание гидросистемы
  xHydraulicValveOn := FALSE;

  // Формирование битовой маски причин аварийного останова
  IF NOT xOilLevelOk THEN
    wInterlockStatus := wInterlockStatus OR 16#0001; // Бит 0: Авария уровня масла
  END_IF;

  IF NOT xOilTempOk THEN
    wInterlockStatus := wInterlockStatus OR 16#0002; // Бит 1: Перегрев масла в баке
  END_IF;
END_IF;
```

Построчный разбор логики:

• Логическое выражение xOilLevelOk AND xOilTempOk выступает в роли сторожевого барьера. Если хотя бы один параметр вышел за пределы нормы, управление переходит в защитную секцию ELSE.

• Инструкция wInterlockStatus OR 16#0001 осуществляет побитовое сложение текущего статуса с шестнадцатеричной маской 16#0001, взводя нулевой бит диагностического слова без затирания остальных битов аварий.

Пример 4. Отработка и локализация типовых ошибок сборки проекта

В данном примере намеренно проанализированы ситуации, с которыми сталкивается каждый инженер на этапе первой компиляции кода в среде CODESYS.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
   xInputSensor      : BOOL;
   rTargetSpeed      : REAL;
   iCalculatedValue   : INT;
END_VAR
```

Исполняемый код с разбором типичных дефектов компиляции:

```
// ДЕФЕКТ 1: Ошибка компилятора C0018: «Недопустимый оператор присваивания»
// Ошибочная запись:  rTargetSpeed = 50.0;  (Одиночный знак равенства вместо :=)
// Корректная запись:
```

```
rTargetSpeed := 50.0;

// ДЕФЕКТ 2: Ошибка компилятора C0006: «Пропущена точка с запятой перед...»
// Ошибочная запись:
// iCalculatedValue := 100
// rTargetSpeed := 25.5;
// Корректная запись:
iCalculatedValue := 100;
rTargetSpeed := 25.5;

// ДЕФЕКТ 3: Ошибка компилятора C0032: «Невозможно неявно преобразовать тип REAL в INT»
// Ошибочная запись:  iCalculatedValue := rTargetSpeed;
// В стандарте IEC 61131-3 запрещено неявное присвоение чисел с плавающей точкой
// целым переменным без явного вызова функции преобразования типов:
iCalculatedValue := REAL_TO_INT(rTargetSpeed);
```

Построчный разбор логики:

- В дефекте 1 показана разница между синтаксисом языков C/C++ (где для присваивания служит знак равенства) и языком Structured Text стандарта IEC, требующим двоеточия со знаком равенства `:=`.
- В дефекте 3 подчеркнута строгость статической типизации: попытка прямой записи вещественного числа `25.5` в целочисленную ячейку памяти блокируется компилятором еще до загрузки программы в контроллер, исключая неконтролируемое усечение дробной части в работающей машине.

Пример 5. Онлайн-отладка: форсирование переменных и циклический мониторинг

Пример программы-стенда, предназначенной для освоения инструментов отладки CODESYS: таблиц наблюдения (Watch Lists), форсирования значений (Force Values) и точек останова (Breakpoints).

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Дискретные сигналы для эмуляции нажатия кнопок на экране отладки
  xManualButton1     : BOOL := FALSE;
  xManualButton2     : BOOL := FALSE;

  // Результат логических операций
  xLogicResultAND    : BOOL;
  xLogicResultOR     : BOOL;
  xLogicResultXOR    : BOOL;

  // Накопительный счетчик для контроля выполнения программы в онлайне
  udiLoopTicks       : UDINT := 0;
END_VAR
```

Исполняемый код программы:

```
// Непрерывный хронометраж каждого рабочего цикла контроллера
udiLoopTicks := udiLoopTicks + 1;
```

```
// Параллельное вычисление базовых булевых функций для одних и тех же входов:

// 1. Конъюнкция (Логическое И): истинно ТОЛЬКО при двух нажатых кнопках одновременно
xLogicResultAND := xManualButton1 AND xManualButton2;

// 2. Дизъюнкция (Логическое ИЛИ): истинно при нажатии ХОТЯ БЫ ОДНОЙ любой кнопки
xLogicResultOR  := xManualButton1 OR xManualButton2;

// 3. Строгая дизъюнкция (Исключающее ИЛИ / XOR):
// Истинно ТОЛЬКО тогда, когда нажата РОВНО ОДНА кнопка.
// Если обе кнопки отпущены или обе нажаты одновременно — выход строго равен FALSE!
xLogicResultXOR := xManualButton1 XOR xManualButton2;
```

Инженерная методика проведения отладки в среде CODESYS:

1. Запустите виртуальный контроллер через меню панели задач Windows: значок CODESYS Control Win V3 -> *Start PLC*.

2. В среде разработки выполните команду меню *Online* -> *Login* (Горячая клавиша Alt + F8). Среда выполнит трансляцию кода в машинные инструкции и загрузит проект в эмулятор.

3. Переведите виртуальный контроллер в режим циклического исполнения: команда *Debug* -> *Start* (F5). Переменная udiLoopTicks начнет непрерывно увеличивать свое значение, свидетельствуя о выполнении циклов задачи.

4. Выполните двойной щелчок левой кнопкой мыши в столбце *Prepared Value* напротив переменной xManualButton1. Значение переключится в TRUE. Нажмите комбинацию клавиш Ctrl + F7 (Write Values) для разовой записи значения в память процесса.

5. Проанализируйте значения выходов: вы увидите, что xLogicResultAND остался равен FALSE, а xLogicResultOR и xLogicResultXOR немедленно приняли значение TRUE.

6. Используйте команду меню *Debug* -> *Force Values* (F7) для жесткого форсирования значения. При форсировании системный планировщик принудительно перезаписывает ячейку памяти перед каждым сканом задачи, игнорируя любые физические сигналы с модулей ввода.

КОНТРОЛЬНЫЕ ВОПРОСЫ И УПРАЖНЕНИЯ К ГЛАВЕ 1

1. Опишите последовательность стадий одного рабочего цикла циклической задачи ПЛК. В какой момент этого цикла вычисленные логические состояния передаются на выходные клеммы контроллера?

2. Почему прямое присваивание значения переменной физического выхода %QX0.0 в нескольких местах программы считается опасной инженерной практикой?

3. В чем заключается фундаментальное различие между областями объявлений VAR, VAR_TEMP и VAR RETAIN? Приведите технологический пример переменной, требующей обязательного размещения в области RETAIN.

4. Дайте определение венгерской нотации. Какие префиксы используются для обозначения типов данных BOOL, REAL, DINT и TIME?

5. Объясните синтаксическую разницу между оператором присваивания := и оператором отношения =. К каким последствиям в поведении алгоритма приведет их ошибочная подмена?

6. Почему символ точки с запятой ; недопустим непосредственно после ключевого слова THEN в условных конструкциях?

7. Что произойдет с выполнением программы в контроллере, если программист организует в тексте на языке ST непрерывный пустой цикл ожидания внешнего датчика?

8. Чем блочный комментарий вида (* ... *) функционально отличается от строчного комментария //?

9. Что представляет собой режим форсирования переменных (Force Values) в CODESYS и почему его непреднамеренное использование на работающем промышленном объекте несет угрозу аварии?

10. Каким образом символическая адресация переменных обеспечивает аппаратную независимость технологического алгоритма от физической модели контроллера?

Практические упражнения к главе 1

Упражнение 1.1. Найдите и исправьте все синтаксические ошибки в представленном фрагменте кода программы управления освещением цеха:

```
PROGRAM PLC_PRG
VAR
  xSensor1 : BOOL
  1_PumpCommand : BOOL;
  rTargetPressure : REAL := 5.0
END_VAR
```

```
IF xSensor1 = TRUE; THEN
   1_PumpCommand = TRUE
ELSE
   1_PumpCommand := FALSE;
END_IF
```

Упражнение 1.2. Разработайте секцию объявления и алгоритм на языке ST для схемы управления конвейером с двух независимых постов. Конвейер должен включаться кнопкой «Пуск» с любого из двух постов и отключаться нажатием кнопки «Стоп» на любом из постов с соблюдением принципа самоподхвата.

Упражнение 1.3. Напишите математическое выражение на языке ST, вычисляющее значение переменной xResult : BOOL по правилу: результат истинен тогда и только тогда, когда из трех дискретных датчиков xSensorA, xSensorB, xSensorC замкнуты ровно два любых датчика одновременно. Запрещено использовать операторы IF и CASE.

ГЛАВА 2. ФУНДАМЕНТАЛЬНАЯ СИСТЕМА ТИПОВ ДАННЫХ

2.1. Логический тип данных (BOOL)

Фундаментом дискретной промышленной автоматизации выступает логический тип данных BOOL. Переменная этого типа способна принимать только одно из двух взаимоисключающих логических значений: TRUE (логическая единица, истина) или FALSE (логический ноль, ложь). С точки зрения физических цепей управления постоянным напряжением 24 В значение TRUE

обычно соответствует наличию потенциала на клемме контроллера (уровень высокого напряжения от 15 до 30 В), а FALSE — отсутствию потенциала или уровню земли (напряжение от -3 до +5 В).

Несмотря на то, что для хранения логического состояния математически достаточен один бит памяти, архитектура современных 32-битных и 64-битных микропроцессоров, на которых функционирует среда CODESYS v3.5, оптимизирована для работы с машинными словами, кратными байту. Вследствие этого одиночная скалярная переменная типа BOOL, объявленная в секции VAR, физически занимает в оперативной памяти (SRAM) контроллера целый байт (8 бит).

В памяти значение FALSE кодируется байтом 16#00, а значение TRUE чаще всего представляется числом 16#01 (в некоторых аппаратных реализациях — 16#FF). При чтении процессором любое ненулевое значение байта трактуется как логическая истина. Однако при размещении логических переменных внутри битовых массивов, структур со специальными прагмами упаковки или при маппинге каналов распределенного ввода-вывода среда разработки выполняет побитовое сжатие, размещая до 8 переменных BOOL в одном физическом байте памяти.

Логические литералы в коде на Structured Text записываются исключительно служебными словами TRUE и FALSE. Использование числовых констант 1 и 0 в прямых присваиваниях вида xValve := 1; в строгом режиме компиляции стандарта IEC 61131-3 вызывает ошибку несовместимости типов.

2.2. Битовые строки: BYTE, WORD, DWORD, LWORD

Битовые строки представляют собой нечисловые контейнеры фиксированной длины, предназначенные для хранения сырых последовательностей

нулей и единиц, битовых масок аварий, управляющих слов полевых шин (Control Word) и регистров статуса технологических приводов (Status Word). В отличие от целых чисел, битовые строки не имеют знака и к ним неприменимы стандартные арифметические законы сложения и вычитания без явного преобразования типов.

Стандарт определяет четыре типа битовых строк в зависимости от их физической разрядности:

- BYTE — однобайтовая битовая строка разрядностью 8 бит (диапазон адресов бит от .0 до .7).
- WORD — машинное слово разрядностью 16 бит (содержит 2 байта, биты от .0 до .15).
- DWORD — двойное слово разрядностью 32 бита (содержит 4 байта, биты от .0 до .31).
- LWORD — длинное слово разрядностью 64 бита (содержит 8 байт, биты от .0 до .63).

Для записи значений битовых строк в коде ST используются специализированные префиксы систем счисления:

- Двоичный формат предваряется префиксом 2#, например: wMask := 2#1010_0000_1111_0001;. Для повышения визуальной читаемости разряды допускается разделять одиночным символом подчеркивания.
- Шестнадцатеричный формат предваряется префиксом 16#, например: dwStatus := 16#FFFF_00A1;.
- Восьмеричный формат предваряется префиксом 8#, например: bOctal := 8#377;.

Среда CODESYS v3.5 предоставляет уникальный механизм побитового доступа к отдельным разрядам битовых строк через оператор точки: wStatusWord.0 возвращает значение младшего (нулевого) бита в виде булевой переменной, а wStatusWord.15 позволяет опросить или перезаписать старший бит слова.

2.3. Целочисленные знаковые и беззнаковые типы данных

Целочисленные типы данных служат для математических расчетов, индексации элементов массивов, подсчета квантов времени, отслеживания количества выпущенных заготовок и шагов технологических циклограмм. Система типов standards IEC разделена на два семейства: знаковые типы (использующие формат дополнительного кода для представления отрицательных величин) и беззнаковые типы (принимающие исключительно неотрицательные значения).

В знаковых типах старший бит числа выступает в роли знакового флага: значение 0 свидетельствует о положительном числе, а 1 — об отрицательном. Отрицательные значения формируются по схеме дополнительного кода: все биты прямого кода инвертируются, после чего к младшему разряду прибавляется единица. Это позволяет микропроцессору ПЛК выполнять операции вычитания на тех же самых сумматорах, которые используются для сложения.

Критически важным аспектом промышленного программирования является явление **аппаратного переполнения (Overflow)**. Если знаковая переменная типа INT достигла значения +32767 и программа пытается выполнить операцию iCounter := iCounter + 1;, в процессоре происходит перенос единицы в знаковый разряд. Значение переменной скачкообразно становится равным -32768. Если этот счетчик управлял позиционированием пильного диска или

приводом подъемника, система получит ложное отрицательное значение координаты, что может спровоцировать разрушительную механическую аварию.

Тип данных	Разрядность (бит)	Занимаемая память	Минимальное значение	Максимальное значение
SINT (Short Integer)	8	1 байт	-128	+127
USINT (Unsigned Short Int)	8	1 байт	0	255
INT (Integer)	16	2 байта	-32 768	+32 767
UINT (Unsigned Integer)	16	2 байта	0	65 535
DINT (Double Integer)	32	4 байта	-2 147 483 648	+2 147 483 647
UDINT (Unsigned Double Int)	32	4 байта	0	4 294 967 295
LINT (Long Integer)	64	8 байт	-2^{63}	$+2^{63}-1$
ULINT (Unsigned Long Int)	64	8 байт	0	$+2^{64}-1$

2.4. Вещественные числа с плавающей точкой (REAL, LREAL)

Для описания непрерывных технологических параметров (давление пара в коллекторе, температура расплава, массовый расход сыпучих сред, pH раствора, угол наклона стрелы манипулятора) целочисленной арифметики

недостаточно. Для представления действительных дробных чисел стандарт IEC 61131-3 опирается на международный стандарт IEEE 754.

В CODESYS v3.5 определены два типа с плавающей точкой:

1. REAL — одинарная точность, занимает в памяти 32 бита (4 байта). Обеспечивает диапазон абсолютных значений приблизительно от $\pm 1.175 \times 10^{-38}$ до $\pm 3.402 \times 10^{+38}$ с точностью 7–8 значащих десятичных цифр.

2. LREAL — двойная точность, занимает в памяти 64 бита (8 байт). Обеспечивает диапазон от $\pm 2.225 \times 10^{-308}$ до $\pm 1.797 \times 10^{+308}$ с точностью 15–17 значащих десятичных цифр.

Структура 32-битного числа REAL в памяти контроллера строго структурирована:

- 1 бит (31-й) — знак числа (0 — положительное, 1 — отрицательное);
- 8 бит (с 30-го по 23-й) — смещенная экспонента (порядок основания 2);
- 23 бита (с 22-го по 0-й) — нормализованная мантисса (дробная часть).

Из дискретной природы хранения мантиссы вытекает фундаментальное правило: **вещественные числа стандарта IEEE 754 хранятся в контроллере приближенно**. Число 0.1 физически невозможно точно представить в виде конечной суммы отрицательных степеней двойки. Накопление погрешностей при непрерывном суммировании малых величин к большим числам способно полностью остановить работу алгоритма.

Кроме того, математический сопроцессор ПЛК формирует специальные нечисловые состояния при грубых вычислительных сбоях:

- NaN (Not a Number — не число): возникает в результате математически неопределенных операций (деление нуля на ноль, извлечение квадратного

корня из отрицательного числа). Любая последующая операция с NaN возвращает NaN, что приводит к заражению всей цепочки вычислений.

• +Infinity и -Infinity (положительная и отрицательная бесконечность): возникают при делении ненулевого вещественного числа на абсолютный математический ноль 0.0.

2.5. Символьные строки (STRING, WSTRING)

Символьные строки предназначены для обмена диагностическими текстовыми сообщениями со SCADA-системами, формирования аварийных журналов, вывода служебных подсказок оператору на экран HMI и передачи параметрических команд по последовательным интерфейсам.

Стандарт разделяет строки на два класса:

• STRING — строка, символы которой кодируются однобайтовой таблицей ASCII или локальными кодовыми страницами (в русскоязычных проектах для CODESYS традиционно используется страница Windows-1251). Каждый символ занимает 1 байт.

• WSTRING — расширенная строка (Wide String), в которой каждый символ кодируется двумя байтами по стандарту Unicode (UTF-16). Позволяет одновременно выводить символы любых национальных алфавитов и специализированные пиктограммы.

По умолчанию в CODESYS переменная типа STRING без явного указания размера вмещает до 80 символов. Для оптимизации оперативной памяти разработчик обязан явно ограничивать размер строки в скобках при объявлении: sSensorName : STRING(15);.

Важнейшая деталь реализации: **строка в памяти ПЛК всегда занимает на один байт больше, чем объявленная емкость символов**. Этот завершающий невидимый байт содержит нулевой символ 16#00 (Null-terminator). Именно по нулевому байту строковые функции определяют фактический конец текста в памяти процесса. Если объявлена переменная STRING(10), компилятор выделит под нее 11 байт памяти. Для строк WSTRING терминатором служит двухбайтовое нулевое слово 16#0000.

Литералы типа STRING обрамляются одиночными прямыми кавычками: 'Conveyor Motor 1'. Литералы типа WSTRING обязаны обрамляться двойными кавычками: "Линия фасовки".

2.6. Типы физического и календарного времени

Время — фундаментальная координата промышленного процесса. Стандарт IEC 61131-3 полностью изолирует разработчика от необходимости вручную вычислять тики аппаратных таймеров через систему строгих временных типов.

Интервальные типы (TIME, LTIME)

Служат для задания продолжительности технологических операций, пауз, фильтрации сигналов и уставок сторожевых таймеров:

• TIME — 32-битное целое число миллисекунд. Диапазон: от T#0ms до T#49d17h2m47s295ms. Литерал формируется префиксом T# или TIME# с последующим указанием дней (d), часов (h), минут (m), секунд (s) и миллисекунд (ms), например: tFillDelay : TIME := T#2m30s;.

• LTIME — 64-битное целое число наносекунд. Диапазон составляет сотни лет, дискретность — 1 нс. Префикс литералов LTIME#, например: ltPulseWidth : LTIME := LTIME#100us250ns;.

Календарные типы

Служат для синхронизации работы технологического оборудования со сменами, суточными графиками и расписаниями:

• DATE — календарная дата. Хранит количество секунд, прошедших с полуночи 1 января 1970 года (эпоха UNIX). Литерал: D#2026-09-24 или DATE#2026-09-24.

• TIME_OF_DAY (сокращенно TOD) — астрономическое время суток с миллисекундной точностью, отсчитываемое от полуночи 00:00:00.000. Литерал: TOD#14:30:15.500.

• DATE_AND_TIME (сокращенно DT) — комбинированная метка абсолютного времени и даты. Литерал: DT#2026-09-24-14:30:15.

2.7. Преобразование типов (Type Casting)

В стандарте IEC 61131-3 действует строгая типизация (Strong Typing). Это означает, что компилятор запрещает смешивать в одном выражении операнды различных типов без явного указания того, как именно память одного типа должна быть интерпретирована в терминах другого типа. Попытка сложить переменную INT с переменной REAL напрямую вызовет ошибку компиляции.

Преобразования типов делятся на:

1. **Неявные расширяющие преобразования (Implicit Conversion).** Допускаются компилятором автоматически только в тех случаях, когда гарантированно отсутствует риск потери данных или знака. Например, переменная типа INT (16 бит) может быть неявно передана в выражение, ожидающее DINT (32 бита).

2. **Явные преобразования типов (Explicit Conversion).** Требуют вызова специализированных операторов приведения. Имя оператора строится по общему шаблону: <ИсходныйТип>_TO_<ЦелевойТип>, либо используется универсальный перегруженный оператор TO_<ЦелевойТип>.

Примеры явных операторов преобразования:

- REAL_TO_INT(rPressure) — округляет вещественное число по правилам математики до ближайшего целого. Если дробная часть равна 0.5, большинство платформ стандарта IEC выполняют округление к ближайшему четному числу (банковское округление).

- INT_TO_REAL(iRawADC) — преобразует целое число в число с плавающей точкой без потери точности.

- TIME_TO_UDINT(tCycleDelay) — возвращает физическое количество миллисекунд интервала в виде стандартного целого числа, пригодного для передачи по сети Modbus в SCADA.

- UDINT_TO_TIME(udiMillis) — восстанавливает тип интервала времени из сырого числа миллисекунд.

- BOOL_TO_INT(xAlarm) — возвращает 1, если флаг равен TRUE, и 0, если флаг равен FALSE.

ПРАКТИКУМ ГЛАВЫ 2

Пример 6. Демонстрация аппаратного переполнения целочисленных типов

В данном примере наглядно демонстрируется разрушительный эффект переполнения диапазона знакового типа `INT` и беззнакового типа `WORD`, а также реализация программного детектора переполнения.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Знаковый счетчик деталей, работающий на пределе диапазона
  iBatchCounter      : INT := 32766;

  // Битовый счетчик тактов
  wBinaryCounter     : WORD := 16#FFFF;

  // Флаг принудительного инкремента с панели отладки
  xTriggerIncrement  : BOOL := FALSE;

  // Сигнал аварийного переполнения счетчика
  xOverflowDetected  : BOOL := FALSE;

  // Сохраненное значение до шага инкремента
  iPreviousValue     : INT;
  xPrevTrigger       : BOOL;
END_VAR
```

Исполняемый код программы:

```
// Отслеживание переднего фронта сигнала инкремента
IF xTriggerIncrement AND NOT xPrevTrigger THEN
  // Сохраняем предыдущее состояние для анализа аномалий
  iPreviousValue := iBatchCounter;

  // ВЫПОЛНЕНИЕ ОПЕРАЦИИ, ВЫЗЫВАЮЩЕЙ ПЕРЕПОЛНЕНИЕ:
  // При значении iBatchCounter = 32767 прибавление единицы
  // вызовет перенос бита в знаковый разряд: результат станет равен -32768!
  iBatchCounter := iBatchCounter + 1;

  // Битовое слово wBinaryCounter при переполнении 16#FFFF сбрасывается в 16#0000:
  wBinaryCounter := wBinaryCounter + 1;

  // АЛГОРИТМИЧЕСКИЙ ДЕТЕКТОР ПЕРЕПОЛНЕНИЯ:
  // Если после прибавления положительного числа результат стал МЕНЬШЕ исходного,
  // значит произошло аппаратное переполнение разрядной сетки процессора.
  IF iBatchCounter < iPreviousValue THEN
    xOverflowDetected := TRUE; // Активация аварийной блокировки
  ELSE
    xOverflowDetected := FALSE;
  END_IF;
END_IF;
```

```
xPrevTrigger := xTriggerIncrement;
```

Построчный разбор логики:

- В объявлении переменная `iBatchCounter` инициализируется числом 32766, что на 1 единицу меньше абсолютного математического максимума знакового двухбайтового целого числа (+32767).

- В строке `iBatchCounter := iBatchCounter + 1;` при повторном нажатии триггера происходит переполнение. Компилятор не генерирует системной ошибки в процессе выполнения, процессор контроллера продолжает работу, однако переменная принимает значение `-32768`.

- Блок проверки `IF iBatchCounter < iPreviousValue` демонстрирует инженерный прием контроля переполнения: в замкнутом кольце сложения инкремент положительного значения не может уменьшить итоговую величину, если не было переполнения.

Пример 7. Прецизионное сравнение вещественных чисел с защитой от погрешности

Попытка прямого сравнения двух переменных типа `REAL` через оператор равенства `=` является одной из самых частых причин скрытых отказов технологического оборудования. Пример реализует сравнение с машинным допуском (эпсилон).

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Текущее физическое давление в магистрали, бар
```

```
    rActualPressure     : REAL := 0.0;

    // Целевая технологическая уставка поддержания давления, бар
    rTargetPressure     : REAL := 3.0;

    // Машинный эпсилон (допустимая зона совпадения)
    rEpsilon            : REAL := 0.0001;

    // Ошибочный флаг прямого сравнения (ненадежный)
    xDirectEqualResult  : BOOL;

    // Надежный флаг прецизионного сравнения через допуск
    xSafeEqualResult    : BOOL;

    // Вспомогательный счетчик цикла суммирования
    iStepIndex          : INT;
END_VAR
```

Исполняемый код программы:

```
// Имитация накопления давления ступенями по 0.1 бар:
// В математике 30 шагов по 0.1 дают ровно 3.0.
// В двоичной арифметике IEEE 754 число 0.1 не имеет точного представления!
rActualPressure := 0.0;
FOR iStepIndex := 1 TO 30 DO
    rActualPressure := rActualPressure + 0.1;
END_FOR;
```

```
// ОШИБОЧНЫЙ ПОДХОД: Прямое сравнение на строгое равенство
// Переменная rActualPressure после цикла будет содержать число 3.00000024...
// В результате операция сравнения вернет FALSE! Клапан не откроется!
xDirectEqualResult := (rActualPressure = rTargetPressure);

// ИНЖЕНЕРНЫЙ ПРАВИЛЬНЫЙ ПОДХОД: Сравнение через модуль разности (ABS)
// Проверяется условие: |rActualPressure - rTargetPressure| < rEpsilon
IF ABS(rActualPressure - rTargetPressure) < rEpsilon THEN
   xSafeEqualResult := TRUE;  // Давление стабильно достигло уставки
ELSE
   xSafeEqualResult := FALSE; // Давление еще не набрано
END_IF;
```

Построчный разбор логики:

- Цикл `FOR` складывает число `0.1` тридцать раз. Накопленная погрешность округления мантиссы приводит к тому, что результат в памяти процессора не равен абсолютному числу `3.0`.
- Инструкция `xDirectEqualResult := (rActualPressure = rTargetPressure);` формирует `FALSE`. Автоматика, написанная новичком, зависает в ожидании уставки.
- Функция `ABS(rActualPressure - rTargetPressure)` вычисляет абсолютную разность между фактом и уставкой. Сравнение этой разности с величиной `rEpsilon` (`0.0001`) надежно фиксирует попадание процесса в целевую точку.

Пример 8. Извлечение компонент времени из типа TIME

Пример демонстрирует разложение технологического интервала TIME на компоненты часов, минут, секунд и миллисекунд для последующего вывода на панель оператора через операторы явного приведения типов.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Полный интервал времени рецептурного цикла
  tRecipeDuration    : TIME := T#14h25m18s750ms;

  // Сырое количество миллисекунд интервала
  udiTotalMilliseconds: UDINT;

  // Выделенные компоненты времени
  uiHours          : UINT;
  uiMinutes        : UINT;
  uiSeconds        : UINT;
  uiMilliseconds   : UINT;
END_VAR
```

Исполняемый код программы:

```
// Шаг 1: Явное преобразование абстрактного типа TIME в скалярное число миллисекунд UDINT
udiTotalMilliseconds := TIME_TO_UDINT(tRecipeDuration);

// Шаг 2: Извлечение миллисекунд (остаток от деления на 1000)
uiMilliseconds := TO_UINT(udiTotalMilliseconds MOD 1000);
```

```
// Шаг 3: Перевод суммарного времени в целые секунды
// Промежуточная переменная используется неявно через целочисленное деление
udiTotalMilliseconds := udiTotalMilliseconds / 1000;

// Шаг 4: Извлечение секунд текущей минуты (остаток от деления на 60)
uiSeconds := TO_UINT(udiTotalMilliseconds MOD 60);

// Шаг 5: Перевод в целые минуты
udiTotalMilliseconds := udiTotalMilliseconds / 60;

// Шаг 6: Извлечение минут текущего часа (остаток от деления на 60)
uiMinutes := TO_UINT(udiTotalMilliseconds MOD 60);

// Шаг 7: Извлечение полных часов (деление оставшихся минут на 60)
uiHours := TO_UINT(udiTotalMilliseconds / 60);
```

Построчный разбор логики:

• Оператор `TIME_TO_UDINT(tRecipeDuration)` открывает доступ к физическому содержимому 32-битного таймера контроллера.

• Последовательное применение операторов деления `/` и взятия остатка `MOD` реализует классический каскад декомпозиции системы счисления времени.

• Явный оператор `TO_UINT(...)` гарантирует безопасное сужение диапазона: промежуточные остатки от деления на 60 и 1000 гарантированно укладываются в 16-битный формат `UINT`.

Пример 9. Формирование строкового сообщения журнала событий

Пример иллюстрирует сборку текстовой аварийной строки с динамической подстановкой идентификатора датчика и измеренной температуры с использованием стандартных библиотечных функций работы со строками.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Идентификатор неисправного термопреобразователя
  iSensorID         : INT := 4;

  // Измеренная температура перегрева
  rOvertempValue    : REAL := 125.6;

  // Итоговая текстовая строка для передачи в SCADA
  sAlarmMessage     : STRING(80);

  // Промежуточные строковые буферы
  sSensorIdStr      : STRING(10);
  sTemperatureStr   : STRING(16);
END_VAR
```

Исполняемый код программы:

```
// Шаг 1: Преобразование целочисленного номера датчика в строковый формат
sSensorIdStr := INT_TO_STRING(iSensorID);

// Шаг 2: Преобразование вещественного числа температуры в текст
```

```
sTemperatureStr := REAL_TO_STRING(rOvertempValue);

// Шаг 3: Пошаговая сборка сообщения с использованием стандартного оператора CONCAT
// Сборка первой части: «АВАРИЯ: Датчик № [ID]»
sAlarmMessage := CONCAT('АВАРИЯ: Датчик № ', sSensorIdStr);

// Добавление разделителя
sAlarmMessage := CONCAT(sAlarmMessage, ', T = ');

// Добавление числового значения температуры
sAlarmMessage := CONCAT(sAlarmMessage, sTemperatureStr);

// Добавление единиц измерения
sAlarmMessage := CONCAT(sAlarmMessage, ' град C');
```

Построчный разбор логики:

• Операторы `INT_TO_STRING` и `REAL_TO_STRING` выполняют форматирование бинарных чисел в последовательность ASCII-символов.

• Стандартная функция `CONCAT(STR1, STR2)` принимает две входные строки, сцепляет их в единую строку и завершает результат нулевым байтом.

• Итоговая строка `sAlarmMessage` содержит читаемое сервисное сообщение: `АВАРИЯ: Датчик № 4, T = 125.6 град C`, полностью готовое к отправке на панель оператора.

Пример 10. Упаковка и распаковка битов: работа со словом состояния

Пример демонстрирует низкоуровневую работу с дискретными флагами: объединение 8 независимых сигналов датчиков в единый коммуникационный байт BYTE и обратное извлечение дискретных сигналов.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Группа физических дискретных датчиков защит
  xSensorEmergencyStop   : BOOL := TRUE;  // Норма (NC контакт)
  xSensorAirPressureOk   : BOOL := TRUE;  // Давление воздуха в норме
  xSensorThermalRelayOk  : BOOL := FALSE; // АВАРИЯ: сработало тепловое реле
  xSensorGuardDoorClosed : BOOL := TRUE;  // Защитная дверь закрыта

  // Результирующий байт для передачи в сеть Modbus
  bPackedStatusByte      : BYTE := 16#00;

  // Принятый по сети байт команд управления
  bReceivedCommandByte   : BYTE := 16#05; // Двоичный код: 2#0000_0101

  // Распакованные индивидуальные команды привода
  xCmdStartDrive         : BOOL;
  xCmdResetFault         : BOOL;
  xCmdEnableHeater       : BOOL;
END_VAR
```

Исполняемый код программы:

```
// ЧАСТЬ 1: УПАКОВКА НЕЗАВИСИМЫХ БИТОВ В ЕДИНЫЙ БАЙТ ЧЕРЕЗ ТОЧЕЧНУЮ НОТАЦИЮ
// Очищаем байт перед упаковкой
bPackedStatusByte := 16#00;

// Прямая запись булевых переменных в отдельные биты байта:
bPackedStatusByte.0 := xSensorEmergencyStop;   // Бит 0: E-Stop
bPackedStatusByte.1 := xSensorAirPressureOk;   // Бит 1: Давление воздуха
bPackedStatusByte.2 := xSensorThermalRelayOk;  // Бит 2: Тепловое реле
bPackedStatusByte.3 := xSensorGuardDoorClosed; // Бит 3: Ограждение

// Биты с 4 по 7 остаются равными нулю (резерв)

// ЧАСТЬ 2: РАСПАКОВКА БАЙТА КОМАНД НА НЕЗАВИСИМЫЕ БУЛЕВЫЕ ФЛАГИ
// Извлечение бита 0: команда «Пуск привода»
xCmdStartDrive := bReceivedCommandByte.0;

// Извлечение бита 1: команда «Сброс аварии»
xCmdResetFault := bReceivedCommandByte.1;

// Извлечение бита 2: команда «Включение нагревателя»
xCmdEnableHeater := bReceivedCommandByte.2;
```

Построчный разбор логики:

• Точечная адресация вида bPackedStatusByte.0 предоставляет быстрый интерфейс модификации отдельных битов переменной без использования сложных математических операций умножения и сдвигов.

• Распаковка xCmdStartDrive := bReceivedCommandByte.0; напрямую транслирует логическое состояние конкретного разряда сетевого регистра в алгоритмическую переменную программы на ST.

Пример 11. Упаковка 16 флагов состояний в единое коммуникационное слово WORD

В промышленных сетях Modbus TCP / RTU базовой единицей обмена выступает 16-битный регистр хранения (Holding Register). Данный пример показывает сборку регистра аварий агрегата с использованием побитовых логических сдвигов и масок.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Массив из 16 булевых сигналов технологических аварий
  axAlarmFlags      : ARRAY[0..15] OF BOOL;

  // Итоговое 16-битное коммуникационное слово аварий для SCADA
  wAlarmWord        : WORD := 16#0000;

  // Индекс цикла
  iBitIndex         : INT;
```

```
END_VAR
```

Исполняемый код программы:

```
// Имитация активности некоторых аварийных каналов:
axAlarmFlags[0]  := TRUE;  // Авария 0: Обрыв датчика давления
axAlarmFlags[4]  := TRUE;  // Авария 4: Превышение температуры шпинделя
axAlarmFlags[15] := TRUE;  // Авария 15: Главный контактор не включился

// СБОРКА СЛОВА АВАРИЙ С ИСПОЛЬЗОВАНИЕМ БИТОВОЙ МАТЕМАТИКИ:
wAlarmWord := 16#0000; // Сброс слова перед упаковкой

FOR iBitIndex := 0 TO 15 DO
  IF axAlarmFlags[iBitIndex] THEN
    // Если флаг активен, выставляем соответствующий бит через операцию побитового
    // логического сдвига единицы влево (SHL) и логического сложения (OR):
    wAlarmWord := wAlarmWord OR ROL(WORD#16#0001, iBitIndex);
  END_IF;
END_FOR;
```

Построчный разбор логики:

• Инструкция `WORD#16#0001` формирует типизированную шестнадцатеричную константу слова с установленным нулевым битом (`2#0000_0000_0000_0001`).

• Функция `ROL(константа, iBitIndex)` выполняет циклический сдвиг этого бита на позицию, соответствующую текущему индексу цикла.

• Оператор wAlarmWord OR ... объединяет сдвинутый бит с текущим значением слова, взводя нужный разряд и гарантируя неприкосновенность ранее записанных бит.

Пример 12. Линейное масштабирование сигнала датчика без потери точности

Аналоговые модули ввода контроллера преобразуют ток 4–20 мА в целочисленные отсчеты АЦП (например, диапазон от 0 до 27648). Пример реализует надежное масштабирование в физические единицы давления (от 0.0 до 16.0 бар) с превентивной защитой от потери точности.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Сырое значение из аналогового модуля ввода (%IW)
  iRawAdcInput      : INT := 13824; // Середина диапазона (соответствует 12 мА)

  // Технологический диапазон датчика давления, бар
  rPressureMinScale  : REAL := 0.0;
  rPressureMaxScale  : REAL := 16.0;

  // Границы сырого диапазона АЦП модуля
  iAdcMinLimit      : INT := 0;
  iAdcMaxLimit      : INT := 27648;

  // Выходное инженерное значение давления, бар
  rEngineeredPressure : REAL;
```

```
  // Флаг недостоверности сигнала датчика (обрыв токовой петли)
  xSensorFault       : BOOL;
END_VAR
```

Исполняемый код программы:

```
// ШАГ 1: ВАЛИДАЦИЯ ДИАПАЗОНА (Контроль обрыва и короткого замыкания цепи 4-20 мА)
// Если отсчеты АЦП падают ниже нуля более чем на 5% (отрицательный ток) — зафиксирован обрыв:
IF (iRawAdcInput < -1382) OR (iRawAdcInput > 29030) THEN
  xSensorFault := TRUE;
  rEngineeredPressure := 0.0; // Безопасное аварийное значение
ELSE
  xSensorFault := FALSE;

  // ШАГ 2: ПРИВЕДЕНИЕ ТИПОВ ДО ВЫПОЛНЕНИЯ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ
  // Грубая ошибка новичков: вычисление (iRawAdcInput / 27648) в целых числах.
  // При целочисленном делении результат всегда будет равен 0!
  // Необходимо принудительно перевести сырое значение в формат REAL:

  rEngineeredPressure := (TO_REAL(iRawAdcInput - iAdcMinLimit) /
                TO_REAL(iAdcMaxLimit - iAdcMinLimit)) *
                (rPressureMaxScale - rPressureMinScale) +
                rPressureMinScale;
```

```
  // ШАГ 3: ОГРАНИЧЕНИЕ ВЫХОДНОГО СИГНАЛА (CLAMPING)
  IF rEngineeredPressure > rPressureMaxScale THEN
    rEngineeredPressure := rPressureMaxScale;
  ELSIF rEngineeredPressure < rPressureMinScale THEN
    rEngineeredPressure := rPressureMinScale;
  END_IF;
END_IF;
```

Построчный разбор логики:

• Проверка `IF (iRawAdcInput < -1382)` отсекает аварийное состояние измерительного канала до начала расчетов.

• Вызов `TO_REAL(iRawAdcInput - iAdcMinLimit)` гарантирует, что деление будет выполняться в вещественной арифметике с плавающей точкой. Значение `13824 / 27648.0` дает ровно `0.5`.

• Умножение на размах шкалы `(16.0 - 0.0)` возвращает физическое давление `8.0` бар.

• Блок отсечки (Clamping) исключает появление технологически невозможных значений при кратковременных выбросах помех на входе модуля.

КОНТРОЛЬНЫЕ ВОПРОСЫ И УПРАЖНЕНИЯ К ГЛАВЕ 2

1. Сколько бит памяти занимает одиночная скалярная переменная типа `BOOL`, объявленная внутри программного блока `PROGRAM` на 32-битной платформе CODESYS Control Win V3, и почему?

2. В чем заключается физическое различие между типами данных `WORD` и `UINT`, если оба они занимают ровно 16 бит оперативной памяти?

3. Опишите математический механизм формирования отрицательного числа в формате дополнительного кода для типа SINT на примере числа -1.

4. Какое числовое значение примет переменная типа USINT (диапазон 0..255), если к ее значению 255 прибавить единицу в среде CODESYS v3.5?

5. Почему оператор строгого сравнения rValue1 = rValue2 признан некорректным для вещественных чисел типа REAL? Каким алгоритмом следует производить верификацию равенства аналоговых параметров?

6. Что означают нечисловые состояния NaN и +Infinity стандарта IEEE 754 и к каким последствиям приводит их попадание в контур аналогового регулирования?

7. Чем кодировка строки типа STRING отличается от кодировки типа WSTRING? Какое количество байт займет в памяти контроллера переменная, объявленная как sModel : STRING(20);?

8. Назовите предельное время, которое способен зафиксировать таймерный тип данных TIME. Что произойдет с системным хронометражем при переполнении этого диапазона?

9. Объясните опасность операции целочисленного деления при масштабировании аналоговых сигналов датчиков вида iResult := (iADC / 27648) * 100;.

10. Для чего в стандарте IEC 61131-3 используется двухэтапное преобразование типов данных и в каких случаях допустимо неявное приведение?

Практические упражнения к главе 2

Упражнение 2.1. Рассчитайте суммарный объем памяти в байтах, который займет следующая группа переменных в оперативной памяти контроллера с учетом стандартного выравнивания по границам слов:

```
VAR
  xInterlock1    : BOOL;
  wStatusRegister : WORD;
  rPressureSensor : REAL;
  bErrorCode     : BYTE;
  udiTotalCycles  : UDINT;
END_VAR
```

Упражнение 2.2. Напишите фрагмент кода на языке ST, который извлекает шестнадцатеричный байт старшего разряда (High Byte) и байт младшего разряда (Low Byte) из 16-битного коммуникационного слова `wInputWord : WORD` и сохраняет их в переменные `bHighByte : BYTE` и `bLowByte : BYTE` без использования указателей.

Упражнение 2.3. Разработайте алгоритмическую функцию на Structured Text, принимающую на вход целочисленный параметр `udiSeconds : UDINT` (количество секунд работы конвейера) и формирующую результирующую переменную типа `TIME`, пригодную для прямой передачи на вход уставки стандартного таймера `TON`.

ГЛАВА 3. ОПЕРАТОРЫ ПРИСВАИВАНИЯ, АРИФМЕТИКА И БИТОВАЯ ЛОГИКА

3.1. Оператор присваивания и семантика копирования значений

Оператор присваивания `:=` (двоеточие со знаком равенства) является фундаментальным исполнительным механизмом языка Structured Text, который инициирует перемещение данных между ячейками памяти процесса. При

выполнении инструкции вида rTargetSpeed := rMeasuredSpeed; процессор контроллера выполняет две последовательные операции: сначала вычисляет значение выражения, расположенного справа от оператора (Right-Hand Side, RHS), а затем копирует полученный битовый паттерн в область памяти, зарезервированную под переменную, имя которой указано слева (Left-Hand Side, LHS).

Важно понимать, что оператор присваивания в стандарте IEC 61131-3 работает по принципу копирования значения (Pass by Value). Это означает, что после выполнения присваивания переменные rTargetSpeed и rMeasuredSpeed становятся абсолютно независимыми друг от друга. Любое последующее изменение значения переменной rMeasuredSpeed в других частях программы не окажет никакого влияния на уже присвоенное значение rTargetSpeed.

Порядок вычисления правой части оператора строго регламентирован: компилятор сначала полностью вычисляет все подвыражения, выполняет все необходимые преобразования типов и только после получения окончательного результата выполняет запись в левый операнд. В случае присваивания сложных структур данных или массивов компилятор генерирует машинный код, который физически копирует каждый байт исходной области памяти в целевую область. Это накладывает ограничения на производительность при работе с крупными массивами данных в высокоскоростных задачах с циклом сканирования менее 1 миллисекунды.

3.2. Арифметические операторы и их особенности в ПЛК

Арифметические операторы языка Structured Text позволяют выполнять базовые математические преобразования технологических параметров, расчеты дозировок, скорости подачи, температурных градиентов и других

физических величин. В стандарте IEC 61131-3 определены следующие арифметические операторы:

• **Сложение +** — бинарный оператор, складывающий значения двух операндов. Результат принимает тип данных более широкого диапазона из двух операндов.

• **Вычитание -** — бинарный оператор, вычитающий значение правого операнда из левого.

• **Умножение *** — бинарный оператор, перемножающий значения операндов. При умножении двух целых чисел результат также будет целочисленным, что может привести к потере дробной части.

• **Деление /** — бинарный оператор деления. Критически важно: если оба операнда являются целочисленными, выполняется целочисленное деление с отбрасыванием дробной части.

• **Взятие остатка MOD** — бинарный оператор, возвращающий остаток от целочисленного деления. Применяется для циклического сброса счетчиков, выделения разрядов чисел и реализации кольцевых буферов.

Особенностью арифметики в программируемых контроллерах является отсутствие автоматической обработки переполнений на уровне процессорных исключений. При выполнении операции `iCounter := iCounter + 1;` когда переменная `iCounter` уже содержит максимальное значение своего типа, происходит аппаратное переполнение с потерей знакового бита. Контроллер не генерирует аварийного события, а продолжает выполнение программы с искаженными данными, что может привести к технологической аварии.

3.3. Логические (булевы) операторы

Логические операторы оперируют только булевыми значениями TRUE и FALSE и формируют булевый результат. Эти операторы составляют основу алгоритмов дискретной автоматики, защитных блокировок и комбинаторных схем управления.

• **Логическое И AND** — бинарный оператор, возвращающий TRUE только в том случае, когда оба операнда истинны. В таблице истинности: TRUE AND TRUE = TRUE, все остальные комбинации дают FALSE.

• **Логическое ИЛИ OR** — бинарный оператор, возвращающий TRUE если хотя бы один из операндов истинен. Ложным результат будет только при условии FALSE OR FALSE = FALSE.

• **Логическое НЕ NOT** — унарный оператор инверсии, меняющий значение на противоположное: NOT TRUE = FALSE и NOT FALSE = TRUE.

• **Исключающее ИЛИ XOR** — бинарный оператор, возвращающий TRUE только тогда, когда операнды имеют различные значения. Если оба операнда одинаковы (оба TRUE или оба FALSE), результат равен FALSE.

В CODESYS v3.5 логические операторы могут применяться не только к булевым переменным, но и к битовым строкам (BYTE, WORD, DWORD). В этом случае операция выполняется побитово над соответствующими разрядами операндов.

3.4. Операторы отношения и сравнения

Операторы отношения сравнивают два операнда и возвращают булев результат, отвечающий на вопрос о соотношении их величин. Эти операторы используются в условиях ветвлений, циклов и защитных алгоритмов.

• **Равенство** `=` — возвращает `TRUE`, если значения операндов идентичны.

• **Неравенство** `<>` — возвращает `TRUE`, если значения операндов различны.

• **Меньше** `<` — возвращает `TRUE`, если левый операнд строго меньше правого.

• **Больше** `>` — возвращает `TRUE`, если левый операнд строго больше правого.

• **Меньше или равно** `<=` — возвращает `TRUE`, если левый операнд меньше или равен правому.

• **Больше или равно** `>=` — возвращает `TRUE`, если левый операнд больше или равен правому.

Важное ограничение: операторы сравнения `<`, `>`, `<=`, `>=` неприменимы к булевым переменным и битовым строкам. Они работают только с упорядоченными типами данных: целыми и вещественными числами, временем, датами и строками.

3.5. Побитовые сдвиги и вращения

Операторы побитовых сдвигов и вращений выполняют манипуляции с разрядами битовых строк и целочисленных переменных. Эти операции широко используются для работы с коммуникационными протоколами, распаковки телеграмм полевых шин, реализации быстрых математических операций умножения и деления на степени двойки, а также для формирования управляющих масок.

• **Логический сдвиг влево SHL** — сдвигает все биты операнда влево на указанное количество позиций. Освободившиеся младшие биты заполняются нулями. Старшие биты, выходящие за разрядную сетку, безвозвратно теряются.

• **Логический сдвиг вправо SHR** — сдвигает все биты операнда вправо на указанное количество позиций. Освободившиеся старшие биты заполняются нулями. Младшие биты, выходящие за разрядную сетку, теряются.

• **Циклический сдвиг влево ROL** — сдвигает биты влево, при этом старшие биты, выходящие за разрядную сетку, заносятся в младшие биты результата.

• **Циклический сдвиг вправо ROR** — сдвигает биты вправо, при этом младшие биты, выходящие за разрядную сетку, заносятся в старшие биты результата.

В среде CODESYS v3.5 эти операции реализованы в виде стандартных функций, принимающих два параметра: сдвигаемое значение и количество позиций сдвига. Например: `wResult := SHL(wInput, 3);` сдвинет содержимое слова `wInput` на 3 бита влево.

3.6. Выбор максимального, минимального и ограничение диапазона

Стандарт IEC 61131-3 определяет набор вспомогательных операторов для работы с числовыми диапазонами, которые критически важны для безопасного управления технологическими процессами.

• **Минимум MIN** — бинарный оператор, возвращающий меньшее из двух значений. Например: `rFlow := MIN(rDemand, rMaxFlow);` гарантирует, что поток никогда не превысит максимально допустимое значение.

• **Максимум MAX** — бинарный оператор, возвращающий большее из двух значений. Используется для задания нижних границ: rSpeed := MAX(rSetpoint, rMinSpeed); обеспечивает работу привода не ниже минимальной скорости.

• **Ограничение LIMIT** — тернарный оператор, принимающий три параметра: нижнюю границу, ограничиваемое значение и верхнюю границу. Результатом будет значение, приведенное к допустимому диапазону. Например: rOutput := LIMIT(0.0, rRawSignal, 100.0); обрежет сигнал, выходящий за пределы 0–100.

• **Селектор SEL** — тернарный оператор выбора по булевому условию. Синтаксис: SEL(G, IN0, IN1). Если G = FALSE, возвращается IN0, если G = TRUE, возвращается IN1.

• **Мультиплексор MUX** — оператор выбора из массива значений по целочисленному индексу. Синтаксис: MUX(K, IN0, IN1, IN2, IN3). Возвращает тот операнд, номер которого соответствует значению K.

3.7. Таблица старшинства и приоритета операторов

При вычислении сложных выражений компилятор CODESYS v3.5 следует строгому порядку приоритетов операторов. Операторы с более высоким приоритетом выполняются раньше операторов с низким приоритетом. При одинаковом приоритете вычисление производится слева направо.

Порядок убывания приоритета операторов в Structured Text:

1. **Скобки ()** — выражения в скобках вычисляются в первую очередь.

2. **Унарные операции** — унарный минус -, логическое отрицание NOT, взятие адреса ADR.

3. **Мультипликативные операции** — умножение *, деление /, взятие остатка MOD.

4. **Аддитивные операции** — сложение +, вычитание -.

5. **Операторы отношения** — <, >, <=, >=.

6. **Операторы равенства** — =, <>.

7. **Логическое И AND** и побитовое умножение.

8. **Логическое исключающее ИЛИ XOR** и побитовое сложение.

9. **Логическое ИЛИ OR** и побитовое сложение.

Использование круглых скобок для явного группирования операндов является обязательной инженерной практикой, исключающей неоднозначное толкование выражений и ошибки, связанные с неверным порядком вычислений.

ПРАКТИКУМ ГЛАВЫ 3

Пример 13. Реализация релейной логики «Пуск-Стоп» с самоподхватом

Классическая схема управления электродвигателем с самоподхватом демонстрирует применение логических операторов для создания устойчивого состояния без использования операторов ветвления.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Кнопка «Пуск» (нормально разомкнутая, NO)
  xBtnStart        : BOOL := FALSE;
```

```
  // Кнопка «Стоп» (нормально замкнутая, NC)
  xBtnStop          : BOOL := TRUE;

  // Тепловое реле защиты двигателя (нормально замкнутое, NC)
  xThermalRelayOk    : BOOL := TRUE;

  // Внутреннее состояние самоподхвата схемы
  xMotorContactorState : BOOL := FALSE;

  // Выходной сигнал на силовой контактор двигателя
  xMotorContactorOut  : BOOL := FALSE;
END_VAR
```

Исполняемый код программы:

```
// Реализация схемы самоподхвата через логическое выражение:
// Контактор включается, если нажата кнопка ПУСК ИЛИ уже включен самоподхват,
// И при этом не нажата кнопка СТОП И исправно тепловое реле.

// Формула самоподхвата:
// xMotorContactorState = (xBtnStart OR xMotorContactorState) AND xBtnStop AND xThermalRelayOk

xMotorContactorState := (xBtnStart OR xMotorContactorState) AND
                        xBtnStop AND
                        xThermalRelayOk;
```

```
// Прямая передача состояния на физический выход контактора
xMotorContactorOut := xMotorContactorState;
```

Построчный разбор логики:

• Выражение (xBtnStart OR xMotorContactorState) реализует параллельное соединение кнопки пуска и нормально разомкнутого вспомогательного контакта контактора.

• Оператор AND xBtnStop выступает в роли размыкателя цепи: при нажатии кнопки «Стоп» (размыкание контакта, сигнал FALSE) вся логическая цепочка обесточивается.

• Оператор AND xThermalRelayOk добавляет защитную блокировку: при перегрузке двигателя тепловое реле размыкает цепь управления.

• Присваивание xMotorContactorState := ... обновляет состояние самоподхвата в каждом цикле сканирования, обеспечивая непрерывное удержание контактора во включенном состоянии после отпускания кнопки «Пуск».

Пример 14. Программный селектор аналогового датчика с резервированием 2 из 3

Пример реализует алгоритм голосования по схеме «2 из 3» для трех независимых датчиков давления, обеспечивая безаварийную работу системы при отказе одного из сенсоров.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Три независимых аналоговых датчика давления, бар
```

```
  rPressureSensor1    : REAL := 5.2;
  rPressureSensor2    : REAL := 5.3;
  rPressureSensor3    : REAL := 12.5; // АВАРИЙНЫЙ датчик с завышенным показанием

  // Максимально допустимая погрешность между датчиками, бар
  rMaxDeviation       : REAL := 1.0;

  // Выходное усредненное значение давления
  rPressureOutput     : REAL := 0.0;

  // Флаги исправности каждого датчика
  xSensor1Ok          : BOOL;
  xSensor2Ok          : BOOL;
  xSensor3Ok          : BOOL;

  // Флаг общей аварийной ситуации
  xSystemFault        : BOOL;
END_VAR
```

Исполняемый код программы:

```
// ШАГ 1: ВЗАИМНОЕ СРАВНЕНИЕ ПОКАЗАНИЙ ДАТЧИКОВ
// Датчик считается исправным, если его отклонение от среднего арифметического
// не превышает допустимую погрешность

rPressureOutput := (rPressureSensor1 + rPressureSensor2 + rPressureSensor3) / 3.0;
```

```
xSensor1Ok := ABS(rPressureSensor1 - rPressureOutput) <= rMaxDeviation;
xSensor2Ok := ABS(rPressureSensor2 - rPressureOutput) <= rMaxDeviation;
xSensor3Ok := ABS(rPressureSensor3 - rPressureOutput) <= rMaxDeviation;

// ШАГ 2: АЛГОРИТМ ГОЛОСОВАНИЯ ПО СХЕМЕ «2 ИЗ 3»
// Система работает, если хотя бы два датчика исправны
xSystemFault := NOT ((xSensor1Ok AND xSensor2Ok) OR
                (xSensor1Ok AND xSensor3Ok) OR
                (xSensor2Ok AND xSensor3Ok));

// ШАГ 3: ФОРМИРОВАНИЕ ВЫХОДНОГО СИГНАЛА
IF xSystemFault THEN
   // Авария: менее двух датчиков исправны
   rPressureOutput := 0.0; // Безопасное значение
ELSE
   // Нормальный режим: усредняем показания только исправных датчиков
   CASE TRUE OF
      (xSensor1Ok AND xSensor2Ok AND NOT xSensor3Ok):
         // Неисправен датчик 3
         rPressureOutput := (rPressureSensor1 + rPressureSensor2) / 2.0;

      (xSensor1Ok AND NOT xSensor2Ok AND xSensor3Ok):
         // Неисправен датчик 2
         rPressureOutput := (rPressureSensor1 + rPressureSensor3) / 2.0;

      (NOT xSensor1Ok AND xSensor2Ok AND xSensor3Ok):
         // Неисправен датчик 1
         rPressureOutput := (rPressureSensor2 + rPressureSensor3) / 2.0;
```

```
    (xSensor1Ok AND xSensor2Ok AND xSensor3Ok):
      // Все три датчика исправны — среднее арифметическое
      rPressureOutput := (rPressureSensor1 + rPressureSensor2 +
rPressureSensor3) / 3.0;

    ELSE
      // Менее двух датчиков исправны — аварийное отключение
      rPressureOutput := 0.0;
  END_CASE;
END_IF;
```

Построчный разбор логики:

• Вычисление среднего арифметического (`rPressureSensor1 + rPressureSensor2 + rPressureSensor3) / 3.0` дает эталонное значение для сравнения.

• Функция `ABS(...)` вычисляет абсолютное отклонение каждого датчика от среднего.

• Логическое выражение голосования проверяет наличие хотя бы двух исправных датчиков из трех возможных пар.

• Оператор `CASE TRUE OF` реализует детерминированный выбор формулы усреднения в зависимости от комбинации исправных сенсоров.

Пример 15. Циклическое бегущее реле на базе битового сдвига ROL

Пример демонстрирует создание световой индикации «бегущего огня» с использованием циклического сдвига единичного бита в слове состояния.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // 16-битное слово состояния бегущего реле
  wRunningLightWord   : WORD := 16#0001;

  // Временная переменная для хранения предыдущего состояния
  wPrevWord          : WORD;

  // Сигнал тактирования от таймера (импульс каждые 500 мс)
  xClockPulse        : BOOL := FALSE;
  xPrevClockPulse    : BOOL := FALSE;

  // Массив для визуализации состояния отдельных ламп
  axLampStatus       : ARRAY[0..15] OF BOOL;

  // Индекс цикла отображения
  iLampIndex         : INT;
END_VAR
```

Исполняемый код программы:

```
// Генерация тактового импульса (в реальном проекте здесь будет контакт таймера TON)
// Для демонстрации эмулируем фронт импульса
xClockPulse := NOT xClockPulse; // Эмуляция мигания

// Детектор переднего фронта тактового сигнала
IF xClockPulse AND NOT xPrevClockPulse THEN
```

```
    // Выполняем циклический сдвиг слова влево на 1 бит
    // Единичный бит перемещается с позиции 0 на позицию 1, затем на 2, и т.д.
    // При достижении 15-го бита он возвращается на позицию 0 (цикличность)
    wRunningLightWord := ROL(wRunningLightWord, 1);
END_IF;

xPrevClockPulse := xClockPulse;

// Распаковка слова состояния в массив отдельных булевых флагов
FOR iLampIndex := 0 TO 15 DO
    axLampStatus[iLampIndex] := wRunningLightWord.iLampIndex;
END_FOR;
```

Построчный разбор логики:

- Инициализация `wRunningLightWord := 16#0001;` устанавливает начальный паттерн с единственным активным битом на позиции 0.
- Функция `ROL(wRunningLightWord, 1)` выполняет циклический сдвиг на 1 бит влево. Бит, выходящий за 15-й разряд, автоматически заносится в 0-й разряд, обеспечивая непрерывное циклическое движение.
- Детектор фронта `IF xClockPulse AND NOT xPrevClockPulse` гарантирует, что сдвиг происходит ровно один раз за каждое изменение тактового сигнала.
- Цикл `FOR` с точечной адресацией `wRunningLightWord.iLampIndex` извлекает состояние каждого бита для индивидуального управления лампами индикации.

Пример 16. Ограничение максимальной скорости нарастания аналогового задания (Ramp-ограничитель)

Пример реализует программный ограничитель скорости изменения управляющего сигнала (Ramp Function), предотвращающий резкие скачки задания на приводы и гидравлические клапаны.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Мгновенное целевое задание от оператора или SCADA
  rTargetSetpoint    : REAL := 50.0;

  // Текущее сглаженное выходное задание на привод
  rRampedOutput      : REAL := 0.0;

  // Предыдущее значение выхода (для вычисления дельты)
  rPrevOutput        : REAL := 0.0;

  // Максимально допустимое изменение за один цикл сканирования
  rMaxDeltaPerCycle  : REAL := 0.5;

  // Флаг достижения целевого значения
  xSetpointReached   : BOOL;
END_VAR
```

Исполняемый код программы:

```
// Вычисление разности между целевым и текущим значением
// Функция ABS возвращает абсолютное значение разности
IF ABS(rTargetSetpoint - rRampedOutput) <= rMaxDeltaPerCycle THEN
  // Если разница меньше допустимой дельты — сразу переходим к заданию
  rRampedOutput := rTargetSetpoint;
  xSetpointReached := TRUE;
ELSE
  // Если разница велика — ограничиваем скорость нарастания через функцию LIMIT
  // Нижняя граница: rPrevOutput - rMaxDeltaPerCycle (плавное снижение)
  // Верхняя граница: rPrevOutput + rMaxDeltaPerCycle (плавный рост)
  rRampedOutput := LIMIT(rPrevOutput - rMaxDeltaPerCycle,
                         rTargetSetpoint,
                         rPrevOutput + rMaxDeltaPerCycle);
  xSetpointReached := FALSE;
END_IF;

// Сохранение текущего значения для следующего цикла
rPrevOutput := rRampedOutput;
```

Построчный разбор логики:

- Условие `ABS(rTargetSetpoint - rRampedOutput) <= rMaxDeltaPerCycle` проверяет, достигли ли мы целевой точки с допустимой точностью.
- Функция `LIMIT(Min, Value, Max)` гарантирует, что новое значение `rRampedOutput` не отклонится от предыдущего более чем на `rMaxDeltaPerCycle` в любую сторону.

• Если целевое значение растет быстрее допустимой скорости, LIMIT обрежет его до rPrevOutput + rMaxDeltaPerCycle.

• Если целевое значение падает быстрее допустимой скорости, LIMIT поднимет его до rPrevOutput - rMaxDeltaPerCycle.

Пример 17. Деление секунд на часы, минуты и остаток секунд

Пример демонстрирует применение операторов целочисленного деления / и взятия остатка MOD для декомпозиции временного интервала.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Общее количество секунд работы оборудования
  udiTotalSeconds    : UDINT := 3725;

  // Выделенные компоненты времени
  uiHours          : UINT;
  uiMinutes        : UINT;
  uiSeconds        : UINT;

  // Промежуточные переменные вычислений
  udiRemainingSeconds : UDINT;
END_VAR
```

Исполняемый код программы:

```
// Инициализация промежуточной переменной
udiRemainingSeconds := udiTotalSeconds;
```

```
// ШАГ 1: Выделение полных часов (3600 секунд в одном часе)
uiHours := TO_UINT(udiRemainingSeconds / 3600);

// ШАГ 2: Оставшиеся секунды после извлечения часов
udiRemainingSeconds := udiRemainingSeconds MOD 3600;

// ШАГ 3: Выделение полных минут (60 секунд в одной минуте)
uiMinutes := TO_UINT(udiRemainingSeconds / 60);

// ШАГ 4: Оставшиеся секунды после извлечения минут — это и есть итоговые секунды
uiSeconds := TO_UINT(udiRemainingSeconds MOD 60);
```

Построчный разбор логики:

- Оператор `/ 3600` выполняет целочисленное деление общего количества секунд на количество секунд в часе, отбрасывая дробную часть.
- Оператор `MOD 3600` возвращает остаток от деления на 3600, то есть секунды, не вошедшие в полные часы.
- Аналогично, `/ 60` выделяет полные минуты, а `MOD 60` — оставшиеся секунды.
- Явное преобразование `TO_UINT(...)` необходимо для сужения 32-битного `UDINT` до 16-битного `UINT`.

Пример 18. Программная установка, сброс и инвертирование бита в управляющем слове

Пример иллюстрирует работу с битовыми масками для управления отдельными разрядами коммуникационного слова без влияния на соседние биты.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // 16-битное управляющее слово привода
  wControlWord      : WORD := 16#0000;

  // Номер бита для манипуляций (от 0 до 15)
  iBitNumber        : INT := 5;

  // Команды управления битом
  xCmdSetBit        : BOOL := FALSE;  // Установить бит в 1
  xCmdResetBit      : BOOL := FALSE;  // Сбросить бит в 0
  xCmdToggleBit     : BOOL := FALSE;  // Инвертировать бит
END_VAR
```

Исполняемый код программы:

```
// ОПЕРАЦИЯ 1: Установка бита в 1 (SET)
// Создаем маску с единицей в позиции iBitNumber через сдвиг ROL
// Логическое ИЛИ (OR) устанавливает целевой бит, не затрагивая остальные
IF xCmdSetBit THEN
```

```
  wControlWord := wControlWord OR ROL(WORD#16#0001, iBitNumber);
END_IF;

// ОПЕРАЦИЯ 2: Сброс бита в 0 (RESET)
// Создаем инвертированную маску: все биты 1, кроме целевого (который 0)
// Логическое И (AND) с инвертированной маской обнуляет целевой бит
IF xCmdResetBit THEN
  wControlWord := wControlWord AND NOT ROL(WORD#16#0001, iBitNumber);
END_IF;

// ОПЕРАЦИЯ 3: Инвертирование бита (TOGGLE)
// Исключающее ИЛИ (XOR) с маской инвертирует только целевой бит
IF xCmdToggleBit THEN
  wControlWord := wControlWord XOR ROL(WORD#16#0001, iBitNumber);
END_IF;
```

Построчный разбор логики:

• Выражение `ROL(WORD#16#0001, iBitNumber)` формирует маску с единственным единичным битом в позиции `iBitNumber`.

• Оператор `OR` с маской устанавливает бит в 1, так как `X OR 1 = 1` для любого `X`, при этом `X OR 0 = X` сохраняет остальные биты.

• Оператор `AND NOT` с маской сбрасывает бит в 0, так как `X AND 0 = 0` для любого `X`, при этом `X AND 1 = X` сохраняет остальные биты.

• Оператор `XOR` с маской инвертирует бит, так как `0 XOR 1 = 1` и `1 XOR 1 = 0`, при этом `X XOR 0 = X` сохраняет остальные биты.

Пример 19. Вычисление евклидова расстояния между двумя точками на плоскости

Пример демонстрирует применение математических функций стандартной библиотеки CODESYS для расчета расстояния по теореме Пифагора.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Координаты точки A
  rPointA_X        : REAL := 10.0;
  rPointA_Y        : REAL := 20.0;

  // Координаты точки B
  rPointB_X        : REAL := 40.0;
  rPointB_Y        : REAL := 60.0;

  // Результирующее расстояние между точками
  rEuclideanDistance  : REAL;

  // Разности координат по осям
  rDeltaX          : REAL;
  rDeltaY          : REAL;
END_VAR
```

Исполняемый код программы:

```
// ШАГ 1: Вычисление разности координат по осям X и Y
rDeltaX := rPointB_X - rPointA_X;
rDeltaY := rPointB_Y - rPointA_Y;
```

```
// ШАГ 2: Применение теоремы Пифагора: c = sqrt(a^2 + b^2)
// Функция SQRT возвращает квадратный корень из аргумента
// Оператор ** возводит в степень (в CODESYS v3.5 используется MUL или функция POW)
rEuclideanDistance := SQRT(rDeltaX * rDeltaX + rDeltaY * rDeltaY);

// АЛЬТЕРНАТИВНЫЙ ВАРИАНТ с использованием функции возведения в степень:
// rEuclideanDistance := SQRT(POW(rDeltaX, 2.0) + POW(rDeltaY, 2.0));
```

Построчный разбор логики:

• Вычисление разностей `rDeltaX` и `rDeltaY` определяет проекции отрезка на оси координат.

• Выражение `rDeltaX * rDeltaX` возводит разность в квадрат (умножение числа само на себя).

• Функция `SQRT(...)` извлекает квадратный корень из суммы квадратов, завершая расчет по теореме Пифагора.

• В CODESYS v3.5 стандартная библиотека `Standard.lib` содержит функцию `SQRT` для извлечения корня и `POW` для возведения в произвольную степень.

КОНТРОЛЬНЫЕ ВОПРОСЫ И УПРАЖНЕНИЯ К ГЛАВЕ 3

1. Объясните разницу между операторами `:=` и `=` в контексте выполнения программ на языке Structured Text. Что произойдет, если ошибочно использовать `=` вместо `:=` в присваивании?

2. Почему выражение iResult := 10 / 3; при объявлении iResult : INT; даст результат 3, а не 3.333? Как получить правильный вещественный результат?

3. Опишите таблицу истинности для оператора XOR. В каких технологических ситуациях применяется исключающее ИЛИ?

4. Что вернет выражение wResult := SHL(WORD#16#0001, 4);? Объясните побитовый механизм выполнения.

5. Каков результат выполнения оператора SEL(FALSE, 100, 200)? Объясните логику работы селектора.

6. Расставьте скобки в выражении rY := 5.0 + 3.0 * 2.0 - 1.0 / 4.0; согласно приоритетам операторов и вычислите итоговое значение.

7. Почему операторы сравнения <, >, <=, >= неприменимы к переменным типа BOOL?

8. Объясните разницу между логическим сдвигом SHR и циклическим сдвигом ROR. В каких случаях каждый из них используется?

9. Как с помощью одного выражения на языке ST установить в 1 биты 0, 3 и 7 слова wControlWord, не изменяя остальные биты?

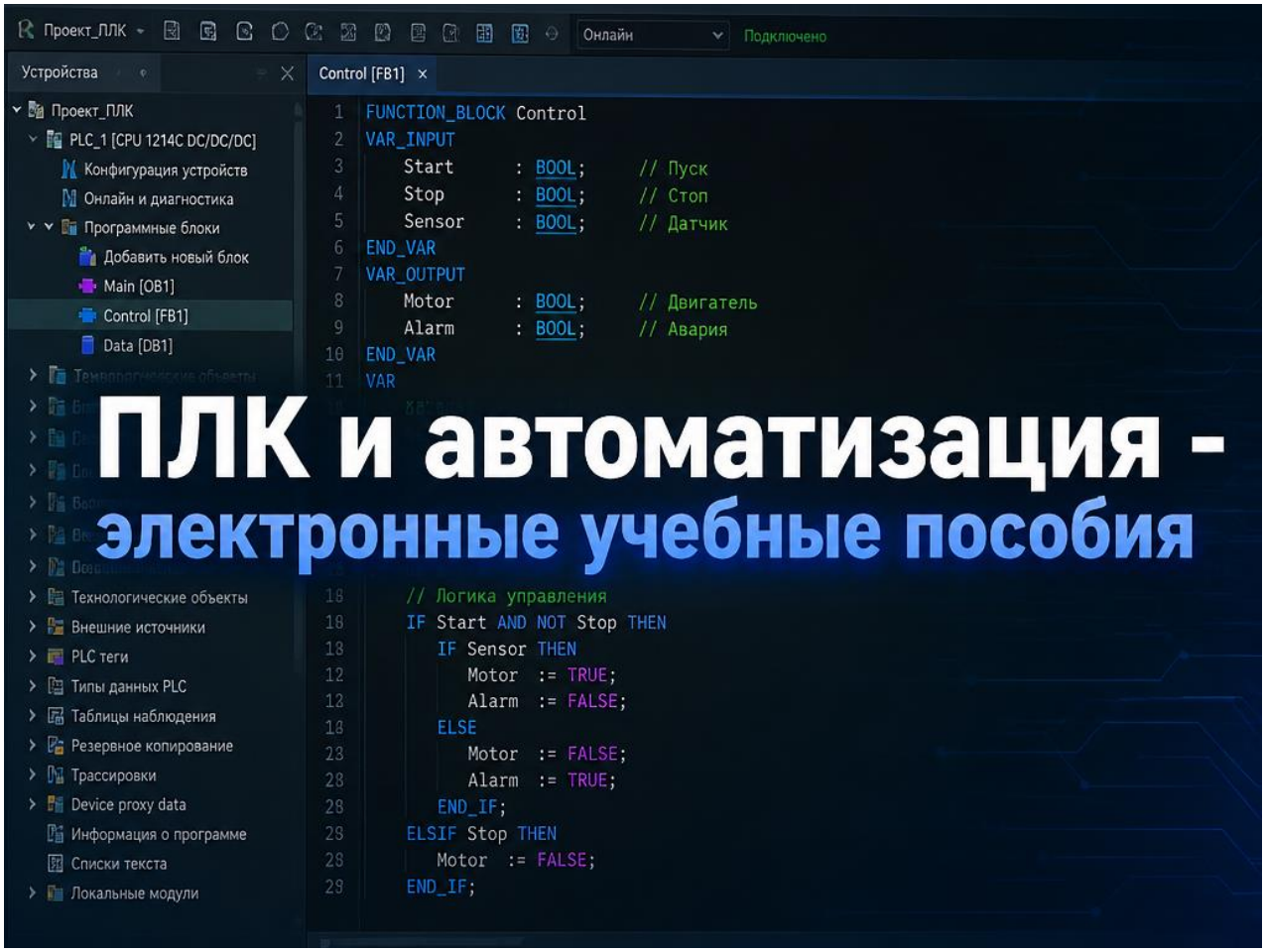
10. Что вернет функция LIMIT(0.0, -5.0, 100.0)? Объясните механизм работы ограничителя.

Практические упражнения к главе 3

Упражнение 3.1. Напишите выражение на языке ST, которое вычисляет значение переменной xResult : BOOL по правилу: результат истинен тогда и только тогда, когда из трех дискретных датчиков xSensorA, xSensorB, xSensorC замкнуты ровно два любых датчика одновременно. Запрещено использовать операторы IF и CASE.

Упражнение 3.2. Разработайте алгоритм на языке ST для вычисления среднего арифметического трех вещественных чисел rValue1, rValue2, rValue3, исключив из расчета максимальное и минимальное значение (то есть усредняется только среднее по величине число).

Упражнение 3.3. Напишите фрагмент кода, который циклически сдвигает единичный бит в байте bShiftRegister : BYTE := 16#01; вправо на одну позицию при каждом срабатывании флага xShiftClock : BOOL;. При достижении младшего бита (позиция 0) единичный бит должен возвращаться в старший разряд (позиция 7).



ПЛК и автоматизация — электронные учебные пособия

Практические электронные учебные пособия по программированию ПЛК, языку Structured Text и промышленной автоматизации.

В каталоге собраны книги для студентов, начинающих инженеров, специалистов АСУ ТП, электриков, наладчиков и преподавателей. Материалы помогают последовательно пройти путь от основ программирования контроллеров до разработки архитектуры промышленных систем управления.

В пособиях рассматриваются:

- устройство и принцип работы программируемых логических контроллеров;

- программирование ПЛК на языке Structured Text по IEC 61131-3;
- типы данных, операторы, условия и циклы;
- таймеры, счётчики и конечные автоматы;
- функции и функциональные блоки;
- массивы, структуры и пользовательские типы данных;
- дискретные и аналоговые входы и выходы;
- управление электродвигателями, насосами, клапанами и нагревателями;
- масштабирование сигналов и работа с аналоговыми каналами;
- диагностика, аварийные блокировки и обработка ошибок;
- архитектура ПЛК-проектов;
- взаимодействие ПЛК с HMI и SCADA;
- автоматизация печей, теплиц, станций водоочистки и других технологических объектов.

Материалы построены на практических примерах. Вместо сухого перечисления команд читатель видит, как с помощью Structured Text реализовать реальные алгоритмы управления: самоподхват двигателя, контроль уровня в резервуаре, управление насосной станцией, обработку аварий, работу с аналоговыми датчиками и последовательность технологических операций.

Для начала обучения доступно бесплатное пособие **«Structured Text с нуля»**. В нём разобраны базовый синтаксис, типы данных, операторы, циклы, функции, функциональные блоки и первые проекты в CODESYS.

Если вы хотите изучать тему последовательно, в магазине также доступны тематические пособия и комплекты электронных книг:

- **«Structured Text в программировании ПЛК: от основ к промышленным проектам»**;
- **«Практика на Structured Text»**;
- **«Библиотека программиста ПЛК»**;
- пособия по управлению двигателями, аналоговыми сигналами, функциональным блокам, диагностике и промышленным объектам;
- комплекс лекций по основам автоматизации производства.

Магазин электронных учебных пособий

Все книги и комплекты собраны в каталоге:

Перейти в магазин электронных учебных пособий

Издания можно выбирать по отдельным темам или приобретать готовыми комплектами — для начального, практического или углублённого изучения программирования ПЛК и промышленной автоматизации.

ГЛАВА 4. УПРАВЛЕНИЕ ПОТОКОМ ВЫПОЛНЕНИЯ: ВЕТВЛЕНИЯ И ЦИКЛЫ

4.1. Условный оператор IF-THEN-ELSE и его полные формы

Условный оператор IF-THEN-ELSE представляет собой базовый механизм ветвления алгоритма, позволяющий выполнять различные участки кода в зависимости от логического состояния булевого выражения. В отличие от непрерывного выполнения арифметических операций, оператор IF обеспечивает дискретное переключение логики работы программы, что критически важно для реализации режимов работы оборудования, аварийных блокировок и технологических переходов.

Полная синтаксическая форма оператора в стандарте IEC 61131-3 выглядит следующим образом:

```
IF <Булево_выражение_1> THEN
  <Инструкции_для_выполнения_при_истине_1>
ELSIF <Булево_выражение_2> THEN
  <Инструкции_для_выполнения_при_истине_2>
ELSIF <Булево_выражение_3> THEN
  <Инструкции_для_выполнения_при_истине_3>
ELSE
  <Инструкции_для_выполнения_при_лжи_всех_условий>
END_IF;
```

Ключевое правило выполнения оператора IF: компилятор последовательно проверяет каждое условие начиная с первого. Как только обнаруживается первое истинное выражение, выполняются соответствующие ему инструкции, после чего управление немедленно передается на строку,

следующую за END_IF. Остальные условия ELSIF и ветвь ELSE в этом случае полностью игнорируются, даже если они также истинны.

Порядок следования условий в каскаде IF-ELSIF имеет принципиальное значение для быстродействия алгоритма. Инженер обязан размещать наиболее вероятные состояния оборудования в начале цепочки проверок. Например, если технологический процесс 95% времени находится в нормальном режиме работы, проверка условия xNormalMode должна стоять первой, чтобы процессор не тратил циклы на анализ редких аварийных ситуаций.

Ветвь ELSE является необязательной, но ее наличие гарантирует детерминированное поведение алгоритма в любых, даже нештатных ситуациях. Отсутствие ветви ELSE означает, что при ложности всех условий программа просто пропустит блок IF без выполнения каких-либо действий, что может привести к зависанию исполнительных механизмов в неопределенном состоянии.

4.2. Многопутевой выбор CASE-OF и его преимущества

Оператор множественного выбора CASE-OF предназначен для ситуаций, когда необходимо выполнить одно из множества альтернативных действий в зависимости от значения целочисленной или перечислимой переменной. В отличие от каскада IF-ELSIF, который проверяет логические выражения, оператор CASE сравнивает значение селектора с конкретными константами или диапазонами.

Синтаксическая структура оператора CASE:

```
CASE <Целочисленная_переменная> OF
  <Константа_1>:
    <Инструкции_для_константы_1>;
  <Константа_2>, <Константа_3>:
```

```
    <Инструкции_для_констант_2_и_3>;
  <Начало_диапазона>..<Конец_диапазона>:
    <Инструкции_для_диапазона>;
  ELSE
    <Инструкции_для_несоответствия_ни_одному_варианту>;
END_CASE;
```

Важнейшее требование стандарта: ветвь ELSE в операторе CASE является обязательной. Компилятор CODESYS v3.5 выдаст ошибку, если программист не предусмотрит обработку значения селектора, не попавшего ни в один из явных вариантов. Это требование обеспечивает безопасность алгоритма: даже при получении некорректного или непредусмотренного значения программа выполнит определенную логику, а не пропустит блок без внимания.

Оператор CASE обладает существенным преимуществом перед каскадом IF-ELSIF в производительности. Компилятор генерирует оптимизированный код с использованием таблиц переходов (jump tables), что обеспечивает константное время выполнения независимо от количества вариантов. В то время как каскад из 20 условий ELSIF в худшем случае потребует 20 последовательных проверок, оператор CASE выполнит выбор за один машинный такт.

Диапазоны значений в CASE записываются через две точки: 1..10: означает все целые числа от 1 до 10 включительно. Несколько отдельных констант перечисляются через запятую: 1, 3, 5, 7: охватит все нечетные числа первого десятка.

4.3. Опасности циклических конструкций в программируемых контроллерах

Циклические конструкции в языке Structured Text представляют собой наиболее мощный и одновременно наиболее опасный инструмент разработчика. В отличие от операторов ветвления, которые выполняются однократно за цикл сканирования задачи, циклы могут многократно повторять один и тот же участок кода, что создает риск выхода за пределы допустимого времени выполнения задачи.

Критическая особенность программируемых контроллеров: программа на языке ST не выполняется однократно от начала до конца. Она вызывается циклическим планировщиком реального времени (Real-Time Scheduler) через строго фиксированные интервалы, называемые временем цикла задачи (Task Cycle Time). Для высокоскоростных задач управления приводами это время составляет 1–4 миллисекунды, для стандартных задач логики — 10–20 миллисекунд.

Аппаратный сторожевой таймер (Hardware Watchdog) представляет собой независимый микроконтроллер внутри ПЛК, который непрерывно отслеживает время выполнения пользовательской программы. Если программа не завершает свой цикл в течение заданного времени (обычно 150–200% от номинального Task Cycle Time), сторожевой таймер генерирует аппаратный сброс контроллера. Это приводит к отключению всех силовых выходов, останову технологического процесса и переходу системы в безопасное состояние.

Бесконечный цикл вида `WHILE TRUE DO ... END_WHILE;` является фатальной ошибкой в программировании ПЛК. Такой код никогда не вернет управление системному планировщику, сторожевой таймер зафиксирует

превышение времени и выполнит аварийный сброс контроллера. В производственной среде это равносильно полной потере управления технологическим процессом.

4.4. Детерминированный цикл FOR-TO-BY-DO

Цикл FOR представляет собой детерминированную циклическую конструкцию с заранее известным количеством итераций. Этот тип цикла является наиболее безопасным и предпочтительным для использования в программируемых контроллерах, поскольку количество повторений задается явно в заголовке цикла.

Синтаксическая форма цикла FOR:

```
FOR <Счетчик> := <Начальное_значение> TO <Конечное_значение> BY <Шаг>
DO
  <Тело_цикла>;
END_FOR;
```

Параметры цикла:

- **Счетчик** — переменная целочисленного типа, которая автоматически инкрементируется на величину шага после каждой итерации.
- **Начальное значение** — значение, с которого начинается цикл.
- **Конечное значение** — предельное значение, при достижении которого цикл завершается.
- **Шаг (BY)** — необязательный параметр, определяющий величину приращения счетчика. По умолчанию равен 1.

Важная особенность: переменная-счетчик цикла FOR не может быть модифицирована внутри тела цикла. Любая попытка присвоить ей новое значение внутри тела цикла приведет к неопределенному поведению или ошибке компиляции.

Цикл FOR идеально подходит для обработки массивов фиксированного размера, выполнения математических вычислений с известным количеством шагов и итерационных алгоритмов с гарантированным завершением.

4.5. Итерационные циклы с предусловием и постусловием

Циклы с предусловием WHILE-DO и постусловием REPEAT-UNTIL представляют собой более гибкие, но и более опасные циклические конструкции. Их выполнение продолжается до тех пор, пока выполняется (или не выполняется) определенное логическое условие, что делает количество итераций заранее неизвестным.

Цикл с предусловием WHILE-DO

Синтаксическая форма:

```
WHILE <Булево_условие> DO
  <Тело_цикла>;
END_WHILE;
```

Механизм выполнения: перед каждой итерацией цикла проверяется булево условие. Если условие истинно (TRUE), выполняется тело цикла. Если условие ложно (FALSE), цикл немедленно завершается, и управление передается на инструкцию после END_WHILE.

Критическая опасность: если условие цикла никогда не становится ложным, цикл становится бесконечным и вызывает срабатывание сторожевого таймера. Программист обязан гарантировать, что внутри тела цикла происходит изменение переменных, участвующих в условии, таким образом, чтобы цикл обязательно завершился.

Цикл с постусловием REPEAT-UNTIL

Синтаксическая форма:

```
REPEAT
  <Тело_цикла>;
UNTIL <Булево_условие>
END_REPEAT;
```

Механизм выполнения: тело цикла выполняется как минимум один раз, после чего проверяется условие завершения. Если условие истинно (TRUE), цикл завершается. Если условие ложно (FALSE), выполняется следующая итерация.

Отличие от WHILE: цикл REPEAT-UNTIL гарантирует однократное выполнение тела цикла даже при изначально истинном условии завершения, тогда как WHILE может не выполниться ни разу.

4.6. Операторы прерывания и пропуска итераций

Стандарт IEC 61131-3 определяет три оператора управления потоком выполнения внутри циклов и программных блоков, позволяющих гибко контролировать ход вычислений.

Оператор досрочного выхода EXIT

Оператор EXIT обеспечивает немедленный выход из текущего цикла, независимо от состояния условия продолжения. Управление передается на инструкцию, следующую за закрывающим ключевым словом цикла (END_FOR, END_WHILE или END_REPEAT).

Синтаксис:

```
EXIT;
```

Применение: используется для оптимизации алгоритмов поиска, когда целевой элемент найден и дальнейший перебор не имеет смысла. Также применяется для аварийного выхода из цикла при обнаружении нештатной ситуации.

Оператор пропуска итерации CONTINUE

Оператор CONTINUE прерывает текущую итерацию цикла и передает управление на начало следующей итерации. При этом условие продолжения цикла проверяется в обычном порядке.

Синтаксис:

```
CONTINUE;
```

Применение: используется для пропуска отдельных элементов массива, не удовлетворяющих определенным критериям, без прерывания всего цикла.

Оператор досрочного возврата RETURN

Оператор RETURN обеспечивает немедленный выход из текущей программной единицы (POU). Управление возвращается вызывающему коду, а для функций дополнительно указывается возвращаемое значение.

Синтаксис:

```
RETURN;
```

Для функций с возвращаемым значением:

```
RETURN <Выражение>;
```

Применение: используется для досрочного завершения функции при обнаружении недопустимых входных данных или аварийных условий.

ПРАКТИКУМ ГЛАВЫ 4

Пример 20. Трехпозиционный селектор режимов работы упаковочной машины

Пример демонстрирует применение оператора CASE для реализации детерминированного выбора алгоритма управления в зависимости от режима работы оборудования.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Текущий режим работы машины (0 = Выключено, 1 = Наладка, 2 = Автомат)
```

```
  iMachineMode       : INT := 0;

  // Выходные сигналы исполнительных механизмов
  xConveyorMotor     : BOOL := FALSE;
  xSealingHeater     : BOOL := FALSE;
  xCuttingBlade      : BOOL := FALSE;
  xWarningLamp       : BOOL := FALSE;

  // Сигналы от датчиков и кнопок
  xStartButton       : BOOL := FALSE;
  xEmergencyStop     : BOOL := TRUE;
  xGuardDoorClosed   : BOOL := FALSE;
END_VAR
```

Исполняемый код программы:

```
// Детерминированный выбор алгоритма по режиму работы
CASE iMachineMode OF
  0:
    // РЕЖИМ 0: ВЫКЛЮЧЕНО
    // Все исполнительные механизмы обесточены, горит лампа готовности
    xConveyorMotor := FALSE;
    xSealingHeater := FALSE;
    xCuttingBlade := FALSE;
    xWarningLamp := TRUE;

  1:
    // РЕЖИМ 1: НАЛАДКА (ручное управление)
    // Двигатель конвейера работает только при нажатой кнопке Старт
```

```
    // Нагреватель и нож отключены в целях безопасности
    xConveyorMotor := xStartButton AND xEmergencyStop AND
xGuardDoorClosed;
    xSealingHeater := FALSE;
    xCuttingBlade := FALSE;
    xWarningLamp := FALSE;

  2:
    // РЕЖИМ 2: АВТОМАТИЧЕСКАЯ РАБОТА
    // Все механизмы работают по полной технологической циклограмме
    // Требуется закрытое ограждение и отсутствие аварий
    IF xEmergencyStop AND xGuardDoorClosed THEN
      xConveyorMotor := TRUE;
      xSealingHeater := TRUE;
      xCuttingBlade := TRUE;
    ELSE
      // Аварийное отключение при открытой двери или нажатом Стоп
      xConveyorMotor := FALSE;
      xSealingHeater := FALSE;
      xCuttingBlade := FALSE;
    END_IF;
    xWarningLamp := FALSE;

  ELSE
    // НЕИЗВЕСТНЫЙ РЕЖИМ: безопасное состояние
    // Любое значение iMachineMode, не равное 0, 1 или 2
    xConveyorMotor := FALSE;
    xSealingHeater := FALSE;
    xCuttingBlade := FALSE;
```

```
    xWarningLamp := TRUE; // Мигание лампы сигнализирует об ошибке режима
END_CASE;
```

Построчный разбор логики:

- Селектор `CASE iMachineMode OF` обеспечивает четкое разделение алгоритмов для трех технологических режимов.
- В режиме 0 все механизмы принудительно отключены, что соответствует безопасному состоянию при выключенной машине.
- В режиме 1 (наладка) двигатель конвейера работает только при одновременном выполнении трех условий: нажата кнопка пуска, не нажат аварийный стоп и закрыто ограждение.
- В режиме 2 (автомат) добавляется проверка аварийных цепей перед включением всех механизмов.
- Ветвь `ELSE` обрабатывает некорректные значения переменной режима (3, 4, -1 и т.д.), переводя машину в безопасное состояние с индикацией неисправности.

Пример 21. Поиск максимального и минимального значения в массиве температур

Пример иллюстрирует применение цикла `FOR` для обработки массива данных с целью нахождения экстремальных значений.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
```

```
  // Массив из 16 температурных датчиков печи, град С
  aiTemperatures     : ARRAY[0..15] OF REAL := [150.5, 152.3, 148.7, 155.1,
                                                149.8, 153.2, 151.0, 154.6,
                                                150.2, 152.8, 149.5, 153.9,
                                                151.3, 150.7, 152.1, 154.2];

  // Переменные для хранения результатов
  rMaxTemperature    : REAL;
  rMinTemperature    : REAL;
  iMaxIndex          : INT;
  iMinIndex          : INT;

  // Счетчик цикла
  iSensorIndex       : INT;
END_VAR
```

Исполняемый код программы:

```
// Инициализация переменных результата первым элементом массива
rMaxTemperature := aiTemperatures[0];
rMinTemperature := aiTemperatures[0];
iMaxIndex := 0;
iMinIndex := 0;

// Циклический перебор всех элементов массива начиная со второго
FOR iSensorIndex := 1 TO 15 DO
  // Поиск максимального значения
  IF aiTemperatures[iSensorIndex] > rMaxTemperature THEN
    rMaxTemperature := aiTemperatures[iSensorIndex];
```

```
    iMaxIndex := iSensorIndex;
  END_IF;

  // Поиск минимального значения
  IF aiTemperatures[iSensorIndex] < rMinTemperature THEN
    rMinTemperature := aiTemperatures[iSensorIndex];
    iMinIndex := iSensorIndex;
  END_IF;
END_FOR;
```

Построчный разбор логики:

• Инициализация `rMaxTemperature` и `rMinTemperature` первым элементом массива гарантирует, что у переменных есть корректное начальное значение для сравнения.

• Цикл `FOR iSensorIndex := 1 TO 15 DO` начинает перебор со второго элемента, так как первый уже использован для инициализации.

• Условие `IF aiTemperatures[iSensorIndex] > rMaxTemperature` проверяет, превышает ли текущий элемент найденный максимум. Если да — обновляются и значение, и индекс.

• Аналогично работает поиск минимума через условие `< rMinTemperature`.

• После завершения цикла переменные содержат экстремальные значения и индексы датчиков, на которых они зафиксированы.

Пример 22. Линейный поиск индекса аварийного датчика с прерыванием EXIT

Пример демонстрирует оптимизацию алгоритма поиска через досрочный выход из цикла при обнаружении целевого элемента.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Массив из 32 дискретных датчиков заготовок
  axPartSensors      : ARRAY[0..31] OF BOOL;

  // Индекс найденного аварийного датчика
  iFaultSensorIndex   : INT := -1;

  // Флаг наличия аварии
  xFaultDetected     : BOOL := FALSE;

  // Счетчик цикла
  iSensorIndex       : INT;
END_VAR
```

Исполняемый код программы:

```
// Инициализация результата значением «не найдено»
iFaultSensorIndex := -1;
xFaultDetected := FALSE;

// Последовательный перебор всех датчиков
FOR iSensorIndex := 0 TO 31 DO
```

```
  // Проверка состояния текущего датчика (FALSE = отсутствие заготовки = авария)
  IF NOT axPartSensors[iSensorIndex] THEN
    // Аварийный датчик найден
    iFaultSensorIndex := iSensorIndex;
    xFaultDetected := TRUE;

    // НЕМЕДЛЕННЫЙ ВЫХОД ИЗ ЦИКЛА
    // Дальнейший перебор не имеет смысла — авария уже обнаружена
    EXIT;
  END_IF;
END_FOR;
```

Построчный разбор логики:

• Инициализация `iFaultSensorIndex := -1;` задает значение по умолчанию, указывающее на отсутствие аварий.

• Цикл `FOR iSensorIndex := 0 TO 31 DO` последовательно проверяет каждый датчик массива.

• Условие `IF NOT axPartSensors[iSensorIndex]` обнаруживает первый датчик со значением `FALSE`.

• Оператор `EXIT;` немедленно прерывает цикл, экономя процессорное время на проверке оставшихся 31–k датчиков, где k — индекс найденной аварии.

• Если цикл завершается естественно (без выполнения `EXIT`), переменная `iFaultSensorIndex` остается равной -1, что сигнализирует об отсутствии аварий.

Пример 23. Безопасный алгоритм расчета контрольной суммы CRC-8

Пример реализует вычисление контрольной суммы байтового массива с защитой от зависания через ограничение максимального числа итераций.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Массив данных для расчета CRC (например, телеграмма Modbus)
  abDataBuffer       : ARRAY[0..63] OF BYTE;

  // Количество байт в буфере (динамическая длина телеграммы)
  uiDataLength       : UINT := 20;

  // Результирующая контрольная сумма CRC-8
  bCRC8              : BYTE := 16#00;

  // Счетчик цикла
  iByteIndex         : INT;

  // Полином CRC-8 (стандартный 0x07)
  bPolynomial        : BYTE := 16#07;

  // Защитный счетчик итераций
  uiIterationCounter  : UINT := 0;
  uiMaxIterations     : UINT := 100; // Защита от бесконечного цикла
END_VAR
```

Исполняемый код программы:

```
// Инициализация CRC нулем
bCRC8 := 16#00;

// Проверка корректности длины данных
IF uiDataLength = 0 OR uiDataLength > 64 THEN
  // Некорректная длина — аварийный выход
  bCRC8 := 16#FF; // Специальный код ошибки
  RETURN;
END_IF;

// Цикл вычисления CRC с защитой от зависания
uiIterationCounter := 0;
iByteIndex := 0;

WHILE iByteIndex < INT#uiDataLength DO
  // ЗАЩИТНЫЙ МЕХАНИЗМ: счетчик итераций
  uiIterationCounter := uiIterationCounter + 1;
  IF uiIterationCounter > uiMaxIterations THEN
    // Превышено максимальное число итераций — аварийный выход
    bCRC8 := 16#FE; // Код ошибки зависания
    EXIT;
  END_IF;

  // Вычисление CRC-8 для текущего байта
  bCRC8 := bCRC8 XOR abDataBuffer[iByteIndex];

  // 8 итераций побитовой обработки
```

```
  FOR iBitIndex := 0 TO 7 DO
    IF (bCRC8 AND 16#80) <> 0 THEN
      bCRC8 := (bCRC8 SHL 1) XOR bPolynomial;
    ELSE
      bCRC8 := bCRC8 SHL 1;
    END_IF;
  END_FOR;

  // Переход к следующему байту
  iByteIndex := iByteIndex + 1;
END_WHILE;
```

Построчный разбор логики:

- Проверка `IF uiDataLength = 0 OR uiDataLength > 64` отсекает некорректные входные данные до начала вычислений.
- Счетчик `uiIterationCounter` инкрементируется на каждой итерации внешнего цикла `WHILE`.
- Условие `IF uiIterationCounter > uiMaxIterations` гарантирует, что цикл выполнится не более 100 раз, даже при ошибке в логике.
- Внутренний цикл `FOR iBitIndex := 0 TO 7 DO` обрабатывает каждый бит текущего байта по алгоритму CRC-8.
- Оператор `EXIT` в защитном механизме немедленно прерывает вычисления при обнаружении аномалии.

Пример 24. Расчет факториала с отсечкой по времени выполнения

Пример демонстрирует технику ограничения вычислительно сложных алгоритмов через контроль времени выполнения.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Входное число для расчета факториала
  uiFactorialInput    : UINT := 5;

  // Результат вычисления факториала
  ulFactorialResult   : ULINT := 1;

  // Таймер контроля времени выполнения
  tStartTime          : TIME;
  tCurrentTime        : TIME;
  tElapsedTime        : TIME;

  // Максимально допустимое время вычисления (5 мс)
  tMaxExecutionTime   : TIME := T#5ms;

  // Флаг успешного завершения
  xCalculationSuccess : BOOL := FALSE;

  // Счетчик цикла
  uiCounter           : UINT;
END_VAR
```

Исполняемый код программы:

```
// Фиксация времени начала вычислений
tStartTime := TIME_OF_DAY(); // В реальном проекте используется системный таймер

// Проверка допустимости входных данных
IF uiFactorialInput > 20 THEN
   // Факториал 21! превышает диапазон ULINT
   ulFactorialResult := 0; // Код переполнения
   xCalculationSuccess := FALSE;
   RETURN;
END_IF;

// Инициализация результата
ulFactorialResult := 1;

// Цикл вычисления факториала с контролем времени
FOR uiCounter := 1 TO uiFactorialInput DO
   // Проверка времени выполнения на каждой итерации
   tCurrentTime := TIME_OF_DAY();
   tElapsedTime := tCurrentTime - tStartTime;
   IF tElapsedTime > tMaxExecutionTime THEN
      // Превышено допустимое время — аварийный выход
      ulFactorialResult := 0; // Код ошибки таймаута
      xCalculationSuccess := FALSE;
      EXIT;
   END_IF;
```

```
    // Вычисление факториала
    ulFactorialResult := ulFactorialResult * uiCounter;
END_FOR;

// Если цикл завершен без прерывания — успех
xCalculationSuccess := (ulFactorialResult <> 0);
```

Построчный разбор логики:

• Проверка `IF uiFactorialInput > 20` предотвращает вычисление факториалов, заведомо вызывающих переполнение 64-битного типа `ULINT`.

• Фиксация `tStartTime` в начале вычислений позволяет контролировать общее время выполнения алгоритма.

• На каждой итерации цикла вычисляется прошедшее время `tElapsedTime` и сравнивается с лимитом `tMaxExecutionTime`.

• Оператор `EXIT` прерывает вычисления при превышении временного лимита, предотвращая срабатывание сторожевого таймера контроллера.

Пример 25. Сортировка массива методом пузырька с оптимизацией

Пример реализует классический алгоритм сортировки с использованием вложенных циклов и флага оптимизации.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
    // Массив для сортировки (например, приоритеты заданий)
```

```
  aiPriorityArray    : ARRAY[0..9] OF INT := [45, 12, 78, 23, 56, 89, 34, 67, 91, 15];

  // Временная переменная для обмена элементов
  iTempValue         : INT;

  // Счетчики внешнего и внутреннего циклов
  iOuterIndex        : INT;
  iInnerIndex        : INT;

  // Флаг наличия перестановок в текущей итерации
  xSwapsOccurred     : BOOL;

  // Счетчик выполненных итераций для защиты
  uiIterationCount   : UINT := 0;
  uiMaxIterations    : UINT := 200;
END_VAR
```

Исполняемый код программы:

```
// Внешний цикл: количество проходов равно количеству элементов
FOR iOuterIndex := 0 TO 8 DO
  // Инициализация флага перестановок
  xSwapsOccurred := FALSE;

  // Внутренний цикл: сравнение соседних элементов
  FOR iInnerIndex := 0 TO (8 - iOuterIndex) DO
    // Защита от зависания вложенного цикла
    uiIterationCount := uiIterationCount + 1;
    IF uiIterationCount > uiMaxIterations THEN
```

```
        EXIT; // Аварийный выход из вложенного цикла
      END_IF;

      // Сравнение соседних элементов
      IF aiPriorityArray[iInnerIndex] > aiPriorityArray[iInnerIndex + 1] THEN
        // Обмен элементов местами (пузырьковая сортировка)
        iTempValue := aiPriorityArray[iInnerIndex];
        aiPriorityArray[iInnerIndex] := aiPriorityArray[iInnerIndex + 1];
        aiPriorityArray[iInnerIndex + 1] := iTempValue;

        // Фиксация факта перестановки
        xSwapsOccurred := TRUE;
      END_IF;
    END_FOR;

    // Оптимизация: если за проход не было перестановок, массив уже отсортирован
    IF NOT xSwapsOccurred THEN
      EXIT; // Досрочный выход из внешнего цикла
    END_IF;
  END_FOR;
```

Построчный разбор логики:

- Внешний цикл `FOR iOuterIndex := 0 TO 8 DO` определяет количество проходов по массиву.

- Внутренний цикл FOR iInnerIndex := 0 TO (8 - iOuterIndex) DO сокращается с каждым проходом, так как наибольшие элементы «всплывают» в конец массива.

- Условие IF aiPriorityArray[iInnerIndex] > aiPriorityArray[iInnerIndex + 1] обнаруживает пару элементов в неправильном порядке.

- Блок обмена через временную переменную iTempValue меняет элементы местами.

- Флаг xSwapsOccurred отслеживает наличие перестановок. Если за полный проход не было ни одной перестановки, массив уже отсортирован и алгоритм досрочно завершается через EXIT.

Пример 26. Двухуровневый термостат с гистерезисом

Пример реализует алгоритм управления нагревателем с двумя порогами срабатывания для предотвращения частых переключений.

Секция объявления переменных:

```
PROGRAM PLC_PRG
VAR
  // Текущая температура печи, град С
  rCurrentTemperature : REAL := 185.5;

  // Целевая температура поддержания
  rSetpoint         : REAL := 200.0;

  // Величина гистерезиса, град С
  rHysteresis       : REAL := 10.0;
```

```
  // Выходной сигнал на нагреватель
  xHeaterOutput      : BOOL := FALSE;

  // Предыдущее состояние нагревателя (память)
  xHeaterPrevState    : BOOL := FALSE;

  // Вычисленные пороги срабатывания
  rTurnOnThreshold    : REAL;
  rTurnOffThreshold   : REAL;
END_VAR
```

Исполняемый код программы:

```
// Вычисление порогов срабатывания
rTurnOnThreshold := rSetpoint - rHysteresis;   // 190.0 град C
rTurnOffThreshold := rSetpoint;                // 200.0 град C

// Алгоритм двухпозиционного регулирования с гистерезисом
IF xHeaterPrevState THEN
  // Нагреватель ВКЛЮЧЕН: отключаем только при превышении верхнего порога
  IF rCurrentTemperature >= rTurnOffThreshold THEN
    xHeaterOutput := FALSE; // Отключение по достижении уставки
  ELSE
    xHeaterOutput := TRUE;  // Продолжение нагрева
  END_IF;
ELSE
  // Нагреватель ВЫКЛЮЧЕН: включаем только при падении ниже нижнего порога
```

```
  IF rCurrentTemperature <= rTurnOnThreshold THEN
    xHeaterOutput := TRUE;  // Включение по падению температуры
  ELSE
    xHeaterOutput := FALSE; // Продолжение ожидания
  END_IF;
END_IF;

// Сохранение текущего состояния для следующего цикла
xHeaterPrevState := xHeaterOutput;
```

Построчный разбор логики:

• Порог включения rTurnOnThreshold вычисляется как уставка минус гистерезис (190 град С).

• Порог отключения rTurnOffThreshold равен уставке (200 град С).

• Если нагреватель включен (xHeaterPrevState = TRUE), он отключается только при достижении температуры 200 град С. При температуре от 190 до 200 град С нагреватель продолжает работать.

• Если нагреватель выключен (xHeaterPrevState = FALSE), он включается только при падении температуры до 190 град С. При температуре от 190 до 200 град С нагреватель остается выключенным.

• Зона от 190 до 200 град С называется зоной гистерезиса: в этой зоне состояние нагревателя зависит от его предыдущего состояния, что предотвращает частые переключения при колебаниях температуры вокруг уставки.

КОНТРОЛЬНЫЕ ВОПРОСЫ И УПРАЖНЕНИЯ К ГЛАВЕ 4

1. Объясните разницу в выполнении между каскадом IF-ELSIF-ELSE и оператором CASE-OF. В каких ситуациях предпочтительнее использовать каждый из них?

2. Почему ветвь ELSE является обязательной в операторе CASE, но необязательной в операторе IF?

3. Что произойдет с контроллером, если в программе на языке ST организовать бесконечный цикл WHILE TRUE DO ... END_WHILE;?

4. Объясните разницу между циклами WHILE-DO и REPEAT-UNTIL. Сколько раз выполнится тело каждого цикла, если условие изначально истинно?

5. Для чего используется оператор EXIT внутри цикла FOR? Приведите пример алгоритма, где его применение критически важно.

6. Что делает оператор CONTINUE и чем его действие отличается от EXIT?

7. Почему переменная-счетчик цикла FOR не может быть изменена внутри тела цикла?

8. Объясните принцип работы алгоритма пузырьковой сортировки. Почему флаг xSwapsOccurred позволяет ускорить выполнение?

9. Что такое гистерезис в системе терморегулирования и зачем он нужен?

10. Как защитный счетчик итераций uiIterationCounter предотвращает аварийный сброс контроллера?

Практические упражнения к главе 4

Упражнение 4.1. Напишите программу на языке ST, которая принимает на вход целое число `iGrade : INT` (оценка от 0 до 100) и с помощью оператора `CASE` определяет буквенную оценку по шкале: 90–100 = A, 80–89 = B, 70–79 = C, 60–69 = D, 0–59 = F. Предусмотрите обработку некорректных значений (<0 или >100).

Упражнение 4.2. Разработайте алгоритм на языке ST, который находит второй по величине элемент в массиве из 10 целых чисел `aiNumbers : ARRAY[0..9] OF INT;`. Запрещено использовать сортировку массива.

Упражнение 4.3. Напишите функцию вычисления наибольшего общего делителя (НОД) двух целых чисел методом Евклида с использованием цикла `WHILE`. Функция должна включать защиту от бесконечного цикла через счетчик итераций.

ГЛАВА 5. ПОЛЬЗОВАТЕЛЬСКИЕ ТИПЫ ДАННЫХ (DUT)

5.1. Концепция DUT и назначение пользовательских типов

Data Unit Type (DUT) представляет собой фундаментальный механизм языка Structured Text, позволяющий разработчику создавать собственные составные типы данных, адаптированные под специфику конкретной технологической задачи. В отличие от элементарных типов стандарта IEC 61131-3 (BOOL, INT, REAL), которые описывают одиночные физические величины, пользовательские типы дают возможность группировать разнородные переменные в единую логическую структуру, отражающую архитектуру реального технологического оборудования.

Необходимость в DUT возникает при проектировании сложных систем автоматизации, где требуется управлять десятками однотипных агрегатов (насосов, вентиляторов, конвейерных линий), каждый из которых обладает собственным набором параметров: дискретные сигналы состояния, аналоговые показания датчиков, счетчики моточасов, коды аварий и уставки регулирования. Объявление всех этих переменных как плоского списка в секции VAR программы приводит к экспоненциальному росту сложности проекта, затрудняет поиск нужных сигналов и провоцирует ошибки при модификации кода.

Пользовательский тип данных объявляется в отдельном объекте проекта CODESYS v3.5 через меню *Add Object -> Data Unit Type* (или *Data Type* в зависимости от версии среды). В отличие от программных блоков POU, типы данных не содержат исполняемого кода — они описывают исключительно структуру и организацию памяти. После объявления типа разработчик создает переменные этого типа в секции VAR любой программы или функционального блока, получая готовый контейнер со всеми необходимыми полями.

Ключевое преимущество DUT заключается в централизованном управлении структурой данных. Если в процессе эксплуатации объекта выясняется, что всем насосам требуется дополнительный датчик вибрации, инженер вносит одно изменение в определение типа ST_Pump, и это поле автоматически появляется во всех экземплярах насосов проекта — от главного привода до резервного агрегата.

5.2. Статические массивы ARRAY и работа с индексацией

Массив представляет собой упорядоченную коллекцию элементов одного типа данных, доступ к которым осуществляется через целочисленный индекс. В стандарте IEC 61131-3 массивы являются статическими: их размер

фиксируется на этапе компиляции и не может быть изменен в процессе выполнения программы.

Синтаксис объявления массива:

```
<Имя_переменной> : ARRAY[<Нижняя_граница>..<Верхняя_граница>] OF
<Тип_элементов>;
```

Границы индексов могут быть любыми целыми числами, включая отрицательные. Например, объявление `ARRAY[-5..5] OF REAL;` создает массив из 11 элементов с индексами от -5 до +5. Однако в промышленной практике принято использовать индексы от 0 или от 1 в соответствии с нумерацией каналов модулей ввода-вывода.

Многомерные массивы объявляются через точку с запятой между диапазонами индексов:

```
aiMatrix : ARRAY[0..3, 0..3] OF INT;  // Двумерный массив 4x4
aiCube   : ARRAY[0..2, 0..2, 0..2] OF INT;  // Трехмерный массив 3x3x3
```

Доступ к элементу массива осуществляется через квадратные скобки: `aiTemperatures[5]` обращается к шестому элементу массива (при нумерации от 0). Для многомерных массивов индексы перечисляются через запятую: `aiMatrix[2, 3]`.

Важнейшее правило безопасности: CODESYS v3.5 по умолчанию включает проверку границ массива (Check Bounds). При попытке доступа к несуществующему индексу (например, `aiArray[10]` для массива размером 0..9) генерируется исключение времени выполнения, и контроллер переходит в

состояние аварии. Отключение этой проверки повышает производительность на 10–15%, но делает программу уязвимой к фатальным ошибкам адресации памяти.

5.3. Перечислимые типы ENUM и режимы работы оборудования

Перечислимый тип данных (ENUM) представляет собой именованный набор целочисленных констант, каждая из которых имеет уникальное символьное обозначение. Этот тип данных предназначен для описания дискретных состояний технологического процесса, режимов работы машины, кодов аварий и других ситуаций, где переменная может принимать только одно значение из фиксированного набора.

Синтаксис объявления перечисления:

```
TYPE <Имя_типа> :
(
  <Элемент_1> := <Числовое_значение_1>,
  <Элемент_2> := <Числовое_значение_2>,
  <Элемент_3> := <Числовое_значение_3>
) INT;
```

Если числовые значения не указаны явно, компилятор автоматически присваивает элементам последовательные числа начиная с 0. Например, объявление (eIdle, eRunning, eFault) INT; создаст константы: eIdle = 0, eRunning = 1, eFault = 2.

Явное задание числовых значений применяется в ситуациях, когда перечисление должно соответствовать кодам состояний внешнего устройства или протокола связи. Например, частотный преобразователь может передавать

коды ошибок 16#0010, 16#0020, 16#0040, и перечисление позволяет дать этим кодам осмысленные имена.

Прагма {attribute 'qualified_only'} запрещает использование элементов перечисления без квалификатора типа. Это означает, что вместо eState := Running; программист обязан писать eState := E_MachineState.Running;, что повышает читаемость кода и исключает конфликты имен.

Прагма {attribute 'strict'} включает строгий режим проверки типов, запрещая неявное преобразование перечисления в целое число и обратно без явных операторов приведения.

5.4. Структуры данных STRUCT и точечная нотация доступа

Структура (STRUCT) представляет собой составной тип данных, объединяющий несколько разнородных переменных (полей) под единым именем. В отличие от массива, где все элементы имеют одинаковый тип, структура позволяет сгруппировать булевы флаги, целочисленные счетчики, вещественные параметры и строковые идентификаторы в единую технологическую сущность.

Синтаксис объявления структуры:

```
TYPE <Имя_типа> :
STRUCT
  <Поле_1> : <Тип_1>;
  <Поле_2> : <Тип_2>;
  <Поле_3> : <Тип_3>;
END_STRUCT;
```

Доступ к полям структуры осуществляется через точечную нотацию: stMotor.rSpeed обращается к полю rSpeed внутри структуры stMotor. Эта нотация может быть многоуровневой: stStation.stPump.rPressure извлекает давление из насоса, входящего в состав станции.

Вложенные структуры позволяют создавать иерархические модели оборудования, отражающие физическую архитектуру объекта. Например, структура ST_ProductionLine может содержать массив структур ST_WorkStation, каждая из которых включает структуру ST_Robot с полями координат, скоростей и ускорений.

Важная особенность: структура копируется как единый блок памяти при присваивании. Инструкция stTarget := stSource; копирует все поля структуры, независимо от их количества и типов. Это обеспечивает атомарность обновления сложных объектов данных.

5.5. Псевдонимы типов ALIAS и ограниченные диапазоны SUBRANGE

Псевдоним типа (ALIAS) создает новое имя для существующего базового типа без изменения его внутренней структуры. Этот механизм применяется для повышения семантической ясности кода и документирования назначения переменных.

Синтаксис объявления псевдонима:

```
TYPE <Имя_псевдонима> : ALIAS <Базовый_тип> END_TYPE;
```

Например, объявление TYPE T_Voltage : ALIAS REAL END_TYPE; позволяет объявлять переменные rSupplyVoltage : T_Voltage;, что явно указывает на

физический смысл величины (напряжение), а не просто на ее тип (вещественное число).

Ограниченный диапазон (SUBRANGE) определяет подтип целочисленного или вещественного типа с явным указанием допустимых границ значений. Компилятор использует эту информацию для автоматической проверки выхода за пределы диапазона.

Синтаксис объявления диапазона:

```
TYPE <Имя_типа> : <Базовый_тип> (<Мин_значение>..<Макс_значение>);
```

Например, `TYPE T_Percentage : INT (0..100);` создает тип для процентных величин. Присваивание значения 150 переменной этого типа вызовет ошибку компиляции или выполнение (в зависимости от настроек проверки границ).

5.6. Битовые объединения UNION и работа с перекрывающейся памятью

Объединение (UNION) представляет собой специальный тип данных, в котором все поля занимают одну и ту же область памяти. Размер объединения равен размеру его наибольшего поля. Этот механизм позволяет интерпретировать один и тот же битовый паттерн различными способами.

Синтаксис объявления объединения:

```
TYPE <Имя_типа> :
UNION
  <Поле_1> : <Тип_1>;
  <Поле_2> : <Тип_2>;
END_UNION;
```

Классическое применение UNION в промышленной автоматизации — распаковка телеграмм полевых шин. Например, 32-битное слово DWORD может одновременно трактоваться как четыре отдельных байта BYTE или как вещественное число REAL. Изменение любого из полей мгновенно отражается на всех остальных, так как они ссылаются на одну физическую память.

Важное предупреждение: использование UNION требует глубокого понимания выравнивания памяти и порядка байт (Endianness) целевой платформы. Неправильное применение объединений может привести к некорректной интерпретации данных при обмене между контроллерами разных производителей.

ПРАКТИКУМ ГЛАВЫ 5

Пример 27. Создание паспорта электропривода: структура ST_MotorData

Пример демонстрирует разработку комплексной структуры данных для описания состояния электродвигателя с интеграцией дискретных сигналов, аналоговых параметров и диагностической информации.

Объявление пользовательского типа данных в редакторе DUT:

```
ST_MotorData :
TYPE STRUCT
  // ДИСКРЕТНЫЕ СИГНАЛЫ СОСТОЯНИЯ
  xPowerOn          : BOOL;  // Питание включено
  xMotorRunning     : BOOL;  // Двигатель вращается
  xReady            : BOOL;  // Готовность привода
  xWarning          : BOOL;  // Предупреждение (не авария)
  xFault            : BOOL;  // Аварийное состояние
```

```
  // АНАЛОГОВЫЕ ПАРАМЕТРЫ
  rCurrentAmp       : REAL;  // Ток двигателя, Ампер
  rVoltageVolt      : REAL;  // Напряжение питания, Вольт
  rSpeedRpm         : REAL;  // Фактическая скорость, об/мин
  rTemperatureC     : REAL;  // Температура обмоток, град С
  rTorquePercent    : REAL;  // Момент на валу, % от номинала

  // СЧЕТЧИКИ И НАКОПИТЕЛЬНЫЕ ВЕЛИЧИНЫ
  udiOperatingHours   : UDINT; // Моточасы наработки
  udiStartCounter     : UDINT; // Количество пусков
  udiFaultCode        : UDINT; // Код последней аварии

  // УСТАВКИ И ПАРАМЕТРЫ РЕГУЛИРОВАНИЯ
  rSpeedSetpoint    : REAL;  // Заданная скорость, об/мин
  rCurrentLimit     : REAL;  // Ограничение тока, Ампер
  rAccelTime        : REAL;  // Время разгона, секунды
  rDecelTime        : REAL;  // Время торможения, секунды

  // СТРОКОВАЯ ИДЕНТИФИКАЦИЯ
  sMotorTag         : STRING(20);  // Бирочное имя двигателя
  sManufacturer     : STRING(30);  // Производитель
END_STRUCT;
END_TYPE
```

Секция объявления переменных программы:

```
PROGRAM PLC_PRG
VAR
```

```
  // Экземпляр структуры паспорта двигателя
  stMainMotor       : ST_MotorData;

  // Локальная переменная для временных вычислений
  rPowerFactor      : REAL;
END_VAR
```

Исполняемый код программы:

```
// Инициализация строковых полей структуры
stMainMotor.sMotorTag := 'M-101: Feed Pump';
stMainMotor.sManufacturer := 'Siemens';

// Запись уставок регулирования
stMainMotor.rSpeedSetpoint := 1450.0;  // об/мин
stMainMotor.rCurrentLimit := 25.5;     // Ампер
stMainMotor.rAccelTime := 5.0;         // секунды
stMainMotor.rDecelTime := 3.0;         // секунды

// Моделирование показаний датчиков
stMainMotor.rCurrentAmp := 18.3;
stMainMotor.rVoltageVolt := 380.5;
stMainMotor.rSpeedRpm := 1435.0;
stMainMotor.rTemperatureC := 65.2;
stMainMotor.rTorquePercent := 72.5;

// Обновление дискретных статусов
stMainMotor.xPowerOn := TRUE;
stMainMotor.xMotorRunning := TRUE;
```

```
stMainMotor.xReady := TRUE;
stMainMotor.xWarning := FALSE;
stMainMotor.xFault := FALSE;

// Инкрементирование счетчиков (эмуляция наработки)
stMainMotor.udiOperatingHours := stMainMotor.udiOperatingHours + 1;

// Вычисление коэффициента мощности через поля структуры
IF (stMainMotor.rVoltageVolt * stMainMotor.rCurrentAmp) > 0.0 THEN
   rPowerFactor := (stMainMotor.rTorquePercent / 100.0) /
                   (stMainMotor.rVoltageVolt * stMainMotor.rCurrentAmp);
END_IF;
```

Построчный разбор логики:

- Объявление `stMainMotor : ST_MotorData;` создает в памяти программы полноценный паспорт двигателя со всеми 20 полями.
- Точечная нотация `stMainMotor.rSpeedSetpoint` обеспечивает адресацию к конкретному параметру внутри сложной структуры.
- Инициализация строковых полей `sMotorTag` и `sManufacturer` демонстрирует работу с текстовыми данными внутри структур.
- Вычисление `rPowerFactor` показывает возможность использования полей структуры в математических выражениях наравне с обычными переменными.

Пример 28. Кольцевой буфер событий на базе массива структур

Пример реализует циклический буфер для регистрации последних 10 аварийных событий с сохранением временных меток и кодов неисправностей.

Объявление пользовательского типа данных:

```
// Тип одного события в журнале
TYPE ST_EventRecord :
STRUCT
  tTimestamp       : TIME;   // Время события
  udiEventCode     : UDINT;  // Код аварии
  sDescription     : STRING(40); // Текстовое описание
END_STRUCT;
END_TYPE;

// Тип кольцевого буфера
TYPE ST_EventBuffer :
STRUCT
  arEvents         : ARRAY[0..9] OF ST_EventRecord; // 10 записей
  uiWriteIndex     : UINT;    // Индекс текущей записи
  uiEventCount     : UINT;    // Общее количество записанных событий
END_STRUCT;
END_TYPE;
```

Секция объявления переменных программы:

```
PROGRAM PLC_PRG
VAR
  // Экземпляр буфера событий
  stAlarmBuffer    : ST_EventBuffer;
```

```
    // Сигнал новой аварии от технологического оборудования
    xNewFault         : BOOL := FALSE;
    xPrevFault        : BOOL := FALSE;

    // Код текущей аварии
    udiCurrentFaultCode : UDINT := 0;

    // Временная метка события
    tCurrentTime      : TIME := T#0ms;
END_VAR
```

Исполняемый код программы:

```
// Детектор переднего фронта сигнала аварии
IF xNewFault AND NOT xPrevFault THEN
    // Инкрементирование общего счетчика событий
    stAlarmBuffer.uiEventCount := stAlarmBuffer.uiEventCount + 1;

    // Запись временной метки (в реальном проекте — от системного времени)
    tCurrentTime := T#1h20m30s500ms;
    stAlarmBuffer.arEvents[stAlarmBuffer.uiWriteIndex].tTimestamp :=
tCurrentTime;

    // Запись кода аварии
    stAlarmBuffer.arEvents[stAlarmBuffer.uiWriteIndex].udiEventCode :=
udiCurrentFaultCode;

    // Формирование текстового описания
```

```
    stAlarmBuffer.arEvents[stAlarmBuffer.uiWriteIndex].sDescription :=
        CONCAT('Fault code: ', UDINT_TO_STRING(udiCurrentFaultCode));

    // Циклическое обновление индекса записи
    stAlarmBuffer.uiWriteIndex := stAlarmBuffer.uiWriteIndex + 1;

    // Сброс индекса в ноль при достижении конца массива (кольцевая струк-
тура)
    IF stAlarmBuffer.uiWriteIndex > 9 THEN
        stAlarmBuffer.uiWriteIndex := 0;
    END_IF;
END_IF;

xPrevFault := xNewFault;
```

Построчный разбор логики:

• Вложенная структура ST_EventBuffer содержит массив из 10 элементов типа ST_, каждый из которых в свою очередь состоит из трех полей.

• Обращение EventRecord stAlarmBuffer.arEvents[stAlarmBuffer.uiWriteIndex].tTimestamp демонстрирует многоуровневую адресацию: буфер -> массив -> элемент по индексу -> поле записи.

• Условие IF stAlarmBuffer.uiWriteIndex > 9 THEN реализует кольцевую структуру: после записи в десятую ячейку (индекс 9) указатель возвращается к началу массива, затирая oldest событие.

• Счетчик uiEventCount продолжает расти неограниченно, позволяя определить общее количество зарегистрированных аварий за все время работы.

Пример 29. Матрица рецептурного дозирования: двумерный массив

Пример показывает организацию двумернИсполняемый код программы:

```
// Инициализация названий компонентов
stRecipeData.asComponentNames[1] := 'Base Polymer';
stRecipeData.asComponentNames[2] := 'Plasticizer';
stRecipeData.asComponentNames[3] := 'Stabilizer';
stRecipeData.asComponentNames[4] := 'Colorant';
stRecipeData.asComponentNames[5] := 'Filler';

// Инициализация названий продуктов
stRecipeData.asProductNames[1] := 'Product A - Standard';
stRecipeData.asProductNames[2] := 'Product B - High Impact';
stRecipeData.asProductNames[3] := 'Product C - UV Resistant';

// Заполнение матрицы рецептур для Продукта 1
stRecipeData.arRecipe[1, 1] := 850.0;  // Base Polymer: 850 г
stRecipeData.arRecipe[1, 2] := 100.0;  // Plasticizer: 100 г
stRecipeData.arRecipe[1, 3] := 25.0;   // Stabilizer: 25 г
stRecipeData.arRecipe[1, 4] := 15.0;   // Colorant: 15 г
stRecipeData.arRecipe[1, 5] := 10.0;   // Filler: 10 г

// Заполнение матрицы рецептур для Продукта 2
stRecipeData.arRecipe[2, 1] := 800.0;
```

```
stRecipeData.arRecipe[2, 2] := 150.0;
stRecipeData.arRecipe[2, 3] := 25.0;
stRecipeData.arRecipe[2, 4] := 10.0;
stRecipeData.arRecipe[2, 5] := 15.0;

// Установка активного продукта
stRecipeData.uiActiveProduct := 1;

// Чтение дозы текущего компонента для активного продукта
rCurrentDose := stRecipeData.arRecipe[stRecipeData.uiActiveProduct, 3];
// Результат: rCurrentDose = 25.0 г (Stabilizer для Продукта 1)
```

Построчный разбор логики:

- Двумерный массив `arRecipe[1..10, 1..5]` организует таблицу 10 строк (продукты) на 5 столбцов (компоненты).
- Доступ `stRecipeData.arRecipe[1, 2]` обращается к ячейке на пересечении первой строки и второго столбца.
- Массивы строк `asComponentNames` и `asProductNames` обеспечивают текстовую привязку числовых рецептур к технологическим наименованиям.
- Переменная `uiActiveProduct` выступает селектором текущей строки матрицы для дозирования.

Пример 30. Описание режимов работы машины через перечисление E_MachineMode

Пример демонстрирует применение типизированного перечисления для безопасного описания состояний автоматической линии.

Объявление пользовательского типа данных:

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_MachineMode :
(
   eMode_Off := 0,          // Питание отключено
   eMode_Initializing := 1,  // Инициализация системы
   eMode_Standby := 2,      // Ожидание команды
   eMode_Manual := 3,       // Ручной режим наладки
   eMode_Automatic := 4,    // Автоматическая работа
   eMode_Error := 5,        // Аварийное состояние
   eMode_Maintenance := 6    // Режим технического обслуживания
) INT;
END_TYPE
```

Секция объявления переменных программы:

```
PROGRAM PLC_PRG
VAR
   // Переменная состояния машины
   eCurrentMode       : E_MachineMode := E_MachineMode.eMode_Off;

   // Выходные сигналы
   xMainContactor     : BOOL := FALSE;
   xWarningLamp       : BOOL := FALSE;
   xAlarmBuzzer       : BOOL := FALSE;

   // Флаг запроса перехода в автоматический режим
   xAutoModeRequest    : BOOL := FALSE;
```

```
  // Флаг готовности всех систем
  xAllSystemsReady    : BOOL := TRUE;
END_VAR
```

Исполняемый код программы:

```
// Детерминированное управление по режимам через CASE
CASE eCurrentMode OF
  E_MachineMode.eMode_Off:
    // Полное обесточивание оборудования
    xMainContactor := FALSE;
    xWarningLamp := FALSE;
    xAlarmBuzzer := FALSE;

    // Автоматический переход в инициализацию при подаче питания
    IF xAllSystemsReady THEN
      eCurrentMode := E_MachineMode.eMode_Initializing;
    END_IF;

  E_MachineMode.eMode_Initializing:
    // Кратковременная проверка систем
    xWarningLamp := TRUE; // Мигание лампы
    xAlarmBuzzer := FALSE;

    // Переход в режим ожидания после проверки
    eCurrentMode := E_MachineMode.eMode_Standby;

  E_MachineMode.eMode_Standby:
```

```
    // Ожидание команды оператора
    xMainContactor := FALSE;
    xWarningLamp := TRUE; // Постоянное свечение
    xAlarmBuzzer := FALSE;

    // Переход в автоматический режим по запросу
    IF xAutoModeRequest AND xAllSystemsReady THEN
        eCurrentMode := E_MachineMode.eMode_Automatic;
    END_IF;

E_MachineMode.eMode_Automatic:
    // Полная рабочая готовность
    xMainContactor := TRUE;
    xWarningLamp := FALSE;
    xAlarmBuzzer := FALSE;

    // Аварийный переход при потере готовности
    IF NOT xAllSystemsReady THEN
        eCurrentMode := E_MachineMode.eMode_Error;
    END_IF;

E_MachineMode.eMode_Error:
    // Аварийное отключение с звуковой сигнализацией
    xMainContactor := FALSE;
    xWarningLamp := TRUE;
    xAlarmBuzzer := TRUE;

ELSE
    // Обработка некорректного значения (защита)
```

```
    eCurrentMode := E_MachineMode.eMode_Error;
    xMainContactor := FALSE;
    xAlarmBuzzer := TRUE;
END_CASE;
```

Построчный разбор логики:

• Прагмы {attribute 'qualified_only'} и {attribute 'strict'} требуют использования полного имени E_MachineMode.eMode_Off, что исключает ошибки чтения кода.

• Оператор CASE eCurrentMode OF обеспечивает явное разделение алгоритмов для каждого режима.

• Присваивание eCurrentMode := E_MachineMode.eMode_Initializing; меняет состояние автомата, что в следующем цикле сканирования приведет к выполнению другой ветви CASE.

• Ветвь ELSE обрабатывает гипотетические некорректные значения переменной режима (например, 99 при повреждении памяти).

Пример 31. Быстрая распаковка телеграммы Modbus через UNION

Пример иллюстрирует эффективное извлечение отдельных байт и битов из 32-битного регистра данных.

Объявление пользовательского типа данных:

```
TYPE UT_ModbusRegister :
UNION
  // Представление как единое 32-битное слово
  dwValue          : DWORD;
```

```
  // Представление как четыре отдельных байта
  arBytes          : ARRAY[0..3] OF BYTE;

  // Представление как два 16-битных слова
  arWords          : ARRAY[0..1] OF WORD;

  // Представление как вещественное число (IEEE 754)
  rFloatValue      : REAL;
END_UNION;
END_TYPE
```

Секция объявления переменных программы:

```
PROGRAM PLC_PRG
VAR
  // Экземпляр объединения для работы с регистром
  utRegister       : UT_ModbusRegister;

  // Извлеченные компоненты
  bByte0           : BYTE;
  bByte1           : BYTE;
  wWord0           : WORD;
  rSensorValue     : REAL;

  // Флаги состояния из младшего байта
  xBit0            : BOOL;
  xBit1            : BOOL;
```

```
  xBit2          : BOOL;
END_VAR
```

Исполняемый код программы:

```
// Запись сырого значения из сети Modbus (эмуляция)
utRegister.dwValue := 16#42C80000;

// ЧТЕНИЕ ЧЕРЕЗ РАЗЛИЧНЫЕ ПРЕДСТАВЛЕНИЯ ПАМЯТИ:

// Извлечение байтов через массив
bByte0 := utRegister.arBytes[0];  // Младший байт: 16#00
bByte1 := utRegister.arBytes[1];  // Второй байт: 16#00

// Извлечение 16-битных слов
wWord0 := utRegister.arWords[0];  // Младшее слово: 16#0000

// Интерпретация как вещественное число IEEE 754
rSensorValue := utRegister.rFloatValue;  // Результат: 100.0

// Побитовое чтение флагов из младшего байта
xBit0 := utRegister.arBytes[0].0;
xBit1 := utRegister.arBytes[0].1;
xBit2 := utRegister.arBytes[0].2;
```

Построчный разбор логики:

• Все четыре поля `dwValue`, `arBytes`, `arWords` и `rFloatValue` занимают одну и ту же область памяти размером 4 байта.

• Запись в utRegister.dwValue := 16#42C80000; мгновенно отражается на всех остальных полях объединения.

• Чтение utRegister.rFloatValue интерпретирует тот же битовый паттерн 16#42C80000 как вещественное число стандарта IEEE 754, давая значение 100.0.

• Массив arBytes[0..3] обеспечивает побайтовую адресацию для работы с отдельными октетами телеграммы.

Пример 32. Функция инициализации полей структуры значениями по умолчанию

Пример демонстрирует создание универсальной функции для сброса структуры в заводские настройки.

Объявление пользовательского типа данных:

```
TYPE ST_PumpConfig :
STRUCT
  xEnabled          : BOOL;
  rMaxPressure      : REAL;
  rMinPressure      : REAL;
  tStartDelay       : TIME;
  udiMaxStartsPerHour : UINT;
  sPumpName         : STRING(30);
END_STRUCT;
END_TYPE;
```

Секция объявления переменных программы:

```
PROGRAM PLC_PRG
VAR
  // Три экземпляра конфигурации насосов
  stPump1          : ST_PumpConfig;
  stPump2          : ST_PumpConfig;
  stPump3          : ST_PumpConfig;

  // Флаг запроса инициализации
  xInitializeAll   : BOOL := TRUE;
END_VAR
```

Текст функции инициализации (отдельный POU типа FUNCTION):

```
FUNCTION FB_InitPumpConfig : ST_PumpConfig
VAR_INPUT
  // Опциональный префикс имени насоса
  sNamePrefix      : STRING(10) := 'Pump';
  // Номер насоса для формирования имени
  uiPumpNumber     : UINT := 1;
END_VAR
VAR
  // Локальная структура для возврата
  stConfig         : ST_PumpConfig;
  sFullName        : STRING(30);
END_VAR

// Формирование полного имени насоса
sFullName := CONCAT(sNamePrefix, UINT_TO_STRING(uiPumpNumber));
```

```
// Инициализация всех полей структуры заводскими значениями
stConfig.xEnabled := FALSE;
stConfig.rMaxPressure := 10.0;     // бар
stConfig.rMinPressure := 2.0;      // бар
stConfig.tStartDelay := T#5s;
stConfig.udiMaxStartsPerHour := 20; // пусков в час
stConfig.sPumpName := sFullName;

// Возврат инициализированной структуры
FB_InitPumpConfig := stConfig;
```

Исполняемый код программы:

```
// Массовая инициализация всех насосов при старте системы
IF xInitializeAll THEN
   stPump1 := FB_InitPumpConfig('Pump', 1);
   stPump2 := FB_InitPumpConfig('Pump', 2);
   stPump3 := FB_InitPumpConfig('Pump', 3);

   // Сброс флага после инициализации
   xInitializeAll := FALSE;
END_IF;
```

Построчный разбор логики:

• Функция FB_InitPumpConfig принимает параметры префикса и номера насоса, формируя уникальное имя для каждого агрегата.

• Возвращаемое значение функции имеет тип ST_PumpConfig, что позволяет присваивать результат вызова функции напрямую структуре.

• Присваивание stPump1 := FB_InitPumpConfig(...) копирует всю структуру целиком, обеспечивая атомарность инициализации.

• Конкатенация CONCAT(sNamePrefix, UINT_TO_STRING(uiPumpNumber)) создает осмысленные имена 'Pump1', 'Pump2', 'Pump3'.

Пример 33. Таблица калибровочных точек датчика через массив структур

Пример реализует полиномиальную линеаризацию нелинейного датчика с использованием таблицы калибровочных точек.

Объявление пользовательского типа данных:

```
TYPE ST_CalibrationPoint :
STRUCT
  rRawADC          : REAL;   // Сырое значение АЦП
  rEngineeredValue   : REAL;   // Инженерное значение (физическая величина)
END_STRUCT;
END_TYPE;

TYPE ST_SensorCalibration :
STRUCT
  arPoints         : ARRAY[0..4] OF ST_CalibrationPoint; // 5 точек
  uiPointCount      : UINT;    // Количество заполненных точек
  sSensorTag        : STRING(20);
END_STRUCT;
END_TYPE;
```

Секция объявления переменных программы:

```
PROGRAM PLC_PRG
VAR
  // Калибровочная таблица для датчика давления
  stPressureCalibration : ST_SensorCalibration;

  // Текущее сырое значение с АЦП
  rCurrentRawADC       : REAL := 16384.0;

  // Результат линеаризации
  rCalibratedPressure   : REAL;

  // Индексы для поиска интервала
  i                : INT;
  rSlope           : REAL;
  rIntercept       : REAL;
END_VAR
```

Исполняемый код программы:

```
// Инициализация калибровочной таблицы (5 точек)
stPressureCalibration.uiPointCount := 5;
stPressureCalibration.sSensorTag := 'PT-101';

// Точка 0: 0 бар
stPressureCalibration.arPoints[0].rRawADC := 0.0;
stPressureCalibration.arPoints[0].rEngineeredValue := 0.0;

// Точка 1: 2.5 бар
```

```
stPressureCalibration.arPoints[1].rRawADC := 6898.0;
stPressureCalibration.arPoints[1].rEngineeredValue := 2.5;

// Точка 2: 5.0 бар
stPressureCalibration.arPoints[2].rRawADC := 13796.0;
stPressureCalibration.arPoints[2].rEngineeredValue := 5.0;

// Точка 3: 7.5 бар
stPressureCalibration.arPoints[3].rRawADC := 20694.0;
stPressureCalibration.arPoints[3].rEngineeredValue := 7.5;

// Точка 4: 10.0 бар
stPressureCalibration.arPoints[4].rRawADC := 27648.0;
stPressureCalibration.arPoints[4].rEngineeredValue := 10.0;

// ПОИСК ИНТЕРВАЛА И ЛИНЕЙНАЯ ИНТЕРПОЛЯЦИЯ:

// Поиск интервала, в который попадает текущее значение rCurrentRawADC
FOR i := 0 TO (INT#stPressureCalibration.uiPointCount - 2) DO
   IF (rCurrentRawADC >= stPressureCalibration.arPoints[i].rRawADC) AND
     (rCurrentRawADC <= stPressureCalibration.arPoints[i+1].rRawADC) THEN

      // Вычисление коэффициента линейной интерполяции
      rSlope := (stPressureCalibration.arPoints[i+1].rEngineeredValue -
              stPressureCalibration.arPoints[i].rEngineeredValue) /
             (stPressureCalibration.arPoints[i+1].rRawADC -
              stPressureCalibration.arPoints[i].rRawADC);

      rIntercept := stPressureCalibration.arPoints[i].rEngineeredValue -
```

```
                (rSlope * stPressureCalibration.arPoints[i].rRawADC);

    // Вычисление калиброванного значения
    rCalibratedPressure := (rSlope * rCurrentRawADC) + rIntercept;

    // Выход из цикла после нахождения интервала
    EXIT;
  END_IF;
END_FOR;
```

Построчный разбор логики:

• Массив arPoints[0..4] хранит 5 пар значений (сырое АЦП, инженерная величина), определяющих калибровочную характеристику.

• Цикл FOR последовательно проверяет, в какой интервал попадает текущее измерение rCurrentRawADC.

• Формула линейной интерполяции y = slope * x + intercept вычисляет угловой коэффициент и свободный член для найденного интервала.

• Оператор EXIT прерывает цикл сразу после нахождения подходящего интервала, экономя процессорное время.

КОНТРОЛЬНЫЕ ВОПРОСЫ И УПРАЖНЕНИЯ К ГЛАВЕ 5

1. Объясните разницу между массивом и структурой. В каких ситуациях целесообразно использовать каждый из этих типов?

2. Что произойдет при выполнении инструкции aiArray[10] := 5; для массива, объявленного как ARRAY[0..9] OF INT; при включенной проверке границ?

3. Для чего применяются прагмы {attribute 'qualified_only'} и {attribute 'strict'} в перечислимых типах?

4. Какое количество байт памяти займет структура, содержащая: BOOL, REAL, INT, BYTE? Объясните влияние выравнивания памяти.

5. Объясните принцип работы объединения UNION. Почему изменение одного поля мгновенно отражается на всех остальных?

6. В чем заключается преимущество инициализации структуры через функцию возврата по сравнению с поочередным присваиванием каждого поля?

7. Что такое кольцевой буфер и зачем нужен сброс индекса записи в ноль при достижении конца массива?

8. Как объявить двумерный массив 8x8 для хранения состояний шахматной доски?

9. Зачем в калибровочной таблице используется цикл с поиском интервала вместо прямой формулы пересчета?

10. Можно ли присвоить структуру одного типа структуре другого типа, если они имеют идентичную внутреннюю структуру?

Практические упражнения к главе 5

Упражнение 5.1. Разработайте структуру ST_TankData для описания резервуара с жидкостью, включающую: уровень (%), температуру (град С), давление (бар), 3 дискретных флага состояния (авария, готовность, перемешивание), моточасы мешалки и строковое имя резервуара. Создайте массив из 4 таких резервуаров.

Упражнение 5.2. Объявите перечислимый тип E_AlarmPriority с уровнями: Low, Medium, High, Critical. Напишите функцию, которая по коду аварии возвращает приоритет и соответствующее цветовое обозначение для панели оператора.

Упражнение 5.3. Создайте объединение UT_AnalogChannel, позволяющее интерпретировать 16-битное слово одновременно как: целое число INT, беззнаковое число UINT, два отдельных байта BYTE и 16 отдельных битов BOOL. Напишите программу демонстрации чтения одного и того же значения через разные поля.

ГЛАВА 6. ПРОГРАММНЫЕ КОМПОНЕНТЫ: ФУНКЦИИ (FUN) И ФУНКЦИОНАЛЬНЫЕ БЛОКИ (FB)

6.1. Классификация POU и архитектура модульного проекта

Программная организационная единица (Program Organization Unit, POU) представляет собой автономный программный компонент стандарта IEC 61131-3, инкапсулирующий определенную технологическую логику и данные. Модульная архитектура проекта является обязательным требованием для создания поддерживаемых, масштабируемых и надежных систем промышленной автоматизации. Разбиение монолитной программы на специализированные POU позволяет изолировать функциональность, упростить отладку отдельных узлов и организовать многократное использование проверенных алгоритмов.

Стандарт определяет три фундаментальных типа POU, каждый из которых имеет строго очерченную область применения и жизненный цикл в памяти контроллера:

Программа (PROGRAM) — это POU верхнего уровня, не имеющая входных и выходных параметров в классическом понимании. Программа напрямую оперирует глобальными переменными проекта и образами процесса ввода-вывода. В каждом проекте CODESYS v3.5 автоматически создается программа PLC_PRG, которая выступает точкой входа и главным циклом управления. Программы не могут быть вызваны из других программных блоков — они вызываются исключительно системным планировщиком задач в соответствии с конфигурацией Task.

Функция (FUNCTION) — это POU, предназначенная для выполнения детерминированных вычислений без сохранения внутреннего состояния между вызовами. Функция обязательно возвращает единственное значение через свое имя и не может иметь внутренних статических переменных, сохраняющих значение после завершения работы. Функции идеальны для математических преобразований, логических операций, масштабирования сигналов и любых расчетов, где результат зависит исключительно от входных параметров.

Функциональный блок (FUNCTION_BLOCK) — это POU, обладающая внутренней памятью состояния и способная хранить данные между последовательными вызовами. Каждый экземпляр функционального блока занимает в памяти контроллера область, равную сумме размеров всех его переменных. Функциональные блоки являются фундаментом объектно-ориентированного подхода в автоматизации: они моделируют физические агрегаты (насосы, клапаны, приводы) с их технологической памятью (счетчики, таймеры, флаги готовности).

6.2. Анатомия функции и правила разработки

Функция в стандарте IEC 61131-3 подчиняется принципу детерминизма: при одинаковых входных параметрах она всегда возвращает одинаковый результат, независимо от того, сколько раз и в какой момент времени она была вызвана. Это свойство делает функции предсказуемыми и легкими для тестирования компонентами.

Секция объявления функции всегда начинается с ключевого слова FUNCTION, за которым следует имя возвращаемого типа и собственное имя функции:

```
FUNCTION <Имя_функции> : <Тип_возвращаемого_значения>
```

Входные параметры функции объявляются в секции VAR_INPUT и передаются по значению. Это означает, что функция получает копию данных, и любые изменения входных переменных внутри функции не влияют на оригинальные переменные вызывающего кода.

Выходные параметры в классическом понимании у функций отсутствуют. Единственный способ вернуть результат — присваивание имени самой функции. Присвоить имя функции можно многократно в теле функции, но итоговое возвращаемое значение будет равно тому, которое было записано последним перед завершением работы.

Временные переменные VAR_TEMP размещаются на системном стеке и уничтожаются после выхода из функции. Они не сохраняют значение между вызовами и не требуют явной инициализации.

Важное ограничение: внутри тела функции запрещено использование операторов RETURN с возвратом значения в старых версиях стандарта, однако CODESYS v3.5 поддерживает современный синтаксис с явным возвратом.

6.3. Анатомия функционального блока и жизненный цикл экземпляра

Функциональный блок представляет собой программный объект с внутренним состоянием, которое сохраняется между вызовами. Это фундаментальное отличие от функций делает функциональные блоки пригодными для моделирования динамических систем с памятью: таймеров, счетчиков, интеграторов, триггеров и регуляторов.

Объявление функционального блока начинается с ключевого слова FUNCTION_BLOCK:

```
FUNCTION_BLOCK <Имя_блока>
```

Секция VAR_INPUT объявляет входные параметры, которые передаются в блок из внешнего кода при каждом вызове. Эти переменные доступны только для чтения внутри блока.

Секция VAR_OUTPUT определяет выходные параметры, через которые блок передает результаты вычислений во внешний мир. Присваивание значений этим переменным внутри блока обновляет соответствующие переменные вызывающего кода.

Секция VAR_IN_OUT объявляет двунаправленные параметры, передаваемые по ссылке (адресу памяти). Это означает, что блок получает не копию данных, а прямой доступ к памяти внешней переменой. Любое изменение

переменной VAR_IN_OUT внутри блока немедленно отражается на оригинальной переменной. Этот механизм критически важен для эффективной передачи крупных массивов и структур без накладных расходов на копирование.

Секция VAR определяет внутренние статические переменные блока, которые сохраняют свои значения между вызовами. Именно здесь размещаются счетчики, таймеры, флаги состояния и буферы данных, составляющие «память» функционального блока.

Секция VAR_TEMP объявляет временные переменные, которые создаются заново при каждом входе в блок и уничтожаются при выходе. Они не сохраняют предысторию и используются для промежуточных вычислений.

Жизненный цикл экземпляра функционального блока включает три фазы:

1. **Создание экземпляра:** при объявлении переменной типа функционального блока в секции VAR программы компилятор резервирует в памяти процесса область, достаточную для хранения всех внутренних переменных блока.

2. **Инициализация:** при первом запуске контроллера или перезагрузке проекта все переменные блока инициализируются значениями по умолчанию или явными начальными значениями из объявления.

3. **Циклическое выполнение:** при каждом вызове экземпляра в программе блок выполняет свой алгоритм, обновляет внутренние переменные и формирует выходы.

6.4. Размещение экземпляров в памяти и накладные расходы

Каждый экземпляр функционального блока занимает в оперативной памяти контроллера непрерывную область, размер которой равен сумме размеров всех его переменных всех категорий (VAR_INPUT, VAR_OUTPUT, VAR, VAR_IN_OUT). Переменные VAR_TEMP не учитываются в этом размере, так как размещаются на системном стеке выполнения.

При объявлении в программе десяти экземпляров одного и того же функционального блока (например, десяти насосов fbPump1 … fbPump10) контроллер резервирует десять независимых областей памяти, каждая из которых содержит полный набор внутренних переменных. Это обеспечивает полную изоляцию агрегатов: изменение внутреннего счетчика одного насоса никак не влияет на счетчик другого.

Накладные расходы на экземпляр включают не только размер данных, но и время вызова. Каждый вызов функционального блока требует сохранения контекста выполнения, передачи параметров и восстановления контекста после возврата. Для высокоскоростных задач с циклом 1 мс создание сотен экземпляров сложных блоков может привести к превышению допустимого времени выполнения.

Оптимизация памяти достигается через:

- Использование VAR_IN_OUT для передачи крупных структур и массивов по ссылке вместо копирования.
- Минимизацию количества внутренних переменных VAR за счет переиспользования временных ячеек.
- Применение прагм упаковки структур для устранения выравнивания памяти.

6.5. Передача параметров по ссылке и значению

Понимание механизма передачи параметров является ключом к эффективному проектированию интерфейсов функциональных блоков.

Передача по значению (Pass by Value) применяется к параметрам VAR_INPUT. При вызове блока создается копия данных вызывающей переменной, и эта копия помещается в область памяти входа блока. Изменение входной переменой внутри блока не влияет на оригинал. Этот механизм безопасен, но неэффективен для крупных массивов: копирование структуры размером 100 байт требует 100 операций записи в память.

Передача по ссылке (Pass by Reference) применяется к параметрам VAR_IN_OUT. Блок получает не копию, а адрес (указатель) на память оригинальной переменной. Любое обращение к параметру VAR_IN_OUT внутри блока напрямую читает или пишет в память вызывающего кода. Это обеспечивает нулевые накладные расходы на копирование и возможность модификации внешних данных. Однако требуется осторожность: случайное изменение входного массива внутри блока может привести к трудноотлавливаемым ошибкам.

Выходные параметры VAR_OUTPUT передаются по механизму, аналогичному VAR_IN_OUT, но с семантикой только записи. Блок обязан присвоить значение каждому выходу перед завершением, иначе вызывающий код получит неопределенные данные.

6.6. Стандартные функциональные блоки библиотек

Среда CODESYS v3.5 предоставляет обширную библиотеку стандартных функциональных блоков, реализующих типовые алгоритмы автоматизации.

Эти блоки протестированы, оптимизированы и полностью соответствуют стандарту IEC 61131-3.

Таймеры (Timers):

• TON (Timer On Delay) — таймер задержки включения. Выход Q становится TRUE через заданное время PT после появления TRUE на входе IN.

• TOF (Timer Off Delay) — таймер задержки выключения. Выход Q остается TRUE в течение времени PT после исчезновения сигнала IN.

• TP (Pulse Timer) — таймер импульса. Формирует импульс фиксированной длительности PT при каждом переднем фронте входа IN.

Счетчики (Counters):

• CTU (Count Up) — счетчик инкремента. Увеличивает значение CV на единицу при каждом переднем фронте входа CU.

• CTD (Count Down) — счетчик декремента. Уменьшает значение CV на единицу при каждом переднем фронте входа CD.

• CTUD (Count Up/Down) — реверсивный счетчик с двумя входами импульсов.

Триггеры (Edges):

• R_TRIG (Rising Edge Trigger) — детектор переднего фронта. Формирует единичный импульс на выходе Q при переходе входа CLK из FALSE в TRUE.

• F_TRIG (Falling Edge Trigger) — детектор заднего фронта. Формирует импульс при переходе CLK из TRUE в FALSE.

Математические блоки:

• ADD, SUB, MUL, DIV — арифметические операции.

- MIN, MAX, LIMIT — операции сравнения и ограничения.
- SEL, MUX — операции выбора.

ПРАКТИКУМ ГЛАВЫ 6

Пример 34. Чистая математическая функция: полиномиальное масштабирование аналогового сигнала

Пример демонстрирует создание функции для нелинейного масштабирования сигнала датчика по полиному второй степени.

Текст функции (отдельный POU типа FUNCTION):

```
FUNCTION FUN_ScalePolynomial : REAL
VAR_INPUT
  // Сырое значение АЦП (0..27648)
  rRawInput       : REAL;

  // Коэффициенты полинома y = a*x^2 + b*x + c
  rCoefficientA    : REAL;
  rCoefficientB    : REAL;
  rCoefficientC    : REAL;
END_VAR
VAR
  // Промежуточное значение квадрата входа
  rSquaredInput    : REAL;
END_VAR

// Вычисление квадрата входного сигнала
rSquaredInput := rRawInput * rRawInput;
```

```
// Вычисление полиномиальной зависимости
FUN_ScalePolynomial := (rCoefficientA * rSquaredInput) +
                       (rCoefficientB * rRawInput) +
                       rCoefficientC;
```

Секция объявления переменных программы:

```
PROGRAM PLC_PRG
VAR
  // Сырой сигнал датчика давления
  rSensorRawADC      : REAL := 13824.0;

  // Калиброванное давление
  rCalibratedPressure : REAL;

  // Коэффициенты калибровки (определяются экспериментально)
  rCoeffA            : REAL := 0.0000001;
  rCoeffB            : REAL := 0.0005;
  rCoeffC            : REAL := -0.5;
END_VAR
```

Исполняемый код программы:

```
// Вызов функции масштабирования
rCalibratedPressure := FUN_ScalePolynomial(
  rRawInput := rSensorRawADC,
  rCoefficientA := rCoeffA,
  rCoefficientB := rCoeffB,
```

```
  rCoefficientC := rCoeffC
);
```

Построчный разбор логики:

• Функция FUN_ScalePolynomial объявляет возвращаемый тип REAL, что позволяет использовать ее в математических выражениях.

• Все входные параметры объявлены в VAR_INPUT и передаются по значению.

• Временная переменная rSquaredInput хранит промежуточный результат возведения в квадрат.

• Присваивание FUN_ScalePolynomial := ... формирует возвращаемое значение функции.

• Вызов функции в программе использует явное именование параметров для повышения читаемости.

Пример 35. Логическая функция контроля четности байта

Пример реализует функцию вычисления бита четности (Parity Bit) для обнаружения ошибок передачи данных.

Текст функции:

```
FUNCTION FUN_ParityCheck : BOOL
VAR_INPUT
  // Проверяемый байт данных
  bDataByte       : BYTE;
END_VAR
VAR
```

```
  // Счетчик единичных битов
  iBitCount       : INT;
  iBitIndex       : INT;
END_VAR

// Инициализация счетчика
iBitCount := 0;

// Подсчет количества единичных битов в байте
FOR iBitIndex := 0 TO 7 DO
  IF bDataByte.iBitIndex THEN
    iBitCount := iBitCount + 1;
  END_IF;
END_FOR;

// Четность: TRUE если количество единиц нечетное (нечетный паритет)
// FALSE если количество единиц четное (четный паритет)
FUN_ParityCheck := (iBitCount MOD 2) <> 0;
```

Секция объявления переменных программы:

```
PROGRAM PLC_PRG
VAR
  // Тестируемые байты
  bTestByte1       : BYTE := 16#55; // 2#01010101 — 4 единицы
  bTestByte2       : BYTE := 16#0F; // 2#00001111 — 4 единицы
  bTestByte3       : BYTE := 16#87; // 2#10000111 — 4 единицы

  // Результаты проверки
```

```
  xParity1        : BOOL;
  xParity2        : BOOL;
  xParity3        : BOOL;
END_VAR
```

Исполняемый код программы:

```
// Вызов функции для разных байтов
xParity1 := FUN_ParityCheck(bDataByte := bTestByte1);
xParity2 := FUN_ParityCheck(bDataByte := bTestByte2);
xParity3 := FUN_ParityCheck(bDataByte := bTestByte3);
```

Построчный разбор логики:

• Функция подсчитывает количество единичных битов в байте через цикл FOR с побитовой адресацией.

• Оператор MOD 2 определяет четность счетчика.

• Возвращаемое значение TRUE указывает на нечетное количество единиц, FALSE — на четное.

• Функция не имеет внутреннего состояния и полностью детерминирована.

Пример 36. Функциональный блок реверсивного пускателя FB_MotorStarter

Пример реализует функциональный блок управления реверсивным пускателем двигателя с блокировкой одновременного включения прямого и обратного хода.

Текст функционального блока:

```
FUNCTION_BLOCK FB_MotorStarter
VAR_INPUT
  // Команда пуска вперед
  xCmdForward      : BOOL;

  // Команда пуска назад
  xCmdReverse      : BOOL;

  // Команда остановки
  xCmdStop         : BOOL;

  // Сигнал аварийного останова (NC контакт)
  xEmergencyStop   : BOOL;

  // Сигнал тепловой защиты (NC контакт)
  xThermalFault    : BOOL;
END_VAR
VAR_OUTPUT
  // Состояние контактора вперед
  xContactorFwd    : BOOL;

  // Состояние контактора назад
  xContactorRev    : BOOL;

  // Флаг готовности
  xReady           : BOOL;
```

```
  // Флаг аварии
  xFault          : BOOL;
END_VAR
VAR
  // Внутреннее состояние самоподхвата вперед
  xLatchForward    : BOOL := FALSE;

  // Внутреннее состояние самоподхвата назад
  xLatchReverse    : BOOL := FALSE;
END_VAR

// Проверка аварийных условий
IF NOT xEmergencyStop OR xThermalFault THEN
  // Авария: обесточивание обоих контакторов
  xContactorFwd := FALSE;
  xContactorRev := FALSE;
  xLatchForward := FALSE;
  xLatchReverse := FALSE;
  xReady := FALSE;
  xFault := TRUE;
ELSE
  // Нормальный режим работы
  xFault := FALSE;
  xReady := TRUE;

  // Логика прямого хода с самоподхватом
  IF xCmdForward AND NOT xLatchReverse AND NOT xCmdStop THEN
    xLatchForward := TRUE;
  ELSIF xCmdStop OR xCmdReverse THEN
```

```
    xLatchForward := FALSE;
  END_IF;

  // Логика обратного хода с самоподхватом
  IF xCmdReverse AND NOT xLatchForward AND NOT xCmdStop THEN
    xLatchReverse := TRUE;
  ELSIF xCmdStop OR xCmdForward THEN
    xLatchReverse := FALSE;
  END_IF;

  // Выходные контакторы
  xContactorFwd := xLatchForward;
  xContactorRev := xLatchReverse;
END_IF;
```

Секция объявления переменных программы:

```
PROGRAM PLC_PRG
VAR
  // Экземпляр функционального блока пускателя
  fbMotorStarter     : FB_MotorStarter;

  // Сигналы управления от кнопок
  xBtnFwd          : BOOL := FALSE;
  xBtnRev          : BOOL := FALSE;
  xBtnStop         : BOOL := FALSE;
  xEStop           : BOOL := TRUE;
  xThermal         : BOOL := TRUE;
```

```
END_VAR
```

Исполняемый код программы:

```
// Вызов функционального блока с передачей параметров
fbMotorStarter(
  xCmdForward := xBtnFwd,
  xCmdReverse := xBtnRev,
  xCmdStop := xBtnStop,
  xEmergencyStop := xEStop,
  xThermalFault := xThermal
);

// Чтение выходов блока
IF fbMotorStarter.xFault THEN
  // Обработка аварии
END_IF;
```

Построчный разбор логики:

• Внутренние переменные `xLatchForward` и `xLatchReverse` хранят состояние самоподхвата между вызовами блока.

• Блокировка `NOT xLatchReverse` в логике прямого хода предотвращает одновременное включение контакторов.

• Аварийные входы `xEmergencyStop` и `xThermalFault` имеют приоритет над командами пуска.

• Выходы `xContactorFwd` и `xContactorRev` передают состояние во внешний код.

Пример 37. Функциональный блок цифрового фильтра скользящего среднего FB_MovingAverage

Пример реализует блок сглаживания зашумленного аналогового сигнала методом скользящего среднего.

Текст функционального блока:

```
FUNCTION_BLOCK FB_MovingAverage
VAR_INPUT
  // Входной зашумленный сигнал
  rRawInput       : REAL;

  // Количество точек усреднения
  uiWindowSize    : UINT := 10;

  // Сигнал сброса буфера
  xReset          : BOOL;
END_VAR
VAR_OUTPUT
  // Сглаженный выходной сигнал
  rFilteredOutput  : REAL;
END_VAR
VAR
  // Циклический буфер последних значений
  arBuffer        : ARRAY[0..99] OF REAL;

  // Индекс текущей записи
  uiWriteIndex    : UINT := 0;
```

```
  // Количество заполненных ячеек
  uiValidCount      : UINT := 0;

  // Накопленная сумма для быстрого расчета среднего
  rAccumulator      : REAL := 0.0;

  // Предыдущее значение reset
  xPrevReset        : BOOL := FALSE;
END_VAR
VAR_TEMP
  i                 : INT;
  rOldValue         : REAL;
END_VAR

// Детектор сброса буфера
IF xReset AND NOT xPrevReset THEN
  // Очистка буфера и аккумуляторов
  FOR i := 0 TO 99 DO
    arBuffer[i] := 0.0;
  END_FOR;
  uiWriteIndex := 0;
  uiValidCount := 0;
  rAccumulator := 0.0;
  rFilteredOutput := 0.0;
END_IF;

xPrevReset := xReset;

// Ограничение размера окна
```

```
IF uiWindowSize > 100 THEN
   uiWindowSize := 100;
END_IF;

// Вычисление скользящего среднего
IF uiValidCount < uiWindowSize THEN
   // Буфер еще не заполнен полностью
   arBuffer[uiWriteIndex] := rRawInput;
   rAccumulator := rAccumulator + rRawInput;
   uiValidCount := uiValidCount + 1;
ELSE
   // Буфер заполнен: вычитаем старое значение, добавляем новое
   rOldValue := arBuffer[uiWriteIndex];
   arBuffer[uiWriteIndex] := rRawInput;
   rAccumulator := rAccumulator - rOldValue + rRawInput;
END_IF;

// Вычисление среднего
rFilteredOutput := rAccumulator / TO_REAL(uiValidCount);

// Циклическое обновление индекса
uiWriteIndex := uiWriteIndex + 1;
IF uiWriteIndex >= uiWindowSize THEN
   uiWriteIndex := 0;
END_IF;
```

Секция объявления переменных программы:

```
PROGRAM PLC_PRG
VAR
   // Экземпляр фильтра
   fbSignalFilter     : FB_MovingAverage;

   // Зашумленный входной сигнал
   rNoisySignal       : REAL := 0.0;

   // Сглаженный выход
   rCleanSignal       : REAL;

   // Флаг сброса
   xFilterReset       : BOOL := FALSE;
END_VAR
```

Исполняемый код программы:

```
// Вызов фильтра
fbSignalFilter(
   rRawInput := rNoisySignal,
   uiWindowSize := 20,
   xReset := xFilterReset
);

// Чтение результата
rCleanSignal := fbSignalFilter.rFilteredOutput;
```

Построчный разбор логики:

• Массив arBuffer[0..99] хранит последние 100 значений сигнала.

• Переменная rAccumulator содержит текущую сумму элементов в окне, что позволяет вычислять среднее за O(1) без цикла суммирования.

• При заполненном буфере алгоритм вычитает старое значение и добавляет новое, поддерживая актуальную сумму.

• Сброс xReset обнуляет все внутренние переменные для корректного старта.

Пример 38. Параметризуемый генератор импульсов FB_Blinker

Пример создает функциональный блок мигающего индикатора с настраиваемыми периодами включения и выключения.

Текст функционального блока:

```
FUNCTION_BLOCK FB_Blinker
VAR_INPUT
  // Разрешение работы мигалки
  xEnable         : BOOL;

  // Длительность включения, мс
  tOnTime         : TIME := T#500ms;

  // Длительность выключения, мс
  tOffTime        : TIME := T#500ms;

  // Синхронизация фазы (для группового мигания)
  xSyncPulse      : BOOL;
```

```
END_VAR
VAR_OUTPUT
  // Выходной сигнал мигания
  xBlinkOutput     : BOOL;

  // Счетчик полных циклов
  udiCycleCount    : UDINT;
END_VAR
VAR
  // Внутренний таймер
  tElapsedTime     : TIME := T#0ms;

  // Текущее состояние (TRUE = включено)
  xCurrentPhase    : BOOL := FALSE;

  // Предыдущее состояние sync
  xPrevSync        : BOOL := FALSE;
END_VAR
VAR_TEMP
  bRisingEdge      : BOOL;
END_VAR

// Детектор синхроимпульса
bRisingEdge := xSyncPulse AND NOT xPrevSync;
xPrevSync := xSyncPulse;

IF bRisingEdge THEN
  // Синхронизация фазы по внешнему импульсу
  tElapsedTime := T#0ms;
```

```
    xCurrentPhase := TRUE;
END_IF;

IF NOT xEnable THEN
    // Выключено: выход в 0, таймер в 0
    xBlinkOutput := FALSE;
    tElapsedTime := T#0ms;
    xCurrentPhase := FALSE;
ELSE
    // Работающий режим
    tElapsedTime := tElapsedTime + T#10ms; // Эмуляция шага времени

    IF xCurrentPhase THEN
        // Фаза включения
        xBlinkOutput := TRUE;

        IF tElapsedTime >= tOnTime THEN
            // Переход в фазу выключения
            xCurrentPhase := FALSE;
            tElapsedTime := T#0ms;
        END_IF;
    ELSE
        // Фаза выключения
        xBlinkOutput := FALSE;

        IF tElapsedTime >= tOffTime THEN
            // Переход в фазу включения
            xCurrentPhase := TRUE;
            tElapsedTime := T#0ms;
```

```
        udiCycleCount := udiCycleCount + 1;
      END_IF;
   END_IF;
END_IF;
```

Секция объявления переменных программы:

```
PROGRAM PLC_PRG
VAR
   // Три независимых мигающих индикатора
   fbWarningLamp      : FB_Blinker;
   fbAlarmLamp        : FB_Blinker;
   fbStatusLamp       : FB_Blinker;

   // Общий синхроимпульс
   xSyncClock         : BOOL := FALSE;
END_VAR
```

Исполняемый код программы:

```
// Вызов блоков с разными уставками
fbWarningLamp(
   xEnable := TRUE,
   tOnTime := T#200ms,
   tOffTime := T#200ms,
   xSyncPulse := xSyncClock
);

fbAlarmLamp(
```

```
  xEnable := TRUE,
  tOnTime := T#100ms,
  tOffTime := T#100ms,
  xSyncPulse := xSyncClock
);

fbStatusLamp(
  xEnable := TRUE,
  tOnTime := T#1s,
  tOffTime := T#1s,
  xSyncPulse := xSyncClock
);
```

Построчный разбор логики:

• Переменная xCurrentPhase хранит текущее состояние мигалки между вызовами.

• Таймер tElapsedTime отсчитывает длительность текущей фазы.

• Синхроимпульс xSyncPulse позволяет синхронизировать фазу нескольких мигалок.

• Счетчик udiCycleCount фиксирует количество полных циклов мигания.

Пример 39. Вложенные вызовы: задвижка FB_Valve с внутренними таймерами

Пример демонстрирует композицию функциональных блоков: блок задвижки содержит внутри себя экземпляры стандартных таймеров.

Текст функционального блока:

```
FUNCTION_BLOCK FB_Valve
VAR_INPUT
  // Команда открытия
  xCmdOpen        : BOOL;

  // Команда закрытия
  xCmdClose       : BOOL;

  // Разрешение работы
  xEnable         : BOOL;

  // Время полного хода, мс
  tStrokeTime     : TIME := T#5s;
END_VAR
VAR_OUTPUT
  // Состояние: полностью открыто
  xFullyOpen      : BOOL;

  // Состояние: полностью закрыто
  xFullyClosed    : BOOL;

  // Состояние: в движении
  xMoving         : BOOL;

  // Процент открытия (0..100)
  rOpenPercent    : REAL;
END_VAR
VAR
  // Внутренние таймеры движения
```

```
  fbOpenTimer       : TON;
  fbCloseTimer      : TON;

  // Внутреннее состояние
  xValveOpen        : BOOL := FALSE;
END_VAR
VAR_TEMP
  rElapsedPercent   : REAL;
END_VAR

IF NOT xEnable THEN
  // Запрет работы: стоп и сброс
  fbOpenTimer(IN := FALSE);
  fbCloseTimer(IN := FALSE);
  xMoving := FALSE;
  xFullyOpen := FALSE;
  xFullyClosed := FALSE;
  rOpenPercent := 0.0;
ELSE
  // Логика открытия
  IF xCmdOpen AND NOT xCmdClose AND NOT xValveOpen THEN
    fbOpenTimer(IN := TRUE, PT := tStrokeTime);
    xMoving := TRUE;
  ELSE
    fbOpenTimer(IN := FALSE);
  END_IF;

  // Логика закрытия
  IF xCmdClose AND NOT xCmdOpen AND xValveOpen THEN
```

```
    fbCloseTimer(IN := TRUE, PT := tStrokeTime);
    xMoving := TRUE;
ELSE
    fbCloseTimer(IN := FALSE);
END_IF;

// Завершение открытия
IF fbOpenTimer.Q THEN
    xValveOpen := TRUE;
    xMoving := FALSE;
END_IF;

// Завершение закрытия
IF fbCloseTimer.Q THEN
    xValveOpen := FALSE;
    xMoving := FALSE;
END_IF;

// Вычисление процента открытия
IF xMoving THEN
    IF xCmdOpen THEN
        rElapsedPercent := TO_REAL(TIME_TO_UDINT(fbOpenTimer.ET)) /
                           TO_REAL(TIME_TO_UDINT(tStrokeTime)) * 100.0;
        rOpenPercent := rElapsedPercent;
    ELSE
        rElapsedPercent := TO_REAL(TIME_TO_UDINT(fbCloseTimer.ET)) /
                           TO_REAL(TIME_TO_UDINT(tStrokeTime)) * 100.0;
        rOpenPercent := 100.0 - rElapsedPercent;
    END_IF;
```

```
  ELSIF xValveOpen THEN
    rOpenPercent := 100.0;
  ELSE
    rOpenPercent := 0.0;
  END_IF;

  // Выходные флаги
  xFullyOpen := xValveOpen AND NOT xMoving;
  xFullyClosed := NOT xValveOpen AND NOT xMoving;
END_IF;
```

Секция объявления переменных программы:

```
PROGRAM PLC_PRG
VAR
  // Экземпляр задвижки
  fbMainValve       : FB_Valve;

  // Команды управления
  xOpenCmd          : BOOL := FALSE;
  xCloseCmd         : BOOL := FALSE;
END_VAR
```

Исполняемый код программы:

```
// Вызов задвижки
fbMainValve(
  xCmdOpen := xOpenCmd,
  xCmdClose := xCloseCmd,
```

```
  xEnable := TRUE,
  tStrokeTime := T#10s
);
```

Построчный разбор логики:

• Внутренние экземпляры fbOpenTimer и fbCloseTimer являются полноценными таймерами TON со своей памятью.

• Вызов таймера fbOpenTimer(IN := TRUE, PT := tStrokeTime); передает параметры в блок таймера.

• Чтение выхода fbOpenTimer.Q и elapsed time fbOpenTimer.ET позволяет отслеживать прогресс движения.

• Вложенная композиция блоков создает иерархическую модель оборудования.

Пример 40. Сквозной проект: сортировочная станция

Пример интегрирует функции и функциональные блоки в единую систему управления автоматической сортировочной станцией.

Объявление пользовательских типов:

```
// Тип изделия
TYPE E_ProductType :
(
  eType_None := 0,
  eType_A := 1,
  eType_B := 2,
  eType_Reject := 3
) INT;
```

```
END_TYPE;

// Тип станции
TYPE ST_StationData :
STRUCT
   xSensorPresent    : BOOL;
   eProductType      : E_ProductType;
   uiCountA          : UINT;
   uiCountB          : UINT;
   uiCountReject     : UINT;
END_STRUCT;
END_TYPE;
```

Текст функции классификации:

```
FUNCTION FUN_ClassifyProduct : E_ProductType
VAR_INPUT
   rWeight           : REAL;
   rLength           : REAL;
END_VAR

// Классификация по весу и длине
IF rWeight < 0.0 OR rLength < 0.0 THEN
   FUN_ClassifyProduct := E_ProductType.eType_Reject;
ELSIF rWeight < 50.0 AND rLength < 100.0 THEN
   FUN_ClassifyProduct := E_ProductType.eType_A;
ELSIF rWeight >= 50.0 AND rLength >= 100.0 THEN
   FUN_ClassifyProduct := E_ProductType.eType_B;
ELSE
```

```
  FUN_ClassifyProduct := E_ProductType.eType_Reject;
END_IF;
```

Текст функционального блока счетчика:

```
FUNCTION_BLOCK FB_ProductCounter
VAR_INPUT
  xProductDetected  : BOOL;
  eType             : E_ProductType;
END_VAR
VAR_OUTPUT
  uiTotalCount      : UINT;
END_VAR
VAR
  xPrevDetect       : BOOL := FALSE;
END_VAR

IF xProductDetected AND NOT xPrevDetect THEN
  CASE eType OF
    E_ProductType.eType_A:
      uiCountA := uiCountA + 1;
    E_ProductType.eType_B:
      uiCountB := uiCountB + 1;
    E_ProductType.eType_Reject:
      uiCountReject := uiCountReject + 1;
  END_CASE;
END_IF;

xPrevDetect := xProductDetected;
```

```
uiTotalCount := uiCountA + uiCountB + uiCountReject;
```

Секция объявления переменных программы:

```
PROGRAM PLC_PRG
VAR
  // Станция сортировки
  stStation         : ST_StationData;

  // Счетчик изделий
  fbCounter         : FB_ProductCounter;

  // Аналоговые датчики
  rScaleWeight      : REAL := 45.5;
  rOpticalLength    : REAL := 95.0;

  // Выходы отбраковки
  xRejectActuator   : BOOL := FALSE;
  xDiverterA        : BOOL := FALSE;
  xDiverterB        : BOOL := FALSE;
END_VAR
```

Исполняемый код программы:

```
// Классификация изделия
stStation.eProductType := FUN_ClassifyProduct(
  rWeight := rScaleWeight,
  rLength := rOpticalLength
);
```

```
// Подсчет изделий
fbCounter(
   xProductDetected := stStation.xSensorPresent,
   eType := stStation.eProductType
);

// Управление отбраковкой
CASE stStation.eProductType OF
   E_ProductType.eType_A:
      xDiverterA := TRUE;
      xDiverterB := FALSE;
      xRejectActuator := FALSE;
   E_ProductType.eType_B:
      xDiverterA := FALSE;
      xDiverterB := TRUE;
      xRejectActuator := FALSE;
   E_ProductType.eType_Reject:
      xDiverterA := FALSE;
      xDiverterB := FALSE;
      xRejectActuator := TRUE;
   ELSE
      xDiverterA := FALSE;
      xDiverterB := FALSE;
      xRejectActuator := FALSE;
END_CASE;
```

Построчный разбор логики:

• Функция FUN_ClassifyProduct выполняет детерминированную классификацию по двум параметрам.

• Функциональный блок FB_ProductCounter хранит накопленные счетчики между вызовами.

• Оператор CASE управляет исполнительными механизмами в зависимости от типа изделия.

• Композиция функции и блока создает модульную архитектуру станции.

КОНТРОЛЬНЫЕ ВОПРОСЫ И УПРАЖНЕНИЯ К ГЛАВЕ 6

1. Объясните фундаментальное различие между функцией и функциональным блоком с точки зрения сохранения состояния.

2. Почему функции не могут иметь переменных в секции VAR со статическим хранением?

3. В чем разница между передачей параметров по значению (VAR_INPUT) и по ссылке (VAR_IN_OUT)?

4. Сколько байт памяти займет экземпляр функционального блока с: 2 BOOL, 1 REAL, 1 INT, 1 TON?

5. Зачем нужно присваивать значение имени функции перед завершением ее работы?

6. Объясните жизненный цикл экземпляра функционального блока от объявления до выполнения.

7. Почему стандартные таймеры TON и счетчики CTU реализованы как функциональные блоки, а не функции?

8. Можно ли вызвать функцию внутри функционального блока? Можно ли вызвать функциональный блок внутри функции?

9. Что произойдет, если не присвоить значение выходной переменной VAR_OUTPUT функционального блока?

10. В каких случаях целесообразно использовать VAR_IN_OUT вместо комбинации VAR_INPUT и VAR_OUTPUT?

Практические упражнения к главе 6

Упражнение 6.1. Разработайте функцию FUN_Deadband с гистерезисом, которая принимает входное значение, уставку и ширину зоны нечувствительности, возвращая отфильтрованный сигнал без дребезга.

Упражнение 6.2. Создайте функциональный блок FB_RampGenerator, формирующий плавный разгон и торможение сигнала от текущего значения к заданному с настраиваемой скоростью нарастания.

Упражнение 6.3. Разработайте функциональный блок FB_ThreePositionController для управления трехпозиционным исполнительным механизмом (открыть/закрыть/стоп) с зоной нечувствительности и ограничением числа переключений.

ПРИЛОЖЕНИЯ

Приложение А. Сводная таблица стандартных операторов языка Structured Text

Настоящая таблица систематизирует все стандартные операторы языка Structured Text стандарта IEC 61131-3 с указанием типов операндов, синтаксиса и ограничений применения в среде CODESYS v3.5.

Категория	Оператор	Синтаксис	Типы операндов	Возвращаемый тип	Примечания
Арифметические	Сложение	`A + B`	INT, UINT, DINT, UDINT, REAL, LREAL, TIME	Тот же, что у операндов	При смешанных типах результат приводится к более широкому типу
Арифметические	Вычитание	`A - B`	INT, UINT, DINT, UDINT, REAL, LREAL, TIME	Тот же, что у операндов	Результат может быть отрицательным для знаковых типов
Арифметические	Умножение	`A * B`	INT, UINT, DINT, UDINT, REAL, LREAL	Тот же, что у операндов	Риск переполнения для целочисленных типов
Арифметические	Деление	`A / B`	INT, UINT, DINT, UDINT, REAL, LREAL	Тот же, что у операндов	Целочисленное деление отбрасывает дробную часть

Ариф-мети-ческие	Остаток	A MOD B	INT, UINT, DINT, UDINT	Тот же, что у операн-дов	Неприменим к веществен-ным типам
Логи-ческие	И	A AND B	BOOL, BYTE, WORD, DWORD	Тот же, что у операн-дов	Побитовое И для битовых строк
Логи-ческие	ИЛИ	A OR B	BOOL, BYTE, WORD, DWORD	Тот же, что у операн-дов	Побитовое ИЛИ для би-товых строк
Логи-ческие	Исключаю-щее ИЛИ	A XOR B	BOOL, BYTE, WORD, DWORD	Тот же, что у операн-дов	Побитовое XOR для би-товых строк
Логи-ческие	НЕ	NOT A	BOOL, BYTE, WORD, DWORD	Тот же, что у опе-ранда	Побитовая инверсия для битовых строк
Отно-шения	Равно	A = B	Любые совмести-мые типы	BOOL	Неприме-нимо к ARRAY и STRUCT
Отно-шения	Не равно	A <> B	Любые совмести-мые типы	BOOL	Неприме-нимо к ARRAY и STRUCT
Отно-шения	Меньше	A < B	Числовые, TIME, DATE, STRING	BOOL	Неприме-нимо к BOOL и битовым строкам

Отношения	Больше	`A > B`	Числовые, TIME, DATE, STRING	BOOL	Неприменимо к BOOL и битовым строкам
Отношения	Меньше или равно	`A <= B`	Числовые, TIME, DATE, STRING	BOOL	Неприменимо к BOOL и битовым строкам
Отношения	Больше или равно	`A >= B`	Числовые, TIME, DATE, STRING	BOOL	Неприменимо к BOOL и битовым строкам
Сдвиги	Сдвиг влево	`SHL(A, N)`	BYTE, WORD, DWORD, INT, UINT, DINT, UDINT	Тот же, что у A	N — количество позиций сдвига
Сдвиги	Сдвиг вправо	`SHR(A, N)`	BYTE, WORD, DWORD, INT, UINT, DINT, UDINT	Тот же, что у A	N — количество позиций сдвига
Сдвиги	Циклический сдвиг влево	`ROL(A, N)`	BYTE, WORD, DWORD, INT, UINT, DINT, UDINT	Тот же, что у A	Старшие биты заносятся в младшие
Сдвиги	Циклический сдвиг вправо	`ROR(A, N)`	BYTE, WORD, DWORD, INT, UINT, DINT, UDINT	Тот же, что у A	Младшие биты заносятся в старшие

Выбор	Минимум	MIN(A, B)	Числовые, TIME	Тот же, что у операн-дов	Возвращает меньшее значение
Выбор	Максимум	MAX(A, B)	Числовые, TIME	Тот же, что у операн-дов	Возвращает большее значение
Выбор	Ограниче-ние	LIMIT(MN, V, MX)	Числовые, TIME	Тот же, что у операн-дов	Возвращает V, приведен-ный к диапа-зону [MN..MX]
Выбор	Селектор	SEL(G, IN0, IN1)	G: BOOL; IN0, IN1: любые совмести-мые	Тот же, что у IN0/IN1	Возвращает IN0 при G=FALSE, IN1 при G=TRUE
Выбор	Мульти-плексор	MUX(K, IN0, IN1, ...)	K: INT; IN: любые совмести-мые	Тот же, что у IN	Возвращает IN с номером K
Преоб-разова-ния	TO_BOOL	TO_BOOL(A)	Числовые, TIME	BOOL	0 = FALSE, не-0 = TRUE
Преоб-разова-ния	TO_INT	TO_INT(A)	Числовые, BOOL	INT	Округление REAL до INT
Преоб-разова-ния	TO_REAL	TO_REAL(A)	Числовые, BOOL	REAL	Преобразо-вание в ве-щественный тип

Преобразования	TO_STRING	TO_STRING(A)	Числовые, BOOL, TIME	STRING	Форматирование в строку
Преобразования	TO_TIME	TO_TIME(A)	UDINT, TIME	TIME	Интерпретация миллисекунд как TIME
Присваивание	Присваивание	A := B	Любые совместимые типы	—	Копирование значения из B в A

Приложение Б. Таблица приоритетов выполнения операций

При вычислении сложных выражений компилятор CODESYS v3.5 следует строгой иерархии приоритетов операторов. Операторы с более высоким приоритетом (меньший номер в таблице) выполняются раньше операторов с низким приоритетом. При одинаковом приоритете вычисление производится слева направо.

Приоритет	Категория	Операторы	Ассоциативность
1	Скобки	( )	Вычисление выражений в скобках в первую очередь
2	Унарные операции	- (минус), NOT, ADR	Справа налево
3	Мультипликативные	*, /, MOD	Слева направо
4	Аддитивные	+, - (плюс, минус бинарные)	Слева направо
5	Отношения	<, >, <=, >=	Слева направо

6	Равенство	=, <>	Слева направо
7	Логическое И	AND	Слева направо
8	Исключающее ИЛИ	XOR	Слева направо
9	Логическое ИЛИ	OR	Слева направо
10	Присваивание	:=	Справа налево

Пример вычисления по приоритетам:

Выражение: `rY := 5.0 + 3.0 * 2.0 - 1.0 / 4.0;`

Порядок вычисления:

1. `3.0 * 2.0 = 6.0` (приоритет 3, умножение)
2. `1.0 / 4.0 = 0.25` (приоритет 3, деление)
3. `5.0 + 6.0 = 11.0` (приоритет 4, сложение)
4. `11.0 - 0.25 = 10.75` (приоритет 4, вычитание)
5. `rY := 10.75;` (приоритет 10, присваивание)

Итоговый результат: `rY = 10.75`

Пример с круглыми скобками:

Выражение: `rY := (5.0 + 3.0) * (2.0 - 1.0) / 4.0;`

Порядок вычисления:

1. `(5.0 + 3.0) = 8.0` (приоритет 1, скобки)
2. `(2.0 - 1.0) = 1.0` (приоритет 1, скобки)

3. 8.0 * 1.0 = 8.0 (приоритет 3, умножение)

4. 8.0 / 4.0 = 2.0 (приоритет 3, деление)

5. rY := 2.0; (приоритет 10, присваивание)

Итоговый результат: rY = 2.0

Приложение В. Таблица префиксов венгерской нотации

Настоящая таблица фиксирует стандартные префиксы венгерской нотации, применяемые в промышленном программировании на языке Structured Text для повышения читаемости и самодокументированности кода.

Префикс	Тип данных	Пример имени	Технологический смысл
x	BOOL	xEmergencyStop	Дискретный сигнал, флаг, бит состояния
b	BYTE	bControlByte	Однобайтовая битовая строка, управляющий байт
w	WORD	wStatusWord	Двухбайтовое слово статуса, регистр
dw	DWORD	dwErrorCode	Четырехбайтовая маска ошибок, код события
i	INT	iStepIndex	16-битное знаковое целое, шаг, индекс
ui	UINT	uiBatchCount	16-битное беззнаковое, количество, счетчик
di	DINT	diPositionMm	32-битное знаковое, координата, позиция
udi	UDINT	udiTotalCycles	32-битное беззнаковое, наработка, циклы

r	REAL	rPressureBar	Вещественное одинарной точности, аналоговый параметр
lr	LREAL	lrCoordinateMm	Вещественное двойной точности, высокоточные координаты
t	TIME	tDelayOn	Интервал времени, уставка таймера
lt	LTIME	ltPulseWidth	Длинный интервал времени (наносекунды)
d	DATE	dStartDate	Календарная дата
tod	TOD	todShiftStart	Время суток
dt	DT	dtEventTime	Дата и время события
s	STRING	sRecipeName	Текстовая строка ASCII
ws	WSTRING	wsMessage	Текстовая строка Unicode
a	ARRAY	aiTemperatures	Массив данных (префикс уточняет тип элементов)
st	STRUCT	stMotorData	Структура пользовательского типа
e	ENUM	eMachineState	Элемент перечислимого типа, состояние
fb	FUNCTION_BLOCK	fbPumpTimer	Экземпляр функционального блока
p	POINTER	pDataPointer	Указатель на область памяти

Приложение Г. Эталонные решения к расчетно-практическим упражнениям

Решение упражнения 1.1 (Глава 1)

Исправленный код программы управления освещением:

```
PROGRAM PLC_PRG
VAR
  xSensor1       : BOOL;
  xPumpCommand    : BOOL;  // Исправлено: убрана цифра из начала имени
  rTargetPressure : REAL := 5.0;  // Добавлена точка с запятой
END_VAR

// Исправлено: убрана точка с запятой после TRUE, добавлен THEN
IF xSensor1 = TRUE THEN
  xPumpCommand := TRUE;  // Добавлена точка с запятой, исправлен оператор на :=
ELSE
  xPumpCommand := FALSE;
END_IF;  // Добавлено закрытие END_IF
```

Исправленные ошибки:

1. Отсутствующая точка с запятой после объявления `xSensor1 : BOOL`

2. Идентификатор `1_PumpCommand` не может начинаться с цифры — переименовано в `xPumpCommand`

3. Отсутствующая точка с запятой после инициализации `rTargetPressure`

4. Точка с запятой после `TRUE` в условии `IF` — синтаксическая ошибка

5. Оператор присваивания = вместо := в теле IF

6. Отсутствующая точка с запятой после TRUE в присваивании

7. Отсутствующее ключевое слово END_IF

Решение упражнения 1.2 (Глава 1)

Схема управления конвейером с двух постов:

```
PROGRAM PLC_PRG
VAR
  // Кнопки Пуск с двух постов (нормально разомкнутые)
  xBtnStart1      : BOOL := FALSE;
  xBtnStart2      : BOOL := FALSE;

  // Кнопки Стоп с двух постов (нормально замкнутые)
  xBtnStop1       : BOOL := TRUE;
  xBtnStop2       : BOOL := TRUE;

  // Внутреннее состояние самоподхвата
  xConveyorRunning  : BOOL := FALSE;

  // Выход на двигатель конвейера
  xConveyorMotor    : BOOL := FALSE;
END_VAR

// Логика самоподхвата с двух постов:
// Пуск с любого поста ИЛИ уже работающий конвейер
// И не нажат Стоп на любом посту
IF (xBtnStart1 OR xBtnStart2 OR xConveyorRunning) AND
  xBtnStop1 AND xBtnStop2 THEN
```

```
  xConveyorRunning := TRUE;
ELSE
  xConveyorRunning := FALSE;
END_IF;

xConveyorMotor := xConveyorRunning;
```

Решение упражнения 1.3 (Глава 1)

Выражение для ровно двух замкнутых датчиков из трех:

```
// Результат истинен, когда замкнуты РОВНО ДВА датчика из трех
xResult := (xSensorA AND xSensorB AND NOT xSensorC) OR
      (xSensorA AND NOT xSensorB AND xSensorC) OR
      (NOT xSensorA AND xSensorB AND xSensorC);
```

Объяснение: выражение представляет собой дизъюнкцию трех конъюнкций, каждая из которых соответствует одной из трех возможных комбинаций ровно двух замкнутых датчиков.

Решение упражнения 2.1 (Глава 2)

Расчет памяти с учетом выравнивания:

```
VAR
  xInterlock1    : BOOL;   // 1 байт (выравнивание до байта)
  wStatusRegister : WORD;   // 2 байта
  rPressureSensor : REAL;   // 4 байта
  bErrorCode     : BYTE;   // 1 байт
  udiTotalCycles  : UDINT;  // 4 байта
```

```
END_VAR
```

Суммарный объем: 1 + 2 + 4 + 1 + 4 = **12 байт**

Примечание: в некоторых реализациях CODESYS переменная `BOOL` может занимать 1 байт, но при размещении в структуре возможно выравнивание до 2 или 4 байт в зависимости от настроек компилятора.

Решение упражнения 2.2 (Глава 2)

Извлечение старшего и младшего байта из WORD:

```
PROGRAM PLC_PRG
VAR
  wInputWord    : WORD := 16#1234;
  bHighByte     : BYTE;
  bLowByte      : BYTE;
END_VAR

// Младший байт (биты 0..7)
bLowByte := WORD_TO_BYTE(wInputWord AND 16#00FF);

// Старший байт (биты 8..15) со сдвигом вправо на 8 бит
bHighByte := WORD_TO_BYTE(SHR(wInputWord, 8));
```

Альтернативный вариант через побитовую адресацию:

```
bLowByte := 16#00;
bLowByte.0 := wInputWord.0;
bLowByte.1 := wInputWord.1;
```

```
// ... и так далее для всех 8 бит (неэффективно)
```

Решение упражнения 2.3 (Глава 2)

Функция преобразования секунд в TIME:

```
FUNCTION FUN_SecondsToTime : TIME
VAR_INPUT
  udiSeconds   : UDINT;
END_VAR

// Преобразование секунд в миллисекунды и затем в TIME
FUN_SecondsToTime := UDINT_TO_TIME(udiSeconds * 1000);
```

Вызов функции:

```
tDelay := FUN_SecondsToTime(udiSeconds := 120);   // Результат: T#2m0s
```

Решение упражнения 3.1 (Глава 3)

Выражение для ровно двух замкнутых датчиков (альтернативное решение через арифметику):

```
// Подсчет количества замкнутых датчиков через преобразование BOOL в INT
xResult := (BOOL_TO_INT(xSensorA) + BOOL_TO_INT(xSensorB) + BOOL_TO_INT(xSensorC)) = 2;
```

Это решение более компактно и читаемо по сравнению с дизъюнкцией трех конъюнкций.

Решение упражнения 3.2 (Глава 3)

Усреднение среднего по величине числа:

```
PROGRAM PLC_PRG
VAR
   rValue1       : REAL := 10.0;
   rValue2       : REAL := 20.0;
   rValue3       : REAL := 30.0;
   rMiddleValue    : REAL;
END_VAR

// Находим минимум и максимум
// Среднее = (сумма - мин - макс) = среднее по величине число
rMiddleValue := (rValue1 + rValue2 + rValue3) -
            MIN(rValue1, MIN(rValue2, rValue3)) -
            MAX(rValue1, MAX(rValue2, rValue3));
```

Для значений 10, 20, 30 результат: `rMiddleValue = 20.0`

Решение упражнения 3.3 (Глава 3)

Циклический сдвиг байта вправо:

```
PROGRAM PLC_PRG
VAR
   bShiftRegister  : BYTE := 16#01;
   xShiftClock     : BOOL := FALSE;
   xPrevClock      : BOOL := FALSE;
END_VAR
```

```
// Детектор переднего фронта тактового сигнала
IF xShiftClock AND NOT xPrevClock THEN
   // Проверка: если младший бит равен 1, перед сдвигом устанавливаем
старший бит
   IF bShiftRegister.0 THEN
      bShiftRegister := bShiftRegister OR 16#80;  // Устанавливаем бит 7
   END_IF;

   // Логический сдвиг вправо на 1 бит
   bShiftRegister := SHR(bShiftRegister, 1);
END_IF;

xPrevClock := xShiftClock;
```

Альтернативное решение через циклический сдвиг ROR:

```
IF xShiftClock AND NOT xPrevClock THEN
   bShiftRegister := ROR(bShiftRegister, 1);
END_IF;
```

Решение упражнения 4.1 (Глава 4)

Оценка по буквенной шкале через CASE:

```
PROGRAM PLC_PRG
VAR
   iGrade        : INT := 85;
   sLetterGrade    : STRING(1);
END_VAR
```

```
CASE iGrade OF
  90..100:
    sLetterGrade := 'A';
  80..89:
    sLetterGrade := 'B';
  70..79:
    sLetterGrade := 'C';
  60..69:
    sLetterGrade := 'D';
  0..59:
    sLetterGrade := 'F';
  ELSE
    // Некорректное значение (<0 или >100)
    sLetterGrade := 'X';  // Ошибка
END_CASE;
```

Решение упражнения 4.2 (Глава 4)

Поиск второго по величине элемента без сортировки:

```
PROGRAM PLC_PRG
VAR
  aiNumbers      : ARRAY[0..9] OF INT := [45, 12, 78, 23, 56, 89, 34, 67, 91, 15];
  iLargest       : INT;
  iSecondLargest  : INT;
  iIndex         : INT;
END_VAR
```

```
// Инициализация первыми двумя элементами
IF aiNumbers[0] > aiNumbers[1] THEN
   iLargest := aiNumbers[0];
   iSecondLargest := aiNumbers[1];
ELSE
   iLargest := aiNumbers[1];
   iSecondLargest := aiNumbers[0];
END_IF;

// Перебор остальных элементов
FOR iIndex := 2 TO 9 DO
   IF aiNumbers[iIndex] > iLargest THEN
      // Новый максимум: старый максимум становится вторым
      iSecondLargest := iLargest;
      iLargest := aiNumbers[iIndex];
   ELSIF aiNumbers[iIndex] > iSecondLargest AND aiNumbers[iIndex] <> iLargest
THEN
      // Новый второй максимум
      iSecondLargest := aiNumbers[iIndex];
   END_IF;
END_FOR;
```

Решение упражнения 4.3 (Глава 4)

НОД методом Евклида с защитой:

```
FUNCTION FUN_GCD : INT
VAR_INPUT
   iA       : INT;
```

```
  iB        : INT;
END_VAR
VAR
  iTemp     : INT;
  uiCounter  : UINT := 0;
  uiMaxIter  : UINT := 1000;
END_VAR

// Работа с абсолютными значениями
IF iA < 0 THEN iA := -iA; END_IF;
IF iB < 0 THEN iB := -iB; END_IF;

// Алгоритм Евклида
WHILE iB <> 0 DO
  uiCounter := uiCounter + 1;
  IF uiCounter > uiMaxIter THEN
    FUN_GCD := 0;  // Ошибка: превышение итераций
    RETURN;
  END_IF;

  iTemp := iB;
  iB := iA MOD iB;
  iA := iTemp;
END_WHILE;

FUN_GCD := iA;
```

Решение упражнения 5.1 (Глава 5)

Структура резервуара и массив из 4 емкостей:

```
TYPE ST_TankData :
STRUCT
  rLevelPercent    : REAL;   // Уровень, %
  rTemperatureC    : REAL;   // Температура, град C
  rPressureBar     : REAL;   // Давление, бар
  xAlarm           : BOOL;   // Флаг аварии
  xReady           : BOOL;   // Готовность
  xMixerRunning    : BOOL;   // Перемешивание включено
  udiMixerHours    : UDINT;  // Моточасы мешалки
  sTankName        : STRING(20); // Имя резервуара
END_STRUCT;
END_TYPE;

PROGRAM PLC_PRG
VAR
  // Массив из 4 резервуаров
  arTanks          : ARRAY[1..4] OF ST_TankData;
END_VAR

// Инициализация имен
arTanks[1].sTankName := 'TK-101';
arTanks[2].sTankName := 'TK-102';
arTanks[3].sTankName := 'TK-103';
arTanks[4].sTankName := 'TK-104';
```

Решение упражнения 5.2 (Глава 5)

Перечисление приоритетов и функция цвета:

```
TYPE E_AlarmPriority :
(
   eLow := 0,
   eMedium := 1,
   eHigh := 2,
   eCritical := 3
) INT;
END_TYPE;

FUNCTION FUN_GetAlarmColor : STRING
VAR_INPUT
   ePriority     : E_AlarmPriority;
END_VAR

CASE ePriority OF
   E_AlarmPriority.eLow:
      FUN_GetAlarmColor := 'Green';
   E_AlarmPriority.eMedium:
      FUN_GetAlarmColor := 'Yellow';
   E_AlarmPriority.eHigh:
      FUN_GetAlarmColor := 'Orange';
   E_AlarmPriority.eCritical:
      FUN_GetAlarmColor := 'Red';
   ELSE
      FUN_GetAlarmColor := 'Gray';
```

```
END_CASE;
```

Решение упражнения 5.3 (Глава 5)

Объединение для аналогового канала:

```
TYPE UT_AnalogChannel :
UNION
   iValue          : INT;
   uiValue         : UINT;
   arBytes         : ARRAY[0..1] OF BYTE;
   axBits          : ARRAY[0..15] OF BOOL;
END_UNION;
END_TYPE;

PROGRAM PLC_PRG
VAR
   utChannel       : UT_AnalogChannel;
   iReadInt        : INT;
   uiReadUint      : UINT;
   bReadByte0      : BYTE;
   xReadBit0       : BOOL;
END_VAR

// Запись значения
utChannel.iValue := -1234;

// Чтение через разные представления
iReadInt := utChannel.iValue;      // -1234
```

```
uiReadUint := utChannel.uiValue;    // 64302 (дополнительный код)
bReadByte0 := utChannel.arBytes[0]; // Младший байт
xReadBit0 := utChannel.axBits[0];   // Младший бит
```

Решение упражнения 6.1 (Глава 6)

Функция с гистерезисом (мертвая зона):

```
FUNCTION FUN_Deadband : REAL
VAR_INPUT
   rInput         : REAL;
   rSetpoint      : REAL;
   rDeadbandWidth  : REAL;
END_VAR
VAR
   rUpperLimit     : REAL;
   rLowerLimit     : REAL;
END_VAR

rUpperLimit := rSetpoint + rDeadbandWidth / 2.0;
rLowerLimit := rSetpoint - rDeadbandWidth / 2.0;

IF rInput > rUpperLimit THEN
   FUN_Deadband := rInput;
ELSIF rInput < rLowerLimit THEN
   FUN_Deadband := rInput;
ELSE
   // Вход в мертвой зоне — возвращаем уставку
   FUN_Deadband := rSetpoint;
```

```
END_IF;
```

Решение упражнения 6.2 (Глава 6)

Функциональный блок генератора рампы:

```
FUNCTION_BLOCK FB_RampGenerator
VAR_INPUT
   rTarget       : REAL;
   rSlewRate     : REAL;  // Единиц в секунду
   xReset        : BOOL;
END_VAR
VAR_OUTPUT
   rOutput       : REAL;
END_VAR
VAR
   rPrevOutput    : REAL := 0.0;
   xPrevReset     : BOOL := FALSE;
   rMaxDelta      : REAL;
END_VAR

// Детектор сброса
IF xReset AND NOT xPrevReset THEN
   rOutput := 0.0;
   rPrevOutput := 0.0;
END_IF;

xPrevReset := xReset;
```

```
// Максимальное изменение за цикл (предполагаем цикл 10 мс = 0.01 с)
rMaxDelta := rSlewRate * 0.01;

IF rTarget > rPrevOutput + rMaxDelta THEN
   rOutput := rPrevOutput + rMaxDelta;
ELSIF rTarget < rPrevOutput - rMaxDelta THEN
   rOutput := rPrevOutput - rMaxDelta;
ELSE
   rOutput := rTarget;
END_IF;

rPrevOutput := rOutput;
```

Решение упражнения 6.3 (Глава 6)

Трехпозиционный контроллер:

```
FUNCTION_BLOCK FB_ThreePositionController
VAR_INPUT
   rError        : REAL;  // Рассогласование
   rDeadband     : REAL;  // Зона нечувствительности
   xEnable       : BOOL;
END_VAR
VAR_OUTPUT
   xOpenCmd      : BOOL;
   xCloseCmd     : BOOL;
END_VAR
VAR
   xPrevOpen     : BOOL := FALSE;
```

```
  xPrevClose     : BOOL := FALSE;
  tMinOffTime    : TIME := T#2s;  // Минимальное время выключения
  fbOffTimer     : TON;
END_VAR

IF NOT xEnable THEN
  xOpenCmd := FALSE;
  xCloseCmd := FALSE;
ELSE
  // Зона нечувствительности
  IF ABS(rError) < rDeadband THEN
    xOpenCmd := FALSE;
    xCloseCmd := FALSE;
  ELSIF rError > 0 THEN
    // Требуется открытие
    fbOffTimer(IN := FALSE, PT := tMinOffTime);
    xOpenCmd := TRUE;
    xCloseCmd := FALSE;
  ELSE
    // Требуется закрытие
    fbOffTimer(IN := FALSE, PT := tMinOffTime);
    xOpenCmd := FALSE;
    xCloseCmd := TRUE;
  END_IF;

  // Ограничение числа переключений через таймер
  IF NOT xPrevOpen AND NOT xPrevClose THEN
    fbOffTimer(IN := TRUE, PT := tMinOffTime);
  END_IF;
```

```
  IF NOT fbOffTimer.Q THEN
    xOpenCmd := FALSE;
    xCloseCmd := FALSE;
  END_IF;
END_IF;

xPrevOpen := xOpenCmd;
xPrevClose := xCloseCmd;
```

ЗАКЛЮЧИТЕЛЬНЫЕ РЕКОМЕНДАЦИИ

Настоящее учебное пособие заложило фундаментальные основы программирования на языке Structured Text в среде CODESYS v3.5. Для дальнейшего профессионального развития инженеру рекомендуется:

1. **Изучить объектно-ориентированные расширения IEC 61131-3 Ed. 3:** интерфейсы, наследование функциональных блоков, полиморфизм, обобщенные типы (generics).

2. **Освоить стандартные библиотеки CODESYS:** Standard.lib, Util.lib, Oscat.lib — сотни готовых функций и блоков для работы со строками, массивами, сетевыми протоколами и математикой.

3. **Практиковаться в отладке:** глубокое понимание режимов Online Monitoring, Force Values, Breakpoints, Watch Lists и Trace-функционала CODESYS.

4. **Изучить сетевые протоколы:** Modbus TCP/RTU, OPC UA, EtherCAT, Profinet — интеграция ПЛК в распределенные системы автоматизации.

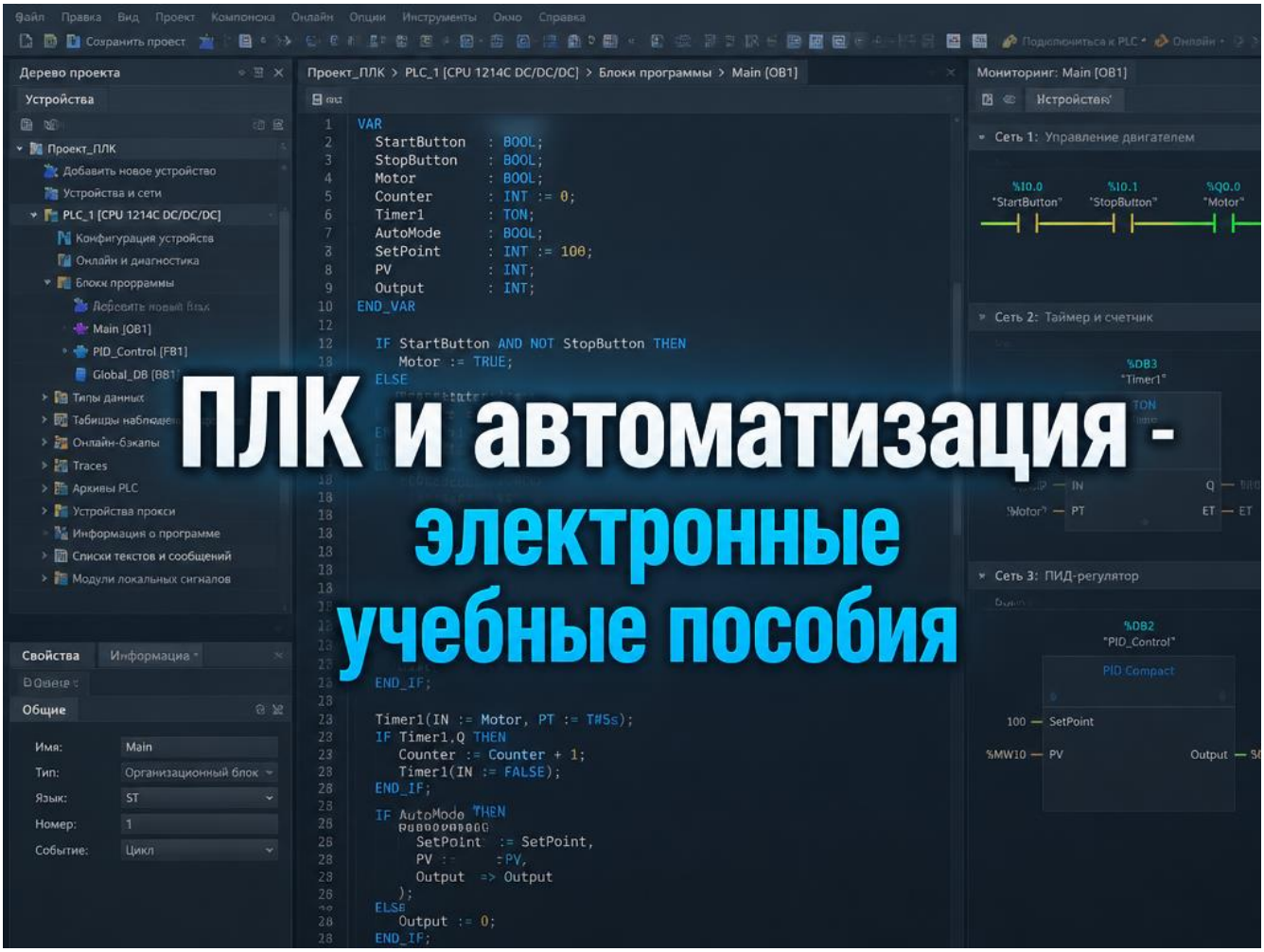
5. **Освоить систему контроля версий Git:** ведение истории проекта, ветвление, слияние кода, ревью изменений.

6. **Изучить основы кибербезопасности ПЛК:** защита проектов паролем, отключение незадействованных сервисов, сегментация сетей, аудит изменений.

7. **Практиковать код-ревью:** чтение и анализ чужого кода, выявление антипаттернов, рефакторинг legacy-проектов.

8. **Следить за обновлениями стандарта:** IEC 61131-3 регулярно развивается, добавляя новые возможности и лучшие практики.

Успешное освоение Structured Text открывает инженеру путь к проектированию сложных распределенных систем автоматизации, интеграции с SCADA и MES-системами, разработке универсальных библиотек функциональных блоков и созданию масштабируемых архитектур промышленного интернета вещей (IIoT).



Практические электронные учебные пособия по программированию ПЛК, Structured Text и промышленной автоматизации.

Электронные учебные пособия

Здесь собраны материалы для тех, кто хочет не просто познакомиться с теорией, а научиться разрабатывать, отлаживать и сопровождать реальные алгоритмы управления. Пособия построены по принципу «от простого к сложному»: сначала базовый синтаксис и типы данных, затем функции, функциональные блоки, диагностика, оборудование и законченные технологические проекты.

С чего начать

Если вы только начинаете изучать Structured Text, оптимальная последовательность выглядит так:

1. **Базовый курс по языку ST для ПЛК** — первое знакомство с синтаксисом, переменными, условиями, циклами, таймерами и простыми проектами.
2. **Structured Text с нуля** — системное изучение языка в CODESYS V3.5: типы данных, операторы, массивы, структуры, функции и функциональные блоки.
3. **100 примеров программирования ПЛК на языке Structured Text** — переход от отдельных конструкций к готовым алгоритмам автоматизации.
4. **Практические пособия по отдельным задачам** — двигатели, аналоговые сигналы, оборудование, диагностика и технологические объекты.
5. **Комплекты учебных пособий** — последовательное профессиональное обучение по сниженной цене.

Учебные пособия

Базовый курс по языку ST для ПЛК

Короткий практический курс для первого знакомства с Structured Text. В пособии разобраны синтаксис ST по IEC 61131-3, типы данных, условия, циклы, функции, таймеры, счётчики, конечные автоматы, работа с входами и выходами ПЛК.

В конце читатель самостоятельно реализует несколько простых проектов:

- управление светофором;

- конвейерную линию;
- резервуар с контролем уровня;
- программы PROGRAM с подробными комментариями.

Подойдёт тем, кто хочет быстро получить базовые навыки и начать писать собственные программы.

[Купить пособие]

Практическое руководство по ошибкам в программировании циклов и условной логики в ПЛК

Ошибки в условиях и циклах могут привести к зависанию задачи, неправильному переключению режимов и аварийной остановке оборудования.

В пособии разобраны:

- бесконечные циклы;
- неправильные условия выхода;
- приоритеты логических операторов;
- ошибки во вложенных IF;
- применение CASE;
- безопасная работа с FOR, WHILE и REPEAT;
- использование EXIT, CONTINUE и RETURN;
- анализ типичных ошибок в коде.

Практическое продолжение бесплатного пособия для тех, кто хочет научиться писать более надёжный код.

[Купить пособие]

100 примеров программирования ПЛК на языке Structured Text

Сборник из 100 практических примеров по IEC 61131-3.

Примеры расположены по принципу постепенного усложнения и охватывают:

- дискретную логику;
- кнопки и самоподхват;
- таймеры и счётчики;
- конечные автоматы;
- аналоговые сигналы;
- масштабирование;
- массивы и структуры;
- функциональные блоки;
- насосные станции;
- конвейеры;
- нагреватели;
- резервирование датчиков;
- аварийную диагностику;
- технологические линии.

Это практическое продолжение бесплатного пособия для тех, кто хочет не только читать код, но и регулярно решать инженерные задачи.

[Купить пособие]

Управление двигателями на ПЛК на языке Structured Text

Пособие посвящено управлению электродвигателями с помощью ПЛК. Рассматриваются не только команды «Пуск» и «Стоп», но и типовые защитные функции, обратная связь и режимы работы.

Внутри:

- схема управления двигателем с самоподхватом;
- кнопки «Пуск» и «Стоп»;
- тепловая защита;
- контроль включения контактора;
- ручной и автоматический режимы;
- блокировки;
- аварийное отключение;
- реверсивный пускатель;
- контроль готовности двигателя;
- программная диагностика отказов.

Подойдёт студентам, наладчикам и инженерам, которые хотят связать программирование ST с реальными электрическими схемами.

[Купить пособие]

Управление массивами и структурами данных в ST для ПЛК

Когда в проекте появляется несколько десятков датчиков, приводов, рецептов и аварий, набор отдельных переменных быстро превращается в неуправляемый код.

Пособие показывает, как организовать данные с помощью:

- одномерных и многомерных массивов;
- структур `STRUCT`;
- перечислений `ENUM`;
- пользовательских типов данных;
- массивов структур;
- рецептурных таблиц;
- кольцевых буферов;
- журналов событий;
- технологических паспортов механизмов.

Это следующий шаг после изучения базового синтаксиса: от отдельных переменных — к структурированной модели технологического объекта.

[Купить пособие]

Функциональные блоки в Structured Text

Функциональные блоки — основа модульного программирования ПЛК. Они позволяют один раз разработать алгоритм, а затем использовать его для нескольких одинаковых механизмов.

В пособии рассматриваются:

- различия между `FUNCTION`, `FUNCTION_BLOCK` и `PROGRAM`;
- входы, выходы и внутреннее состояние блока;
- экземпляры функциональных блоков;

- повторное использование алгоритмов;
- таймеры и счётчики внутри FB;
- блоки двигателей, клапанов и фильтров;
- цифровой фильтр скользящего среднего;
- генератор импульсов;
- контроль обратной связи;
- построение комплексного агрегата из нескольких блоков.

Материал поможет перейти от отдельных программ к профессиональной архитектуре ПЛК-проекта.

[Купить пособие]

Обработка строк и текстовых данных в Structured Text

Строки используются в аварийных сообщениях, рецептах, журналах событий, командах оператора и обмене данными с внешними системами.

В пособии разобраны:

- типы `STRING` и `WSTRING`;
- объявление строковых буферов;
- объединение строк;
- поиск и замена фрагментов;
- преобразование чисел в текст;
- разбор команд оператора;
- выделение параметров из строки;

- формирование диагностических сообщений;
- контроль длины буфера;
- типичные ошибки при работе со строками.

Подойдёт тем, кто хочет создавать полноценную диагностику и обмен данными с HMI, SCADA и внешними устройствами.

[Купить пособие]

Аналоговые выходы ЦАП в ПЛК

Аналоговый выход позволяет плавно управлять клапанами, приводами, нагревателями и другими исполнительными устройствами.

В пособии объясняется:

- чем аналоговый сигнал отличается от дискретного;
- как работают диапазоны 0–10 В и 4–20 мА;
- как преобразовать числовое значение ПЛК в инженерную величину;
- как выполнять масштабирование;
- как ограничивать диапазон сигнала;
- как реализовать ручной и автоматический режим;
- как применять ограничение скорости изменения задания;

[Купить пособие]

Руководство по конфигурации входов и выходов ПЛК

Ошибки конфигурации I/O связывают программную логику с неправильными физическими каналами. В результате двигатель может не запуститься,

клапан — открыться в неподходящий момент, а аналоговый сигнал — отображаться с неверным масштабом.

В пособии рассмотрены:

- нумерация физических каналов;
- различия между %I, %Q и %M;
- символическая адресация;
- настройка дискретных входов и выходов;
- нормально замкнутые и нормально разомкнутые цепи;
- сигналы аварийного останова;
- аналоговые входы;
- диапазоны 0–10 В и 4–20 мА;
- фильтрация;
- масштабирование;
- типовые ошибки I/O Mapping.

Незаменимый материал перед переходом к полноценным технологическим проектам.

[Купить пособие]

Архитектура ПЛК-проекта: сигналы, HMI и диагностика

Рабочий ПЛК-код — это не только набор правильно написанных операторов. В промышленном проекте важны структура, масштабируемость, диагностика и удобство сопровождения.

Пособие посвящено:

- разделению проекта на функциональные зоны;
- организации переменных;
- именованию сигналов;
- связи ПЛК с HMI;
- структуре команд и состояний;
- обработке аварий;
- диагностическим словам;
- режимам «Ручной», «Автомат» и «Наладка»;
- журналированию событий;
- подготовке к расширению проекта;
- защите от «магических» адресов и неструктурированного кода.

Для тех, кто уже знает базовый ST и хочет создавать проекты, которые удобно читать, отлаживать и передавать другим специалистам.

[Купить пособие]

Автоматизация электрических печей нагрева сопротивлением

Практическое пособие по автоматизации электрических печей, где качество продукции зависит от точности поддержания температуры.

Рассматриваются:

- структура системы управления печью;
- нагревательные элементы;
- температурные датчики;

- дискретное и аналоговое управление;
- двухпозиционное регулирование;
- гистерезис;
- ПИД-регулирование;
- аварийные ограничения;
- контроль перегрева;
- управление вентиляцией;
- этапы технологического цикла;
- аварийная сигнализация.

[Купить пособие]

Управление парогенератором на ПЛК

Пособие посвящено автоматизации парогенератора и связанных с ним насосов, клапанов, датчиков давления и температуры.

Материал включает:

- подготовку оборудования к запуску;
- контроль уровня воды;
- управление насосом;
- управление горелкой или нагревателем;
- регулирование давления;
- защиту от сухого хода;
- аварийное отключение;

- контроль датчиков;
- последовательность запуска и останова;
- формирование диагностических сообщений.

Рекомендуется специалистам по автоматизации котельного и теплотехнического оборудования.

[Купить пособие]

Управление микроклиматом промышленной теплицы на ST

Пример комплексной автоматизации объекта с большим количеством аналоговых и дискретных параметров.

В пособии разобраны алгоритмы управления:

- температурой;
- влажностью;
- вентиляцией;
- отоплением;
- освещением;
- концентрацией CO_2;
- поливом;
- насосами;
- клапанами;
- аварийными режимами;
- расписаниями и технологическими режимами.

Хороший пример того, как объединить датчики, исполнительные механизмы, режимы работы и диагностику в одном проекте.

[Купить пособие]

Программирование станции водоочистки на языке ST

Комплексный пример автоматизации станции водоочистки с насосами, клапанами, фильтрами и анализаторами.

Рассматриваются:

- структура технологического процесса;
- последовательность этапов очистки;
- управление насосными группами;
- промывка фильтров;
- коагуляция и флокуляция;
- озонирование;
- УФ-обработка;
- контроль уровня и давления;
- блокировки;
- аварийные режимы;
- диагностика оборудования.

Пособие показывает, как применять Structured Text для многоэтапного технологического объекта.

[Купить пособие]

Комплекты для системного обучения

Structured Text: от основ к промышленным проектам

Комплект из 7 учебных пособий и 3 бонусов в одном ZIP-архиве.

Это последовательная программа обучения, которая помогает пройти путь:

1. от синтаксиса и типов данных;
2. к условиям, циклам и функциям;
3. затем к массивам и структурам;
4. функциональным блокам;
5. аналоговым сигналам;
6. архитектуре проекта;
7. полноценным промышленным алгоритмам.

Для кого: студентов, начинающих инженеров, специалистов по АСУ ТП и практикующих программистов ПЛК.

[Купить комплект]

Практика на Structured Text

Комплект из 12 учебных пособий и бонуса.

Включает практические темы:

- управление двигателями;
- функциональные блоки;
- массивы;
- структуры;

- аналоговые сигналы;
- строки;
- диагностика;
- управление технологическими объектами;
- архитектура ПЛК-проектов;
- промышленные алгоритмы;
- работа с насосами, клапанами, нагревателями и приводами.

Для тех, кому недостаточно отдельных примеров и нужен большой объём практики на разных задачах.

[Купить комплект]

Библиотека программиста ПЛК

Комплект из 11 учебных пособий.

Это расширенная библиотека по методологии, архитектуре и теории промышленного программирования на Structured Text.

Комплект помогает решить типичные проблемы разработчика:

- монолитные программы;
- повторяющийся код;
- неструктурированные переменные;
- отсутствие диагностики;
- сложное сопровождение;
- ошибки в ПИД-регулировании;

- неправильное использование состояний;
- неудобная связь с HMI;
- отсутствие единых правил оформления.

Подойдёт тем, кто хочет перейти от отдельных рабочих фрагментов к инженерной системе разработки ПЛК-программ.

[Купить комплект]

Электронные учебные пособия по автоматизации производства

Комплекс лекций по основам автоматизации производства

Полный курс объёмом более 500 страниц.

Включает:

- основы автоматизации;
- датчики;
- исполнительные механизмы;
- усилители;
- ПЛК;
- логические элементы;
- системы контроля и регулирования;
- ПИД-регулирование;
- автоматизацию электроприводов;
- автоматизацию деревообрабатывающего производства;

Подойдёт студентам и специалистам, которым необходимо системно изучить не только программирование ПЛК, но и автоматизацию производства в целом.

[Купить курс]

Крупные издания

Изучаем Structured Text — МЭК 61131-3

Полная электронная версия книги по языку Structured Text. Автор – Сергей Романов

Пособие предназначено для глубокого изучения языка программирования ПЛК и может использоваться как подробный учебник и справочник.

В нём последовательно рассматриваются:

- синтаксис ST;
- типы данных;
- операторы;
- условия;
- циклы;
- функции;
- функциональные блоки;
- массивы;
- структуры;
- таймеры;

- счётчики;
- работа с вводом-выводом;

[Купить электронную книгу]

Рекомендуемый путь обучения

Уровень	Что изучить	Рекомендуемое издание
Начальный	Основы ST и первые программы	Бесплатное пособие
Базовый	Синтаксис, типы данных, условия и циклы	Базовый курс по ST
Практический	Готовые алгоритмы и инженерные задачи	100 примеров программирования ПЛК
Средний	Двигатели, массивы, аналоговые сигналы и FB	Практические тематические пособия
Продвинутый	Архитектура, диагностика и сопровождение	Архитектура ПЛК-проекта
Профессиональный	Комплексная система знаний	Комплекты и библиотека программиста ПЛК